



# Adaptive Server<sup>®</sup> Anywhere SQL User's Guide

Last modified: October 2002  
Part Number: 38124-01-0802-01

Copyright © 1989–2002 Sybase, Inc. Portions copyright © 2001–2002 iAnywhere Solutions, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of iAnywhere Solutions, Inc. iAnywhere Solutions, Inc. is a subsidiary of Sybase, Inc.

Sybase, SYBASE (logo), AccelaTrade, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, AnswerBase, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-FORMS, APT-Library, APT-Translator, ASEP, Backup Server, BayCam, Bit-Wise, BizTracker, Certified PowerBuilder Developer, Certified SYBASE Professional, Certified SYBASE Professional (logo), ClearConnect, Client Services, Client-Library, CodeBank, Column Design, ComponentPack, Connection Manager, Convoy/DM, Copernicus, CSP, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, Dynamo, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC-GATEWAY, ECMAP, ECRT, eFulfillment Accelerator, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, Financial Fusion, Financial Fusion Server, First Impression, Formula One, Gateway Manager, GeoPoint, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InstaHelp, Intellidex, InternetBuilder, iremote, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, MainframeConnect, Maintenance Express, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MethodSet, ML Query, MobiCATS, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASIS, OASIS (logo), ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Business Interchange, Open Client, Open Client/Server, Open Client/Server Interfaces, Open ClientConnect, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, Partnerships that Work, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power Through Knowledge, Power++, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, Powersoft Portfolio, Powersoft Professional, PowerStage, PowerStudio, PowerTips, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Rapport, Relational Beans, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Report Workbench, Report-Execute, Resource Manager, RW-DisplayLib, RW-Library, S Designer, S-Designer, S.W.I.F.T. Message Format Libraries, SAFE, SAFE/PRO, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL Server SNMP SubAgent, SQL Server/CFT, SQL Server/DBM, SQL SMART, SQL Station, SQL Toolset, SQLJ, Stage III Engineering, Startup.Com, STEP, SupportNow, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase Learning Connection, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase User Workbench, Sybase Virtual Server Architecture, SybaseWare, Syber Financial, SyberAssist, SyBMD, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, The Enterprise Client/Server Company, The Extensible Software Platform, The Future Is Wide Open, The Learning Connection, The Model For Client/Server Solutions, The Online Information Center, The Power of One, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Viewer, Visual Components, VisualSpeller, VisualWriter, VQL, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, WarehouseArchitect, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server, and XP Server are trademarks of Sybase, Inc. or its subsidiaries.

All other trademarks are property of their respective owners.

Last modified October 2002. Part number 38124-01-0802-01.

# Contents

	<b>About This Manual.....</b>	<b>ix</b>
	SQL Anywhere Studio documentation.....	x
	Documentation conventions.....	xiii
	The Adaptive Server Anywhere sample database.....	xvi
	Finding out more and providing feedback.....	xvii
 <b>PART ONE</b>		
	<b>Relational Database Concepts.....</b>	<b>1</b>
<b>1</b>	<b>Designing Your Database .....</b>	<b>3</b>
	Introduction .....	4
	Database design concepts.....	5
	The design process.....	11
	Designing the database table properties .....	25
<b>2</b>	<b>Working with Database Objects .....</b>	<b>27</b>
	Introduction .....	28
	Working with databases.....	29
	Working with tables .....	38
	Working with views.....	51
	Working with indexes .....	58
	Temporary tables .....	63
<b>3</b>	<b>Ensuring Data Integrity .....</b>	<b>65</b>
	Data integrity overview.....	66
	Using column defaults.....	70
	Using table and column constraints .....	76
	Using domains .....	80
	Enforcing entity and referential integrity.....	83
	Integrity rules in the system tables.....	88

---

<b>4</b>	<b>Using Transactions and Isolation Levels.....</b>	<b>89</b>
	Introduction to transactions.....	90
	Isolation levels and consistency.....	94
	Transaction blocking and deadlock .....	100
	Choosing isolation levels .....	102
	Isolation level tutorials.....	106
	How locking works .....	121
	Particular concurrency issues .....	135
	Replication and concurrency.....	138
	Summary.....	140
 <b>5</b>	 <b>Monitoring and Improving Performance.....</b>	 <b>143</b>
	Top performance tips .....	144
	Using the cache to improve performance .....	152
	Using keys to improve query performance .....	157
	Sorting query results .....	159
	Use of work tables in query processing .....	160
	Monitoring database performance .....	162
	Fragmentation .....	168
	Profiling database procedures .....	172

## **PART TWO**

	<b>Working with Databases .....</b>	<b>181</b>
 <b>6</b>	 <b>Queries: Selecting Data from a Table .....</b>	 <b>183</b>
	Query overview .....	184
	The SELECT list: specifying columns.....	187
	The FROM clause: specifying tables .....	194
	The WHERE clause: specifying rows .....	195
 <b>7</b>	 <b>Summarizing, Grouping and Sorting Query Results...</b>	 <b>207</b>
	Summarizing query results using aggregate functions.....	208
	The GROUP BY clause: organizing query results into groups .....	213
	Understanding GROUP BY.....	214
	The HAVING clause: selecting groups of data .....	218
	The ORDER BY clause: sorting query results .....	220
	The UNION operation: combining queries.....	223
	Standards and compatibility.....	225

---

<b>8</b>	<b>Joins: Retrieving Data from Several Tables .....</b>	<b>227</b>
	Sample database schema .....	228
	How joins work .....	229
	Joins overview.....	230
	Explicit join conditions (the ON phrase).....	236
	Cross joins .....	239
	Inner and outer joins .....	241
	Specialized joins .....	248
	Natural joins .....	255
	Key joins.....	259
<b>9</b>	<b>Using Subqueries .....</b>	<b>273</b>
	Introduction to subqueries.....	274
	Using subqueries in the WHERE clause.....	275
	Subqueries in the HAVING clause.....	276
	Subquery comparison test .....	278
	Quantified comparison tests with ANY and ALL .....	279
	Testing set membership with IN conditions .....	282
	Existence test.....	284
	Outer references .....	286
	Subqueries and joins .....	287
	Nested subqueries .....	289
	How subqueries work.....	291
<b>10</b>	<b>Adding, Changing, and Deleting Data .....</b>	<b>301</b>
	Data modification statements.....	302
	Adding data using INSERT .....	303
	Changing data using UPDATE .....	308
	Changing data using INSERT.....	310
	Deleting data using DELETE .....	311
<b>11</b>	<b>Query Optimization and Execution.....</b>	<b>313</b>
	The role of the optimizer .....	314
	How the optimizer works.....	315
	Query execution algorithms .....	324
	Physical data organization and access.....	337
	Indexes.....	340
	Semantic query transformations .....	349
	Subquery and function caching.....	363
	Reading access plans.....	364

---

## PART THREE

### SQL Dialects and Compatibility..... 381

12	<b>Transact-SQL Compatibility..... 383</b>
	An overview of Transact-SQL support..... 384
	Adaptive Server architectures..... 387
	Configuring databases for Transact-SQL compatibility ..... 393
	Writing compatible SQL statements ..... 401
	Transact-SQL procedure language overview ..... 406
	Automatic translation of stored procedures ..... 409
	Returning result sets from Transact-SQL procedures ..... 410
	Variables in Transact-SQL procedures..... 411
	Error handling in Transact-SQL procedures ..... 412
13	<b>Differences from Other SQL Dialects..... 415</b>
	Adaptive Server Anywhere SQL features ..... 416

## PART FOUR

### Accessing and Moving Data ..... 419

14	<b>Importing and Exporting Data ..... 421</b>
	Introduction to import and export ..... 422
	Importing and exporting data ..... 424
	Importing ..... 428
	Exporting ..... 433
	Rebuilding databases ..... 440
	Extracting data ..... 448
	Migrating databases to Adaptive Server Anywhere ..... 449
	Adaptive Server Enterprise compatibility ..... 454
15	<b>Accessing Remote Data..... 455</b>
	Introduction ..... 456
	Basic concepts to access remote data ..... 458
	Working with remote servers ..... 460
	Working with external logins ..... 465
	Working with proxy tables ..... 467
	Joining remote tables..... 472
	Joining tables from multiple local databases ..... 474
	Sending native statements to remote servers ..... 475

---

	Using remote procedure calls (RPCs) .....	476
	Transaction management and remote data .....	479
	Internal operations .....	481
	Troubleshooting remote data access.....	485
<b>16</b>	<b>Server Classes for Remote Data Access .....</b>	<b>487</b>
	Overview .....	488
	JDBC-based server classes.....	489
	ODBC-based server classes.....	492
 <b>PART FIVE</b>		
	<b>Adding Logic to the Database .....</b>	<b>505</b>
<b>17</b>	<b>Using Procedures, Triggers, and Batches .....</b>	<b>507</b>
	Procedure and trigger overview .....	509
	Benefits of procedures and triggers.....	510
	Introduction to procedures .....	511
	Introduction to user-defined functions.....	518
	Introduction to triggers .....	522
	Introduction to batches.....	529
	Control statements .....	531
	The structure of procedures and triggers.....	535
	Returning results from procedures.....	539
	Using cursors in procedures and triggers .....	545
	Errors and warnings in procedures and triggers.....	548
	Using the EXECUTE IMMEDIATE statement in procedures .....	557
	Transactions and savepoints in procedures and triggers .....	558
	Tips for writing procedures.....	559
	Statements allowed in batches .....	561
	Calling external libraries from procedures .....	562
<b>18</b>	<b>Debugging Logic in the Database .....</b>	<b>571</b>
	Introduction to debugging in the database.....	572
	Tutorial: Getting started with the debugger.....	574
	Common debugger tasks.....	585
	Starting the debugger .....	586
	Working with breakpoints.....	589
	Examining variables .....	593
	Configuring the debugger .....	595

---

<b>Index.....</b>	<b>597</b>
-------------------	------------



# About This Manual

Subject	This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
Audience	This manual is for all users of Adaptive Server Anywhere.
Before you begin	This manual assumes that you have an elementary familiarity with database-management systems and Adaptive Server Anywhere in particular. If you do not have such a familiarity, you should consider reading <i>Adaptive Server Anywhere Getting Started</i> before reading this manual.

---

# SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

## The SQL Anywhere Studio documentation set

The SQL Anywhere Studio documentation set consists of the following books:

- ◆ **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.
- ◆ **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.
- ◆ **Adaptive Server Anywhere Getting Started** This book is for people new to relational databases or new to Adaptive Server Anywhere. It provides a quick start to using the Adaptive Server Anywhere database-management system and introductory material on designing, building, and working with databases.
- ◆ **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases.
- ◆ **Adaptive Server Anywhere SQL User's Guide** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- ◆ **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.
- ◆ **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.

- 
- ◆ **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.
  - ◆ **Adaptive Server Anywhere C2 Security Supplement** Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment. The book does *not* include the security features added to the product since certification.
  - ◆ **MobiLink Synchronization User's Guide** This book describes all aspects of the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.
  - ◆ **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.
  - ◆ **UltraLite User's Guide** This book describes how to build database applications for small devices such as handheld organizers using the UltraLite deployment technology for Adaptive Server Anywhere databases.
  - ◆ **UltraLite User's Guide for PenRight! MobileBuilder** This book is for users of the PenRight! MobileBuilder development tool. It describes how to use UltraLite technology in the MobileBuilder programming environment.
  - ◆ **SQL Anywhere Studio Help** This book is provided online only. It includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools.

In addition to this documentation set, SQL Modeler and InfoMaker include their own online documentation.

## Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- 
- ◆ **Online books** The online books include the complete SQL Anywhere Studio documentation, including both the printed books and the context-sensitive help for SQL Anywhere tools. The online books are updated with each maintenance release of the product, and are the most complete and up-to-date source of documentation.

To access the online books on Windows operating systems, choose Start►Programs►Sybase SQL Anywhere 8►Online Books. You can navigate the online books using the HTML Help table of contents, index, and search facility in the left pane, and using the links and menus in the right pane.

To access the online books on UNIX operating systems, run the following command at a command prompt:

```
dbbooks
```

- ◆ **Printable books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

The PDF files are available on the CD ROM in the *pdf\_docs* directory. You can choose to install them when running the setup program.

- ◆ **Printed books** The following books are included in the SQL Anywhere Studio box:
  - ◆ *Introducing SQL Anywhere Studio.*
  - ◆ *Adaptive Server Anywhere Getting Started.*
  - ◆ *SQL Anywhere Studio Quick Reference.* This book is available only in printed form.

The complete set of books is available as the SQL Anywhere Documentation set from Sybase sales or from e-Shop, the Sybase online store, at <http://e-shop.sybase.com/cgi-bin/eshop.storefront/>.

---

# Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

## Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- ◆ **Keywords** All SQL keywords are shown like the words ALTER TABLE in the following example:

**ALTER TABLE** [ *owner*.]*table-name*

- ◆ **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example.

**ALTER TABLE** [ *owner*.]*table-name*

- ◆ **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

**ADD** *column-definition* [ *column-constraint*, ... ]

One or more list elements are allowed. If more than one is specified, they must be separated by commas.

- ◆ **Optional portions** Optional portions of a statement are enclosed by square brackets.

**RELEASE SAVEPOINT** [ *savepoint-name* ]

These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- ◆ **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

[ **ASC** | **DESC** ]

For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- ◆ **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces.

[ **QUOTES** { **ON** | **OFF** } ]

---

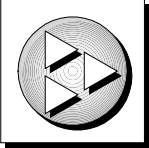
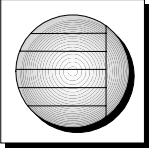
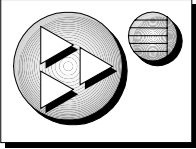
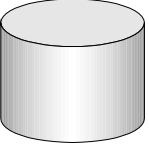

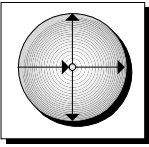

If the QUOTES option is chosen, one of ON or OFF must be provided.  
The brackets and braces should not be typed.

- ◆ **One or more options** If you choose more than one, separate your choices with commas.

{ **CONNECT, DBA, RESOURCE** }

## Graphic icons

The following icons are used in this documentation:

Icon	Meaning
	A client application.
	A database server, such as Sybase Adaptive Server Anywhere or Adaptive Server Enterprise.
	An UltraLite application and database server. In UltraLite, the database server and the application are part of the same process.
	A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.
	Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink Synchronization Server, SQL Remote Message Agent, and the Replication Agent (Log Transfer Manager) for use with Replication Server.
	A Sybase Replication Server.
	A programming interface.

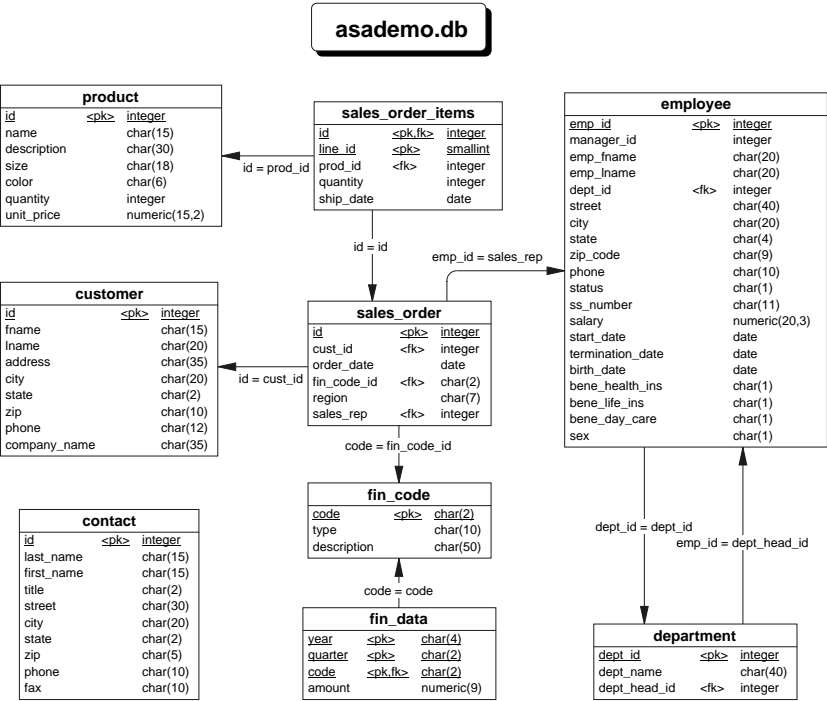
# The Adaptive Server Anywhere sample database

Many of the examples throughout the documentation use the Adaptive Server Anywhere sample database.

The sample database is held in a file named *asademo.db*, and is located in your SQL Anywhere directory.

The sample database represents a small company. It contains internal information about the company (employees, departments, and finances) as well as product information and sales information (sales orders, customers, and contacts). All information in the database is fictional.

The following figure shows the tables in the sample database and how they relate to each other.





---

# Finding out more and providing feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can provide feedback on this documentation and on the software through newsgroups set up to discuss SQL Anywhere technologies. These newsgroups can be found on the *forums.sybase.com* news server.

The newsgroups include the following:

- ◆ sybase.public.sqlanywhere.general.
- ◆ sybase.public.sqlanywhere.linux.
- ◆ sybase.public.sqlanywhere.mobilink.
- ◆ sybase.public.sqlanywhere.product\_futures\_discussion.
- ◆ sybase.public.sqlanywhere.replication.
- ◆ sybase.public.sqlanywhere.ultralite.

## **Newsgroup disclaimer**

iAnywhere Solutions has no obligation to provide solutions, information or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and insure its operation and availability.

iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

---

PART ONE

# Relational Database Concepts

This part describes key concepts and strategies for effective use of Adaptive  
Server Anywhere.

---

## CHAPTER 1

# Designing Your Database

### About this chapter

This chapter introduces the basic concepts of relational database design and gives you step-by-step suggestions for designing your own databases. It uses the expedient technique known as conceptual data modeling, which focuses on entities and the relationships between them.

### Contents

Topic	Page
Introduction	4
Database design concepts	5
The design process	11
Designing the database table properties	25

# Introduction

While designing a database is not a difficult task for small and medium sized databases, it is an important one. Bad database design can lead to an inefficient and possibly unreliable database system. Because client applications are built to work on specific parts of a database, and rely on the database design, a bad design can be difficult to revise at a later date.

☞ For more information, you may also wish to consult an introductory book such as *A Database Primer* by C. J. Date. If you are interested in pursuing database theory, C. J. Date's *An Introduction to Database Systems* is an excellent textbook on the subject.

## Java classes and database design

The addition of Java classes to the available data types extends the relational database concepts on which this chapter is based. Database design involving Java classes is not discussed in this chapter.

☞ For more information on designing databases that take advantage of Java class data types, see "Java database design" on page 121 of the book *ASA Programming Guide*.

# Database design concepts

In designing a database, you plan what things you want to store information about, and what information you will keep about each one. You also determine how these things are related. In the common language of database design, what you are creating during this step is a **conceptual database model**.

## Entities and relationships

The distinguishable objects or things that you want to store information about are called **entities**. The associations between them are called **relationships**. In the language of database description, you can think of entities as nouns and relationships as verbs.

Conceptual models are useful because they make a clean distinction between the entities and relationships. These models hide the details involved in implementing a design in any particular database-management system. They allow you to focus on fundamental database structure. Hence, they also form a common language for the discussion of database design.

## Entity-relationship diagrams

The main component of a conceptual database model is a diagram that shows the entities and relationships. This diagram is commonly called an **entity-relationship diagram**. In consequence, many people use the name entity-relationship modeling to refer to the task of creating a conceptual database model.

Conceptual database design is a top-down design method. There are now sophisticated tools such as Sybase PowerDesigner that help you pursue this method, or other approaches. This chapter is an introductory chapter only, but it does contain enough information for the design of straightforward databases.

## Entities

An entity is the database equivalent of a noun. Distinguishable objects such as employees, order items, departments and products are all examples of entities. In a database, a table represents each entity. The entities that you build into your database arise from the activities for which you will be using the database, such as tracking sales calls and maintaining employee information.

## Attributes

Each entity contains a number of **attributes**. Attributes are particular characteristics of the things that you would like to store. For example, in an employee entity, you might want to store an employee ID number, first and last names, an address, and other particular information that pertains to a particular employee. Attributes are also known as properties.

You depict an entity using a rectangular box. Inside, you list the attributes associated with that entity.

Employee
<u>Employee Number</u>
First Name
Last Name
Address

An identifier is one or more attributes on which all the other attributes depend. It uniquely identifies an item in the entity. Underline the names of attributes that you wish to form part of an identifier.

In the Employee entity, above, the Employee Number uniquely identifies an employee. All the other attributes store information that pertains only to that one employee. For example, an employee number uniquely determines an employee's name and address. Two employees might have the same name or the same address, but you can make sure that they don't have the same employee number. Employee Number is underlined to show that it is an identifier.

It is good practice to create an identifier for each entity. As will be explained later, these identifiers become primary keys within your tables. Primary key values must be unique and cannot be null or undefined. They identify each row in a table uniquely and improve the performance of the database server.

## Relationships

A relationship between entities is the database equivalent of a verb. An employee is a member of a department, or an office is located in a city. Relationships in a database may appear as foreign key relationships between tables, or may appear as separate tables themselves. You will see examples of each in this chapter.

The relationships in the database are an encoding of rules or practices that govern the data in the entities. If each department has one department head, you can create a one-to-one relationship between departments and employees to identify the department head.

Once a relationship is built into the structure of the database, there is no provision for exceptions. There is nowhere to put a second department head. Duplicating the department entry would involve duplicating the department ID, which is the identifier. Duplicate identifiers are not allowed.



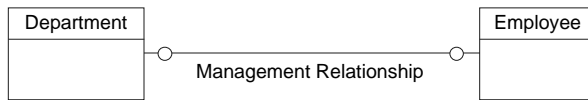
**Tip**

Strict database structure can benefit you, because it can eliminate inconsistencies, such as a department with two managers. On the other hand, you as the designer should make your design flexible enough to allow some expansion for unforeseen uses. Extending a well-designed database is usually not too difficult, but modifying the existing table structure can render an entire database and its client applications obsolete.

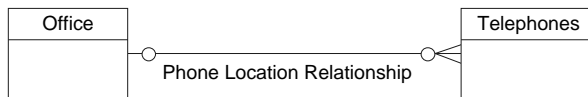
## Cardinality of relationships

There are three kinds of relationships between tables. These correspond to the **cardinality** (number) of the entities involved in the relationship.

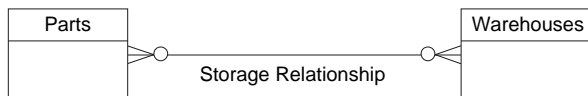
- ◆ **One-to-one relationships** You depict a relationship by drawing a line between two entities. The line may have other markings on it such as the two little circles shown. Later sections explain the purpose of these marks. In the following diagram, one employee manages one department.



- ◆ **One-to-many relationships** The fact that one item contained in Entity 1 can be associated with multiple entities in Entity 2 is denoted by the multiple lines forming the attachment to Entity 2. In the following diagram, one office can have many phones.



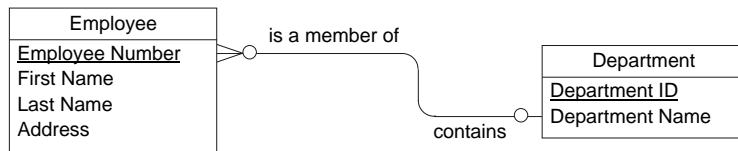
- ◆ **Many-to-many relationships** In this case, draw multiple lines for the connections to both entities. This means that one warehouse can hold many different parts, and one type of part can be stored at many warehouses.



## Roles

You can describe each relationship with two **roles**. Roles are verbs or phrases that describe the relationship from each point of view. For example, a relationship between employees and departments might be described by the following two roles.

- 1 An employee *is a member of* a department.
- 2 A department *contains* an employee.



Roles are very important because they afford you a convenient and effective means of verifying your work.

### Tip

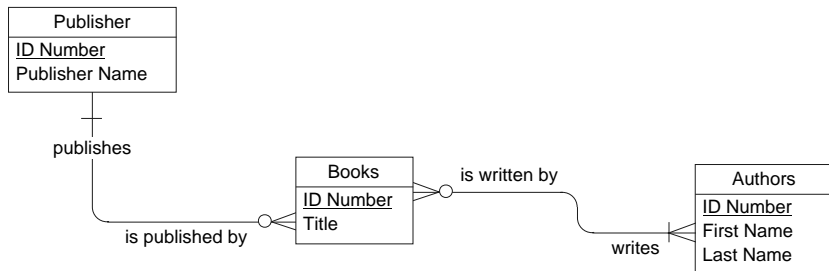
Whether reading from left-to-right or from right-to-left, the following rule makes it easy to read these diagrams: Read the

- 1 name of the first entity,
- 2 role next to the *first entity*,
- 3 cardinality from the connection to the *second entity*, and
- 4 name of the second entity.

### Mandatory elements

The little circles just before the end of the line that denotes the relation serve an important purpose. A circle means that an element can exist in the one entity without a corresponding element in the other entity.

If a cross bar appears in place of the circle, that entity must contain *at least* one element for each element in the other entity. An example will clarify these statements.



This diagram corresponds to the following four statements.

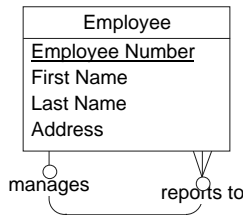
- 1 A publisher publishes *zero or more* books.
- 2 A book is published by *exactly one* publisher.
- 3 A book is written by *one or more* authors.
- 4 An author writes *zero or more* books.

**Tip**

Think of the little circle as the digit 0 and the cross bar as the number one. The circle means *at least zero*. The cross bar means *at least one*.

**Reflexive relationships**

Sometimes, a relationship will exist between entries in a single entity. In this case, the relationship is said to be **reflexive**. Both ends of the relationship attach to a single entity.



This diagram corresponds to the following two statements.

- 1 An employee reports to at most one other employee.
- 2 An employee manages zero or more or more employees.

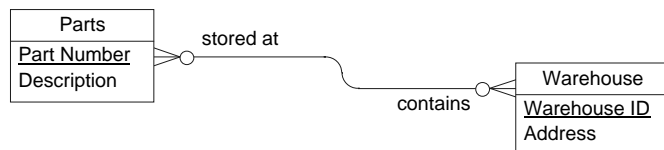
Notice that in the case of this relation, it is essential that the relation be optional in both directions. Some employees are not managers. Similarly, at least one employee should head the organization and hence report to no one.

☞ Naturally, you would also like to specify that an employee cannot be his or her own manager. This restriction is a type of *business rule*. Business rules are discussed later as part of "The design process" on page 11.

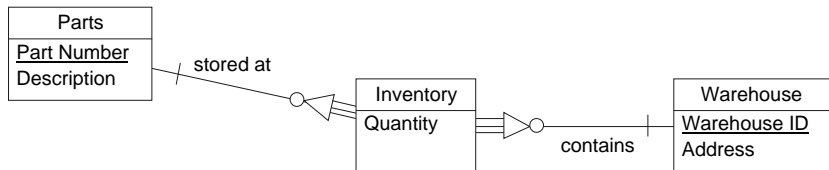
**Changing many-to-many relationships into entities**

When you have attributes associated with a *relationship*, rather than an entity, you can change the relationship into an entity. This situation sometimes arises with many-to-many relationships, when you have attributes that are particular to the relationship and so you cannot reasonably add them to either entity.

Suppose that your parts inventory is located at a number of different warehouses. You have drawn the following diagram.



But you wish to record the quantity of each part stored at each location. This attribute can only be associated with the relationship. Each quantity depends on both the parts and the warehouse involved. To represent this situation, you can redraw the diagram as follows:



Notice the following details of the transformation:


- 1 Two new relations join the relation entity with each of the two original entities. They inherit their names from the two roles of the original relationship: *stored at* and *contains*, respectively.
- 2 Each entry in the Inventory entity demands one mandatory entry in the Parts entity and one mandatory entry in the Warehouse entity. These relationships are mandatory because a storage relationship only makes sense if it is associated with one particular part and one particular warehouse.
- 3 The new entity is dependent on both the Parts entity and on the Warehouse entity, meaning that the new entity is identified by the identifiers of both of these entities. In this new diagram, one identifier from the Parts entity and one identifier from the Warehouse entity uniquely identify an entry in the Inventory entity. The triangles that appear between the circles and the multiple lines that join the two new relationships to the new Inventory entity denote the dependencies.

Do not add either a Part Number or Warehouse ID attribute to the Inventory entity. Each entry in the Inventory entity does depend on both a particular part and a particular warehouse, but the triangles denote this dependence more clearly.

# The design process

There are five major steps in the design process.

- ◆ "Step 1: Identify entities and relationships" on page 11.
- ◆ "Step 2: Identify the required data" on page 14.
- ◆ "Step 3: Normalize the data" on page 16.
- ◆ "Step 4: Resolve the relationships" on page 20.
- ◆ "Step 5: Verify the design" on page 23.

 For more information about implementing the database design, see "Working with Database Objects" on page 27.

## Step 1: Identify entities and relationships

### ❖ To identify the entities in your design and their relationship to each other:

- 1 **Define high-level activities** Identify the general activities for which you will use this database. For example, you may want to keep track of information about employees.
- 2 **Identify entities** For the list of activities, identify the subject areas you need to maintain information about. These subjects will become entities. For example, hire *employees*, assign to a *department*, and determine a *skill* level.
- 3 **Identify relationships** Look at the activities and determine what the relationships will be between the entities. For example, there is a relationship between parts and warehouses. Define two roles to describe each relationship.
- 4 **Break down the activities** You started out with high-level activities. Now, examine these activities more carefully to see if some of them can be broken down into lower-level activities. For example, a high-level activity such as *maintain employee information* can be broken down into:
  - ◆ Add new employees.
  - ◆ Change existing employee information.
  - ◆ Delete terminated employees.

- 5 **Identify business rules** Look at your business description and see what rules you follow. For example, one business rule might be that a department has one and only one department head. These rules will be built into the structure of the database.

## Entity and relationship example

### Example

ACME Corporation is a small company with offices in five locations. Currently, 75 employees work for ACME. The company is preparing for rapid growth and has identified nine departments, each with its own department head.

To help in its search for new employees, the personnel department has identified 68 skills that it believes the company will need in its future employee base. When an employee is hired, the employee's level of expertise for each skill is identified.

### Define high-level activities

Some of the high-level activities for ACME Corporation are:

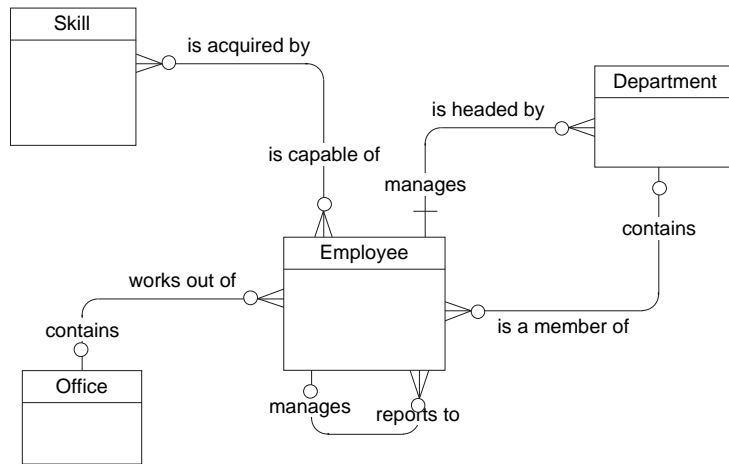
- ◆ Hire employees.
- ◆ Terminate employees.
- ◆ Maintain personal employee information.
- ◆ Maintain information on skills required for the company.
- ◆ Maintain information on which employees have which skills.
- ◆ Maintain information on departments.
- ◆ Maintain information on offices.

### Identify the entities and relationships

Identify the entities (subjects) and the relationships (roles) that connect them. Create a diagram based on the description and high-level activities.

Use boxes to show entities and lines to show relationships. Use the two roles to label each relationship. You should also identify those relationships that are one-to-many, one-to-one, and many-to-many using the appropriate annotation.

Following is a rough entity-relationship diagram. It will be refined throughout the chapter.



Break down the high-level activities

The following lower-level activities below are based on the high-level activities listed above:

- ◆ Add or delete an employee.
- ◆ Add or delete an office.
- ◆ List employees for a department.
- ◆ Add a skill to the skill list.
- ◆ Identify the skills of an employee.
- ◆ Identify an employee's skill level for each skill.
- ◆ Identify all employees that have the same skill level for a particular skill.
- ◆ Change an employee's skill level.

These lower-level activities can be used to identify if any new tables or relationships are needed.

Identify business rules

Business rules often identify one-to-many, one-to-one, and many-to-many relationships.

The kind of business rules that may be relevant include the following:

- ◆ There are now five offices; expansion plans allow for a maximum of ten.
- ◆ Employees can change department or office.
- ◆ Each department has one department head.
- ◆ Each office has a maximum of three telephone numbers.
- ◆ Each telephone number has one or more extensions.

- ◆ When an employee is hired, the level of expertise in each of several skills is identified.
- ◆ Each employee can have from three to twenty skills.
- ◆ An employee may or may not be assigned to an office.

Step 2: Identify the required data

❖ To identify the required data:

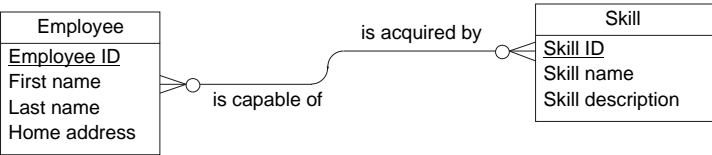
- 1 Identify supporting data.
- 2 List all the data you need to track.
- 3 Set up data for each entity.
- 4 List the available data for each entity. The data that describes an entity (subject) answers the questions who, what, where, when, and why.
- 5 List any data required for each relationship (verb).
- 6 List the data, if any, that applies to each relationship.

Identify supporting data

The supporting data you identify will become the names of the attributes of the entity. For example, the data below might apply to the Employee entity, the Skill entity, and the Expert In relationship.

Employee	Skill	Expert In
Employee ID	Skill ID	Skill level
Employee first name	Skill name	Date skill was acquired
Employee last name	Description of skill	
Employee department		
Employee office		
Employee address		

If you make a diagram of this data, it will look something like this picture:



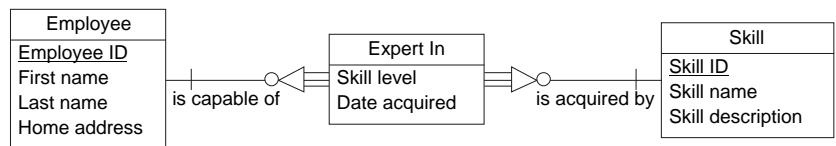
Observe that not all of the attributes you listed appear in this diagram. The missing items fall into two categories:



- 1 Some are contained implicitly in other relationships; for example, Employee department and Employee office are denoted by the relations to the Department and Office entities, respectively.
- 2 Others are not present because they are associated not with either of these entities, but rather the relationship between them. The above diagram is inadequate.

The first category of items will fall naturally into place when you draw the entire entity-relationship diagram.

You can add the second category by converting this many-to-many relationship into an entity.



The new entity depends on both the Employee and the Skill entities. It borrows its identifiers from these entities because it depends on both of them.

## Notes


- ◆ When you are identifying the supporting data, be sure to refer to the activities you identified earlier to see how you will access the data.  
  
For example, you may need to list employees by first name in some situations and by last name in others. To accommodate this requirement, create a First Name attribute and a Last Name attribute, rather than a single attribute that contains both names. With the names separate, you can later create two indexes, one suited to each task.
- ◆ Choose consistent names. Consistency makes it easier to maintain your database and easier to read reports and output windows.  
  
For example, if you choose to use an abbreviated name such as Emp\_status for one attribute, you should not use a full name, such as Employee\_ID, for another attribute. Instead, the names should be Emp\_status and Emp\_ID.
- ◆ At this stage, it is not crucial that the data be associated with the correct entity. You can use your intuition. In the next section, you'll apply tests to check your judgment.

## Step 3: Normalize the data

Normalization is a series of tests that eliminate redundancy in the data and make sure the data is associated with the correct entity or relationship. There are five tests. This section presents the first three of them. These three tests are the most important and so the most frequently used.

### Why normalize?

The goals of normalization are to remove redundancy and to improve consistency. For example, if you store a customer's address in multiple locations, it is difficult to update all copies correctly when they move.

 For more information about the normalization tests, see a book on database design.

### Normal forms

There are several tests for data normalization. When your data passes the first test, it is considered to be in first normal form. When it passes the second test, it is in second normal form, and when it passes the third test, it is in third normal form.

#### ❖ To normalize data in a database:

- 1 List the data.
  - ◆ Identify at least one key for each entity. Each entity must have an identifier.
  - ◆ Identify keys for relationships. The keys for a relationship are the keys from the two entities that it joins.
  - ◆ Check for calculated data in your supporting data list. Calculated data is not normally stored in a relational database.
- 2 Put data in first normal form.
  - ◆ If an attribute can have several different values for the same entry, remove these repeated values.
  - ◆ Create one or more entities or relationships with the data that you remove.
- 3 Put data in second normal form.
  - ◆ Identify entities and relationships with more than one key.
  - ◆ Remove data that depends on only one part of the key.
  - ◆ Create one or more entities and relationships with the data that you remove.
- 4 Put data in third normal form.

- ◆ Remove data that depends on other data in the entity or relationship, not on the key.
- ◆ Create one or more entities and relationships with the data that you remove.

#### Data and identifiers

Before you begin to normalize (test your design), simply list the data and identify a unique identifier each table. The identifier can be made up of one piece of data (attribute) or several (a compound identifier).

The identifier is the set of attributes that uniquely identifies each row in an entity. For example, the identifier for the Employee entity is the Employee ID attribute. The identifier for the Works In relationship consists of the Office Code and Employee ID attributes.

You can make an identifier for each relationship in your database by taking the identifiers from each of the entities that it connects. In the following table, the attributes identified with an asterisk are the identifiers for the entity or relationship.

Entity or Relationship	Attributes
Office	*Office code Office address Phone number
<i>Works in</i>	*Office code *Employee ID
Department	*Department ID Department name
<i>Heads</i>	*Department ID *Employee ID
<i>Member of</i>	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
<i>Expert in</i>	*Skill ID *Employee ID Skill level Date acquired
Employee	*Employee ID last name first name Social security number Address phone number date of birth

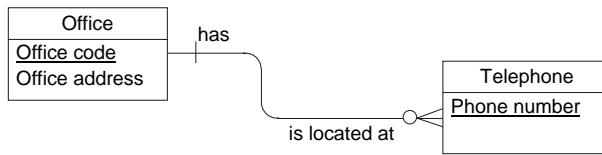
Putting data in first normal form

- ◆ To test for first normal form, look for attributes that can have repeating values.
- ◆ Remove attributes when multiple values can apply to a single item. Move these repeating attributes to a new entity.

In the entity below, Phone number can repeat—an office can have more than one telephone number.

Office and Phone
Office code
Office address
Phone number

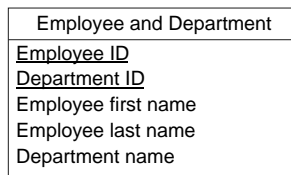
Remove the repeating attribute and make a new entity called Telephone. Set up a relationship between Office and Telephone.



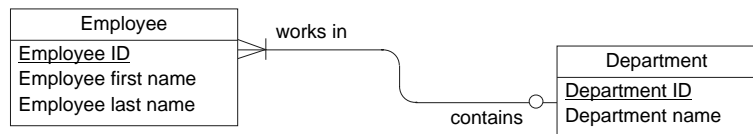
Putting data in  
second normal  
form

- ◆ Remove data that does not depend on the whole key.
- ◆ Look only at entities and relationships whose identifier is composed of more than one attribute. To test for second normal form, remove any data that does not depend on the whole identifier. Each attribute should depend on all of the attributes that comprise the identifier.

In this example, the identifier of the Employee and Department entity is composed of two attributes. Some of the data does not depend on both identifier attributes; for example, the department name depends on only one of those attributes, Department ID, and Employee first name depends only on Employee ID.



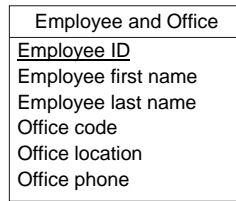
Move the identifier Department ID, which the other employee data does not depend on, to a entity of its own called Department. Also move any attributes that depend on it. Create a relationship between Employee and Department.



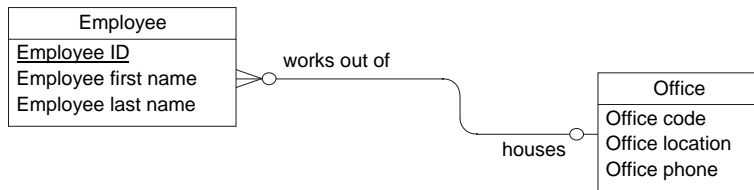
Putting data in third  
normal form

- ◆ Remove data that doesn't depend directly on the key.
- ◆ To test for third normal form, remove any attributes that depend on other attributes, rather than directly on the identifier.

In this example, the Employee and Office entity contains some attributes that depend on its identifier, Employee ID. However, attributes such as Office location and Office phone depend on another attribute, Office code. They do not depend directly on the identifier, Employee ID.



Remove Office code and those attributes that depend on it. Make another entity called Office. Then, create a relationship that connects Employee with Office.



## Step 4: Resolve the relationships

When you finish the normalization process, your design is almost complete. All you need to do is to generate the **physical data model** that corresponds to your conceptual data model. This process is also known as resolving the relationships, because a large portion of the task involves converting the relationships in the conceptual model into the corresponding tables and foreign-key relationships.

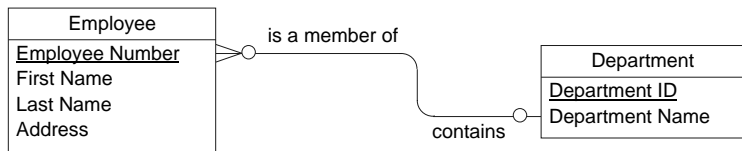
Whereas the conceptual model is largely independent of implementation details, the physical data model is tightly bound to the table structure and options available in a particular database application. In this case, that application is Adaptive Server Anywhere.

Resolving  
relationships that  
do not carry data

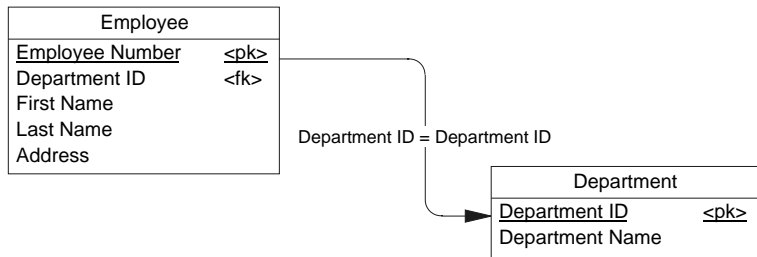
In order to implement relationships that do not carry data, you define foreign keys. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

A database design tool such as the DataArchitect component of Sybase PowerDesigner can generate the physical data model for you. However, if you're doing it yourself there are some basic rules that help you decide where to put the keys.

- ◆ **One to many** An one-to-many relationship always becomes an entity and a foreign key relationship.

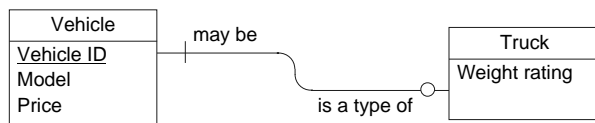


Notice that entities become tables. Identifiers in entities become (at least part of) the primary key in a table. Attributes become columns. In a one-to-many relationship, the identifier in the *one* entity will appear as a new foreign key column in the *many* table.

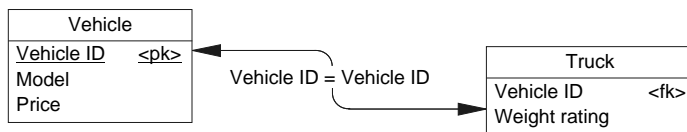


In this example, the Employee *entity* becomes an Employee *table*. Similarly, the Department *entity* becomes a Department *table*. A foreign key called Department ID appears in the Employee table.

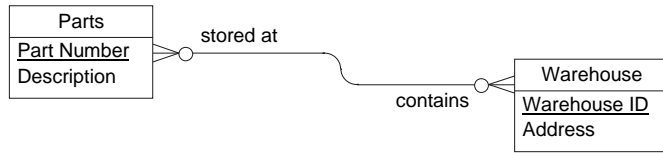
- ◆ **One to one** In a one-to-one relationship, the foreign key can go into either table. If the relationship is mandatory on one side, but optional on the other, it should go on the optional side. In this example, put the foreign key (Vehicle ID) in the Truck table because a vehicle does not have to be a truck.



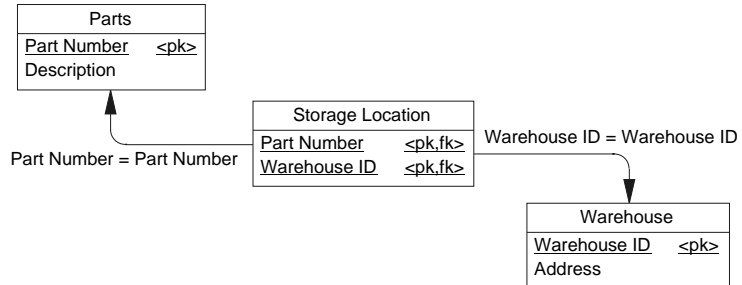
The above entity-relationship model thus resolves the database base structure, below.



- ◆ **Many to many** In a many-to-many relationship, a new table is created with two foreign keys. This arrangement is necessary to make the database efficient.

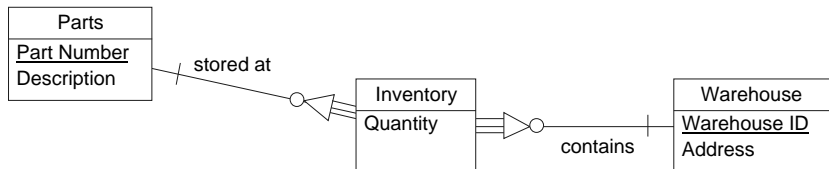


The new Storage Location table relates the Parts and Warehouse tables.

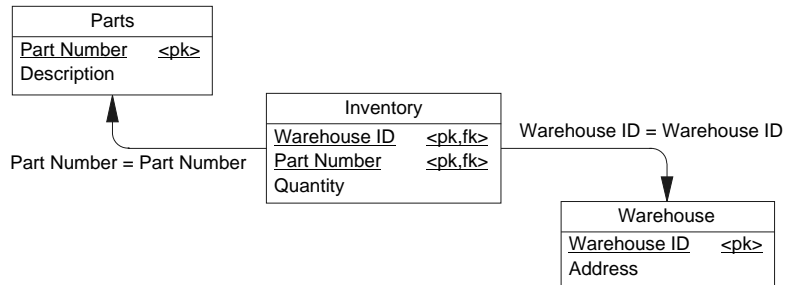


Resolving relationships that carry data

Some of your relationships may carry data. This situation often occurs in many-to-many relationships.



If this is the case, each entity resolves to a table. Each role becomes a foreign key that points to another table.



The **Inventory** entity borrows its identifiers from the **Parts** and **Warehouse** tables, because it depends on both of them. Once resolved, these borrowed identifiers form the primary key of the **Inventory** table.



**Tip**

A conceptual data model simplifies the design process because it hides a lot of details. For example, a many-to-many relationship always generates an extra table and two foreign key references. In a conceptual data model, you can usually denote all of this structure with a single connection.

## Step 5: Verify the design

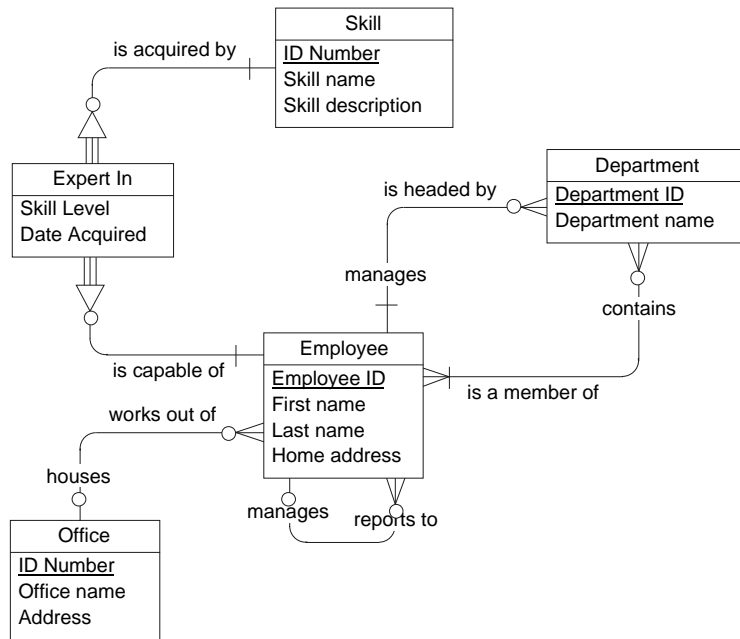
Before you implement your design, you need to make sure that it supports your needs. Examine the activities you identified at the start of the design process and make sure you can access all of the data that the activities require.

- ◆ Can you find a path to get the information you need?
- ◆ Does the design meet your needs?
- ◆ Is all of the required data available?

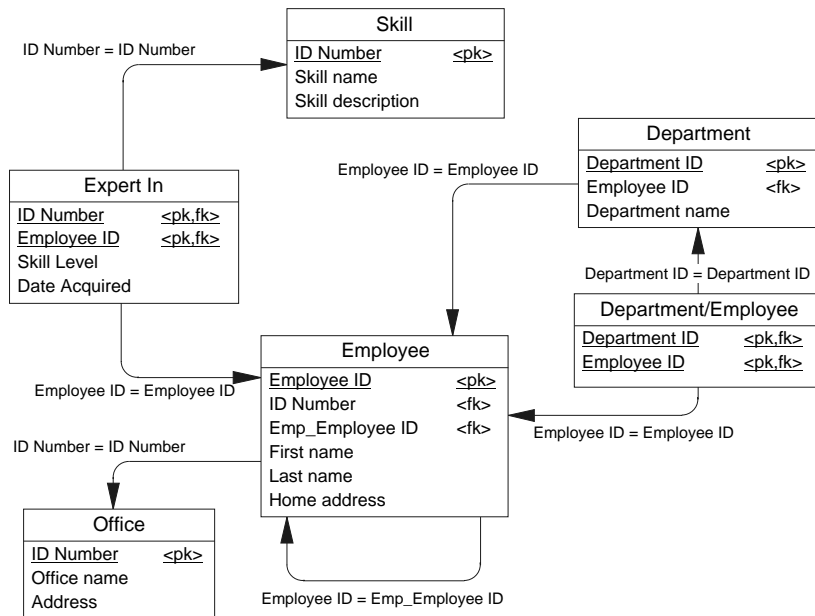
If you can answer yes to all the questions above, you are ready to implement your design.

### Final design

Applying steps 1 through 3 to the database for the little company produces the following entity-relationship diagram. This database is now in third normal form.



The corresponding physical data model appears below.



## Designing the database table properties

The database design specifies which tables you have and what columns each table contains. This section describes how to specify each column's properties.

For each column, you must decide the column name, the data type and size, whether or not NULL values are allowed, and whether you want the database to restrict the values allowed in the column.

### Choosing column names

A column name can be any set of letters, numbers or symbols. However, you must enclose a column name in double quotes if it contains characters other than letters, numbers, or underscores, if it does not begin with a letter, or if it is the same as a keyword.

### Choosing data types for columns

Available data types in Adaptive Server Anywhere include the following:

- ◆ Integer data types
- ◆ Decimal data types
- ◆ Floating-point data types
- ◆ Character data types
- ◆ Binary data types
- ◆ Date/time data types
- ◆ Domains (user-defined data types)
- ◆ Java class data types

🔗 For more information about data types, see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*.

The long binary data type can be used to store information such as images (for instance, stored as bitmaps) or word-processing documents in a database. These types of information are commonly called binary large objects, or BLOBS.

🔗 For more information about each data type, see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*.

## NULL and NOT NULL

If the column value is mandatory for a record, you define the column as being NOT NULL. Otherwise, the column is allowed to contain the NULL value, which represents no value. The default in SQL is to allow NULL values, but you should explicitly declare columns NOT NULL unless there is a good reason to allow NULL values.

☞ For more information about the NULL value, see "NULL value" on page 48 of the book *ASA SQL Reference Manual*. For information on its use in comparisons, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*.

## Choosing constraints

Although the data type of a column restricts the values that are allowed in that column (for example, only numbers or only dates), you may want to further restrict the allowed values.

You can restrict the values of any column by specifying a CHECK constraint. You can use any valid condition that could appear in a WHERE clause to restrict the allowed values. Most CHECK constraints use either the BETWEEN or IN condition.

☞ For more information about valid conditions, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*. For more information about assigning constraints to tables and columns, see "Ensuring Data Integrity" on page 65.

### Example

The sample database has a table called *Department*, which has columns named *dept\_id*, *dept\_name*, and *dept\_head\_id*. Its definition is as follows:

Column	Data Type	Size	Null/Not Null	Constraint
dept_id	integer	—	not null	None
dept_name	char	40	not null	None
dept_head_id	integer	—	null	None

If you specify NOT NULL, a column value must be supplied for every row in the table.

C H A P T E R   2

Working with Database Objects

About this chapter      This chapter describes the mechanics of creating, altering, and deleting database objects such as tables, views, and indexes.

Contents	<table><tr><th>Topic</th><th>Page</th></tr><tr><td>Introduction</td><td>28</td></tr><tr><td>Working with databases</td><td>29</td></tr><tr><td>Working with tables</td><td>38</td></tr><tr><td>Working with views</td><td>51</td></tr><tr><td>Working with indexes</td><td>58</td></tr><tr><td>Temporary tables</td><td>63</td></tr></table>	Topic	Page	Introduction	28	Working with databases	29	Working with tables	38	Working with views	51	Working with indexes	58	Temporary tables	63
Topic	Page														
Introduction	28														
Working with databases	29														
Working with tables	38														
Working with views	51														
Working with indexes	58														
Temporary tables	63														


# Introduction

With the Adaptive Server Anywhere tools, you can create a database file to hold your data. Once this file is created, you can begin managing the database. For example, you can add database objects, such as tables or users, and you can set overall database properties.

This chapter describes how to create a database and the objects within it. It includes procedures for Sybase Central, Interactive SQL, and command-line utilities. If you want more conceptual information before you begin, see the following chapters:

- ◆ "Designing Your Database" on page 3
- ◆ "Ensuring Data Integrity" on page 65
- ◆ "About Sybase Central" on page 50 of the book *Introducing SQL Anywhere Studio*
- ◆ "Using Interactive SQL" on page 75 of the book *ASA Getting Started*

The SQL statements for carrying out the tasks in this chapter are called the **data definition language** (DDL). The definitions of the database objects form the database schema: you can think of the schema as an empty database.

 Procedures and triggers are also database objects, but they are discussed in "Using Procedures, Triggers, and Batches" on page 507.

## Chapter contents

This chapter contains the following material:

- ◆ An introduction to working with database objects (this section)
- ◆ A description of how to create and work with the database itself
- ◆ A description of how to create and alter tables, views, and indexes

# Working with databases

This section describes how to create and work with a database. As you read this section, keep in mind the following simple concepts:


- ◆ The databases that you can create (called relational databases) are a collection of tables, related by primary and foreign keys. These tables hold the information in a database, and the tables and keys together define the structure of the database. A database may be stored in one or more database files, on one or more devices.
- ◆ A database file also contains the system tables, which hold the schema definition as you build your database.

## Creating a database

Adaptive Server Anywhere provides a number of ways to create a database: in Sybase Central, in Interactive SQL, and at the command line. Creating a database is also called initializing it. Once the database is created, you can connect to it and build the tables and other objects that you need in the database.

### Transaction log

When you create a database, you must decide where to place the transaction log. This log stores all changes made to a database, in the order in which they are made. In the event of a media failure on a database file, the transaction log is essential for database recovery. It also makes your work more efficient. By default, it is placed in the same directory as the database file, but this is not recommended for production use.

 For more information on placing the transaction log, see "Configuring your database for data protection" on page 317 of the book *ASA Database Administration Guide*.

### Database file compatibility

An Adaptive Server Anywhere database is an operating system file. It can be copied to other locations just as any other file is copied.

Database files are compatible among all operating systems, except where file system file size limitations or Adaptive Server Anywhere support for large files apply. A database created from any operating system can be used from another operating system by copying the database file(s). Similarly, a database created with a personal server can be used with a network server. Adaptive Server Anywhere servers can manage databases created with earlier versions of the software, but old servers cannot manage newer databases.

## Using other applications to create databases

For more information about limitations, see "Size and number limitations" on page 636 of the book *ASA Database Administration Guide*.

Some application design systems, such as Sybase PowerBuilder, contain tools for creating database objects. These tools construct SQL statements that are submitted to the server, typically through its ODBC interface. If you are using one of these tools, you do not need to construct SQL statements to create tables, assign permissions, and so on.

This chapter describes the SQL statements for defining database objects. You can use these statements directly if you are building your database from an interactive SQL tool, such as Interactive SQL. Even if you are using an application design tool, you may want to use SQL statements to add features to the database if they are not supported by the design tool.

For more advanced use, database design tools such as Sybase PowerDesigner provide a more thorough and reliable approach to developing well-designed databases.

For more information about database design, see "Designing Your Database" on page 3.

## Creating databases (Sybase Central)

You can create a database in Sybase Central using the Create Database utility. After you have created a database, it appears under its server in the Sybase Central object tree.

For more information, see "Creating databases (SQL)" on page 31, and "Creating databases (command line)" on page 31.

### ❖ To create a new database (Sybase Central):

- 1 Choose Tools►Adaptive Server Anywhere 8►Create Database.
- 2 Follow the instructions in the wizard.

### ❖ To create a new database based on a current connection:

- 1 Connect to a database.
- 2 Open the Utilities folder (located within the server folder).
- 3 In the right pane, double-click Create Database.
- 4 Follow the instructions in the wizard.

## Creating databases for Windows CE

You can create databases for Windows CE by copying an Adaptive Server Anywhere database file to the device.



Sybase Central has features to make database creation easy for Windows CE databases. If you have Windows CE services installed on your Windows or Windows NT desktop, you have the option to create a Windows CE database when you create a database from Sybase Central. Sybase Central enforces the requirements for Windows CE databases, and optionally copies the resulting database file to your Windows CE machine.

## Creating databases (SQL)

In Interactive SQL, you can use the CREATE DATABASE statement to create databases. You need to connect to an existing database before you can use this statement.

### ❖ To create a new database (SQL):

- 1 Connect to an existing database.
- 2 Execute a CREATE DATABASE statement.

 For more information, see "CREATE DATABASE statement" on page 273 of the book *ASA SQL Reference Manual*.

### Example

Create a database file in the *c:\temp* directory with the database name *temp.db*.

```
CREATE DATABASE 'c:\\temp\\temp.db'
```

The directory path is relative to the database server. You set the permissions required to execute this statement on the server command line, using the `-gu` command-line option. The default setting requires DBA authority.

The backslash is an escape character in SQL, and must be doubled. For more information, see "Strings" on page 9 of the book *ASA SQL Reference Manual*.

## Creating databases (command line)

You can create a database from a command line with the *dbinit* utility. With this utility, you can include command-line parameters to specify different options for the database.

### ❖ To create a new database (command line):

- 1 Open a command prompt.
- 2 Run the *dbinit* utility. Include any necessary parameters.

For example, to create a database called *company.db* with a 4 Kb page size, type:

```
dbinit -p 4096 company.db
```

🔗 For more information, see "The Initialization utility" on page 465 of the book *ASA Database Administration Guide*.

## Erasing a database

Erasing a database deletes all tables and data from disk, including the transaction log that records alterations to the database. All database files are read-only to prevent accidental modification or deletion of the database files.

In Sybase Central, you can erase a database using the Erase Database utility. You need to connect to a database to access this utility, but the Erase Database wizard lets you specify any database for erasing. In order to erase a non-running database, the database server must be running.

In Interactive SQL, you can erase a database using the DROP DATABASE statement. Required permissions can be set using the database server `-gu` command-line option. The default setting is to require DBA authority.

You can also erase a database from a command line with the *dberase* utility. The database to be erased must not be running when this utility is used.

### ❖ To erase a database (Sybase Central):

- 1 Open the Utilities folder.
- 2 In the right pane, double-click Erase Database.
- 3 Follow the instructions in the wizard.

### ❖ To erase a database (SQL):

- ◆ Execute a DROP DATABASE statement.

For example,

```
DROP DATABASE 'c:\temp\temp.db'
```

### ❖ To erase a database (command line):

- ◆ From a command line, run the *dberase* utility.

For example,

```
dberase company.db
```

🔗 For more information, see "DROP DATABASE statement" on page 399 of the book *ASA SQL Reference Manual*, and "The Erase utility" on page 458 of the book *ASA Database Administration Guide*.

## Disconnecting from a database

When you are finished working with a database, you can disconnect from it. Adaptive Server Anywhere also gives you the ability to disconnect other users from a given database; for more information about doing this in Sybase Central, see "Managing connected users" on page 368 of the book *ASA Database Administration Guide*.

You can obtain a user's *connection-id* using the **connection\_property** function to request the connection number. The following statement returns the connection ID of the current connection:

```
SELECT connection_property( 'number' )
```

### ❖ To disconnect from a database (Sybase Central):

- 1 Select a database.
- 2 Click Tools ► Disconnect.

### ❖ To disconnect from a database (SQL):

- ◆ Execute a DISCONNECT statement.

#### Example 1

The following statement shows how to use DISCONNECT from Interactive SQL to disconnect all connections:

```
DISCONNECT ALL
```

#### Example 2

The following statement shows how to use DISCONNECT in Embedded SQL:

```
EXEC SQL DISCONNECT :conn_name
```

### ❖ To disconnect other users from a database (SQL):

- 1 Connect to an existing database with DBA authority.
- 2 Execute a DROP CONNECTION statement.

#### Example

The following statement drops the connection with ID number 4.

```
DROP CONNECTION 4
```

🔗 For more information, see "DISCONNECT statement [ESQL] [Interactive SQL]" on page 396 of the book *ASA SQL Reference Manual*, and "DROP CONNECTION statement" on page 400 of the book *ASA SQL Reference Manual*.

## Setting properties for database objects

Most database objects (including the database itself) have properties that you can either view or set. Some properties are non-configurable and reflect the settings chosen when you created the database or object. Other properties are configurable.

The best way to view and set properties is to use the property sheets in Sybase Central.

If you are not using Sybase Central, properties can be specified when you create the object with a CREATE statement. If the object already exists, you can modify options with a SET OPTION statement.

### ❖ To view and edit the properties of a database object (Sybase Central):

- 1 In the Sybase Central object tree, open the folder or container in which the object resides.
- 2 Right-click the object and choose Properties from the popup menu.
- 3 Edit the desired properties.

## Setting database options

Database options are configurable settings that change the way the database behaves or performs. In Sybase Central, all of these options are grouped together in the Database Options dialog. In Interactive SQL, you can specify an option in a SET OPTION statement.

### ❖ To set options for a database (Sybase Central):

- 1 Open the desired server.
- 2 Right-click the desired database and choose Options from the popup menu.
- 3 Edit the desired values.

### ❖ To set the options for a database (SQL):

- ◆ Specify the desired properties within a SET OPTION statement.

**Tips**

With the Database Options dialog, you can also set database options for specific users and groups (when you open this dialog for a user or group, it is called the User Options dialog or Group Options dialog respectively).

When you set options for the database itself, you are actually setting options for the PUBLIC group in that database because all users and groups inherit option settings from PUBLIC.

## Specifying a consolidated database

In Sybase Central, you can specify the consolidated database for SQL Remote replication. The consolidated database is the one that serves as the "master" database in the replication setup. The consolidated database contains all of the data to be replicated, while its remote databases may only contain their own subsets of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of all data.

 For more information, see "Consolidated and remote databases" on page 21 of the book *Introducing SQL Anywhere Studio*.

❖ **To set a consolidated database (Sybase Central):**

- 1 Right-click the desired database and choose Properties from the popup menu.
- 2 Click the SQL Remote tab.
- 3 Click the Change button beside the Consolidated Database text box.
- 4 Configure the desired settings.

## Displaying system objects in a database


In a database, a table, view, stored procedures, or domain is a system object. System tables store information about the database itself, while system views, procedures, and domains largely support Sybase Transact-SQL compatibility.

In Interactive SQL, you cannot display a list of all system objects, but you can browse the contents of a system table; for more information, see "Displaying system tables" on page 50.

❖ **To display system objects in a database (Sybase Central):**

- 1 Open the desired server.
- 2 Right-click the desired connected database and choose Filter Objects by Owner.
- 3 Enable **SYS** and **dbo**, and click OK.

The system tables, system views, system procedures, and system domains appear in their respective folders (for example, system tables appear alongside normal tables in the Tables folder).

 For more information, see "System Tables" on page 595 of the book *ASA SQL Reference Manual*.

## Logging SQL statements as you work with a database

As you work with a database in Sybase Central, the application automatically generates SQL statements depending on your actions. You can keep track of these statements either in a separate window or in a specified file.

When you work with Interactive SQL, you can also log statements that you execute; for more information, see "Logging commands" on page 92 of the book *ASA Getting Started*.

❖ **To log SQL statements generated by Sybase Central:**

- 1 Right-click the database and choose Log SQL Statements from the popup menu.
- 2 In the resulting dialog, specify the desired settings.

## Starting a database without connecting

With both Sybase Central and Interactive SQL, you can start a database without connecting to it.

❖ **To start a database on a server without connecting (Sybase Central):**

- 1 Right-click the desired server and choose Start Database from the popup menu.
- 2 In the Start Database dialog, type the desired values.

The database appears under the server as a disconnected database.

❖ **To start a database on a server without connecting (SQL):**

- 1 Start Interactive SQL.
- 2 Execute a START DATABASE statement.

**Example**

Start the database file *c:\asa8\sample\_2.db* as sam2 on the server named sample.

```
START DATABASE 'c:\asa8\sample_2.db'  
AS sam2  
ON sample
```

 For more information, see "START DATABASE statement" on page 549 of the book *ASA SQL Reference Manual*.

## Installing the jConnect metadata support to an existing database

If a database was created without the jConnect metadata support, you can use Sybase Central to install it at a later date.

❖ **To add jConnect metadata support to an existing database (Sybase Central):**

- 1 Connect to the database you want to upgrade.
- 2 Right-click the database and choose Upgrade Database.
- 3 Click Next on the introductory page of the wizard.
- 4 Select the database you want to upgrade from the list.
- 5 You can choose to create a backup of the database if you wish. Click Next.
- 6 Select the Install jConnect Meta-Information Support option.
- 7 Follow the remaining instructions in the wizard.

## Working with tables

When the database is first created, the only tables in the database are the system tables. System tables hold the database schema.

This section describes how to create, alter, and delete tables. You can execute the examples in Interactive SQL, but the SQL statements are independent of the administration tool you use. When you execute queries in Interactive SQL, you can edit the values in the result set.

🔗 For more information, see "Editing table values in Interactive SQL" on page 84 of the book *ASA Getting Started*.

To make it easier for you to re-create the database schema when necessary, create command files to define the tables in your database. The command files should contain the CREATE TABLE and ALTER TABLE statements.

🔗 For more information about groups, tables, and connecting as another user, see "Referring to tables owned by groups" on page 373 of the book *ASA Database Administration Guide* and "Database object names and prefixes" on page 376 of the book *ASA Database Administration Guide*.

## Using the Sybase Central Table Editor

The Table Editor is a tool that you can access through Sybase Central. It provides a quick way of creating and editing tables and their columns.

With the Table Editor, you can create, edit, and delete columns. You can also add and remove columns in the primary key.

### ❖ To access the Table Editor for creating a new table:

- 1 Open the Tables folder.
- 2 In the right pane, double-click Add Table.

### ❖ To access the Table Editor for editing an existing table:

- 1 Open the Tables folder.
- 2 Right-click the table and choose Edit from the popup menu.

Table Editor  
toolbar

Once you have opened the Table Editor, a toolbar provides you with the fields and buttons for common commands.



The first half of this toolbar displays the name of the current table and its owner (or creator). For a new table, you can specify new settings in both of these fields. For an existing table, you can type a new name but you cannot change the owner.

With the buttons on the second half of the toolbar, you can:

- ◆ Add a new column. It appears at the bottom of the list of existing columns.
- ◆ Delete selected columns
- ◆ View the Column property sheet for the selected column
- ◆ View the Advanced Table Properties for the entire table
- ◆ View and change the Table Editor options
- ◆ Save the table but keep the Table Editor open
- ◆ Save the table and close the Table Editor in a single step

As an easy reminder of what these buttons do, you can hold your cursor over each button to see a popup description.

## Using the Advanced Table Properties dialog

In the Table Editor, you can use the Advanced Table Properties dialog to set or inspect a table's type or to type a comment for the table.

### ❖ To view or set advanced table properties:

- 1 Open the Table Editor.
- 2 On the toolbar, click the Advanced Table Properties button.
- 3 Edit the desired values.

#### Advanced Table Properties dialog components

- ◆ **Base table** Designates this table as a base table (one that permanently holds the data until you delete it). You cannot change this setting for existing tables; you can only set the table type when you create a new table.
- ◆ **Dbospace** Lets you select the dbospace used by the table. This option is only available if you create a new table as a base table.
- ◆ **Global temporary table** Designates this table as a global temporary table (one that holds data for a single connection only). You cannot change this setting for existing tables; you can only set the table type when you create a new table.
- ◆ **Delete rows** Sets the global temporary table to delete its rows when a COMMIT is executed.

- ◆ **Preserve rows** Sets the global temporary table to preserve its rows when a COMMIT is executed.
- ◆ **Primary key maximum hash size** The hash size is the number of bytes used to store a value in an index. You can select a hash size only for Adaptive Server Anywhere version 7 databases.
- ◆ **Comment** Provides a place for you to type a comment (text description) of this object. For example, you could use this area to describe the object's purpose in the system.

**Tip**


The table type and comment also appear in the table's property sheet.

## Creating tables

When a database is first created, the only tables in the database are the system tables, which hold the database schema. You can then create new tables to hold your actual data, either with SQL statements in Interactive SQL or with Sybase Central.

There are two types of tables that you can create:

- ◆ **Base table** A table that holds persistent data. The table and its data continue to exist until you explicitly delete the data or drop the table. It is called a base table to distinguish it from temporary tables and from views.
- ◆ **Temporary table** Data in a temporary table is held for a single connection only. Global temporary table definitions (but not data) are kept in the database until dropped. Local temporary table definitions and data exist for the duration of a single connection only.

 For more information about temporary tables, see "Temporary tables" on page 63.

Tables consist of rows and columns. Each column carries a particular kind of information, such as a phone number or a name, while each row specifies a particular entry.

❖ **To create a table (Sybase Central):**

- 1 Connect to the database.
- 2 Open the Tables folder.
- 3 In the right pane, double-click Add Table.

- 4 In the Table Editor:
  - ◆ Type a name for the new table.
  - ◆ Select an owner for the table.
  - ◆ Create columns using the Add Column button on the toolbar.
  - ◆ Specify column properties.
- 5 Save the table and close the Table Editor when finished.

❖ **To create a table (SQL):**

- 1 Connect to the database with DBA authority.
- 2 Execute a CREATE TABLE statement.

**Example**

The following statement creates a new table to describe qualifications of employees within a company. The table has columns to hold an identifying number, a name, and a type (technical or administrative) for each skill.

```
CREATE TABLE skill (  
    skill_id INTEGER NOT NULL,  
    skill_name CHAR( 20 ) NOT NULL,  
    skill_type CHAR( 20 ) NOT NULL  
)
```

🔗 For more information, see "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*, and "Using the Sybase Central Table Editor" on page 38.

## Altering tables

This section describes how to change the structure or column definitions of a table. For example, you can add columns, change various column attributes, or delete columns entirely.

In Sybase Central, you can perform these tasks using the buttons on the Table Editor toolbar. In Interactive SQL, you can perform these tasks with the ALTER TABLE statement.

If you are working with Sybase Central, you can also manage columns (add or remove them from the primary key, change their properties, or delete them) by working with menu commands when you have a column selected in the Columns folder.

🔗 For information on altering database object properties, see "Setting properties for database objects" on page 34.

☞ For information on granting and revoking table permissions, see "Granting permissions on tables" on page 359 of the book *ASA Database Administration Guide* and "Revoking user permissions" on page 366 of the book *ASA Database Administration Guide*.

## Altering tables (Sybase Central)

You can alter tables in Sybase Central using the Table Editor. For example, you can add or delete columns, change column definitions, or change table or column properties.

### ❖ To alter an existing table (Sybase Central):

- 1 Connect to the database.
- 2 Open the Tables folder.
- 3 Right-click the table you want to alter and choose Edit from the popup menu.
- 4 In the Table Editor, make the necessary changes.

#### **Tips**

You can also add columns by opening a table's Columns folder and double-clicking Add Column.

You can also delete columns by opening a table's Columns folder, right-clicking the column, and choosing Delete from the popup menu.

☞ For more information, see "Using the Sybase Central Table Editor" on page 38, and "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*.

## Altering tables (SQL)

You can alter tables in Interactive SQL using the ALTER TABLE statement.

### ❖ To alter an existing table (SQL):

- 1 Connect to the database with DBA authority.
- 2 Execute an ALTER TABLE statement.

### Examples

The following command adds a column to the *skill* table to allow space for an optional description of the skill:

```
ALTER TABLE skill
ADD skill_description CHAR( 254 )
```

This statement adds a column called *skill\_description* that holds up to a few sentences describing the skill.

You can also modify column attributes with the ALTER TABLE statement. The following statement shortens the *skill\_description* column of the sample database from a maximum of 254 characters to a maximum of 80:

```
ALTER TABLE skill
MODIFY skill_description CHAR( 80 )
```

Any current entries that are longer than 80 characters are trimmed to conform to the 80-character limit, and a warning appears.

The following statement changes the name of the *skill\_type* column to *classification*:

```
ALTER TABLE skill
RENAME skill_type TO classification
```


The following statement deletes the *classification* column.

```
ALTER TABLE skill
DROP classification
```

The following statement changes the name of the entire table:

```
ALTER TABLE skill
RENAME qualification
```

These examples show how to change the structure of the database. The ALTER TABLE statement can change just about anything pertaining to a table—you can use it to add or delete foreign keys, change columns from one type to another, and so on. In all these cases, once you make the change, stored procedures, views and any other item referring to this table will no longer work.

 For more information, see "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*, and "Ensuring Data Integrity" on page 65.

## Deleting tables

This section describes how to delete tables from a database. You can use either Sybase Central or Interactive SQL to perform this task. In Interactive SQL deleting a table is also called dropping it.

You cannot delete a table that is being used as an article in a SQL Remote publication. If you try to do this in Sybase Central, an error appears.

### ❖ To delete a table (Sybase Central):

- 1 Connect to the database.

- 2 Open the Tables folder for that database.
- 3 Right-click the table and choose Delete from the popup menu.

❖ **To delete a table (SQL):**

- 1 Connect to the database with DBA authority.
- 2 Execute a DROP TABLE statement.

**Example**

The following DROP TABLE command deletes all the records in the *skill* table and then removes the definition of the *skill* table from the database

```
DROP TABLE skill
```

Like the CREATE statement, the DROP statement automatically executes a COMMIT statement before and after dropping the table. This makes permanent all changes to the database since the last COMMIT or ROLLBACK. The drop statement also drops all indexes on the table.

☞ For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Browsing the information in tables

You can use Sybase Central or Interactive SQL to browse the data held within the tables of a database.

If you are working in Sybase Central, view the data in a table by clicking the Data tab in the right pane.

If you are working in Interactive SQL, execute the following statement:

```
SELECT * FROM table-name
```

You can edit the data in the table from the Interactive SQL Results tab or from Sybase Central.

## Managing primary keys

The **primary key** is a unique identifier that is comprised of a column or combination of columns with values that do not change over the life of the data in the row. Because uniqueness is essential to good database design, it is best to specify a primary key when you define the table.

This section describes how to create and edit primary keys in your database. You can use either Sybase Central or Interactive SQL to perform these tasks.

**Column order in multi-column primary keys**

Primary key column order is determined by the order of the columns during table creation. It is not based on the order of the columns as specified in the primary key declaration.

**Managing primary keys (Sybase Central)**

In Sybase Central, the primary key of a table appears in several places:

- ◆ On the Columns tab of the table's property sheet.
- ◆ In the Columns folder of the table.
- ◆ In the Table Editor.

The primary key columns have special icons to distinguish them from non-key columns. The lists in both the table property sheet and the Columns folder display the primary key columns (along with the non-key columns) in the order that they were created in the database. This may differ from the actual ordering of columns in the primary key.

**❖ To create and edit the primary key using a property sheet:**

- 1 Open the Tables folder.
- 2 Right-click the table and choose Properties from the popup menu.
- 3 Click the Columns tab of the property sheet.
- 4 Use the buttons to change the primary key.

**❖ To create and edit the primary key using the Columns folder:**

- 1 Open the Tables folder and double-click a table.
- 2 Open the Columns folder for that table and right-click a column.
- 3 From the popup menu, do one of the following:
  - ◆ Choose Add to Primary Key if the column is not yet part of the primary key and you want to add it.
  - ◆ Choose Remove From Primary Key if the column is part of the primary key and you want to remove it.

**❖ To create and edit the primary key using the Table Editor:**

- 1 Open the Tables folder.
- 2 Right-click a table and choose Edit from the popup menu.

- 3 In the Table Editor, click the icons in the Key fields (at the far left of the Table Editor) to add a column to the primary key or remove it from the key.

## Managing primary keys (SQL)

You can create and modify the primary key in Interactive SQL using the CREATE TABLE and ALTER TABLE statements. These statements let you set many table attributes, including column constraints and checks.

Columns in the primary key cannot contain NULL values. You must specify NOT NULL on the column in the primary key.

### ❖ To modify the primary key of an existing table (SQL):

- 1 Connect to the database with DBA authority.
- 2 Execute a ALTER TABLE statement.

#### Example 1

The following statement creates the same *skill* table as before, except that it adds a primary key:

```
CREATE TABLE skill (  
    skill_id INTEGER NOT NULL,  
    skill_name CHAR( 20 ) NOT NULL,  
    skill_type CHAR( 20 ) NOT NULL,  
    primary key( skill_id )  
)
```

The primary key values must be unique for each row in the table which, in this case, means that you cannot have more than one row with a given *skill\_id*. Each row in a table is uniquely identified by its primary key.

#### Example 2

The following statement adds the columns *skill\_id* and *skill\_type* to the primary key for the *skill* table:

```
ALTER TABLE skill (  
    ADD PRIMARY KEY ( "skill_id", "skill_type" )  
)
```

If a PRIMARY KEY clause is specified in an ALTER TABLE statement, the table must not already have a primary key that was created by the CREATE TABLE statement or another ALTER TABLE statement.

#### Example 3

The following statement removes all columns from the primary key for the *skill* table. Before you delete a primary key, make sure you are aware of the consequences in your database.

```
ALTER TABLE skill  
    DELETE PRIMARY KEY
```



For more information, see "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*, and "Managing primary keys (Sybase Central)" on page 45.

## Managing foreign keys

This section describes how to create and edit foreign keys in your database. You can use either Sybase Central or Interactive SQL to perform these tasks.

Foreign keys are used to relate values in a child table (or foreign table) to those in a parent table (or primary table). A table may have multiple foreign keys that refer to multiple parent tables linking various types of information.

### Managing foreign keys (Sybase Central)

In Sybase Central, the foreign key of a table appears in the Foreign Keys folder (located within the table container).

You cannot create a foreign key in a table if the table contains values for the foreign columns that can't be matched to values in the primary table's primary key.

After you have created a foreign key, you can keep track of them in each table's Referenced By folder; this folder displays any foreign tables that reference the currently selected table.

#### ❖ To create a new foreign key in a given table (Sybase Central):

- 1 In the Tables folder, open a table.
- 2 Open the Foreign Keys folder for that table.
- 3 In the right pane, double-click Add Foreign Key.
- 4 Follow the instructions in the wizard.

#### ❖ To delete a foreign key (Sybase Central):

- 1 In the Tables folder, open a table.
- 2 Open the Foreign Keys folder for that table.
- 3 Right-click the foreign key you want to delete and choose Delete from the popup menu.

#### ❖ To display which tables have foreign keys that reference a given table (Sybase Central):

- 1 Open the desired table.

- 2 Open the Referenced By folder.

**Tips**

When you create a foreign key using the wizard, you can set properties for the foreign key. To set or change properties after the foreign key is created, right-click the foreign key and choose Properties from the popup menu.

You can view the properties of a referencing table by right-clicking the table and choosing Properties from the popup menu.

## Managing foreign keys (SQL)

You can create and modify the foreign key in Interactive SQL using the CREATE TABLE and ALTER TABLE statements. These statements let you set many table attributes, including column constraints and checks.

A table can only have one primary key defined, but it may have as many foreign keys as necessary.

### ❖ To modify the foreign key of an existing table (SQL):

- 1 Connect to the database with DBA authority.
- 2 Execute a ALTER TABLE statement.

#### Example 1

You can create a table named *emp\_skill*, which holds a description of each employee's skill level for each skill in which they are qualified, as follows:

```
CREATE TABLE emp_skill(  
    emp_id INTEGER NOT NULL,  
    skill_id INTEGER NOT NULL,  
    "skill level" INTEGER NOT NULL,  
    PRIMARY KEY( emp_id, skill_id ),  
    FOREIGN KEY REFERENCES employee,  
    FOREIGN KEY REFERENCES skill  
)
```

The *emp\_skill* table definition has a primary key that consists of two columns: the *emp\_id* column and the *skill\_id* column. An employee may have more than one skill, and so appear in several rows, and several employees may possess a given skill, so that the *skill\_id* may appear several times. However, there may be no more than one entry for a given employee and skill combination.

The *emp\_skill* table also has two foreign keys. The foreign key entries indicate that the *emp\_id* column must contain a valid employee number from the *employee* table, and that the *skill\_id* must contain a valid entry from the *skill* table.

#### Example 2

You can add a foreign key called *foreignkey* to the existing table *skill* and reference this foreign key to the primary key in the table *contact*, as follows:

```
ALTER TABLE skill
ADD FOREIGN KEY "foreignkey" ("skill_id")
REFERENCES "DBA"."contact" ("id")
```


This example creates a relationship between the *skill\_id* column of the table *skill* (the foreign table) and the *id* column of the table *contact* (the primary table). The "DBA" signifies the owner of the table *contact*.

#### Example 3

You can specify properties for the foreign key as you create it. For example, the following statement creates the same foreign key as in Example 2, but it defines the foreign key as NOT NULL along with restrictions for when you update or delete.

```
ALTER TABLE skill
ADD NOT NULL FOREIGN KEY "foreignkey" ("skill_id")
REFERENCES "DBA"."contact" ("id")
ON UPDATE RESTRICT
ON DELETE RESTRICT
```

In Sybase Central, you can also specify properties in the foreign key creation wizard or on the foreign key's property sheet.

 For more information, see "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*, and "Managing foreign keys (Sybase Central)" on page 47.

## Copying tables or columns within/between databases

With Sybase Central, you can copy existing tables or columns and insert them into another location in the same database or into a completely different database.

 If you are not using Sybase Central, see one of the following locations:

- ◆ To insert SELECT statement results into a given location, see "SELECT statement" on page 526 of the book *ASA SQL Reference Manual*.
- ◆ To insert a row or selection of rows from elsewhere in the database into a table, see "INSERT statement" on page 463 of the book *ASA SQL Reference Manual*.

## Displaying system tables

In a database, a table, view, stored procedure, or domain is a system object. **System tables** store the database's schema, or information about the database itself. System views, procedures, and domains largely support Sybase Transact-SQL compatibility.

All the information about tables in a database appears in the system tables. The information is distributed among several tables.

🔗 For more information, see "System Tables" on page 595 of the book *ASA SQL Reference Manual*.

### ❖ To display system tables (Sybase Central):

- 1 Right-click the desired connected database and choose Filter Objects by Owner from the popup menu.
- 2 Select the checkbox beside SYS and click OK.

The system tables, system views, system procedures, and system domains appear in their respective folders (for example, system tables appear alongside normal tables in the Tables folder).

### ❖ To browse system tables (SQL):

- 1 Connect to a database.
- 2 Execute a SELECT statement, specifying the system table you want to browse. The system tables are owned by the SYS user ID.

#### Example

Display the contents of the table *sys.systable* on the Results tab in the Results pane.

```
SELECT *  
FROM SYS.SYSTABLE
```

# Working with views

	<p>Views are computed tables. You can use views to show database users exactly the information you want to present, in a format you can control.</p>
<p>Similarities between views and base tables</p>	<p>Views are similar to the permanent tables of the database (a permanent table is also called a base table) in many ways:</p> <ul style="list-style-type: none"><li>◆ You can assign access permissions to views just as to base tables.</li><li>◆ You can perform SELECT queries on views.</li><li>◆ You can perform UPDATE, INSERT, and DELETE operations on some views.</li><li>◆ You can create views based on other views.</li></ul>
<p>Differences between views and permanent tables</p>	<p>There are some differences between views and permanent tables:</p> <ul style="list-style-type: none"><li>◆ You cannot create indexes on views.</li><li>◆ You cannot perform UPDATE, INSERT, and DELETE operations on all views.</li><li>◆ You cannot assign integrity constraints and keys to views.</li><li>◆ Views refer to the information in base tables, but do not hold copies of that information. Views are recomputed each time you invoke them.</li></ul>
<p>Benefits of tailoring access</p>	<p>Views let you tailor access to data in the database. Tailoring access serves several purposes:</p> <ul style="list-style-type: none"><li>◆ <b>Improved security</b> By allowing access to only the information that is relevant.</li><li>◆ <b>Improved usability</b> By presenting users and application developers with data in a more easily understood form than in the base tables.</li><li>◆ <b>Improved consistency</b> By centralizing in the database the definition of common queries.</li></ul>

## Creating views

When you browse data, a SELECT statement operates on one or more tables and produces a result set that is also a table. Just like a base table, a result set from a SELECT query has columns and rows. A view gives a name to a particular query, and holds the definition in the database system tables.

Suppose you frequently need to list the number of employees in each department. You can get this list with the following statement:

```
SELECT dept_ID, count(*)
FROM employee
GROUP BY dept_ID
```

You can create a view containing the results of this statement using either Sybase Central or Interactive SQL.

❖ **To create a new view (Sybase Central):**

- 1 Connect to a database.
- 2 Open the Views folder for that database.
- 3 In the right pane, double-click Add View (Wizard).
- 4 Follow the instructions in the wizard. When you're finished, the Code Editor automatically opens.
- 5 Complete the code by entering the table and the columns you want to use. For the example above, type **employee** and **dept\_ID**.
- 6 From the File menu of the Code Editor, choose Save/Execute in Database.

New views appear in the Views folder.

❖ **To create a new view (SQL):**

- 1 Connect to a database.
- 2 Execute a CREATE VIEW statement.

Example

Create a view called *DepartmentSize* that contains the results of the SELECT statement given at the beginning of this section:

```
CREATE VIEW DepartmentSize AS
SELECT dept_ID, count(*)
FROM employee
GROUP BY dept_ID
```

Since the information in a view is not stored separately in the database, referring to the view executes the associated SELECT statement to retrieve the appropriate data.

On one hand, this is good because it means that if someone modifies the *employee* table, the information in the *DepartmentSize* view is automatically brought up to date. On the other hand, complicated SELECT statements may increase the amount of time SQL requires to find the correct information every time you use the view.

For more information, see "CREATE VIEW statement" on page 371 of the book *ASA SQL Reference Manual*.

## Using views

### Restrictions on SELECT statements

There are some restrictions on the SELECT statements you can use as views. In particular, you cannot use an ORDER BY clause in the SELECT query. A characteristic of relational tables is that there is no significance to the ordering of the rows or columns, and using an ORDER BY clause would impose an order on the rows of the view. You can use the GROUP BY clause, subqueries, and joins in view definitions.

To develop a view, tune the SELECT query by itself until it provides exactly the results you need in the format you want. Once you have the SELECT query just right, you can add a phrase in front of the query to create the view. For example,

```
CREATE VIEW viewname AS
```

### Updating views

UPDATE, INSERT, and DELETE statements are allowed on some views, but not on others, depending on its associated SELECT statement.

You cannot update views containing aggregate functions, such as COUNT(\*). Nor can you update views containing a GROUP BY clause in the SELECT statement, or views containing a UNION operation. In all these cases, there is no way to translate the UPDATE into an action on the underlying tables.

### Copying views

In Sybase Central, you can copy views between databases. To do so, select the view in the right pane of Sybase Central and drag it to the Views folder of another connected database. A new view is then created, and the original view's code is copied to it.

Note that only the view code is copied to the new view. The other view properties, such as permissions, are not copied.

## Using the WITH CHECK OPTION clause

Even when INSERT and UPDATE statements are allowed against a view, it is possible that the inserted or updated rows in the underlying tables may not meet the requirements for the view itself. For example, the view has no new rows even though the INSERT or UPDATE modified the underlying tables.

### Examples using the WITH CHECK OPTION clause

The following example illustrates the usefulness of the WITH CHECK OPTION clause. This optional clause is the final clause in the CREATE VIEW statement.

❖ **To create a view displaying the employees in the sales department (SQL):**

- ◆ Type the following statements:

```
CREATE VIEW sales_employee
AS SELECT emp_id,
         emp_fname,
         emp_lname,
         dept_id
FROM employee
WHERE dept_id = 200
```

The contents of this view are as follows:

```
SELECT *
FROM sales_employee
```

They appear in Interactive SQL as follows:

Emp_id	Emp_fname	Emp_lname	Dept_id
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
...	...	...	...

- ◆ **Transfer Philip Chin to the marketing department** This view update causes the entry to vanish from the view, as it no longer meets the view selection criterion.

```
UPDATE sales_employee
SET dept_id = 400
WHERE emp_id = 129
```

- ◆ **List all employees in the sales department** Inspect the view.

```
SELECT *
FROM sales_employee
```

Emp_id	emp_fname	emp_lname	dept_id
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
641	Thomas	Powell	200
...	...	...	...



When you create a view using the **WITH CHECK OPTION**, any **UPDATE** or **INSERT** statement on the view is checked to ensure that the new row matches the view condition. If it does not, the operation causes an error and is rejected.

The following modified `sales_employee` view rejects the update statement, generating the following error message:

Invalid value for column 'dept\_id' in table 'employee'

- ◆ **Create a view displaying the employees in the sales department (second attempt)** Use **WITH CHECK OPTION** this time.

```
CREATE VIEW sales_employee
AS SELECT emp_id, emp_fname, emp_lname, dept_id
FROM employee
WHERE dept_id = 200
WITH CHECK OPTION
```

The check option is inherited

If a view (say `V2`) is defined on the `sales_employee` view, any updates or inserts on `V2` that cause the **WITH CHECK OPTION** criterion on `sales_employee` to fail are rejected, even if `V2` is defined without a check option.

## Modifying views

You can modify a view using both Sybase Central and Interactive SQL. When doing so, you cannot rename an existing view directly. Instead, you must create a new view with the new name, copy the previous code to it, and then delete the old view.

In Sybase Central, a Code Editor lets you edit the code of views, procedures, and functions. In Interactive SQL, you can use the **ALTER VIEW** statement to modify a view. The **ALTER VIEW** statement replaces a view definition with a new definition, but it maintains the permissions on the view.

🔗 For more information on altering database object properties, see "Setting properties for database objects" on page 34.

🔗 For more information on setting permissions, see "Granting permissions on tables" on page 359 of the book *ASA Database Administration Guide* and "Granting permissions on views" on page 361 of the book *ASA Database Administration Guide*. For information about revoking permissions, see "Revoking user permissions" on page 366 of the book *ASA Database Administration Guide*.

### ❖ To edit a view definition (Sybase Central):

- 1 Open the Views folder.

- 2 Right-click the desired view and choose Edit from the popup menu.
- 3 In the Code Editor, edit the view's code.
- 4 To execute the code in the database, choose File►Save View.

❖ **To edit a view definition (SQL):**

- 1 Connect to a database with DBA authority or as the owner of the view.
- 2 Execute an ALTER VIEW statement.

Example

Rename the column names of the *DepartmentSize* view (described in the "Creating views" on page 51 section) so that they have more informative names.

```
ALTER VIEW DepartmentSize
  (Dept_ID, NumEmployees)
AS
  SELECT dept_ID, count(*)
  FROM Employee
  GROUP BY dept_ID
```

🔗 For more information, see "ALTER VIEW statement" on page 241 of the book *ASA SQL Reference Manual*.

## Deleting views

You can delete a view in both Sybase Central and Interactive SQL.

❖ **To delete a view (Sybase Central):**

- 1 Open the Views folder.
- 2 Right-click the desired view and choose Delete from the popup menu.

❖ **To delete a view (SQL):**

- 1 Connect to a database with DBA authority or as the owner of the view.
- 2 Execute a DROP VIEW statement.

Examples

Remove a view called *DepartmentSize*.

```
DROP VIEW DepartmentSize
```

🔗 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Browsing the information in views

To browse the data held within the views, you can use the Interactive SQL utility. This utility lets you execute queries to identify the data you want to view. For more information about using these queries, see "Queries: Selecting Data from a Table" on page 183.

If you are working in Sybase Central, you can right-click a view on which you have permission and choose View Data in Interactive SQL from the popup menu. This command opens Interactive SQL with the view contents displayed on the Results tab in the Results pane. To browse the view, Interactive SQL executes a `select * from owner.view` statement.

## Views in the system tables

All the information about views in a database is held in the system table `SYS.SYSTABLE`. The information is presented in a more readable format in the system view `SYS.SYSVIEWS`. For more information about these, see "SYSTABLE system table" on page 657 of the book *ASA SQL Reference Manual*, and "SYSVIEWS system view" on page 678 of the book *ASA SQL Reference Manual*.

You can use Interactive SQL to browse the information in these tables. Type the following statement in the SQL Statements pane to see all the columns in the `SYS.SYSVIEWS` view:

```
SELECT *  
FROM SYS.SYSVIEWS
```

To extract a text file containing the definition of a specific view, use a statement such as the following:

```
SELECT viewtext FROM SYS.SYSVIEWS  
WHERE viewname = 'DepartmentSize';  
OUTPUT TO viewtext.SQL  
FORMAT ASCII
```

## Working with indexes

Performance is an important consideration when designing and creating your database. Indexes can dramatically improve the performance of statements that search for a specific row or a specific subset of the rows.

### When to use indexes


An index provides an ordering on the columns of a table. An index is like a telephone book that initially sorts people by surname, and then sorts identical surnames by first names. This ordering speeds up searches for phone numbers for a particular surname, but it does not provide help in finding the phone number at a particular address. In the same way, a database index is useful only for searches on a specific column or columns.

The database server automatically tries to use an index when a suitable index exists and when using one will improve performance.

Indexes get more useful as the size of the table increases. The average time to find a phone number at a given address increases with the size of the phone book, while it does not take much longer to find the phone number of, say, K. Kaminski, in a large phone book than in a small phone book.


### Use indexes for frequently-searched columns

Indexes require extra space and may slightly reduce the performance of statements that modify the data in the table, such as INSERT, UPDATE, and DELETE statements. However, they can improve search performance dramatically and are highly recommended whenever you search data frequently.

 For more information about performance, see "Using indexes" on page 146.

If a column is already a primary key or foreign key, searches are fast on it because Adaptive Server Anywhere automatically indexes key columns. Thus, manually creating an index on a key column is not necessary and generally not recommended. If a column is only part of a key, an index may help.

Adaptive Server Anywhere automatically uses indexes to improve the performance of any database statement whenever it can. There is no need to refer to indexes once they are created. Also, the index is updated automatically when rows are deleted, updated or inserted.

 For information on altering database object properties, see "Setting properties for database objects" on page 34.

### Using Clustered Indexes

Although standard indexes can dramatically improve the performance of statements that search for a specific row or a specific subset of the rows, two rows appearing sequentially in the index do not necessarily appear on the same page in the database.

However, you can further improve the performance of indexes by creating clustered indexes. Clustered indexes in Adaptive Server Anywhere store the table rows in approximately the same order as they appear in the corresponding index.

Using the clustered index feature increases the chance that the two rows will appear on the same page in the database. This can lead to performance benefits by further reducing the number of times each page needs to be read into memory.

For example, in a case where you select two rows that appear sequentially in a clustered index, it is possible that you are retrieving two rows that appear sequentially on the same page, thus reducing the number of pages to read into memory by half.

The clustering of indexes in Adaptive Server Anywhere is approximate. While the server attempts to preserve the key order, total clustering is not guaranteed. As well, the clustering degrades over time, as more and more rows are inserted into your database.

You can implement clustered indexes using the following statements:

- ◆ The CREATE TABLE statement
- ◆ The ALTER TABLE statement
- ◆ The CREATE INDEX statement
- ◆ The DECLARE LOCAL TEMPORARY TABLE statement

Several statements work in conjunction with each other to allow you to maintain and restore the clustering effect:

- ◆ The UNLOAD TABLE statement allows you to unload a table in the order of the index key.
- ◆ The LOAD TABLE statement inserts rows into the table in the order of the index key.
- ◆ The INSERT statement attempts to put new rows on the same table page as the one containing adjacent rows as per the primary key order.
- ◆ The REORGANIZE table statement can restore the clustering by rearranging the rows according to the clustering index. On tables where clustering is not specified, tables are ordered using the primary key.

The Optimizer assumes that the table rows are stored in key order and costs index scans accordingly.

## Creating indexes

Indexes are created on a specified table. You cannot create an index on a view. To create an index, you can use either Sybase Central or Interactive SQL.

❖ **To create a new index for a given table (Sybase Central):**

- 1 Open the Tables folder.
- 2 Double-click a table.
- 3 Open the Indexes folder for that table.
- 4 In the right pane, double-click Add Index.
- 5 Follow the instructions in the wizard.

All indexes appear in the Indexes folder of the associated table.

❖ **To create a new index for a given table (SQL):**

- 1 Connect to a database with DBA authority or as the owner of the table on which the index is created.
- 2 Execute a CREATE INDEX statement.

### Example

To speed up a search on employee surnames in the sample database, you could create an index called *EmpNames* with the following statement:

```
CREATE INDEX EmpNames  
ON employee (emp_lname, emp_fname)
```

🔗 For more information, see "CREATE INDEX statement" on page 300 of the book *ASA SQL Reference Manual*, and "Monitoring and Improving Performance" on page 143.

## Validating indexes

You can validate an index to ensure that every row referenced in the index actually exists in the table. For foreign key indexes, a validation check also ensures that the corresponding row exists in the primary table, and that their hash values match. This check complements the validity checking carried out by the VALIDATE TABLE statement.

❖ **To validate an index (Sybase Central):**

- 1 Connect to a database with DBA authority or as the owner of the table on which the index is created.

- 2 Open the Tables folder.
- 3 Double-click a table.
- 4 Open the Indexes folder for that table.
- 5 Right-click the index and choose Validate from the popup menu.

❖ **To validate an index (SQL):**

- 1 Connect to a database with DBA authority or as the owner of the table on which the index is created.
- 2 Execute a `VALIDATE INDEX` statement.

❖ **To validate an index (command line):**

- 1 Open a command prompt.
- 2 Run the *dbvalid* utility.

## Examples

Validate an index called *EmployeeIndex*. If you supply a table name instead of an index name, the primary key index is validated.

```
VALIDATE INDEX EmployeeIndex
```

Validate an index called *EmployeeIndex*. The `-i` switch specifies that each object name given is an index.

```
dbvalid -i EmployeeIndex
```

☞ For more information, see "VALIDATE INDEX statement" on page 585 of the book *ASA SQL Reference Manual*, and "The Validation utility" on page 526 of the book *ASA Database Administration Guide*.

## Deleting indexes

If an index is no longer required, you can remove it from the database in Sybase Central or in Interactive SQL.

❖ **To delete an index (Sybase Central):**

- 1 For the desired table, open the Indexes folder.
- 2 Right-click the desired index and choose Delete from the popup menu.

❖ **To delete an index (SQL):**

- 1 Connect to a database with DBA authority or as the owner of the table associated with the index.

- 2 Execute a DROP INDEX statement.

**Example**

The following statement removes the index from the database:

```
DROP INDEX EmpNames
```

🔗 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Indexes in the system tables

All the information about indexes in a database is held in the system tables SYS.SYSINDEX and SYS.SYSIXCOL. The information is presented in a more readable format in the system view SYS.SYSINDEXES. You can use Sybase Central or Interactive SQL to browse the information in these tables.



## Temporary tables

Temporary tables, whether local or global, serve the same purpose: temporary storage of data. The difference between the two, and the advantages of each, however, lies in the duration each table exists.

A **local temporary** table exists only for the duration of a connection or, if defined inside a compound statement, for the duration of the compound statement.

The definition of the **global temporary** table remains in the database permanently, but the rows exist only within a given connection. When you close the database connection, the data in the global temporary table disappears. However, the table definition remains with the database for you to access when you open your database next time.

Temporary tables are stored in the temporary file. Like any other dbspace, pages from the temporary file can be cached. Operations on temporary tables are never written to the transaction log.



CHAPTER 3

Ensuring Data Integrity

About this chapter

Building integrity constraints right into the database is the surest way to make sure your data stays in good shape. This chapter describes the facilities in Adaptive Server Anywhere for ensuring that the data in your database is valid and reliable.

You can enforce several types of integrity constraints. For example, you can ensure individual entries are correct by imposing constraints and CHECK conditions on tables and columns. Setting column properties by choosing an appropriate data type or setting special default values assists this task.

The SQL statements in this chapter use the CREATE TABLE and ALTER TABLE statements, basic forms of which were introduced in "Working with Database Objects" on page 27.

Contents

Topic	Page
Data integrity overview	66
Using column defaults	70
Using table and column constraints	76
Using domains	80
Enforcing entity and referential integrity	83
Integrity rules in the system tables	88

## Data integrity overview

If data has integrity, the data is valid—correct and accurate—and the relational structure of the database is intact. Referential integrity constraints enforce the relational structure of the database. These rules maintain the consistency of data between tables.

Adaptive Server Anywhere supports stored procedures, which give you detailed control over how data enters the database. You can also create triggers, or customized stored procedures invoked automatically when a certain action, such as an update of a particular column, occurs.

For more information on procedures and triggers see "Using Procedures, Triggers, and Batches" on page 507.

### How data can become invalid

Here are a few examples of how the data in a database may become invalid if proper checks are not made. You can prevent each of these examples from occurring using facilities described in this chapter.

Incorrect  
information

- ◆ An operator types the date of a sales transaction incorrectly.
- ◆ An employee's salary becomes ten times too small because the operator missed a digit.

Duplicated data

- ◆ Two different people add the same new department (with **dept\_id** 200) to the department table of the organization's database.

Foreign key  
relations  
invalidated

- ◆ The department identified by **dept\_id** 300 closes down and one employee record inadvertently remains unassigned to a new department.

### Integrity constraints belong in the database

To ensure the validity of data in a database, you need to formulate checks to define valid and invalid data, and design rules to which data must adhere (also known as business rules). Together, checks and rules become **constraints**.

Build integrity  
constraints into  
database

Constraints that are built into the database itself are more reliable than constraints that are built into client applications or that are spelled out as instructions to database users. Constraints built into the database become part of the definition of the database itself, and the database enforces them consistently across all applications. Setting a constraint once in the database imposes it for all subsequent interactions with the database.

In contrast, constraints built into client applications are vulnerable every time the software changes, and may need to be imposed in several applications, or in several places in a single client application.

## How database contents change

Changes occur to information in database tables when you submit SQL statements from client applications. Only a few SQL statements actually modify the information in a database. You can:

- ◆ Update information in a row of a table using the UPDATE statement.
- ◆ Delete an existing row of a table using the DELETE statement.
- ◆ Insert a new row into a table using the INSERT statement.


## Data integrity tools

To assist in maintaining data integrity, you can use defaults, data constraints, and constraints that maintain the referential structure of the database.

### Defaults

You can assign default values to columns to make certain kinds of data entry more reliable. For example:

- ◆ A column can have a current date default value for recording the date of transactions with any user or client application action.
- ◆ Other types of default values allow column values to increment automatically without any specific user action other than entering a new row. With this feature, you can guarantee that items (such as purchase orders for example) are unique, sequential numbers.

 For more information on these and other column defaults, see "Using column defaults" on page 70.

### Constraints

You can apply several types of constraints to the data in individual columns or tables. For example:

- ◆ A NOT NULL constraint prevents a column from containing a null entry.
- ◆ A CHECK condition assigned to a column can ensure that every item in the column meets a particular condition. For example, you can ensure that salary column entries fit within a specified range and thus protect against user error when typing in new values.

- ◆ CHECK conditions can be made on the relative values in different columns. For example, you can ensure that a *date\_returned* entry is later than a *date\_borrowed* entry in a library database.
- ◆ Triggers can enforce more sophisticated CHECK conditions. For more information on triggers, see "Using Procedures, Triggers, and Batches" on page 507.

As well, column constraints can be inherited from domains. For more information on these and other table and column constraints, see "Using table and column constraints" on page 76.

### Entity and referential integrity

Relationships, defined by the primary keys and foreign keys, tie together the information in relational database tables. You must build these relations directly into the database design. The following integrity rules maintain the structure of the database:

- ◆ **Entity integrity** Keeps track of the primary keys. It guarantees that every row of a given table can be uniquely identified by a primary key that guarantees IS NOT NULL.
- ◆ **Referential integrity** Keeps track of the foreign keys that define the relationships between tables. It guarantees that all foreign key values either match a value in the corresponding primary key or contain the NULL value if they are defined to allow NULL.

🔗 For more information about enforcing referential integrity, see "Enforcing entity and referential integrity" on page 83. For more information about designing appropriate primary and foreign key relations, see "Designing Your Database" on page 3.

### Triggers for advanced integrity rules

You can also use triggers to maintain data integrity. A **trigger** is a procedure stored in the database and executed automatically whenever the information in a specified table changes. Triggers are a powerful mechanism for database administrators and developers to ensure that data remains reliable.

🔗 For more information about triggers, see "Using Procedures, Triggers, and Batches" on page 507.

## SQL statements for implementing integrity constraints

The following SQL statements implement integrity constraints:

- ◆ **CREATE TABLE statement** This statement implements integrity constraints during creation of the database.
- ◆ **ALTER TABLE statement** This statement adds integrity constraints to an existing database, or modifies constraints for an existing database.

- ◆ **CREATE TRIGGER statement** This statement creates triggers that enforce more complex business rules.

✍ For more information about the syntax of these statements, see "SQL Statements" on page 199 of the book *ASA SQL Reference Manual*.

## Using column defaults

Column defaults automatically assign a specified value to particular columns whenever someone enters a new row into a database table. The default value assigned requires no any action on the part of the client application, however if the client application does specify a value for the column, the new value overrides the column default value.

Column defaults can quickly and automatically fill columns with information, such as the date or time a row is inserted, or the user ID of the person typing the information. Using column defaults encourages data integrity, but does not enforce it. Client applications can always override defaults.

### Supported default values

SQL supports the following default values:

- ◆ A string specified in the CREATE TABLE statement or ALTER TABLE statement
- ◆ A number specified in the CREATE TABLE statement or ALTER TABLE statement
- ◆ An automatically incremented number: one more than the previous highest value in the column
- ◆ Universally Unique Identifiers (UUIDs) and Globally Unique Identifiers (GUIDs) generated using the NEWID function.
- ◆ The current date, time, or timestamp
- ◆ The current user ID of the database user
- ◆ A NULL value
- ◆ A constant expression, as long as it does not reference database objects

## Creating column defaults

You can use the CREATE TABLE statement to create column defaults at the time a table is created, or the ALTER TABLE statement to add column defaults at a later time.

### Example

The following statement adds a condition to an existing column named **id** in the **sales\_order** table, so that it automatically increments (unless a client application specifies a value):

```
ALTER TABLE sales_order
MODIFY id DEFAULT AUTOINCREMENT
```



Each of the other default values is specified in a similar manner. For more information, see "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual* and "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

## Modifying and deleting column defaults

You can change or remove column defaults using the same form of the ALTER TABLE statement you used to create defaults. The following statement changes the default value of a column named **order\_date** from its current setting to CURRENT DATE:

```
ALTER TABLE sales_order
MODIFY order_date DEFAULT CURRENT DATE
```

You can remove column defaults by modifying them to be NULL. The following statement removes the default from the **order\_date** column:

```
ALTER TABLE sales_order
MODIFY order_date DEFAULT NULL
```

## Working with column defaults in Sybase Central

You can add, alter, and delete column defaults in Sybase Central using the Value tab of the column properties sheet.

### ❖ To display the property sheet for a column:

- 1 Connect to the database.
- 2 Open the Tables folder for that database.
- 3 Double-click the table holding the column you want to change.
- 4 Open the Columns folder for that table.
- 5 Right-click the column and choose Properties from the popup menu.

## Current date and time defaults

For columns with the DATE, TIME, or TIMESTAMP data type, you can use the current date, current time, or current timestamp as a default. The default you choose must be compatible with the column's data type.

Useful examples of current date default

A current date default might be useful to record:

- ◆ dates of phone calls in a contact database

- ◆ dates of orders in a sales entry database
- ◆ the date a patron borrows a book in a library database

### Current timestamp

The current timestamp is similar to the current date default, but offers greater accuracy. For example, a user of a contact management application may have several contacts with a single customer in one day: the current timestamp default would be useful to distinguish these contacts.

Since it records a date and the time down to a precision of millionths of a second, you may also find the current timestamp useful when the sequence of events is important in a database.

🔗 For more information about timestamps, times, and dates, see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*.

## The user ID default

Assigning a DEFAULT USER to a column is an easy and reliable way of identifying the person making an entry in a database. This information may be required; for example, when salespeople are working on commission.

Building a user ID default into the primary key of a table is a useful technique for occasionally connected users, and helps to prevent conflicts during information updates. These users can make a copy of tables relevant to their work on a portable computer, make changes while not connected to a multi-user database, and then apply the transaction log to the server when they return.

## The AUTOINCREMENT default

The AUTOINCREMENT default is useful for numeric data fields where the value of the number itself may have no meaning. The feature assigns each new row a value of one greater than the previous highest value in the column. You can use AUTOINCREMENT columns to record purchase order numbers, to identify customer service calls or other entries where an identifying number is required.

Autoincrement columns are typically primary key columns or columns constrained to hold unique values (see "Enforcing entity integrity" on page 83). For example, autoincrement default is effective when the column is the first column of an index, because the server uses an index or key definition to find the highest value.

While using the autoincrement default in other cases is possible, doing so can adversely affect database performance. For example, in cases where the next value for each column is stored as an integer (4 bytes), using values greater than  $2^{31} - 1$  or large double or numeric values may cause wraparound to negative values.

You can retrieve the most recent value inserted into an autoincrement column using the `@@identity` global variable. For more information, see "`@@identity` global variable" on page 46 of the book *ASA SQL Reference Manual*.

#### Autoincrement and negative numbers


Autoincrement is intended to work with positive integers.

The initial autoincrement value is set to 0 when the table is created. This value remains as the highest value assigned when inserts are done that explicitly insert negative values into the column. An insert where no value is supplied causes the AUTOINCREMENT to generate a value of 1, forcing any other generated values to be positive.

In UltraLite applications, the autoincrement value is not set to 0 when the table is created, and AUTOINCREMENT generates negative numbers when a signed data type is used for the column.

You should define AUTOINCREMENT columns as unsigned to prevent negative values from being used.

#### Autoincrement and the IDENTITY column

 A column with the AUTOINCREMENT default is referred to in Transact-SQL applications as an IDENTITY column. For information on IDENTITY columns, see "The special IDENTITY column" on page 399.

## The NEWID default

UUIDs (Universally Unique IDentifiers), also known as GUIDs (Globally Unique IDentifiers), can be used to uniquely identify rows in a table. The values are generated such that a value produced on one computer will not match that produced on another. They can therefore be used as keys in replication and synchronization environments.

Using UUID values as primary keys has some tradeoffs when you compare them with using GLOBAL AUTOINCREMENT values. For example,

- ◆ UUIDs can be easier to set up than GLOBAL AUTOINCREMENT, since there is no need to assign each remote database a unique database id. There is also no need to consider the number of databases in the system or the number of rows in individual tables. The Extraction utility can be used to deal with the assignment of database ids. This isn't usually a concern for GLOBAL AUTOINCREMENT if the BIGINT datatype is used, but it needs to be considered for smaller datatypes.

- ◆ UUID values are considerably larger than those required for GLOBAL AUTOINCREMENT, and will require more table space in both primary and foreign tables. Indexes on these columns will also be less efficient when UUIDs are used. In short, GLOBAL AUTOINCREMENT is likely to perform better.
- ◆ UUIDs have no implicit ordering. For example, if A and B are UUID values,  $A > B$  does not imply that A was generated after B, even when A and B were generated on the same computer. If you require this behavior, an additional column and index may be necessary.
- ◆ If UUID values are generated by an application (as opposed to the server), the values must be inserted as UUID strings and converted using `strtouuid()`. Attempting to insert binary values directly may result in values colliding with those generated by the server or another system.

🔗 For more information, see the "NEWID function " on page 159 of the book *ASA SQL Reference Manual*, the "STRTOUUID function " on page 185 of the book *ASA SQL Reference Manual*, the "UUIDTOSTR function " on page 193 of the book *ASA SQL Reference Manual*, or the "UNIQUEIDENTIFIER data type [Binary]" on page 73 of the book *ASA SQL Reference Manual*.

## The NULL default

For columns that allow NULL values, specifying a NULL default is exactly the same as not specifying a default at all. If the client inserting the row does not explicitly assign a value, the row automatically receives a NULL value.

You can use NULL defaults when information for some columns is optional or not always available, and when it is not required for the data in the database be correct.

🔗 For more information about the NULL value, see "NULL value" on page 48 of the book *ASA SQL Reference Manual*.

## String and number defaults

You can specify a specific string or number as a default value, as long as the column holds a string or number data type. You must ensure that the default specified can be converted to the column's data type.

Default strings and numbers are useful when there is a typical entry for a given column. For example, if an organization has two offices, the headquarters in **city\_1** and a small office in **city\_2**, you may want to set a default entry for a location column to **city\_1**, to make data entry easier.

## Constant expression defaults

You can use a constant expression as a default value, as long as it does not reference database objects. Constant expressions allow column defaults to contain entries such as *the date fifteen days from today*, which would be entered as

```
... DEFAULT ( dateadd( day, 15, getdate() ) )
```

## Using table and column constraints

Along with the basic table structure (number, name and data type of columns, name and location of the table), the CREATE TABLE statement and ALTER TABLE statement can specify many different table attributes that allow control over data integrity.

### **Caution**

*Altering tables can interfere with other users of the database. Although you can execute the ALTER TABLE statement while other connections are active, you cannot execute the ALTER TABLE statement if any other connection is using the table you want to alter. For large tables, ALTER TABLE is a time-consuming operation, and all other requests referencing the table being altered are prohibited while the statement is processing.*

This section describes how to use constraints to help ensure the accuracy of data in the table.

## Using CHECK conditions on columns

You use a CHECK condition to ensure that the values in a column satisfy some definite criterion or rule. For example, these rules or criteria may simply be required for data to be reasonable, or they may be more rigid rules that reflect organization policies and procedures.

CHECK conditions on individual column values are useful when only a restricted range of values are valid for that column.

### Example 1

- ◆ You can enforce a particular formatting requirement. For example, if a table has a column for phone numbers you may wish to ensure that users type them all in the same manner. For North American phone numbers, you could use a constraint such as:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '(____) ____-____' )
```

### Example 2

- ◆ You can ensure that the entry matches one of a limited number of values. For example, to ensure that a **city** column only contains one of a certain number of allowed cities (say, those cities where the organization has offices), you could use a constraint such as:

```
ALTER TABLE office
MODIFY city
CHECK ( city IN ( 'city_1', 'city_2', 'city_3' ) )
```

## Example 3

- ◆ By default, string comparisons are case insensitive unless the database is explicitly created as a case-sensitive database.
- ◆ You can ensure that a date or number falls in a particular range. For example, you may require that the **start\_date** column of an employee table must be between the date the organization was formed and the current date using the following constraint:
 

```
ALTER TABLE employee
MODIFY start_date
CHECK ( start_date BETWEEN '1983/06/27'
      AND CURRENT DATE )
```
- ◆ You can use several date formats. The YYYY/MM/DD format in this example has the virtue of always being recognized regardless of the current option settings.

Column CHECK tests only fail if the condition returns a value of FALSE. If the condition returns a value of UNKNOWN, the change is allowed.

## Column CHECK conditions from domains

You can attach CHECK conditions to domains. Columns defined on those data types inherit the CHECK conditions. A CHECK condition explicitly specified for the column overrides that from the domain.

Any column defined using the **posint** data type accepts only positive integers unless the column itself has a CHECK condition explicitly specified. In the following example, the domain accepts only positive integers. Since any variable prefixed with the @ sign is replaced by the name of the column when the CHECK condition is evaluated, any variable name prefixed with @ could be used instead of @col.

```
CREATE DATATYPE posint INT
CHECK ( @col > 0 )
```

An ALTER TABLE statement with the DELETE CHECK clause deletes all CHECK conditions from the table definition, including those inherited from domains.

For more information about domains, see "Domains" on page 75 of the book *ASA SQL Reference Manual*.

## Working with table and column constraints in Sybase Central

In Sybase Central, you add, alter, and delete column constraints on the Constraints tab of the table or column property sheet.

❖ **To manage constraints on a table:**

- 1 Open the Tables folder.
- 2 Right-click the table and choose Properties from the popup menu.
- 3 Click the Constraints tab.
- 4 Make the appropriate changes.

❖ **To manage constraints on a column:**

- 1 Open the Tables folder and double-click a table to open it.
- 2 Open the Columns folder.
- 3 Right-click a column and choose Properties from the popup menu.
- 4 Click the Constraints tab.
- 5 Make the appropriate changes.

## Using CHECK conditions on tables

A CHECK condition applied as a constraint on the table typically ensures that two values in a row being added or modified have a proper relation to each other. Column CHECK conditions, in contrast, are held individually in the system tables, and you can replace or delete them individually. Since this is more flexible behavior, use CHECK conditions on individual columns where possible.

For example, in a library database, the **date\_borrowed** must come before the **date\_returned**.

```
ALTER TABLE loan
ADD CHECK(date_returned >= date_borrowed)
```

## Modifying and deleting CHECK conditions

There are several ways to alter the existing set of CHECK conditions on a table.

- ◆ You can add a new CHECK condition to the table or to an individual column, as described above.
- ◆ You can delete a CHECK condition on a column by setting it to NULL. For example, the following statement removes the CHECK condition on the **phone** column in the **customer** table:



```
ALTER TABLE customer
MODIFY phone CHECK NULL
```

- ◆ You can replace a CHECK condition on a column in the same way as you would add a CHECK condition. For example, the following statement adds or replaces a CHECK condition on the **phone** column of the **customer** table:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '____-____-____' )
```

- ◆ There are two ways of modifying a CHECK condition defined on the table, as opposed to a CHECK condition defined on a column:
  - ◆ You can add a new CHECK condition using ALTER TABLE with an ADD table-constraint clause.
  - ◆ You can delete all existing CHECK conditions (including column CHECK conditions and CHECK conditions inherited from domains) using ALTER TABLE DELETE CHECK, and then add in new CHECK conditions.

To use the ALTER TABLE statement with the DELETE CHECK clause:

```
ALTER TABLE table_name
DELETE CHECK
```

Deleting a column from a table does not delete CHECK conditions associated with the column held in the table constraint. Not removing the constraints produces a column not found error message upon any attempt to insert, or even just query, data in the table.


Table CHECK conditions fail only if a value of FALSE is returned. A value of UNKNOWN allows the change.

## Using domains

A **domain** is a user-defined data type that, together with other attributes, can restrict the range of acceptable values or provide defaults. A domain extends one of the built-in data types. The range of permissible values is usually restricted by a check constraint. In addition, a domain can specify a default value and may or may not allow nulls.

You can define your own domains for a number of reasons.

- ◆ A number of common errors can be prevented if inappropriate values cannot be entered. A constraint placed on a domain ensures that all columns and variables intended to hold values in a desired range or format can hold only the intended values. For example, a data type can ensure that all credit card numbers typed into the database contain the correct number of digits.
- ◆ Domains can make it much easier to understand applications and the structure of a database.
- ◆ Domains can prove convenient. For example, you may intend that all table identifiers are positive integers that, by default, auto-increment. You could enforce this restriction by entering the appropriate constraints and defaults each time you define a new table, but it is less work to define a new domain, then simply state that the identifier can take only values from the specified domain.

 For more information about domains, see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*.

## Creating domains (Sybase Central)

You can use Sybase Central to create a domain or assign it to a column.

### ❖ To create a new domain (Sybase Central):

- 1 Open the Domains folder.
- 2 In the right pane, double-click Add Domain.
- 3 Follow the instructions in the wizard.

All domains appear in the Domains folder in Sybase Central.

### ❖ To assign domains to columns (Sybase Central):

- 1 For the desired table, open the Columns folder.

- 2 Right-click the desired column and choose Properties from the popup menu.
- 3 On the Data Type tab of the column's property sheet, assign a domain.

## Creating domains (SQL)

You can use the CREATE DOMAIN statement to create and define domains.

### ❖ To create a new domain (SQL):

- 1 Connect to a database.
- 2 Execute a CREATE DOMAIN statement.

#### Example 1: Simple domains

Some columns in the database are to be used for people's names and others are to store addresses. You might then define type following domains.

```
CREATE DOMAIN persons_name CHAR(30)
CREATE DOMAIN street_address CHAR(35)
```

Having defined these domains, you can use them much as you would the built-in data types. For example, you can use these definitions to define a tables as follows.

```
CREATE TABLE customer (
    id INT DEFAULT AUTOINCREMENT PRIMARY KEY
    name persons_name
    address street_address
)
```

#### Example 2: Default values, check constraints, and identifiers

In the above example, the table's primary key is specified to be of type integer. Indeed, many of your tables may require similar identifiers. Instead of specifying that these are integers, it is much more convenient to create an identifier domain for use in these applications.

When you create a domain, you can specify a default value and provide check constraint to ensure that no inappropriate values are typed into any column of this type.

Integer values are commonly used as table identifiers. A good choice for unique identifiers is to use positive integers. Since such identifiers are likely to be used in many tables, you could define the following domain.

```
CREATE DOMAIN identifier INT
DEFAULT AUTOINCREMENT
CHECK ( @col > 0 )
```

This check constraint uses the variable @col. Using this definition, you can rewrite the definition of the customer table, shown above.

```
CREATE TABLE customer (  
    id identifier PRIMARY KEY  
    name persons_name  
    address street_address  
)
```

### Example 3: Built-in domains

Adaptive Server Anywhere comes with some domains pre-defined. You can use these pre-defined domains as you would a domain that you created yourself. For example, the following monetary domain has already been created for you.

```
CREATE DOMAIN MONEY NUMERIC(19,4)  
NULL
```

🔗 For more information, see "CREATE DOMAIN statement" on page 283 of the book *ASA SQL Reference Manual*.

## Deleting domains

You can use either Sybase Central or a DROP DOMAIN statement to delete a domain.

Only the user DBA or the user who created a domain can drop it. In addition, since a domain cannot be dropped if any variable or column in the database is an instance of the domain, you need to first drop any columns or variables of that type before you can drop the domain.

### ❖ To delete a domain (Sybase Central):

- 1 Open the Domains folder.
- 2 Right-click the desired domain and choose Delete from the popup menu.

### ❖ To delete a domain (SQL):

- 1 Connect to a database.
- 2 Execute a DROP DOMAIN statement.

### Example

The following statement drops the *customer\_name* domain.

```
DROP DOMAIN customer_name
```

🔗 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Enforcing entity and referential integrity

The relational structure of the database enables the personal server to identify information within the database, and ensures that all the rows in each table uphold the relationships between tables (described in the database structure).

### Enforcing entity integrity

When a user inserts or updates a row, the database server ensures that the primary key for the table is still valid: that each row in the table is uniquely identified by the primary key.

#### Example 1

The **employee** table in the sample database uses an employee ID as the primary key. When you add a new employee to the table, the database server checks that the new employee ID value is unique and is not NULL.

#### Example 2

The **sales\_order\_items** table in the sample database uses two columns to define a primary key.

This table holds information about items ordered. One column contains an **id** specifying an order, but there may be several items on each order, so this column by itself cannot be a primary key. An additional **line\_id** column identifies which line corresponds to the item. The columns **id** and **line\_id**, taken together, specify an item uniquely, and form the primary key.

### If a client application breaches entity integrity

Entity integrity requires that each value of a primary key be unique within the table, and that no NULL values exist. If a client application attempts to insert or update a primary key value, providing values that are not unique would breach entity integrity. A breach in entity integrity prevents the new information from being added to the database, and instead sends the client application an error.

The application programmer should decide how to present this information to the user and enable the user to take appropriate action. The appropriate action is usually as simple as asking the user to provide a different, unique value for the primary key.

## Primary keys enforce entity integrity

Once you specify the primary key for each table, maintaining entity integrity requires no further action by either client application developers or by the database administrator.

The table owner defines the primary key for a table when they create it. If they modify the structure of a table at a later date, they can also redefine the primary key.

Some application development systems and database design tools allow you to create and alter database tables. If you are using such a system, you may not have to enter the CREATE TABLE or ALTER TABLE statement explicitly: the application may generate the statement itself from the information you provide.

☞ For more information about creating primary keys, see "Managing primary keys" on page 44. For the detailed syntax of the CREATE TABLE statement, see "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*. For information about changing table structure, see the "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*.

## Enforcing referential integrity

A foreign key (made up of a particular column or combination of columns) relates the information in one table (the **foreign** table) to information in another (**referenced** or **primary**) table. For the foreign key relationship to be valid, the entries in the foreign key must correspond to the primary key values of a row in the referenced table. Occasionally, some other unique column combination may be referenced instead of a primary key.

### Example 1

The sample database contains an employee table and a department table. The primary key for the employee table is the employee ID, and the primary key for the department table is the department ID. In the employee table, the department ID is called a **foreign key** for the department table because each department ID in the employee table corresponds exactly to a department ID in the department table.


The foreign key relationship is a many-to-one relationship. Several entries in the employee table have the same department ID entry, but the department ID is the primary key for the department table, and so is unique. If a foreign key could reference a column in the department table containing duplicate entries, or entries with a NULL value, there would be no way of knowing which row in the department table is the appropriate reference. This is a mandatory foreign key.

**Example 2**

Suppose the database also contained an office table listing office locations. The employee table might have a foreign key for the office table that indicates which city the employee's office is in. The database designer can choose to leave an office location unassigned at the time the employee is hired, for example, either because they haven't been assigned to an office yet, or because they don't work out of an office. In this case, the foreign key can allow NULL values, and is optional.

## Foreign keys enforce referential integrity

Like primary keys, you use the CREATE TABLE or ALTER TABLE statements to create foreign keys. Once you create a foreign key, the column or columns in the key can contain only values that are present as primary key values in the table associated with the foreign key.

 For more information about creating foreign keys, see "Managing primary keys" on page 44.

## Losing referential integrity

Your database can lose referential integrity if someone:

- ◆ updates or deletes a primary key value. All the foreign keys referencing that primary key would become invalid.
- ◆ adds a new row to the foreign table, and enters a value for the foreign key that has no corresponding primary key value. The database would become invalid.

Adaptive Server Anywhere provides protection against both types of integrity loss.

## If a client application breaches referential integrity

If a client application updates or deletes a primary key value in a table, and if a foreign key references that primary key value elsewhere in the database, there is a danger of a breach of referential integrity.

**Example**

If the server allowed the primary key to be updated or deleted, and made no alteration to the foreign keys that referenced it, the foreign key reference would be invalid. Any attempt to use the foreign key reference, for example in a SELECT statement using a KEY JOIN clause, would fail, as no corresponding value in the referenced table exists.

While Adaptive Server Anywhere handles breaches of entity integrity in a generally straightforward fashion by simply refusing to enter the data and returning an error message, potential breaches of referential integrity become more complicated. You have several options (known as referential integrity actions) available to help you maintain referential integrity.

## Referential integrity actions

Maintaining referential integrity when updating or deleting a referenced primary key can be as simple as disallowing the update or delete. Often, however, it is also possible to take a specific action on each foreign key to maintain referential integrity. The `CREATE TABLE` and `ALTER TABLE` statements allow database administrators and table owners to specify what action to take on foreign keys that reference a modified primary key when a breach occurs.

You can specify each of the available referential integrity actions separately for updates and deletes of the primary key:

- ◆ **RESTRICT** Generates an error and prevents the modification if an attempt to modify a referenced primary key value occurs. This is the default referential integrity action.
- ◆ **SET NULL** Sets all foreign keys that reference the modified primary key to `NULL`.
- ◆ **SET DEFAULT** Sets all foreign keys that reference the modified primary key to the default value for that column (as specified in the table definition).
- ◆ **CASCADE** When used with `ON UPDATE`, this action updates all foreign keys that reference the updated primary key to the new value. When used with `ON DELETE`, this action deletes all rows containing foreign keys that reference the deleted primary key.

System triggers implement referential integrity actions. The trigger, defined on the primary table, is executed using the permissions of the owner of the *secondary* table. This behavior means that cascaded operations can take place between tables with different owners, without additional permissions having to be granted.



## Referential integrity checking

For foreign keys defined to RESTRICT operations that would violate referential integrity, default checks occur at the time a statement executes. If you specify a CHECK ON COMMIT clause, then the checks occur only when the transaction is committed.

Using a database option to control check time

Setting the WAIT\_FOR\_COMMIT database option controls the behavior when a foreign key is defined to restrict operations that would violate referential integrity. The CHECK ON COMMIT clause can override this option.

With the default WAIT\_FOR\_COMMIT set to OFF, operations that would leave the database inconsistent cannot execute. For example, an attempt to DELETE a department that still has employees in it is not allowed. The following statement gives the error primary key for row in table 'department' is referenced in another table:

```
DELETE FROM department
WHERE dept_id = 200
```

Setting WAIT\_FOR\_COMMIT to ON causes referential integrity to remain unchecked until a commit executes. If the database is in an inconsistent state, the database disallows the commit and reports an error. In this mode, a database user could delete a department with employees in it, however, the user cannot commit the change to the database until they:

- ◆ Delete or reassign the employees belonging to that department.
- ◆ Redo the search condition on a SELECT statement to select the rows that violate referential integrity.
- ◆ Insert the **dept\_id** row back into the **department** table.
- ◆ Roll back the transaction to undo the DELETE operation.

## Integrity rules in the system tables

All the information about database integrity checks and rules is held in the following system tables:

System table	Description
<i>SYS.SYSTABLE</i>	The <b>view_def</b> column of SYS.SYSTABLE holds CHECK constraints. For views, the <b>view_def</b> holds the CREATE VIEW command that created the view. You can check whether a particular table is a base table or a view by looking at the <b>table_type</b> column, which is BASE or VIEW.
<i>SYS.SYSTRIGGER</i>	<p>SYS.SYSTRIGGER holds referential integrity actions.</p> <p>The <b>referential_action</b> column holds a single character indicating whether the action is cascade (C), delete (D), set null (N), or restrict (R).</p> <p>The <b>event</b> column holds a single character specifying the event that causes the action to occur: insert, delete (A), insert, update (B), update (C), a delete (D), delete, update (E), insert (I), update (U), insert, delete, update (M).</p> <p>The <b>trigger_time</b> column shows whether the action occurs after (A) or before (B) the triggering event.</p>
<i>SYS.SYSFOREIGNKEYS</i>	This view presents the foreign key information from the two tables SYS.SYSFOREIGNKEY and SYS.SYSFKCOL in a more readable format.
<i>SYS.SYSCOLUMNS</i>	This view presents the information from the SYS.SYSCOLUMN table in a more readable format. It includes default settings and primary key information for columns.

🔗 For more information about the contents of each system table, see "System Tables" on page 595 of the book *ASA SQL Reference Manual*. You can use Sybase Central or Interactive SQL to browse these tables and views.

C H A P T E R 4

Using Transactions and Isolation Levels

About this chapter      You can group SQL statements into transactions, which have the property that either all statements are executed or none is executed. You should design each transaction to perform a task that changes your database from one consistent state to another.

                                 This chapter describes transactions and how to use them in applications. It also describes how Adaptive Server Anywhere you can set isolation levels to limit the interference among concurrent transaction.

Contents

Topic	Page
Introduction to transactions	90
Isolation levels and consistency	94
Transaction blocking and deadlock	100
Choosing isolation levels	102
Isolation level tutorials	106
How locking works	121
Particular concurrency issues	135
Replication and concurrency	138
Summary	140

# Introduction to transactions

To ensure data integrity, it is essential that you can identify states in which the information in your database is **consistent**. The concept of consistency is best illustrated through an example:

## Consistency example

Suppose you use your database to handle financial accounts, and you wish to transfer money from one client's account to another. The database is in a consistent state both before and after the money is transferred; but it is not in a consistent state after you have debited money from one account and before you have credited it to the second. During a transfer of money, the database is in a consistent state when the total amount of money in the clients' accounts is as it was before any money was transferred. When the money has been half transferred, the database is in an inconsistent state. Either both or neither of the debit and the credit must be processed.

## Transactions are logical units of work

A **transaction** is a logical unit of work. Each transaction is a sequence of logically related commands that accomplish one task and transform the database from one consistent state into another. The nature of a consistent state depends on your database.

The statements within a transaction are treated as an indivisible unit: either all are executed or none is executed. At the end of each transaction, you **commit** your changes to make them permanent. If for any reason some of the commands in the transaction do not process properly, then any intermediate changes are undone, or **rolled back**. Another way of saying this is that transactions are **atomic**.

Grouping statements into transactions is key both to protecting the consistency of your data (even in the event of media or system failure), and to managing concurrent database operations. Transactions may be safely interleaved and the completion of each transaction marks a point at which the information in the database is consistent.

In the event of a system failure or database crash during normal operation, Adaptive Server Anywhere performs automatic recovery of your data when the database is next started. The automatic recovery process recovers all completed transactions, and rolls back any transactions that were uncommitted when the failure occurred. The atomic character of transactions ensures that databases are recovered to a consistent state.

🔗 For more information about database backups and data recovery, see "Backup and Data Recovery" on page 299 of the book *ASA Database Administration Guide*.

🔗 For more information about concurrent database usage, see "Introduction to concurrency" on page 92.

## Using transactions

Adaptive Server Anywhere expects you to group your commands into transactions. Knowing which commands or actions signify the start or end of a transaction lets you take full advantage of this feature.

### Starting transactions

Transactions start with one of the following events:

- ◆ The first statement following a connection to a database
- ◆ The first statement following the end of a transaction

### Completing transactions

Transactions complete with one of the following events:

- ◆ A COMMIT statement makes the changes to the database permanent.
- ◆ A ROLLBACK statement undoes all the changes made by the transaction.
- ◆ A statement with a side effect of an automatic commit is executed: data definition commands, such as ALTER, CREATE, COMMENT, and DROP all have the side effect of an automatic commit.
- ◆ A disconnection from a database performs an implicit rollback.
- ◆ ODBC and JDBC have an autocommit setting that enforces a COMMIT after each statement. By default, ODBC and JDBC require autocommit to be on, and each statement is a single transaction. If you want to take advantage of transaction design possibilities, then you should turn autocommit off.

🔗 For more information on autocommit, see "Setting autocommit or manual commit mode" on page 44 of the book *ASA Programming Guide*.

- ◆ Setting the database option CHAINED to OFF is similar to enforcing an autocommit after each statement. By default, connections that use jConnect or Open Client applications have CHAINED set to OFF.

🔗 For more information, see "Setting autocommit or manual commit mode" on page 44 of the book *ASA Programming Guide*, and "CHAINED option" on page 557 of the book *ASA Database Administration Guide*.

### Options in Interactive SQL

Interactive SQL lets you control when and how transactions from your application terminate:

- ◆ If you set the option AUTO\_COMMIT to ON, Interactive SQL automatically commits your results following every successful statement and automatically perform a ROLLBACK after each failed statement.

- ◆ The setting of the option `COMMIT_ON_EXIT` controls what happens to uncommitted changes when you exit Interactive SQL. If this option is set to ON (the default), Interactive SQL does a COMMIT; otherwise it undoes your uncommitted changes with a ROLLBACK statement.

🔗 Adaptive Server Anywhere also supports Transact-SQL commands such as `BEGIN TRANSACTION`, for compatibility with Sybase Adaptive Server Enterprise. For further information, see "Transact-SQL Compatibility" on page 383.

## Introduction to concurrency

**Concurrency** is the ability of the database server to process multiple transactions at the same time. Were it not for special mechanisms within the database server, concurrent transactions could interfere with each other to produce inconsistent and incorrect information.

### Example

A database system in a department store must allow many clerks to update customer accounts concurrently. Each clerk must be able to update the status of the accounts as they assist each customer: they cannot afford to wait until no one else is using the database.

### Who needs to know about concurrency

Concurrency is a concern to all database administrators and developers. Even if you are working with a single-user database, you must be concerned with concurrency if you wish to process instructions from multiple applications or even from multiple connections from a single application. These applications and connections can interfere with each other in exactly the same way as multiple users in a network setting.

### Transaction size affects concurrency

The way you group SQL statements into transactions can have significant effects on data integrity and on system performance. If you make a transaction too short and it does not contain an entire logical unit of work, then inconsistencies can be introduced into the database. If you write a transaction that is too long and contains several unrelated actions, then there is greater chance that a ROLLBACK will unnecessarily undo work that could have been committed quite safely into the database.

If your transactions are long, they can lower concurrency by preventing other transactions from being processed concurrently.

There are many factors that determine the appropriate length of a transaction, depending on the type of application and the environment.

## Savepoints within transactions

You may identify important states within a transaction and return to them selectively using **savepoints** to separate groups of related statements.

A `SAVEPOINT` statement defines an intermediate point during a transaction. You can undo all changes after that point using a `ROLLBACK TO SAVEPOINT` statement. Once a `RELEASE SAVEPOINT` statement has been executed or the transaction has ended, you can no longer use the savepoint.

No locks are released by the `RELEASE SAVEPOINT` or `ROLLBACK TO SAVEPOINT` commands: locks are released only at the end of a transaction.

### Naming and nesting savepoints

Using named, nested savepoints, you can have many active savepoints within a transaction. Changes between a `SAVEPOINT` and a `RELEASE SAVEPOINT` can be canceled by rolling back to a previous savepoint or rolling back the transaction itself. Changes within a transaction are not a permanent part of the database until the transaction is committed. All savepoints are released when a transaction ends.

Savepoints cannot be used in bulk operations mode. There is very little additional overhead in using savepoints.

## Isolation levels and consistency

There are four isolation levels

Adaptive Server Anywhere allows you to control the degree to which the operations in one transaction are visible to the operations in other concurrent transactions. You do so by setting a database option called the **isolation level**. Adaptive Server Anywhere has four different isolation levels (numbered 0 through 3) that prevent some or all inconsistent behavior. Level 3 provides the highest level of isolation. Lower levels allow more inconsistencies, but typically have better performance. Level 0 is the default setting.

All isolation levels guarantee that each transaction will execute completely or not at all, and that no updates will be lost.

### Typical types of inconsistency

There are three typical types of inconsistency that can occur during the execution of concurrent transactions. This list is not exhaustive as other types of inconsistencies can also occur. These three types are mentioned in the ISO SQL/92 standard and are important because behavior at lower isolation levels is defined in terms of them.

- ◆ **Dirty read** Transaction A modifies a row, but does not commit or roll back the change. Transaction B reads the modified row. Transaction A then either further changes the row before performing a COMMIT, or rolls back its modification. In either case, transaction B has seen the row in a state which was never committed.

☞ For more information about how isolation levels create dirty reads, see "Dirty read tutorial" on page 106.

- ◆ **Non-repeatable read** Transaction A reads a row. Transaction B then modifies or deletes the row and performs a COMMIT. If transaction A then attempts to read the same row again, the row will have been changed or deleted.

☞ For more information about non-repeatable reads, see "Non-repeatable read tutorial" on page 109.

- ◆ **Phantom row** Transaction A reads a set of rows that satisfy some condition. Transaction B then executes an INSERT, or an UPDATE on a row which did not previously meet A's condition. Transaction B commits these changes. These newly committed rows now satisfy the condition. Transaction A then repeats the initial read and obtains a different set of rows.



For more information about phantom rows, see "Phantom row tutorial" on page 113.

Other types of inconsistencies can also exist. These three were chosen for the ISO SQL/92 standard because they are typical problems and because it was convenient to describe amounts of locking between transactions in terms of them.

Isolation levels and dirty reads, non-repeatable reads, and phantom rows

The isolation levels are different with respect to the type of inconsistent behavior that Adaptive Server Anywhere allows. An **x** means that the behavior is prevented, and a **✓** means that the behavior may occur.

Isolation level	Dirty reads	Non-repeatable reads	Phantom rows
0	✓	✓	✓
1	<b>x</b>	✓	✓
2	<b>x</b>	<b>x</b>	✓
3	<b>x</b>	<b>x</b>	<b>x</b>

This table demonstrates two points:

- ◆ Each isolation level eliminates one of the three typical types of inconsistencies.
- ◆ Each level eliminates the types of inconsistencies eliminated at all lower levels.

The four isolation levels have different names under ODBC. These names are based on the names of the inconsistencies that they prevent, and are described in "The ValuePtr parameter" on page 98.

## Cursor instability

Another significant inconsistency is **cursor instability**. When this inconsistency is present, a transaction can modify a row that is being referenced by another transaction's cursor. Cursor stability ensures that applications using cursors do not introduce inconsistencies into the data in the database.

### Example

Transaction A reads a row using a cursor. Transaction B modifies that row. Not realizing that the row has been modified, Transaction A modifies it, rendering the affected row's data incorrect.

### Eliminating cursor instability

Adaptive Server Anywhere achieves **cursor stability** at isolation levels 1, 2, and 3. Cursor stability ensures that no other transactions can modify information that is contained in the present row of your cursor. The information in a row of a cursor may be the copy of information contained in a particular table or may be a combination of data from different rows of multiple tables. More than one table will likely be involved whenever you use a join or sub-selection within a SELECT statement.

🔗 For information on programming SQL procedures and cursors, see "Using Procedures, Triggers, and Batches" on page 507.

🔗 Cursors are used only when you are using Adaptive Server Anywhere through another application. For more information, see "Using SQL in Applications" on page 9 of the book *ASA Programming Guide*.

A related but distinct concern for applications using cursors is whether changes to underlying data are visible to the application. You can control the changes that are visible to applications by specifying the sensitivity of the cursor.

🔗 For more information about cursor sensitivity, see "Adaptive Server Anywhere cursors" on page 28 of the book *ASA Programming Guide*.

## Setting the isolation level

Each connection to the database has its own isolation level. In addition, the database can store a default isolation level for each user or group. The PUBLIC setting enables you to set a single default isolation level for the entire database's group.

The isolation level is a database option. You change database options using the SET OPTION statement. For example, the following command sets the isolation level for the current user to 3, the highest level.

```
SET OPTION ISOLATION_LEVEL = 3
```

You can change the isolation of your connection and the default level associated with your user ID using the SET OPTION command. If you have permission, you can also change the isolation level for other users or groups.

### ❖ To set the isolation level for the current user ID:

- ◆ Execute the SET OPTION statement. For example, the following statement sets the isolation level to 3 for the current user:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
```

❖ **To set the isolation level for a user or group:**

- 1 Connect to the database as a user with DBA authority.
- 2 Execute the SET OPTION statement, adding the name of the group and a period before ISOLATION\_LEVEL. For example, the following command sets the default isolation for the special group PUBLIC to 3.

```
SET OPTION PUBLIC.ISOLATION_LEVEL = 3
```

❖ **To set the isolation level just for your present session:**

- ◆ Execute the SET OPTION statement using the TEMPORARY keyword. For example, the following statement sets the isolation level to 3 for the duration of your connection:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3
```

Once you disconnect, your isolation level reverts to its previous value.

**Default isolation level**

When you connect to a database, the database server determines your initial isolation level as follows:

- 1 A default isolation level may be set for each user and group. If a level is stored in the database for your user ID, then the database server uses it.
- 2 If not, the database server checks the groups to which you belong until it finds a level. All users are members of the special group PUBLIC. If it finds no other setting first, then Adaptive Server Anywhere will use the level assigned to that group.

🔗 For more information about users and groups, see "Managing User IDs and Permissions" on page 351 of the book *ASA Database Administration Guide*.

🔗 For more information about the SET OPTION statement syntax, see "SET OPTION statement" on page 539 of the book *ASA SQL Reference Manual*.

🔗 You may wish to change the isolation level in mid-transaction if, for example, just one table or group of tables requires serialized access. For information about changing the isolation level within a transaction, see "Changing isolation levels within a transaction" on page 98.

## Setting the isolation level from an ODBC-enabled application

ODBC applications call **SQLSetConnectAttr** with **Attribute** set to **SQL\_ATTR\_TXN\_ISOLATION** and **ValuePtr** set according to the corresponding isolation level:

The ValuePtr parameter

ValuePtr	Isolation Level
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXT_SERIALIZABLE	3

Changing an isolation level via ODBC

You can change the isolation level of your connection via ODBC using the function **SQLSetConnectOption** in the library *ODBC32.dll*.  
  
The **SQLSetConnectOption** function reads three parameters: the value of the ODBC connection handle, the fact that you wish to set the isolation level, and the value corresponding to the isolation level. These values appear in the table below.

String	Value
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXT_SERIALIZABLE	8

Example

The following function call sets the isolation level of the connection MyConnection to isolation level 2:

```
SQLSetConnectOption(MyConnection.hDbc, SQL_TXN_ISOLATION, SQL_TXN_REPEATABLE_READ)
```

ODBC uses the isolation feature to support assorted database lock options. For example, in PowerBuilder you can use the Lock attribute of the transaction object to set the isolation level when you connect to the database. The Lock attribute is a string, and is set as follows:

```
SQLCA.lock = "RU"
```

The Lock option is honored only at the moment the CONNECT occurs. Changes to the Lock attribute after the CONNECT have no effect on the connection.

Changing isolation levels within a transaction


Sometimes you will find that different isolation levels are suitable for different parts of a single transaction. Adaptive Server Anywhere allows you to change the isolation level of your database in the middle of a transaction.

When you change the `ISOLATION_LEVEL` option in the middle of a transaction, the new setting affects only the following:

- ◆ Any cursors opened after the change
- ◆ Any statements executed after the change

You may wish to change the isolation level during a transaction, as doing so affords you control over the number of locks your transaction places. You may find a transaction needs to read a large table, but perform detailed work with only a few of the rows. If an inconsistency would not seriously affect your transaction, set the isolation to a low level while you scan the large table to avoid delaying the work of others.

You may also wish to change the isolation level in mid transaction if, for example, just one table or group of tables requires serialized access.

 For an example in which the isolation level is changed in the middle of a transaction, see "Phantom row tutorial" on page 113.

## Viewing the isolation level

You can inspect the isolation level of the current connection using the `CONNECTION_PROPERTY` function.

### ❖ To view the isolation level for the current connection:

- ◆ Execute the following statement:

```
SELECT CONNECTION_PROPERTY( ' ISOLATION_LEVEL' )
```

# Transaction blocking and deadlock

When a transaction is being executed, the database server places locks on rows to prevent other transactions from interfering with the affected rows.

**Locks** control the amount and types of interference permitted.

Adaptive Server Anywhere uses **transaction blocking** to allow transactions to execute concurrently without interference, or with limited interference. Any transaction can acquire a lock to prevent other concurrent transactions from modifying or even accessing a particular row. This transaction blocking scheme always stops some types of interference. For example, a transaction that is updating a particular row of a table always acquires a lock on that row to ensure that no other transaction can update or delete the same row at the same time.

## Transaction blocking

When a transaction attempts to carry out an operation, but is forbidden by a lock held by another transaction, a conflict arises and the progress of the transaction attempting to carry out the operation is impeded or blocked.

☞ "Two-phase locking" on page 131 describes deadlock, which occurs when two or more transactions are blocked by each other in such a way that none can proceed.

☞ Sometimes a set of transactions arrive at a state where none of them can proceed. For more information, see "Deadlock" on page 101.

## The BLOCKING option

If two transactions have each acquired a read lock on a single row, the behavior when one of them attempts to modify that row depends on the database setting **BLOCKING**. To modify the row, that transaction must block the other, yet it cannot do so while the other transaction has it blocked.

- ◆ If **BLOCKING** is **ON** (the default), then the transaction that attempts to write waits until the other transaction releases its read lock. At that time, the write goes through.
- ◆ If **BLOCKING** has been set to **OFF**, then the transaction that attempts to write receives an error.

When **BLOCKING** is **OFF**, the transaction terminates instead of waiting and any changes it has made are rolled back. In this event, try executing the transaction again, later.

Blocking is more likely to occur at higher isolation levels because more locking and more checking is done. Higher isolation levels usually provide less concurrency. How much less depends on the individual natures of the concurrent transactions.

🔗 For more information about the BLOCKING option, see "BLOCKING option" on page 556 of the book *ASA Database Administration Guide*.

## Deadlock

Transaction blocking can lead to **deadlock**, a situation in which a set of transactions arrive at a state where none of them can proceed.

### Reasons for deadlocks

A deadlock can arise for two reasons:

- ◆ **A cyclical blocking conflict** Transaction A is blocked on transaction B, and transaction B is blocked on transaction A. Clearly, more time will not solve the problem, and one of the transactions must be canceled, allowing the other to proceed. The same situation can arise with more than two transactions blocked in a cycle.
- ◆ **All active database threads are blocked** When a transaction becomes blocked, its database thread is not relinquished. If the database is configured with three threads and transactions A, B, and C are blocked on transaction D which is not currently executing a request, then a deadlock situation has arisen since there are no available threads.

Adaptive Server Anywhere automatically cancels the last transaction that became blocked (eliminating the deadlock situation), and returns an error to that transaction indicating which form of deadlock occurred.

🔗 The number of database threads that the server uses depends on the individual database's setting. For information about setting the number of database threads, see "Controlling threading behavior" on page 15 of the book *ASA Database Administration Guide*.

### Determining who is blocked

You can use the `sa_conn_info` system procedure to determine which connections are blocked on which other connections. This procedure returns a result set consisting of a row for each connection. One column of the result set lists whether the connection is blocked, and if so which other connection it is blocked on.

🔗 For more information, see "sa\_conn\_info system procedure" on page 688 of the book *ASA SQL Reference Manual*.

## Choosing isolation levels

The choice of isolation level depends on the kind of task an application is carrying out. This section gives some guidelines for choosing isolation levels.

When you choose an appropriate isolation level you must balance the need for consistency and accuracy with the need for concurrent transactions to proceed unimpeded. If a transaction involves only one or two specific values in one table, it is unlikely to interfere as much with other processes as one which searches many large tables and may need to lock many rows or entire tables and may take a very long time to complete.

For example, if your transactions involve transferring money between bank accounts or even checking account balances, you will likely want to do your utmost to ensure that the information you return is correct. On the other hand, if you just want a rough estimate of the proportion of inactive accounts, then you may not care whether your transaction waits for others or not and indeed may be willing to sacrifice some accuracy to avoid interfering with other users of the database.

Furthermore, a transfer may affect only the two rows which contain the two account balances, whereas all the accounts must be read in order to calculate the estimate. For this reason, the transfer is less likely to delay other transactions.

Adaptive Server Anywhere provides four levels of isolation: levels 0, 1, 2, and 3. Level 3 provides complete isolation and ensures that transactions are interleaved in such a manner that the schedule is serializable.

## Serializable schedules

To process transactions concurrently, the database server must execute some component statements of one transaction, then some from other transactions, before continuing to process further operations from the first. The order in which the component operations of the various transactions are interleaved is called the **schedule**.

Applying transactions concurrently in this manner can result in many possible outcomes, including the three particular inconsistencies described in the previous section. Sometimes, the final state of the database also could have been achieved had the transactions been executed sequentially, meaning that one transaction was always completed in its entirety before the next was started. A schedule is called **serializable** whenever executing the transactions sequentially, in some order, could have left the database in the same state as the actual schedule.



☞ For more information about how Adaptive Server Anywhere handles serialization, see "Two-phase locking" on page 131.

Serializable is the commonly accepted criterion for correctness. A serializable schedule is accepted as correct because the database is not influenced by the concurrent execution of the transactions.

The isolation level affects a transaction's serializability. At isolation level 3, all schedules are serializable. The default setting is 0.

Serializable means that concurrency has added no effect

Even when transactions are executed sequentially, the final state of the database can depend upon the order in which these transactions are executed. For example, if one transaction sets a particular cell to the value 5 and another sets it to the number 6, then the final value of the cell is determined by which transaction executes last.

Knowing a schedule is serializable does not settle which order transactions would best be executed, but rather states that concurrency has added no effect. Outcomes which may be achieved by executing the set of transactions sequentially in some order are all assumed correct.

Unserializable schedules introduce inconsistencies

The inconsistencies introduced in "Typical types of inconsistency" on page 94 are typical of the types of problems that appear when the schedule is not serializable. In each case, the inconsistency appeared because the statements were interleaved in such a way as to produce a result that would not be possible if all transactions were executed sequentially. For example, a dirty read can only occur if one transaction can select rows while another transaction is in the middle of inserting or updating data in the same row.

## Typical transactions at various isolation levels

Various isolation levels lend themselves to particular types of tasks. Use the information below to help you decide which level is best suited to each particular operation.

Typical level 0 transactions

Transactions that involve browsing or performing data entry may last several minutes, and read a large number of rows. If isolation level 2 or 3 is used, concurrency can suffer. Isolation level of 0 or 1 is typically used for this kind of transaction.

For example, a decision support application that reads large amounts of information from the database to produce statistical summaries may not be significantly affected if it reads a few rows that are later modified. If high isolation is required for such an application, it may acquire read locks on large amounts of data, not allowing other applications write access to it.

### Typical level 1 transactions

Isolation level 1 is particularly useful in conjunction with cursors, because this combination ensures cursor stability without greatly increasing locking requirements. Adaptive Server Anywhere achieves this benefit through the early release of read locks acquired for the present row of a cursor. These locks must persist until the end of the transaction at either levels two or three in order to guarantee repeatable reads.

For example, a transaction that updates inventory levels through a cursor is particularly suited to this level, because each of the adjustments to inventory levels as items are received and sold would not be lost, yet these frequent adjustments would have minimal impact on other transactions.

### Typical level 2 transactions

At isolation level 2, rows that match your criterion cannot be changed by other transactions. You can thus employ this level when you must read rows more than once and rely that rows contained in your first result set won't change.

Because of the relatively large number of read locks required, you should use this isolation level with care. As with level 3 transactions, careful design of your database and indexes reduce the number of locks acquired and hence can improve the performance of your database significantly.

### Typical level 3 transactions

Isolation level 3 is appropriate for transactions that demand the most in security. The elimination of phantom rows lets you perform multi-step operations on a set of rows without fear that new rows will appear partway through your operations and corrupt the result.

However much integrity it provides, isolation level 3 should be used sparingly on large systems that are required to support a large number of concurrent transactions. Adaptive Server Anywhere places more locks at this level than at any other, raising the likelihood that one transaction will impede the process of many others.

## Improving concurrency at isolation levels 2 and 3

Isolation levels 2 and 3 use a lot of locks and so good design is of particular importance for databases that make regular use of these isolation levels.

When you must make use of serializable transactions, it is important that you design your database, in particular the indices, with the business rules of your project in mind. You may also improve performance by breaking large transactions into several smaller ones, thus shortening the length of time that rows are locked.

Although serializable transactions have the most potential to block other transactions, they are not necessarily less efficient. When processing these transactions, Adaptive Server Anywhere can perform certain optimizations that may improve performance, in spite of the increased number of locks. For example, since all rows read must be locked whether or not they match the search criteria, the database server is free to combine the operation of reading rows and placing locks.


## Reducing the impact of locking


You should avoid running transactions at isolation level 3 whenever practical. They tend to place large number of locks and hence impact the efficient execution of other concurrent transactions.

When the nature of an operation demands that it run at isolation level 3, you can lower its impact on concurrency by designing the query to read as few rows and index entries as possible. These steps will help the level 3 transaction run more quickly and, of possibly greater importance, will reduce the number of locks it places.

In particular, you may find that adding an index may greatly help speed up transactions, particularly when at least one of them must execute at isolation level 3. An index can have two benefits:

- ◆ An index enables rows to be located in an efficient manner
- ◆ Searches that make use of the index may need fewer locks.

 For more information about the details of the locking methods employed by Adaptive Server Anywhere is located in "How locking works" on page 121.

 For more information on performance and how Adaptive Server Anywhere plans its access of information to execute your commands, see "Monitoring and Improving Performance" on page 143.

## Isolation level tutorials

The different isolation levels behave in very different ways, and which one you will want to use depends on your database and on the operations you are carrying out. The following set of tutorials will help you determine which isolation levels are suitable for different tasks.

### Dirty read tutorial

The following tutorial demonstrates one type of inconsistency that can occur when multiple transactions are executed concurrently. Two employees at a small merchandising company access the corporate database at the same time. The first person is the company's Sales Manager. The second is the Accountant.

The Sales Manager wants to increase the price of tee shirts sold by their firm by \$0.95, but is having a little trouble with the syntax of the SQL language. At the same time, unknown to the Sales Manager, the Accountant is trying to calculate the retail value of the current inventory to include in a report he volunteered to bring to the next management meeting.

**Tip:**

Before altering your database in the following way, it is prudent to test the change by using SELECT in place of UPDATE.

In this example, you will play the role of two people, both using the demonstration database concurrently.

- 1 Start Interactive SQL.
- 2 Connect to the sample database as the Sales Manager:
  - ◆ In the Connect dialog, choose the ODBC data source *ASA 8.0 Sample*.
  - ◆ On the Advanced tab, type the following string to make the window easier to identify:  

```
ConnectionName=Sales Manager
```
  - ◆ Click OK to connect.
- 3 Start a second instance of Interactive SQL.
- 4 Connect to the sample database as the Accountant:
  - ◆ In the Connect dialog, choose the ODBC data source *ASA 8.0 Sample*.

- ◆ On the Advanced tab, type the following string to make the window easier to identify:

```
ConnectionName=Accountant
```

- ◆ Click OK to connect.

5 As the Sales Manager, raise the price of all the tee shirts by \$0.95:

- ◆ In the window labeled Sales Manager, execute the following commands:

```
SELECT id, name, unit_price
FROM product;

UPDATE PRODUCT
SET unit_price = unit_price + 95
WHERE NAME = 'Tee Shirt'
```

The result is:

id	name	unit_price
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
...	...	...

You observe immediately that you should have entered 0.95 instead of 95, but before you can fix your error, the Accountant accesses the database from another office.

6 The company's Accountant is worried that too much money is tied up in inventory. As the Accountant, execute the following commands to calculate the total retail value of all the merchandise in stock:

```
SELECT SUM( quantity * unit_price )
AS inventory
FROM product
```

The result is:

inventory
21453.00

Unfortunately, this calculation is not accurate. The Sales Manager accidentally raised the price of the visor \$95, and the result reflects this erroneous price. This mistake demonstrates one typical type of inconsistency known as a **dirty read**. You, as the Accountant, accessed data which the Sales Manager has entered, but has not yet committed.

☞ You can eliminate dirty reads and other inconsistencies explained in "Isolation levels and consistency" on page 94.

- 7 As the Sales Manager, fix the error by rolling back your first changes and entering the correct UPDATE command. Check that your new values are correct.

```
ROLLBACK;

UPDATE product
SET unit_price = unit_price + 0.95
WHERE NAME = 'Tee Shirt';
```

id	name	unit_price
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
...	...	...

- 8 The Accountant does not know that the amount he calculated was in error. You can see the correct value by executing his SELECT statement again in his window.

```
SELECT SUM( quantity * unit_price )
AS inventory
FROM product;
```

inventory
6687.15

- 9 Finish the transaction in the Sales Manager's window. She would enter a COMMIT statement to make his changes permanent, but you may wish to enter a ROLLBACK, instead, to avoid changing the copy of the demonstration database on your machine.

```
ROLLBACK;
```

The Accountant unknowingly receives erroneous information from the database because the database server is processing the work of both the Sales Manager and the Accountant concurrently.

## Non-repeatable read tutorial

The example in "Dirty read tutorial" on page 106 demonstrated the first type of inconsistency, namely the dirty read. In that example, an Accountant made a calculation while the Sales Manager was in the process of updating a price. The Accountant's calculation used erroneous information which the Sales Manager had entered and was in the process of fixing.

The following example demonstrates another type of inconsistency: non-repeatable reads. In this example, you will play the role of the same two people, both using the demonstration database concurrently. The Sales Manager wishes to offer a new sales price on plastic visors. The Accountant wishes to verify the prices of some items that appear on a recent order.

This example begins with both connections at isolation level 1, rather than at isolation level 0, which is the default for the demonstration database supplied with Adaptive Server Anywhere. By setting the isolation level to 1, you eliminate the type of inconsistency which the previous tutorial demonstrated, namely the dirty read.

- 1 Start Interactive SQL.
- 2 Connect to the sample database as the Sales Manager:
  - ◆ In the Connect dialog, choose the ODBC data source *ASA 8.0 Sample*.
  - ◆ On the Advanced tab, enter the following string to make the window easier to identify:  

```
ConnectionName=Sales Manager
```
  - ◆ Click OK to connect.
- 3 Start a second instance of Interactive SQL.
- 4 Connect to the sample database as the Accountant:
  - ◆ In the Connect dialog, choose the ODBC data source *ASA 8.0 Sample*.
  - ◆ On the Advanced tab, enter the following string to make the window easier to identify:  

```
ConnectionName=Accountant
```
  - ◆ Click OK to connect.
- 5 Set the isolation level to 1 for the Accountant's connection by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1
```

- 6 Set the isolation level to 1 in the Sales Manager's window by executing the following command:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 1
```

- 7 The Accountant decides to list the prices of the visors. As the Accountant, execute the following command:

```
SELECT id, name, unit_price FROM product
```

id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
...	...	...

- 8 The Sales Manager decides to introduce a new sale price for the plastic visor. As the Sales Manager, execute the following command:

```
SELECT id, name, unit_price FROM product  
WHERE name = 'Visor';
```

```
UPDATE product  
SET unit_price = 5.95 WHERE id = 501;  
COMMIT;
```

id	name	unit_price
500	Visor	7.00
501	Visor	5.95

- 9 Compare the price of the visor in the Sales Manager window with the price for the same visor in the Accountant window. The Accountant window still displays the old price, even though the Sales Manager has entered the new price and committed the change.

This inconsistency is called a **non-repeatable read**, because if the Accountant did the same SELECT a second time in the *same transaction*, he wouldn't get the same results. Try it for yourself. As the Accountant, execute the select command again. Observe that the Sales Manager's sale price now displays.

```
SELECT id, name, price  
FROM product
```



id	name	unit_price
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
...	...	...

Of course if the Accountant had finished his transaction, for example by issuing a COMMIT or ROLLBACK command before using SELECT again, it would be a different matter. The database is available for simultaneous use by multiple users and it is completely permissible for someone to change values either before or after the Accountant's transaction. The change in results is only inconsistent because it happens in the middle of his transaction. Such an event makes the schedule unserializable.

- 10 The Accountant notices this behavior and decides that from now on he doesn't want the prices changing while he looks at them. Repeatable reads are eliminated at isolation level 2. Play the role of the Accountant:

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;

SELECT id, name, unit_price
FROM product
```

- 11 The Sales Manager decides that it would be better to delay the sale on the plastic visor until next week so that she won't have to give the lower price on a big order that she's expecting will arrive tomorrow. In her window, try to execute the following statements. The command will start to execute, and then his window will appear to freeze.

```
UPDATE product
SET unit_price = 7.00
WHERE id = 501
```

The database server must guarantee repeatable reads at isolation level 2. To do so, it places a read lock on each row of the product table that the Accountant reads. When the Sales Manager tries to change the price back, her transaction must acquire a write lock on the plastic visor row of the product table. Since write locks are exclusive, her transaction must wait until the Accountant's transaction releases its read lock.

- 12 The Accountant is finished looking at the prices. He doesn't want to risk accidentally changing the database, so he completes his transaction with a ROLLBACK statement.

```
ROLLBACK
```

Observe that as soon as the database server executes this statement, the Sales Manager's transaction completes.

id	name	unit_price
500	Visor	7.00
501	Visor	7.00

- 13 The Sales Manager can finish now. She wishes to commit her change to restore the original price.

COMMIT

### Types of Locks and different isolation levels

When you upgraded the Accountant's isolation from level 1 to level 2, the database server used read locks where none had previously been acquired. In general, each isolation level is characterized by the types of locks needed and by how locks held by other transactions are treated.

At isolation level 0, the database server needs only write locks. It makes use of these locks to ensure that no two transactions make modifications that conflict. For example, a level 0 transaction acquires a write lock on a row before it updates or deletes it, and inserts any new rows with a write lock already in place.

Level 0 transactions perform no checks on the rows they are reading. For example, when a level 0 transaction reads a row, it doesn't bother to check what locks may or may not have been acquired on that row by other transactions. Since no checks are needed, level 0 transactions are particularly fast. This speed comes at the expense of consistency. Whenever they read a row which is write locked by another transaction, they risk returning dirty data.

At level 1, transactions check for write locks before they read a row. Although one more operation is required, these transactions are assured that all the data they read is committed. Try repeating the first tutorial with the isolation level set to 1 instead of 0. You will find that the Accountant's computation cannot proceed while the Sales Manager's transaction, which updates the price of the tee shirts, remains incomplete.


When the Accountant raised his isolation to level 2, the database server began using read locks. From then on, it acquired a read lock for his transaction on each row that matched his selection.

## Transaction blocking

In the above tutorial, the Sales Manager window froze during the execution of her UPDATE command. The database server began to execute her command, then found that the Accountant's transaction had acquired a read lock on the row that the Sales Manager needed to change. At this point, the database server simply paused the execution of the UPDATE. Once the Accountant finished his transaction with the ROLLBACK, the database server automatically released his locks. Finding no further obstructions, it then proceeded to complete execution of the Sales Manager's UPDATE.

In general, a locking conflict occurs when one transaction attempts to acquire an exclusive lock on a row on which another transaction holds a lock, or attempts to acquire a shared lock on a row on which another transaction holds an exclusive lock. One transaction must wait for another transaction to complete. The transaction that must wait is said to be **blocked** by another transaction.

When the database server identifies a locking conflict which prohibits a transaction from proceeding immediately, it can either pause execution of the transaction, or it can terminate the transaction, roll back any changes, and return an error. You control the route by setting the BLOCKING option. When BLOCKING is ON the second transaction waits, as in the above tutorial.

 For more information about the blocking option, see "The BLOCKING option" on page 100.

## Phantom row tutorial

The following tutorial continues the same scenario. In this case, the Accountant views the department table while the Sales Manager creates a new department. You will observe the appearance of a phantom row.

If you have not done so, do steps 1 through 4 of the previous tutorial, "Non-repeatable read tutorial" on page 109, so that you have two instances of Interactive SQL.

- 1 Set the isolation level to 2 in the Sales Manager window by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

- 2 Set the isolation level to 2 for the Accountant window by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2;
```

- 3 In the Accountant window, enter the following command to list all the departments.

```
SELECT * FROM department
ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
...	...	...

- 4 The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has emp\_id 129, will head the new department.

```
INSERT INTO department
(dept_id, dept_name, dept_head_id)
VALUES(600, 'Foreign Sales', 129);
```

The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

- 5 The Accountant, however, is not aware of the new department. At isolation level 2, the database server places locks to ensure that no row changes, but places no locks that stop other transactions from inserting new rows.

The Accountant will only discover the new row if he executes his SELECT command again. In the Accountant's window, execute the SELECT statement again. You will see the new row appended to the table.

```
SELECT *
FROM department
ORDER BY dept_id;
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
500	Shipping	703
...	...	...

The new row that appears is called a **phantom row** because, from the Accountant's point of view, it appears like an apparition, seemingly from nowhere. The Accountant is connected at isolation level 2. At that level, the database server acquires locks only on the rows that he is using. Other rows are left untouched and hence there is nothing to prevent the Sales Manager from inserting a new row.

- 6 The Accountant would prefer to avoid such surprises in future, so he raises the isolation level of his current transaction to level 3. Enter the following commands for the Accountant.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3

SELECT *
FROM department
ORDER BY dept_id
```

- 7 The Sales Manager would like to add a second department to handle sales initiative aimed at large corporate partners. Execute the following command in the Sales Manager's window.

```
INSERT INTO department
(dept_id, dept_name, dept_head_id)
VALUES(700, 'Major Account Sales', 902)
```

The Sales Manager's window will pause during execution because the Accountant's locks block the command. Click the *Interrupt the SQL Statement* button on the toolbar (or choose Stop from the SQL menu) to interrupt this entry.

- 8 To avoid changing the demonstration database that comes with Adaptive Server Anywhere, you should roll back the insertion of the new departments. Execute the following command in the Sales Manager's window:

```
ROLLBACK
```

When the Accountant raised his isolation to level 3 and again selected all rows in the department table, the database server placed anti-insert locks on each row in the table, and one extra phantom lock to avoid insertion at the end of the table. When the Sales Manager attempted to insert a new row at the end of the table, it was this final lock that blocked her command.

Notice that the Sales Manager's command was blocked even though the Sales Manager is still connected at isolation level 2. The database server places anti-insert locks, like read locks, as demanded by the isolation level and statements of each transactions. Once placed, these locks must be respected by all other concurrent transactions.

☞ For more information on locking, see "How locking works" on page 121.

## Practical locking implications tutorial

The following continues the same scenario. In this tutorial, the Accountant and the Sales Manager both have tasks that involve the sales order and sales order items tables. The Accountant needs to verify the amounts of the commission checks paid to the sales employees for the sales they made during the month of April 2001. The Sales Manager notices that a few orders have not been added to the database and wants to add them.

Their work demonstrates phantom locking. A **phantom lock** is a shared lock placed on an indexed scan position to prevent phantom rows. When a transaction at isolation level 3 selects rows which match a given criterion, the database server places anti-insert locks to stop other transactions from inserting rows which would also match. The number of locks placed on your behalf depends both on the search criterion and on the design of your database.

If you have not done so, do steps 1 through 3 of the previous tutorial which describe how to start two instances of Interactive SQL.

- 1 Set the isolation level to 2 in both the Sales Manager window and the Accountant window by executing the following command.

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 2
```

- 2 Each month, the sales representatives are paid a commission, which is calculated as a percentage of their sales for that month. The Accountant is preparing the commission checks for the month of April 2001. His first task is to calculate the total sales of each representative during this month.

Enter the following command in the Accountant's window. Prices, sales order information, and employee data are stored in separate tables. Join these tables using the foreign key relationships to combine the necessary pieces of information.

```
SELECT emp_id, emp_fname, emp_lname,  
       SUM(sales_order_items.quantity * unit_price)  
       AS "April sales"  
FROM employee  
     KEY JOIN sales_order  
     KEY JOIN sales_order_items  
     KEY JOIN product  
WHERE '2001-04-01' <= order_date  
     AND order_date < '2001-05-01'  
GROUP BY emp_id, emp_fname, emp_lname
```

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

- 3 The Sales Manager notices that a big order sold by Philip Chin was not entered into the database. Philip likes to be paid his commission promptly, so the Sales manager enters the missing order, which was placed on April 25.

In the Sales Manager's window, enter the following commands. The Sales order and the items are entered in separate tables because one order can contain many items. You should create the entry for the sales order before you add items to it. To maintain referential integrity, the database server allows a transaction to add items to an order only if that order already exists.

```
INSERT into sales_order
VALUES ( 2653, 174, '2001-04-22', 'r1',
        'Central', 129);

INSERT into sales_order_items
VALUES ( 2653, 1, 601, 100, '2001-04-25' );

COMMIT;
```

- 4 The Accountant has no way of knowing that the Sales Manager has just added a new order. Had the new order been entered earlier, it would have been included in the calculation of Philip Chin's April sales.

In the Accountant's window, calculate the April sales totals again. Use the same command, and observe that Philip Chin's April sales changes to \$4560.00.

emp_id	emp_fname	emp_lname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

Imagine that the Accountant now marks all orders placed in April to indicate that commission has been paid. The order that the Sales Manager just entered might be found in the second search and marked as paid, even though it was not included in Philip's total April sales!

- 5 At isolation level 3, the database server places anti-insert locks to ensure that no other transactions can add a row which matches the criterion of a search or select.

First, roll back the insertion of Philip's missing order: Execute the following statement in the Sales Manager's window.

```
ROLLBACK
```

- 6 In the Accountant's window, execute the following two statements.

```
ROLLBACK;
```

```
SET TEMPORARY OPTION ISOLATION_LEVEL = 3;
```

- 7 In the Sales Manager's window, execute the following statements to remove the new order.

```
DELETE
FROM sales_order_items
WHERE id = 2653;
```

```
DELETE
FROM sales_order
WHERE id = 2653;
```

```
COMMIT;
```

- 8 In the Accountant's window, execute same query as before.

```
SELECT emp_id, emp_fname, emp_lname,
       SUM(sales_order_items.quantity * unit_price)
       AS "April sales"
FROM employee
     KEY JOIN sales_order
     KEY JOIN sales_order_items
     KEY JOIN product
WHERE '2001-04-01' <= order_date
     AND order_date < '2001-05-01'
GROUP BY emp_id, emp_fname, emp_lname
```

Because you set the isolation to level 3, the database server will automatically place anti-insert locks to ensure that the Sales Manager can't insert April order items until the Accountant finishes his transaction.

- 9 Return to the Sales Manager's window. Again attempt to enter Philip Chin's missing order.



```
INSERT INTO sales_order
VALUES ( 2653, 174, '2001-04-22',
        'r1', 'Central', 129)
```

The Sales Manager's window will hang; the operation will not complete. Click the *Interrupt the SQL Statement* button on the toolbar (or choose Stop from the SQL menu) to interrupt this entry.

- 10 The Sales Manager can't enter the order in April, but you might think that she could still enter it in May.

Change the date of the command to May 05 and try again.

```
INSERT INTO sales_order
VALUES ( 2653, 174, '2001-05-05', 'r1',
        'Central', 129)
```

The Sales Manager's window will hang again. Click the *Interrupt the SQL Statement* button on the toolbar (or choose Stop from the SQL menu) to interrupt this entry. Although the database server places no more locks than necessary to prevent insertions, these locks have the potential to interfere with a large number of other transactions.

The database server places locks in table indices. For example, it places a phantom lock in an index so a new row cannot be inserted immediately before it. However, when no suitable index is present, it must lock every row in the table.

In some situations, anti-insert locks may block some insertions into a table, yet allow others.

- 11 The Sales Manager wishes to add a second item to order 2651. Use the following command.

```
INSERT INTO sales_order_items
VALUES ( 2651, 2, 302, 4, '2001-05-22' )
```

All goes well, so the Sales Manager decides to add the following item to order 2652 as well.

```
INSERT INTO sales_order_items
VALUES ( 2652, 2, 600, 12, '2001-05-25' )
```

The Sales Manager's window will hang. Click the *Interrupt the SQL Statement* button on the toolbar (or choose Stop from the SQL menu) to interrupt this entry.

- 12 Conclude this tutorial by undoing any changes to avoid changing the demonstration database. Enter the following command in the Sales Manager's window.

```
ROLLBACK
```

Enter the same command in the Accountant's window.

ROLLBACK

You may now close both windows.

## How locking works

When the database server processes a transaction, it can lock one or more rows of a table. The locks maintain the reliability of information stored in the database by preventing concurrent access by other transactions. They also improve the accuracy of result queries by identifying information which is in the process of being updated.

The database server places these locks automatically and needs no explicit instruction. It holds all the locks acquired by a transaction until the transaction is completed, for example by either a COMMIT or ROLLBACK statement, with a single exception noted in "Early release of read locks—an exception" on page 132.

The transaction that has access to the row is said to hold the lock. Depending on the type of lock, other transactions may have limited access to the locked row, or none at all.

Obtaining  
information about  
locks on a table

You can use the *sa\_locks* system procedure to list information about locks that are held in the database. For more information, see "sa\_locks system procedure" on page 698 of the book *ASA SQL Reference Manual*.

You can also view locks in Sybase Central. Open the Tables folder for the database, and a tab called Locks appears in the right pane. For each lock, this tab shows you the connection ID, user ID, table name, lock type and lock name.

## Objects that can be locked

Adaptive Server Anywhere places locks on the following objects.

- ◆ **Rows in tables** A transaction can lock a particular row to prevent another transaction from changing it. A transaction must place a write lock on a row if it intends to modify the row.
- ◆ **Insertion points between rows** Transactions typically scan rows using the ordering imposed by an index, or scan rows sequentially. In either case, a lock can be placed on the scan position. For example, placing a lock in an index can prevent another transaction from inserting a row with a specific value or range of values.
- ◆ **Table schemas** A transaction can lock the schema of a table, preventing other transactions from modifying the table's structure.

Of these objects, rows are likely the most intuitive. It is understandable that a transaction reading, updating, deleting, or inserting a row should limit the simultaneous access to other transactions. Similarly, a transaction changing the structure of a table, perhaps inserting a new column, could greatly impact other transactions. In such a case, it is essential to limit the access of other transactions to prevent errors.

### Row orderings

You can use an index to order rows based on a particular criterion established when the index was constructed.

When there is no index, Adaptive Server Anywhere orders rows by their physical placement on disk. In the case of a sequential scan, the specific ordering is defined by the internal workings of the database server. You should not rely on the order of rows in a sequential scan. From the point of view of scanning the rows, however, Adaptive Server Anywhere treats the request similarly to an indexed scan, albeit using an ordering of its own choosing. It can place locks on positions in the scan as it would were it using an index.

Through locking a scan position, a transaction prevents some actions by other transactions relating to a particular range of values in that ordering of the rows. Insert and anti-insert locks are always placed on scan positions.

For example, a transaction might delete a row, hence deleting a particular primary key value. Until this transaction either commits the change or rolls it back, it must protect its right to do either. In the case of a deleted row, it must ensure that no other transaction can insert a row using the same primary key value, hence making a rollback operation impossible. A lock on the scan position this row occupied reserves this right while having the least impact on other transactions.

## Types of locks

Adaptive Server Anywhere uses four distinct types of locks to implement its locking scheme and ensure appropriate levels of isolation between transactions:

- ◆ **read lock** (shared)
- ◆ **phantom lock or anti-insert lock** (shared)
- ◆ **write lock** (exclusive)
- ◆ **anti-phantom lock or insert lock** (shared)

Remember that the database server places these locks automatically and needs no explicit instruction.

Each of these locks has a separate purpose, and they all work together. Each prevents a particular set of inconsistencies that could occur in their absence. Depending on the isolation level you select, the database server will use some or all of them to maintain the degree of consistency you require.

The above types of locks have the following uses:

- ◆ A transaction acquires a **write lock** whenever it inserts, updates, or deletes a row. No other transaction can obtain either a read or a write lock on the same row when a write lock is set. A write lock is an exclusive lock.
- ◆ A transaction can acquire a **read lock** when it reads a row. Several transactions can acquire read locks on the same row (a read lock is a shared or nonexclusive lock). Once a row has been read locked, no other transaction can obtain a write lock on it. Thus, a transaction can ensure that no other transaction modifies or deletes a row by acquiring a read lock.
- ◆ An **anti-insert lock**, or **phantom lock**, is a shared lock placed on an indexed scan position to prevent phantom rows. It prevents other transactions from inserting a row into a table immediately before the row which is anti-insert locked. Anti-insert locks for lookups using indexes require a read lock on each row that is read, and one extra read lock to prevent insertions into the index at the end of the result set. Phantom rows for lookups that do not use indexes require a read lock on all rows in a table to prevent insertions from altering the result set, and so can have a bad effect on concurrency.
- ◆ An **insert lock**, or **anti-phantom lock**, is a shared lock placed on an indexed scan position to reserve the right to insert a row. Once one transaction acquires an insert lock on a row, no other transaction can acquire an anti-insert lock on the same row. A read lock on the corresponding row is always acquired at the same time as an insert lock to ensure that no other process can update or destroy the row, thereby bypassing the insert lock.

Adaptive Server Anywhere uses these four types of locks as necessary to ensure the level of consistency that you require. You do not need to explicitly request the use of a particular lock. Instead, you control the level of consistency, as is explained in the next section. Knowledge of the types of locks will guide you in choosing isolation levels and understanding the impact of each level on performance.

Exclusive versus  
shared locks

These four types of locks each fall into one of two categories:

- ◆ **Exclusive locks** Only one transaction can hold an exclusive lock on a row of a table at one time. No transaction can obtain an exclusive lock while any other transaction holds a lock of any type on the same row. Once a transaction acquires an exclusive lock, requests to lock the row by other transactions will be denied.  
  
Write locks are exclusive.
- ◆ **Shared locks** Any number of transactions may acquire shared locks on any one row at the same time. Shared locks are sometimes referred to as non-exclusive locks.

Read locks, insert locks, and anti-insert locks are shared.

Only one transaction should change any one row at one time. Otherwise, two simultaneous transactions might try to change one value to two different new ones. Hence, it is important that a write lock be exclusive.

By contrast, no difficulty arises if more than one transaction wants to read a row. Since neither is changing it, there is no conflict of interest. Hence, read locks may be shared.

You may apply similar reasoning to anti-insert and insert locks. Many transactions can prevent the insertion of a row in a particular scan position by each acquiring an anti-insert lock. Similar logic applies for insert locks. When a particular transaction requires exclusive access, it can easily achieve exclusive access by obtaining both an anti-insert and an insert lock on the same row. These locks do not conflict when they are held by the same transaction.

Which specific locks conflict?

The following table identifies the combination of locks that conflict.

	read	write	anti-insert	insert
read		conflict		
write	conflict	conflict		
anti-insert				conflict
insert			conflict	

These conflicts arise only when the locks are held by different transactions. For example, one transaction can obtain both anti-insert and insert locks on a single scan position to obtain exclusive access to a location.

## Locking during queries

The locks that Adaptive Server Anywhere uses when a user enters a SELECT statement depend on the transaction's isolation level.

SELECT  
statements at  
isolation level 0

No locking operations are required when executing a SELECT statement at isolation level 0. Each transaction is not protected from changes introduced by other transactions. It is the responsibility of the programmer or database user to interpret the result of these queries with this limitation in mind.

SELECT  
statements at  
isolation level 1

You may be surprised to learn that Adaptive Server Anywhere uses almost no more locks when running a transaction at isolation level 1 than it does at isolation level 0. Indeed, the database server modifies its operation in only two ways.

The first difference in operation has nothing to do with acquiring locks, but rather with respecting them. At isolation level 0, a transaction is free to read any row, whether or not another transaction has acquired a write lock on it. By contrast, before reading each row an isolation level 1 transaction must check whether a write lock is in place. It cannot read past any write-locked rows because doing so might entail reading dirty data.

The second difference in operation creates cursor stability. Cursor stability is achieved by acquiring a read lock on the current row of a cursor. This read lock is released when the cursor is moved. More than one row may be affected if the contents of the cursor is the result of a join. In this case, the database server acquires read locks on all rows which have contributed information to the cursor's current row and removes all these locks as soon as another row of the cursor is selected as current. A read lock placed to ensure cursor stability is the only type of lock that does not persist until the end of a transaction.

SELECT  
statements at  
isolation level 2

At isolation level 2, Adaptive Server Anywhere modifies its procedures to ensure that your reads are repeatable. If your SELECT command returns values from every row in a table, then the database server acquires a read lock on each row of the table as it reads it. If, instead, your SELECT contains a WHERE clause, or another condition which restricts the rows to selected, then the database server instead reads each row, tests the values in the row against your criterion, and then acquires a read lock on the row if it meets your criterion.

As at all isolation levels, the locks acquired at level 2 include all those set at levels 1 and 0. Thus, cursor stability is again ensured and dirty reads are not permitted.

SELECT  
statements at  
isolation level 3

When operating at isolation level 3, Adaptive Server Anywhere is obligated to ensure that all schedules are serializable. In particular, in addition to the requirements imposed at each of the lower levels, it must eliminate phantom rows.

To accommodate this requirement, the database server uses read locks and anti-insert locks. When you make a selection, the database server acquires a read lock on each row that contributes information to your result set. Doing so ensures that no other transactions can modify that material before you have finished using it.

This requirement is similar to the procedures that the database server uses at isolation level 2, but differs in that a lock must be acquired for each row read, *whether or not it meets any attached criteria*. For example, if you select the names of all employees in the sales department, then the server must lock all the rows which contain information about a sales person, whether the transaction is executing at isolation level 2 or 3. At isolation level 3, however, it must also acquire read locks on each of the rows of employees which are *not* in the sales department. Otherwise, someone else accessing the database could potentially transfer another employee to the sales department while you were still using your results.

The fact that a read lock must be acquired on each row whether or not it meets your criteria has two important implications.

- ◆ The database server may need to place many more locks than would be necessary at isolation level 2.
- ◆ The database server can operate a little more efficiently: It can immediately acquire a read lock on each row as it reads it, since the locks must be placed whether or not the information in the row is accepted.


The number of anti-insert locks the server places can vary greatly and depends upon your criteria and on the indexes available in the table. Suppose you select information about the employee with Employee ID 123. If the employee ID is the primary key of the employee table, then the database server can economize its operations. It can use the index, which is automatically built for a primary key, to locate the row efficiently. In addition, there is no danger that another transaction could change another Employee's ID to 123 because primary key values must be unique. The server can guarantee that no second employee is assigned that ID number simply by acquiring a read lock on only the one row containing information about the employee with that number.

By contrast, the database server would acquire more locks were you instead to select all the employees in the sales department. Since any number of employees could be added to the department, the server will likely have to read every row in the employee table and test whether each person is in sales. If this is the case, both read and anti-insert locks must be acquired for each row.



## Locking during inserts

INSERT operations create new rows. Adaptive Server Anywhere employs the following procedure to ensure data integrity.

 For more information about how locks are used during inserts, see "Anti-insert locks" on page 130.

- 1 Make a location in memory to store the new row. The location is initially hidden from the rest of the database, so there is as yet no concern that another transaction could access it.
- 2 Fill the new row with any supplied values.
- 3 Write lock the new row.
- 4 Place an insert lock in the table to which the row is being added. Recall that insert locks are exclusive, so once the insert lock is acquired, no other transaction can block the insertion by acquiring an anti-insert lock.
- 5 Insert the row into the table. Other transactions can now, for the first time, see that the new row exists. They can't modify or delete it, though, because of the write lock acquired earlier.
- 6 Update all affected indexes and verify both referential integrity and uniqueness, where appropriate. Verifying referential integrity means ensuring that no foreign key points to a primary key that does not exist. Primary key values must be unique. Other columns may also be defined to contain only unique values, and if any such columns exist, uniqueness is verified.
- 7 The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file and release all locks.
- 8 Insert other rows as required, if you have selected the cascade option, and fire triggers.

### Uniqueness

You can ensure that all values in a particular column, or combination of columns, are unique. The database server always performs this task by building an index for the unique column, even if you do not explicitly create one.

In particular, all primary key values must be unique. The database server automatically builds an index for the primary key of every table. Thus, you should not ask the database server to create an index on a primary key, as that index would be a redundant index.

### Orphans and referential integrity

A foreign key is a reference to a primary key, usually in another table. When that primary key doesn't exist, the offending foreign key is called an **orphan**. Adaptive Server Anywhere automatically ensures that your database contains no orphans. This process is referred to as **verifying referential integrity**. The database server verifies referential integrity by counting orphans.

### WAIT FOR COMMIT

You can ask the database server to delay verifying referential integrity to the end of your transaction. In this mode, you can insert one row which contains a foreign key, then insert a second row which contains the missing primary key. You must perform both operations in the same transaction. Otherwise, the database server will not allow your operations.

To request that the database server delay referential integrity checks until commit time, set the value of the option `WAIT_FOR_COMMIT` to `ON`. By default, this option is `OFF`. To turn it on, issue the following command:

```
SET OPTION WAIT_FOR_COMMIT = ON;
```

Before committing a transaction, the database server verifies that referential integrity is maintained by checking the number of orphans your transaction has created. At the end of every transaction, that number must be zero.

Even if the necessary primary key exists at the time you insert the row, the database server must ensure that it still exists when you commit your results. It does so by placing a read lock on the target row. With the read lock in place, any other transaction is still free to read that row, but none can delete or alter it.

## Locking during updates

The database server modifies the information contained in a particular record using the following procedure.

- 1 Write lock the affected row.
- 2 If any entries changed are included in an index, delete each index entry corresponding to the old values. Make a record of any orphans created by doing so.
- 3 Update each of the affected values.
- 4 If indexed values were changed, add new index entries. Verify uniqueness where appropriate and verify referential integrity if a primary or foreign key was changed.
- 5 The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file, including the previous values of all entries in the row, and release all locks.

- 6 Cascade the insert or delete operations, if you have selected this option and primary or secondary keys are affected.

You may be surprised to see that the deceptively simple operation of changing a value in a table can necessitate a rather large number of operations. The amount of work that the database server needs to do is much less if the value you are changing is not part of a primary or foreign key. It is lower still if it is not contained in an index, either explicitly or implicitly because you have declared that attribute unique.

The operation of verifying referential integrity during an UPDATE operation is no less simple than when the verification is performed during an INSERT. In fact, when you change the value of a primary key, you may create orphans. When you insert the replacement value, the database server must check for orphans once more.

## Locking during deletes

The DELETE operation follows almost the same steps as the INSERT operation, except in the opposite order.

- 1 Write lock the affected row.
- 2 Delete each index entry present for the any values in the row. Immediately prior to deleting each index entry, acquire one or more anti-insert locks as necessary to prevent another transaction inserting a similar entry before the delete is committed. In order to verify referential integrity, the database server also keeps track of any orphans created as a side effect of the deletion.
- 3 Remove the row from the table so that it is no longer visible to other transactions. The row cannot be destroyed until the transaction is committed because doing so would remove the option of rolling back the transaction.
- 4 The transaction can be committed provided referential integrity will not be violated by doing so: record the operation in the transaction log file including the values of all entries in the row, release all locks, and destroy the row.
- 5 Cascade the delete operation, if you have selected this option and have modified a primary or foreign key.

### Anti-insert locks

The database server must ensure that the DELETE operation can be rolled back. It does so in part by acquiring anti-insert locks. These locks are not exclusive; however, they deny other transactions the right to insert rows that make it impossible to roll back the DELETE operation. For example, the row deleted may have contained a primary key value, or another unique value. Were another transaction allowed to insert a row with the same value, the DELETE could not be undone without violating the uniqueness property.

Adaptive Server Anywhere enforces uniqueness constraints through indexes. In the case of a simple table with only a one-attribute primary key, a single phantom lock may suffice. Other arrangements can quickly escalate the number of locks required. For example, the table may have no primary key or other index associated with any of the attributes. Since the rows in a table have no fundamental ordering, the only way of preventing inserts may be to anti-insert lock the entire table.

Deleting a row can mean acquiring a great many locks. You can minimize the effect on concurrency in your database in a number of ways. As described earlier, indexes and primary keys reduce the number of locks required because they impose an ordering on the rows in the table. The database server automatically takes advantage of these orderings. Instead of acquiring locks on every row in the table, it can simply lock the *next* row. Without the index, the rows have no order and thus the concept of a next row is meaningless.

The database server acquires anti-insert locks on the row following the row deleted. Should you delete the last row of a table, the database server simply places the anti-insert lock on an invisible end row. In fact, if the table contains no index, the number of anti-insert locks required is one more than the number of rows in the table.

### Anti-insert locks and read locks

While one or more anti-insert locks exclude an insert lock and one or more read locks exclude a write lock, no interaction exists between anti-insert/insert locks and read/write locks. For example, although a write lock cannot be acquired on a row that contains a read lock, it can be acquired on a row that has only an anti-insert lock. More options are open to the database server because of this flexible arrangement, but it means that the server must generally take the extra precaution of acquiring a read lock when acquiring an anti-insert lock. Otherwise, another transaction could delete the row.

## Two-phase locking

Often, the general information about locking provided in the earlier sections will suffice to meet your needs. There are times, however, when you may benefit from more knowledge of what goes on inside the database server when you perform basic types of operations. This knowledge will provide you with a better basis from which to understand and predict potential problems that users of your database may encounter.

Two-phase locking is important in the context of ensuring that schedules are serializable. The **two-phase locking protocol** specifies a procedure each transaction follows.

This protocol is important because, if observed by all transactions, it will guarantee a serializable, and thus correct, schedule. It may also help you understand why some methods of locking permit some types of inconsistencies.

### The two-phase locking protocol

- 1 Before operating on any row, a transaction must acquire a lock on that row.
- 2 After releasing a lock, a transaction must never acquire any more locks.

In practice, a transaction normally holds locks until it terminates with either a COMMIT or ROLLBACK statement. Releasing locks before the end of the transaction disallows the operation of rolling back the changes whenever doing so would necessitate operating on rows to return them to an earlier state.

The two-phase locking protocol allows the statement of the following important theorem:

#### **The two-phase locking theorem**

If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.

In other words, if all transactions follow the two-phase locking protocol, then none of the inconsistencies mentioned above are possible.

This protocol defines the operations necessary to ensure complete consistency of your data, but you may decide that some types of inconsistencies are permissible during some operations on your database. Eliminating all inconsistency often means reducing the efficiency of your database.

Write locks are placed on modified, inserted, and deleted rows regardless of isolation level. They are always held until commit and rollback.

Read locks at different isolation levels

Isolation level	Read locks
0	None
1	On rows that appear in the result set; they are held only when a cursor is positioned on a row.
2	On rows that appear in the result set; they are held until the user executes a COMMIT or a ROLLBACK.
3	On all rows read and all insertion points crossed in the computation of a result set

☞ For more information, see "Serializable schedules" on page 102

The details of locking are best broken into two sections: what happens during an INSERT, UPDATE, DELETE or SELECT and how the various isolation levels affect the placement of read, anti-insert, and insert locks.

Although you can control the amount of locking that takes place within the database server by setting the isolation level, there is a good deal of locking that occurs at all levels, even at level 0. These locking operations are fundamental. For example, once one transaction updates a row, no other transaction can modify the same row before the first transaction completes. Without this precaution, you could not rollback the first transaction.

The locking operations that the database server performs at isolation level 0 are the best to learn first exactly because they represent the foundation. The other levels add locking features, but do not remove any present in the lower levels. Thus, moving to higher isolation level adds operations not present at lower levels.

**Early release of read locks—an exception**

At isolation level 3, a transaction acquires a read lock on every row it reads. Ordinarily, a transaction never releases a lock before the end of the transaction. Indeed, it is essential that a transaction does not release locks early if the schedule is to be serializable.

Adaptive Server Anywhere always retains write locks until a transaction completes. If it were to release a lock sooner, another transaction could modify that row making it impossible to roll back the first transaction.

Read locks are released only in one, special circumstance. Under isolation level 1, transactions acquire a read lock on a row only when it becomes the current row of a cursor. Under isolation level 1, however, when that row is no longer current, the lock is released. This behavior is acceptable because the database server does not need to guarantee repeatable reads at isolation level 1.

✍ For more information about isolation levels, see "Choosing isolation levels" on page 102.

## Special optimizations

The previous sections describe the locks acquired when all transactions are operating at a given isolation level. For example, when all transactions are running at isolation level 2, locking is performed as described in the appropriate section, above.

In practice, your database is likely to need to process multiple transactions that are at different levels. A few transactions, such as the transfer of money between accounts, must be serializable and so run at isolation level 3. For other operations, such as updating an address or calculating average daily sales, a lower isolation level will often suffice.

While the database server is not processing any transactions at level 3, it optimizes some operations so as to improve performance. In particular, many extra anti-insert and insert locks are often necessary to support a level 3 transaction. Under some circumstances, the database server can avoid either placing or checking for some types of locks when no level 3 transactions are present.

For example, the database server uses anti-insert locks to guard against two distinct types of circumstances:

- 1 Ensure that deletes in tables with unique attributes can be rolled back.
- 2 Eliminate phantom rows in level 3 transactions.

If no level 3 transactions are using a particular table, then the database server need not place anti-insert locks in the index of a table that contains no unique attributes. If, however, even one level 3 transaction is present, all transactions, even those at level 0, must place anti-insert locks so that the level 3 transactions can identify their operations.

Naturally, the database server always attaches notes to a table when it attempts the types of optimizations described above. Should a level 3 transaction suddenly start, you can be confident that the necessary locks will be put in place for it.

You may have little control over the mix of isolation levels in use at one time as so much will depend on the particular operations that the various users of your database wish to perform. Where possible, however, you may wish to select the time that level 3 operations execute because they have the potential to cause significant slowing of database operations. The impact is magnified because the database server is forced to perform extra operations for lower-level operations.



## Particular concurrency issues

This section discusses the following particular concurrency issues:

- ◆ "Primary key generation" on page 135
- ◆ "Data definition statements and concurrency" on page 136

### Primary key generation

You will encounter situations where the database should automatically generate a unique number. For example, if you are building a table to store sales invoices you might prefer that the database assign unique invoice numbers automatically, rather than require sales staff to pick them.

There are many methods for generating such numbers.

#### Example

For example, invoice numbers could be obtained by adding 1 to the previous invoice number. This method will not work when there is more than one person adding invoices to the database. Two people may decide to use the same invoice number.

There is more than one solution to the problem:

- ◆ Assign a range of invoice numbers to each person who adds new invoices.

You could implement this scheme by creating a table with the columns *user name* and *invoice number*. The table would have one row for each user that adds invoices. Each time a user adds an invoice, the number in the table would be incremented and used for the new invoice. In order to handle all tables in the database, the table should have three columns: table name, user name, and last key value. You should periodically check that each person still has a sufficient supply of numbers.

- ◆ Create a table with the columns: *table name* and *last key value*.

One row in this table would contain the last invoice number used. Each time someone adds an invoice, establish a new connection, increment the number in the table, and commit the change immediately. The incremented number can be used for the new invoice. Other users will be able to grab invoice numbers because you updated the row with a separate transaction that only lasted an instant.

- ◆ Use a column with a default value of NEWID in conjunction with the UNIQUEIDENTIFIER binary data type to generate a universally unique identifier.

UUID and GUID values can be used to uniquely identify rows in a table. The values are generated such that a value produced on one computer will not match that produced on another. They can therefore be used as keys in replication and synchronization environments.

For more information about generating unique identifiers, see "The NEWID default" on page 73.

- ◆ Use a column with a default value of AUTOINCREMENT.


For example,

```
CREATE TABLE orders (  
    order_id INTEGER NOT NULL DEFAULT AUTOINCREMENT,  
    order_date DATE,  
    primary key( order_id )  
)
```

On inserts into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent inserts. The value of the most recently inserted row in an autoincrement column is available as the global variable @@identity.

#### **Unique values in replicated databases**

Different techniques are required if you replicate your database and more than one person can add entries which must later be merged.

 For more information, see "Replication and concurrency" on page 138.

## **Data definition statements and concurrency**

Data definition statements that change an entire table, such as CREATE INDEX, ALTER TABLE, and TRUNCATE TABLE, are prevented whenever the statement table is currently being used by another connection. These data definition statements can be time consuming and the database server will not process requests referencing the same table while the command is being processed.

The CREATE TABLE statement does not cause any concurrency conflicts.

The GRANT statement, REVOKE statement, and SET OPTION statement also do not cause concurrency conflicts. These commands affect any new SQL statements sent to the database server, but do not affect existing outstanding statements.

GRANT and REVOKE for a user are not allowed if that user is connected to the database.

**Data definition statements and replicated databases**

Using data definition statements in replicated databases requires special care. For more information see the separate manual entitled *Data Replication with SQL Remote*.

## Replication and concurrency

Some computers on your network might be portable computers that people take away from the office or which are occasionally connected to the network. There may be several database applications that they would like to use while not connected to the network.

Database replication is the ideal solution to this problem. Using SQL Remote or MobiLink synchronization, you can publish information in a consolidated, or master, database to any number of other computers. You can control precisely the information replicated on any particular computer. Any person can receive particular tables, or even portions of the rows or columns of a table. By customizing the information each receives, you can ensure that their copy of the database is no larger than necessary to contain the information they require.

✍ Extensive information on SQL Remote replication and MobiLink synchronization is provided in the separate manuals entitled *SQL Remote User's Guide* and *MobiLink Synchronization User's Guide*. The information in this section is, thus, not intended to be complete. Rather, it introduces concepts related directly to locking and concurrency considerations.

SQL Remote and MobiLink allow replicated databases to be updated from a central, consolidated database, as well as updating this same central data as the results of transactions processed on the remote machine. Since updates can occur in either direction, this ability is referred to as **bi-directional replication**.

Since the results of transactions can affect the consolidated database, whether they are processed on the central machine or on a remote one, the effect is that of allowing concurrent transactions.

Transactions may happen at the same time on different machines. They may even involve the same data. In this case, though, the machines may not be physically connected. No means may exist by which the remote machine can contact the consolidated database to set any form of lock or identify which rows have changed. Thus, locks can not prevent inconsistencies as they do when all transactions are processed by a single server.

An added complication is introduced by the fact that any given remote machine may not hold a full copy of the database. Consider a transaction executed directly on the main, consolidated database. It may affect rows in two or more tables. The same transaction might not execute on a remote database, as there is no guarantee that one or both of the affected tables is replicated on that machine. Even if the same tables exist, they may not contain exactly the same information, depending upon how recently the information in the two databases has been synchronized.

To accommodate the above constraints, replication is not based on transactions, but rather on operations. An **operation** is a change to one row in a table. This change could be the result of an UPDATE, INSERT, or DELETE statement. An operation resulting from an UPDATE or DELETE identifies the initial values of each column and a transaction resulting from an INSERT or UPDATE records the final values.

A transaction may result in none, one, or more than one operation. One operation will never result from two or more transactions. If two transactions modify a table, then two or more corresponding operations will result.

If an operation results from a transaction processed on a remote computer, then it must be passed to the consolidated database so that the information can be merged. If, on the other hand, an operation results from a transaction on the consolidated computer, then the operation may need to be sent to *some* remote sites, but not others. Since each remote site may contain a replica of a portion of the complete database, SQL Remote knows to pass the operation to a remote site only when it affects that portion of the database.


#### Transaction log based replication

SQL Remote uses a **transaction log based** replication mechanism. When you activate SQL Remote on a machine, it scans the transaction log to identify the operations it must transfer and prepares one or more messages.

SQL Remote can pass these messages between computers using a number of methods. It can create files containing the messages and store them in a designated directory. Alternatively, SQL Remote can pass messages using any of the most common messaging protocols. You likely can use your present e-mail system.

Conflicts may arise when merging operations from remote sites into the consolidated database. For example, two people, each at a different remote site, may have changed the same value in the same table. Whereas the locking facility built into Adaptive Server Anywhere can eliminate conflict between concurrent transactions handled by the same server, it is impossible to automatically eliminate all conflicts between two remote users who both have permission to change the same value.

As the database administrator, you can avoid this potential problem through suitable database design or by writing conflict resolution algorithms. For example, you can decide that only one person will be responsible for updating a particular range of values in a particular table. If such a restriction is impractical, then you can instead use the conflict resolution facilities of SQL Remote to implement triggers and procedures which resolve conflicts in a manner appropriate to the data involved.

 SQL Remote provides the tools and programming facilities you need to take full advantage of database replication. For further information, see the *SQL Remote User's Guide* and the *MobiLink Synchronization User's Guide*.

## Summary

Transactions and locking are perhaps second only in importance to relations between tables. The integrity and performance of any database can benefit from the judicious use of locking and careful construction of transactions. Both are essential to creating databases that must execute a large number of commands concurrently.

Transactions group SQL statements into logical units of work. You may end each by either rolling back any changes you have made or by committing these changes and so making them permanent.

Transactions are essential to data recovery in the event of system failure. They also play a pivotal role in interweaving statements from concurrent transactions.

To improve performance, multiple transactions must be executed concurrently. Each transaction is composed of component SQL statements. When two or more transactions are to be executed concurrently, the database server must schedule the execution of the individual statements. Concurrent transactions have the potential to introduce new, inconsistent results that could not arise were these same transactions executed sequentially.

Many types of inconsistencies are possible, but four typical types are particularly important because they are mentioned in the ISO SQL/92 standard and the isolation levels are defined in terms of them.

- ◆ **Dirty read** One transaction reads data modified, but not yet committed, by another.
- ◆ **Non-repeatable read** A transaction reads the same row twice and gets different values.
- ◆ **Phantom row** A transaction selects rows, using a certain criterion, twice and finds new rows in the second result set.
- ◆ **Lost Update** One transaction's changes to a row are completely lost because another transaction is allowed to save an update based on earlier data.

A schedule is called serializable whenever the effect of executing the statements according to the schedule is the same as could be achieved by executing each of the transactions sequentially. Schedules are said to be **correct** if they are serializable. A serializable schedule will cause none of the above inconsistencies.

Locking controls the amount and types of interference permitted. Adaptive Server Anywhere provides you with four levels of locking: isolation levels 0, 1, 2, and 3. At the highest isolation, level 3, Adaptive Server Anywhere guarantees that the schedule is serializable, meaning that the effect of executing all the transactions is equivalent to running them sequentially.

Unfortunately, locks acquired by one transaction may impede the progress of other transactions. Because of this problem, lower isolation levels are desirable whenever the inconsistencies they may allow are tolerable. Increased isolation to improve data consistency frequently means lowering the concurrency, the efficiency of the database at processing concurrent transactions. You must frequently balance the requirements for consistency against the need for performance to determine the best isolation level for each operation.

Conflicting locking requirements between different transactions may lead to blocking or deadlock. Adaptive Server Anywhere contains mechanisms for dealing with both these situations, and provides you with options to control them.

Transactions at higher isolation levels do not, however, *always* impact concurrency. Other transactions will be impeded only if they require access to locked rows. You can improve concurrency through careful design of your database and transactions. For example, you can shorten the time that locks are held by dividing one transaction into two shorter ones, or you might find that adding an index allows your transaction to operate at higher isolation levels with fewer locks.

The increased popularity of portable computers will frequently mean that your database may need to be replicated. Replication is an extremely convenient feature of Adaptive Server Anywhere, but it introduces new considerations related to concurrency. These topics are covered in a separate manual.





C H A P T E R 5

Monitoring and Improving Performance

About this chapter      This chapter describes how to monitor and improve the performance of your database.

Contents	<table><tr><th>Topic</th><th>Page</th></tr><tr><td>Top performance tips</td><td>144</td></tr><tr><td>Using the cache to improve performance</td><td>152</td></tr><tr><td>Using keys to improve query performance</td><td>157</td></tr><tr><td>Sorting query results</td><td>159</td></tr><tr><td>Use of work tables in query processing</td><td>160</td></tr><tr><td>Monitoring database performance</td><td>162</td></tr><tr><td>Fragmentation</td><td>168</td></tr><tr><td>Profiling database procedures</td><td>172</td></tr></table>	Topic	Page	Top performance tips	144	Using the cache to improve performance	152	Using keys to improve query performance	157	Sorting query results	159	Use of work tables in query processing	160	Monitoring database performance	162	Fragmentation	168	Profiling database procedures	172
Topic	Page																		
Top performance tips	144																		
Using the cache to improve performance	152																		
Using keys to improve query performance	157																		
Sorting query results	159																		
Use of work tables in query processing	160																		
Monitoring database performance	162																		
Fragmentation	168																		
Profiling database procedures	172																		

## Top performance tips

Adaptive Server Anywhere provides excellent performance automatically. However, the following tips will help you achieve the most from the product.

### Always use a transaction log

You might think that Adaptive Server Anywhere would run faster without a transaction log because it would have to maintain less information on disk. Yet, the opposite is actually true. Not only does a transaction log provide a large amount of protection, it can dramatically improve performance.

Operating without a transaction log, Adaptive Server Anywhere must perform a checkpoint at the end of every transaction. Writing these changes consumes considerable resources.

With a transaction log, however, Adaptive Server Anywhere need only write notes detailing the changes as they occur. It can choose to write the new database pages all at once, at the most efficient time. Checkpoints make sure information enters the database file, and that it is consistent and up to date.

#### Tip

Always use a transaction log. It helps protect your data *and* it greatly improves performance.

If you can store the transaction log on a different physical device than the one containing the main database file, you can further improve performance. The extra drive head does not generally have to seek to get to the end of the transaction log.

### Increase the cache size

Adaptive Server Anywhere stores recently used pages in a cache. Should a request need to access the page more than once, or should another connection require the same page, it may find it already in memory and hence avoid having to read information from disk. This is especially an issue for encrypted databases, which require a larger cache than unencrypted.


If your cache is too small, Adaptive Server Anywhere cannot keep pages in memory long enough to reap these benefits.

On UNIX, Windows NT/2000/XP, and Windows 95/98/Me, the database server dynamically changes cache size as needed. However, the cache is still limited by the amount of memory that is physically available, and by the amount used by other applications.

On Windows CE and Novell NetWare, the size of the cache is set on the command line when you launch the database server. Be sure to allocate as much memory to the database cache as possible, given the requirements of the other applications and processes that run concurrently. In particular, databases using Java objects benefit greatly from larger cache sizes. If you use Java in your database, consider a cache of at least 8 Mb.

**Tip**

Increasing the cache size can often improve performance dramatically, since retrieving information from memory is many times faster than reading it from disk. You may find it worthwhile to purchase more RAM to allow a larger cache.

 For more information, see "Using the cache to improve performance" on page 152.

## **Normalize your table structure**

In general, the information in each column of a table should depend solely on the value of the primary key. If this is not the case, then one table may contain multiple copies of the same information, and your table may need to be normalized.

Normalization reduces duplication in a relational database. For example, suppose the people in your company work at a number of offices. To normalize the database, consider placing information about the offices (such as its address and main telephone numbers) in a separate table, rather than duplicating all this information for every employee.

You can, however, take the generally good notion of normalization too far. If the amount of duplicate information is small, you may find it better to duplicate the information and maintain its integrity using triggers or other constraints.

## Use indexes effectively

When executing a query, Adaptive Server Anywhere chooses how to access each table. Indexes greatly speed up the access. When the database server cannot find a suitable index, it instead resorts to scanning the table sequentially—a process that can take a long time.

For example, suppose you need to search a large database for people, but you only know either their first or their last name, but not both. If no index exists, Adaptive Sever Anywhere scans the entire table. If however, you created two indexes (one that contains the last names first, and a second that contains the first names first), Adaptive Sever Anywhere scans the indexes first, and can generally return the information to you faster.


### Using indexes

Although indexes let Adaptive Server Anywhere locate information very efficiently, exercise some caution when adding them. Each index creates extra work every time you insert, delete, or update a row because Adaptive Server Anywhere must also update all affected indexes.

Consider adding an index when it will allow Adaptive Server Anywhere to access data more efficiently. In particular, add an index when it eliminates unnecessarily accessing a large table sequentially. If, however, you need better performance when you add rows to a table, and finding information quickly is not an issue, use as few indexes as possible.

### Using Clustered Indexes

Using clustered indexes stores rows in a table in approximately the same order as they appear in the index.

 For more information, see "Indexes" on page 340.

## Use an appropriate page size

Large page sizes help Adaptive Server Anywhere read databases more efficiently. For example, if you use a large database, or if you access information sequentially, try using a larger page size. Large page sizes also bring other benefits, including improving the fan-out of your indexes, reducing the number of index levels, and letting you create tables with more columns.

You cannot change the page size of an existing database. Instead you must create a new database and use the `-p` flag of `dbinit` to specify the page size. For example, the following command creates a database with 4K pages.

```
dbinit -p 4096 new.db
```

In contrast, however, benefits associated with smaller page sizes often go unrecognized. It is true that smaller pages hold less information and may force less efficient use of space, particularly if you insert rows that are slightly more than half a page in size. However, small page sizes also allow Adaptive Server Anywhere to run with fewer resources because it can store more pages in a cache of the same size. They are particularly useful if your database must run on small machines with limited memory. They can also help in situations when you use your database primarily to retrieve small pieces of information from random locations.

 For more information about larger page sizes, see "Setting a maximum page size" on page 13 of the book *ASA Database Administration Guide*.

#### Scattered reads

If you are working with a Windows NT Service Patch 2 or higher system, or with a Windows 2000/XP system, a page size of at least 4K allows the database server to read a large contiguous region of database pages on disk directly into the appropriate place in cache, bypassing the 64K buffer entirely. This feature can significantly improve performance.

## Place different files on different devices

Disk drives operate much more slowly than modern processors or RAM. Often, simply waiting for the disk to read or write pages is the reason that a database server is slow.

You almost always improve database performance when you put different physical database files on different physical devices. For example, while one disk drive is busy swapping database pages to and from the cache, another device can be writing to the log file.

Notice that to gain these benefits, the devices must be independent. A single disk, partitioned into smaller logical drives, is unlikely to yield benefits.

Adaptive Server Anywhere uses four types of files:

- 1 database (.db)
- 2 transaction log (.log)
- 3 transaction log mirror (.mlg)
- 4 temporary file (.tmp)

The **database file** holds the entire contents of your database. A single file can contain a single database, or you can add up to 12 dbspaces, which are additional files holding the same database. You choose a location for it, appropriate to your needs.

The **transaction log file** is required for recovery of the information in your database in the event of a failure. For extra protection, you can maintain a duplicate in a third type of file called a **transaction log mirror file**.

Adaptive Server Anywhere writes the same information at the same time to each of these files.

**Tip**

By placing the transaction log mirror file (if you use one) on a physically separate drive, you gain better protection against disk failure, and Adaptive Server Anywhere runs faster because it can efficiently write to the log and log mirror files. To specify the location of the transaction log and transaction log mirror files, use the *dblog* command line utility, or the Change Log File Settings utility in Sybase Central.

Adaptive Server Anywhere may need more space than is available to it in the cache for such operations as sorting and forming unions. When it needs this space, it generally uses it intensively. The overall performance of your database becomes heavily dependent on the speed of the device containing the fourth type of file, the **temporary file**.

**Tip**

If the temporary file is on a fast device, physically separate from the one holding the database file, Adaptive Server Anywhere will run faster. This is because many of the operations that necessitate using the temporary file also require retrieving a lot of information from the database. Placing the information on two separate disks allows the operations to take place simultaneously.

On Windows, Adaptive Server Anywhere examines the following environment variables, in the order shown, to determine the directory in which to place the temporary file.

- 1 ASTMP
- 2 TMP
- 3 TMPDIR
- 4 TEMP

If none of these is defined, Adaptive Server Anywhere places its temporary file in the current directory—not a good location for the best performance.

On UNIX, Adaptive Server Anywhere examines the ASTMP environment variable to determine the directory in which to place the temporary file.

If the ASTMP environment variable is not defined, Adaptive Server Anywhere places its temporary file in the */tmp/.SQLAnywhere* directory.

If your machine has a sufficient number of fast devices, you can gain even more performance by placing each of these files on a separate device. You can even divide your database into multiple dbspaces, located on separate devices. In such a case, group tables in the separate dbspaces so that common join operations read information from different files.

A similar strategy involves placing the temporary and database files on a RAID device or a Windows NT stripe set. Although such devices act as a logical drive, they dramatically improve performance by distributing files over many physical drives and accessing the information using multiple heads.

☞ For more information about work tables, see "Use of work tables in query processing" on page 160.

☞ For information about data recovery, see "Backup and Data Recovery" on page 299 of the book *ASA Database Administration Guide*.

☞ For information about transaction logs and the *dbcc* utility, see "Transaction log utility options" on page 509 of the book *ASA Database Administration Guide*.

## Turn off autocommit mode

If your application runs in autocommit mode, then Adaptive Server Anywhere treats each of your statements as a separate transaction. In effect, it is equivalent to appending a COMMIT statement to the end of each of your commands.

Instead of running in autocommit mode, consider grouping your commands so each group performs one logical task. If you do disable autocommit, you must execute an explicit commit after each logical group of commands. Also, be aware that if logical transactions are large, blocking and deadlock can happen.

The cost of using autocommit mode is particularly high if you are not using a transaction log file. Every statement forces a checkpoint—an operation that can involve writing numerous pages of information to disk.

Each application interface has its own way of setting autocommit behavior. For the Open Client, ODBC, and JDBC interfaces, Autocommit is the default behavior.

☞ For more information about autocommit, see "Setting autocommit or manual commit mode" on page 44 of the book *ASA Programming Guide*.

## Check your file, table, and index fragmentation

Performance can suffer if your files, tables, or indexes are excessively fragmented. This becomes more important as your database increases in size. Adaptive Server Anywhere contains stored procedures that generate information about the fragmentation of files, tables, and indexes.

If the decrease in performance is significant, consider rebuilding your database to reduce table and/or index fragmentation. Or, to reduce file fragmentation, you can put the database on a disk partition by itself, or periodically run one of the available Windows utilities.

Sometimes you may want to improve the performance of your database without going through a full rebuild. For example, perhaps a full rebuild is not possible due to requirements for continuous access to the database. You can use the REORGANIZE TABLE statement to defragment rows in a table, or to compress indexes which may have become sparse due to DELETES. Reorganizing the table can reduce the total number of pages used to store the table and its indexes, and it may reduce the number of levels in an index tree as well.

🔗 For more information about detecting and fixing file, table, and index fragmentation, see "Fragmentation" on page 168.

## Use bulk operations methods

If you find yourself loading huge amounts of information into your database, you can benefit from the special tools provided for these tasks.

If you are loading large files, it is more efficient to create indexes on the table after the data is loaded.

🔗 For information on improving bulk operation performance, see "Performance considerations of moving data" on page 422.

## Use the WITH EXPRESS CHECK option when validating tables

If you find that validating large databases with a small cache takes a long time, you can use one of two options to reduce the amount of time it takes. Using the WITH EXPRESS CHECK option with the VALIDATE TABLE statement, or the -fx option with the Validation utility can significantly increase the speed at which your tables validate.

🔗 For information on improving performance when validating databases, see "Improving performance when validating databases" on page 327 of the book *ASA Database Administration Guide*.




## Try using Adaptive Server Anywhere's compression features


Enabling compression for one connection or for all connections, and adjusting the minimum size limit at which packets are compressed can offer significant improvements to Adaptive Server Anywhere performance under some circumstances.

To determine if enabling compression will help in your particular situation, we recommend that you conduct a performance analysis on your particular network and using your particular application before using communication compression in a production environment.

Enabling compression increases the quantity of information stored in data packets, thereby reducing the number of packets required to transmit a particular set of data. By reducing the number of packets, the data can be transmitted more quickly.

Specifying the compression threshold allows you to choose the minimum size of data packets that you want compressed. The optimal value for the compression threshold may be affected by a variety of factors, including the type and speed of network you are using.

 For information about using compression to improve performance, see "Adjusting communication compression settings to improve performance" on page 98 of the book *ASA Database Administration Guide*.


 For more information about compression settings, see the "Compress connection parameter" on page 170 of the book *ASA Database Administration Guide* and the "CompressionThreshold connection parameter" on page 172 of the book *ASA Database Administration Guide*.

## Reduce the number of requests between client and server

If you find yourself in a situation where your:

- ◆ network exhibits poor latency
- ◆ application sends many cursor open and close requests

you can use the LazyClose and PrefetchOnOpen network communication parameters to reduce the number of requests between the client and server and thereby improve performance.

 For information about these parameters, see the "LazyClose connection parameter" on page 182 of the book *ASA Database Administration Guide* and the "PreFetchOnOpen communication parameter" on page 196 of the book *ASA Database Administration Guide*.

## Using the cache to improve performance

The database cache is an area of memory used by the database server to store database pages for repeated fast access. The more pages that are accessible in the cache, the fewer times the database server needs to read data from disk. As reading data from disk is a slow operation, the amount of cache available is often a key factor in determining performance.

You can control the size of the database cache on the database server command line when the database is started.

### Dynamic cache sizing

Adaptive Server Anywhere provides automatic resizing of the database cache. The capabilities are different on different operating systems. On Windows NT/2000/XP, Windows 95/98/Me, and UNIX operating systems, the cache grows and shrinks. On other operating systems, the cache can increase in size, but not decrease. Details are provided in the following sections.

Full **dynamic cache sizing** helps to ensure that the performance of your database server is not impacted by allocating inadequate memory. The cache grows when the database server can usefully use more, as long as memory is available, and shrinks when cache is not required, so that the database server does not unduly impact other applications on the system. The effectiveness of dynamic cache sizing is limited, of course, by the physical memory available on your system.

Dynamic cache sizing removes the need for explicit configuration of database cache in many situations, making Adaptive Server Anywhere even easier to use.

There is no dynamic cache resizing on Windows CE, Novell NetWare, or Linux. When an Address Windowing Extensions (AWE) cache is used, dynamic cache sizing is disabled.

🔗 For more information about AWE caches, see "-cw server option" on page 131 of the book *ASA Database Administration Guide*.

## Limiting the memory used by the cache

The initial, minimum, and maximum cache sizes are all controllable from the database server command line.

- ◆ **Initial cache size** You can control the initial cache size by specifying the database server `-c` command-line option. The default value is as follows:
  - ◆ **Windows CE** The formula is as follows:

```
max( 600K, min( dbsize , physical-memory ) )
```

where *dbsize* is the total size of the database file or files started, and *physical-memory* is 25% of the physical memory on the machine.

- ◆ **Windows NT/2000/XP, Windows 95/98/Me, NetWare** The formula is as follows:

```
max( 2M, min( dbsize , physical-memory ) )
```

where *dbsize* is the total size of the database file or files started, and *physical-memory* is 25% of the physical memory on the machine.

If an AWE cache is used on Windows 2000, Windows XP, or Windows .NET Server the formula is as follows:

```
min( 100% of available memory-128MB, dbsize )
```

An AWE cache is not used if this value is smaller than 3 Gb-128 Mb.

☞ For information about AWE caches, see "–cw server option" on page 131 of the book *ASA Database Administration Guide*.

- ◆ **UNIX** At least 8 Mb.

☞ For information about UNIX initial cache size, see "Dynamic cache sizing (UNIX)" on page 154.

- ◆ **Maximum cache size** You can control the maximum cache size by specifying the database server –ch command-line option. The default is based on an heuristic that depends on the physical memory in your machine.
- ◆ **Minimum cache size** You can control the minimum cache size by specifying the database server –cl command-line option. By default, the minimum cache size is the same as the initial cache size.

You can also disable dynamic cache sizing by using the –ca command-line option.

☞ For more information on command-line options, see "The database server" on page 120 of the book *ASA Database Administration Guide*.

## Dynamic cache sizing (Windows NT/2000/XP, Windows 95/98/Me)

On Windows NT/2000/XP and Windows 95/98/Me, the database server evaluates cache and operating statistics once per minute and computes an optimum cache size. The server computes a target cache size that uses all physical memory currently not in use, except for approximately 5 Mb that is to be left free for system use. The target cache size is never smaller than the specified or implicit minimum cache size. The target cache size never exceeds the specified or implicit maximum cache size, or the sum of the sizes of all open database and temporary files.

To avoid cache size oscillations, the database server increases the cache size incrementally. Rather than immediately adjusting the cache size to the target value, each adjustment modifies the cache size by 75% of the difference between the current and target cache size.

Windows 2000, Windows XP, and Windows .NET Server can use Address Windowing Extensions (AWE) to support large cache sizes by specifying the `-cw` command-line option when starting the database server. AWE caches do not support dynamic cache sizing.

☞ For more information, see "`-cw` server option" on page 131 of the book *ASA Database Administration Guide*.

## Dynamic cache sizing (UNIX)

On UNIX, the database server uses swap space and memory to manage the cache size. The swap space is a system-wide resource on most UNIX operating systems, but not on all. In this section, the sum of memory and swap space is called the **system resources**. See your operating system documentation for details.

On startup, the database allocates the specified maximum cache size from the system resources. It loads some of this into memory (the initial cache size) and keeps the remainder as swap space.

The total amount of system resources used by the database server is constant until the database server shuts down, but the proportion loaded into memory changes. Each minute, the database server evaluates cache and operating statistics. If the database server is busy and demanding of resources, it may move cache pages from swap space into memory. If the server is quiet, it may move them out from memory to swap space.

### Initial cache size

By default, the initial cache size is assigned using an heuristic based on the available system resources. The initial cache size is always less than 1.1 times the total database size.

## Maximum cache size

If the initial cache size is greater than 3/4 of the available system resources, the database server exits with a Not Enough Memory error

The maximum cache must be less than the available system resources on the machine. By default, the maximum cache size is assigned using an heuristic based on the available system resources and the total physical memory on the machine.

If you specify a maximum cache size greater than the available system resources, the server exits with a Not Enough Memory error. If you specify a maximum cache size greater than the available memory, the server warns of performance degradation, but does not exit.

The database server allocates all the *maximum* cache size from the system resources, and does not relinquish it until the server exits. You should be sure that you choose a maximum cache size that gives good Adaptive Server Anywhere performance while leaving space for other applications. The formula for the default maximum cache size is an heuristic that attempts to achieve this balance. You only need to tune the value if the default value is not appropriate on your system.

If you specify a maximum cache size less than 8 Mb, you will not be able to run Java applications. Low maximum cache sizes will impact performance.

☞ You can use the `-ch` server option to set the maximum cache size, and limit automatic cache growth. For more information, see "`-ch` server option" on page 130 of the book *ASA Database Administration Guide*

## Minimum cache size

By default, the minimum cache size on UNIX is 8 Mb.

☞ You can use the `-cl` server option to adjust the minimum cache size For more information, see "`-cl` server option" on page 130 of the book *ASA Database Administration Guide*.

## Monitoring cache size

The following statistics have been added to the Windows Performance Monitor and to the database's property functions.

- ◆ **CurrentCacheSize** The current cache size in kilobytes
- ◆ **MinCacheSize** The minimum allowed cache size in kilobytes
- ◆ **MaxCacheSize** The maximum allowed cache size in kilobytes
- ◆ **PeakCacheSize** The peak cache size in kilobytes

*Note:*

Windows Performance Monitor is available in Windows NT, Windows 2000, and Windows XP.

🔗 For more information on these properties, see "Server-level properties" on page 625 of the book *ASA Database Administration Guide*.

🔗 For information on monitoring performance, see "Monitoring database performance" on page 162.

## Using keys to improve query performance

Primary keys and foreign keys, while used primarily for validation purposes, can also improve database performance.

### Example

The following example illustrates how primary keys can make queries execute more quickly.

```
SELECT *  
FROM employee  
WHERE emp_id = 390
```

The simplest way for the server to execute this query would be to look at all 75 rows in the **employee** table and check the employee ID number in each row to see if it is 390. This does not take very long since there are only 75 employees, but for tables with many thousands of entries a sequential search can take a long time.

The referential integrity constraints embodied by each primary or foreign key are enforced by Adaptive Server Anywhere through the help of an index, implicitly created with each primary or foreign key declaration. The *emp\_id* column is the primary key for the *employee* table. The corresponding primary key index permits the retrieval of employee number 390 quickly. This quick search takes almost the same amount of time whether there are 100 rows or 1,000,000 rows in the employee table.

## Using primary keys to improve query performance

A primary key improves performance on the following statement:

```
SELECT *  
FROM employee  
WHERE emp_id = 390
```

### Information on the Plan tab

The Plan tab in the Results pane contains the following information:

employee <employee>

Whenever the name inside the parentheses on the Plan tab PLAN description is the same as the name of the table, it means that the primary key for the table is used to improve performance.

## Using foreign keys to improve query performance

The following query lists the orders from the customer with customer ID 113:

```
SELECT *  
FROM sales_order  
WHERE cust_id = 113
```

Information on the  
Plan tab

The Plan tab in the Results pane contains the following information:

sales\_order <ky\_so\_customer>

Here *ky\_so\_customer* refers to the foreign key that the *sales\_order* table has for the *customer* table.

## Separate primary and foreign key indexes

Separate indexes are created automatically for primary and foreign keys. This arrangement allows Adaptive Server Anywhere to perform many operations more efficiently. This feature was introduced in version 7.0.



## Sorting query results

Many queries have an `ORDER BY` clause that ensures that the rows appear in a predictable order. Indexes order the information quickly. For example,

```
SELECT *
FROM customer
ORDER BY customer.lname
```

can use the index on the *lname* column of the customer table to access the rows of the customer table in alphabetical order by last name.

### Queries with WHERE and ORDER BY clauses

A potential problem arises when a query has both a `WHERE` clause and an `ORDER BY` clause.

```
SELECT *
FROM customer
WHERE id > 300
ORDER BY company_name
```

The server must decide between two strategies:

- 1 Go through the entire customer table in order by company name, checking each row to see if the customer *id* is greater than 300.
- 2 Use the key on the *id* column to read only the companies with *id* greater than 300. The results would then need to be sorted by company name.

If there are very few *id* values greater than 300, the second strategy is better because only a few rows need to be scanned and quickly sorted. If most of the *id* values are greater than 300, the first strategy is much better because it requires no sorting.

☞ For more information about sorting, see "The `ORDER BY` clause: sorting query results" on page 220, or "The `GROUP BY` clause: organizing query results into groups" on page 213.

## Use of work tables in query processing

Work tables are materialized temporary result sets that are created during the execution of a query. Work tables are used when Adaptive Server Anywhere determines that the cost of using one is less than alternative strategies. Generally, the time to fetch the first few rows is higher when a work table is used, but the cost of retrieving all rows may be substantially lower in some cases if a work table can be used. Because of this difference, Adaptive Server Anywhere chooses different strategies based on the `OPTIMIZATION_GOAL` setting. The default is first-row. When it is set to first-row, Adaptive Server Anywhere tries to avoid work tables. When it is set to all-rows, Adaptive Server Anywhere uses work tables when they reduce the total execution cost of a query.

Work tables are used in the following cases:

When work tables occur

- ◆ When a query has an `ORDER BY`, `GROUP BY`, `OR DISTINCT` clause and Adaptive Server Anywhere does not use an index for sorting the rows. If a suitable index exists and the `OPTIMIZATION_GOAL` setting is first-row, Adaptive Server Anywhere avoids using a work table. However, when `OPTIMIZATION_GOAL` is set to all-rows, it may be more expensive to fetch all the rows of a query using an index than it is to build a work table and sort the rows. Adaptive Server Anywhere chooses the cheaper strategy if the optimization goal is set to all-rows. For `GROUP BY` and `DISTINCT`, the hash-based algorithms use work tables, but are generally more efficient when fetching all the rows out of a query.
- ◆ When a hash join algorithm is chosen, work tables are used to store interim results (if the input doesn't fit into memory) and a work table is used to store the results of the join.
- ◆ When a cursor is opened with sensitive values, a work table is created to hold the row identifiers and primary keys of the base tables. This work table is filled in as rows are fetched from the query in the forward direction. However, if you fetch the last row from the cursor, the entire table is filled in.
- ◆ When a cursor is opened with insensitive semantics, a work table is populated with the results of the query when the query is opened.
- ◆ When a multiple-row `UPDATE` is being performed and the column being updated appears in the `WHERE` clause of the update or in an index being used for the update.
- ◆ When a multiple-row `UPDATE` or `DELETE` has a subquery in the `WHERE` clause that references the table being modified.

- ◆ When performing an INSERT from a SELECT statement and the SELECT statement references the insert table.
- ◆ When performing a multiple row INSERT, UPDATE, or DELETE, and a corresponding trigger is defined on the table that may fire during the operation.

In these cases, the records affected by the operation go into the work table. In certain circumstances, such as keyset-driven cursors, a temporary index is built on the work table. The operation of extracting the required records into a work table can take a significant amount of time before the query results appear. Creating indexes that can be used to do the sorting in the first case, above, improves the time to retrieve the first few rows. However, the total time to fetch all rows may be lower if work tables are used, since these permit query algorithms based on hashing and merge sort. These algorithms use sequential I/O, which is faster than the random I/O used with an index scan.

The query optimizer in the database server analyzes each query to determine whether a work table will give the best performance. Enhancements to the optimizer in new releases of Adaptive Server Anywhere may improve the access plan for queries. No user action is required to take advantage of these optimizations.

#### Notes

The INSERT, UPDATE and DELETE cases above are usually not a performance problem since they are usually one-time operations. However, if problems occur, you may be able to rephrase the command to avoid the conflict and avoid building a work table. This is not always possible.

## Monitoring database performance

Adaptive Server Anywhere provides a set of statistics you can use to monitor database performance. Accessible from Sybase Central, client applications can access the statistics as functions. In addition, the server makes these statistics available to the Windows Performance Monitor.

This section describes how to access performance and related statistics from client applications, how to monitor database performance using Sybase Central, how to monitor database performance using the Windows Performance Monitor, and how to detect file, table, and index fragmentation.

### Obtaining database statistics from a client application

Adaptive Server Anywhere provides a set of system functions that can access information on a per-connection, per-database, or server-wide basis. The kind of information available ranges from static information (such as the server name) to detailed performance-related statistics (such as disk and memory usage).

Functions that  
retrieve system  
information

The following functions retrieve system information:

- ◆ **property function** Provides the value of a given property on an engine-wide basis.
- ◆ **connection\_property function** Provides the value of a given property for a given connection, or by default, for the current connection.
- ◆ **db\_property function** Provides the value of a given property for a given database, or by default, for the current database.

Supply as an argument only the name of the property you wish to retrieve. The functions return the value for the current server, connection, or database.

✍ For more information, see "PROPERTY function" on page 167 of the book *ASA SQL Reference Manual*, "CONNECTION\_PROPERTY function" on page 113 of the book *ASA SQL Reference Manual*, and "DB\_PROPERTY function" on page 128 of the book *ASA SQL Reference Manual*.

✍ For a complete list of the properties available from the system functions, see "System functions" on page 101 of the book *ASA SQL Reference Manual*.

Examples

The following statement sets a variable named *server\_name* to the name of the current server:

```
SET server_name = property( 'name' )
```

The following query returns the user ID for the current connection:

```
SELECT connection_property( 'userid' )
```

The following query returns the filename for the root file of the current database:

```
SELECT db_property( 'file' )
```

### Improving query efficiency

For better performance, a client application monitoring database activity should use the *property\_number* function to identify a named property, and then use the number to repeatedly retrieve the statistic. The following set of statements illustrates the process from Interactive SQL:

```
CREATE VARIABLE propnum INT ;  
CREATE VARIABLE propval INT ;  
SET propnum = property_number( 'cacheread' );  
SET propval = property( propnum )
```

Property names obtained in this way are available for many different database statistics, from the number of transaction log page write operations and the number of checkpoints carried out, to the number of reads of index leaf pages from the memory cache.

You can view many of these statistics in graph form from the Sybase Central database management tool.


## Monitoring database statistics from Sybase Central

With the Sybase Central Performance Monitor, you can graph the statistics of any Adaptive Server Anywhere database server that you can connect to in Sybase Central. All statistics in Sybase Central are shown in the Statistics folder.

Features of the Performance Monitor include:

- ◆ Real-time updates (at adjustable intervals)
- ◆ A color-coded and resizable legend
- ◆ Configurable appearance properties

When you're using the Sybase Central Performance Monitor, note that it uses actual queries against the server to gather its statistics, so the monitor itself affects some statistics (such as Cache Reads/sec). As a more precise alternative, you can graph server statistics using the Windows Performance Monitor.

 For information on setting properties, see "Setting properties for database objects" on page 34.

## Opening the Sybase Central Performance Monitor


You can display the Performance Monitor in the right pane of Sybase Central when you have the Statistics folder open.

### ❖ To open the Performance Monitor:

- 1 Open the Statistics folder for the desired server.
- 2 In the right pane, click the Performance Monitor tab.

#### **Note**

The Performance Monitor only graphs statistics that you have added to it ahead of time.

 See also

- ◆ "Adding and removing statistics" on page 164
- ◆ "Configuring the Sybase Central Performance Monitor" on page 165
- ◆ "Monitoring database statistics from Windows Performance Monitor" on page 165

## Adding and removing statistics

### ❖ To add statistics to the Sybase Central Performance Monitor:

- 1 Open the Statistics folder.
- 2 Make sure that the Statistics Items page in the right pane is showing.
- 3 Right click a statistic that is not currently being graphed and choose Add to Performance Monitor from the popup menu.

### ❖ To remove statistics from the Sybase Central Performance Monitor:

- 1 Do one of the following:
  - ◆ If you are working in the Statistics folder (with the Statistics Items tab showing in the right pane), right click a statistic that is currently being graphed.
  - ◆ If you are working in the Performance Monitor, right click the desired statistic in the legend.
- 2 From the popup menu, choose Remove from Performance Monitor.

**Tip**

You can also add a statistic to or remove one from the Performance Monitor on the statistic's property sheet.

 See also

- ◆ "Opening the Sybase Central Performance Monitor" on page 164
- ◆ "Configuring the Sybase Central Performance Monitor" on page 165
- ◆ "Monitoring database statistics from Windows Performance Monitor" on page 165

## Configuring the Sybase Central Performance Monitor

The Sybase Central Performance Monitor is configurable; you can choose the type of graph it uses and the amount of time between updates to the graph.

❖ **To choose a graph type:**

- 1 Choose Tools►Options.
- 2 On the Options dialog, click the Chart tab.
- 3 Choose a type of graph.

❖ **To set the update interval:**

- 1 Choose Tools►Options.
- 2 On the Options dialog, click the Chart tab.
- 3 Move the slider to reflect a new time value (or type the value directly in the text box provided).

 See also

- ◆ "Opening the Sybase Central Performance Monitor" on page 164
- ◆ "Adding and removing statistics" on page 164
- ◆ "Monitoring database statistics from Windows Performance Monitor" on page 165

## Monitoring database statistics from Windows Performance Monitor

As an alternative to using the Sybase Central Performance Monitor, you can use the Windows Performance Monitor.

The Windows monitor has two advantages:

- ◆ It offers more performance statistics (mainly those concerned with network communications).
- ◆ Unlike the Sybase Central monitor, the Windows monitor is non-intrusive. It uses a shared-memory scheme instead of performing queries against the server, so it does not affect the statistics themselves.

☞ For a complete list of performance statistics you can monitor, see "Performance Monitor statistics" on page 610 of the book *ASA Database Administration Guide*.

*Note:*

Windows Performance Monitor is available in Windows NT, Windows 2000, and Windows XP. If you run multiple versions of Adaptive Server Anywhere simultaneously, it is also possible to run multiple versions of the Performance Monitor simultaneously.

❖ **To use the Windows Performance Monitor in Windows NT:**


- 1 With an Adaptive Server Anywhere engine or database running, start the Performance Monitor:
  - ◆ Choose Start►Programs►Administrative Tools (Common)►Performance Monitor.
- 2 Choose Edit►Add To Chart, or click the Plus sign button on the toolbar. The Add To Chart dialog appears.
- 3 From the Object list, select one of the following:
  - ◆ **Adaptive Server Anywhere Connection** To monitor performance for a single connection. Choose a connection to monitor from the displayed list.
  - ◆ **Adaptive Server Anywhere Database** To monitor performance for a single database. Choose a database to monitor from the displayed list.
  - ◆ **Adaptive Server Anywhere Engine** To monitor performance on a server-wide basis.

The Counter box displays a list of the statistics you can view.

- 4 From the Counter list, click a statistic to view. To select multiple statistics, hold the CTRL or SHIFT keys while clicking.
- 5 If you selected Adaptive Server Anywhere Connection or Adaptive Server Anywhere Database, choose an instance from the Instance box.
- 6 For a description of the selected counter, click Explain.



- 7 To display the counter, click Add.
- 8 When you have selected all the counters you wish to display, click Done.

 For more information about the Windows Performance Monitor, see the online help for the program.

# Fragmentation

As you make changes to your database, the database file, tables, and indexes can become fragmented. Fragmentation can decrease performance. Adaptive Server Anywhere provides information that you can use to assess the level of fragmentation in files, tables, and indexes.

This section describes how to detect fragmentation in files, tables, and indexes, and how to defragment them.


## File fragmentation

Performance can suffer if your database file is excessively fragmented. This is disk fragmentation and it becomes more important as your database increases in size.

The database server determines the number of file fragments in each dbspace when you start a database on Windows NT/2000/XP. The server displays the following information in the server message window when the number of fragments is greater than one:

Database file "mydatabase.db" consists of nnn fragments

You can also obtain the number of database file fragments using the *DBFileFragments* database property.

 For more information, see "Database-level properties" on page 630 of the book *ASA Database Administration Guide*.

### ❖ To eliminate file fragmentation problems:

- ◆ Put the database on a disk partition by itself.
- ◆ Periodically run one of the available Windows disk defragmentation utilities.

## Table fragmentation

When rows are not stored contiguously, or if rows are split onto more than one page, performance decreases because these rows require additional page accesses. Table fragmentation is distinct from file fragmentation.

Adaptive Server Anywhere reserves extra room on each page to allow rows to grow slightly. When an update to a row causes it to grow beyond the original space allocated for it, the row is split and the initial row location contains a pointer to another page where the entire row is stored. For example, filling empty rows with UPDATE statements or inserting new columns into a table can lead to severe row splitting. As more rows are stored on separate pages, more time is required to access the additional pages.

You can reduce the amount of fragmentation in your tables by specifying the percentage of space in a table page that should be reserved for future updates. This PCTFREE specification can be set with CREATE TABLE, ALTER TABLE, DECLARE LOCAL TEMPORARY TABLE, or LOAD TABLE.

☞ For more information, see "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*, "ALTER TABLE statement" on page 233 of the book *ASA SQL Reference Manual*, "DECLARE LOCAL TEMPORARY TABLE statement" on page 386 of the book *ASA SQL Reference Manual*, and "LOAD TABLE statement" on page 472 of the book *ASA SQL Reference Manual*.

You can use the *sa\_table\_fragmentation* stored procedure to obtain information about the degree of fragmentation of your database tables. You must have DBA authority to run this procedure. The following statement calls the *sa\_table\_fragmentation* stored procedure:

```
CALL sa_table_fragmentation ([ 'table_name'
                             [ , 'owner_name' ] ])
```

☞ For more information, see "sa\_table\_fragmentation system procedure" on page 719 of the book *ASA SQL Reference Manual*.

## Defragmenting tables

The following procedures are useful when you detect that performance is poor because a table is highly fragmented. Unloading and reloading the database is more comprehensive in that it defragments all tables, including system tables. To defragment particular tables or parts of tables, run REORGANIZE TABLE. Reorganizing tables does not disrupt database access.

### ❖ To defragment all the tables in a database:

- 1 Unload the database.
- 2 Reload the database to reclaim disk space and improve performance.

☞ For more information about unloading a database, see "Unloading a database using the dbunload command-line utility" on page 514 of the book *ASA Database Administration Guide*.

🔗 For more information about rebuilding a database, see "The Rebuild utility" on page 497 of the book *ASA Database Administration Guide*.

### ❖ To defragment individual tables:

- ◆ Execute a REORGANIZE TABLE statement.

🔗 For more information, see "REORGANIZE TABLE statement" on page 508 of the book *ASA SQL Reference Manual*.

## Index fragmentation

Indexes are designed to speed up searches on particular columns, but they can become fragmented if many DELETES are performed on the indexed table. This may result in reduced performance if the index is accessed frequently and the cache is not large enough to hold all of the index.

The *sa\_index\_density* stored procedure provides information about the degree of fragmentation in a database's indexes. You must have DBA authority to run this procedure. The following statement calls the *sa\_index\_density* stored procedure:

```
CALL sa_index_density ([ 'table_name' [, 'owner_name' ] ] )
```

If your index is highly fragmented, you can run REORGANIZE TABLE. You can also drop the index and recreate it. However, if the index is a primary key, you will also have to drop and recreate the foreign key indexes.

🔗 For more information, see "REORGANIZE TABLE statement" on page 508 of the book *ASA SQL Reference Manual*.

🔗 For more information about dropping an index, see "Deleting indexes" on page 61.

## Monitoring query performance

Adaptive Server Anywhere includes a number of tools for testing the performance of queries. Complete documentation about each tool can be found in a *Readme.txt* file that is located in the same folder as the tool.


fetchtst	<b>Function:</b> Determines the time required for a result set to be retrieved. <b>Location:</b> <i>SQL Anywhere 8\Samples\Asa\PerformanceFetch</i>
odbcfet	<b>Function:</b> Determines the time required for a result set to be retrieved. This function is similar to <i>fetchtst</i> , but with less functionality. <b>Location:</b> <i>SQL Anywhere 8\Samples\Asa\PerformanceFetch</i>

**inctest**                      **Function:** Determines the time required for rows to be inserted into a table.

**Location:**    *SQL Anywhere 8\Samples\Asa\Performance\Insert*

**trantest**                      **Function:** Measures the load that can be handled by a given server configuration given a database design and a set of transactions.

**Location:**    *SQL Anywhere 8\Samples\Asa\Performance\Transaction*


 For information about system procedures that measure query execution times, see "sa\_get\_request\_profile system procedure" on page 694 of the book *ASA SQL Reference Manual* and "sa\_get\_request\_times system procedure" on page 695 of the book *ASA SQL Reference Manual*.

## Profiling database procedures

Procedure profiling shows you how long it takes your stored procedures, functions, events, and triggers to execute. You can also view the execution time for each line of a procedure. Using the database profiling information, you can determine which procedures can be fine-tuned to increase performance within your database.

When profiling is enabled, Adaptive Server Anywhere monitors which stored procedures, functions, events, and triggers are used, keeping track of how long it takes to execute them, and how many times each one is called.

Profiling information is stored in memory by the server and can be viewed in Sybase Central via the Profile tab or in Interactive SQL. Once profiling is enabled, the database gathers profiling information until you disable profiling or until the server is shut down.

 For more information about obtaining profiling information in Interactive SQL, see "Viewing procedure profiling information in Interactive SQL" on page 178.

## Enabling procedure profiling

You can enable profiling in either Sybase Central or Interactive SQL. You must have DBA authority to enable and use procedure profiling.

### ❖ To enable profiling (Sybase Central):

- 1 Connect to your database as a user with DBA authority.
- 2 Select the database in the left pane.
- 3 From the File menu, choose Properties.  
The Database property sheet appears.
- 4 On the Profiling tab, select Enable Profiling on This Database.
- 5 Click OK to close the property sheet.

### Note

You can also right click your database in Sybase Central to enable profiling. From the popup menu, click Profiling ► Profile.

### ❖ To enable profiling (SQL):

- 1 Connect to your database as a user with DBA authority.
- 2 Call the *sa\_server\_option* stored procedure with the ON setting.

For example, enter:

```
CALL sa_server_option ( 'procedure_profiling', 'ON')
```

## Resetting procedure profiling

When you reset profiling, the database clears the old information and immediately starts collecting new information about procedures, functions, events, and triggers.

The following sections assume that you are already connected to your database as a user with DBA authority and that procedure profiling is enabled.

### ❖ To reset profiling (Sybase Central):

- 1 Select the database in the left pane.
- 2 From the File menu, choose Properties.  
The Database property sheet appears.
- 3 On the Profiling tab, click Reset Now.
- 4 Click OK to close the property sheet.

#### Note

You can also right click your database in Sybase Central to reset profiling. From the popup menu, click Profiling ► Reset Profiling.

### ❖ To reset profiling (SQL):

- 1 Call the *sa\_server\_option* stored procedure with the RESET setting.

For example, enter:

```
CALL sa_server_option ('procedure_profiling',  
    'RESET')
```

## Disabling procedure profiling

Once you are finished with the profiling information, you can either disable profiling or you can clear profiling. If you disable profiling, the database stops collecting profiling information and the information that it has collected to that point remains on the Profile tab in Sybase Central. If you clear profiling, the database turns profiling off and removes all the profiling data from the Profile tab in Sybase Central.

❖ **To disable profiling (Sybase Central):**

- 1 Select the database in the left pane.
- 2 From the File menu, choose Properties.  
The Database property sheet appears.
- 3 On the Profiling tab, clear Enable Profiling on This Database.
- 4 Click OK to close the property sheet.

Note

You can also right click your database in Sybase Central to disable profiling. From the popup menu, select Profiling➤Profile.

❖ **To disable profiling (SQL):**

- 1 Call the *sa\_server\_option* stored procedure with the OFF setting.

For example, enter:

```
CALL sa_server_option ('procedure_profiling', 'OFF')
```

❖ **To clear profiling (Sybase Central):**

- 1 Select the database in the left pane.
- 2 From the File menu, choose Properties.  
The Database property sheet appears.
- 3 On the Profiling tab, click Clear Profiling.  
You can only clear profiling if profiling is enabled.
- 4 Click OK to close the property sheet.

Note

You can also right click your database in Sybase Central to clear profiling. From the popup menu, select Profiling➤Clear Profiling.

❖ **To clear profiling (SQL):**

- 1 Call the *sa\_server\_option* stored procedure with the CLEAR setting.

For example, enter:

```
CALL sa_server_option ('procedure_profiling',  
    'CLEAR')
```



## Viewing procedure profiling information in Sybase Central

Procedure profiling provides you with different information depending whether you choose to look at information for your entire database, a specific type of object, or a particular procedure. The information can be displayed in the following ways:

- ◆ details for all profiled objects within the database
- ◆ details for all stored procedures and functions
- ◆ details for all events
- ◆ details for all triggers
- ◆ details for individual profiled objects

You must be connected to your database and have profiling enabled to view profiling information.

When you view profiling information for your entire database, the following columns appear:

- ◆ **Name** lists the name of the object.
- ◆ **Owner** lists the owner of the object.
- ◆ **Milliseconds** lists the total execution time for each object.
- ◆ **Calls** lists the number times each object has been called.
- ◆ **Table** lists which table a trigger belongs to (this column only appears on the database Profile tab).

These columns provide a summary of the profiling information for all of the procedures that have been executed within the database. One procedure can call other procedures, so there may be more items listed than those users call specifically.

### ❖ To view summary profiling information for stored procedures and functions:

- 1 Expand the database in the left pane.
- 2 Select the Procedures and Functions folder in the left pane.

A list of all the stored procedures and functions in your database appears on the Details tab in the right pane.

- 3 Click the Profile tab in the right pane.

Profiling information about all of the stored procedures and functions within your database appears on the Profile tab.

❖ **To view summary profiling information for events:**

- 1 Expand the database in the left pane.
- 2 Select the Events folder in the left pane.  
A list of all the events in your database appears on the Details tab in the right pane.
- 3 Click the Profile tab in the right pane.  
Profiling information about all of the events within your database appears on the Profile tab.

❖ **To view summary profiling information for triggers:**

- 1 Expand the database in the left pane.
- 2 Select the Tables folder in the left pane.  
A list of all the tables in your database appears on the Details tab in the right pane.
- 3 Open the table you want to profile triggers for in the left pane.
- 4 Open the Triggers folder in the left pane.  
A list of all the triggers for the table appears on the Details tab.
- 5 Click the Profile tab in the right pane.  
Profiling information about all of the triggers on your table appears on the Profile tab.

## Viewing profiling information for a specific procedure

Adaptive Server Anywhere provides procedure profiling information about individual stored procedures, functions, events, and triggers. Sybase Central displays different information about individual procedures than it does about all of the stored procedures, functions, events, or triggers within a database.

When you look at the profiling information for a specific procedure, the following columns appear:

- ◆ **Calls** lists the number of times the object has been called.
- ◆ **Milliseconds** lists the total execution time for each object.
- ◆ **Line** lists the line number beside each line of the procedure.
- ◆ **Source** displays the SQL procedure, line by line.

The procedure is broken down line by line and you can examine it to see which lines have longer execution times and therefore might benefit from changes to improve the procedure's performance. You must be connected to the database, have profiling enabled, and have DBA authority to access procedure profiling information.

❖ **To view the profiling information for a stored procedure or function:**

- 1 Expand the database in the left pane.
- 2 Select the Procedures and Functions folder in the left pane.  
A list of all the stored procedures and functions within your database appears on the Details tab in the right pane.
- 3 Click the stored procedure or function you want to profile in the left pane.
- 4 Click the Profile tab in the right pane.  
Profiling information about the specific stored procedure or function appears on the Profile tab in the right pane.

❖ **To view profiling information for an event:**

- 1 Expand the database in the left pane.
- 2 Select the Events folder in the left pane.  
A list of all the events within your database appears on the Details tab in the right pane.
- 3 Click the event you want to profile in the left pane.
- 4 Click the Profile tab in the right pane.  
Profiling information about the specific event appears on the Profile tab in the right pane.

❖ **To view profiling information for triggers:**

- 1 Expand the database in the left pane.
- 2 Select the table you want to profile triggers for in the left pane.
- 3 Open the Triggers folder in the left pane.  
A list of all the triggers within the table appears on the Details tab in the right pane.
- 4 Click the trigger you want to profile in the left pane.
- 5 Click the Profile tab in the right pane.

Profiling information about the specific trigger appears on the Profile tab in the right pane.

## Viewing procedure profiling information in Interactive SQL

You can use stored procedures to view procedure profiling information. The profiling information is the same whether you view it in Sybase Central or in Interactive SQL.

The *sa\_procedure\_profile\_summary* stored procedure provides information about all of the procedures within the database. You can use this procedure to view the profiling data for stored procedures, functions, events, and triggers within the same result set. The following parameters restrict the rows the procedure returns.

- ◆ **p\_object\_name** specify the name of an object to profile.
- ◆ **p\_owner\_name** specify the owner whose objects you want to profile.
- ◆ **p\_table\_name** specify which table you want to profile triggers for.
- ◆ **p\_object\_type** specify the type of object to profile. You can choose from the following four options. Choosing one of these values restricts the result set to only objects of the specified type.
  - ◆ **P** stored procedure
  - ◆ **F** function
  - ◆ **T** trigger
  - ◆ **E** event
- ◆ **p\_ordering** specify the sort order of the result set.

Keep in mind that there may be more items listed than those called specifically by users because one procedure can call another procedure.

The following sections assume that you are already connected to your database as a user with DBA authority and that you have procedure profiling enabled.

### ❖ To view summary profiling information for all procedures:

- 1 Execute the *sa\_procedure\_profile\_summary* stored procedure.

For example, enter:

```
CALL sa_procedure_profile_summary
```

- 2 From the SQL menu, choose Execute.

A result set with information about all of the procedures in your database appears on the Results tab in the Results pane.

🔗 For more information about the *sa\_procedure\_profile\_summary* stored procedure, see "sa\_procedure\_profile\_summary system procedure" on page 713 of the book *ASA SQL Reference Manual*.

## Viewing profiling information for a specific procedure in Interactive SQL

The *sa\_procedure\_profile* stored procedure provides information about individual lines within specific procedures. The result set includes the line number, execution time, and percentage of total execution time for lines within procedures. You can use the following parameters to restrict the rows the procedure returns:

- ◆ **p\_object\_name** specify the name of an object to profile.
- ◆ **p\_owner\_name** specify the owner whose objects you want to profile.
- ◆ **p\_table\_name** specify which table you want to profile triggers for.

If you do not include any parameters in your query, the procedure returns profiling information for all the procedures that have been called.

### ❖ To view profiling information for specific lines within procedures:

- 1 Execute the *sa\_procedure\_profile* stored procedure.

For example, enter:

```
CALL sa_procedure_profile
```

- 2 From the SQL menu, choose Execute.

A result set with profiling information for individual procedure lines appears on the Results tab in the Results pane.

🔗 For more information about the *sa\_procedure\_profile* stored procedure, see "sa\_procedure\_profile system procedure" on page 712 of the book *ASA SQL Reference Manual*.



## PART TWO

# Working with Databases

This part of the manual describes the mechanics of carrying out common tasks with Adaptive Server Anywhere.

---



C H A P T E R 6

Queries: Selecting Data from a Table

About this chapter      The `SELECT` statement retrieves data from the database. You can use it to retrieve a subset of the rows in one or more tables and to retrieve a subset of the columns in one or more tables.

                                 This chapter focuses on the basics of single-table `SELECT` statements. Advanced uses of `SELECT` are described later in this manual.

Contents

Topic	Page
Query overview	184
The <code>SELECT</code> list: specifying columns	187
The <code>FROM</code> clause: specifying tables	194
The <code>WHERE</code> clause: specifying rows	195

## Query overview

A query requests data from the database and receives the results. This process is also known as data retrieval. All SQL queries are expressed using the SELECT statement.

### Queries are made up of clauses

You construct SELECT statements from clauses. In the following SELECT syntax, each new line is a separate clause. Only the more common clauses are listed here.

```
SELECT select-list
  [ FROM table-expression ]
  [ WHERE search-condition ]
  [ GROUP BY column-name ]
  [ HAVING search-condition ]
  [ ORDER BY { expression / integer } ]
```

The clauses in the SELECT statement are as follows:

- ◆ The SELECT clause specifies the columns you want to retrieve. It is the only required clause in the SELECT statement.
- ◆ The FROM clause specifies the tables from which columns are pulled. It is required in all queries that retrieve data from tables. SELECT statements without FROM clauses have a different meaning, and we ignore them in this chapter.
- ◆ The ON clause specifies how tables in the FROM clause are to be joined. It is used only for multi-table queries and is not discussed in this chapter.
- ◆ The WHERE clause specifies the rows in the tables you want to see.
- ◆ The GROUP BY clause allows you to aggregate data.
- ◆ The HAVING clause specifies rows on which aggregate data is to be collected.
- ◆ The ORDER BY clause sorts the rows in the result set. (By default, rows are returned from relational databases in an order that has no meaning.)

Most of the clauses are optional, but if they are included then they must appear in the correct order.

☞ For more information about the SELECT statement syntax, see "SELECT statement" on page 526 of the book *ASA SQL Reference Manual*.

This chapter discusses only the following set of queries:

- ◆ Queries with only a single table in the FROM clause. For information on multi-table queries, see "Joins: Retrieving Data from Several Tables" on page 227.
- ◆ Queries with no GROUP BY, HAVING, or ORDER BY clauses. For information on these, see "Summarizing, Grouping and Sorting Query Results" on page 207.

## SQL queries

In this manual, SELECT statements and other SQL statements appear with each clause on a separate row, and with the SQL keywords in upper case. This is not a requirement. You can type SQL keywords in any case, and you can break lines at any point.

### Keywords and line breaks

For example, the following SELECT statement finds the first and last names of contacts living in California from the *Contact* table.

```
SELECT first_name, last_name
FROM Contact
WHERE state = 'CA'
```

It is equally valid, though not as readable, to enter this statement as follows:

```
SELECT first_name,
last_name from contact
wHere state
= 'CA'
```

### Case sensitivity of strings and identifiers

Identifiers (that is, table names, column names, and so on) are case insensitive in Adaptive Server Anywhere databases.

Strings are case insensitive by default, so that 'CA', 'ca', 'cA', and 'Ca' are equivalent, but if you create a database as case-sensitive then the case of strings is significant. The sample database is case insensitive.

### Qualifying identifiers

You can qualify the names of database identifiers if there is ambiguity about which object is being referred to. For example, the sample database contains several tables with a column called *city*, so you may have to qualify references to *city* with the name of the table. In a larger database you may also have to use the name of the owner of the table to identify the table.

```
SELECT DBA.contact.city
FROM contact
WHERE state = 'CA'
```

Since the examples in this chapter involve single-table queries, column names in syntax models and examples are usually not qualified with the names of the tables or owners to which they belong.

These elements are left out for readability; it is never wrong to include qualifiers.

The remaining sections in this chapter analyze the syntax of the SELECT statement in more detail.

## The SELECT list: specifying columns

### The select list


The select list commonly consists of a series of column names separated by commas, or an asterisk as shorthand to represent all columns.

More generally, the select list can include one or more expressions, separated by commas. The general syntax for the select list looks like this:

**SELECT** *expression* [, *expression* ]...

If any table or column name in the list does not conform to the rules for valid identifiers, you must enclose the identifier in double quotes.

The select list expressions can include \* (all columns), a list of column names, character strings, column headings, and expressions including arithmetic operators. You can also include aggregate functions, which are discussed in "Summarizing, Grouping and Sorting Query Results" on page 207.

 For more information about what expressions can consist of, see "Expressions" on page 15 of the book *ASA SQL Reference Manual*.

The following sections provide examples of the kinds of expressions you can use in a select list.

## Selecting all columns from a table

The asterisk (\*) has a special meaning in SELECT statements. It stands for all the column names in all the tables specified in the FROM clause. You can use it to save typing time and errors when you want to see all the columns in a table.

When you use SELECT \*, the columns are returned in the order in which they were defined when the table was created.

The syntax for selecting all the columns in a table is:

```
SELECT *  
FROM table-expression
```

SELECT \* finds all the columns currently in a table, so that changes in the structure of a table such as adding, removing, or renaming columns automatically modify the results of SELECT \*. Listing the columns individually gives you more precise control over the results.

### Example

The following statement retrieves all columns in the *department* table. No WHERE clause is included; and so this statement retrieves every row in the table:

```
SELECT *
FROM department
```

The results look like this:

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
...	...	...

You get exactly the same results by listing all the column names in the table in order after the *SELECT* keyword:

```
SELECT dept_id, dept_name, dept_head_id
FROM department
```

Like a column name, "\*" can be qualified with a table name, as in the following query:

```
SELECT department.*
FROM department
```

## Selecting specific columns from a table

To *SELECT* only specific columns in a table, use this syntax:

```
SELECT column_name [, column_name ]...
FROM table-name
```

You must separate each column name from the column name that follows it with a comma. For example:

```
SELECT emp_lname, emp_fname
FROM employee
```

### Rearranging the order of columns

The order in which you list the column names determines the order in which the columns are displayed. The two following examples show how to specify column order in a display. Both of them find and display the department names and identification numbers from all five of the rows in the department table, but in a different order.

```
SELECT dept_id, dept_name
FROM department
```

dept_id	dept_name
100	R & D
200	Sales
300	Finance
400	Marketing
...	...

```
SELECT dept_name, dept_id
FROM department
```

dept_name	dept_id
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

## Renaming columns in query results

Query results consist of a set of columns. By default, the heading for each column is the expression supplied in the select list.

When query results are displayed, each column's default heading is the name given to it when it was created. You can specify a different column heading, or **alias**, in one of the following ways:

```
SELECT column-name AS alias
```

```
SELECT column-name alias
```

```
SELECT alias = column-name
```

Providing an alias can produce more readable results. For example, you can change *dept\_name* to *Department* in a listing of departments as follows:

```
SELECT dept_name AS Department ,
       dept_id AS "Identifying Number"
FROM department
```

Department	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
...	...

Using spaces and keywords in alias

The *Identifying Number* alias for *dept\_id* is enclosed in double quotes because it is an identifier. You also use double quotes if you wish to use keywords in aliases. For example, the following query is invalid without the quotation marks:

```
SELECT dept_name AS Department ,
       dept_id AS "integer"
FROM department
```

If you wish to ensure compatibility with Adaptive Server Enterprise, you should use quoted aliases of 30 bytes or less.

Character strings in query results

The *SELECT* statements you have seen so far produce results that consist solely of data from the tables in the *FROM* clause. Strings of characters can also be displayed in query results by enclosing them in single quotation marks and separate them from other elements in the select list with commas.

To enclose a quotation mark in a string, you precede it with another quotation mark.

For example:

```
SELECT 'The department''s name is' AS " ",
       Department = dept_name
FROM department
```

	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping



## Computing values in the SELECT list

### Arithmetic operations

The expressions in the select list can be more complicated than just column names or strings. For example, you can perform computations with data from numeric columns in a select list.

To illustrate the numeric operations you can carry out in the select list, we start with a listing of the names, quantity in stock, and unit price of products in the sample database.

```
SELECT name, quantity, unit_price
FROM product
```

name	quantity	unit_price
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
...	...	...

Suppose the practice is to replenish the stock of a product when there are ten items left in stock. The following query lists the number of each product that must be sold before re-ordering:

```
SELECT name, quantity - 10
      AS "Sell before reorder"
FROM product
```

name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
...	...

You can also combine the values in columns. The following query lists the total value of each product in stock:

```
SELECT name,
      quantity * unit_price AS "Inventory value"
FROM product
```

name	Inventory value
Tee Shirt	252
Tee Shirt	756
Tee Shirt	1050
Baseball Cap	1008
...	...

### Arithmetic operator precedence

When there is more than one arithmetic operator in an expression, multiplication, division, and modulo are calculated first, followed by subtraction and addition. When all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right. Expressions within parentheses take precedence over all other operations.

For example, the following *SELECT* statement calculates the total value of each product in inventory, and then subtracts five dollars from that value.

```
SELECT name, quantity * unit_price - 5
FROM product
```

To avoid misunderstandings, it is recommended that you use parentheses. The following query has the same meaning and gives the same results as the previous one, but some may find it easier to understand:

```
SELECT name, ( quantity * unit_price ) - 5
FROM product
```

🔗 For more information on operator precedence, see "Operator precedence" on page 13 of the book *ASA SQL Reference Manual*.

### String operations

You can concatenate strings using a string concatenation operator. You can use either `||` (SQL/92 compliant) or `+` (supported by Adaptive Server Enterprise) as the concatenation operator.

The following example illustrates the use of the string concatenation operator in the select list:

```
SELECT emp_id, emp_fname || ' ' || emp_lname AS Name
FROM employee
```

emp_id	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

Date and time  
operations

Although you can use operators on date and time columns, this typically involves the use of functions. For information on SQL functions, see "SQL Functions" on page 93 of the book *ASA SQL Reference Manual*.

## Eliminating duplicate query results

The optional **DISTINCT** keyword eliminates duplicate rows from the results of a **SELECT** statement.

If you do not specify **DISTINCT**, you get all rows, including duplicates. Optionally, you can specify **ALL** before the select list to get all rows. For compatibility with other implementations of SQL, Adaptive Server syntax allows the use of **ALL** to explicitly ask for all rows. **ALL** is the default.

For example, if you search for all the cities in the *contact* table without **DISTINCT**, you get 60 rows:

```
SELECT city
FROM contact
```

You can eliminate the duplicate entries using **DISTINCT**. The following query returns only 16 rows.:

```
SELECT DISTINCT city
FROM contact
```

NULL values are  
not distinct

The **DISTINCT** keyword treats NULL values as duplicates of each other. In other words, when **DISTINCT** is included in a **SELECT** statement, only one NULL is returned in the results, no matter how many NULL values are encountered.

## The FROM clause: specifying tables

The FROM clause is required in every SELECT statement involving data from tables or views.

✍ The FROM clause can include JOIN conditions linking two or more tables, and can include joins to other queries (derived tables). For information on these features, see "Joins: Retrieving Data from Several Tables" on page 227.

### Qualifying table names

In the FROM clause, the full naming syntax for tables and views is always permitted, such as:

```
SELECT select-list
FROM owner.table_name
```

Qualifying table and view names is necessary only when there might be some confusion about the name.

### Using correlation names

You can give a table name a correlation name to save typing. You assign the correlation name in the FROM clause by typing it after the table name, like this:

```
SELECT d.dept_id, d.dept_name
FROM Department d
```

All other references to the *Department* table, for example in a WHERE clause, *must* use the correlation name. Correlation names must conform to the rules for valid identifiers.

✍ For more information about the FROM clause, see "FROM clause" on page 433 of the book *ASA SQL Reference Manual*.

## The WHERE clause: specifying rows

The WHERE clause in a SELECT statement specifies the search conditions for exactly which rows are retrieved. The general format is:

```
SELECT select_list
FROM table_list
WHERE search-condition
```

Search conditions, also called qualifications or predicates, in the WHERE clause include the following:

- ◆ **Comparison operators** (=, <, >, and so on) For example, you can list all employees earning more than \$50,000:

```
SELECT emp_lname
FROM employee
WHERE salary > 50000
```

- ◆ **Ranges** (BETWEEN and NOT BETWEEN) For example, you can list all employees earning between \$40,000 and \$60,000:

```
SELECT emp_lname
FROM employee
WHERE salary BETWEEN 40000 AND 60000
```

- ◆ **Lists** (IN, NOT IN) For example, you can list all customers in Ontario, Quebec, or Manitoba:

```
SELECT company_name , state
FROM customer
WHERE state IN( 'ON', 'PQ', 'MB' )
```

- ◆ **Character matches** (LIKE and NOT LIKE) For example, you can list all customers whose phone numbers start with 415. (The phone number is stored as a string in the database):

```
SELECT company_name , phone
FROM customer
WHERE phone LIKE '415%'
```

- ◆ **Unknown values** (IS NULL and IS NOT NULL) For example, you can list all departments with managers:

```
SELECT dept_name
FROM Department
WHERE dept_head_id IS NOT NULL
```

- ◆ **Combinations** (AND, OR) For example, you can list all employees earning over \$50,000 whose first name begins with the letter A.

```
SELECT emp_fname, emp_lname
FROM employee
WHERE salary > 50000
```

```
AND emp_fname like 'A%'
```

In addition, the WHERE keyword can introduce the following:

- ◆ **Transact-SQL join conditions** Joins are discussed in "Joins: Retrieving Data from Several Tables" on page 227.

☞ For more information about search conditions, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*.

The following sections describe how to use WHERE clauses.

## Using comparison operators in the WHERE clause

You can use comparison operators in the WHERE clause. The operators follow the syntax:

**WHERE** *expression comparison-operator expression*

☞ For more information about comparison operators, see "Comparison operators" on page 10 of the book *ASA SQL Reference Manual*. For a description of what an expression can consist of, see "Expressions" on page 15 of the book *ASA SQL Reference Manual*.

Notes on  
comparisons

- ◆ **Sort orders** In comparing character data, < means earlier in the sort order and > means later in the sort order. The sort order is determined by the collation chosen when the database is created. You can find out the collation by running the *dbinfo* command-line utility against the database:

```
dbinfo -c "uid=DBA;pwd=SQL"
```

You can also find the collation from Sybase Central. It is on the Extended Information tab of the database property sheet.

- ◆ **Trailing blanks** When you create a database, you indicate whether trailing blanks are to be ignored or not for the purposes of comparison.

By default, databases are created with trailing blanks not ignored. For example, 'Dirk' is not the same as 'Dirk '. You can create databases with blank padding, so that trailing blanks are ignored. Trailing blanks are ignored by default in Adaptive Server Enterprise databases.

- ◆ **Comparing dates** In comparing dates, < means earlier and > means later.
- ◆ **Case sensitivity** When you create a database, you indicate whether string comparisons are case sensitive or not.

By default, databases are created case insensitive. For example, 'Dirk' is the same as 'DIRK'. You can create databases to be case sensitive, which is the default behavior for Adaptive Server Enterprise databases.

Here are some SELECT statements using comparison operators:

```
SELECT *
FROM product
WHERE quantity < 20

SELECT E.emp_lname, E.emp_fname
FROM employee E
WHERE emp_lname > 'McBadden'

SELECT id, phone
FROM contact
WHERE state != 'CA'
```

### The NOT operator

The NOT operator negates an expression. Either of the following two queries will find all Tee shirts and baseball caps that cost \$10 or less. However, note the difference in position between the negative logical operator (NOT) and the negative comparison operator (!>).

```
SELECT id, name, quantity
FROM product
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND NOT unit_price > 10

SELECT id, name, quantity
FROM product
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND unit_price !> 10
```

## Using ranges (between and not between) in the WHERE clause

The BETWEEN keyword specifies an inclusive range, in which the lower value and the upper value are searched for as well as the values they bracket.

### ❖ To list all the products with prices between \$10 and \$15, inclusive:

- ◆ Type the following query:

```
SELECT name, unit_price
FROM product
WHERE unit_price BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

You can use NOT BETWEEN to find all the rows that are not inside the range.

❖ **To list all the products cheaper than \$10 or more expensive than \$15:**

- ◆ Execute the following query:

```
SELECT name, unit_price
FROM product
WHERE unit_price NOT BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7
...	...

## Using lists in the WHERE clause

The IN keyword allows you to select values that match any one of a list of values. The expression can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery.

For example, without in, if you want a list of the names and states of all the contacts who live in Ontario, Manitoba, or Quebec, you can type this query:

```
SELECT company_name , state
FROM customer
WHERE state = 'ON' OR state = 'MB' OR state = 'PQ'
```

However, you get the same results if you use IN. The items following the IN keyword must be separated by commas and enclosed in parentheses. Put single quotes around character, date, or time values. For example:



```
SELECT company_name , state
FROM customer
WHERE state IN( 'ON', 'MB', 'PQ')
```

Perhaps the most important use for the IN keyword is in nested queries, also called subqueries.

## Matching character strings in the WHERE clause

The LIKE keyword indicates that the following character string is a matching pattern. LIKE is used with character, binary, or date and time data.

The syntax for LIKE is:

```
{ WHERE | HAVING } expression [ NOT ] LIKE match-expression
```

The expression to be matched is compared to a match-expression that can include these special symbols:

Symbols	Meaning
%	Matches any string of 0 or more characters
_	Matches any one character
[specifier]	<p>The specifier in the brackets may take the following forms:</p> <ul style="list-style-type: none"> <li>◆ <b>Range</b> A range is of the form <i>rangespec1-rangespec2</i>, where <i>rangespec1</i> indicates the start of a range of characters, the hyphen indicates a range, and <i>rangespec2</i> indicates the end of a range of characters</li> <li>◆ <b>Set</b> A set can be comprised of any discrete set of values, in any order. For example, [a2bR].</li> </ul> <p>Note that the range [a-f], and the sets [abcdef] and [fcbaed] return the same set of values.</p>
[^specifier]	<p>The caret symbol (^) preceding a specifier indicates non-inclusion. [^a-f] means not in the range a-f; [^a2bR] means not a, 2, b, or R.</p>

You can match the column data to constants, variables, or other columns that contain the wildcard characters displayed in the table. When using constants, you should enclose the match strings and character strings in single quotes.

### Examples

All the following examples use LIKE with the *last\_name* column in the *Contact* table. Queries are of the form:

```
SELECT last_name
FROM contact
WHERE last_name LIKE match-expression
```

The first example would be entered as

```
SELECT last_name
FROM contact
WHERE last_name LIKE 'Mc%'
```

Match expression	Description	Returns
'Mc%'	Search for every name that begins with the letters <b>Mc</b>	McEvoy
'%er'	Search for every name that ends with <b>er</b>	Brier, Miller, Weaver, Rayner
'%en%'	Search for every name containing the letters <b>en</b> .	Pettengill, Lencki, Cohen
'_ish'	Search for every four-letter name ending in <b>ish</b> .	Fish
'Br[iy][ae]r'	Search for Brier, Bryer, Briar, or Bryar.	Brier
'[M-Z]owell'	Search for all names ending with <b>owell</b> that begin with a single letter in the range M to Z.	Powell
'M[^c]%'	Search for all names beginning with M' that do not have c as the second letter	Moore, Mulley, Miller, Masalsky

Wildcards require LIKE

Wildcard characters used without LIKE are interpreted as **literals** rather than as a pattern: they represent exactly their own values. The following query attempts to find any phone numbers that consist of the four characters 415% only. It does not find phone numbers that start with 415.

```
SELECT phone
FROM Contact
WHERE phone = '415%'
```

Using LIKE with date and time values

You can use LIKE on date and time fields as well as on character data. When you use LIKE with date and time values, the dates are converted to the standard DATETIME format, and then to VARCHAR.

One feature of using LIKE when searching for DATETIME values is that, since date and time entries may contain a variety of date parts, an equality test has to be written carefully in order to succeed.

For example, if you insert the value 9:20 and the current date into a column named *arrival\_time*, the clause:

```
WHERE arrival_time = '9:20'
```

fails to find the value, because the entry holds the date as well as the time. However, the clause below would find the 9:20 value:

```
WHERE arrival_time LIKE '%09:20%'
```

## Using NOT LIKE

With NOT LIKE, you can use the same wildcard characters that you can use with LIKE. To find all the phone numbers in the *Contact* table that do not have 415 as the area code, you can use either of these queries:

```
SELECT phone
FROM Contact
WHERE phone NOT LIKE '415%'

SELECT phone
FROM Contact
WHERE NOT phone LIKE '415%'
```

## Character strings and quotation marks

When you enter or search for character and date data, you must enclose it in single quotes, as in the following example.

```
SELECT first_name, last_name
FROM contact
WHERE first_name = 'John'
```

If the *quoted\_identifier* database option is set to OFF (it is ON by default), you can also use double quotes around character or date data.

Interactive SQL automatically sets *quoted\_identifier* to ON for the duration of the Interactive SQL session.

### ❖ To set the *quoted\_identifier* option off for the current user ID:

- ◆ Type the following command:

```
SET OPTION quoted_identifier = 'OFF'
```

The *quoted\_identifier* option is provided for compatibility with Adaptive Server Enterprise. By default, the Adaptive Server Enterprise option is *quoted\_identifier* OFF and the Adaptive Server Anywhere option is *quoted\_identifier* ON.

## Quotation marks in strings

There are two ways to specify literal quotations within a character entry. The first method is to use two consecutive quotation marks. For example, if you have begun a character entry with a single quotation mark and want to include a single quotation mark as part of the entry, use two single quotation marks:

```
'I don''t understand.'
```

With double quotation marks (*quoted\_identifier* OFF):

```
"He said, ""It is not really confusing."""
```

The second method, applicable only with *quoted\_identifier* OFF, is to enclose a quotation in the other kind of quotation mark. In other words, surround an entry containing double quotation marks with single quotation marks, or vice versa. Here are some examples:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn't there a better way?'"
```

## Unknown Values: NULL

A NULL in a column means that the user or application has made no entry in that column. A data value for the column is unknown or not available

NULL does not mean the same as zero (numerical values) or blank (character values). Rather, NULL values allow you to distinguish between a deliberate entry of zero for numeric columns or blank for character columns and a non-entry, which is NULL for both numeric and character columns.

### Entering NULL

NULL can be entered in a column where NULL values are permitted, as specified in the create table statement, in two ways:

- ◆ **Default** If no data is entered, and the column has no other default setting, NULL is entered.
- ◆ **Explicit entry** You can explicitly enter the value NULL by typing the word NULL (without quotation marks).

If the word NULL is typed in a character column with quotation marks, it is treated as data, not as a null value.

For example, the *dept\_head\_id* column of the department table allows nulls. You can enter two rows for departments with no manager as follows:

```
INSERT INTO department (dept_id, dept_name)
VALUES (201, 'Eastern Sales')

INSERT INTO department
VALUES (202, 'Western Sales', null)
```

### When NULLs are retrieved

When NULLS are retrieved, displays of query results in Interactive SQL show (NULL) in the appropriate position:

```
SELECT *
FROM department
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(NULL)
202	Western Sales	(NULL)

## Testing a column for NULL

You can use `IS NULL` in search conditions to compare column values to `NULL` and to select them or perform a particular action based on the results of the comparison. Only columns that return a value of `TRUE` are selected or result in the specified action; those that return `FALSE` or `UNKNOWN` do not.

The following example selects only rows for which *unit\_price* is less than \$15 or is `NULL`:

```
SELECT quantity , unit_price
FROM product
WHERE unit_price < 15
OR unit_price IS NULL
```

The result of comparing any value to `NULL` is `UNKNOWN`, since it is not possible to determine whether `NULL` is equal (or not equal) to a given value or to another `NULL`.

There are some conditions that never return true, so that queries using these conditions do not return result sets. For example, the following comparison can never be determined to be true, since `NULL` means having an unknown value:

```
WHERE column1 > NULL
```

This logic also applies when you use two column names in a `WHERE` clause, that is, when you join two tables. A clause containing the condition

```
WHERE column1 = column2
```

does not return rows where the columns contain `NULL`.

You can also find `NULL` or non-`NULL` with this pattern:

```
WHERE column_name IS [NOT] NULL
```

For example:

```
WHERE advance < $5000
OR advance IS NULL
```

☞ For more information, see "NULL value" on page 48 of the book *ASA SQL Reference Manual*.

## Properties of NULL

The following list expands on the properties of NULL.

- ◆ **The difference between FALSE and UNKNOWN** Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN, because the opposite of false ("not false") is true. For example,

```
1 = 2
```

evaluates to false and its opposite,

```
1 != 2
```

evaluates to true. But "not unknown" is still unknown. If null values are included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value.

- ◆ **Substituting a value for NULLs** Use the ISNULL built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

```
ISNULL( expression, value )
```

For example, use the following statement to select all the rows from test, and display all the null values in column t1 with the value unknown.

```
SELECT ISNULL(t1, 'unknown')
FROM test
```

- ◆ **Expressions that evaluate to NULL** An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are null. For example:

```
1 + column1
```

evaluates to NULL if column1 is NULL.

- ◆ **Concatenating strings and NULL** If you concatenate a string and NULL, the expression evaluates to the string. For example:

```
SELECT 'abc' || NULL || 'def'
```

returns the string **abcdef**.

## Connecting conditions with logical operators

The logical operators AND, OR, and NOT are used to connect search conditions in WHERE clauses.

### Using AND

The AND operator joins two or more conditions and returns results only when all of the conditions are true. For example, the following query finds only the rows in which the contact's last name is Purcell and the contact's first name is Beth. It does not find the row for Beth Glassmann.

```
SELECT *
FROM contact
WHERE first_name = 'Beth'
      AND last_name = 'Purcell'
```

### Using OR

The OR operator also connects two or more conditions, but it returns results when *any* of the conditions is true. The following query searches for rows containing variants of Elizabeth in the *first\_name* column.

```
SELECT *
FROM contact
WHERE first_name = 'Beth'
      OR first_name = 'Liz'
```

### Using NOT

The NOT operator negates the expression that follows it. The following query lists all the contacts who do not live in California:

```
SELECT *
FROM contact
WHERE NOT state = 'CA'
```

When more than one logical operator is used in a statement, AND operators are normally evaluated before OR operators. You can change the order of execution with parentheses. For example:

```
SELECT *
FROM contact
WHERE ( city = 'Lexington'
      OR city = 'Burlington' )
      AND state = 'MA'
```





C H A P T E R   7

# Summarizing, Grouping and Sorting Query Results

About this chapter      Aggregate functions display summaries of the values in specified columns. You can also use the GROUP BY clause, HAVING clause, and ORDER BY clause to group and sort the results of queries using aggregate functions, and the UNION operator to combine the results of queries.

                                 This chapter describes how to group and sort query results.

Contents	<b>Topic</b>	<b>Page</b>
	Summarizing query results using aggregate functions	208
	The GROUP BY clause: organizing query results into groups	213
	Understanding GROUP BY	214
	The HAVING clause: selecting groups of data	218
	The ORDER BY clause: sorting query results	220
	The UNION operation: combining queries	223
	Standards and compatibility	225

## Summarizing query results using aggregate functions

You can apply aggregate functions to all the rows in a table, to a subset of the table specified by a **WHERE** clause, or to one or more groups of rows in the table. From each set of rows to which an aggregate function is applied, Adaptive Server Anywhere generates a single value.

The following aggregate functions are available:

- ◆ **avg( expression )** The mean of the supplied expression over the returned rows.
- ◆ **count( expression )** The number of rows in the supplied group where the expression is not **NULL**.
- ◆ **count(\*)** The number of rows in each group.
- ◆ **list( string-expr )** A string containing a comma-separated list composed of all the values for *string-expr* in each group of rows.
- ◆ **max( expression )** The maximum value of the expression, over the returned rows.
- ◆ **min( expression )** The minimum value of the expression, over the returned rows.
- ◆ **sum( expression )** The sum of the expression, over the returned rows.

You can use the optional keyword **DISTINCT** with **AVG**, **SUM**, **LIST**, and **COUNT** to eliminate duplicate values before the aggregate function is applied.

The expression to which the syntax statement refers is usually a column name. It can also be a more general expression.

For example, with this statement you can find what the average price of all products would be if one dollar were added to each price:

```
SELECT AVG (unit_price + 1)
FROM product
```

### Example


The following query calculates the total payroll from the annual salaries in the employee table:

```
SELECT SUM(salary)
FROM employee
```

To use aggregate functions, you must give the function name followed by an expression on whose values it will operate. The expression, which is the *salary* column in this example, is the function's argument and must be specified inside parentheses.

## Where you can use aggregate functions

The aggregate functions can be used in a select list, as in the previous examples, or in the HAVING clause of a select statement that includes a GROUP BY clause.

 For more information about the HAVING clause, see "The HAVING clause: selecting groups of data" on page 218.

You cannot use aggregate functions in a WHERE clause or in a JOIN condition. However, a SELECT statement with aggregate functions in its select list often includes a WHERE clause that restricts the rows to which the aggregate is applied.

If a SELECT statement includes a WHERE clause, but not a GROUP BY clause, an aggregate function produces a single value for the subset of rows that the WHERE clause specifies.

Whenever an aggregate function is used in a SELECT statement that does not include a GROUP BY clause, it produces a single value. This is true whether it is operating on all the rows in a table or on a subset of rows defined by a where clause.

You can use more than one aggregate function in the same select list, and produce more than one scalar aggregate in a single SELECT statement.

### Aggregate functions and outer references

Adaptive Server Anywhere version 8 follows new SQL/99 standards for clarifying the use of aggregate functions when they appear in a subquery. These changes affect the behavior of statements written for previous versions of the software: previously correct queries may now produce error messages, and result sets may change.

When an aggregate function appears in a subquery, and the column referenced by the aggregate function is an outer reference, the entire aggregate function itself is now treated as an outer reference. This means that the aggregate function is now computed in the outer block, not in the subquery, and becomes a constant within the subquery.

The following restrictions now apply to the use of outer reference aggregate functions in subqueries:

- ◆ The outer reference aggregate function can only appear in subqueries that are in the SELECT list or HAVING clause, and these clauses must be in the immediate outer block.

- ◆ Outer reference aggregate functions can only contain one outer column reference.
- ◆ Local column references and outer column references cannot be mixed in the same aggregate function.

Note that some problems related to the new standards can be circumvented by rewriting the aggregate function so that it only includes local references. For example, the subquery `(SELECT MAX(S.y + R.y) FROM S)` contains both a local column reference (`S.y`) and an outer column reference (`R.y`), which is now illegal. It can be rewritten as `(SELECT MAX(S.y) + R.y FROM S)`. In the rewrite, the aggregate function has only a local column reference. The same sort of rewrite can be used when an outer reference aggregate function appears in clauses other than `SELECT` or `HAVING`.

### Example

The following query produced the following results in Adaptive Server Anywhere version 7.

```
SELECT name, (SELECT SUM(p.quantity) FROM
              sales_order_items)
FROM product p
```

name	sum(p.quantity)
Tee shirt	30,716
Tee shirt	59,238

In version 8, the same query produces the error message ASA Error -149: Function or column reference to 'name' must also appear in a `GROUP BY`. The reason that the statement is no longer valid is that the outer reference aggregate function `sum(p.quantity)` is now computed in the outer block. In version 8, the query is semantically equivalent to the following (except that `Z` does not appear as part of the result set):

```
SELECT name,
       SUM(p.quantity) as Z,
       (SELECT Z FROM sales_order_items)
FROM product p
```

Since the outer block now computes an aggregate function, the outer block is treated as a grouped query and column *name* must appear in a `GROUP BY` clause in order to appear in the `SELECT` list.

## Aggregate functions and data types

There are some aggregate functions that have meaning only for certain kinds of data. For example, you can use `SUM` and `AVG` with numeric columns only.

However, you can use MIN to find the lowest value—the one closest to the beginning of the alphabet—in a character type column:

```
SELECT MIN(last_name)
FROM   contact
```

## Using count (\*)

The COUNT(\*) function does not require an expression as an argument because, by definition, it does not use information about any particular column. The COUNT(\*) function finds the total number of rows in a table. This statement finds the total number of employees:

```
SELECT COUNT(*)
FROM   employee
```

COUNT(\*) returns the number of rows in the specified table without eliminating duplicates. It counts each row separately, including rows that contain NULL.

Like other aggregate functions, you can combine count(\*) with other aggregates in the select list, with where clauses, and so on:

```
SELECT count(*), AVG(unit_price)
FROM   product
WHERE  unit_price > 10
```

count(*)	AVG(product.unit_price)
5	18.2

## Using aggregate functions with DISTINCT

The DISTINCT keyword is optional with SUM, AVG, and COUNT. When you use DISTINCT, duplicate values are eliminated before calculating the sum, average, or count.

For example, to find the number of different cities in which there are contacts, type:

```
SELECT count(DISTINCT city)
FROM   contact
```

count(distinct contact.city)
16

## Aggregate functions and NULL

Any NULLS in the column on which the aggregate function is operating are ignored for the purposes of the function except COUNT(\*), which includes them. If all the values in a column are NULL, COUNT(column\_name) returns 0.

If no rows meet the conditions specified in the WHERE clause, COUNT returns a value of 0. The other functions all return NULL. Here are examples:

```
SELECT COUNT (DISTINCT name)
FROM product
WHERE unit_price > 50
```

**count(DISTINCT name)**

---

0

```
SELECT AVG(unit_price)
FROM product
WHERE unit_price > 50
```

**AVG(product.unit\_price)**

---

( NULL )

# The GROUP BY clause: organizing query results into groups

The GROUP BY clause divides the output of a table into groups. You can GROUP BY one or more column names, or by the results of computed columns using numeric data types in an expression.

## Using GROUP BY with aggregate functions

A GROUP BY clause almost always appears in statements that include aggregate functions, in which case the aggregate produces a value for each group. These values are called **vector aggregates**. (Remember that a scalar aggregate is a single value produced by an aggregate function without a GROUP BY clause.)

### Example

The following query lists the average price of each kind of product:

```
SELECT name, AVG(unit_price) AS Price
FROM product
GROUP BY name
```

name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
...	...

The summary values (vector aggregates) produced by SELECT statements with aggregates and a GROUP BY appear as columns in each row of the results. By contrast, the summary values (scalar aggregates) produced by queries with aggregates and no GROUP BY also appear as columns, but with only one row. For example:

```
SELECT AVG(unit_price)
FROM product
```

**AVG(product.unit\_price)**

13.3

## Understanding GROUP BY

Understanding which queries are valid and which are not can be difficult when the query involves a GROUP BY clause. This section describes a way to think about queries with GROUP BY so that you may understand the results and the validity of queries better.

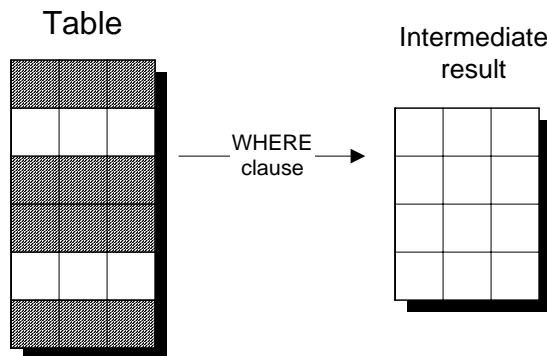
### How queries with GROUP BY are executed

Consider a single-table query of the following form:

```
SELECT select-list
FROM table
WHERE where-search-condition
GROUP BY group-by-expression
HAVING having-search-condition
```

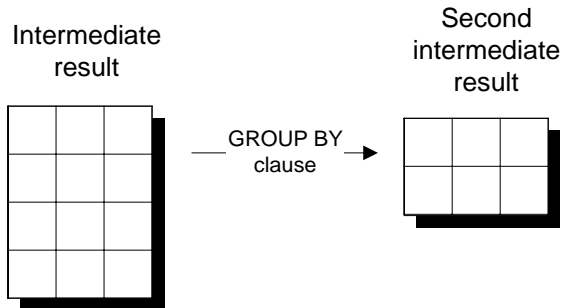
This query can be thought of as being executed in the following manner:

- 1 **Apply the WHERE clause** This generates an intermediate result that contains only some of the rows of the table.

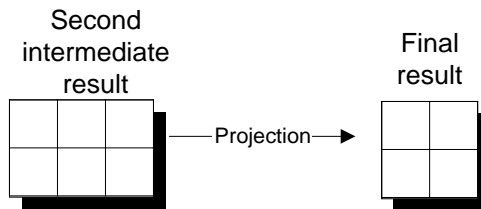


- 2 **Partition the result into groups** This action generates an intermediate result with one row for each group as dictated by the GROUP BY clause. Each generated row contains the *group-by-expression* for each group, and the computed aggregate functions in the *select-list* and *having-search-condition*.





- 3 **Apply the HAVING clause** Any rows from this second intermediate result that do not meet the criteria of the HAVING clause are removed at this point.
- 4 **Project out the results to display** This action takes from step 3 only those columns that need to be displayed in the result set of the query—that is, it takes only those columns corresponding to the expressions from the *select-list*.



This process makes requirements on queries with a GROUP BY clause:

- ◆ The WHERE clause is evaluated first. Therefore, any aggregate functions are evaluated only over those rows that satisfy the WHERE clause.
- ◆ The final result set is built from the second intermediate result, which holds the partitioned rows. The second intermediate result holds rows corresponding to the *group-by-expression*. Therefore, if an expression that is not an aggregate function appears in the *select-list*, then it must also appear in the *group-by-expression*. No function evaluation can be carried out during the projection step.
- ◆ An expression can be included in the *group-by-expression* but not in the *select-list*. It is projected out in the result.

## GROUP BY with multiple columns

You can list more than one expression in the GROUP BY clause in order to nest groups—that is, you can group a table by any combination of expressions.

The following query lists the average price of products, grouped first by name and then by size:

```
SELECT name, size, AVG(unit_price)
FROM product
GROUP BY name, size
```

name	size	AVG(product.un it_price)
Tee Shirt	Small	9
Tee Shirt	Medium	14
Tee Shirt	One size fits all	14
Baseball Cap	One size fits all	9.5
...	...	...

Columns in GROUP BY that are not in the select list

A Sybase extension to the SQL/92 standard that is supported by both Adaptive Server Enterprise and Adaptive Server Anywhere is to allow expressions to the GROUP BY clause that are not in the select list. For example, the following query lists the number of contacts in each city:

```
SELECT state, count(id)
FROM contact
GROUP BY state, city
```

## WHERE clause and GROUP BY

You can use a WHERE clause in a statement with GROUP BY. The WHERE clause is evaluated before the GROUP BY clause. Rows that do not satisfy the conditions in the WHERE clause are eliminated before any grouping is done. Here is an example:

```
SELECT name, AVG(unit_price)
FROM product
WHERE id > 400
GROUP BY name
```

Only the rows with *id* values of more than 400 are included in the groups that are used to produce the query results.

Example

The following query illustrates the use of WHERE, GROUP BY, and HAVING clauses in one query:

```
SELECT name, SUM(quantity)
FROM product
WHERE name LIKE '%shirt%'
GROUP BY name
HAVING SUM(quantity) > 100
```

name	SUM(product.quantity)
Tee Shirt	157

In this example:

- ◆ The WHERE clause includes only rows that have a name including the word *shirt* (Tee Shirt, Sweatshirt).
- ◆ The GROUP BY clause collects the rows with a common name.
- ◆ The SUM aggregate calculates the total quantity of products available for each group.
- ◆ The HAVING clause excludes from the final results the groups whose inventory totals do not exceed 100.

## The HAVING clause: selecting groups of data

The HAVING clause restricts the rows returned by a query. It sets conditions for the GROUP BY clause similar to the way in which WHERE sets conditions for the SELECT clause.

The HAVING clause search conditions are identical to WHERE search conditions except that WHERE search conditions cannot include aggregates, while HAVING search conditions often do. The example below is legal:

```
HAVING AVG(unit_price) > 20
```

But this example is not legal:

```
WHERE AVG(unit_price) > 20
```

Using HAVING  
with aggregate  
functions

The following statement is an example of simple use of the HAVING clause with an aggregate function.

To list those products available in more than one size or color, you need a query to group the rows in the *product* table by name, but eliminate the groups that include only one distinct product:

```
SELECT name
FROM product
GROUP BY name
HAVING COUNT(*) > 1
```

### **name**

---

Tee Shirt

Baseball Cap

Visor

Sweatshirt

☞ For information about when you can use aggregate functions in HAVING clauses, see "Where you can use aggregate functions" on page 209.

Using HAVING  
without aggregate  
functions

The HAVING clause can also be used without aggregates.

The following query groups the products, and then restricts the result set to only those groups for which the *name* starts with B.

```
SELECT name
FROM product
GROUP BY name
HAVING name LIKE 'B%'
```

**name**

---

Baseball Cap

More than one  
condition in  
HAVING

More than one condition can be included in the HAVING clause. They are combined with the AND, OR, or NOT operators, as in the following example.

To list those products available in more than one size or color, for which one version costs more than \$10, you need a query to group the rows in the *product* table by name, but eliminate the groups that include only one distinct product, and eliminate those groups for which the maximum unit price is under \$10.

```
SELECT name
FROM product
GROUP BY name
HAVING COUNT(*) > 1
AND MAX(unit_price) > 10
```

**name**

---

Tee Shirt

Sweatshirt

## The ORDER BY clause: sorting query results

The ORDER BY clause allows sorting of query results by one or more columns. Each sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed.

### A simple example

The following query returns results ordered by *name*:

```
SELECT id, name
FROM product
ORDER BY name
```

id	name
400	Baseball Cap
401	Baseball Cap
700	Shorts
600	Sweatshirt
...	...

### Sorting by more than one column

If you name more than one column in the ORDER BY clause, the sorts are nested.

The following statement sorts the shirts in the *product* table first by name in ascending order, then by quantity (descending) within each name:

```
SELECT id, name, quantity
FROM product
WHERE name like '%shirt%'
ORDER BY name, quantity DESC
```

id	name	quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75
301	Tee Shirt	54
...	...	...

### Using the column position

You can use the position number of a column in a select list instead of the column name. Column names and select list numbers can be mixed. Both of the following statements produce the same results as the preceding one.

```
SELECT id, name, quantity
FROM product
```

```
WHERE name like '%shirt%'
ORDER BY 2, 3 DESC

SELECT id, name, quantity
FROM product
WHERE name like '%shirt%'
ORDER BY 2, quantity DESC
```

Most versions of SQL require that ORDER BY items appear in the select list, but Adaptive Server Anywhere has no such restriction. The following query orders the results by quantity, although that column does not appear in the select list:

```
SELECT id, name
FROM product
WHERE name like '%shirt%'
ORDER BY 2, quantity DESC
```

ORDER BY and  
NULL

With ORDER BY, NULL comes before all other values, whether the sort order is ascending or descending.

ORDER BY and  
case sensitivity

The effects of an ORDER BY clause on mixed-case data depend on the database collation and case sensitivity specified when the database is created.

## Retrieving the first few rows of a query

You can limit the results of a query to the first few rows returned using the FIRST or TOP keywords. While you can use these with any query, they are most useful with queries that use the ORDER BY clause.

Examples

The following query returns information about the first employee sorted by last name:

```
SELECT FIRST *
FROM employee
ORDER BY emp_lname
```

The following query returns the first five employees sorted by last name comes earliest in the alphabet:

```
SELECT TOP 5 *
FROM employee
ORDER BY emp_lname
```

Restrictions on use  
of FIRST and TOP

FIRST and TOP are supported in the outermost SELECT block of a request. They should be used only in conjunction with an ORDER BY clause to ensure consistent results. FIRST is also supported in a subqueries that are either in a query's SELECT list, or are involved in a comparison predicate—the subquery is not part of a quantified predicate involving IN, ANY, SOME, or ALL. For example, the nested query

```
SELECT *
FROM sales_order_items
WHERE prod_id = (SELECT FIRST from product)
```

is supported, whereas

```
SELECT *
FROM sales_order_items
WHERE prod_id = ANY(SELECT FIRST from product)
```

is not. Unsupported instances of FIRST or TOP n may not trigger a syntax error, but will likely yield unexpected or unpredictable results. For this reason you should refrain from using FIRST or TOP in SQL constructs other than the two mentioned above. Specifically you should not specify FIRST or TOP in a derived table, view, or quantified subquery.

## ORDER BY and GROUP BY

You can use an ORDER BY clause to order the results of a GROUP BY in a particular way.

### Example

The following query finds the average price of each product and orders the results by average price:

```
SELECT name, AVG(unit_price)
FROM product
GROUP BY name
ORDER BY AVG(unit_price)
```

name	AVG(product.unit_price)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
...	...



## The UNION operation: combining queries

The UNION operator combines the results of two or more queries into a single result set.

By default, the UNION operator removes duplicate rows from the result set. If you use the ALL option, duplicates are not removed. The columns in the result set have the same names as the columns in the first table referenced. Any number of union operators may be used. For example:

```
x UNION y UNION z
```

By default, a statement containing multiple UNION operators is evaluated from left to right. Parentheses may be used to specify the order of evaluation.

For example, the following two expressions are not equivalent, due to the way that duplicate rows are removed from result sets:

```
x UNION ALL (y UNION z)
```

```
(x UNION ALL y) UNION z
```

In the first expression, duplicates are eliminated in the UNION between y and z. In the UNION between that set and x, duplicates are not eliminated. In the second expression, duplicates are included in the union between x and y, but are then eliminated in the subsequent union with z.

### Guidelines for UNION queries

The following guidelines apply union statements:

- ◆ **Same number of items in the select lists** All select lists in the union statement must have the same number of expressions (such as column names, arithmetic expressions, and aggregate functions). The following statement is invalid because the first select list is longer than the second:

```
-- This is an example of an invalid statement
SELECT stor_id, city, state
FROM stores
UNION
SELECT stor_id, city
FROM stores_east
```

- ◆ **Data types must match** Corresponding expressions in the SELECT lists must be of the same data type, or an implicit data conversion must be possible between the two data types, or an explicit conversion should be supplied.

For example, a UNION is not possible between a column of the CHAR data type and one of the INT data type, unless an explicit conversion is supplied. However, a union is possible between a column of the MONEY data type and one of the INT data type.

- ◆ **Column ordering** You must place corresponding expressions in the individual queries of a UNION statement in the same order, because UNION compares the expressions one to one in the order given in the individual queries in the SELECT clauses.
- ◆ **Multiple unions** You can string several UNION operations together, as in the following example:

```
SELECT city AS Cities
FROM contact
UNION
SELECT city
FROM customer
UNION
SELECT city
FROM employee
```

Only one ORDER BY clause is permitted, at the end of the statement. That is, no individual SELECT statement in a UNION query may contain an ORDER BY clause.

- ◆ **Column headings** The column names in the table resulting from a UNION are taken from the first individual query in the statement. If you want to define a new column heading for the result set, you must do so in the first query, as in the following example:

```
SELECT city AS Cities
FROM contact
UNION
SELECT city
FROM customer
```

In the following query, the column heading remains as *city*, as it is defined in the first query of the UNION statement.

```
SELECT city
FROM contact
UNION
SELECT city AS Cities
FROM customer
```

- ◆ You can use a single ORDER BY clause at the end of the list of queries, but you must use integers rather than column names, as in the following example:

```
SELECT Cities = city
FROM contact
UNION
SELECT city
FROM customer
ORDER BY 1
```

## Standards and compatibility

This section describes standards and compatibility aspects of the Adaptive Server Anywhere GROUP BY clause.

### GROUP BY and the SQL/92 standard

The SQL/92 standard for GROUP BY requires the following:

- ◆ A column used in an expression of the SELECT clause must be in the GROUP BY clause. Otherwise, the expression using that column is an aggregate function.
- ◆ A GROUP BY expression can only contain column names from the select list, but not those used only as arguments for vector aggregates.

The results of a standard GROUP BY with vector aggregate functions produce one row with one value per group.

Adaptive Server Anywhere and Adaptive Server Enterprise support extensions to HAVING that allow aggregate functions not in the select list and not in the GROUP BY clause.

### Compatibility with Adaptive Server Enterprise

Adaptive Server Enterprise supports several extensions to the GROUP BY clause that are not supported in Adaptive Server Anywhere. These include the following:

- ◆ **Non-grouped columns in the select list** Adaptive Server Enterprise permits column names in the select list that do not appear in the group by clause. For example, the following is valid in Adaptive Server Enterprise:

```
SELECT name, unit_price
FROM product
GROUP BY name
```

This syntax is not supported in Adaptive Server Anywhere.

- ◆ **Nested aggregate functions** The following query, which nests a vector aggregate inside a scalar aggregate, is valid in Adaptive Server Enterprise but not in Adaptive Server Anywhere:

```
SELECT MAX(AVG(unit_price))
FROM product
GROUP BY name
```

- ◆ **GROUP BY and ALL** Adaptive Server Anywhere does not support the use of ALL in the GROUP BY clause.
- ◆ **HAVING with no GROUP BY** Adaptive Server Anywhere does not support the use of HAVING with no GROUP BY clause unless all the expressions in the select and having clauses are aggregate functions. For example, the following query is valid in Adaptive Server Enterprise, but is not supported in Adaptive Server Anywhere:

```
--unsupported syntax  
  
SELECT unit_price  
FROM product  
HAVING COUNT(*) > 8
```

However, the following statement is valid in Adaptive Server Anywhere, because the functions MAX and COUNT are aggregate functions:

```
SELECT MAX(unit_price)  
FROM product  
HAVING COUNT(*) > 8
```

- ◆ **HAVING conditions** Adaptive Server Enterprise supports extensions to HAVING that allow non-aggregate functions not in the select list and not in the GROUP BY clause. Only aggregate functions of this type are allowed in Adaptive Server Anywhere.
- ◆ **DISTINCT with ORDER BY or GROUP BY** Adaptive Server Enterprise permits the use of columns in the ORDER BY or GROUP BY clause that do not appear in the select list, even in SELECT DISTINCT queries. This can lead to repeated values in the SELECT DISTINCT result set. Adaptive Server Anywhere does not support this behavior.
- ◆ **Column names in UNIONS** Adaptive Server Enterprise permits the use of columns in the ORDER BY clause in unions of queries. In Adaptive Server Anywhere, the ORDER BY clause must use an integer to mark the column by which the results are being ordered.

C H A P T E R 8

Joins: Retrieving Data from Several Tables

About this chapter

When you create a database, you normalize the data by placing information specific to different objects in different tables, rather than in one large table with many redundant entries.

A join operation recreates a larger table using the information from two or more tables (or views). Using different joins, you can construct a variety of these virtual tables, each suited to a particular task.

Before your start

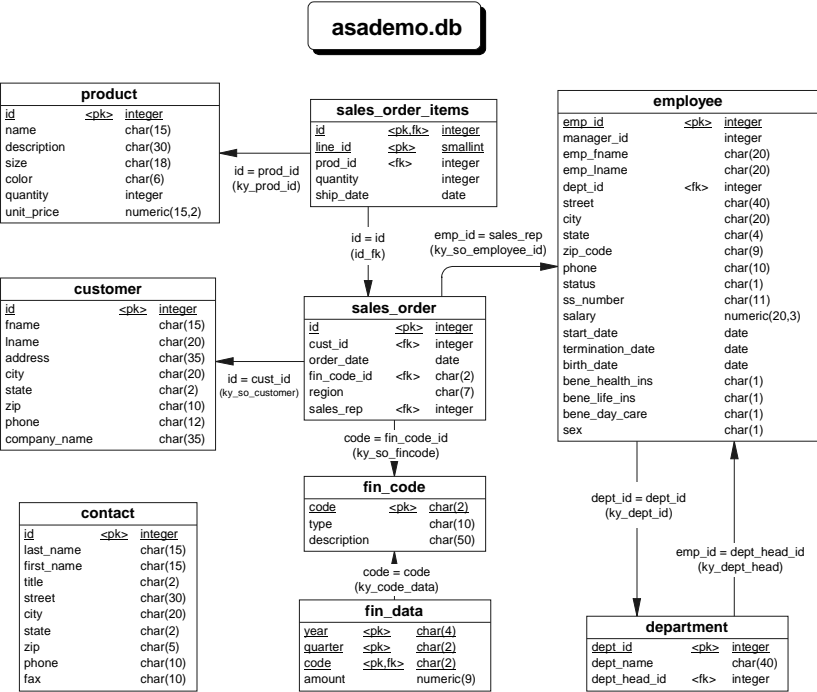
This chapter assumes some knowledge of queries and the syntax of the select statement. Information about queries appears in "Queries: Selecting Data from a Table" on page 183.

Contents

Topic	Page
Sample database schema	228
How joins work	229
Joins overview	230
Explicit join conditions (the ON phrase)	236
Cross joins	239
Inner and outer joins	241
Specialized joins	248
Natural joins	255
Key joins	259

# Sample database schema

This chapter makes frequent reference to the sample database. In the following diagram, the sample database is shown with the names of the foreign keys that relate the tables. The sample database is held in a file called *asademo.db*, and is located in your installation directory.



## How joins work

A relational database stores information about different types of objects in different tables. For example, information particular to employees appears in one table, and information that pertains to departments in another. The employee table contains information such as employee names and addresses. The department table contains information about one department, such as the name of the department and who the department head is.

Most questions can only be answered using a combination of information from the various tables. For example, you may want to answer the question "Who manages the Sales department?" To find the name of this person, you must identify the correct person using information from the department table, then look up that person's name in the employee table.

Joins are a means of answering such questions by forming a new virtual table that includes information from multiple tables. For example, you could create a list of the department heads by combining the information contained in the employee table and the department table. You specify which tables contain the information you need using the `FROM` clause.

To make the join useful, you must combine the correct columns of each table. To list department heads, each row of the combined table should contain the name of a department and the name of the employee who manages it. You control how columns are matched in the composite table by either specifying a particular type of join operation or using the `ON` phrase.

## Joins overview

A **join** is an operation that combines the rows in tables by comparing the values in specified columns. This section is an overview of Adaptive Server Anywhere join syntax. All of the concepts are explored in greater detail in later sections.

### The FROM clause

Use the FROM clause to specify which base tables, temporary tables, views or derived tables to join. The FROM clause can be used in SELECT or UPDATE statements.

**FROM** *table\_expression*, ...

where:

*table\_expression*:

- table*
- | *view*
- | *derived table*
- | *joined table*
- | ( *table\_expression*, ... )

*table or view*:

[ *userid.* ] *table-or-view-name* [ [ **AS** ] *correlation-name* ]

*derived table*:

( *select-statement* ) [ **AS** ] *correlation-name* [ ( *column-name*, ... ) ]

*joined table*:

*table\_expression* *join\_operator* *table\_expression* [ **ON** *join\_condition* ]

*join\_operator*:     [ **KEY** | **NATURAL** ] [ *join\_type* ] **JOIN**  
                     | **CROSS JOIN**

*join\_type*:

- INNER**
- | **FULL** [ **OUTER** ]
- | **LEFT** [ **OUTER** ]
- | **RIGHT** [ **OUTER** ]

#### Notes

You cannot use an ON phrase with CROSS JOIN.

☞ For the syntax of the ON phrase, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*.



## Join conditions

Tables can be joined using **join conditions**. A join condition is simply a search condition. It chooses a subset of rows from the joined tables based on the relationship between values in the columns. For example, the following query retrieves data from the *product* and *sales\_order\_items* tables.

```
SELECT *  
FROM product JOIN sales_order_items  
ON product.id = sales_order_items.prod_id
```

The join condition in this query is

```
product.id = sales_order_items.prod_id
```

This join condition means that rows can be combined in the result set only if they have the same product ID in both tables.

Join conditions can be explicit or generated. An **explicit join condition** is a join condition that is put in an ON phrase or a WHERE clause. The following query uses an ON phrase. It produces a cross product of the two tables (all combinations of rows), but with rows excluded if the id numbers do not match. The result is a list of customers with details of their orders.

```
SELECT *  
FROM customer JOIN sales_order  
ON sales_order.cust_id = customer.id
```

A **generated join condition** is a join condition that is automatically created when you specify KEY JOIN or NATURAL JOIN. In the case of a key join, the generated join condition is based on the foreign key relationships between the tables. In the case of a natural join, the generated join condition is based on columns that have the same name.

*Tip:*

Both key join syntax and natural join syntax are shortcuts: you get identical results from using the keyword JOIN *without* KEY or NATURAL, and then explicitly stating the same join condition in an ON phrase.

When you use an ON phrase with a key join or natural join, the join condition that is used is the **conjunction** of the explicitly specified join condition with the generated join condition. This means that the join conditions are combined with the keyword AND.

## Joined tables

Adaptive Server Anywhere supports the following classes of joined tables.

- ◆ **CROSS JOIN** A cross join of two tables produces all possible combinations of rows from the two tables. The size of the result set is the number of rows in the first table multiplied by the number of rows in the second table. A cross join is also called a cross product or Cartesian product. You cannot use an ON phrase with a cross join.
- ◆ **KEY JOIN (default)** A join condition is automatically generated based on the foreign key relationships that have been built into the database. Key join is the default when the JOIN keyword is used without specifying a join type and there is no ON phrase.
- ◆ **NATURAL JOIN** A join condition is automatically generated based on columns having the same name.
- ◆ **Join using an ON phrase** You specify an explicit join condition. When used with a key join or natural join, the join condition contains both the generated join condition and the explicit join condition. When used with the keyword JOIN without the keywords KEY or NATURAL, there is no generated join condition. You cannot use an ON clause with a cross join.

#### Inner and outer joins

Key joins, natural joins and joins with an ON clause may be qualified by specifying INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER. The default is INNER. When using the keywords LEFT, RIGHT or FULL, the keyword OUTER is optional.

In an inner join, each row in the result satisfies the join condition.

In a left or right outer join, all rows are preserved for one of the tables, and for the other table nulls are returned for rows that do not satisfy the join condition. For example, in a right outer join the right side is preserved and the left side is null-supplying.

In a full outer join, all rows are preserved for both of the tables, and nulls are supplied for rows that do not satisfy the join condition.

## Joining two tables


To understand how a simple inner join is computed, consider the following query. It answers the question: which product sizes have been ordered in the same quantity as the quantity in stock?

```
SELECT DISTINCT name, size,
               sales_order_items.quantity
FROM product JOIN sales_order_items
ON product.id = sales_order_items.prod_id
   AND product.quantity = sales_order_items.quantity
```

name	size	quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

You can interpret the query as follows. Note that this is a conceptual explanation of the processing of this query, used to illustrate the semantics of a query involving a join. It does not represent how Adaptive Server Anywhere actually computes the result set.

- ◆ Create a cross product of the *product* table and *sales\_order\_items* table. A cross product contains every combination of rows from the two tables.
- ◆ Exclude all rows where the product IDs are not identical (because of the join condition `product.id = sales_order_items.prod_id`).
- ◆ Exclude all rows where the quantity is not identical (because of the join condition `product.quantity = sales_order_items.quantity`).
- ◆ Create a result table with three columns: *product.name*, *product.size*, and *sales\_order\_items.quantity*.
- ◆ Exclude all duplicate rows (because of the `DISTINCT` keyword).

 For a description of how outer joins are computed, see "Outer joins" on page 241.

## Joining more than two tables

With Adaptive Server Anywhere, there is no fixed limit on the number of tables you can join.

When joining more than two tables, parentheses are optional. If you do not use parentheses, Adaptive Server Anywhere evaluates the statement from left to right. Therefore, `A JOIN B JOIN C` is equivalent to `(A JOIN B) JOIN C`. Another example:

```
SELECT *
FROM A JOIN B JOIN C JOIN D
```

is equivalent to

```
SELECT *
FROM ((A JOIN B) JOIN C) JOIN D
```

Whenever more than two tables are joined, the join involves table expressions. In the example `A JOIN B JOIN C`, the table expression `A JOIN B` is joined to `C`. This means, conceptually, that `A` and `B` are joined, and then the result is joined to `C`.

The order of joins is important if the table expression contains outer joins. For example, `A JOIN B LEFT OUTER JOIN C` is interpreted as `(A JOIN B) LEFT OUTER JOIN C`. This means that the table expression `A JOIN B` is joined to `C`. The table expression `A JOIN B` is preserved and table `C` is null-supplying.

☞ For more information about outer joins, see "Outer joins" on page 241.

☞ For more information about how Adaptive Server Anywhere performs a key join of table expressions, see "Key joins of table expressions" on page 262.

☞ For more information about how Adaptive Server Anywhere performs a natural join of table expressions, see "Natural joins of table expressions" on page 256.

## Join compatible data types

When you join two tables, the columns you compare must have the same or compatible data types.

☞ For more information about data type conversion in joins, see "Conversion when using comparison operators" on page 82 of the book *ASA SQL Reference Manual*.

## Using joins in delete, update, and insert statements

You can use joins in `DELETE`, `UPDATE` and `INSERT` statements, as well as in `SELECT` statements. You can update some cursors that contain joins if the option `ANSI_UPDATE_CONSTRAINTS` option is set to `OFF`. This is the default for databases created before Adaptive Server Anywhere version 7. For databases created in version 7 or later, the default is `ON`.

☞ For more information, see "ANSI\_UPDATE\_CONSTRAINTS option" on page 552 of the book *ASA Database Administration Guide*.

## Non-ANSI joins

Adaptive Server Anywhere supports ISO/ANSI standards for joins. It also supports the following non-standard joins:

- ◆ "Transact-SQL outer joins (`*=` or `=*`)" on page 245
- ◆ "Duplicate correlation names in joins (star joins)" on page 250
- ◆ "Key joins" on page 259

- ◆ "Natural joins" on page 255

You can use the REWRITE function to see the ANSI equivalent of a non-ANSI join.

🔗 For more information, see "REWRITE function" on page 172 of the book *ASA SQL Reference Manual*.

## Explicit join conditions (the ON phrase)

Instead of, or along with, a key or natural join, you can specify a join using an explicit join condition. You specify a join condition by inserting an ON phrase immediately after the join. The join condition always refers to the join immediately preceding it.

### Example

In the following query, the first ON phrase is used to join *sales\_order* to *customer*. The second ON phrase is used to join the table expression (*sales\_order JOIN customer*) to the base table *sales\_order\_item*.

```
SELECT *
FROM sales_order JOIN customer
    ON sales_order.cust_id = customer.id
JOIN sales_order_items
    ON sales_order_items.id = sales_order.id
```

### Tables that can be referenced

The tables that are referenced in an ON phrase must be part of the join that the ON phrase modifies. For example, the following is invalid:

```
FROM (A KEY JOIN B) JOIN (C JOIN D ON A.x = C.x)
```

The problem is that the join condition *A.x = C.x* references table *A*, which is not part of the join it modifies (in this case, *C JOIN D*).

However, as of the ANSI/ISO standard SQL99 and Adaptive Server Anywhere 7.0, there is an exception to this rule: if you use commas between table expressions, an ON condition of a join can reference a table that precedes it syntactically in the FROM clause. Therefore, the following is valid:

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

☞ For more information about commas, see "Commas" on page 239.

## Generated joins and the ON phrase

Key joins are the default if the keyword JOIN is used and no join type is specified—unless you use an ON phrase. If you use an ON phrase with an unspecified JOIN, key join is not the default and no generated join condition is applied.

For example, the following is a key join, because key join is the default when the keyword JOIN is used and there is no ON phrase:

```
SELECT *
FROM A JOIN B
```

The following is a join between table A and table B with the join condition `A.x = B.y`. It is *not* a key join.


```
SELECT *
FROM A JOIN B ON A.x = B.y
```

If you specify a **KEY JOIN** or **NATURAL JOIN** *and* use an **ON** phrase, the final join condition is the conjunction of the generated join condition and the explicit join condition(s). For example, the following statement has two join conditions: one generated because of the key join, and one explicitly stated in the **ON** phrase.

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y
```

If the join condition generated by the key join is `A.w = B.z`, then the following statement is equivalent:

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z
```

 For more information about key joins, see "Key joins" on page 259.

## Types of explicit join conditions

Most join conditions are based on equality, and so are called **equijoins**. For example,


```
SELECT *
FROM department JOIN employee
  ON department.dept_id = employee.dept_id
```

However, you do not have to use equality (`=`) in a join condition. You can use any search condition, such as conditions containing **LIKE**, **SOUNDEX**, **BETWEEN**, **>** (greater than), and **!=** (not equal to).

### Example

The following example answers the question: For which products has someone ordered more than the quantity in stock?

```
SELECT DISTINCT product.name
FROM product JOIN sales_order_items
  ON product.id = sales_order_items.prod_id
  AND product.quantity > sales_order_items.quantity
```

 For more information about search conditions, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*.


## Using the WHERE clause for join conditions

Except when using outer joins, you can specify join conditions in the WHERE clause instead of the ON phrase. However, you should be aware that there may be semantic differences between the two if the query contains outer joins.


The ON phrase is part of the FROM clause, and so is processed before the WHERE clause. This does not make a difference to results except in the case of outer joins, where using the WHERE clause can convert the join to an inner join.

When deciding whether to put join conditions in an ON phrase or WHERE clause, keep the following rules in mind:

- ◆ When you specify an outer join, putting a join condition in the WHERE clause may convert the outer join to an inner join.

 For more information about the WHERE clause and outer joins, see "Outer joins and join conditions" on page 243.

- ◆ Conditions in an ON phrase can only refer to tables that are in the table expressions joined by the associated JOIN. However, conditions in a WHERE clause can refer to any tables, even if they are not part of the join.
- ◆ You cannot use an ON phrase with the keywords CROSS JOIN, but you can always use a WHERE clause.
- ◆ When join conditions are in an ON phrase, key join is not the default. However, key join can be the default if join conditions are put in a WHERE clause.

 For more information about the conditions under which key join is the default, see "When key join is the default" on page 259.

In the examples in this documentation, join conditions are put in an ON phrase. In examples using outer joins, this is necessary. In other cases it is done to make it obvious that they are join conditions and not general search conditions.



## Cross joins

A cross join of two tables produces all possible combinations of rows from the two tables. A cross join is also called a cross product or Cartesian product.

Each row of the first table appears once with each row of the second table. Hence, the number of rows in the result set is the product of the number of rows in the first table and the number of rows in the second table, minus any rows that are omitted because of restrictions in a WHERE clause.

You cannot use an ON phrase with cross joins. However, you can put restrictions in a WHERE clause.

Inner and outer modifiers do not apply to cross joins

Except in the presence of additional restrictions in the WHERE clause, all rows of both tables always appear in the result set of cross joins. Thus, the keywords INNER, LEFT OUTER and RIGHT OUTER are not applicable to cross joins.

For example, the following statement joins two tables.

```
SELECT *  
FROM A CROSS JOIN B
```

The result set from this query includes all columns in A and all columns in B. There is one row in the result set for each combination of a row in A and a row in B. If A has  $n$  rows and B has  $m$  rows, the query returns  $n \times m$  rows.

## Commas

A comma works like a join operator, but is not one. A comma creates a cross product exactly as the keyword CROSS JOIN does. However, join keywords create table expressions, and commas create lists of table expressions.

In the following simple inner join of two tables, a comma and the keywords CROSS JOIN are equivalent:

```
Select *  
FROM A CROSS JOIN B CROSS JOIN C  
WHERE A.x = B.y
```

and

```
Select *  
FROM A, B, C  
WHERE A.x = B.y
```

Generally, you can use a comma instead of the keywords `CROSS JOIN`. The comma syntax is equivalent to cross join syntax, except in the case of generated join conditions in table expressions using commas.

☞ For information about how commas work with generated join conditions, see "Key joins of table expressions" on page 262.

☞ In the syntax of star joins, commas have a special use. For more information, see "Duplicate correlation names in joins (star joins)" on page 250.

## Inner and outer joins

The keywords **INNER**, **LEFT OUTER**, **RIGHT OUTER**, and **FULL OUTER** may be used to modify key joins, natural joins, and joins with an **ON** phrase. The default is **INNER**. The keyword **OUTER** is optional. These modifiers do not apply to cross joins.

### Inner joins

By default, joins are **inner joins**. This means that rows are included in the result set only if they satisfy the join condition.

#### Example

For example, each row of the result set of the following query contains the information from one *customer* row and one *sales\_order* row, satisfying the key join condition. If a particular customer has placed no orders, the condition is not satisfied and the result set does not contain the row corresponding to that customer.

```
SELECT fname, lname, order_date
FROM customer KEY INNER JOIN sales_order
ORDER BY order_date
```

fname	lname	order_date
Hardy	Mums	1/2/00
Aram	Najarian	1/3/00
Tommie	Wooten	1/3/00
Alfredo	Margolis	1/6/00
...	...	...

Because inner joins and key joins are the defaults, you obtain the same result using the following **FROM** clause.

```
FROM customer JOIN sales_order
```

### Outer joins

A left or right **outer join** of two tables preserves all the rows in one table, and supplies nulls for the other table when it does not meet the join condition. A **left outer join** preserves every row in the left-hand table, and a **right outer join** preserves every row in the right-hand table. In a **full outer join**, all rows from both tables are preserved.

The table expressions on either side of a left or right outer join are referred to as **preserved** and **null-supplying**. In a left outer join, the left-hand table expression is preserved and the right-hand table expression is null-supplying.

☞ For information about creating outer joins with Transact-SQL syntax, see "Transact-SQL outer joins (\*= or =\*)" on page 245.

### Example

For example, the following statement includes all customers, whether or not they have placed an order. If a particular customer has placed no orders, each column in the result that corresponds to order information contains the NULL value.

```
SELECT lname, order_date, city
FROM customer LEFT OUTER JOIN sales_order
    ON customer.id = sales_order.cust_id
WHERE customer.state = 'NY'
ORDER BY order_date
```

lname	order_date	city
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
...	...	...

You can interpret the outer join in this statement as follows. Note that this is a conceptual explanation, and does not represent how Adaptive Server Anywhere actually computes the result set.

- ◆ Return one row for every sales order placed by a customer. More than one row is returned when the customer placed two or more sales orders, because a row is returned for each sales order. This is the same result as an inner join. The ON condition is used to match customer and sales order rows. The WHERE clause is not used for this step.
- ◆ Include one row for every customer who has not placed any sales orders. This ensures that every row in the customer table is included. For all of these rows, the columns from *sales\_order* are filled with nulls. These rows are added because the keyword OUTER is used, and would not have appeared in an inner join. Neither the ON condition nor the WHERE clause is used for this step.
- ◆ Exclude every row where the customer does not live in New York, using the WHERE clause.

## Outer joins and join conditions

A common mistake with outer joins is the placement of the join condition. In most cases, if you place restrictions on the null-supplying table in a WHERE clause, the join is equivalent to an inner join.

The reason for this is that most search conditions cannot evaluate to TRUE when any of their inputs are NULL. The WHERE clause restriction on the null-supplying table compares values to null, resulting in the elimination of the row from the result set. The rows in the preserved table are not preserved and so the join is an inner join.

The exception to this is comparisons that can evaluate to true when any of their inputs are NULL. These include IS NULL, IS UNKNOWN, IS FALSE, IS NOT TRUE, IS NULL, and expressions involving ISNULL or COALESCE.

### Example

For example, the following statement computes a left outer join.

```
SELECT *  
FROM customer KEY LEFT OUTER JOIN sales_order  
ON sales_order.order_date < '2000-01-03'
```

In contrast, the following statement creates an inner join.

```
SELECT lname, order_date  
FROM customer KEY LEFT OUTER JOIN sales_order  
WHERE sales_order.order_date < '2000-01-03'
```

The first of these two statements can be thought of as follows: First, left-outer join the *customer* table to the *sales\_order* table. The result set includes every row in the customer table. For those customers who have no orders prior to January 3 2000, fill the sales order fields with nulls.

In the second statement, first left-outer join *customer* and *sales\_order*. The result set includes every row in the customer table. For those customers who have no orders, fill the sales order fields with nulls. Next, apply the WHERE condition by selecting only those rows in which the customer has placed an order since January 3 2000. For those customers who have not placed orders, these values are NULL. Comparing any value to NULL evaluates to UNKNOWN. Hence, these rows are eliminated and the statement reduces to an inner join.

☞ For more information about search conditions, see "Search conditions" on page 24 of the book *ASA SQL Reference Manual*.

## Understanding complex outer joins

The order of joins is important when a query includes table expressions using outer joins. For example, `A JOIN B LEFT OUTER JOIN C` is interpreted as `(A JOIN B) LEFT OUTER JOIN C`. This means that the table expression `(A JOIN B)` is joined to `C`. The table expression `(A JOIN B)` is preserved and table `C` is null-supplying.

Consider the following statement, in which `A`, `B` and `C` are tables:

```
SELECT *
FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C
```

To understand this statement, first remember that Adaptive Server Anywhere evaluates statements from left to right, adding parentheses. This results in

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C
```

Next, you may want to convert the right outer join to a left outer join so that both joins are the same type. To do this, simply reverse the position of the tables in the right outer join, resulting in:

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B)
```

`A` is the preserved table and `B` is the null-supplying table for the nested outer join. `C` is the preserved table for the first outer join.

You can interpret this join as follows:

- ◆ Join `A` to `B`, preserving all rows in `A`.
- ◆ Next, join `C` to the results of the join of `A` and `B`, preserving all rows in `C`.

The join does not have an `ON` phrase, and so is by default a key join. The way Adaptive Server Anywhere generates join conditions for this type of join is explained in "Key joins of table expressions that do not contain commas " on page 263 on page 263.

In addition, the join condition for an outer join must only include tables that have previously been referenced in the `FROM` clause. This restriction is according to the ANSI/ISO standard, and is enforced to avoid ambiguity. For example, the following two statements are syntactically incorrect, because `C` is referenced in the join condition before the table itself is referenced.

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C
```

and

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C
```

## Outer joins of views and derived tables

Outer joins can also be specified for views and derived tables.

The statement

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x)
```

can be interpreted as follows:

- ◆ Compute the view *V*.
- ◆ Join all the rows from the computed view *V* with *A* by preserving all the rows from *V*, using the join condition  $V.x = A.x$ .

### Example

The following example defines a view called *V* that returns the employee IDs and department names of women who make over \$60 000.

```
CREATE VIEW V AS
SELECT employee.emp_id, dept_name
FROM employee JOIN department
ON employee.dept_id = department.dept_id
WHERE sex = 'F' and salary > 60000
```

Next, use this view to add a list of the departments where the women work and the regions where they have sold. The view *V* is preserved and *sales\_order* is null-supplying.

```
SELECT DISTINCT V.emp_id, region, V.dept_name
FROM V LEFT OUTER JOIN sales_order
ON V.emp_id = sales_order.sales_rep
```

emp_id	region	dept_name
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
...	...	...

## Transact-SQL outer joins (\*= or =\*)

In accordance with ANSI/ISO SQL standards, Adaptive Server Anywhere supports the LEFT OUTER, RIGHT OUTER, and FULL OUTER keywords. For compatibility with Adaptive Server Enterprise prior to version 12, Adaptive Server Anywhere also supports the Transact-SQL counterparts of these keywords, \*= and =\*. However, there are some limitations and potential problems with the Transact-SQL semantics.

For a detailed discussion of Transact-SQL outer joins, see the whitepaper *Semantics and Compatibility of Transact-SQL Outer Joins*, which is available at <http://www.sybase.com/detail?id=1017447>.

**Warning:**

When you are creating outer joins, do *not* mix \*= syntax with ON phrase syntax. This also applies to views that are referenced in the query.

In the Transact-SQL dialect, you create outer joins by supplying a comma-separated list of tables in the FROM clause, and using the special operators \*= or =\* in the WHERE clause. In Adaptive Server Enterprise prior to version 12, the join condition must appear in the WHERE clause (ON was not supported).

### Example

For example, the following left outer join lists all customers and finds their order dates (if any):

```
SELECT fname, lname, order_date
FROM customer, sales_order
WHERE customer.id *= sales_order.cust_id
ORDER BY order_date
```

This statement is equivalent to the following statement, in which ANSI/ISO syntax is used:

```
SELECT fname, lname, order_date
FROM customer LEFT OUTER JOIN sales_order
ON customer.id = sales_order.cust_id
ORDER BY order_date
```

## Transact-SQL outer join limitations

There are several restrictions for Transact-SQL outer joins:

- ◆ If you specify an outer join and a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The qualification in the query does not exclude rows from the result set, but rather affects the values that appear in the rows of the result set. For rows that do not meet the qualification, a null value appears in the null-supplying table.
- ◆ You cannot mix ANSI/ISO SQL syntax and Transact-SQL outer join syntax in a single query. If a view is defined using one dialect for an outer join, you must use the same dialect for any outer-join queries on that view.
- ◆ A null-supplying table cannot participate in both a Transact-SQL outer join and a regular join or two outer joins. For example, the following WHERE clause is not allowed, because table S violates this limitation.



```
WHERE R.x *= S.x  
AND S.y = T.y
```

When you cannot rewrite your query to avoid using a table in both an outer join and a regular join clause, you must divide your statement into two separate queries, or use only ANSI/ISO SQL syntax.

- ◆ You cannot use a subquery that contains a join condition involving the null-supplying table of an outer join. For example, the following WHERE clause is not allowed:

```
WHERE R.x *= S.y  
AND EXISTS ( SELECT *  
              FROM T  
              WHERE T.x = S.x )
```

### Using views with Transact-SQL outer joins

If you define a view with an outer join, and then query the view with a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The query returns all rows from the null-supplying table. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through views that contain outer joins:

- ◆ INSERT and DELETE statements are not allowed on outer join views.
- ◆ UPDATE statements are allowed on outer join views. If the view is defined WITH CHECK option, the update fails if any of the affected columns appears in the WHERE clause in an expression that includes columns from more than one table.

### How NULL affects Transact-SQL joins

NULL values in tables or views being joined never match each other in a Transact-SQL outer join. The result of comparing a NULL value with any other NULL value is FALSE.

## Specialized joins

This section describes how to create some specialized joins such as self-joins, star joins, and joins using derived tables.

### Self-joins

In a **self-join**, a table is joined to itself by referring to the same table using a different correlation name.

#### Example 1

The following self-join produces a list of pairs of employees. Each employee name appears in combination with every employee name.

```
SELECT a.emp_fname, a.emp_lname,  
       b.emp_fname, b.emp_lname  
FROM employee AS a CROSS JOIN employee AS b
```

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
...	...	...	...

Since the *employee* table has 75 rows, this join contains  $75 \times 75 = 5\,625$  rows. It includes, as well, rows that list each employee with themselves. For example, it contains the row

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney

If you want to exclude rows that contain the same name twice, add the join condition that the employee IDs should not be equal to each other.

```
SELECT a.emp_fname, a.emp_lname,  
       b.emp_fname, b.emp_lname  
FROM employee AS a CROSS JOIN employee AS b  
WHERE a.emp_id != b.emp_id
```

Without these duplicate rows, the join contains  $75 \times 74 = 5\,550$  rows.

This new join contains rows that pair each employee with every other employee, but because each pair of names can appear in two possible orders, each pair appears twice. For example, the result of the above join contains the following two rows.

emp_fname	emp_lname	emp_fname	emp_lname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

If the order of the names is not important, you can produce a list of the  $(75 \times 74)/2 = 2\,775$  unique pairs.

```
SELECT a.emp_fname, a.emp_lname,
       b.emp_fname, b.emp_lname
FROM employee AS a CROSS JOIN employee AS b
WHERE a.emp_id < b.emp_id
```

This statement eliminates duplicate lines by selecting only those rows in which the *emp\_id* of employee *a* is less than that of employee *b*.

## Example 2

The following self-join uses the correlation names *report* and *manager* to distinguish two instances of the *employee* table, and creates a list of employees and their managers.

```
SELECT report.emp_fname, report.emp_lname,
       manager.emp_fname, manager.emp_lname
FROM employee AS report JOIN employee AS manager
     ON (report.manager_id = manager.emp_id)
ORDER BY report.emp_lname, report.emp_fname
```

This statement produces the result shown partially below. The employee names appear in the two left-hand columns, and the names of their managers are on the right.

emp_fname	emp_lname	emp_fname	emp_lname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
...	...	...	...

## Duplicate correlation names in joins (star joins)

The reason for using duplicate table names is to create a **star join**. In a star join, one table or view is joined to several others.

To create a star join, you use the same table name, view name, or correlation name more than once in the FROM clause. This is an extension to the ANSI/ISO SQL standard. The ability to use duplicate names does not add any additional functionality, but it makes it much easier to formulate certain queries.

The duplicate names must be in different joins for the syntax to make sense. When a table name or view name is used twice in the same join, the second instance is ignored. For example, FROM A ,A and FROM A CROSS JOIN A are both interpreted as FROM A.

The following example, in which A, B and C are tables, is valid in Adaptive Server Anywhere. In this example, the same instance of table A is joined both to B and C. Note that a comma is required to separate the joins in a star join. The use of a comma in star joins is specific to the syntax of star joins.

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
      A LEFT OUTER JOIN C ON A.y = C.y
```

The next example is equivalent.

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
      C RIGHT OUTER JOIN A ON A.y = C.y
```

Both of these are equivalent to the following standard ANSI/ISO syntax. (The parentheses are optional.)

```
SELECT *
FROM (A LEFT OUTER JOIN B ON A.x = B.x)
LEFT OUTER JOIN C ON A.y = C.y
```

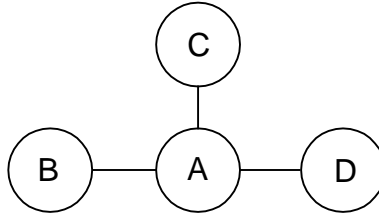
In the next example, table A is joined to three tables: B, C and D.

```
SELECT *
FROM A JOIN B ON A.x = B.x,
      A JOIN C ON A.y = C.y,
      A JOIN D ON A.w = D.w
```

This is equivalent to the following standard ANSI/ISO syntax. (The parentheses are optional.)

```
SELECT *
FROM ((A JOIN B ON A.x = B.x)
JOIN C ON A.y = C.y)
JOIN D ON A.w = D.w
```

With complex joins, it can help to draw a diagram. The previous example can be described by the following diagram, which illustrates that tables B, C and D are joined via table A.



*Note*

You can use duplicate table names only if the EXTENDED\_JOIN\_SYNTAX option is ON (the default).

For more information, see the "EXTENDED\_JOIN\_SYNTAX option" on page 567 of the book *ASA Database Administration Guide*.

Example 1

Create a list of the names of the customers who have placed orders with Rollin Overbey. Notice that one of the tables in the FROM clause, *employee*, does not contribute any columns to the results. Nor do any of the columns that are joined—such as *customer.id* or *employee.id*—appear in the results. Nonetheless, this join is possible only using the *employee* table in the FROM clause.

```

SELECT customer.fname, customer.lname,
       sales_order.order_date
FROM   sales_order KEY JOIN customer,
       sales_order KEY JOIN employee
WHERE  employee.emp_fname = 'Rollin'
       AND employee.emp_lname = 'Overbey'
ORDER BY sales_order.order_date
  
```

fname	lname	order_date
Tommie	Wooten	1/3/00
Michael	Agliori	1/8/00
Salton	Pepper	1/17/00
Tommie	Wooten	1/23/00
...	...	...

Following is the equivalent statement in standard ANSI/ISO syntax:

```

SELECT customer.fname, customer.lname,
       sales_order.order_date
FROM sales_order JOIN customer
  
```

```
        ON sales_order.cust_id = customer.id
JOIN employee
        ON sales_order.sales_rep = employee.emp_id
WHERE employee.emp_fname = 'Rollin'
      AND employee.emp_lname = 'Overbey'
ORDER BY sales_order.order_date
```

### Example 2

This example answers the question: How much of each product has each customer ordered, and who is the manager of the salesperson who took the order?

To answer the question, start by listing the information you need to retrieve. In this case, it is product, quantity, customer name, and manager name. Next, list the tables that hold this information. They are *product*, *sales\_order\_items*, *customer*, and *employee*. When you look at the structure of the sample database (see "Sample database schema" on page 228), you will notice that these tables are all related through the *sales\_order* table. You can create a star join on the *sales\_order* table to retrieve the information from the other tables.

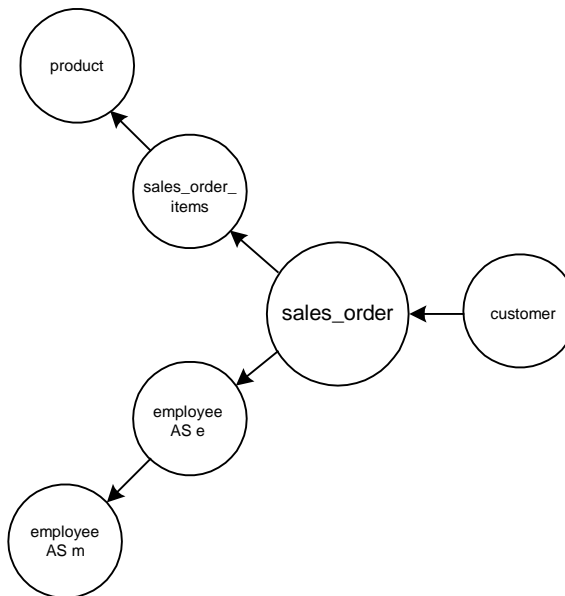
In addition, you need to create a self-join in order to get the name of the manager, because the *employee* table contains ID numbers for managers and the names of all employees, but not a column listing only manager names. For more information, see "Self-joins" on page 248 on page 248.

The following statement creates a star join around the *sales\_order* table. The joins are all outer joins so that the result set will include all customers. Some customers have not placed orders, so the other values for these customers are NULL. The columns in the result set are customer, product, quantity ordered, and the name of the manager of the salesperson.

```
SELECT customer.fname, product.name,
       SUM(sales_order_items.quantity), m.emp_fname
FROM   sales_order
       KEY RIGHT OUTER JOIN customer,
       sales_order
       KEY LEFT OUTER JOIN sales_order_items
       KEY LEFT OUTER JOIN product,
       sales_order
       KEY LEFT OUTER JOIN employee AS e
       LEFT OUTER JOIN employee AS m
       ON (e.manager_id = m.emp_id)
WHERE  customer.state = 'CA'
GROUP BY customer.fname, product.name, m.emp_fname
ORDER BY SUM(sales_order_items.quantity) DESC,
         customer.fname
```

fname	name	SUM(sales_order_items.quantity)	emp_fname
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
...	...	...	...

Following is a diagram of the tables in this star join. The arrows indicate the directionality (left or right) of the outer joins. As you can see, the complete list of customers is maintained throughout all the joins.



The following standard ANSI/ISO syntax is equivalent to the star join in Example 2.

```
SELECT customer.fname, product.name,
       SUM(sales_order_items.quantity), m.emp_fname
FROM sales_order LEFT OUTER JOIN sales_order_items
  ON sales_order.id = sales_order_items.id
LEFT OUTER JOIN product
  ON sales_order_items.prod_id = product.id
LEFT OUTER JOIN employee as e
  ON sales_order.sales_rep = e.emp_id
LEFT OUTER JOIN employee as m
  ON e.manager_id = m.emp_id
RIGHT OUTER JOIN customer
  ON sales_order.cust_id = customer.id
WHERE customer.state = 'CA'
GROUP BY customer.fname, product.name, m.emp_fname
ORDER BY SUM(sales_order_items.quantity) DESC,
customer.fname
```

## Joins involving derived tables

Derived tables allow you to nest queries within a FROM clause. With derived tables, you can perform grouping of groups, or you can construct a join with a group, without having to create a view.

In the following example, the inner SELECT statement (enclosed in parentheses) creates a derived table, grouped by customer id values. The outer SELECT statement assigns this table the correlation name *sales\_order\_counts* and joins it to the *customer* table using a join condition.

```
SELECT lname, fname, number_of_orders
FROM customer JOIN
  ( SELECT cust_id, count(*)
    FROM sales_order
    GROUP BY cust_id )
  AS sales_order_counts (cust_id, number_of_orders)
  ON (customer.id = sales_order_counts.cust_id)
WHERE number_of_orders > 3
```

The result is a table of the names of those customers who have placed more than three orders, including the number of orders each has placed.

☞ For an explanation of key joins of derived tables, see "Key joins of views and derived tables" on page 268.

☞ For an explanation of natural joins of derived tables, see "Natural joins of views and derived tables" on page 257.

☞ For an explanation of outer joins of derived tables, see "Outer joins of views and derived tables" on page 245.



## Natural joins

When you specify a natural join, Adaptive Server Anywhere generates a join condition based on columns with the same name. For this to work in a natural join of base tables, there must be at least one pair of columns with the same name, with one column from each table. If there is no common column name, an error is issued.

If table A and table B have one column name in common, and that column is called *x*, then

```
SELECT *
FROM A NATURAL JOIN B
```

is equivalent to the following:

```
SELECT *
FROM A JOIN B
ON A.x = B.x
```

If table A and table B have two column names in common, and they are called *a* and *b*, then `A NATURAL JOIN B` is equivalent to the following:

```
A JOIN B
ON A.a = B.a
AND A.b = B.b
```

### Example

For example, you can join the *employee* and *department* tables using a natural join because they have a column name in common, the *dept\_id* column.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee NATURAL JOIN department
ORDER BY dept_name, emp_lname, emp_fname
```

emp_fname	emp_lname	dept_name
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
...	...	...

The following statement is equivalent. It explicitly specifies the join condition that was generated in the previous example.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
    ON (employee.dept_id = department.dept_id)
ORDER BY dept_name, emp_lname, emp_fname
```

## Natural joins with an ON phrase

When you specify a NATURAL JOIN *and* put a join condition in an ON phrase, the result is the conjunction of the two join conditions.

For example, the following two queries are equivalent. In the first query, Adaptive Server Anywhere generates the join condition `employee.dept_id = department.dept_id`. The query also contains an explicit join condition.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee NATURAL JOIN department
    ON employee.manager_id = department.dept_head_id
```

The next query is equivalent. In it, the natural join condition that was generated in the previous query is specified in the ON phrase.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
    ON employee.manager_id = department.dept_head_id
    AND employee.dept_id = department.dept_id
```

## Natural joins of table expressions

When there is a multiple-table expression on at least one side of a natural join, Adaptive Server Anywhere generates a join condition by comparing the set of columns for each side of the join operator, and looking for columns that have the same name.

For example, in the statement

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D)
```

there are two table expressions. The column names in the table expression `A JOIN B` are compared to the column names in the table expression `C JOIN D`, and a join condition is generated for each unambiguous pair of matching column names. An *unambiguous pair of matching columns* means that the column name occurs in both table expressions, but does not occur twice in the same table expression.

If there is a pair of ambiguous column names, an error is issued. However, a column name may occur twice in the same table expression, as long as it doesn't also match the name of a column in the other table expression.

## Natural joins of lists

When a list of table expressions is on at least one side of a natural join, a separate join condition is generated for each table expression in the list.

Consider the following tables:

- ◆ table A consists of columns called *a*, *b* and *c*
- ◆ table B consists of columns called *a* and *d*
- ◆ table C consists of columns called *d* and *c*

In this case, the join (A,B) NATURAL JOIN C causes Adaptive Server Anywhere to generate two join conditions:

```
ON A.c = C.c
AND B.d = C.d
```

If there is no common column name for A-C or B-C, an error is issued.

If table C consists of columns *a*, *d*, and *c*, then the join (A,B) NATURAL JOIN C is invalid. The reason is that column *a* appears in all three tables, and so the join is ambiguous.

## Example

The following example answers the question: for each sale, provide information about what was sold and who sold it.

```
SELECT *
FROM (employee KEY JOIN sales_order)
     NATURAL JOIN (sales_order_items KEY JOIN product)
```

This is equivalent to

```
SELECT *
FROM (employee KEY JOIN sales_order)
     JOIN (sales_order_items KEY JOIN product)
       ON sales_order.id = sales_order_items.id
```

## Natural joins of views and derived tables

An extension to the ANSI/ISO SQL standard is that you can specify views or derived tables on either side of a natural join. In the following statement,

```
SELECT *
FROM View1 NATURAL JOIN View2
```

the columns in *View1* are compared to the columns in *View2*. If, for example, a column called *emp\_id* is found to occur in both views, and there are no other columns that have identical names, then the generated join condition is (*View1.emp\_id* = *View2.emp\_id*).

### Example

The following example illustrates that a view used in a natural join can include expressions, and not just columns, and they are treated the same way in the natural join. First, create the view *V* with a column called *x*, as follows:

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R
```

Next, create a natural join of the view to a derived table. The derived table has a correlation name *T* with a column called *x*.

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x)
```

This join is equivalent to the following:

```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x)
```

## Key joins

When you specify a key join, Adaptive Server Anywhere generates a join condition based on the foreign key relationships in the database. To use a key join, there must be a foreign key relationship between the tables, or an error is issued.

The key join is a Sybase extension to the ANSI/ISO SQL standard. It does not provide any greater functionality, but it makes it easier to formulate certain queries.

When key join is the default

Key join is the default in Adaptive Server Anywhere when all of the following apply:

- ◆ the keyword JOIN is used.
- ◆ the keywords CROSS, NATURAL or KEY are *not* specified.
- ◆ there is no ON phrase.

Example

For example, the following query is a simple key join that joins the tables *product* and *sales\_order\_items* based on the foreign key relationship in the database.

```
SELECT *  
FROM product KEY JOIN sales_order_items
```

The next query is equivalent. It leaves out the word KEY, but by default a JOIN without an ON phrase is a KEY JOIN.

```
SELECT *  
FROM product JOIN sales_order_items
```

The next query is also equivalent, because the join condition specified in the ON phrase happens to be the same as the join condition that Adaptive Server Anywhere generates for these tables based on their foreign key relationship in the sample database.

```
SELECT *  
FROM product JOIN sales_order_items  
ON sales_order_items.prod_id = product.id
```

## Key joins with an ON phrase

When you specify a KEY JOIN *and* put a join condition in an ON phrase, the result is the conjunction of the two join conditions. For example,

```
SELECT *  
FROM A KEY JOIN B  
ON A.x = B.y
```

If the join condition generated by the key join of A and B is  $A.w = B.z$ , then this query is equivalent to

```
SELECT *  
FROM A JOIN B  
ON A.x = B.y AND A.w = B.z
```

## Key joins when there are multiple foreign key relationships

When Adaptive Server Anywhere attempts to generate a join condition based on a foreign key relationship, it sometimes finds more than one relationship. In these cases, Adaptive Server Anywhere determines which foreign key relationship to use by matching the role name of the foreign key to the correlation name of the primary key table that the foreign key references.

✍ The following sections describe how Adaptive Server Anywhere generates join conditions for key joins. This information is summarized in "Rules describing the operation of key joins" on page 270.

### Correlation name and role name

A **correlation name** is the name of a table or view that is used in the FROM clause of the query—either its original name, or an alias that is defined in the FROM clause.

A **role name** is the name of the foreign key. It must be unique for a given foreign (child) table.

If you do not specify a role name for a foreign key, the name is assigned as follows:

- ◆ If there is no foreign key with the same name as the primary table name, the primary table name is assigned as the role name.
- ◆ If the primary table name is already being used by another foreign key, the role name is the primary table name concatenated with a zero-padded three-digit number unique to the foreign table.

If you don't know the role name of a foreign key, you can find it in Sybase Central by expanding the database folder in the left pane. Every table in the database has a sub-folder called *Foreign Keys*. Click on it and the foreign keys appear in the right pane.

✍ See "Sample database schema" on page 228 for a diagram that includes the role names of all foreign keys in the sample database.

### Generating join conditions

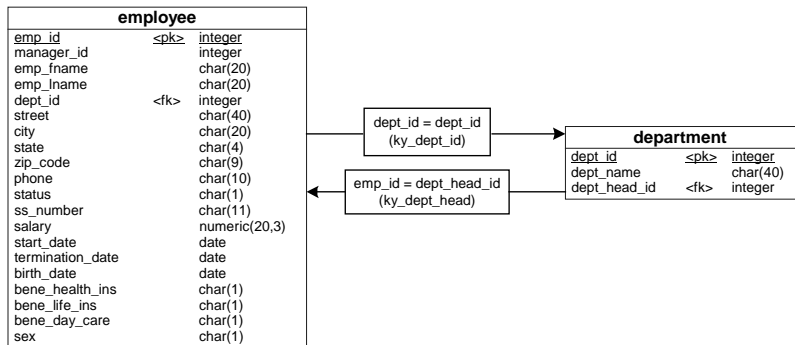
Adaptive Server Anywhere looks for a foreign key that has the same role name as the correlation name of the primary key table:

- ◆ If there is exactly one foreign key with the same name as a table in the join, Adaptive Server Anywhere uses it to generate the join condition.

- ◆ If there is more than one foreign key with the same name as a table, the join is ambiguous and an error is issued.
- ◆ If there is no foreign key with the same name as the table, Adaptive Server Anywhere looks for any foreign key relationship, even if the names don't match. If there is more than one foreign key relationship, the join is ambiguous and an error is issued.

### Example 1

In the sample database, two foreign key relationships are defined between the tables *employee* and *department*: the foreign key *ky\_dept\_id* in the *employee* table references the *department* table; and the foreign key *ky\_dept\_head* in the *department* table references the *employee* table.



The following query is ambiguous because there are two foreign key relationships and neither has the same role name as the primary key table name. Therefore, attempting this query results in the syntax error **SQL\_E\_AMBIGUOUS\_JOIN (-147)**.

```
SELECT employee.emp_lname, department.dept_name
FROM employee KEY JOIN department
```

### Example 2

This query modifies the query in Example 1 by specifying the correlation name *ky\_dept\_id* for the *department* table. Now, the foreign key *ky\_dept\_id* has the same name as the table it references, and so it is used to define the join condition. The result includes all the employee last names and the departments where they work.

```
SELECT employee.emp_lname, ky_dept_id.dept_name
FROM employee KEY JOIN department AS ky_dept_id
```

The following query is equivalent. It is not necessary to create an alias for the *department* table in this example. The same join condition that was generated above is specified in the **ON** phrase in this query:

```
SELECT employee.emp_lname, department.dept_name
FROM employee JOIN department
ON department.dept_id = employee.dept_id
```

### Example 3

If the intent was to list all the employees that are the head of a department, then the foreign key *ky\_dept\_head* should be used and Example 1 should be rewritten as follows. This query imposes the use of the foreign key *ky\_dept\_head* by specifying the correlation name *ky\_dept\_head* for the primary key table *employee*.

```
SELECT ky_dept_head.emp_lname, department.dept_name
FROM employee AS ky_dept_head KEY JOIN department
```

The following query is equivalent. The join condition that was generated above is specified in the ON phrase in this query:

```
SELECT employee.emp_lname, department.dept_name
FROM employee JOIN department
ON department.dept_head_id = employee.emp_id
```

### Example 4

A correlation name is not needed if the foreign key role name is identical to the primary key table name. For example, we can define the foreign key *department* for the *employee* table:

```
ALTER TABLE employee ADD FOREIGN KEY department
(dept_id) REFERENCES department (dept_id)
```

Now, this foreign key relationship is the default join condition when a KEY JOIN is specified between the two tables. If the foreign key *department* is defined, then the following query is equivalent to Example 3.

```
SELECT employee.emp_lname, department.dept_name
FROM employee KEY JOIN department
```

### Note

If you try this example in Interactive SQL, you should reverse the change to the sample database with the following statement:

```
ALTER TABLE employee DROP FOREIGN KEY department
```

## Key joins of table expressions

Adaptive Server Anywhere generates join conditions for the key join of table expressions by examining the foreign key relationship of each pair of tables in the statement.

The following example joins four pairs of tables.

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D)
```



The table-pairs are A-C, A-D, B-C and B-D. Adaptive Server Anywhere considers the relationship within each pair and then creates a generated join condition for the table expression as a whole. How Adaptive Server Anywhere does this depends on whether the table expressions use commas or not. Therefore, the generated join conditions in the following two examples are different. A `JOIN` B is a *table expression that does not contain commas*, and `(A, B)` is a *table expression list*.

```
SELECT *
FROM (A JOIN B) KEY JOIN C
```

is semantically different from

```
SELECT *
FROM (A,B) KEY JOIN C
```

The two types of join behavior are explained in the following sections:

- ◆ "Key joins of table expressions that do not contain commas" on page 263
- ◆ "Key joins of table expression lists" on page 264

## Key joins of table expressions that do not contain commas

When both of the two table expressions being joined do not contain commas, Adaptive Server Anywhere examines the foreign key relationships in the pairs of tables in the statement, and generates a *single* join condition.

For example, the following join has two table-pairs, A-C and B-C.

```
(A NATURAL JOIN B) KEY JOIN C
```

Adaptive Server Anywhere generates a single join condition for joining C with `(A NATURAL JOIN B)` by looking at the foreign key relationships within the table-pairs A-C and B-C. It generates one join condition for the two pairs according to the rules for determining key joins when there are multiple foreign key relationships:

- ◆ First, it looks at both A-C and B-C for a single foreign key that has the same role name as the correlation name of one of the primary key tables it references. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one foreign key with the same role name as the correlation name of a table, the join is considered to be ambiguous and an error is issued.
- ◆ If there is no foreign key with the same name as the correlation name of a table, Adaptive Server Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.

- ◆ If there is no foreign key relationship, an error is issued.

🔗 For more information, see "Key joins when there are multiple foreign key relationships" on page 260.

### Example

The following query finds all the employees who are sales representatives, and their departments.

```
SELECT employee.emp_lname, ky_dept_id.dept_name
FROM (employee KEY JOIN department as ky_dept_id)
     KEY JOIN sales_order
```

You can interpret this query as follows.

- ◆ Adaptive Server Anywhere considers the table expression (employee KEY JOIN department as ky\_dept\_id) and generates the join condition `employee.dept_id = ky_dept_id.dept_id` based on the foreign key `ky_dept_id`.
- ◆ Adaptive Server Anywhere then considers the table-pairs *employee-sales\_order* and *ky\_dept\_id-sales\_order*. Note that only one foreign key can exist between the tables *sales\_order* and *employee* and between *sales\_order* and *ky\_dept\_id*, or the join is ambiguous. As it happens, there is exactly one foreign key relationship between the tables *sales\_order* and *employee* (*ky\_so\_employee\_id*), and no foreign key relationship between *sales\_order* and *ky\_dept\_id*. Hence, the generated join condition is `sales_order.emp_id = employee.sales_rep`.

The following query is therefore equivalent to the previous query:

```
SELECT employee.emp_lname, department.dept_name
FROM (employee JOIN department
      ON ( employee.dept_id = department.dept_id ) )
JOIN sales_order
      ON (employee.emp_id = sales_order.sales_rep)
```

## Key joins of table expression lists

To generate a join condition for the key join of two table expression lists, Adaptive Server Anywhere examines the pairs of tables in the statement, and generates a join condition for *each pair*. The final join condition is the conjunction of the join conditions for each pair. There must be a foreign key relationship between each pair.

The following example joins two table-pairs, A-C and B-C.

```
SELECT *
FROM (A,B) KEY JOIN C
```

Adaptive Server Anywhere generates a join condition for joining C with (A,B) by generating a join condition for each of the two pairs A-C and B-C. It does so according to the rules for key joins when there are multiple foreign key relationships:

- ◆ For each pair, Adaptive Server Anywhere looks for a foreign key that has the same role name as the correlation name of the primary key table. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- ◆ For each pair, if there is no foreign key with the same name as the correlation name of the table, Adaptive Server Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- ◆ For each pair, if there is no foreign key relationship, an error is issued.
- ◆ If Adaptive Server Anywhere is able to determine exactly one join condition for each pair, it combines the join conditions using AND.

For more information, see "Key joins when there are multiple foreign key relationships" on page 260.

### Example

The following query returns the names of all salespeople who have sold at least one order to a specific region.

```
SELECT DISTINCT employee.emp_lname,
               ky_dept_id.dept_name, sales_order.region
FROM (sales_order, department AS ky_dept_id)
KEY JOIN employee
```

emp_lname	dept_name	region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
...	...	...

This query deals with two pairs of tables: *sales\_order* and *employee*; and *department AS ky\_dept\_id* and *employee*.

For the pair *sales\_order* and *employee*, there is no foreign key with the same role name as one of the tables. However, there is a foreign key (*ky\_so\_employee\_id*) relating the two tables. It is the only foreign key relating the two tables, and so it is used, resulting in the generated join condition (*employee.emp\_id = sales\_order.sales\_rep*).

For the pair *department AS ky\_dept\_id* and *employee*, there is one foreign key that has the same role name as the primary key table. It is *ky\_dept\_id*, and it matches the correlation name given to the *department* table in the query. There are no other foreign keys with the same name as the correlation name of the primary key table, so *ky\_dept\_id* is used to form the join condition for the table-pair. The join condition that is generated is `(employee.dept_id = ky_dept_id.dept_id)`. Note that there is another foreign key relating the two tables, but as it has a different name from either of the tables, it is not a factor.

The final join condition adds together the join condition generated for each table-pair. Therefore, the following query is equivalent:

```
SELECT DISTINCT employee.emp_lname,  
department.dept_name, sales_order.region  
FROM sales_order, department  
JOIN employee  
ON employee.emp_id = sales_order.sales_rep  
AND employee.dept_id = department.dept_id
```

### Key joins of lists and table expressions that do not contain commas

When table expression lists are joined via key join with table expressions that do not contain commas, Adaptive Server Anywhere generates a join condition for each table in the table expression list.

For example, the following statement is the key join of a table expression list with a table expression that does not contain commas. This example generates a join condition for table A with table expression `C NATURAL JOIN D`, and for table B with table expression `C NATURAL JOIN D`.

```
SELECT *  
FROM (A,B) KEY JOIN (C NATURAL JOIN D)
```

`(A,B)` is a list of table expressions and `C NATURAL JOIN D` is a table expression. Adaptive Server Anywhere must therefore generate two join conditions: it generates one join condition for the pairs A-C and A-D, and a second join condition for the pairs B-C and B-D. It does so according to the rules for key joins when there are multiple foreign key relationships:

- ◆ For each set of table-pairs, Adaptive Server Anywhere looks for a foreign key that has the same role name as the correlation name of one of the primary key tables. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one, the join is ambiguous and an error is issued.

- ◆ For each set of table-pairs, if there is no foreign key with the same name as the correlation name of a table, Adaptive Server Anywhere looks for any foreign key relationship between the tables. If there is exactly one relationship, it uses it. If there is more than one, the join is ambiguous and an error is issued.
- ◆ For each set of pairs, if there is no foreign key relationship, an error is issued.
- ◆ If Adaptive Server Anywhere is able to determine exactly one join condition for each set of pairs, it combines the join conditions with the keyword AND.

### Example 1

Consider the following join of five tables:

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

In this case, Adaptive Server Anywhere generates a join condition for the key join to E by generating a condition *either* between (A,B) and E *or* between C NATURAL JOIN D and E. This is as described in "Key joins of table expressions that do not contain commas" on page 263 on page 263.

If Adaptive Server Anywhere generates a join condition between (A,B) and E, it needs to create two join conditions, one for A-E and one for B-E. It must find a valid foreign key relationship within each table-pair. This is as described in "Key joins of table expression lists" on page 264 on page 264.

If Adaptive Server Anywhere creates a join condition between C NATURAL JOIN D and E, it creates only one join condition, and so must find only one foreign key relationship in the pairs C-E and D-E. This is as described in "Key joins of table expressions that do not contain commas" on page 263 on page 263.

### Example 2

The following is an example of a key join of a table expression and a list of table expressions. The example provides the name and department of employees who are sales representatives and also managers.

```
SELECT DISTINCT employee.emp_lname, ky_dept_id.dept_name
FROM (sales_order, department AS ky_dept_id)
KEY JOIN (employee JOIN department AS d
ON employee.emp_id = d.dept_head_id)
```

Adaptive Server Anywhere generates two join conditions:

- ◆ There must be exactly one foreign key relationship between the table-pairs *sales\_order-employee* and *sales\_order-d*. There is; it is `sales_order.sales_rep = employee.emp_id`.
- ◆ There must be exactly one foreign key relationship between the table-pairs *ky\_dept\_id-employee* and *ky\_dept\_id-d*. There is; it is `ky_dept_id.dept_id = employee.dept_id`.

This example is equivalent to the following. In the following version, it is not necessary to create the correlation name `department AS ky_dept_id`, because that was only needed to clarify which of two foreign keys should be used to join *employee* and *department*.

```
SELECT DISTINCT employee.emp_lname, department.dept_name
FROM (sales_order, department)
JOIN (employee JOIN department AS d
      ON employee.emp_id = d.dept_head_id)
ON sales_order.sales_rep = employee.emp_id
AND department.dept_id = employee.dept_id
```

## Key joins of views and derived tables

When you include a view or derived table in a key join, Adaptive Server Anywhere follows the same basic procedure as with tables, but with these differences:

- ◆ For each key join, Adaptive Server Anywhere considers the pairs of tables in the FROM clause of the query and the view, and generates *one* join condition for the set of all pairs, regardless of whether the FROM clause in the view contains commas or join keywords.
- ◆ Adaptive Server Anywhere joins the tables based on the foreign key that has the same role name as the correlation name of the view or derived table.
- ◆ When you include a view or derived table in a key join, the view cannot contain UNION, ORDER BY, DISTINCT, GROUP BY, or an aggregate function. If it contains any of those items, an error is issued.

A derived table works identically to a view. The only difference is that instead of referencing a predefined view, the definition for the table is included in the statement.

### Example 1

For example, in the following statement, *View1* is a view.

```
SELECT *
FROM View1 KEY JOIN B
```

The definition of *View1* can be any of the following and result in the same join condition to B. (The result set will differ, but the join conditions will be identical.)

```
SELECT *
FROM C CROSS JOIN D
```

or

```
SELECT *
FROM C,D
```

or

```
SELECT *  
FROM C JOIN D ON (C.x = D.y)
```

In each case, to generate a join condition for the key join of *View1* and B, Adaptive Server Anywhere considers the table-pairs C-B and D-B, and generates a single join condition. It generates the join condition based on the rules for multiple foreign key relationships described in "Key joins of table expressions" on page 262 on page 262, except that it looks for a foreign key with the same name as the correlation name of the view (rather than a table referenced in the view).

Using any of the view definitions above, you can interpret the processing of *View1* KEY JOIN B as follows:

Adaptive Server Anywhere generates a single join condition by considering the table-pairs C-B and D-B. It generates the join condition according to the rules for determining key joins when there are multiple foreign key relationships:


- ◆ First, it looks at both C-B and D-B for a single foreign key that has the same role name as the correlation name of the view. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one foreign key with the same role name as the correlation name of the view, the join is considered to be ambiguous and an error is issued.
- ◆ If there is no foreign key with the same name as the correlation name of the view, Adaptive Server Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- ◆ If there is no foreign key relationship, an error is issued.

Assume this generated join condition is *B.y = D.z*. We can now expand the original join.

```
SELECT *  
FROM View1 KEY JOIN B
```

is equivalent to

```
SELECT *  
FROM View1 JOIN B ON B.y = View1.z
```

 For more information, see "Key joins when there are multiple foreign key relationships" on page 260.

## Example 2

The following view contains all the employee information about the manager of each department.

```
CREATE VIEW V AS
SELECT department.dept_name, employee.*
FROM employee JOIN department
      ON employee.emp_id = department.dept_head_id
```

The following query joins the view to a table expression.

```
SELECT *
FROM V KEY JOIN (sales_order, department ky_dept_id)
```

This is equivalent to

```
SELECT *
FROM V JOIN (sales_order, department ky_dept_id)
ON (V.emp_id = sales_order.sales_rep
AND V.dept_id = ky_dept_id.dept_id)
```

## Rules describing the operation of key joins

The following rules summarize the information provided above.

Rule 1: key join of  
two tables

This rule applies to A KEY JOIN B, where A and B are base or temporary tables.

- 1 Find all foreign keys from A referencing B.  
  
If there exists a foreign key whose role name is the correlation name of table B, then mark it as a preferred foreign key.
- 2 Find all foreign keys from B referencing A.  
  
If there exists a foreign key whose role name is the correlation name of table A, then mark it as a preferred foreign key.
- 3 If there is more than one preferred key, the join is ambiguous. The syntax error `SQL*ERR-147` is issued.
- 4 If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
- 5 If there is no preferred key, then other foreign keys between A and B are used:
  - ◆ If there is more than one foreign key between A and B, then the join is ambiguous. The syntax error `SQL*ERR-147` is issued.
  - ◆ If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
  - ◆ If there is no foreign key, then the join is invalid and an error is generated.



Rule 2: key join of table expressions that do not contain commas

This rule applies to `A KEY JOIN B`, where A and B are table expressions that do not contain commas.

- 1 For each pair of tables; one from expression A and one from expression B, list all foreign keys, and mark all preferred foreign keys between the tables. The rule for determining a preferred foreign key is given in Rule 1, above.
- 2 If there is more than one preferred key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
- 3 If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this `KEY JOIN` expression.
- 4 If there is no preferred key, then other foreign keys between pairs of tables are used:
  - ◆ If there is more than one foreign key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
  - ◆ If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this `KEY JOIN` expression.
  - ◆ If there is no foreign key, then the join is invalid and an error is generated.

Rule 3: key join of table expression lists

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that do not contain commas.

- 1 For each pair of table expressions  $A_i$  and  $B_j$ , find a unique generated join condition for the table expression `( $A_i$  KEY JOIN  $B_j$ )` by applying Rule 1 or 2. If any `KEY JOIN` for a pair of table expressions is ambiguous by Rule 1 or 2, a syntax error is generated.
- 2 The generated join condition for this `KEY JOIN` expression is the conjunction of the join conditions found in step 1.

Rule 4: key join of lists and table expressions that do not contain commas

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that may contain commas.

- 1 For each pair of table expressions  $A_i$  and  $B_j$ , find a unique generated join condition for the table expression `( $A_i$  KEY JOIN  $B_j$ )` by applying Rule 1, 2, or 3. If any `KEY JOIN` for a pair of table expressions is ambiguous by Rule 1, 2, or 3, then a syntax error is generated.
- 2 The generated join condition for this `KEY JOIN` expression is the conjunction of the join conditions found in step 1.



C H A P T E R   9

# Using Subqueries

About this chapter

When you create a query, you use WHERE and HAVING clauses to restrict the rows that the query returns.

Sometimes, the rows you select depend on information stored in more than one table. A subquery in the WHERE or HAVING clause allows you to select rows from one table according to specifications obtained from another table. Additional ways to do this can be found in "Joins: Retrieving Data from Several Tables" on page 227.

Before your start

This chapter assumes some knowledge of queries and the syntax of the select statement. Information about queries appears in "Queries: Selecting Data from a Table" on page 183.

Contents

Topic	Page
Introduction to subqueries	274
Using subqueries in the WHERE clause	275
Subqueries in the HAVING clause	276
Subquery comparison test	278
Quantified comparison tests with ANY and ALL	279
Testing set membership with IN conditions	282
Existence test	284
Outer references	286
Subqueries and joins	287
Nested subqueries	289
How subqueries work	291

## Introduction to subqueries

A relational database stores information about different types of objects in different tables. For example, you should store information particular to products in one table, and information that pertains to sales orders in another. The product table contains the information about the various products. The sales order items table contains information about customers' orders.

In general, only the simplest questions can be answered using only one table. For example, if the company reorders products when there are fewer than 50 of them in stock, then it is possible to answer the question "Which products are nearly out of stock?" with this query:

```
SELECT id, name, description, quantity
FROM product
WHERE quantity < 50
```

However, if "nearly out of stock" depends on how many items of each type the typical customer orders, the number "50" will have to be replaced by a value obtained from the *sales\_order\_items* table.

### Structure of the subquery

A subquery is structured like a regular query, and appears in the main query's SELECT, FROM, WHERE, or HAVING clause. Continuing with the previous example, you can use a subquery to select the average number of items that a customer orders, and then use that figure in the main query to find products that are nearly out of stock. The following query finds the names and descriptions of the products which number less than twice the average number of items of each type that a customer orders.

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items
)
```

In the WHERE clause, subqueries help select the rows from the tables listed in the FROM clause that appear in the query results. In the HAVING clause, they help select the row groups, as specified by the main query's GROUP BY clause, that appear in the query results.

## Using subqueries in the WHERE clause

Subqueries in the WHERE clause work as part of the row selection process. You use a subquery in the WHERE clause when the criteria you use to select rows depend on the results of another table.

### Example

Find the products whose in-stock quantities are less than double the average ordered quantity.

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items)
```

This is a two-step query: first, find the average number of items requested per order; and then find which products in stock number less than double that quantity.

### The query in two steps

The *quantity* column of the *sales\_order\_items* table stores the number of items requested per item type, customer, and order. The subquery is

```
SELECT avg(quantity)
FROM sales_order_items
```

It returns the average quantity of items in the *sales\_order\_items* table, which is 25.851413.

The next query returns the names and descriptions of the items whose in-stock quantities are less than twice the previously-extracted value.

```
SELECT name, description
FROM product
WHERE quantity < 2*25.851413
```

Using a subquery combines the two steps into a single operation.

### Purpose of a subquery in the WHERE clause

A subquery in the WHERE clause is part of a search condition. The chapter "Queries: Selecting Data from a Table" on page 183 describes simple search conditions you can use in the WHERE clause.

# Subqueries in the HAVING clause

Although you usually use subqueries as search conditions in the WHERE clause, sometimes you can also use them in the HAVING clause of a query. When a subquery appears in the HAVING clause, like any expression in the HAVING clause, it is used as part of the row group selection.

Here is a request that lends itself naturally to a query with a subquery in the HAVING clause: "Which products' average in-stock quantity is more than double the average number of each item ordered per customer?"

Example

```
SELECT name, avg(quantity)
FROM product
GROUP BY name
HAVING avg(quantity) > 2* (
    SELECT avg(quantity)
    FROM sales_order_items
)
```

name	avg(product.quantity)
Tee Shirt	52.333333
Baseball Cap	62
Shorts	80

The query executes as follows:

- ◆ The subquery calculates the average quantity of items in the *sales\_order\_items* table.
- ◆ The main query then goes through the *product* table, calculating the average quantity product, grouping by product name.
- ◆ The HAVING clause then checks if each average quantity is more than double the quantity found by the subquery. If so, the main query returns that row group; otherwise, it doesn't.
- ◆ The SELECT clause produces one summary row for each group, displaying the name of each product and its in-stock average quantity.

You can also use outer references in a HAVING clause, as shown in the following example, a slight variation on the one above.

Example

"Find the product ID numbers and line ID numbers of those products whose average ordered quantities is more than half the in-stock quantities of those products."

```

SELECT prod_id, line_id
FROM sales_order_items
GROUP BY prod_id, line_id
HAVING 2* avg(quantity) > (
    SELECT quantity
    FROM product
    WHERE product.id = sales_order_items.prod_id)

```

prod_id	line_id
401	2
401	1
401	4
501	3
...	...

In this example, the subquery must produce the in-stock quantity of the product corresponding to the row group being tested by the HAVING clause. The subquery selects records for that particular product, using the outer reference *sales\_order\_items.prod\_id*.

A subquery with a comparison returns a single value

This query uses the comparison ">", suggesting that the subquery must return exactly one value. In this case, it does. Since the *id* field of the *product* table is a primary key, there is only one record in the *product* table corresponding to any particular product id.

## Subquery tests

The chapter "Queries: Selecting Data from a Table" on page 183 describes simple search conditions you can use in the HAVING clause. Since a subquery is just an expression that appears in the WHERE or HAVING clauses, the search conditions on subqueries may look familiar.

They include:

- ◆ **Subquery comparison test** Compares the value of an expression to a single value produced by the subquery for each record in the table(s) in the main query.
- ◆ **Quantified comparison test** Compares the value of an expression to each of the set of values produced by a subquery.
- ◆ **Subquery set membership test** Checks if the value of an expression matches one of the set of values produced by a subquery.
- ◆ **Existence test** Checks if the subquery produces any rows.

# Subquery comparison test

The subquery comparison test (=, <>, <, <=, >, >=) is a modified version of the simple comparison test. The only difference between the two is that in the former, the expression following the operator is a subquery. This test is used to compare a value from a row in the main query to a *single* value produced by the subquery.

Example

This query contains an example of a subquery comparison test:

```
SELECT name, description, quantity
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
...	...	...

The following subquery retrieves a single value—the average quantity of items of each type per customer's order—from the *sales\_order\_items* table.

```
SELECT avg(quantity)
FROM sales_order_items
```

Then the main query compares the quantity of each in-stock item to that value.

A subquery in a comparison test returns one value

A subquery in a comparison test must return exactly one value. Consider this query, whose subquery extracts two columns from the *sales\_order\_items* table:

```
SELECT name, description, quantity
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity), max (quantity)
    FROM sales_order_items)
```

It returns the error Subquery allowed only one select list item.



## Quantified comparison tests with ANY and ALL

The quantified comparison test has two categories, the ALL test and the ANY test:

### The ANY test

The ANY test, used in conjunction with one of the SQL comparison operators (=, <>, <, <=, >, >=), compares a single value to the column of data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column. If *any* of the comparisons yields a TRUE result, the ANY test returns TRUE.

A subquery used with ANY must return a single column.

#### Example

Find the order and customer IDs of those orders placed after the first product of the order #2005 was shipped.

```
SELECT id, cust_id
FROM sales_order
WHERE order_date > ANY (
    SELECT ship_date
    FROM sales_order_items
    WHERE id=2005)
```

id	cust_id
2006	105
2007	106
2008	107
2009	108
...	...

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of the order #2005. If an order date is greater than the shipping date for *one* shipment of order #2005, then that id and customer id from the *sales\_order* table are part of the result set. The ANY test is thus analogous to the OR operator: the above query can be read, "Was this sales order placed after the first product of the order #2005 was shipped, or after the second product of order #2005 was shipped, or..."

### Understanding the ANY operator

The ANY operator can be a bit confusing. It is tempting to read the query as "Return those orders placed after any products of order #2005 were shipped." But this means the query will return the order IDs and customer IDs for the orders placed after *all* products of order #2005 were shipped—which is not what the query does.

Instead, try reading the query like this: "Return the order and customer IDs for those orders placed after *at least one* product of order #2005 was shipped." Using the keyword SOME may provide a more intuitive way to phrase the query. The following query is equivalent to the previous query.

```
SELECT id, cust_id
FROM sales_order
WHERE order_date > SOME (
    SELECT ship_date
    FROM sales_order_items
    WHERE id=2005)
```

The keyword SOME is equivalent to the keyword ANY.

### Notes about the ANY operator

There are two additional important characteristics of the ANY test:

- ◆ **Empty subquery result set** If the subquery produces an empty result set, the ANY test returns FALSE. This makes sense, since if there are no results, then it is not true that at least one result satisfies the comparison test.
- ◆ **NULL values in subquery result set** Assume that there is at least one NULL value in the subquery result set. If the comparison test is false for all non-NULL data values in the result set, the ANY search returns FALSE. This is because in this situation, you cannot conclusively state whether there is a value for the subquery for which the comparison test holds. There may or may not be a value, depending on the "correct" values for the NULL data in the result set.

## The ALL test

Like the ANY test, the ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single value to the data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the result set. If *all* of the comparisons yield TRUE results, the ALL test returns TRUE.

### Example

Here is a request naturally handled with the ALL test: "Find the order and customer IDs of those orders placed after all products of order #2001 were shipped."

```

SELECT id, cust_id
FROM sales_order
WHERE order_date > ALL (
    SELECT ship_date
    FROM sales_order_items
    WHERE id=2001)

```

id	cust_id
2002	102
2003	103
2004	104
2005	101
...	...

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of order #2001. If an order date is greater than the shipping date for *every* shipment of order #2001, then the id and customer id from the *sales\_order* table are part of the result set. The ALL test is thus analogous to the AND operator: the above query can be read, "Was this sales order placed before the first product of order #2001 was shipped, and before the second product of order #2001 was shipped, and..."

#### Notes about the ALL operator

There are three additional important characteristics of the ALL test:

- ◆ **Empty subquery result set** If the subquery produces an empty result set, the ALL test returns TRUE. This makes sense, since if there are no results, then it is true that the comparison test holds for every value in the result set.
- ◆ **NULL values in subquery result set** If the comparison test is false for any values in the result set, the ALL search returns FALSE. It returns TRUE if all values are true. Otherwise, it returns UNKNOWN—for example, this can occur if there is a NULL value in the subquery result set but the search condition is TRUE for all non-NULL values.
- ◆ **Negating the ALL test** The following expressions are *not* equivalent.

`NOT a = ALL (subquery)`

`a <> ALL (subquery)`

☞ For more information about this test, see "Quantified comparison test" on page 293.

# Testing set membership with IN conditions

You can use the subquery set membership test to compare a value from the main query to more than one value in the subquery.

The subquery set membership test compares a single data value for each row in the main query to the single column of data values produced by the subquery. If the data value from the main query matches *one* of the data values in the column, the subquery returns TRUE.

## Example

Select the names of the employees who head the Shipping or Finance departments:

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

emp_fname	emp_lname
Jose	Martinez
Mary Anne	Shea

The subquery in this example

```
SELECT dept_head_id
FROM department
WHERE (dept_name='Finance' OR dept_name = 'Shipping')
```

extracts from the *department* table the id numbers that correspond to the heads of the Shipping and Finance departments. The main query then returns the names of the employees whose id numbers match one of the two found by the subquery.

Set membership  
test is equivalent to  
=ANY test

The subquery set membership test is equivalent to the =ANY test. The following query is equivalent to the query from the above example.

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id =ANY (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

**Negation of the set membership test**

You can also use the subquery set membership test to extract those rows whose column values are not equal to any of those produced by a subquery. To negate a set membership test, insert the word **NOT** in front of the keyword **IN**.

**Example**

The subquery in this query returns the first and last names of the employees that are not heads of the Finance or Shipping departments.

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id NOT IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

# Existence test

Subqueries used in the subquery comparison test and set membership test both return data values from the subquery table. Sometimes, however, you may be more concerned with whether the subquery returns *any* results, rather than *which* results. The existence test (EXISTS) checks whether a subquery produces any rows of query results. If the subquery produces one or more rows of results, the EXISTS test returns TRUE. Otherwise, it returns FALSE.

## Example

Here is an example of a request expressed using a subquery: "Which customers placed orders after July 13, 2001?"

```
SELECT fname, lname
FROM customer
WHERE EXISTS (
  SELECT *
  FROM sales_order
  WHERE (order_date > '2001-07-13') AND
    (customer.id = sales_order.cust_id))
```

fname	lname
Almen	de Joie
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

## Explanation of the existence test

Here, for each row in the *customer* table, the subquery checks if that customer ID corresponds to one that has placed an order after July 13, 2001. If it does, the query extracts the first and last names of that customer from the main table.

The EXISTS test does not use the results of the subquery; it just checks if the subquery produces any rows. So the existence test applied to the following two subqueries return the same results. These are subqueries and cannot be processed on their own, because they refer to the customer table which is part of the main query, but not part of the subquery.

☞ For more information, see "Correlated subqueries" on page 291.

```
SELECT *
FROM sales_order
WHERE (order_date > '2001-07-13') AND (customer.id =
  sales_order.cust_id)
```

```
SELECT ship_date
FROM sales_order
WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id)
```

It does not matter which columns from the *sales\_order* table appear in the *SELECT* statement, though by convention, the "*SELECT \**" notation is used.

#### Negating the existence test

You can reverse the logic of the *EXISTS* test using the *NOT EXISTS* form. In this case, the test returns *TRUE* if the subquery produces no rows, and *FALSE* otherwise.

#### Correlated subqueries

You may have noticed that the subquery contains a reference to the *id* column from the *customer* table. A reference to columns or expressions in the main table(s) is called an **outer reference** and the subquery is said to be **correlated**. Conceptually, SQL processes the above query by going through the *customer* table, and performing the subquery for each customer. If the order date in the *sales\_order* table is after July 13, 2001, and the customer ID in the *customer* and *sales\_order* tables match, then the first and last names from the customer table appear. Since the subquery references the main query, the subquery in this section, unlike those from previous sections, returns an error if you attempt to run it by itself.

## Outer references

Within the body of a subquery, it is often necessary to refer to the value of a column in the active row of the main query. Consider the following query:

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items
    WHERE product.id = sales_order_items.prod_id)
```

This query extracts the names and descriptions of the products whose in-stock quantities are less than double the average ordered quantity of that product—specifically, the product being tested by the WHERE clause in the main query. The subquery does this by scanning the *sales\_order\_items* table. But the *product.id* column in the WHERE clause of the subquery refers to a column in the table named in the FROM clause of the *main* query—not the subquery. As SQL moves through each row of the *product* table, it uses the *id* value of the current row when it evaluates the WHERE clause of the subquery.

### Description of an outer reference

The *product.id* column in this subquery is an example of an outer reference. A subquery that uses an outer reference is a correlated subquery. An outer reference is a column name that does not refer to any of the columns in any of the tables in the FROM clause of the subquery. Instead, the column name refers to a column of a table specified in the FROM clause of the main query. As the above example shows, the value of a column in an outer reference comes from the row currently being tested by the main query.



## Subqueries and joins

The subquery optimizer automatically rewrites as joins many of the queries that make use of subqueries.

### Example

Consider the request, "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" It can be answered with the following query:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id IN (
    SELECT id
    FROM customer
    WHERE lname = 'Clarke' OR fname = 'Suresh')
```

Order_date	sales_rep
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
...	...

The subquery yields a list of customer IDs that correspond to the two customers whose names are listed in the **WHERE** clause, and the main query finds the order dates and sales representatives corresponding to those two people's orders.

### Replacing a subquery with a join

The same question can be answered using joins. Here is an alternative form of the query, using a two-table join:

```
SELECT order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND
      (lname = 'Clarke' OR fname = 'Suresh')
```

This form of the query joins the *sales\_order* table to the *customer* table to find the orders for each customer, and then returns only those records for Suresh and Clarke.

### Some joins cannot be written as subqueries

Both of these queries find the correct order dates and sales representatives, and neither is more right than the other. Many people will find the subquery form more natural, because the request doesn't ask for any information about customer IDs, and because it might seem odd to join the *sales\_order* and *customer* tables together to answer the question.

If, however, the request changes to include some information from the *customer* table, the subquery form no longer works. For example, the request "When did Mrs. Clarke and Suresh place their orders, and by which representatives, and what are their full names?", it is necessary to include the *customer* table in the main WHERE clause:

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

fname	lname	order_date	sales_rep
Belinda	Clarke	1/5/01	1596
Belinda	Clarke	1/27/00	667
Belinda	Clarke	11/11/00	467
Belinda	Clarke	2/4/01	195
...	...	...	...

Some subqueries cannot be written as joins

Similarly, there are cases where a subquery will work but a join will not. For example:

```
SELECT name, description, quantity
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
...	...	...

In this case, the inner query is a summary query and the outer query is not, so there is no way to combine the two queries by a simple join.

🔗 For more information on joins, see "Joins: Retrieving Data from Several Tables" on page 227.

## Nested subqueries

As we have seen, subqueries always appear in the HAVING clause or the WHERE clause of a query. A subquery may itself contain a WHERE clause and/or a HAVING clause, and, consequently, a subquery may appear in another subquery. Subqueries inside other subqueries are called **nested subqueries**.

### Examples

List the order IDs and line IDs of those orders shipped on the same day when any item in the fees department was ordered.

```
SELECT id, line_id
FROM sales_order_items
WHERE ship_date = ANY (
    SELECT order_date
    FROM sales_order
    WHERE fin_code_id IN (
        SELECT code
        FROM fin_code
        WHERE (description = 'Fees')))
```

id	line_id
2001	1
2001	2
2001	3
2002	1
...	...

### Explanation of the nested subqueries

- ◆ In this example, the innermost subquery produces a column of financial codes whose descriptions are "Fees":

```
SELECT code
FROM fin_code
WHERE (description = 'Fees')
```

- ◆ The next subquery finds the order dates of the items whose codes match one of the codes selected in the innermost subquery:

```
SELECT order_date
FROM sales_order
WHERE fin_code_id IN (subquery)
```

- ◆ Finally, the outermost query finds the order IDs and line IDs of the orders shipped on one of the dates found in the subquery.

```
SELECT id, line_id
FROM sales_order_items
WHERE ship_date = ANY (subquery)
```

Nested subqueries can also have more than three levels. Though there is no maximum number of levels, queries with three or more levels take considerably longer to run than do smaller queries.

## How subqueries work

Understanding which queries are valid and which ones aren't can be complicated when a query contains a subquery. Similarly, figuring out what a multi-level query does can also be very involved, and it helps to understand how Adaptive Server Anywhere processes subqueries. For general information about processing queries, see "Summarizing, Grouping and Sorting Query Results" on page 207.

### Correlated subqueries

In a simple query, the database server evaluates and processes the query's WHERE clause once for each row of the query. Sometimes, though, the subquery returns only one result, making it unnecessary for the database server to evaluate it more than once for the entire result set.

#### Uncorrelated subqueries

Consider this query:

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items)
```

In this example, the subquery calculates exactly one value: the average quantity from the *sales\_order\_items* table. In evaluating the query, the database server computes this value once, and compares each value in the *quantity* field of the *product* table to it to determine whether to select the corresponding row.

#### Correlated subqueries

When a subquery contains an outer reference, you cannot use this shortcut. For instance, the subquery in the query

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items
    WHERE product.id=sales_order_items.prod_id)
```

returns a value dependent upon the active row in the *product* table. Such a subquery is called a correlated subquery. In these cases, the subquery might return a different value for each row of the outer query, making it necessary for the database server to perform more than one evaluation.

## Converting subqueries in the WHERE clause to joins

In general, a query using joins executes faster than a multi-level query. For this reason, whenever possible, the Adaptive Server Anywhere query optimizer converts a multi-level query to a query using joins. The conversion is carried out without any user action. This section describes which subqueries can be converted to joins so you can understand the performance of queries in your database.

### Example

The question "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" can be written as a two-level query:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id IN (
    SELECT id
    FROM customer
    WHERE lname = 'Clarke' OR fname = 'Suresh')
```

An alternate, and equally correct way to write the query uses joins:

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND
      (lname = 'Clarke' OR fname = 'Suresh')
```

The criteria that must be satisfied in order for a multi-level query to be able to be rewritten with joins differ for the various types of operators. Recall that when a subquery appears in the query's WHERE clause, it is of the form

```
SELECT select-list
FROM table
WHERE
    [ NOT ] expression comparison-operator ( subquery )
    | [ NOT ] expression comparison-operator { ANY | SOME } ( subquery )
    )
    | [ NOT ] expression comparison-operator ALL ( subquery )
    | [ NOT ] expression IN ( subquery )
    | [ NOT ] EXISTS ( subquery )
GROUP BY group-by-expression
HAVING search-condition
```

Whether a subquery can be converted to a join depends on a number of factors, such as the type of operator and the structures of the query and of the subquery.

## Comparison operators

A subquery that follows a comparison operator (`=`, `<>`, `<`, `<=`, `>`, `>=`) must satisfy certain conditions if it is to be converted into a join. Subqueries that follow comparison operators are valid only if they return exactly one value for each row of the main query. In addition to this criterion, a subquery is converted to a join only if the subquery

- ◆ does not contain a `GROUP BY` clause
- ◆ does not contain the keyword `DISTINCT`
- ◆ is not a `UNION` query
- ◆ is not an aggregate query

### Example

Suppose the request "When were Suresh's products ordered, and by which sales representative?" were phrased as the subquery

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = (
    SELECT id
    FROM customer
    WHERE fname = 'Suresh')
```

This query satisfies the criteria, and therefore, it would be converted to a query using a join:

```
SELECT order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

However, the request, "Find the products whose in-stock quantities are less than double the average ordered quantity" cannot be converted to a join, as the subquery contains the aggregate function `avg`:

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
    SELECT avg(quantity)
    FROM sales_order_items)
```

## Quantified comparison test

A subquery that follows one of the keywords `ALL`, `ANY` and `SOME` is converted into a join only if it satisfies certain criteria.

- ◆ The main query does not contain a `GROUP BY` clause, and is not an aggregate query, or the subquery returns exactly one value.

- ◆ The subquery does not contain a GROUP BY clause.
- ◆ The subquery does not contain the keyword DISTINCT.
- ◆ The subquery is not a UNION query.
- ◆ The subquery is not an aggregate query.
- ◆ The conjunct 'expression comparison-operator ANY/SOME (subquery)' must be negated.
- ◆ The conjunct 'expression comparison-operator ALL (subquery)' must not be negated.

The first four of these conditions are relatively straightforward.

### Example

The request "When did Ms. Clarke and Suresh place their orders, and by which sales representatives?" can be handled in subquery form:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = ANY (
    SELECT id
    FROM customer
    WHERE lname = 'Clarke' OR fname = 'Suresh')
```

Alternately, it can be phrased in join form

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

However, the request, "When did Ms. Clarke, Suresh, and any employee who is also a customer, place their orders?" would be phrased as a union query, and thus cannot be converted to a join:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = ANY (
    SELECT id
    FROM customer
    WHERE lname = 'Clarke' OR fname = 'Suresh'
    UNION
    SELECT id
    FROM employee)
```

Similarly, the request "Find the order IDs and customer IDs of those orders that were not placed after all products of order #2001 were shipped," is naturally expressed with a subquery:



```

SELECT id, cust_id
FROM sales_order
WHERE NOT order_date > ALL (
    SELECT ship_date
    FROM sales_order_items
    WHERE id=2001)

```

It would be converted to the join:

```

SELECT sales_order.id, cust_id
FROM sales_order, sales_order_items
WHERE (sales_order_items.id=2001) and (order_date <=
ship_date)

```

However, the request "Find the order IDs and customer IDs of those orders not shipped after the first shipping dates of all the products" would be phrased as the aggregate query:

```

SELECT id, cust_id
FROM sales_order
WHERE NOT order_date > ALL (
    SELECT first (ship_date)
    FROM sales_order_items )

```

Therefore, it would not be converted to a join.

Negating  
subqueries with the  
ANY and ALL  
operators

The fifth criterion is a little more puzzling: queries of the form

```

SELECT select-list
FROM table
WHERE NOT expression comparison-operator ALL( subquery )

```

are converted to joins, as are queries of the form

```

SELECT select-list
FROM table
WHERE expression comparison-operator ANY( subquery )

```

but the queries

```

SELECT select-list
FROM table
WHERE expression comparison-operator ALL( subquery )

```

and

```

SELECT select-list
FROM table
WHERE NOT expression comparison-operator ANY( subquery )

```

are not.

Logical  
equivalence of  
ANY and ALL  
expressions

This is because the first two queries are in fact equivalent, as are the last two. Recall that the any operator is analogous to the OR operator, but with a variable number of arguments; and that the ALL operator is similarly analogous to the AND operator. Just as the expression

```
NOT ((X > A) AND (X > B))
```

is equivalent to the expression

```
(X <= A) OR (X <= B)
```

the expression

```
NOT order_date > ALL (
  SELECT first (ship_date)
  FROM sales_order_items )
```

is equivalent to the expression

```
order_date <= ANY (
  SELECT first (ship_date)
  FROM sales_order_items )
```

Negating the ANY  
and ALL  
expressions

In general, the expression

```
NOT column-name operator ANY( subquery )
```

is equivalent to the expression

```
column-name inverse-operator ALL( subquery )
```

and the expression

```
NOT column-name operator ALL( subquery )
```

is equivalent to the expression

```
column-name inverse-operator ANY( subquery )
```

where *inverse-operator* is obtained by negating *operator*, as shown in the table:

Table of operators  
and their inverses

The following table lists the inverse of each operator.

Operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

## Set membership test

A query containing a subquery that follows the keyword IN is converted into a join only if:

- ◆ The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.
- ◆ The subquery does not contain a GROUP BY clause.
- ◆ The subquery does not contain the keyword DISTINCT.
- ◆ The subquery is not a UNION query.
- ◆ The subquery is not an aggregate query.
- ◆ The conjunct 'expression IN (subquery)' must not be negated.

### Example

So, the request "Find the names of the employees who are also department heads", expressed by the query:

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name = 'Shipping'))
```

would be converted to a joined query, as it satisfies the conditions. However, the request, "Find the names of the employees who are either department heads or customers" would not be converted to a join if it were expressed by the UNION query.

A UNION query following the IN operator can't be converted

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name = 'Shipping')
    UNION
    SELECT cust_id
    FROM sales_order)
```

Similarly, the request "Find the names of employees who are not department heads" is formulated as the negated subquery

```
SELECT emp_fname, emp_lname
FROM employee
WHERE NOT emp_id IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' OR dept_name = 'Shipping'))
```

and would not be converted.

A query with an IN operator can be converted to one with an ANY operator

The conditions that must be fulfilled for a subquery that follows the IN keyword and the ANY keyword to be converted to a join are identical. This is not a coincidence, and the reason for this is that the expression

**WHERE column-name IN( subquery )**

is logically equivalent to the expression

**WHERE column-name = ANY( subquery )**

So the query

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name = 'Shipping'))
```

is equivalent to the query

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id = ANY (
    SELECT dept_head_id
    FROM department
    WHERE (dept_name='Finance' or dept_name = 'Shipping'))
```

Conceptually, Adaptive Server Anywhere converts a query with the IN operator to one with an ANY operator, and decides accordingly whether to convert the subquery to a join.

## Existence test

A subquery that follows the keyword EXISTS is converted to a join only if it satisfies the following two conditions:

- ◆ The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.
- ◆ The conjunct 'EXISTS (subquery)' is not negated.
- ◆ The subquery is correlated; that is, it contains an outer reference.

Example

Therefore, the request, "Which customers placed orders after July 13, 2001?", which can be formulated by this query whose non-negated subquery contains the outer reference **customer.id = sales\_order.cust\_id**, could be converted to a join.

```
SELECT fname, lname
FROM customer
WHERE EXISTS (
    SELECT *
    FROM sales_order
    WHERE (order_date > '2001-07-13') AND (customer.id =
sales_order.cust_id))
```

The EXISTS keyword essentially tells the database server to check for empty result sets. When using inner joins, the database server automatically displays only the rows where there is data from all of the tables in the FROM clause. So, this query returns the same rows as does the one with the subquery:

```
SELECT fname, lname
FROM customer, sales_order
WHERE (sales_order.order_date > '2001-07-13') AND
(customer.id = sales_order.cust_id)
```



# Adding, Changing, and Deleting Data

About this chapter      This chapter describes how to modify the data in a database.

Most of the chapter is devoted to the INSERT, UPDATE, and DELETE statements, as well as statements for bulk loading and unloading.

Contents	<table><tr><th>Topic</th><th>Page</th></tr><tr><td>Data modification statements</td><td>302</td></tr><tr><td>Adding data using INSERT</td><td>303</td></tr><tr><td>Changing data using UPDATE</td><td>308</td></tr><tr><td>Changing data using INSERT</td><td>310</td></tr><tr><td>Deleting data using DELETE</td><td>311</td></tr></table>	Topic	Page	Data modification statements	302	Adding data using INSERT	303	Changing data using UPDATE	308	Changing data using INSERT	310	Deleting data using DELETE	311
Topic	Page												
Data modification statements	302												
Adding data using INSERT	303												
Changing data using UPDATE	308												
Changing data using INSERT	310												
Deleting data using DELETE	311												

## Data modification statements

The statements you use to add, change, or delete data are called **data modification** statements. The most common such statements include:

- ◆ **Insert** adds new rows to a table
- ◆ **Update** changes existing rows in a table
- ◆ **Delete** removes specific rows from a table

Any single INSERT, UPDATE, or DELETE statement changes the data in only one table or view.

In addition to the common statements, the LOAD TABLE and TRUNCATE TABLE statements are especially useful for bulk loading and deleting of data.

Sometimes, the data modification statements are collectively known as the **data modification language** (DML) part of SQL.

## Permissions for data modification

You can only execute data modification statements if you have the proper permissions on the database tables you want to modify. The database administrator and the owners of database objects use the GRANT and REVOKE statements to decide who has access to which data modification functions.

✍ Permissions can be granted to individual users, groups, or the public group. For more information on permissions, see "Managing User IDs and Permissions" on page 351 of the book *ASA Database Administration Guide*.

## Transactions and data modification

When you modify data, the transaction log stores a copy of the old and new state of each row affected by each data modification statement. This means that if you begin a transaction, realize you have made a mistake, and roll the transaction back, you also restore the database to its previous condition.

✍ For more information about transactions, see "Using Transactions and Isolation Levels" on page 89.



## Adding data using INSERT

You add rows to the database using the INSERT statement. The INSERT statement has two forms: you can use the VALUES keyword or a SELECT statement:

INSERT using values

The VALUES keyword specifies values for some or all of the columns in a new row. A simplified version of the syntax for the INSERT statement using the VALUES keyword is:

```
INSERT [ INTO ] table-name [ ( column-name, ... ) ]  
VALUES ( expression , ... )
```


You can omit the list of column names if you provide a value for each column in the table, in the order in which they appear when you execute a query using SELECT \*.

INSERT from SELECT

You can use SELECT within an INSERT statement to pull values from one or more tables. If the table you are inserting data into has a large number of columns, you can also use WITH AUTO NAME to simplify the syntax. Using WITH AUTO NAME, you only need to specify the column names in the SELECT statement, rather than in both the INSERT and the SELECT statements. The names in the SELECT statement must be column references or aliased expressions.

A simplified version of the syntax for the INSERT statement using a select statement is:

```
INSERT [ INTO ] table-name WITH AUTO NAME select-statement
```

 For more information about the INSERT statement, see the "INSERT statement" on page 463 of the book *ASA SQL Reference Manual*.

## Inserting values into all columns of a row

The following INSERT statement adds a new row to the *department* table, giving a value for every column in the row:

```
INSERT INTO department  
VALUES ( 702, 'Eastern Sales', 902 )
```

Notes

- ◆ Type the values in the same order as the column names in the original CREATE TABLE statement, that is, first the ID number, then the name, then the department head ID.
- ◆ Surround the values by parentheses.
- ◆ Enclose all character data in single quotes.

- ◆ Use a separate insert statement for each row you add.

## Inserting values into specific columns

You can add data to some columns in a row by specifying only those columns and their values. Define all other columns not included in the column list must to allow NULL or have defaults. If you skip a column that has a default value, the default appears in that column.

Adding data in only two columns, for example, *dept\_id* and *dept\_name*, requires a statement like this:

```
INSERT INTO department (dept_id, dept_name)
VALUES ( 703, 'Western Sales' )
```

The *dept\_head\_id* column has no default, but can allow NULL. A NULL is assigned to that column.

The order in which you list the column names must match the order in which you list the values. The following example produces the same results as the previous one:

```
INSERT INTO department (dept_name, dept_id )
VALUES ( 'Western Sales', 703)
```

Values for  
unspecified  
columns

When you specify values for only some of the columns in a row, one of four things can happen to the columns with no values specified:

- ◆ **NULL entered** NULL appears if the column allows NULL and no default value exists for the column.
- ◆ **A default value entered** The default value appears if a default exists for the column.
- ◆ **A unique, sequential value entered** A unique, sequential value appears if the column has the AUTOINCREMENT default or the IDENTITY property.
- ◆ **INSERT rejected, and an error message appears** An error message appears if the column does not allow NULL and no default exists.

By default, columns allow NULL unless you explicitly state NOT NULL in the column definition when creating tables. You can alter the default using the ALLOW\_NULLS\_BY\_DEFAULT option.

Restricting column  
data using  
constraints

You can create constraints for a column or domain. Constraints govern the kind of data you can or cannot add.

✍ For more information on constraints, see "Using table and column constraints" on page 76.

### Explicitly inserting NULL


You can explicitly insert NULL into a column by typing NULL. Do not enclose this in quotes, or it will be taken as a string.

For example, the following statement explicitly inserts NULL into the *dept\_head\_id* column:

```
INSERT INTO department
VALUES (703, 'Western Sales', NULL )
```

### Using defaults to supply values

You can define a column so that, even though the column receives no value, a default value automatically appears whenever a row is inserted. You do this by supplying a default for the column.

 For more information about defaults, see "Using column defaults" on page 70.

## Adding new rows with SELECT

To pull values into a table from one or more other tables, you can use a SELECT clause in the INSERT statement. The select clause can insert values into some or all of the columns in a row.

Inserting values for only some columns can come in handy when you want to take some values from an existing table. Then, you can use update to add the values for the other columns.

Before inserting values for some, but not all, columns in a table, make sure that either a default exists, or you specify NULL for the columns for which you are not inserting values. Otherwise, an error appears.

When you insert rows from one table into another, the two tables must have compatible structures—that is, the matching columns must be either the same data types or data types between which Adaptive Server automatically converts.

### Example

If the columns are in the same order in their create table statements, you do not need to specify column names in either table. Suppose you have a table named *newproduct* that contains some rows of product information in the same format as in the *product* table. To add to *product* all the rows in *newproduct*:

```
INSERT product
SELECT *
FROM newproduct
```

You can use expressions in a SELECT statement inside an INSERT statement.

### Inserting data into some columns

You can use the SELECT statement to add data to some, but not all, columns in a row just as you do with the VALUES clause. Simply specify the columns to which you want to add data in the INSERT clause.

### Inserting Data from the Same Table

You can insert data into a table based on other data in the same table. Essentially, this means copying all or part of a row.

For example, you can insert new products, based on existing products, into the *product* table. The following statement adds new Extra Large Tee Shirts (of Tank Top, V-neck, and Crew Neck varieties) into the *product* table. The identification number is ten greater than the existing sized shirt:

```
INSERT INTO product
SELECT id+ 10, name, description,
       'Extra large', color, 50, unit_price
FROM product
WHERE name = 'Tee Shirt'
```

## Inserting documents and images

If you want to store documents or images in LONG BINARY columns in your database, you can write an application that reads the contents of the file into a variable, and supplies that variable as a value for an INSERT statement.

☞ For more information about adding INSERT statements to applications, see "How to use prepared statements" on page 13 of the book *ASA Programming Guide*, and "SET statement" on page 531 of the book *ASA SQL Reference Manual*.

You can also use the *xp\_read\_file* system function to insert file contents into a table. This function is useful if you want to insert file contents from Interactive SQL, or some other environment that does not provide a full programming language.

DBA authority is required to use this external function.

### Example

In this example, you create a table, and insert an image into a column of the table. You can carry out these steps from Interactive SQL.

- 1 Create a table to hold some images.

```
CREATE TABLE pictures
( c1 INT DEFAULT AUTOINCREMENT PRIMARY KEY,
  filename VARCHAR(254),
  picture LONG BINARY )
```

- 2 Insert the contents of *portrait.gif*, in the current working directory of the database server, into the table.

```
INSERT INTO pictures (filename, picture)
VALUES ( 'portrait.gif',
        xp_read_file( 'portrait.gif' ) )
```

☞ For more information, see "xp\_read\_file system procedure" on page 734 of the book *ASA SQL Reference Manual*.

## Changing data using UPDATE

You can use the UPDATE statement, followed by the name of the table or view, to change single rows, groups of rows, or all rows in a table. As in all data modification statements, you can change the data in only one table or view at a time.

The UPDATE statement specifies the row or rows you want changed and the new data. The new data can be a constant or an expression that you specify or data pulled from other tables.

If an UPDATE statement violates an integrity constraint, the update does not take place and an error message appears. For example, if one of the values being added is the wrong data type, or if it violates a constraint defined for one of the columns or data types involved, the update does not take place.

### UPDATE syntax

A simplified version of the UPDATE syntax is:

```
UPDATE table-name
SET column_name = expression
WHERE search-condition
```

If the company Newton Ent. (in the customer table of the sample database) is taken over by Einstein, Inc., you can update the name of the company using a statement such as the following:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE company_name = 'Newton Ent.'
```

You can use any expression in the WHERE clause. If you are not sure how the company name was spelled, you could try updating any company called Newton, with a statement such as the following:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE company_name LIKE 'Newton%'
```

The search condition need not refer to the column being updated. The company ID for Newton Entertainments is 109. As the ID value is the primary key for the table, you could be sure of updating the correct row using the following statement:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE id = 109
```

### The SET clause

The SET clause specifies the columns to be updated, and their new values. The WHERE clause determines the row or rows to be updated. If you do not have a WHERE clause, the specified columns of all rows are updated with the values given in the SET clause.

You can provide any expression of the correct data type in the SET clause.

**The WHERE clause**

The WHERE clause specifies the rows to be updated. For example, the following statement replaces the One Size Fits All Tee Shirt with an Extra Large Tee Shirt

```
UPDATE product
SET size = 'Extra Large'
WHERE name = 'Tee Shirt'
      AND size = 'One Size Fits All'
```

**The FROM clause**


You can use a FROM clause to pull data from one or more tables into the table you are updating.

## Changing data using INSERT

You can use the ON EXISTING clause of the INSERT statement to update existing rows in a table (based on primary key lookup) with new values. This clause can only be used on tables that have a primary key. Attempting to use this clause on tables without primary keys generates a syntax error.

Specifying the ON EXISTING clause causes the server to do a primary key lookup for each input row. If the corresponding row does not exist, it inserts the new row. For rows already existing in the table, you can choose to:

- ◆ generate an error for duplicate key values. This is the default behavior if the ON EXISTING clause is not specified.
- ◆ silently ignore the input row, without generating any errors.
- ◆ update the existing row with the values in the input row

 For more information, see the "INSERT statement" on page 463 of the book *ASA SQL Reference Manual*.



## Deleting data using DELETE

Simple DELETE statements have the following form:

```
DELETE [ FROM ] table-name
WHERE column-name = expression
```

You can also use a more complex form, as follows

```
DELETE [ FROM ] table-name
FROM table-list
WHERE search-condition
```

The WHERE clause

Use the WHERE clause to specify which rows to remove. If no WHERE clause appears, the DELETE statement remove all rows in the table.

The FROM clause

The FROM clause in the second position of a DELETE statement is a special feature allowing you to select data from a table or tables and delete corresponding data from the first-named table. The rows you select in the FROM clause specify the conditions for the delete.

Example

This example uses the sample database. To execute the statements in the example, you should set the option WAIT\_FOR\_COMMIT to OFF. The following statement does this for the current connection only:

```
SET TEMPORARY OPTION WAIT_FOR_COMMIT = 'OFF'
```

This allows you to delete rows even if they contain primary keys referenced by a foreign key, but does not permit a COMMIT unless the corresponding foreign key is deleted also.

The following view displays products and the value of that product that has been sold:

```
CREATE VIEW ProductPopularity as
SELECT product.id,
       SUM(product.unit_price * sales_order_items.quantity)
as "Value Sold"
FROM product JOIN sales_order_items
ON product.id = sales_order_items.prod_id
GROUP BY product.id
```

Using this view, you can delete those products which have sold less than \$20,000 from the *product* table.

```
DELETE
FROM product
FROM product NATURAL JOIN ProductPopularity
WHERE "Value Sold" < 20000
```

You should roll back your changes when you have completed the example:

```
ROLLBACK
```

## Deleting all rows from a table

You can use the TRUNCATE TABLE statement as a fast method of deleting all the rows in a table. It is faster than a DELETE statement with no conditions, because the delete logs each change, while the transaction log does not record truncate table operations individually.

The table definition for a table emptied with the TRUNCATE TABLE statement remains in the database, along with its indexes and other associated objects, unless you execute a DROP TABLE statement.

You cannot use TRUNCATE TABLE if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table.

### TRUNCATE TABLE syntax

The syntax of truncate table is:

**TRUNCATE TABLE** *table-name*

For example, to remove all the data in the *sales\_order* table, type the following:

```
TRUNCATE TABLE sales_order
```

A TRUNCATE TABLE statement does not fire triggers defined on the table.

# Query Optimization and Execution

About this chapter

Once each query is parsed, the optimizer analyzes it and decides on an access plan that will compute the result using as few resources as possible. This chapter describes the steps the optimizer goes through to optimize a query. It documents the assumptions that underlie the design of the optimizer, and discusses selectivity estimation, cost estimation, and the other steps of optimization.

Although update, insert, and delete statements must also be optimized, the focus of this chapter is on select queries. The optimization of these other commands follows similar principles.

Contents

Topic	Page
The role of the optimizer	314
How the optimizer works	315
Query execution algorithms	324
Physical data organization and access	337
Indexes	340
Semantic query transformations	349
Subquery and function caching	363
Reading access plans	364

## The role of the optimizer

The role of the optimizer is to devise an efficient way to execute the SQL statement. The optimizer expresses its chosen method in the form of an **access plan**. The access plan describes which tables to scan, which index, if any, to use for each table, which join strategy to use, and what order to read the tables in. Often, a great number of plans exist that all accomplish the same goal. Other variables may further enlarge the number of possible access plans.

### Cost based

The optimizer begins selecting for the choices available using efficient, and in some cases proprietary, algorithms. It bases its decisions on predictions of the resources each query requires. The optimizer takes into account both the cost of disk access operations and the estimated CPU cost of each operation.

### Syntax independent

Most commands may be expressed in many different ways using the SQL language. These expressions are semantically equivalent in that they accomplish the same task, but may differ substantially in syntax. With few exceptions, the Adaptive Server Anywhere optimizer devises a suitable access plan based only on the semantics of each statement.

Syntactic differences, although they may appear to be substantial, usually have no effect. For example, differences in the order of predicates, tables, and attributes in the query syntax have no effect on the choice of access plan. Neither is the optimizer affected by whether or not a query contains a view.

### A good plan, not necessarily the best plan

The goal of the optimizer is to find a good access plan. Ideally, the optimizer would identify the most efficient access plan possible, but this goal is often impractical. Given a complicated query, a great number of possibilities may exist.

However efficient the optimizer, analyzing each option takes time and resources. The optimizer compares the cost of further optimization with the cost of executing the best plan it has found so far. If a plan has been devised that has a relatively low cost, the optimizer stops and allows execution of that plan to proceed. Further optimization might consume more resources than would execution of an access plan already found.

In the case of expensive and complicated queries, the optimizer works longer. In the case of very expensive queries, it may run long enough to cause a discernable delay.

## How the optimizer works

The Adaptive Server Anywhere optimizer must decide the order in which to access tables in a query, and whether or not to use an index for each table. The optimizer attempts to pick the best strategy.

The best strategy for executing each query is the one that gets the results in the shortest period of time, with the least cost. The optimizer determines the cost of each strategy by estimating the number of disk reads and writes required, and chooses the strategy with the lowest cost.

The optimizer uses a generic disk access cost model to differentiate the relative performance differences between random and sequential retrieval on the database file. It is possible to calibrate a database for a particular hardware configuration using an `ALTER DATABASE` statement. The particulars of a specific cost model can be determined with the `sa_get_dtt()` stored procedure.

By default, query processing is optimized towards returning the first row quickly. You can change this, using the `OPTIMIZATION_GOAL` option, to minimize the cost of returning the complete result set.

You can view the access plan for any query in Interactive SQL by opening the Plan tab in the Results pane. To change the degree of detail that is displayed, change the setting in the Plan tab of the Options dialog (available from the Tools menu).

🔗 For more information about optimization goals, see "OPTIMIZATION\_GOAL option" on page 587 of the book *ASA Database Administration Guide*.

🔗 For more information about reading access plans, see "Reading access plans" on page 364.

## Optimizer estimates

The optimizer chooses a strategy for processing a statement based on histograms that are stored in the database and heuristics (educated guesses).

Histograms, also called column statistics, store information about the distribution of values in a column. In Adaptive Server Anywhere, a histogram represents the data distribution for a column by dividing the domain of the column into a set of consecutive value ranges and by remembering, for each value range, the number of rows in the table for which the column value falls.

Adaptive Server Anywhere pays particular attention to single column values that are represented in a large number of rows in the table. Effectively, any value that is present in more than 1% of the table rows is remembered as a singleton value range in the histogram.

Given the histogram on a column, Adaptive Server Anywhere attempts to estimate the number of rows satisfying a given query predicate on the column by adding up the number of rows in all value ranges that overlap the values satisfying the specified predicate. For value ranges (or buckets) in the histograms that are partially contained in the query result set, Adaptive Server Anywhere uses interpolation within the value range.

Adaptive Server Anywhere uses an implementation of histograms that causes the histograms to be more refined as a byproduct of query execution. As queries are executed, Adaptive Server Anywhere compares the number of rows estimated by the histograms for a given predicate with the number of rows actually found to satisfy the predicate, and then adjusts the values in the histogram to reduce the margin of error for subsequent optimizations.

For each table in a potential execution plan, the optimizer estimates the number of rows that will form part of the results. The number of rows depends on the size of the table and the restrictions in the WHERE clause or the ON clause of the query.

In many cases, the optimizer uses more sophisticated heuristics. For example, the optimizer uses default estimates only in cases where better statistics are unavailable. As well, the optimizer makes use of indexes and keys to improve its guess of the number of rows. The following are a few single-column examples:

- ◆ Equating a column to a value: estimate one row when the column has a unique index or is the primary key.
- ◆ A comparison of an indexed column to a constant: probe the index to estimate the percentage of rows that satisfy the comparison.
- ◆ Equating a foreign key to a primary key (key join): use relative table sizes in determining an estimate. For example, if a 5000 row table has a foreign key to a 1000 row table, the optimizer guesses that there are five foreign key rows for each primary key row.

🔗 For more information about column statistics, see "SYSCOLSTAT system table" on page 608 of the book *ASA SQL Reference Manual*.

🔗 For information about obtaining the selectivities of predicates and the distribution of column values, see:

- ◆ "sa\_get\_histogram system procedure" on page 694 of the book *ASA SQL Reference Manual*

- ◆ "The Histogram utility" on page 461 of the book *ASA Database Administration Guide*
- ◆ "ESTIMATE function" on page 130 of the book *ASA SQL Reference Manual*
- ◆ "ESTIMATE\_SOURCE function" on page 131 of the book *ASA SQL Reference Manual*

## Updating column statistics

Column statistics are stored permanently in the database in the system table SYSCOLSTAT.

Occasionally, these statistics can inaccurately reflect the current column values. This situation arises most commonly following actions that dramatically alter the contents of a table. Examples of such actions are the deletion or insertion of a large number of rows.

In such situations, you may want to execute the statements DROP STATISTICS or CREATE STATISTICS. CREATE STATISTICS deletes old statistics and creates new ones, while DROP STATISTICS just deletes the existing statistics.

With more accurate statistics available to it, the optimizer can compute better estimates, thus improving the performance of subsequent queries. However, incorrect estimates are only a problem if they lead to poorly optimized queries.

When you execute LOAD TABLE, statistics are created for the table. However, when rows are inserted, deleted or updated in a table, the statistics are not updated.

For small tables, a histogram does not significantly improve the optimizer's ability to choose an efficient plan. You can specify the minimum table size for which histograms are created. The default is 1000 rows. However, when a CREATE STATISTICS statement is executed, a histogram is created for every table, regardless of the number of rows.

☞ For more information, see "MIN\_TABLE\_SIZE\_FOR\_HISTOGRAM option" on page 583 of the book *ASA Database Administration Guide*.

☞ For more information about column statistics, see:

- ◆ "SYSCOLSTAT system table" on page 608 of the book *ASA SQL Reference Manual*
- ◆ "DROP STATISTICS statement" on page 406 of the book *ASA SQL Reference Manual*

- ◆ "CREATE STATISTICS statement" on page 323 of the book *ASA SQL Reference Manual*

## Automatic performance tuning

One of the most common constraints in a query is equality with a column value. The following example tests for equality of the `sex` column.

```
SELECT *  
FROM employee  
WHERE sex = 'f'
```

Queries often optimize differently at the second execution. For the above type of constraint, Adaptive Server Anywhere learns from experience, automatically allowing for columns that have an unusual distribution of values. The database stores this information permanently unless you explicitly delete it using the `DROP STATISTICS` command. Note that subsequent queries with predicates over that column may cause the server to recreate a histogram on the column.

## Underlying assumptions

A number of assumptions underlie the design direction and philosophy of the Adaptive Server Anywhere query optimizer. You can improve the quality or performance of your own applications through an understanding of the optimizer's decisions. These assumptions provide a context in which you may understand the information contained in the remaining sections.

### Minimal administration work

Traditionally, high performance database servers have relied heavily on the presence of a knowledgeable, dedicated, database administrator. This person spent a great deal of time adjusting data storage and performance controls of all kinds to achieve good database performance. These controls often required continuing adjustment as the data in the database changed.

Adaptive Server Anywhere learns and adjusts as the database grows and changes. Each query better its knowledge of the data distribution in the database. Adaptive Server Anywhere automatically stores and uses this information to optimize future queries.

Every query both contributes to this internal knowledge and benefits from it. Every user can benefit from knowledge that Adaptive Server Anywhere has gained through executing another user's query.



Statistics-gathering mechanisms are thus an integral part of the database server, and require no external mechanism. Should you find an occasion where it would help, you can provide the database server with estimates of data distributions to use during optimization. If you encode these into a trigger or procedure, for example, you then assume responsibility for maintaining these estimates and updating them whenever appropriate.

### **Optimize for first rows or for entire result set**

The `OPTIMIZATION_GOAL` option allows you to specify whether query processing should be optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set. The default is first row.

🔗 For more information, see "OPTIMIZATION\_GOAL option" on page 587 of the book *ASA Database Administration Guide*.

### **Statistics are present and correct**

The optimizer is self-tuning, storing all the needed information internally. The system table `SYSSTAT` is a persistent repository of data distributions and predicate selectivity estimates. At the completion of each query, Adaptive Server Anywhere uses statistics gathered during query execution to update `SYSSTAT`. In consequence, all subsequent queries gain access to more accurate estimates.

The optimizer relies heavily on these statistics and, therefore, the quality of the access plans it generates depends heavily on them. If you recently inserted a lot of new rows, these statistics may no longer accurately describe the data. You may find that your subsequent queries execute unusually slowly.

If you have significantly altered your data, and you find that query execution is slow, you may want to execute `DROP STATISTICS` and/or `CREATE STATISTICS`.

### **An index can usually be found to satisfy a predicate**

Often, Adaptive Server Anywhere can evaluate predicates with the aid of an index. Using an index, the optimizer speeds access to data and reduces the amount of information read. For example, when `OPTIMIZATION_GOAL` is set to first-row, the Adaptive Server Anywhere optimizer will try to use indexes to satisfy `ORDER BY` and `GROUP BY` clauses.

When the optimizer cannot find a suitable index, it resorts to a sequential table scan, which can be expensive. An index can improve performance dramatically when joining tables. Add indexes to tables or rewrite queries wherever doing so facilitates the efficient processing of common requests.

### Virtual Memory is a scarce resource

The operating system and a number of applications frequently share the memory of a typical computer. Adaptive Server Anywhere treats memory as a scarce resource. Because it uses memory economically, Adaptive Server Anywhere can run on relatively small computers. This economy is important if you wish your database to operate on portable computers or on older machines.

Reserving extra memory, for example to hold the contents of a cursor, may be expensive. If the buffer cache is full, one or more pages may have to be written to disk to make room for new pages. Some pages may need to be re-read to complete a subsequent operation.

In recognition of this situation, Adaptive Server Anywhere associates a higher cost with execution plans that require additional buffer cache overhead. This cost discourages the optimizer from choosing plans that use work tables.

On the other hand, it is careful to use memory where it improves performance. For example, it caches the results of subqueries when they will be needed repeatedly during the processing of the query.

### Rewriting subqueries as EXISTS predicates

The assumptions which underlie the design of Adaptive Server Anywhere require that it conserves memory and that by default it returns the first few results of a cursor as quickly as possible. In keeping with these objectives, Adaptive Server Anywhere rewrites all set-operation subqueries, such as IN, ANY, or SOME predicates, as EXISTS predicates. By doing so, Adaptive Server Anywhere avoids creating unnecessary work tables and may more easily identify a suitable index through which to access a table.

**Non-correlated subqueries** are subqueries that contain no explicit reference to the table or tables contained in the rest higher-level portions of the query.

The following is an ordinary query that contains a non-correlated subquery. It selects information about all the customers who did not place an order on January 1, 2001.

Non-correlated  
subquery

```
SELECT *
FROM customer c
WHERE c.id NOT IN
    ( SELECT o.cust_id
      FROM sales_order o
      WHERE o.order_date = '2001-01-01' )
```

One possible way to evaluate this query is to first read the *sales\_order* table and create a work table of all the customers who placed orders on January 1, 2001, then read the *customer* table and extract one row for each customer listed in the work table.

However, Adaptive Server Anywhere avoids materializing results as work tables. It also gives preference to plans that return the first few rows of a result most quickly. Thus, the optimizer rewrites such queries using EXISTS predicates. In this form, the subquery becomes **correlated**: the subquery now contains an explicit reference to the *id* column of the *customer* table.

Correlated  
subquery

```
SELECT *
FROM customer c
WHERE NOT EXISTS
    ( SELECT *
      FROM sales_order o
      WHERE o.order_date = '2000-01-01'
            AND o.cust_id = c.id )
```

c<seq> : o<key\_so\_customer>

This query is semantically equivalent to the one above, but when expressed in this new syntax, two advantages become apparent.

- 1 The optimizer can choose to use either the index on the *cust\_id* attribute or the *order\_date* attribute of the *sales\_order* table. (However, in the sample database, only the *id* and *cust\_id* columns are indexed.)
- 2 The optimizer has the option of choosing to evaluate the subquery without materializing intermediate results as work tables.

Adaptive Server Anywhere can cache the results of this correlated subquery during processing. This strategy lets Adaptive Server Anywhere reuse previously computed results. In the case of query above, caching does not help because customer identification numbers are unique in the customer table.

🔗 Further information on subquery caching is located in "Subquery and function caching" on page 363.

## Access plan caching

Normally, the optimizer selects an access plan for a query every time the query is executed. Optimizing at execution time allows the optimizer to choose a plan based on current system state, as well as the values of current selectivity estimates and estimates based on the values of host variables. For queries that are executed very frequently, the cost of query optimization can outweigh the benefits of optimizing at execution time. For queries and INSERT, UPDATE and DELETE statements performed inside stored procedures, stored functions, and triggers, the optimizer caches execution plans between executions of the query.

After a statement in a stored procedure, stored function, or trigger has been executed several times by one connection, the optimizer builds a reusable plan for the statement. A reusable plan does not use the values of host variables for selectivity estimation or rewrite optimizations. The reusable plan may have a higher cost because of this. If the cost of the reusable plan is close to the best observed cost for the statement, the optimizer will choose to add the reusable plan to a plan cache. Otherwise, the benefit of optimizing on each execution outweighs the savings from avoiding optimization, and the execution plan is not cached.

The plan cache is a per-connection cache of the data structures used to execute an access plan. Reusing the cached plan involves looking up the plan in the cache and resetting it to an initial state. This is typically substantially faster than optimizing the statement. Cached plans may be stored to disk if they are used infrequently, and they do not increase the cache usage. The optimizer periodically re-optimizes queries to verify that the cached plan is still relatively efficient.

You can use the database/connection property `QueryCachePages` to determine the number of pages used to cache execution plans. These pages occupy space in the temporary file, but are not necessarily resident in memory.

The maximum number of plans to cache is specified with the option setting `MAX_PLANS_CACHED`. The default is 20. To disable plan caching, set this option to 0.

✍ For more information, see "MAX\_PLANS\_CACHED option" on page 581 of the book *ASA Database Administration Guide*.


## Steps in optimization

The steps the Adaptive Server Anywhere optimizer follows in generating a suitable access plan include the following.

- 1 The parser converts the SQL query into an internal representation. It may rewrite the query, converting it to a syntactically different, but semantically equivalent, form. For example, a subquery may be rewritten as a join. These conversions make the statement easier to analyze.
- 2 Optimization proper commences just before execution. If you are using cursors in your application, optimization commences when the cursor is opened. Unlike many other commercial database systems, Adaptive Server Anywhere optimizes each statement just before executing it.
- 3 The optimizer performs semantic optimization on the statement. It rewrites each SQL statement whenever doing so leads to better, more efficient access plans.
- 4 The optimizer performs join enumeration for each subquery.
- 5 The optimizer optimizes access order.

Because Adaptive Server Anywhere performs just-in-time optimization of each statement, the optimizer has access to the values of host variables and stored procedure variables. Hence, it makes better choices because it performs better selectivity analysis.

Adaptive Server Anywhere optimizes each query you execute, regardless of how many times you executed it before, with the exception of queries that are contained in stored procedures or user-defined functions. For queries contained in stored procedures or user-defined functions, the optimizer may cache the access plans so that they can be reused.

 For more information, see "Access plan caching" on page 322.

Because Adaptive Server Anywhere saves statistics each time it executes a query, the optimizer can learn from the experience of executing previous plans and can adjust its choices when appropriate.

## Simple queries

If a query is recognized as a simple query, a heuristic rather than cost-based optimization is used—the optimizer decides whether to use an index scan or sequential table scan, and builds and executes the access plan immediately. Steps 4 and 5 are bypassed.

A simple query is a DYNAMIC SCROLL or NO SCROLL cursor that does not contain any kind of subquery, more than one table, a proxy table, user defined functions, NUMBER(\*), UNION, aggregation, DISTINCT, GROUP BY, or more than one predicate on a single column. Simple queries can contain ORDER BY only as long as the WHERE clause contains conditions on each primary key column.

# Query execution algorithms

The following is an explanation of the algorithms Adaptive Server Anywhere uses to compute queries.

## Accessing tables

The basic ways to access single tables are the index scan and the sequential table scan.

### Index scans

An index scan uses an index to determine which rows satisfy a search condition. It reads only those pages that satisfy the condition. Indexes can return rows in sorted order.

Index scans are displayed in the short and long plan as *correlation\_name*<*index\_name*>, where *correlation\_name* is the correlation name specified in the FROM clause, or the table name if none was specified; and *index\_name* is the name of the index.

Indexes provide an efficient mechanism for reading a few rows from a large table. However, index scans cause pages to be read from the database in random order, which is more expensive than sequential reads. Index scans may also reference the same table page multiple times if there are several rows on the page that satisfy the search condition. If only a few pages are matched by the index scan, it is likely that the pages will remain in cache, and multiple access does not lead to extra I/O. However, if many pages are matched by the search condition, they may not all fit in cache. This can lead to the index scan reading the same page from disk multiple times.

The optimizer will tend to prefer index scans over sequential table scans if the OPTIMIZATION\_GOAL setting is first-row (the default). This is because indexes tend to return the first few rows of a query faster than table scans.

Indexes can also be used to satisfy an ordering requirement, either explicitly defined in an ORDER BY clause, or implicitly needed for a GROUP BY or DISTINCT clause. Ordered group-by and ordered distinct methods can return initial rows faster than hash-based grouping and distinct, but they may be slower at returning the entire result set.

The optimizer uses an index scan to satisfy a search condition if the search condition is sargable, and if the optimizer's estimate of the selectivity of the search condition is sufficiently low for the index scan to be cheaper than a sequential table scan.

An index scan can also evaluate non-sargable search conditions after rows are fetched from the index. Evaluating conditions in the index scan is slightly more efficient than evaluating them in a filter after the index scan.

☞ For more information about when Adaptive Server Anywhere can make use of indexes, see "Predicate analysis" on page 350.

☞ For more information about optimization goals, see "OPTIMIZATION\_GOAL option" on page 587 of the book *ASA Database Administration Guide*.

## Sequential table scans

A sequential table scan reads all the rows in all the pages of a table in the order in which they are stored in the database.

Sequential table scans are displayed in the short and long plan as *correlation\_name*<seq>, where *correlation\_name* is the correlation name specified in the FROM clause, or the table name if none was specified.

This type of scan is used when it is likely that a majority of table pages have a row that match the query's search condition or a suitable index is not defined.

Although sequential table scans may read more pages than index scans, the disk I/O can be substantially cheaper because the pages are read in contiguous blocks from the disk (this performance improvement is best if the database file is not fragmented on the disk). Sequential I/O minimizes overhead due to disk head movement and rotational latency. For large tables, sequential table scans also read groups of several pages at a time. This further reduces the cost of sequential table scans relative to index scans.

Although sequential table scans may take less time than index scans that match many rows, they also cannot exploit the cache as effectively as index scans if the scan is executed many times. Since index scans are likely to access fewer table pages, it is more likely that the pages will be available in the cache, resulting in faster access. Because of this, it is much better to have an index scan for table accesses that are repeated, such as the right hand side of a nested loops join.

For isolation level 3, Adaptive Server Anywhere acquires a lock on each row that is accessed—even if it does not satisfy the search condition. For this level, sequential table scans acquire locks on all of the rows in the table, while index scans only acquire locks on the rows that match the search condition. This means that sequential table scans may substantially reduce the throughput in multi-user environments. For this reason, the optimizer prefers indexed access over sequential access at isolation level 3. Sequential scans can efficiently evaluate simple comparison predicates between table columns and constants during the scan. Other search conditions that refer only to the table being scanned are evaluated after these simple comparisons, and this approach is slightly more efficient than evaluating the conditions in a filter after the sequential scan.

### IN list

The IN list algorithm is used in cases where an IN predicate can be satisfied using an index. For example, in the following query, the optimizer recognizes that it can access the employee table using its primary key index.

```
SELECT *  
FROM employee  
WHERE emp_id in (102, 105, 129)
```

In order to accomplish this, a join is built with a special IN list table on the left hand side. Rows are fetched from the IN list and used to probe the employee table. Multiple IN lists can be satisfied using the same index. If the optimizer chooses not to use an index to satisfy the IN predicate (perhaps because another index gives better performance), then the IN list appears as a predicate in a filter.

### Join algorithms

Join algorithms are required when more than one table appears in the FROM clause. You cannot specify which join algorithm is used—the choice is made by the optimizer.

#### Nested loops join

The nested loops join computes the join of its left and right sides by completely reading the right hand side for each row of the left hand side. (The syntactic order of tables in the query does not matter, because the optimizer chooses the appropriate join order for each block in the request.)

The optimizer may choose nested loops join if the join condition does not contain an equality condition, or if it is optimizing for first-row time.



Since a nested loops join reads the right hand side many times, it is very sensitive to the cost of the right hand side. If the right hand side is an index scan or a small table, then the right hand side can likely be computed using cached pages from previous iterations. On the other hand, if the right hand side is a sequential table scan or an index scan that matches many rows, then the right hand side needs to be read from disk many times. Typically, a nested loops join is less efficient than other join methods. However, nested loops join can provide the first matching row quickly compared to join methods that must compute their entire result before returning.

Nested loops join is the only join algorithm that can provide sensitive semantics for queries containing joins. This means that sensitive cursors on joins can only be executed with a nested loops join.

A semijoin fetches only the first matching row from the right hand side. It is a more efficient version of the nested loops join, but can only be used when an EXISTS, or sometimes a DISTINCT, keyword is used.

### **Nested loops semijoin**

Similar to the nested loop join described above, the nested loops semijoin algorithm joins each row of the left-hand side with the right-hand side using a nested loops algorithm. As with nested loops join, the right-hand side may be read many times, so for larger inputs an index scan is preferable. However, nested loops semijoin differs from nested loop join in two respects. First, semijoin only outputs values from the left-hand side; the right hand side is used only for restricting which rows of the left-hand side appear in the result. Second, the nested loops semijoin algorithm stops each search of the right-hand side as soon as the first match is encountered. Nested loops semijoin can be used as the join algorithm when join's inputs include table expressions from an existentially-quantified (IN, SOME, ANY, EXISTS) nested query that has been rewritten as a join.

## **Nested loops anti-semijoin**

Nested loop anti-semijoin is the second variant of nested loop join supported by Adaptive Server Anywhere. Once again, the right-hand side is used only to determine which rows of the left-hand side appear in the result. Unlike the semijoin algorithm above, however, a row from the left-hand side will become part of the result only if that row does not join with any row from the right-hand side. Nested loops semijoin can be used as the join algorithm when join's inputs include table expressions from a universally-quantified (NOT IN, ALL, NOT EXISTS) nested query that has been rewritten as an anti-join. As with the other two nested loop algorithms, the right-hand side will be accessed for each row of the left-hand side. Hence the algorithms total cost is dependent on the cost of accessing rows from the right-hand side. In most cases, indexed retrieval of the right-hand side will provide the best performance.

## **Nested block join and sorted block**

The nested block join (also called the block nested loops join) reads a block of rows from the left hand side, and sorts the rows by the join attributes (the columns used in the join conditions). The left hand child of a nested block join is called a sorted block node. For each block of rows with equal join attributes, the right hand side is scanned once. This algorithm improves on the nested loops join if there are several rows on the left hand side that join with each row of the right hand side.

A nested block join will be chosen by the optimizer if the left hand side has many rows with the same values for join attributes and the right hand side has an index that satisfies the search condition.

Every nested block join has a left child that is a sorted block node. The cost shown for this node is the cost of reading and sorting the rows from the left input.

The left hand input is read into memory in blocks. Changes to tables in the left hand input may not be visible in the results. Because of this, a nested block join cannot provide sensitive semantics.

Nested block joins locks rows on the left input before they are copied to memory.

## Hash join

The hash join algorithm builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches, which are written to a work table. If the smaller input does not fit into memory, the hash join operator partitions both inputs into smaller work tables. These smaller work tables are processed recursively until the smaller input fits into memory.

The hash join algorithm has the best performance if the smaller input fits into memory, regardless of the size of the larger input. In general, the optimizer will choose hash join if one of the inputs is expected to be substantially smaller than the other.

If the hash join algorithm executes in an environment where there is not enough cache memory to hold all the rows that have a particular value of the join attributes, then it is not able to complete. In this case, the hash join method discards the interim results and an indexed-based nested loops join is used instead. All of the rows of the smaller table are read and used to probe the work table to find matches. This indexed-based strategy is significantly slower than other join methods, and the optimizer will avoid generating access plans using a hash join if it detects that a low memory situation may occur during query execution. When the nested loops strategy is needed due to low memory, a performance counter is incremented. You can read this monitor with the `QueryLowMemoryStrategy` database/connection property, or in the "Query: Low Memory Strategies" counter in Windows Performance Monitor.

*Note:* Windows Performance Monitor may not be available on Windows CE, 95, 98, or ME.

☞ For more information, see `QueryLowMemoryStrategy` in "Connection-level properties" on page 618 of the book *ASA Database Administration Guide*.

The hash join algorithm computes all of the rows of its result before returning the first row.

The hash join algorithm uses a work table, which provides insensitive semantics unless a value-sensitive cursor has been requested.

Hash join locks rows in its inputs before they are copied to memory.

## Hash semijoin

The hash semijoin variant of the hash join algorithm performs a semijoin between the left-hand side and the right-hand side. As with nested loop semijoin described above, the right-hand side is only used to determine which rows from the left-hand side appear in the result. With hash semijoin the right-hand side is read to form an in-memory hash table which is subsequently probed by each row from the left-hand side. As soon as any match is found, the left-hand row is output to the result and the match process starts again for the next left-hand row. At least one equality join condition must be present in order for hash semijoin to be considered by the query optimizer. As with nested loop semijoin, hash semijoin will be utilized in cases where the join's inputs include table expressions from an existentially-quantified (IN, SOME, ANY, EXISTS) nested query that has been rewritten as a join. Hash semijoin will tend to outperform nested loop semijoin when the join condition includes inequalities, or if a suitable index does not exist to make indexed retrieval of the right-hand side sufficiently inexpensive.

As with hash join, the hash semijoin algorithm may revert to a nested loops semijoin strategy if there is insufficient cache memory to enable the operation to complete. Should this occur, a performance counter is incremented. You can read this monitor with the `QueryLowMemoryStrategy` database/connection property, or in the `Query: Low Memory Strategies` counter in Windows Performance Monitor.

*Note:* Windows Performance Monitor may not be available on Windows CE, 95, 98, or ME.

☞ For more information, see `QueryLowMemoryStrategy` in "Connection-level properties" on page 618 of the book *ASA Database Administration Guide*.

## Hash antisemijoin

The hash anti-semijoin variant of the hash join algorithm performs an anti-semijoin between the left-hand side and the right-hand side. As with nested loop anti-semijoin described above, the right-hand side is only used to determine which rows from the left-hand side appear in the result. With hash anti-semijoin the right-hand side is read to form an in-memory hash table which is subsequently probed by each row from the left-hand side. Each left-hand row is output only if it fails to match any row from the right-hand side. As with nested loop anti-semijoin, hash anti-semijoin will be utilized in cases where the join's inputs include table expressions from a universally-quantified (NOT IN, ALL, NOT EXISTS) nested query that has been rewritten as an anti-join. Hash anti-semijoin will tend to outperform nested loop anti-semijoin when the join condition includes inequalities, or if a suitable index does not exist to make indexed retrieval of the right-hand side sufficiently inexpensive.

As with hash join, the hash anti-semijoin algorithm may revert to a nested loops anti-semijoin strategy if there is insufficient cache memory to enable the operation to complete. Should this occur, a performance counter is incremented. You can read this monitor with the `QueryLowMemoryStrategy` database/connection property, or in the Query: Low Memory Strategies counter in Windows Performance Monitor.

*Note:* Windows Performance Monitor may not be available on Windows CE, 95, 98, or ME.

☞ For more information, see `QueryLowMemoryStrategy` in "Connection-level properties" on page 618 of the book *ASA Database Administration Guide*.

## Merge join

The merge join reads two inputs which are both ordered by the join attributes. For each row of the left input, the algorithm reads all of the matching rows of the right input by accessing the rows in sorted order.

If the inputs are not already ordered by the join attributes (perhaps because of an earlier merge join or because an index was used to satisfy a search condition), then the optimizer adds a sort to produce the correct row order. This sort adds cost to the merge join.

One advantage of a merge join compared to a hash join is that the cost of sorting can be amortized over several joins, provided that the merge joins are over the same attributes. The optimizer will choose merge join over a hash join if the sizes of the inputs are likely to be similar, or if it can amortize the cost of the sort over several operations.

## Duplicate elimination

A duplicate elimination operator produces an output that has no duplicate rows. Duplicate elimination nodes may be introduced by the optimizer, for example, when converting a nested query into a join.

☞ For more information, see "Hash distinct" on page 332, "Ordered distinct" on page 332, and "Indexed distinct" on page 333.

### Hash distinct

The hash distinct algorithm reads its input, and builds an in-memory hash table. If an input row is found in the hash table, it is ignored; otherwise it is written to a work table. If the input does not completely fit into the in-memory hash table, it is partitioned into smaller work tables, and processed recursively.

The hash distinct algorithm works very well if the distinct rows fit into an in-memory table, irrespective of the total number of rows in the input.

The hash distinct uses a work table, and as such can provide insensitive or value sensitive semantics.

If the hash distinct algorithm executes in an environment where there is very little cache memory available, then it will not be able to complete. In this case, the hash distinct method discards its interim results, and the indexed distinct algorithm is used instead. The optimizer avoids generating access plans using the hash distinct algorithm if it detects that a low memory situation may occur during query execution.

The hash distinct returns a row as soon as it finds one that has not previously been returned. However, the results of a hash distinct must be fully materialized before returning from the query. If necessary, the optimizer adds a work table to the execution plan to ensure this.

Hash distinct locks the rows of its input.

### Ordered distinct

If the input is ordered by all the columns, then an ordered distinct can be used. This algorithm reads each row and compares it to the previous row. If it is the same, it is ignored; otherwise, it is output. The ordered distinct is effective if rows are already ordered (perhaps because of an index or a merge join); if the input is not ordered, the optimizer inserts a sort. No work table is used by the ordered distinct itself, but one is used by any inserted sort.

## Indexed distinct

The indexed distinct algorithm maintains a work table of the unique rows from the input. As rows are read from the input, an index on the work table is searched to find a previously seen duplicate of the input row. If one is found, the input row is ignored. Otherwise, the input row is inserted into the work table. The work table index is created on all the columns of the SELECT list; in order to improve index performance, a hash expression is included as the first expression. This hash expression is a computed value embodying the values of all the columns in the SELECT list.

The indexed distinct method returns distinct rows as they are encountered. This allows it to return the first few rows quickly compared to other duplicate elimination methods. The indexed distinct algorithm only stores two rows in memory at a time, and can work well in extremely low memory situations. However, if the number of distinct rows is large, the execution cost of the indexed distinct algorithm is typically worse than hash distinct. The work table used to store distinct rows may not fit in cache, leading to rereading work table pages many times in a random access pattern.

Since the indexed distinct method uses a work table, it cannot provide fully sensitive semantics; however, it also does not provide fully insensitive semantics, and another work table is required for insensitive cursors.

The indexed distinct method locks the rows of its input.

## Grouping

Grouping algorithms compute a summary of their input. They are applicable only if the query contains a GROUP BY clause, or if the query contains aggregate functions (such as `SELECT COUNT(*) FROM T`).

 For more information, see "Hash group by" on page 334, "Ordered group by" on page 334, "Indexed group by" on page 334, and "Single group by" on page 334.

## Hash group by

The hash group by algorithm creates an in-memory hash table of group rows. As rows are read from the input, the group rows are updated. If the hash table doesn't fit into memory, the input is partitioned into smaller work tables which are recursively partitioned until they fit into memory. If there is not enough memory for the partitions, the optimizer discards the interim results from the hash group by, and uses an indexed group by algorithm. The optimizer avoids generating access plans using the hash group by algorithm if it detects that a low memory situation may occur during query execution.

The hash group by algorithm works very well if the groups fit into memory, irrespective of the size of the input.

The hash group by algorithm computes all of the rows of its result before returning the first row, and can be used to satisfy a fully sensitive or values sensitive cursor. The results of the hash group by must be fully materialized before returning from the query. If necessary, the optimizer adds a work table to the execution plan to ensure this.

## Ordered group by

The ordered group by reads an input that is ordered by the grouping columns. As each row is read, it is compared to the previous row. If the grouping columns match, then the current group is updated; otherwise, the current group is output and a new group is started.

## Indexed group by

The indexed group by algorithm is similar to the indexed distinct algorithm. It builds a work table containing one row per group. As input rows are read, the associated group is looked up in the work table using an index. The aggregate functions are updated, and the group row is rewritten to the work table. If no group record is found, a new group record is initialized and inserted into the work table.

Indexed group by is chosen when the optimizer is reasonably certain that the size of the input is very small.

The indexed group by computes all the rows of its result before returning the first row, and fully materializes its input. It can be used to satisfy a FULLY INSENSITIVE requirement.


## Single group by

When no GROUP BY is specified, a single row aggregate is produced.



## Sorting and unions

Sorting algorithms are applicable when the query includes an order by clause, and union algorithms are applicable for union queries.

 For more information, see "Merge sort" on page 335 and "Union all" on page 335.

### Merge sort

The sort operator reads its input into memory, sorts it in memory, and then outputs the sorted results. If the input does not completely fit into memory, then several sorted runs are created and then merged together. Sort does not return any rows until it has read all of the input rows. Sort locks its input rows.

If the merge sort algorithm executes in an environment where there is very little cache memory available, it may not be able to complete. In this case, the merge sort orders the remainder of the input using an indexed-based sort method. Input rows are read and inserted into a work table, and an index is built on the ordering columns of the work table. In this case, rows are read from the work table using a complex index scan. This indexed-based strategy is significantly slower than other join methods. The optimizer avoids generating access plans using a merge sort algorithm if it detects that a low memory situation may occur during query execution. When the index-based strategy is needed due to low memory, a performance counter is incremented; you can read this monitor with the `QueryLowMemoryStrategy` property, or in the "Query: Low Memory Strategies" counter in Windows Performance Monitor.

Sort performance is affected by the size of the sort key, the row size, and the total size of the input. For large rows, it may be cheaper to use a `VALUES SENSITIVE` cursor. In that case, columns in the `SELECT` list are not copied into the work tables used by the sort. While the sort does not write output rows to a work table, the results of the sort must be materialized before rows are returned to the application. If necessary, the optimizer adds a work table to ensure this.

### Union all

The union all algorithm reads rows from each of its inputs and outputs them, regardless of duplicates. This algorithm is used to implement `UNION` and `UNION ALL` clauses. In the `UNION` case, a duplicate elimination algorithm is needed to remove any duplicates generated by the union all.

## Miscellaneous

The following are additional methods that may be used in an access plan.

### Filter, pre-filter and hash-filter

Filters apply search conditions. The search conditions appear in the statement in the WHERE and HAVING clauses and in the ON conditions of JOINS in the FROM clause.

Pre-filter is the same as filter, except that it doesn't depend on input. For example, in the case of WHERE  $1 = 2$  a pre-filter applies.

A hash-filter can be used when an execution plan fetches all of the rows of one table before fetching any rows from another table that joins to it. The hash filter builds an array of bits, and turns one bit on for each row of the first table that matches search conditions on that table. When accessing the second table, rows that could not possibly join with the first table are rejected.

For example, consider the plan:

```
R<idx> *JH S<seq> JH* T<idx>
```

Here we are joining R to S and T. We will have read all of the rows of R before reading any row from T, and we can immediately reject rows of T that can not possibly join with R. This reduces the number of rows that must be stored in the second hash join.

☞ For more information, see "The WHERE clause: specifying rows" on page 195.

### Lock

Lock indicates that there is a lock at a certain isolation level. For example, at isolation level 1, a lock is maintained for only one row at a time. If you are at isolation level 0, no lock is acquired, but the node will still be called Lock. In this case, the lock node verifies that the row still exists.

☞ For more information, see "How locking works" on page 121.

### Row limit

Row limits are set by the TOP n or FIRST clause of the SELECT statement.

☞ For more information, see "SELECT statement" on page 526 of the book *ASA SQL Reference Manual*.

## Physical data organization and access

Storage allocations for each table or entry have a large impact on the efficiency of queries. The following points are of particular importance because each influence how fast your queries execute.

### Disk allocation for inserted rows

Adaptive Server Anywhere stores rows contiguously, if possible

Every new row that is smaller than the page size of the database file will always be stored on a single page. If no present page has enough free space for the new row, Adaptive Server Anywhere writes the row to a new page. For example, if the new row requires 600 bytes of space but only 500 bytes are available on a partially filled page, then Adaptive Server Anywhere places the row on a new page.

To make table pages more contiguous on the disk, Adaptive Server Anywhere allocates table pages in blocks of eight pages. For example, when it needs to allocate a page it allocates eight pages, inserts the page in the block, and then fills up with the block with the next seven pages. In addition, it uses a free page bitmap to find contiguous blocks of pages within the dbspace, and performs sequential scans by reading groups of 64K, using the bitmap to find relevant pages. This leads to more efficient sequential scans.

Adaptive Server Anywhere may store rows in any order

Adaptive Server Anywhere locates space on pages and inserts rows in the order it receives them in. It assigns each to a page, but the locations it chooses in the table may not correspond to the order they were inserted in. For example, the engine may have to start a new page to store a long row contiguously. Should the next row be shorter, it may fit in an empty location on a previous page.

The rows of all tables are unordered. If the order that you receive or process the rows is important, use an `ORDER BY` clause in your `SELECT` statement to apply an ordering to the result. Applications that rely on the order of rows in a table can fail without warning.

If you frequently require the rows of a table to be in a particular order, consider creating an index on those columns specified in the query's `ORDER BY` clause.

Space is not reserved for NULL columns

Whenever Adaptive Server Anywhere inserts a row, it reserves only the space necessary to store the row with the values it contains at the time of creation. It reserves no space to store values that are NULL. It reserves no extra space to accommodate fields, such as text strings, which may enlarge.

Once inserted, rows identifiers are immutable

Once assigned a home position on a page, a row never moves from that page. If an update changes any of the values in the row so it no longer fits in its assigned page, then the row splits and the extra information is inserted on another page.

This characteristic deserves special attention, especially since Adaptive Server Anywhere allows no extra space when you insert the row. For example, suppose you insert a large number of empty rows into a table, then fill in the values, one column at a time, using update statements. The result would be that almost every value in a single row will be stored on a separate page. To retrieve all the values from one row, the engine may need to read several disk pages. This simple operation would become extremely and unnecessarily slow.

You should consider filling new rows with data at the time of insertion. Once inserted, they then have sufficient room for the data you expect them to hold.

A database file never shrinks

As you insert and delete rows from the database, Adaptive Server Anywhere automatically reuses the space they occupy. Thus, Adaptive Server Anywhere may insert a row into space formerly occupied by another row.

Adaptive Server Anywhere keeps a record of the amount of empty space on each page. When you ask it to insert a new row, it first searches its record of space on existing pages. If it finds enough space on an existing page, it places the new row on that page, reorganizing the contents of the page if necessary. If not, it starts a new page.

Over time, however, if you delete a number of rows and don't insert new rows small enough to use the empty space, the information in the database may become sparse. You can reload the table, or use the `REORGANIZE TABLE` statement to defragment the table.

🔗 For more information, see "REORGANIZE TABLE statement" on page 508 of the book *ASA SQL Reference Manual*.

## Table and page sizes

The page size you choose for your database can affect the performance of your database. In general, smaller page sizes are likely to benefit operations that retrieve a relatively small number of rows from random locations. By contrast, larger pages tend to benefit queries that perform sequential table scans, particularly when the rows are stored on pages in the order the rows are retrieved via an index. In this situation, reading one page into memory to obtain the values of one row may have the side effect of loading the contents of the next few rows into memory. Often, the physical design of disks permits them to retrieve fewer large blocks more efficiently than many small ones.

Adaptive Server Anywhere creates a bitmap for sufficiently large tables within databases that have at least a 2K page size. Each table's bitmap reflects the position of each table page in the entire dbspace file. For databases of 2K, 4K, or 8K pages, the server utilizes the bitmap to read large blocks (64K) of table pages instead of single pages at a time, reducing the total number of I/O operations to disk and hence improving performance. Users cannot control the server's criteria for bitmap creation or usage.

Note that bitmaps, also called page maps, are only available for databases created in version 8.0 and higher. If a database is upgraded from an older version, the server will not create a bitmap for database tables, even if they meet its criteria. Bitmaps are not created for work tables or system tables.

Should you choose a larger page size, such as 4 kb, you may wish to increase the size of the cache. Fewer large pages can fit into the same space. For example, 1 Mb of memory can hold 1000 pages that are each 1 kb in size, but only 250 pages that are 4 kb in size. How many pages is enough depends entirely on your database and the nature of the queries your application performs. You can conduct performance tests with various cache sizes. If your cache cannot hold enough pages, performance suffers as Adaptive Server Anywhere begins swapping frequently-used pages to disk.

Page sizes also affect indexes. By default, index pages have a hash size of 10 bytes: they store approximately the first 10 bytes of data for each index entry. This allows for a fan-out of roughly 200 using 4K pages, meaning that each index page holds 200 rows, or 40 000 rows with a two-level index. Each new level of an index allows for a table 200 times larger. Page size can significantly affect fan-out, in turn affecting the depth of index required for a table. Large databases should have 4K pages.

Adaptive Server Anywhere attempts to fill pages as much as possible. Empty space accumulates only when new objects are too large to fit empty space on existing pages. Consequently, adjusting the page size may not significantly affect the overall size of your database.

# Indexes


Indexes can greatly improve the performance of searches on the indexed column(s). However, indexes take up space within the database and slow down insert, update, and delete operations. This section will help you to determine when you should create an index and tell you how to achieve maximum performance from your index.

There are many situations in which creating an index improves the performance of a database. An index provides an ordering of the rows of a table on the basis of the values in some or all of the columns. An index allows Adaptive Server Anywhere to find rows quickly. It permits greater concurrency by limiting the number of database pages accessed. An index also affords Adaptive Server Anywhere a convenient means of enforcing a uniqueness constraint on the rows in a table.

## When to create an index

There is no simple formula to determine whether or not an index should be created for a particular column. You must consider the tradeoff of the benefits of indexed retrieval versus the maintenance overhead of that index. The following factors may help to determine whether you should create an index.

- ◆ **Keys and unique columns** Adaptive Server Anywhere automatically creates indexes on primary keys, foreign keys, and unique columns. You should not create additional indexes on these columns. The exception is composite keys, which can sometimes be enhanced with additional indexes.

 For more information, see "Composite indexes" on page 342.

- ◆ **Frequency of search** If a particular column is searched frequently, you can achieve performance benefits by creating an index on that column. Creating an index on a column that is rarely searched may not be worthwhile.
- ◆ **Size of table** Indexes on relatively large tables with many rows provide greater benefits than indexes on relatively small tables. For example, a table with only 20 rows is unlikely to benefit from an index, since a sequential scan would not take any longer than an index lookup.

- ◆ **Number of updates** An index is updated every time a row is inserted or deleted from the table and every time an indexed column is updated. An index on a column slows the performance of inserts, updates and deletes. A database that is frequently updated should have fewer indexes than one that is read-only.
- ◆ **Space considerations** Indexes take up space within the database. If database size is a primary concern, you should create indexes sparingly.
- ◆ **Data distribution** If an index lookup returns too many values, it is more costly than a sequential scan. Adaptive Server Anywhere does not make use of the index when it recognizes this condition. For example, Adaptive Server Anywhere would not make use of an index on a column with only two values, such as *employee.sex* in the sample database. For this reason, you should not create an index on a column that has only a few distinct values.

### Temporary tables

You can create indexes on both local and global temporary tables. You may want to consider indexing a temporary table if you expect it will be large and accessed several times in sorted order or in a join. Otherwise, any improvement in performance for queries is likely to be outweighed by the cost of creating and dropping the index.

☞ For more information, see "Working with indexes" on page 58.

## Improving index performance

If your index is not performing as well as expected, you may want to consider the following actions.

- ◆ Reorganize composite indexes.
- ◆ Increase the page size.

These measures are aimed at increasing index selectivity and index fan-out, as explained below.

### Index selectivity

**Index selectivity** refers to the ability of an index to locate a desired index entry without having to read additional data.

If selectivity is low, additional information must be retrieved from the table page that the index references. These retrievals are called **full compares**, and they have a negative effect on index performance.

The *FullCompare* property function keeps track of the number of full compares that have occurred. You can also monitor this statistic using the Sybase Central Performance monitor or the Windows Performance Monitor.

*Note:* Windows Performance Monitor may not be available on Windows CE, 95, 98, or ME.

In addition, the number of full compares is provided in the graphical plan with statistics. For more information, see "Common statistics used in the plan" on page 365.

🔗 For more information on the *FullCompare* function, see "Database-level properties" on page 630 of the book *ASA Database Administration Guide*.

### Index structure and index fan-out

Indexes are organized in a number of levels, like a tree. The first page of an index, called the root page, branches into one or more pages at the next level, and each of those pages branch again, until the lowest level of the index is reached. These lowest level index pages are called leaf pages. To locate a specific row, an index with  $n$  levels requires  $n$  reads for index pages and one read for the data page containing the actual row. In general, fewer than  $n$  reads from disk are needed, since index pages that are used frequently tend to be stored in cache.

The **index fan-out** is the number of index entries stored on a page. An index with a higher fan-out may have fewer levels than an index with a lower fan-out. Therefore, higher index fan-out generally means better index performance.

You can see the number of levels in an index by using the *sa\_index\_levels* system procedure.

🔗 For more information, see "sa\_index\_levels system procedure" on page 697 of the book *ASA SQL Reference Manual*.

## Composite indexes

An index can contain one, two, or more columns. An index on two or more columns is called a **composite index**. For example, the following statement creates a two-column composite index:

```
CREATE INDEX name
ON employee (emp_lname, emp_fname)
```

A composite index is useful if the first column alone does not provide high selectivity. For example, a composite index on *emp\_lname* and *emp\_fname* is useful when many employees have the same last name. A composite index on *emp\_id* and *emp\_lname* would not be useful because each employee has a unique ID, so the column *emp\_lname* does not provide any additional selectivity.



Additional columns in an index can allow you to narrow down your search, but having a two-column index is not the same as having two separate indexes. A composite index is structured like a telephone book, which first sorts people by their last names, and then all the people with the same last name by their first names. A telephone book is useful if you know the last name, even more useful if you know both the first name and the last name, but worthless if you only know the first name and not the last name.

The compressed B-tree index method that is introduced with Adaptive Server Anywhere version 8 substantially improves performance for composite indexes.

## Column order

When you create composite indexes, you should think carefully about the order of the columns. Composite indexes are useful for doing searches on all of the columns in the index or on the first columns only; they are not useful for doing searches on any of the later columns alone.

If you are likely to do many searches on one column only, that column should be the first column in the composite index. If you are likely to do individual searches on both columns of a two-column index, you may want to consider creating a second index that contains the second column only.

Primary keys that have more than one column are always automatically indexed as composite indexes with their columns in the order that they appear in the table definition, not in the order that they are specified in the primary key definition. You should consider the searches that you will execute involving the primary key to determine which column should come first. Consider adding an extra index on any later column of the primary key that is frequently searched.

For example, suppose you create a composite index on two columns. One column contains employee's first names, the other their last names. You could create an index that contains their first name, then their last name. Alternatively, you could index the last name, then the first name. Although these two indexes organize the information in both columns, they have different functions.

```
CREATE INDEX fname_lname
ON employee emp_fname, emp_lname;

CREATE INDEX lname_fname
ON employee emp_lname, emp_fname;
```

Suppose you then want to search for the first name John. The only useful index is the one containing the first name in the first column of the index. The index organized by last name then first name is of no use because someone with the first name John could appear anywhere in the index.

If you think it likely that you will need to look up people by first name only or second name only, then you should consider creating both of these indexes.

Alternatively, you could make two indexes, each containing only one of the columns. Remember, however, that Adaptive Server Anywhere only uses one index to access any one table while processing a single query. Even if you know both names, it is likely that Adaptive Server Anywhere will need to read extra rows, looking for those with the correct second name.

When you create an index using the `CREATE INDEX` command, as in the example above, the columns appear in the order shown in your command.

Primary key indexes and column order

The order of the columns in a primary key index is enforced to be the same as the order in which the columns appear in the table's definition, regardless as to the ordering of the columns specified in the `PRIMARY KEY` constraint. Moreover, Adaptive Server Anywhere enforces an additional constraint that a table's primary key columns must be at the beginning of each row. Hence if a primary key is added to an existing table the server may rewrite the entire table to ensure that the key columns are at the beginning of each row.

In situations where more than one column appears in a primary key, you should consider the types of searches needed. If appropriate, switch the order of the columns in the table definition so the most frequently searched-for column appears first, or create separate indexes, as required, for the other columns.

Composite indexes and ORDER BY

By default, the columns of an index are sorted in ascending order, but they can optionally be sorted in descending order by specifying `DESC` in the `CREATE INDEX` statement.

Adaptive Server Anywhere can choose to use an index to optimize an `ORDER BY` query as long as the `ORDER BY` clause contains only columns included in that index. In addition, the columns in the index must be ordered in exactly the same way, or in exactly the opposite way, as the `ORDER BY` clause. For single-column indexes, the ordering is always such that it can be optimized, but composite indexes require slightly more thought. The table below shows the possibilities for a two-column index.

Index columns	Optimizable ORDER BY queries	Not optimizable ORDER BY queries
ASC, ASC	ASC, ASC or DESC, DESC	ASC, DESC or DESC, ASC
ASC, DESC	ASC, DESC or DESC, ASC	ASC, ASC or DESC, DESC
DESC, ASC	DESC, ASC or ASC, DESC	ASC, ASC or DESC, DESC
DESC, DESC	DESC, DESC or ASC, ASC	ASC, DESC or DESC, ASC

An index with more than two columns follows the same general rule as above. For example, suppose you have the following index:

```
CREATE INDEX idx_example
```

```
ON table1 (col1 ASC, col2 DESC, col3 ASC)
```

In this case, the following queries can be optimized:

```
SELECT col1, col2, col3 from table1  
ORDER BY col1 ASC, col2 DESC, col3 ASC
```

and

```
SELECT col1, col2, col3 from example  
ORDER BY col1 DESC, col2 ASC, col3 DESC
```


The index is not used to optimize a query with any other pattern of ASC and DESC in the ORDER BY clause. For example:

```
SELECT col1, col2, col3 from table1  
ORDER BY col1 ASC, col2 ASC, col3 ASC
```

is not optimized.

## Other uses for indexes

Adaptive Server Anywhere uses indexes to achieve other performance benefits. Having an index allows Adaptive Server Anywhere to enforce column uniqueness, to reduce the number of rows and pages that must be locked, and to better estimate the selectivity of a predicate.

- ◆ **Enforce column uniqueness** Without an index, Adaptive Server Anywhere has to scan the entire table every time that a value is inserted to ensure that it is unique. For this reason, Adaptive Server Anywhere automatically builds an index on every column with a uniqueness constraint.
- ◆ **Reduce locks** Indexes reduce the number of rows and pages that must be locked during inserts, updates, and deletes. This reduction is a result of the ordering that indexes impose on a table.  
 For more information on indexes and locking, see "How locking works" on page 121.
- ◆ **Estimate selectivity** Because an index is ordered, the optimizer can estimate the percentage of values that satisfy a given query by scanning the upper levels of the index. This action is called a partial index scan.

## Types of index

Adaptive Server Anywhere supports two types of index, and automatically chooses between them depending on the declared width of the indexed columns. For a total column width that is less than 10 bytes, Adaptive Server Anywhere uses a B-tree index that contains an order-preserving encoding, or hash value, that represents the indexed data. Hash B-tree indexes are also used when the index key length is longer than one-eighth of the page size for the database or 256 bytes (whichever is lower). For data values whose combined declared length is between these two bounds, Adaptive Server Anywhere uses a compressed B-tree index that stores each key in a compressed form.

### Hash B-tree indexes

A hash B-tree index does not store the actual row value(s) from the table. Instead, a hash B-tree index stores an order-preserving encoding of the original data. The number of bytes in each index entry used to store this hash value is termed the hash size, and is automatically chosen by the server based on the declared width of all of the indexed columns. The server compares these hashed values as it searches through an index to find a particular row.

When you index a small storage type, such as an integer, the hash value that Adaptive Server Anywhere creates takes the same amount of space as the original value. For example, the hash value for an integer is 4 bytes in size, the same amount of space as required to store an integer. Because the hash value is the same size, Adaptive Server Anywhere can use hash values with a one-to-one correspondence to the actual value. Adaptive Server Anywhere can always tell whether two values are equal, or which is greater by comparing their hash values. However, it can retrieve the actual value only by reading the row from the corresponding table.

When you index a column containing larger data types, the hash value will often be shorter than the size of the type. For example, if you index a column of string values, the hash value used is at most 9 bytes in length.

Consequently, Adaptive Server Anywhere cannot always compare two strings using only the hash values. If the hash values are equal, Adaptive Server Anywhere must retrieve and compare the two actual values from their corresponding rows in the table.

### Hash values

Adaptive Server Anywhere must represent values in an index to decide how to order them. For example, if you index a column of names, then it must know that Amos comes before Smith.

For each value in your index, Adaptive Server Anywhere creates a corresponding hash value. It stores the hash value in the index, rather than the actual value. Adaptive Server Anywhere can perform operations with the hash value. For example, it can tell when two values are equal or which of two values is greater.

When you index a small storage type, such as an integer, the hash value that Adaptive Server Anywhere creates takes the same amount of space as the original value. For example, the hash value for an integer is 4 bytes in size, the same amount of space as required to store an integer. Because the hash value is the same size, Adaptive Server Anywhere can use hash values with a one-to-one correspondence to the actual value. Adaptive Server Anywhere can always tell whether two values are equal, or which is greater by comparing their hash values. However, it can retrieve the actual value only by reading the entry from the corresponding table.

When you index a column containing larger data types, the hash value will often be shorter than the size of the type. For example, if you index a column of string values, the hash value used is at most 9 bytes in length.

Consequently, Adaptive Server Anywhere cannot always compare two strings using only the hash values. If the hash values are equal, Adaptive Server Anywhere must retrieve and compare the actual two values from the table.

For example, suppose you index the titles of books, many of which are similar. If you wish to search for a particular title, the index may identify only a set of possible rows. In this case, Adaptive Server Anywhere must retrieve each of the candidate rows and examine the full title.

## Composite indexes

An ordered sequence of columns is also called a composite index. However, each index key in these indexes is at most a 9 byte hash value. Hence, the hash value cannot necessarily identify the correct row uniquely. When two hash values are equal, Adaptive Server Anywhere must retrieve and compare the actual values.

## **Compressed B-tree indexes**

Compressed B-tree indexes store a compressed form of each indexed value in the index's internal nodes. To do this, compressed B-tree indexes store the values using Patricia tries, an optimized form of a trie data structure that is augmented with a skip-count to compress its representation. As a result, compressed B-tree indexes offer substantial improvements over hash indexes when the overall data length is reasonably large. More significantly, the compaction algorithm efficiently handles index values that are identical (or nearly so), so common substrings within the indexed values have negligible impact on storage requirements and performance. Compressed B-tree indexes are chosen automatically if the sum of the declared width of the indexed columns is between 10 bytes and one-eighth of the database page size to a maximum of 256 bytes.

## **Recommended page sizes**

The page size of the database can have a significant effect on the index fan-out. The index fan-out approximately doubles as the page size doubles.

Each index lookup requires one page read for each of the levels of the index plus one page read for the table page, and a single query can require several thousand index lookups. A large fan-out often means that fewer index levels are required, which can improve searches considerably. For this reason, consider using a large page size, such as 4K, to improve index performance. You may also want to consider using a larger page size when you wish to index long string columns using compressed B-tree indexes, but the size limit on smaller page sizes is preventing their creation.

## Semantic query transformations

To operate efficiently, Adaptive Server Anywhere usually rewrites your query, possibly in several steps, into a new form. It ensures that the new version computes the same result, even though it expresses the query in a new way. In other words, Adaptive Server Anywhere rewrites your queries into semantically equivalent, but syntactically different, forms.

Adaptive Server Anywhere can perform a number of different rewrite operations. If you read the access plans, you will frequently find that they do not correspond to a literal interpretation of your original statement. For example, the optimizer tries as much as possible to rewrite subqueries with joins. The fact that the optimizer has the freedom to rewrite your SQL statements and some of the ways in which it does so, are of importance to you.

### Example

Unlike the SQL language definition, some languages mandate strict behavior for AND and OR operations. Some guarantee that the left-hand condition will be evaluated first. If the truth of the entire condition can then be determined, the compiler guarantees that the right-hand condition will not be evaluated.

This arrangement lets you combine conditions that would otherwise require two nested IF statements into one. For example, in C you can test whether a pointer is NULL before you use it as follows. You can replace the nested conditions

```
if ( X != NULL ) {
    if ( X->var != 0 ) {
        ... statements ...
    }
}
```

with the more compact expression

```
if ( X != NULL && X->var != 0 ) {
    ... statements ...
}
```

Unlike C, SQL has no such rules concerning execution order. Adaptive Server Anywhere is free to rearrange the order of such conditions as it sees fit. The reordered form is semantically equivalent because the SQL language specification makes no distinction. In particular, query optimizers are completely free to reorder predicates in a WHERE, HAVING, and ON clause.

## Predicate analysis

A **predicate** is a conditional expression that, combined with the logical operators AND and OR, makes up the set of conditions in a WHERE, HAVING, or ON clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

A predicate that can exploit an index to retrieve rows from a table is called **sargable**. This name comes from the phrase *search argument-able*. Predicates that involve comparisons of a column with constants, other columns, or expressions may be sargable.

The predicate in the following statement is sargable. Adaptive Server Anywhere can evaluate it efficiently using the primary index of the employee table.

```
SELECT *
FROM employee
WHERE employee.emp_id = 123

employee<employee>
```

In contrast, the following predicate is *not* sargable. Although the *emp\_id* column is indexed in the primary index, using this index does not expedite the computation because the result contains all, or all except one, row.

```
SELECT *
FROM employee
where employee.emp_id <> 123

employee<seq>
```

Similarly, no index can assist in a search for all employees whose first name *ends* in the letter "k". Again, the only means of computing this result is to examine each of the rows individually.

## Functions

In general, a predicate that has a function on the column name is not sargable. For example, an index would not be used on the following query:

```
SELECT * from sales_order
WHERE year(order_date)='2000'
```

You can sometimes rewrite a query to avoid using a function, thus making it sargable. For example, you can rephrase the above query:

```
SELECT * from sales_order
WHERE order_date > '1999-12-31'
AND order_date < '2001-01-01'
```



A query that uses a function becomes sargable if you store the function values in a computed column and build an index on this column. A **computed column** is a column whose values are obtained from other columns in the table. For example, if you have a column called *order\_date* that holds the date of an order, you can create a computed column called *order\_year* that holds the values for the year extracted from the *order\_date* column.

```
ALTER TABLE sales_order
ADD order_year INTEGER
COMPUTE year(order_date)
```

You can then add an index on the column *order\_year* in the ordinary way:

```
CREATE INDEX idx_year
ON sales_order (order_year)
```

If you then execute the following statement

```
SELECT * from sales_order
WHERE year(order_date) = '2000'
```

the server recognizes that there is an indexed column that holds that information and uses that index to answer the query.

The domain of the computed column must be equivalent to the domain of the COMPUTE expression in order for the column substitution to be made. In the above example, if *year(order\_date)* had returned a string instead of an integer, the optimizer would not have substituted the computed column for the expression, and consequently the index *idx\_year* could not have been used to retrieve the required rows.

☞ For more information about computed columns, see "Using computed columns with Java classes" on page 124 of the book *ASA Programming Guide*.

Examples

In each of these examples, attributes *x* and *y* are each columns of a single table. Attribute *z* is contained in a separate table. Assume that an index exists for each of these attributes.

Sargable	Non-sargable
$x = 10$	$x \diamond 10$
$x \text{ IS NULL}$	$x \text{ IS NOT NULL}$
$x > 25$	$x = 4 \text{ OR } y = 5$
$x = z$	$x = y$
$x \text{ IN } (4, 5, 6)$	$x \text{ NOT IN } (4, 5, 6)$
$x \text{ LIKE 'pat\%'}$	$x \text{ LIKE '\%tern'}$
$x = 20 - 2$	$x + 2 = 20$

Sometimes it may not be obvious whether a predicate is sargable. In these cases, you may be able to rewrite the predicate so it is sargable. For each example, you could rewrite the predicate  $x$  LIKE 'pat%' using the fact that "u" is the next letter in the alphabet after "t":  $x \geq \text{'pat'}$  and  $x < \text{'pau'}$ . In this form, an index on attribute  $x$  is helpful in locating values in the restricted range. Fortunately, Adaptive Server Anywhere makes this particular transformation for you automatically.

A sargable predicate used for indexed retrieval on a table is a **matching** predicate. A WHERE clause can have a number of matching predicates. Which is most suitable can depend on the join strategy. The optimizer re-evaluates its choice of matching predicates when considering alternate join strategies.

## Types of semantic transformations

The optimizer can perform a number of transformations in search of more efficient and convenient representations of your query. Because the optimizer performs these transformations, the plan may look quite different from a literal interpretation of your original query. Common manipulations include:

- ◆ unnecessary DISTINCT elimination
- ◆ subquery unnesting
- ◆ predicate pushdown in UNION or GROUPed views
- ◆ join elimination
- ◆ optimization for minimum or maximum functions
- ◆ OR, in-list optimization
- ◆ LIKE optimizations
- ◆ conversion of outer joins to inner joins
- ◆ discovery of exploitable conditions through predicate inference
- ◆ elimination of unnecessary case translation

The following subsections discuss each of these operations.

### Unnecessary DISTINCT elimination

Sometimes a DISTINCT condition is unnecessary. For example, the properties of one or more column in your result may contain a UNIQUE condition, either explicitly, or implicitly because it is in fact a primary key.

**Examples**

The distinct keyword in the following command is unnecessary because the product table contains the primary key *p.id*, which is part of the result set.

```
SELECT DISTINCT p.id, p.quantity
FROM product p
```

p<seq>

The database server actually executes the semantically equivalent query:

```
SELECT p.id, p.quantity
FROM product p
```

Similarly, the result of the following query contains the primary keys of both tables so each row in the result must be distinct.

```
SELECT DISTINCT *
FROM sales_order o JOIN customer c
    ON o.cust_id = c.id
WHERE c.state = 'NY'
```

c<seq> JNL o<ix\_sales\_cust>

**Subquery unnesting**

You may express statements as nested queries, given the convenient syntax provided in the SQL language. However, rewriting nested queries as joins often leads to more efficient execution and more effective optimization, since Adaptive Server Anywhere can take better advantage of highly selective conditions in a subquery's WHERE clause.

**Examples**

The subquery in the following example can match at most one row for each row in the outer block. Because it can match at most one row, Adaptive Server Anywhere recognizes that it can convert it to an inner join.

```
SELECT s.*
FROM sales_order_items s
WHERE EXISTS
    ( SELECT *
      FROM product p
      WHERE s.prod_id = p.id
        AND p.id = 300 AND p.quantity > 300)
```

Following conversion, this same statement is expressed internally using join syntax:

```
SELECT s.*
FROM product p JOIN sales_order_items s
    ON p.id = s.prod_id
WHERE p.id = 300 AND p.quantity > 20
```

p<seq> JNL s<ky\_prod\_id>

Similarly, the following query contains a conjunctive EXISTS predicate in the subquery. This subquery can match more than one row.

```
SELECT p.*
FROM product p
WHERE EXISTS
  ( SELECT *
    FROM sales_order_items s
    WHERE s.prod_id = p.id
      AND s.id = 2001)
```

Adaptive Server Anywhere converts this query to an inner join, with a DISTINCT in the SELECT list.

```
SELECT DISTINCT p.*
FROM product p JOIN sales_order_items s
  ON p.id = s.prod_id
WHERE s.id = 2001
```

Distl[ s<id\_fk> JNL p<product> ]

Adaptive Server Anywhere can also eliminate subqueries in comparisons, when the subquery can match at most one row for each row in the outer block. Such is the case in the following query.

```
SELECT *
FROM product p
WHERE p.id =
  ( SELECT s.prod_id
    FROM sales_order_items s
    WHERE s.id = 2001
      AND s.line_id = 1 )
```

Adaptive Server Anywhere rewrites this query as follows.

```
SELECT p.*
FROM product p, sales_order_items s
WHERE p.id = s.prod_id
  AND s.id = 2001
  AND s.line_id = 1
```

p<seq> JNL s<sales\_order\_items>

The DUMMY table is treated as a special table when subquery unnesting rewrite optimizations are performed. Subquery flattening is always done on subqueries of the form `SELECT expression FROM DUMMY`, even if the subquery is not correlated.

## Predicate pushdown into GROUPed or UNION views

It is quite common for queries to restrict the result of a view so that only a few of the records are returned. In cases where the view contains GROUP BY or UNION, it would be preferable for the server to only compute the result for the desired rows.

### Example

Suppose we have the view `product_summary` defined as

```
CREATE VIEW product_summary( product_id, num_orders,
total_qty) as
SELECT prod_id, count(*), sum( quantity )
FROM sales_order_items
GROUP BY prod_id
```

which returns, for each product ordered, a count of the number of orders that include it, and the sum of the quantities ordered over all of the orders. Now consider the following query over this view:

```
SELECT *
FROM product_summary
WHERE product_id = 300
```

which restricts the output to that for product id 300. The query and the query from the view could be combined into one semantically equivalent SELECT statement, namely:

```
SELECT prod_id, count(*), sum( quantity )
FROM sales_order_items
GROUP BY prod_id
HAVING prod_id = 300.
```

A naive execution plan for this query would involve computing the aggregates for each product, and then restricting the result to only the single row for product ID 300. However, the HAVING predicate on the `product_id` column can be pushed into the query's WHERE clause since it is a grouping column, yielding

```
SELECT prod_id, count(*), sum( quantity )
FROM sales_order_items
WHERE prod_id = 300
GROUP BY prod_id
```

which significantly reduces the computation required. If this predicate is sufficiently selective, the optimizer could now use an index on `prod_id` to quickly retrieve only those rows for product 300, rather than sequentially scanning the `sales_order_items` table.

The same optimization is also used for views involving UNION or UNION ALL.

## Join elimination

The join elimination rewrite optimization reduces the join degree of the query by eliminating tables from the query when it is safe to do so. Typically, this optimization is used when the query contains a primary key-foreign key join, and only primary key columns from the primary table are referenced in the query.

### Example

For example, the query

```
SELECT s.id, s.line_id, p.id
FROM sales_order_items s KEY JOIN product p
```

would be rewritten as

```
SELECT s.id, s.line_id, s.prod_id
FROM sales_order_items s
WHERE s.prod_id IS NOT NULL.
```

The second query is semantically equivalent to the first because any row from the `sales_order_items` table that has a `NULL` foreign key to product will not appear in the result.

The join elimination optimization can also apply to tables involved in outer joins, although the conditions for which the optimization is valid are much more complex. Under certain other conditions, tables involved in primary key-primary key joins may also be candidates for elimination.

Users should be aware that when this optimization is used, the result of a `DESCRIBE` can differ from the expected result due to the substitution of columns. In addition, an `UPDATE` or `DELETE WHERE CURRENT` request may fail if the update statement refers to one or more of the eliminated base tables. To circumvent this problem, either ensure that additional columns from the eliminated table are present in the query's `SELECT` list (to avoid the optimization in the first place), or update the necessary row(s) using a separate statement.

## Optimization for minimum or maximum functions

The min/max rewrite optimization is designed to exploit an existing index to efficiently compute the result of a simple aggregation query involving the `MAX()` or `MIN()` aggregate functions. The goal of this optimization is to be able to compute the result with a single-row lookup using the index. To be a candidate for this optimization, the query:

- ◆ must not contain a `GROUP BY` clause
- ◆ must be over a single table
- ◆ cannot contain anything other than conjunctive equality conditions in the `WHERE` clause

- ◆ must contain only a single aggregate function (MAX or MIN) in the query's SELECT list

### Example

To illustrate this optimization, assume that an index *prod\_qty* on (*prod\_id* ASC, *quantity* ASC) exists on the *sales\_order\_items* table. Then the query

```
SELECT MIN( quantity )
FROM sales_order_items
Where prod_id = 300
```

is rewritten internally as

```
SELECT MIN( quantity )
FROM ( SELECT FIRST quantity
      FROM sales_order_items
      WHERE prod_id = 300 and quantity IS NOT NULL
      ORDER BY prod_id ASC, quantity ASC ) as
s(quantity)
```

The rewritten query is technically illegal, due to the presence of an ORDER BY clause in the derived table expression. The NULL\_VALUE\_ELIMINATED warning may not be generated for aggregate queries when this optimization is applied.

The access plan (short form) for the rewritten query is:

```
GrByS[ RL[ sales_order_items<prod_qty> ] ]
```

## IN-list optimization

The Adaptive Server Anywhere optimizer supports a special optimization for exploiting IN predicates on indexed columns. This optimization also applies equally to multiple predicates on the same indexed column that are ORed together, since the two are semantically equivalent. To enable the optimization, the IN-list must contain only constants.

When the optimizer encounters a qualifying IN-list predicate, and the IN-list predicate is sufficiently selective to consider indexed retrieval, the optimizer converts the IN-list predicate into a nested-loop join. The following example illustrates how the optimization works.

Suppose we have the query

```
SELECT *
FROM sales_order
WHERE sales_rep = 142 or sales_rep = 1596
```

that lists all of the orders for these two sales reps. This query is semantically equivalent to

```
SELECT *
FROM sales_order
WHERE sales_rep IN (142, 1596)
```

The optimizer estimates the combined selectivity of the IN-list predicate to be high enough to warrant indexed retrieval. Consequently the optimizer treats the IN-list as a virtual table, and joins this virtual table to the *sales\_order* table on the *sales\_rep* attribute. While the net effect of the optimization is to include an additional "join" in the access plan, the join degree of the query is not increased, so optimization time should not be affected.

There are two main advantages of this optimization. First, the IN-list predicate can be treated as a sargable predicate and exploited for indexed retrieval. Second, the optimizer can sort the IN-list to match the sort sequence of the index, leading to more efficient retrieval.

The short form of the access plan for the above query is

```
IN JNL sales_order<ky_so_employee_id>
```

## LIKE optimizations

LIKE predicates involving patterns that are either literal constants or host variables are very common. Depending on the pattern, the optimizer may rewrite the LIKE predicate entirely, or augment it with additional conditions that could be exploited to perform indexed retrieval on the corresponding table.

### Examples

In each of the following examples, assume that the pattern in the LIKE predicate is a literal constant or host variable, and X is a column in a base table.

- ◆ X LIKE '%' is rewritten as X IS NOT NULL
- ◆ X LIKE 'abc' is rewritten as X = 'abc'
- ◆ X LIKE 'abc%' is augmented with the predicates X < 'abcZ' and X >= 'abc\_'

where Z and \_ represent the corresponding high values and low values for the collating sequence of this database. If the database is configured to store blank-padded strings, the second comparison operator is >, not >=, to ensure correct semantics.



## Conversion of outer joins to inner joins

For the most part, the optimizer generates a left-deep processing tree for its access plans. The only exception to this rule is the existence of a right-deep nested outer join expression. The query execution engine's algorithms for computing LEFT or RIGHT OUTER JOINS require that preserved tables must precede null-supplying tables in any join strategy. Consequently the optimizer looks for opportunities to convert LEFT or RIGHT outer joins to inner joins whenever possible, since inner joins are commutable and give the optimizer greater degrees of freedom when performing join enumeration.

A LEFT or RIGHT OUTER JOIN can be converted to an inner join when a null-intolerant predicate on the null-supplying table is present in the query's WHERE clause. Since this predicate is null-intolerant, any all-NULL row that would be produced by the outer join will be eliminated from the result, hence making the query semantically equivalent to an inner join.

### Example

For example, consider the query

```
SELECT *
FROM product p KEY LEFT OUTER JOIN sales_order_items s
WHERE s.quantity > 15
```

which is intended to list all products and their orders for larger quantities; the LEFT OUTER JOIN is intended to ensure that all products are listed, even if they have no orders. The problem with this query is that the predicate in the WHERE clause will eliminate from the result any product with no orders, because the predicate `s.quantity > 15` will be interpreted as FALSE if `s.quantity` is NULL. Hence the query is semantically equivalent to

```
SELECT *
FROM product p KEY JOIN sales_order_items s
WHERE s.quantity > 15
```

and it is this rewritten form of the query that the server will optimize.

In this example, the query is almost certainly written incorrectly; it should probably read

```
SELECT *
FROM product p
KEY LEFT OUTER JOIN sales_order_items s
ON s.quantity > 15
```

so that the test of quantity is part of the outer join condition.

While it is rare for this optimization to apply to straightforward outer join queries, it can often apply when a query refers to one or more views that are written using outer joins. The query's WHERE clause may include conditions that restrict the output of the view such that all null-supplying rows from one or more table expressions would be eliminated, hence making this optimization applicable.

## Discovery of exploitable conditions

An efficient access strategy for virtually any query relies on the presence of sargable conditions in the WHERE/ON/HAVING clauses. Indexed retrieval is possible only by exploiting sargable conditions as matching predicates. In addition, hash, merge, and block-nested loop joins can only be used when an equijoin condition is present. For these reasons, Adaptive Server Anywhere does detailed analysis of the search conditions in the original query text in order to discover simplified or implied conditions that can be exploited by the optimizer.

As a preprocessing step, several simplifications are made to predicates in the original statement once view expansion and merging have taken place. For example:

- ◆  $X = X$  is rewritten as  $X \text{ IS NOT NULL}$  if  $X$  is nullable; otherwise the predicate is eliminated.
- ◆  $\text{ISNULL}(X, X)$  is rewritten as simply  $X$ .
- ◆  $X+0$  is rewritten as  $X$  if  $X$  is a numeric column.
- ◆  $\text{AND } 1=1$  is eliminated.
- ◆  $\text{OR } 1=0$  is eliminated.
- ◆ IN-list predicates that consist of a single element are converted to simple equality conditions.

After this preprocessing step, Adaptive Server Anywhere attempts to normalize the original search condition into conjunctive normal form (CNF). For an expression to be in CNF, each term in the expression must be ANDed together. Each term is either made up of a single atomic condition, or a set of conditions ORed together.

Converting an arbitrary condition into CNF may yield an expression of similar complexity but with a much larger set of conditions. Adaptive Server Anywhere recognizes this situation, and refrains from naively converting the condition into CNF. Instead, Adaptive Server Anywhere analyzes the original expression for exploitable predicates that are implied by the original search condition, and ANDs these inferred conditions to the query. Complete normalization is also avoided if this would require duplication of an expensive predicate (for example, a quantified subquery predicate). However, the algorithm will merge IN-list predicates together whenever feasible.

Once the search condition has either been completely normalized or the exploitable conditions have been found, the optimizer performs transitivity analysis to discover transitive equality conditions, primarily transitive join conditions and conditions with a constant. In doing so the optimizer will increase its degrees of freedom when performing join enumeration during its cost-based optimization phase, since these transitive conditions may permit additional alternative join orders.

### Example

Suppose the original query is

```
SELECT e.emp_lname, s.id, s.order_date
FROM sales_order s, employee e
WHERE (e.emp_id = s.sales_rep and
      (s.sales_rep = 142 or s.sales_rep = 1596)
)
OR
( e.emp_id = s.sales_rep and s.cust_id = 667)
```

This query has no conjunctive equijoin condition; hence without detailed predicate analysis the optimizer would fail to discover an efficient access plan. Fortunately, Adaptive Server Anywhere is able to convert the entire expression to CNF, yielding the equivalent query

```
SELECT e.emp_lname, s.id, s.order_date
FROM sales_order as s, employee as e
WHERE e.emp_id = s.sales_rep AND
      (s.sales_rep = 142 or s.sales_rep = 1596 or s.cust_id
= 667)'
```

which can now be efficiently optimized as an inner join query.

## Elimination of unnecessary case translation

By default, Adaptive Server Anywhere databases support case-insensitive string comparisons. Occasionally the optimizer may encounter queries where the user is explicitly forcing text conversion through the use of the UPPER() or LOWER() built-in functions when such conversion is unnecessary. Adaptive Server Anywhere will automatically eliminate this unnecessary conversion when the database's collating sequence permits it. Eliminating the case translation may then permit the comparison predicate to be used for indexed retrieval of the corresponding table.

### Example

On a case insensitive database, the query

```
SELECT *  
FROM customer  
WHERE UPPER(lname) = 'SMITH'
```

is rewritten internally as

```
SELECT *  
FROM customer  
WHERE lname = 'SMITH'
```

and the optimizer can now consider using an index on *customer.lname*.

## Subquery and function caching


When Adaptive Server Anywhere processes a subquery, it caches the result. This caching is done on a request-by-request basis; cached results are never shared by concurrent requests or connections. Should Adaptive Server Anywhere need to re-evaluate the subquery for the same set of correlation values, it can simply retrieve the result from the cache. In this way, Adaptive Server Anywhere avoids many repetitious and redundant computations. When the request is completed (the query's cursor is closed), Adaptive Server Anywhere releases the cached values.

As the processing of a query progresses, Adaptive Server Anywhere monitors the frequency with which cached subquery values are reused. If the values of the correlated variable rarely repeat, then Adaptive Server Anywhere needs to compute most values only once. In this situation, Adaptive Server Anywhere recognizes that it is more efficient to recompute occasional duplicate values, than to cache numerous entries that occur only once. Hence the server suspends the caching of this subquery for the remainder of the statement and proceeds to re-evaluate the subquery for each and every row in the outer query block.

Adaptive Server Anywhere also does not cache if the size of the dependent column is more than 255 bytes. In such cases, you may wish to rewrite your query or add another column to your table to make such operations more efficient.

### User-defined function caching

User-defined functions are cached in the same way that subquery results are cached. This can result in a substantial improvement for expensive functions that are called during query processing with the same parameters. However, it may mean that a function is called less times than would otherwise be expected. The optimizer assumes that functions used in query processing always return the same result for a given set of parameters, and that these functions do not have side effects.

 For more information about user-defined functions, see "CREATE FUNCTION statement" on page 296 of the book *ASA SQL Reference Manual*.

# Reading access plans

The optimizer can tell you the query optimization strategy (plan) it has chosen in response to any statement.

The optimizer’s job is to understand the semantics of your query and to construct a plan that computes its result. This plan may not correspond exactly to the syntax you used. The optimizer is free to rewrite your query in any semantically equivalent form.

☞ For more information about the rules Adaptive Server Anywhere obeys when rewriting your query, see "Rewriting subqueries as EXISTS predicates" on page 320 and "Semantic query transformations" on page 349.

☞ For information about the methods that the optimizer uses to implement your query, see "Query execution algorithms" on page 324.

You can view the plan in Interactive SQL or using SQL functions. You can choose to retrieve the access plan in several different formats:

- ◆ Short text
- ◆ Long text
- ◆ Graphical
- ◆ Graphical with statistics
- ◆ UltraLite (short, long, or graphical)

As well, you can obtain plans for SQL queries with a particular cursor type.

☞ For more information about how to access the plan, see "Accessing the plan" on page 378. For information about how to read plans, see "Text plans" on page 370 and "Graphical plans" on page 373.

Following is an explanation of the statistics and other items that are displayed in access plans.

Abbreviations used in the plan

Following are the abbreviations that are used in short plan, and in the short name form of the graphical plans:

Name	Short Plan / Short name
Hash group by	GrByH
Ordered group by	GrByO
Single row group by	GrByS
Indexed group by	GrByI
Hash distinct	DistH

Name	Short Plan / Short name
Indexed distinct	DistI
Ordered distinct	DistO
Sorted block	SrtBl
Nested loops semijoin	JNLS
Hash exists	JHE
Hash not exists	JHNE
Hash join	JH
Hash join	JHO
Nested block join	JNB
Left outer nested block join	JNBO
Nested loops join	JNL
Nested loops antisemijoin	JNLA
Left outer nested loops join	JNLO
Full outer nested loops join	JNLFO
Merge join	JM
Left outer merge join	JMO
Full outer merge join	JMFO
Row limit	RL
Union all	UA
Table scan	In short plan is <i>tablename</i> <seq>. In graphical plans is just the table name.
Index scan	In short plan is <i>tablename</i> <indexname>. In graphical plans is just the table name.
In list	IN

For an explanation of the algorithms, see "Query execution algorithms" on page 324.

Common statistics  
used in the plan

The following statistics are actual, measured amounts.

<b>Statistic</b>	<b>Explanation</b>
Invocations	Number of times a row was requested from the sub tree.
RowsReturned	Number of rows returned for the current node.
RunTime	Time required for execution of the sub-tree, including time for children.
CacheHits	Number of successful reads of the cache.
CacheRead	Number of database pages that have been looked up in the cache.
CacheReadTable	Number of table pages that have been read from the cache.
CacheReadIndLeaf	Number of index leaf pages that have been read from the cache.
CacheReadIndInt	Number of index internal node pages that have been read from the cache.
DiskRead	Number of pages that have been read from disk.
DiskReadTable	Number of table pages that have been read from disk.
DiskReadIndLeaf	Number of index leaf pages that have been read from disk.
DiskReadIndInt	Number of index internal node pages that have been read from disk.
DiskWrite	Number of pages that have been written to disk (work table pages or modified table pages).
IndAdd	Number of entries that have been added to indexes.
IndLookup	Number of entries that have been looked up in indexes.
FullCompare	Number of comparisons that have been performed beyond the hash value in an index.



Common estimates  
used in the plan

Statistic	Explanation
EstRowCount	Estimated number of rows that the node will return each time it is invoked.
AvgRowCount	Average number of rows returned on each invocation. This is not an estimate, but is calculated as $\text{RowsReturned} / \text{Invocations}$ . If this value is significantly different from EstRowCount, the selectivity estimates may be poor.
EstRunTime	Estimated time required for execution (sum of EstDiskReadTime, EstDiskWriteTime, and EstCpuTime).
AvgRunTime	Average time required for execution (measured).
EstDiskReads	Estimated number of read operations from the disk.
AvgDiskReads	Average number of read operations from the disk (measured).
EstDiskWrites	Estimated number of write operations to the disk.
AvgDiskWrites	Average number of write operations to the disk (measured).
EstDiskReadTime	Estimated time required for reading rows from the disk.
EstDiskWriteTime	Estimated time required for writing rows to the disk.
EstCpuTime	Estimated processor time required for execution.

Items in the plan related to SELECT, INSERT, UPDATE, and DELETE

Item	Explanation
Optimization Goal	Determines whether query processing is optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set.
ANSI update constraints	Controls the range of updates that are permitted (options are OFF, CURSORS, and STRICT).
Optimization level	Reserved for future use.
Select list	List of expressions selected by the query.


Items in the plan related to locks

Item	Explanation
Locked tables	List of all locked tables and their isolation levels.


Items in the plan related to scans

Item	Explanation
Table name	Actual name of the table.
Correlation name	Alias for the table.
Estimated rows	Estimated number of rows in the table.
Estimated pages	Estimated number of pages in the table.
Estimated row size	Estimated row size for the table.
Page maps	YES when a page map is used to read multiple pages.

Items in the plan related to index scans

Item	Explanation
Index name	Name of the index.
Key type	Can be one of PRIMARY KEY, FOREIGN KEY, CONSTRAINT (unique constraint), or UNIQUE (unique index). The key type is not displayed if the index is a non-unique secondary index.
Depth	Height of the index.  For more information, see "Table and page sizes" on page 338.
Estimated leaf pages	Estimated number of leaf pages.
Cardinality	The cardinality of the index if it is different from the estimated number of rows. This applies only to Adaptive Server Anywhere databases version 6.0 and earlier.
Selectivity	The estimated number of rows that match the range bounds.
Direction	FORWARD or BACKWARD.
Range bounds	Range bounds are shown as a list (col_name=value) or col_name IN [low, high].

Items in the plan related to joins, filter, and pre-filter

Item	Explanation
Predicate	The search condition that is evaluated in this node, along with selectivity estimates and measurement.  For more information, see "Selectivity in the plan" on page 377

Items in the plan related to GROUP BY

Item	Explanation
Aggregates	All the aggregate functions.
Group-by list	All the columns in the group by clause.

Items in the plan related to DISTINCT	Item	Explanation
	Distinct list	All the columns in the distinct clause.
Items in the plan related to IN LIST	Item	Explanation
	In List	All the expressions in the specified set.
	Expression SQL	Expressions to compare to the list.
Items in the plan related to SORT	Item	Explanation
	Order-by	List of all expressions to sort by.
Items in the plan related to row limits	Item	Explanation
	Row limit count	Maximum number of rows returned as specified by FIRST or TOP n.

Text plans

There are two types of text plan: short and long. To choose a plan type in Interactive SQL, open the Options dialog from the Tools menu, and click the Plan tab. To use SQL functions to access the plan, see "Accessing the Plan with SQL functions" on page 378.

Colons separate join strategies

The following command contains two **query blocks**: the outer select statement from the *sales\_order* and *sales\_order\_items* tables, and the subquery that selects from the product table.

```
SELECT *
FROM sales_order AS o
  KEY JOIN sales_order_items AS i
WHERE EXISTS
  ( SELECT *
    FROM product p
    WHERE p.id = 300 )
```

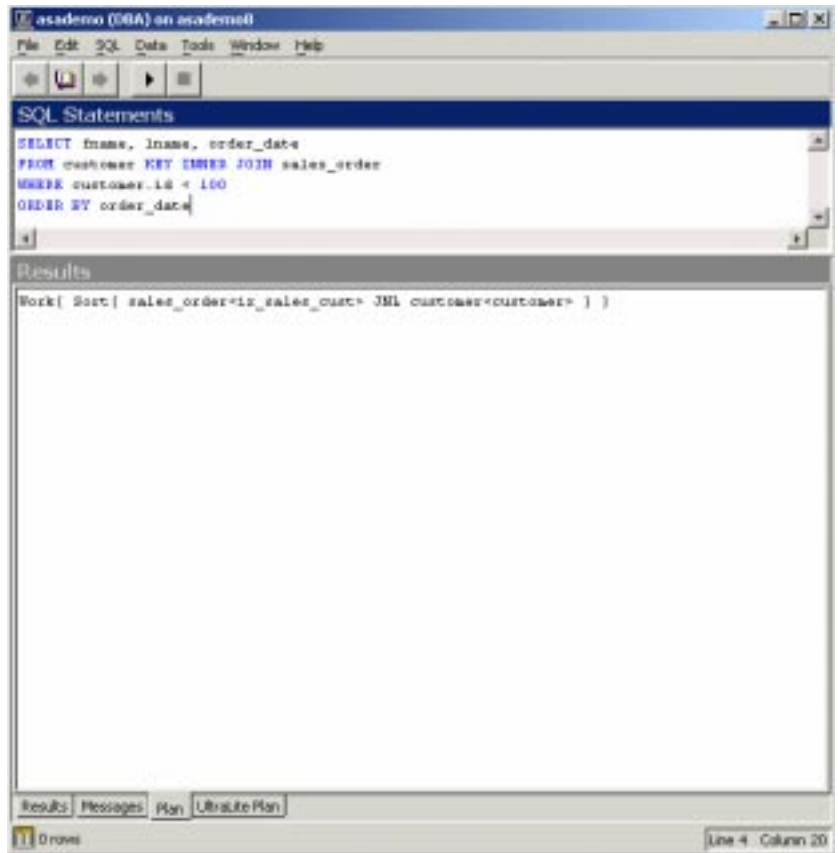
```
i<seq> JNL o<sales_order> : p<seq>
```

Colons separate join strategies. Plans always list the join strategy for the main block first. Join strategies for other query blocks follow. The order of join strategies for these other query blocks may not correspond to the order in your statement nor to the order in which they execute.

## Short text plan

The short plan is useful when you want to compare plans quickly. It provides the least amount of information of all the access plan formats, but it provides it on a single line.

In the following example, the plan starts with the word SORT because the ORDER BY clause causes the entire result set to be sorted. The *customer* table is accessed by its primary key index, also called *customer*. An index scan is used to satisfy the search condition because the column *customer.id* is a primary key. The abbreviation JNL indicates that the optimizer is using a nested loops join to process the query. Finally, the *sales\_order* table is accessed using the foreign key index *ky\_so\_customer* to find matching rows in the *customer* table.



For more information about code words used in the plan, see "Abbreviations used in the plan" on page 364.



## Graphical plans

There are two types of graphical plan: the graphical plan, and the graphical plan with statistics. To choose a plan type in Interactive SQL, open the Options dialog from the Tools menu, and click the Plan tab. To access the plan with SQL functions, see "Accessing the Plan with SQL functions" on page 378.

Once the graphical plan is displayed, you can configure the way it is displayed by right-clicking the left pane and choosing Customize.

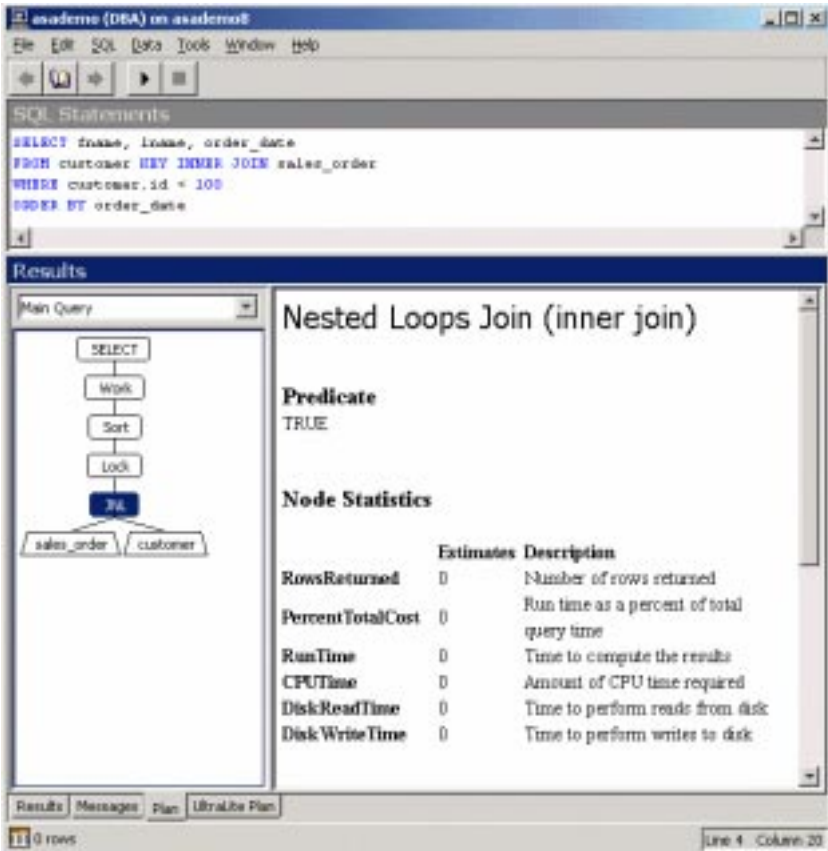
To obtain context-sensitive help for each node in the graphical plan, select the node, right-click it and choose Help. For example, right-click Nested Loops Join and choose Help for information about the resources used by that part of the execution. There is also pop-up information that is available by hovering your cursor over each element in the graphical plan.

### Graphical plan

The graphical plan provides a great deal more information than the short or long plans. You can choose to see either the graphical plan, or the graphical plan with statistics. Both allow you to quickly view which parts of the plan have been estimated as the most expensive. The graphical plan with statistics, though more expensive to view, also provides the actual query execution statistics as monitored by the server when the query is executed, and permits direct comparison between the estimates used by the query optimizer in constructing the access plan with the actual statistics monitored during execution. Note, however, that the optimizer is often unable to precisely estimate a query's cost, so expect there to be differences. The graphical plan is the default format for access plans.

Following is the same query that was used to describe the short and long text plans, presented with the graphical plan. The diagram is in the form of a tree, indicating that each node requests rows from the nodes beneath it. The Lock node indicates that the result set is materialized, or that a row is returned to an application. In this case, the sort requires that results are materialized. At level 0, rows aren't really locked: Adaptive Server Anywhere just ensures that the row has not been deleted since it was read from the base tables. At level 1, a row is locked only until the next row is accessed. At levels 2 and 3, read locks are applied and held until COMMIT.

You can obtain detailed information about the nodes in the plan by clicking the node in the graphical diagram. In this example, the nested loops join node is selected. The information in the right pane pertains only to that node.



For more information about code words used in the plan, see "Abbreviations used in the plan" on page 364.

Graphical plan with statistics

The graphical plan with statistics shows you all the estimates that are provided with the graphical plan, but also shows actual runtime costs of executing the statement. To do this, the statement must actually be executed. This means that there may be a delay in accessing the plan for expensive queries. It also means that any parts of your query such as deletes or updates are actually executed, although you can perform a rollback to undo these changes.



Use the graphical plan with statistics when you are having performance problems, and the estimated row count or run time differs from your expectations. The graphical plan with statistics provides estimates and actual statistics for you to compare. A large difference between actual and estimate is a warning sign that the optimizer might not have sufficient information to prepare correct estimates.

Following are some of the key statistics you can check in the graphical plan with statistics, and some possible remedies:

- ◆ **Row count actuals and estimates** Row count measures the rows in the result set. If the estimated row count is significantly different from the actual row count, the selectivity is probably incorrect.
- ◆ **Selectivity actuals and estimates** Accurate selectivity estimates are critical for the proper operation of the query optimizer. For example, if the optimizer mistakenly estimates a predicate to be highly selective (with, say, a selectivity of 5%), but the actual selectivity is much lower (for example, 50%), then performance may suffer. In general, estimates will not be precise. However, a significantly large error does indicate a possible problem. If the predicate is over a base column for which there does not exist a histogram, executing a `CREATE STATISTICS` statement to create a histogram may correct the problem. If selectivity error remains a problem then, as a last resort, you may wish to consider specifying a user estimate of selectivity along with the predicate in the query text.

🔗 For more information about selectivity, see "Selectivity in the plan" on page 377.

🔗 For more information about creating statistics, see "CREATE STATISTICS statement" on page 323 of the book *ASA SQL Reference Manual*.

🔗 For more information about user estimates, see "Explicit selectivity estimates" on page 31 of the book *ASA SQL Reference Manual*.

- ◆ **Runtime actuals and estimates** Runtime measures the time to execute the query. If the runtime is incorrect for a table scan or index scan, you may improve performance by executing the `REORGANIZE TABLE` statement.

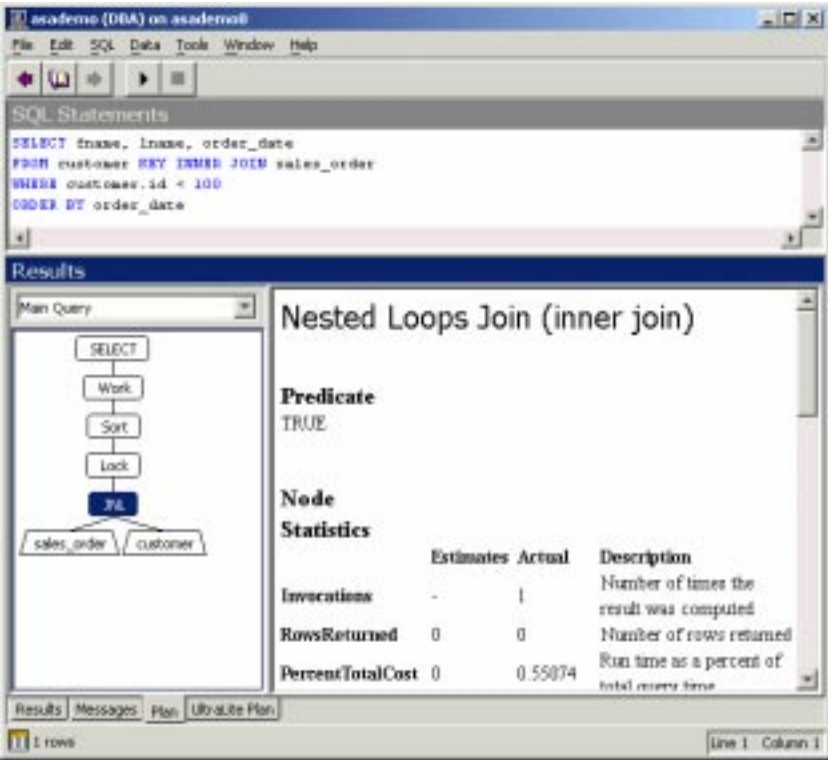
🔗 For more information, see "REORGANIZE TABLE statement" on page 508 of the book *ASA SQL Reference Manual*.

- ◆ **Estimate source** When the source of the estimate is Guess, the optimizer has no information to use, which may indicate a problem. If the estimate source is Index and the selectivity estimate is incorrect, your problem may be that the index is skewed: you may benefit from defragmenting the index with the `REORGANIZE TABLE` statement.

☞ For a complete list of the possible sources of selectivity estimates, see "ESTIMATE\_SOURCE function" on page 131 of the book *ASA SQL Reference Manual*.

- ◆ **Cache reads and hits** If the number of cache reads and cache hits are exactly the same, then your entire database is in cache—an excellent thing. When reads are greater than hits, it means that the server is attempting to go to cache but failing, and that it must read from disk. In some cases, such as hash joins, this is expected. In other cases, such as nested loops joins, a poor cache-hit ratio may indicate a performance problem, and you may benefit from increasing your cache size.

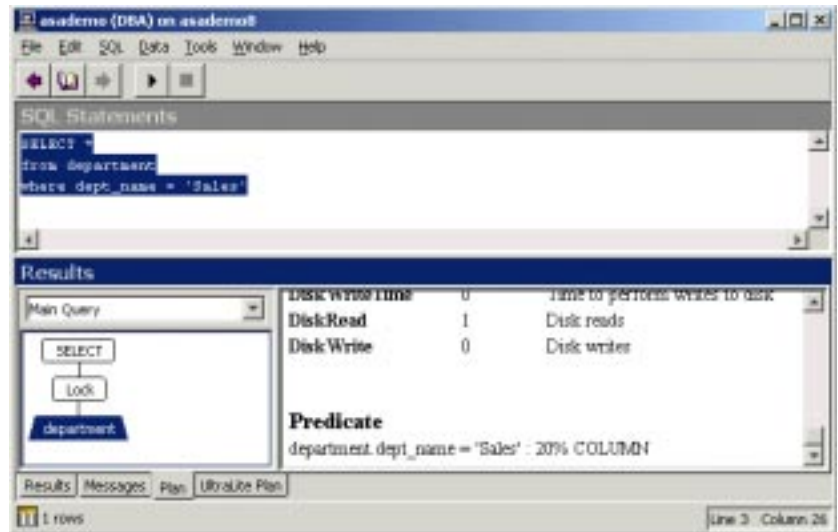
Following is an example of the graphical plan with statistics. Again, the nested loops join node is selected. The statistics in the right pane indicate the resources used by that part of the query.



☞ For more information about code words used in the plan, see "Abbreviations used in the plan" on page 364.

## Selectivity in the plan

Following is an example of the Predicate showing selectivity of a search condition. In this example, the Filter node is selected, and the statistics pane shows the Predicate as the search condition and selectivity statistics.



This predicate is

```
department.dept_name = 'Sales' : 20% COLUMN
```

This can be read as follows:

- ◆ `department.dept_name = 'Sales'` is the search condition.
- ◆ 20% is the optimizer's estimate of the selectivity. This is the same output as is provided by the `ESTIMATE` function. For more information, see "ESTIMATE function" on page 130 of the book *ASA SQL Reference Manual*.
- ◆ The source of the estimate is `COLUMN`. This is the same output as is provided by the `ESTIMATE_SOURCE` function. For a complete list of the possible sources of selectivity estimates, see "ESTIMATE\_SOURCE function" on page 131 of the book *ASA SQL Reference Manual*.

*Note:*

If you select the graphical plan, but not the graphical plan with statistics, the final two statistics are not displayed.

## Accessing the plan

You can see the plan in Interactive SQL, or using SQL functions.

### Accessing the plan in Interactive SQL

There are four types of plan available in Interactive SQL:

- ◆ **Short plan**
- ◆ **Long plan**
- ◆ **Graphical plan**
- ◆ **Graphical plan with statistics**

To choose the type of plan you want to see, click Tools►Options and select the Plan tab.

To see the plan, execute a query and then open the Plan tab. The Plan tab is located at the bottom of the Interactive SQL window.

#### UltraLite plan

You can also view the UltraLite plan for your query. The UltraLite plan does not include statistics.

To view the UltraLite plan in Interactive SQL, open the Options dialog from the Tools menu and select Show UltraLite Plan. This option is selected by default. You control the UltraLite plan type by selecting one of the types above (graphical, short plan, or long plan). If you choose graphical plan with statistics, you get the graphical plan.

The UltraLite Plan tab appears at the bottom of the Interactive SQL window. For some queries, the UltraLite execution plan may differ from the plan selected for Adaptive Server Anywhere.

For more information, see "GRAPHICAL\_ULPLAN function" on page 140 of the book *ASA SQL Reference Manual*.

### Accessing the Plan with SQL functions

You can access the plan using SQL functions, and retrieve the output in XML format.

- ◆ To access the short plan, see the "EXPLANATION function" on page 136 of the book *ASA SQL Reference Manual*.
- ◆ To access the long plan, see the "PLAN function" on page 165 of the book *ASA SQL Reference Manual*.
- ◆ To access the graphical plan, see the "GRAPHICAL\_PLAN function" on page 138 of the book *ASA SQL Reference Manual*.
- ◆ To access the UltraLite plan, see the "GRAPHICAL\_ULPLAN function" on page 140 of the book *ASA SQL Reference Manual*, "SHORT\_ULPLAN function" on page 177 of the book *ASA SQL Reference Manual*, or "LONG\_ULPLAN function" on page 152 of the book *ASA SQL Reference Manual*.



## PART THREE

# SQL Dialects and Compatibility

This part describes Transact SQL compatibility and those features of Adaptive Server Anywhere that are not commonly found in other SQL implementations.

---



# Transact-SQL Compatibility

About this chapter      Transact-SQL is the dialect of SQL supported by Sybase Adaptive Server Enterprise.

                                 This chapter is a guide for creating applications that are compatible with both Adaptive Server Anywhere and Adaptive Server Enterprise. It describes Adaptive Server Anywhere support for Transact-SQL language elements and statements, and for Adaptive Server Enterprise system tables, views, and procedures.

Contents

Topic	Page
An overview of Transact-SQL support	384
Adaptive Server architectures	387
Configuring databases for Transact-SQL compatibility	393
Writing compatible SQL statements	401
Transact-SQL procedure language overview	406
Automatic translation of stored procedures	409
Returning result sets from Transact-SQL procedures	410
Variables in Transact-SQL procedures	411
Error handling in Transact-SQL procedures	412

## An overview of Transact-SQL support

Adaptive Server Anywhere supports a large subset of **Transact-SQL**, which is the dialect of SQL supported by Sybase Adaptive Server Enterprise. This chapter describes compatibility of SQL between Adaptive Server Anywhere and Adaptive Server Enterprise.

### Goals

The goals of Transact-SQL support in Adaptive Server Anywhere are as follows:

- ◆ **Application portability** Many applications, stored procedures, and batch files can be written for use with both Adaptive Server Enterprise and Adaptive Server Anywhere databases.
- ◆ **Data portability** Adaptive Server Anywhere and Adaptive Server Enterprise databases can exchange and replicate data between each other with minimum effort.

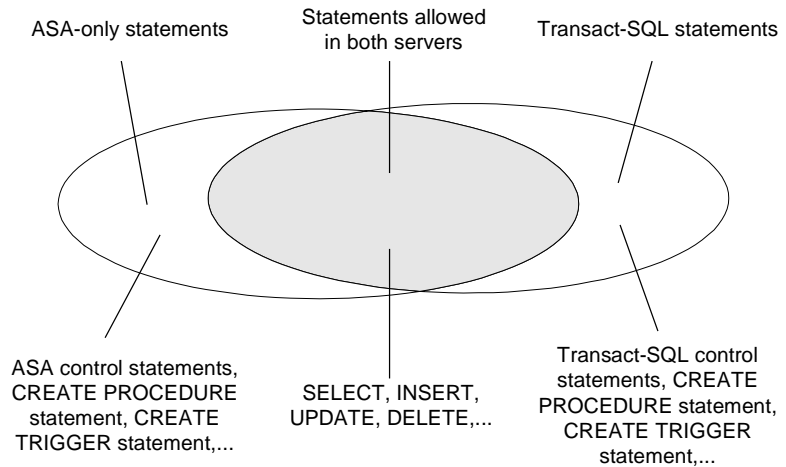
The aim is to write applications to work with both Adaptive Server Enterprise and Adaptive Server Anywhere. Existing Adaptive Server Enterprise applications generally require some changes to run on an Adaptive Server Anywhere database.

### How Transact-SQL is supported

Transact-SQL support in Adaptive Server Anywhere takes the following form:

- ◆ Many SQL statements are compatible between Adaptive Server Anywhere and Adaptive Server Enterprise.
- ◆ For some statements, particularly in the procedure language used in procedures, triggers, and batches, a separate Transact-SQL statement is supported together with the syntax supported in previous versions of Adaptive Server Anywhere. For these statements, Adaptive Server Anywhere supports two **dialects** of SQL. In this chapter, we name those dialects Transact-SQL and Watcom-SQL.
- ◆ A procedure, trigger, or batch is executed in either the Transact-SQL or Watcom-SQL dialect. You must use control statements from one dialect only throughout the batch or procedure. For example, each dialect has different flow control statements.

The following diagram illustrates how the two dialects overlap.



### Similarities and differences

Adaptive Server Anywhere supports a very high percentage of Transact-SQL language elements, functions, and statements for working with existing data. For example, Adaptive Server Anywhere supports all of the numeric functions, all but one of the string functions, all aggregate functions, and all date and time functions. As another example, Adaptive Server Anywhere supports Transact-SQL outer joins (using `=*` and `*=` operators) and extended DELETE and UPDATE statements using joins.

Further, Adaptive Server Anywhere supports a very high percentage of the Transact-SQL stored procedure language (CREATE PROCEDURE and CREATE TRIGGER syntax, control statements, and so on) and many, but not all, aspects of Transact-SQL data definition language statements.

There are design differences in the architectural and configuration facilities supported by each product. Device management, user management, and maintenance tasks such as backups tend to be system-specific. Even here, Adaptive Server Anywhere provides Transact-SQL system tables as views, where the tables that are not meaningful in Adaptive Server Anywhere have no rows. Also, Adaptive Server Anywhere provides a set of system procedures for some of the more common administrative tasks.

This chapter looks first at some system-level issues where differences are most noticeable, before discussing data manipulation and data definition language aspects of the dialects where compatibility is high.

### Transact-SQL only

Some SQL statements supported by Adaptive Server Anywhere are part of one dialect, but not the other. You cannot mix the two dialects within a procedure, trigger, or batch. For example, Adaptive Server Anywhere supports the following statements, but as part of the Transact-SQL dialect only:

- ◆ Transact-SQL control statements IF and WHILE

- ◆ Transact-SQL EXECUTE statement
- ◆ Transact-SQL CREATE PROCEDURE and CREATE TRIGGER statements
- ◆ Transact-SQL BEGIN TRANSACTION statement
- ◆ SQL statements *not* separated by semicolons are part of a Transact-SQL procedure or batch

**Adaptive Server  
Anywhere only**

Adaptive Server Enterprise does not support the following statements:

- ◆ control statements CASE, LOOP, and FOR
- ◆ Adaptive Server Anywhere versions of IF and WHILE
- ◆ CALL statement
- ◆ Adaptive Server Anywhere versions of the CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER statements
- ◆ SQL statements separated by semicolons

**Notes**

The two dialects cannot be mixed within a procedure, trigger, or batch. This means that:

- ◆ You can include Transact-SQL-only statements together with statements that are part of both dialects in a batch, procedure, or trigger.
- ◆ You can include statements not supported by Adaptive Server Enterprise together with statements that are supported by both servers in a batch, procedure, or trigger.
- ◆ You cannot include Transact-SQL-only statements together with Adaptive Server Anywhere-only statements in a batch, procedure, or trigger.

## Adaptive Server architectures

Adaptive Server Enterprise and Adaptive Server Anywhere are complementary products, with architectures designed to suit their distinct purposes. Adaptive Server Anywhere works well as a workgroup or departmental server requiring little administration, and as a personal database. Adaptive Server Enterprise works well as an enterprise-level server for the largest databases.

This section describes architectural differences between Adaptive Server Enterprise and Adaptive Server Anywhere. It also describes the Adaptive Server Enterprise-like tools that Adaptive Server Anywhere includes for compatible database management.

### Servers and databases

The relationship between servers and databases is different in Adaptive Server Enterprise and Adaptive Server Anywhere.

In Adaptive Server Enterprise, each database exists inside a server, and each server can contain several databases. Users can have login rights to the server, and can connect to the server. They can then use each database on that server for which they have permissions. System-wide system tables, held in a master database, contain information common to all databases on the server.

No master  
database in  
Adaptive Server  
Anywhere

In Adaptive Server Anywhere, there is no level corresponding to the Adaptive Server Enterprise master database. Instead, each database is an independent entity, containing all of its system tables. Users can have connection rights to a database, not to the server. When a user connects, they connect to an individual database. There is no system-wide set of system tables maintained at a master database level. Each Adaptive Server Anywhere database server can dynamically load and unload multiple databases, and users can maintain independent connections on each.

Adaptive Server Anywhere provides tools in its Transact-SQL support and in its Open Server support to allow some tasks to be carried out in a manner similar to Adaptive Server Enterprise. For example, Adaptive Server Anywhere provides an implementation of the Adaptive Server Enterprise **sp\_addlogin** system procedure that carries out the nearest equivalent action: adding a user to a database.

☞ For information about Open Server support, see "Adaptive Server Anywhere as an Open Server" on page 105 of the book *ASA Database Administration Guide*.

#### File manipulation statements

Adaptive Server Anywhere does not support the Transact-SQL statements DUMP DATABASE and LOAD DATABASE. Adaptive Server Anywhere has its own CREATE DATABASE and DROP DATABASE statements with different syntax.

## Device management

Adaptive Server Anywhere and Adaptive Server Enterprise use different models for managing devices and disk space, reflecting the different uses for the two products. While Adaptive Server Enterprise sets out a comprehensive resource management scheme using a variety of Transact-SQL statements, Adaptive Server Anywhere manages its own resources automatically, and its databases are regular operating system files.

Adaptive Server Anywhere does not support Transact-SQL DISK statements, such as DISK INIT, DISK MIRROR, DISK REFIT, DISK REINIT, DISK REMIRROR, and DISK UNMIRROR.

🔗 For information on disk management, see "Working with Database Files" on page 217 of the book *ASA Database Administration Guide*

## Defaults and rules

Adaptive Server Anywhere does not support the Transact-SQL CREATE DEFAULT statement or CREATE RULE statement. The CREATE DOMAIN statement allows you to incorporate a default and a rule (called a CHECK condition) into the definition of a domain, and so provides similar functionality to the Transact-SQL CREATE DEFAULT and CREATE RULE statements.

In Adaptive Server Enterprise, the CREATE DEFAULT statement creates a named **default**. This default can be used as a default value for columns by binding the default to a particular column or as a default value for all columns of a domain by binding the default to the data type using the **sp\_bindefault** system procedure.

The CREATE RULE statement creates a named **rule** which can be used to define the domain for columns by binding the rule to a particular column or as a rule for all columns of a domain by binding the rule to the data type. A rule is bound to a data type or column using the **sp\_bindrule** system procedure.

In Adaptive Server Anywhere, a domain can have a default value and a CHECK condition associated with it, which are applied to all columns defined on that data type. You create the domain using the CREATE DOMAIN statement.

You can define default values and rules, or CHECK conditions, for individual columns using the CREATE TABLE statement or the ALTER TABLE statement.

☞ For a description of the Adaptive Server Anywhere syntax for these statements, see "SQL Statements" on page 199 of the book *ASA SQL Reference Manual*.

## System tables

In addition to its own system tables, Adaptive Server Anywhere provides a set of system views that mimic relevant parts of the Adaptive Server Enterprise system tables. You'll find a list and individual descriptions in "Views for Transact-SQL Compatibility" on page 679 of the book *ASA SQL Reference Manual*, which describes the system catalogs of the two products. This section provides a brief overview of the differences.

The Adaptive Server Anywhere system tables rest entirely within each database, while the Adaptive Server Enterprise system tables rest partly inside each database and partly in the master database. The Adaptive Server Anywhere architecture does not include a master database.

In Adaptive Server Enterprise, the database owner (user ID **dbo**) owns the system tables. In Adaptive Server Anywhere, the system owner (user ID **SYS**) owns the system tables. A **dbo** user ID owns the Adaptive Server Enterprise-compatible system views provided by Adaptive Server Anywhere.

## Administrative roles

Adaptive Server Enterprise has a more elaborate set of administrative roles than Adaptive Server Anywhere. In Adaptive Server Enterprise there is a set of distinct roles, although more than one login account on an Adaptive Server Enterprise can be granted any role, and one account can possess more than one role.

### Adaptive Server Enterprise roles

In Adaptive Server Enterprise distinct roles include:

- ◆ **System Administrator** Responsible for general administrative tasks unrelated to specific applications; can access any database object.

- ◆ **System Security Officer** Responsible for security-sensitive tasks in Adaptive Server Enterprise, but has no special permissions on database objects.
- ◆ **Database Owner** Has full permissions on objects inside the database he or she owns, can add users to a database and grant other users the permission to create objects and execute commands within the database.
- ◆ **Data definition statements** Permissions can be granted to users for specific data definition statements, such as CREATE TABLE or CREATE VIEW, enabling the user to create database objects.
- ◆ **Object owner** Each database object has an owner who may grant permissions to other users to access the object. The owner of an object automatically has all permissions on the object.

In Adaptive Server Anywhere, the following database-wide permissions have administrative roles:

- ◆ The Database Administrator (DBA authority) has, like the Adaptive Server Enterprise database owner, full permissions on all objects inside the database (other than objects owned by SYS) and can grant other users the permission to create objects and execute commands within the database. The default database administrator is user ID **DBA**.
- ◆ The RESOURCE permission allows a user to create any kind of object within a database. This is instead of the Adaptive Server Enterprise scheme of granting permissions on individual CREATE statements.
- ◆ Adaptive Server Anywhere has object owners in the same way that Adaptive Server Enterprise does. The owner of an object automatically has all permissions on the object, including the right to grant permissions.

For seamless access to data held in both Adaptive Server Enterprise and Adaptive Server Anywhere, you should create user IDs with appropriate permissions in the database (RESOURCE in Adaptive Server Anywhere, or permission on individual CREATE statements in Adaptive Server Enterprise) and create objects from that user ID. If you use the same user ID in each environment, object names and qualifiers can be identical in the two databases, ensuring compatible access.

## Users and groups

There are some differences between the Adaptive Server Enterprise and Adaptive Server Anywhere models of users and groups.



In Adaptive Server Enterprise, users connect to a server, and each user requires a login ID and password to the server as well as a user ID for each database they want to access on that server. Each user of a database can only be a member of one group.

In Adaptive Server Anywhere, users connect directly to a database and do not require a separate login ID to the database server. Instead, each user receives a user ID and password on a database so they can use that database. Users can be members of many groups, and group hierarchies are allowed.

Both servers support user groups, so you can grant permissions to many users at one time. However, there are differences in the specifics of groups in the two servers. For example, Adaptive Server Enterprise allows each user to be a member of only one group, while Adaptive Server Anywhere has no such restriction. You should compare the documentation on users and groups in the two products for specific information.

Both Adaptive Server Enterprise and Adaptive Server Anywhere have a *public* group, for defining default permissions. Every user automatically becomes a member of the *public* group.

Adaptive Server Anywhere supports the following Adaptive Server Enterprise system procedures for managing users and groups.

☞ For the arguments to each procedure, see "Adaptive Server Enterprise system and catalog procedures" on page 737 of the book *ASA SQL Reference Manual*.

System procedure	Description
sp_addlogin	In Adaptive Server Enterprise, this adds a user to the server. In Adaptive Server Anywhere, this adds a user to a database.
sp_adduser	In Adaptive Server Enterprise and Adaptive Server Anywhere, this adds a user to a database. While this is a distinct task from <b>sp_addlogin</b> in Adaptive Server Enterprise, in Adaptive Server Anywhere, they are the same.
sp_addgroup	Adds a group to a database.
sp_changegroup	Adds a user to a group, or moves a user from one group to another.
sp_droplogin	In Adaptive Server Enterprise, removes a user from the server. In Adaptive Server Anywhere, removes a user from the database.
sp_dropuser	Removes a user from the database.
sp_dropgroup	Removes a group from the database.

Database object permissions

In Adaptive Server Enterprise, login IDs are server-wide. In Adaptive Server Anywhere, users belong to individual databases.

The Adaptive Server Enterprise and Adaptive Server Anywhere GRANT and REVOKE statements for granting permissions on individual database objects are very similar. Both allow SELECT, INSERT, DELETE, UPDATE, and REFERENCES permissions on database tables and views, and UPDATE permissions on selected columns of database tables. Both allow EXECUTE permissions to be granted on stored procedures.

For example, the following statement is valid in both Adaptive Server Enterprise and Adaptive Server Anywhere:

```
GRANT INSERT, DELETE
ON TITLES
TO MARY, SALES
```

This statement grants permission to use the INSERT and DELETE statements on the **TITLES** table to user **MARY** and to the **SALES** group.

Both Adaptive Server Anywhere and Adaptive Server Enterprise support the WITH GRANT OPTION clause, allowing the recipient of permissions to grant them in turn, although Adaptive Server Anywhere does not permit WITH GRANT OPTION to be used on a GRANT EXECUTE statement.

Database-wide permissions

Adaptive Server Enterprise and Adaptive Server Anywhere use different models for database-wide user permissions. These are discussed in "Users and groups" on page 390. Adaptive Server Anywhere employs DBA permissions to allow a user full authority within a database. The System Administrator in Adaptive Server Enterprise enjoys this permission for all databases on a server. However, DBA authority on an Adaptive Server Anywhere database is different from the permissions of an Adaptive Server Enterprise Database Owner, who must use the Adaptive Server Enterprise **SETUSER** statement to gain permissions on objects owned by other users.

Adaptive Server Anywhere employs RESOURCE permissions to allow a user the right to create objects in a database. A closely corresponding Adaptive Server Enterprise permission is GRANT ALL, used by a Database Owner.

# Configuring databases for Transact-SQL compatibility

You can eliminate some differences in behavior between Adaptive Server Anywhere and Adaptive Server Enterprise by selecting appropriate options when creating a database or, if you are working on an existing database, when rebuilding the database. You can control other differences by connection level options using the SET TEMPORARY OPTION statement in Adaptive Server Anywhere or the SET statement in Adaptive Server Enterprise.

## Creating a Transact-SQL-compatible database

This section describes choices you must make when creating or rebuilding a database.

### Quick start

Here are the steps you need to take to create a Transact-SQL-compatible database. The remainder of the section describes which options you need to set.

#### ❖ To create a Transact-SQL compatible database (Sybase Central):

- 1 Start Sybase Central.
- 2 Choose Tools►Adaptive Server Anywhere 8►Create Database.
- 3 Follow the instructions in the wizard.
- 4 When you see a button called Emulate Adaptive Server Enterprise, click it, and then click Next.
- 5 Follow the remaining instructions in the wizard.

#### ❖ To create a Transact-SQL compatible database (Command line):

- ◆ Enter the following command at a system prompt:

```
dbinit -b -c -k db-name.db
```

🔗 For more information about these options, see "Initialization utility options" on page 467 of the book *ASA Database Administration Guide*.

#### ❖ To create a Transact-SQL compatible database (SQL):

- 1 Connect to any Adaptive Server Anywhere database.
- 2 Enter the following statement, for example, in Interactive SQL:

```
CREATE DATABASE 'db-name.db'  
ASE COMPATIBLE
```

In this statement, ASE COMPATIBLE means compatible with Adaptive Server Enterprise.

### Make the database case-sensitive

By default, string comparisons in Adaptive Server Enterprise databases are case-sensitive, while those in Adaptive Server Anywhere are case insensitive.

When building an Adaptive Server Enterprise-compatible database using Adaptive Server Anywhere, choose the case-sensitive option.

- ◆ If you are using Sybase Central, this option is in the Create Database wizard.
- ◆ If you are using the *dbinit* command-line utility, specify the *-c* command-line switch.

### Ignore trailing blanks in comparisons

When building an Adaptive Server Enterprise-compatible database using Adaptive Server Anywhere, choose the option to ignore trailing blanks in comparisons.

- ◆ If you are using Sybase Central, this option is in the Create Database wizard.
- ◆ If you are using the *dbinit* command line utility, specify the *-b* command-line switch.

When you choose this option, Adaptive Server Enterprise and Adaptive Server Anywhere considers the following two strings equal:

```
'ignore the trailing blanks '  
'ignore the trailing blanks'
```

If you do not choose this option, Adaptive Server Anywhere considers the two strings above different.

A side effect of this choosing this option is that strings are padded with blanks when fetched by a client application.

### Remove historical system views

Older versions of Adaptive Server Anywhere employed two system views whose names conflict with the Adaptive Server Enterprise system views provided for compatibility. These views include SYSCOLUMNS and SYSINDEXES. If you are using Open Client or JDBC interfaces, create your database excluding these views. You can do this with the *dbinit -k* command-line switch.

If you do not use this option when creating your database, the following two statements return different results:

```
SELECT * FROM SYSCOLUMNS ;
```

```
SELECT * FROM dbo.syscolumns ;
```

❖ **To drop the Adaptive Server Anywhere system views from an existing database:**

- 1 Connect to the database as a user with DBA authority.
- 2 Execute the following statements:

```
DROP VIEW SYS.SYSCOLUMNS ;
```

```
DROP VIEW SYS.SYSINDEXES
```

**Caution**

*Ensure that you do not drop the `dbo.syscolumns` or `dbo.sysindexes` system view.*

## Setting options for Transact-SQL compatibility

You set Adaptive Server Anywhere database options using the SET OPTION statement. Several database option settings are relevant to Transact-SQL behavior.

Set the  
`allow_nulls_by_default`  
option

By default, Adaptive Server Enterprise disallows NULLs on new columns unless you explicitly tell the column to allow NULLs. Adaptive Server Anywhere permits NULL in new columns by default, which is compatible with the SQL/92 ISO standard.

To make Adaptive Server Enterprise behave in a SQL/92-compatible manner, use the **sp\_dboption** system procedure to set the **allow\_nulls\_by\_default** option to true.

To make Adaptive Server Anywhere behave in a Transact-SQL-compatible manner, set the **allow\_nulls\_by\_default** option to OFF. You can do this using the SET OPTION statement as follows:

```
SET OPTION PUBLIC.allow_nulls_by_default = 'OFF'
```

Set the  
`quoted_identifier`  
option

By default, Adaptive Server Enterprise treats identifiers and strings differently than Adaptive Server Anywhere, which matches the SQL/92 ISO standard.

The **quoted\_identifier** option is available in both Adaptive Server Enterprise and Adaptive Server Anywhere. Ensure the option is set to the same value in both databases, for identifiers and strings to be treated in a compatible manner.

For SQL/92 behavior, set the **quoted\_identifier** option to ON in both Adaptive Server Enterprise and Adaptive Server Anywhere.

For Transact-SQL behavior, set the **quoted\_identifier** option to OFF in both Adaptive Server Enterprise and Adaptive Server Anywhere. If you choose this, you can no longer use identifiers that are the same as keywords, enclosed in double quotes.

☞ For more information on the **quoted\_identifier** option, see "QUOTED\_IDENTIFIER option" on page 594 of the book *ASA Database Administration Guide*.

Set the automatic\_timestamp option to ON

Transact-SQL defines a **timestamp** column with special properties. With the **automatic\_timestamp** option set to ON, the Adaptive Server Anywhere treatment of **timestamp** columns is similar to Adaptive Server Enterprise behavior.

With the **automatic\_timestamp** option set to ON in Adaptive Server Anywhere (the default setting is OFF), any new columns with the **TIMESTAMP** data type that do not have an explicit default value defined receive a default value of **timestamp**.

☞ For information on **timestamp** columns, see "The special Transact-SQL timestamp column and data type" on page 398.

Set the string\_rtruncation option

Both Adaptive Server Enterprise and Adaptive Server Anywhere support the **string\_rtruncation** option, which affects error message reporting when an INSERT or UPDATE string is truncated. Ensure that each database has the option set to the same value.

☞ For more information on the **STRING\_RTRUNCATION** option, see "STRING\_RTRUNCATION option" on page 600 of the book *ASA Database Administration Guide*.

☞ For more information on database options for Transact-SQL compatibility, see "Compatibility options" on page 544 of the book *ASA Database Administration Guide*.

## Case-sensitivity

Case sensitivity in databases refers to:

- ◆ **Data** The case sensitivity of the data is reflected in indexes, in the results of queries, and so on.
- ◆ **Identifiers** Identifiers include table names, column names, user IDs, and so on.
- ◆ **Passwords** Case sensitivity of passwords is treated differently to other identifiers.

**Case sensitivity of data**

You decide the case-sensitivity of Adaptive Server Anywhere data in comparisons when you create the database. By default, Adaptive Server Anywhere databases are case-insensitive in comparisons, although data is always held in the case in which you enter it.

Adaptive Server Enterprise's sensitivity to case depends on the sort order installed on the Adaptive Server Enterprise system. Case sensitivity can be changed for single-byte character sets by reconfiguring the Adaptive Server Enterprise sort order.

**Case sensitivity of identifiers**

Adaptive Server Anywhere does not support case-sensitive identifiers. In Adaptive Server Enterprise, the case sensitivity of identifiers follows the case sensitivity of the data. User IDs are treated like any other identifier, and are always case insensitive. The default user ID for databases is upper case **DBA**.

In Adaptive Server Enterprise, domain names are case sensitive. In Adaptive Server Anywhere, they are case insensitive, with the exception of Java data types.

**User IDs and passwords**

In Adaptive Server Anywhere, passwords follow the case sensitivity of the data. The default password for databases is upper case **SQL**.

In Adaptive Server Enterprise, the case sensitivity of user IDs and passwords follows the case sensitivity of the server.

## Ensuring compatible object names

Each database object must have a unique name within a certain **name space**. Outside this name space, duplicate names are allowed. Some database objects occupy different name spaces in Adaptive Server Enterprise and Adaptive Server Anywhere.

In Adaptive Server Anywhere, indexes and triggers are owned by the owner of the table on which they are created. Index and trigger names must be unique for a given owner. For example, while the tables *t1* owned by user *user1* and *t2* owned by user *user2* may have indexes of the same name, no two tables owned by a single user may have an index of the same name.

Adaptive Server Enterprise has a less restrictive name space for index names than Adaptive Server Anywhere. Index names must be unique on a given table, but any two tables may have an index of the same name. For compatible SQL, stay within the Adaptive Server Anywhere restriction of unique index names for a given table owner.

Adaptive Server Enterprise has a more restrictive name space on trigger names than Adaptive Server Anywhere. Trigger names must be unique in the database. For compatible SQL, you should stay within the Adaptive Server Enterprise restriction and make your trigger names unique in the database.

## The special Transact-SQL timestamp column and data type

Adaptive Server Anywhere supports the Transact-SQL special *timestamp* column. The *timestamp* column, together with the *tsequal* system function, checks whether a row has been updated.

### Two meanings of timestamp

Adaptive Server Anywhere has a `TIMESTAMP` data type, which holds accurate date and time information. It is distinct from the special Transact-SQL `TIMESTAMP` column and data type.

Creating a  
Transact-SQL  
timestamp column  
in Adaptive Server  
Anywhere

To create a Transact-SQL *timestamp* column, create a column that has the (Adaptive Server Anywhere) data type `TIMESTAMP` and a default setting of *timestamp*. The column can have any name, although the name *timestamp* is common.

For example, the following `CREATE TABLE` statement includes a Transact-SQL *timestamp* column:

```
CREATE TABLE table_name (  
    column_1 INTEGER ,  
    column_2 TIMESTAMP DEFAULT TIMESTAMP  
)
```

The following `ALTER TABLE` statement adds a Transact-SQL *timestamp* column to the *sales\_order* table:

```
ALTER TABLE sales_order  
ADD timestamp TIMESTAMP DEFAULT TIMESTAMP
```

In Adaptive Server Enterprise a column with the name *timestamp* and no data type specified automatically receives a `TIMESTAMP` data type. In Adaptive Server Anywhere you must explicitly assign the data type yourself.

If you have the `AUTOMATIC_TIMESTAMP` database option set to `ON`, you do not need to set the default value: any new column created with `TIMESTAMP` data type and with no explicit default receives a default value of *timestamp*. The following statement sets `AUTOMATIC_TIMESTAMP` to `ON`:

```
SET OPTION PUBLIC.AUTOMATIC_TIMESTAMP='ON'
```

The data type of a  
timestamp column

Adaptive Server Enterprise treats a *timestamp* column as a domain that is `VARBINARY(8)`, allowing `NULL`, while Adaptive Server Anywhere treats a *timestamp* column as the `TIMESTAMP` data type, which consists of the date and time, with fractions of a second held to six decimal places.

When fetching from the table for later updates, the variable into which the timestamp value is fetched should correspond to the column description.



### Timestamping an existing table

If you add a special *timestamp* column to an existing table, all existing rows have a NULL value in the *timestamp* column. To enter a timestamp value (the current timestamp) for existing rows, update all rows in the table such that the data does not change. For example, the following statement updates all rows in the *sales\_order* table, without changing the values in any of the rows:

```
UPDATE sales_order
SET region = region
```

In Interactive SQL, you may need to set the `TIMESTAMP_FORMAT` option to see the differences in values for the rows. The following statement sets the `TIMESTAMP_FORMAT` option to display all six digits in the fractions of a second:

```
SET OPTION TIMESTAMP_FORMAT='YYYY-MM-DD HH:NN:ss.SSSSSS'
```

If all six digits are not shown, some *timestamp* column values may appear to be equal: they are not.

### Using tsequal for updates

With the *tsequal* system function you can tell whether a *timestamp* column has been updated or not.

For example, an application may `SELECT` a *timestamp* column into a variable. When an `UPDATE` of one of the selected rows is submitted, it can use the *tsequal* function to check whether the row has been modified. The *tsequal* function compares the timestamp value in the table with the timestamp value obtained in the `SELECT`. Identical timestamps means there are no changes. If the timestamps differ, the row has been changed since the `SELECT` was carried out.

A typical `UPDATE` statement using the *tsequal* function looks like this:

```
UPDATE publishers
SET city = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, '1995/10/25 11:08:34.173226')
```

The first argument to the *tsequal* function is the name of the special *timestamp* column; the second argument is the timestamp retrieved in the `SELECT` statement. In Embedded SQL, the second argument is likely to be a host variable containing a `TIMESTAMP` value from a recent `FETCH` on the column.

## The special IDENTITY column

To create an `IDENTITY` column, use the following `CREATE TABLE` syntax:

```
CREATE TABLE table-name (  
    ...  
    column-name numeric(n,0) IDENTITY NOT NULL,  
    ...  
)
```

where  $n$  is large enough to hold the value of the maximum number of rows that may be inserted into the table.

The IDENTITY column stores sequential numbers, such as invoice numbers or employee numbers, which are automatically generated. The value of the IDENTITY column uniquely identifies each row in a table.

In Adaptive Server Enterprise, each table in a database can have one IDENTITY column. The data type must be numeric with scale zero, and the IDENTITY column should not allow nulls.

In Adaptive Server Anywhere, the IDENTITY column is a column default setting. You can explicitly insert values that are not part of the sequence into the column with an INSERT statement. Adaptive Server Enterprise does not allow INSERTs into identity columns unless the *identity\_insert* option is *on*. In Adaptive Server Anywhere, you need to set the NOT NULL property yourself and ensure that only one column is an IDENTITY column. Adaptive Server Anywhere allows any numeric data type to be an IDENTITY column.

In Adaptive Server Anywhere the *identity* column and the AUTOINCREMENT default setting for a column are identical.

## Retrieving IDENTITY column values with @@identity

The first time you insert a row into the table, an IDENTITY column has a value of 1 assigned to it. On each subsequent insert, the value of the column increases by one. The value most recently inserted into an identity column is available in the @@identity global variable.

The value of @@identity changes each time a statement attempts to insert a row into a table.

- ◆ If the statement affects a table without an IDENTITY column, @@identity is set to 0.
- ◆ If the statement inserts multiple rows, @@identity reflects the last value inserted into the IDENTITY column.

This change is permanent. @@identity does not revert to its previous value if the statement fails or if the transaction that contains it is rolled back.

☞ For more information on the behavior of @@identity, see " @@identity global variable" on page 46 of the book *ASA SQL Reference Manual*.

## Writing compatible SQL statements

This section describes general guidelines for writing SQL for use on more than one database-management system, and discusses compatibility issues between Adaptive Server Enterprise and Adaptive Server Anywhere at the SQL statement level.

### General guidelines for writing portable SQL

When writing SQL for use on more than one database-management system, make your SQL statements as explicit as possible. Even if more than one server supports a given SQL statement, it may be a mistake to assume that default behavior is the same on each system. General guidelines applicable to writing compatible SQL include:

- ◆ Spell out all of the available options, rather than using default behavior.
- ◆ Use parentheses to make the order of execution within statements explicit, rather than assuming identical default order of precedence for operators.
- ◆ Use the Transact-SQL convention of an @ sign preceding variable names for Adaptive Server Enterprise portability.
- ◆ Declare variables and cursors in procedures, triggers, and batches immediately following a BEGIN statement. Adaptive Server Anywhere requires this, although Adaptive Server Enterprise allows declarations to be made anywhere in a procedure, trigger, or batch.
- ◆ Avoid using reserved words from either Adaptive Server Enterprise or Adaptive Server Anywhere as identifiers in your databases.
- ◆ Assume large namespaces. For example, ensure that each index has a unique name.

### Creating compatible tables

Adaptive Server Anywhere supports domains which allow constraint and default definitions to be encapsulated in the data type definition. It also supports explicit defaults and CHECK conditions in the CREATE TABLE statement. It does not, however, support named constraints or named defaults.

## NULL

Adaptive Server Anywhere and Adaptive Server Enterprise differ in some respects in their treatment of NULL. In Adaptive Server Enterprise, NULL is sometimes treated as if it were a value.

For example, a unique index in Adaptive Server Enterprise cannot contain rows that hold null and are otherwise identical. In Adaptive Server Anywhere, a unique index can contain such rows.

By default, columns in Adaptive Server Enterprise default to NOT NULL, whereas in Adaptive Server Anywhere the default setting is NULL. You can control this setting using the **allow\_nulls\_by\_default** option. Specify explicitly NULL or NOT NULL to make your data definition statements transferable.

🔗 For information on this option, see "Setting options for Transact-SQL compatibility" on page 395.

## Temporary tables

You can create a temporary table by placing a pound sign (#) in front of a CREATE TABLE statement. These temporary tables are Adaptive Server Anywhere declared temporary tables, and are available only in the current connection. For information about declared temporary tables in Adaptive Server Anywhere, see "DECLARE LOCAL TEMPORARY TABLE statement" on page 386 of the book *ASA SQL Reference Manual*.

Physical placement of a table is carried out differently in Adaptive Server Enterprise and in Adaptive Server Anywhere. Adaptive Server Anywhere supports the **ON segment-name** clause, but *segment-name* refers to an Adaptive Server Anywhere dbspace.

🔗 For information about the CREATE TABLE statement, see "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

## Writing compatible queries

There are two criteria for writing a query that runs on both Adaptive Server Anywhere and Adaptive Server Enterprise databases:

- ◆ The data types, expressions, and search conditions in the query must be compatible.
- ◆ The syntax of the SELECT statement itself must be compatible.

This section explains compatible SELECT statement syntax, and assumes compatible data types, expressions, and search conditions. The examples assume the QUOTED\_IDENTIFIER setting is OFF: the default Adaptive Server Enterprise setting, but not the default Adaptive Server Anywhere setting.

Adaptive Server Anywhere supports the following subset of the Transact-SQL SELECT statement.

**Syntax**

```
SELECT [ ALL | DISTINCT ] select-list
...[ INTO #temporary-table-name ]
...[ FROM table-spec [ HOLDLOCK | NOHOLDLOCK ],
...      table-spec [ HOLDLOCK | NOHOLDLOCK ], ... ]
...[ WHERE search-condition ]
...[ GROUP BY column-name, ... ]
...[ HAVING search-condition ]
    [ ORDER BY { expression | integer } [ ASC | DESC ], ... ]
```

**Parameters**

*select-list*:

```
table-name.*
| *
| expression
| alias-name = expression
| expression as identifier
| expression as T_string
```

*table-spec*:

```
[ owner . ] table-name
...[ [ AS ] correlation-name ]
...[ ( INDEX index_name [ PREFETCH size ] [ LRU | MRU ] ) ]
```

*alias-name*:

```
identifier | 'string' | "string"
```

☞ For a full description of the SELECT statement, see "SELECT statement" on page 526 of the book *ASA SQL Reference Manual*.

Adaptive Server Anywhere does not support the following keywords and clauses of the Transact-SQL SELECT statement syntax:

- ◆ **SHARED** keyword
- ◆ **COMPUTE** clause
- ◆ **FOR BROWSE** clause
- ◆ **GROUP BY ALL** clause

**Notes**

- ◆ The **INTO** *table\_name* clause, which creates a new table based on the SELECT statement result set, is supported only for declared temporary tables where the table name starts with a #. Declared temporary tables exist for a single connection only.
- ◆ Adaptive Server Anywhere does not support the Transact-SQL extension to the **GROUP BY** clause allowing references to columns and expressions that are not used for creating groups. In Adaptive Server Enterprise, this extension produces summary reports.
- ◆ The **FOR READ ONLY** clause and the **FOR UPDATE** clause are parsed, but have no effect.

- ◆ The performance parameters part of the table specification is parsed, but has no effect.
- ◆ The **HOLDLOCK** keyword is supported by Adaptive Server Anywhere. It makes a shared lock on a specified table or view more restrictive by holding it until the completion of a transaction (instead of releasing the shared lock as soon as the required data page is no longer needed, whether or not the transaction has been completed). For the purposes of the table for which the **HOLDLOCK** is specified, the query is carried out at isolation level 3.
- ◆ The **HOLDLOCK** option applies only to the table or view for which it is specified, and only for the duration of the transaction defined by the statement in which it is used. Setting the isolation level to 3 applies a holdlock for each select within a transaction. You cannot specify both a **HOLDLOCK** and **NOHOLDLOCK** option in a query.
- ◆ The **NOHOLDLOCK** keyword is recognized by Adaptive Server Anywhere, but has no effect.
- ◆ Transact-SQL uses the **SELECT** statement to assign values to local variables:

```
SELECT @localvar = 42
```

The corresponding statement in Adaptive Server Anywhere is the **SET** statement:

```
SET localvar = 42
```

The variable name can optionally be set using the **SET** statement and the Transact-SQL convention of an **@** sign preceding the name:

```
SET @localvar = 42
```

- ◆ Adaptive Server Enterprise does not support the following clauses of the **SELECT** statement syntax:
  - ◆ **INTO** host-variable-list
  - ◆ **INTO** variable-list.
  - ◆ Parenthesized queries.
- ◆ Adaptive Server Enterprise uses join operators in the **WHERE** clause , rather than the **FROM** clause and the **ON** condition for joins.

## Compatibility of joins

In Transact-SQL, joins appear in the **WHERE** clause, using the following syntax:

```

start of select, update, insert, delete, or subquery
FROM { table-list | view-list } WHERE [ NOT ]
...[ table-name.| view name.]column-name
    join-operator
...[ table-name.| view-name.]column_name
...[ { AND | OR } [ NOT ]
... [ table-name.| view-name.]column_name
    join-operator
    [ table-name.| view-name.]column-name
]...
end of select, update, insert, delete, or subquery

```

The *join-operator* in the WHERE clause may be any of the comparison operators, or may be either of the following **outer-join operators**:

- ◆ \*= Left outer join operator
- ◆ =\* Right outer join operator.

Adaptive Server Anywhere supports the Transact-SQL outer-join operators as an alternative to the native SQL/92 syntax. You cannot mix dialects within a query. This rule applies also to views used by a query—an outer-join query on a view must follow the dialect used by the view-defining query.

Adaptive Server Anywhere also provides a SQL/92 syntax for joins other than outer joins, in which the joins are placed in the FROM clause rather than the WHERE clause.

☞ For information about joins in Adaptive Server Anywhere and in the ANSI/ISO SQL standards, see "Joins: Retrieving Data from Several Tables" on page 227, and "FROM clause" on page 433 of the book *ASA SQL Reference Manual*.

☞ For more information on Transact-SQL compatibility of joins, see "Transact-SQL outer joins (\*= or =\*)" on page 245.

# Transact-SQL procedure language overview

The **stored procedure language** is the part of SQL used in stored procedures, triggers, and batches.

Adaptive Server Anywhere supports a large part of the Transact-SQL stored procedure language in addition to the Watcom-SQL dialect based on SQL/92.

## Transact-SQL stored procedure overview

Based on the ISO/ANSI draft standard, the Adaptive Server Anywhere stored procedure language differs from the Transact-SQL dialect in many ways. Many of the concepts and features are similar, but the syntax is different. Adaptive Server Anywhere support for Transact-SQL takes advantage of the similar concepts by providing automatic translation between dialects. However, a procedure must be written exclusively in one of the two dialects, not in a mixture of the two.

Adaptive Server  
Anywhere support  
for Transact-SQL  
stored procedures

There are a variety of aspects to Adaptive Server Anywhere support for Transact-SQL stored procedures, including:

- ◆ Passing parameters
- ◆ Returning result sets
- ◆ Returning status information
- ◆ Providing default values for parameters
- ◆ Control statements
- ◆ Error handling

## Transact-SQL trigger overview

Trigger compatibility requires compatibility of trigger features and syntax. This section provides an overview of the feature compatibility of Transact-SQL and Adaptive Server Anywhere triggers.

Adaptive Server Enterprise executes triggers after the triggering statement has completed: they are **statement level, after** triggers. Adaptive Server Anywhere supports both **row level** triggers (which execute before or after each row has been modified) and statement level triggers (which execute after the entire statement).



### Description of unsupported or different Transact-SQL triggers

Row-level triggers are not part of the Transact-SQL compatibility features, and are discussed in "Using Procedures, Triggers, and Batches" on page 507.

Features of Transact-SQL triggers that are either unsupported or different in Adaptive Server Anywhere include:

- ◆ **Triggers firing other triggers** Suppose a trigger carries out an action that would, if carried out directly by a user, fire another trigger. Adaptive Server Anywhere and Adaptive Server Enterprise respond slightly differently to this situation. By default in Adaptive Server Enterprise, triggers fire other triggers up to a configurable nesting level, which has the default value of 16. You can control the nesting level with the Adaptive Server Enterprise **nested triggers** option. In Adaptive Server Anywhere, triggers fire other triggers without limit unless there is insufficient memory.
- ◆ **Triggers firing themselves** Suppose a trigger carries out an action that would, if carried out directly by a user, fire the same trigger. Adaptive Server Anywhere and Adaptive Server Enterprise respond slightly differently to this situation. In Adaptive Server Anywhere, non-Transact-SQL triggers fire themselves recursively, while Transact-SQL dialect triggers do not fire themselves recursively.

By default in Adaptive Server Enterprise, a trigger does not call itself recursively, but you can turn on the **self\_recursion** option to allow triggers to call themselves recursively.

- ◆ **ROLLBACK statement in triggers** Adaptive Server Enterprise permits the ROLLBACK TRANSACTION statement within triggers, to roll back the entire transaction of which the trigger is a part. Adaptive Server Anywhere does not permit ROLLBACK (or ROLLBACK TRANSACTION) statements in triggers because a triggering action and its trigger together form an atomic statement.

Adaptive Server Anywhere does provide the Adaptive Server Enterprise-compatible ROLLBACK TRIGGER statement to undo actions within triggers. See "ROLLBACK TRIGGER statement" on page 524 of the book *ASA SQL Reference Manual*.

## Transact-SQL batch overview

In Transact-SQL, a **batch** is a set of SQL statements submitted together and executed as a group, one after the other. Batches can be stored in command files. The Interactive SQL utility in Adaptive Server Anywhere and the *isql* utility in Adaptive Server Enterprise provide similar capabilities for executing batches interactively.

The control statements used in procedures can also be used in batches. Adaptive Server Anywhere supports the use of control statements in batches and the Transact-SQL-like use of non-delimited groups of statements terminated with a GO statement to signify the end of a batch.

For batches stored in command files, Adaptive Server Anywhere supports the use of parameters in command files. Adaptive Server Enterprise does not support parameters.

↪ For information on parameters, see "PARAMETERS statement [Interactive SQL]" on page 493 of the book *ASA SQL Reference Manual*.

## Automatic translation of stored procedures

In addition to supporting Transact-SQL alternative syntax, Adaptive Server Anywhere provides aids for translating statements between the Watcom-SQL and Transact-SQL dialects. Functions returning information about SQL statements and enabling automatic translation of SQL statements include:

- ◆ **SQLDialect(statement)** Returns **Watcom-SQL** or **Transact-SQL**.
- ◆ **WatcomSQL(statement)** Returns the Watcom-SQL syntax for the statement.
- ◆ **TransactSQL(statement)** Returns the Transact-SQL syntax for the statement.

These are functions, and so can be accessed using a select statement from Interactive SQL. For example:

```
select SqlDialect('select * from employee')
```

returns the value Watcom-SQL.

## Using Sybase Central to translate stored procedures

Sybase Central has facilities for creating, viewing, and altering procedures and triggers.

### ❖ To translate a stored procedure using Sybase Central:

- 1 Connect to a database using Sybase Central, either as owner of the procedure you wish to change, or as a DBA user.
- 2 Open the Procedures & Functions folder.
- 3 Right-click the procedure you want to translate and from the popup menu choose one of the Open As commands, depending on the dialect you want to use.

The procedure appears in the Code Editor in the selected dialect. If the selected dialect is not the one in which the procedure is stored, the server translates it to that dialect. Any untranslated lines appear as comments.

- 4 Rewrite any untranslated lines as needed.
- 5 When finished, choose File ► Save/Execute in Database to save the translated version to the database. You can also export the text to a file for editing outside Sybase Central.

## Returning result sets from Transact-SQL procedures

Adaptive Server Anywhere uses a **RESULT** clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

Example of  
Transact-SQL  
procedure


The following Transact-SQL procedure illustrates how Transact-SQL stored procedures returns result sets:

```
CREATE PROCEDURE showdept (@deptname varchar(30))
AS
    SELECT employee.emp_lname, employee.emp_fname
    FROM department, employee
    WHERE department.dept_name = @deptname
    AND department.dept_id = employee.dept_id
```

Example of  
Watcom-SQL  
procedure

The following is the corresponding Adaptive Server Anywhere procedure:

```
CREATE PROCEDURE showdept(in deptname varchar(30))
RESULT ( lastname char(20), firstname char(20))
BEGIN
    SELECT employee.emp_lname, employee.emp_fname
    FROM department, employee
    WHERE department.dept_name = deptname
    AND department.dept_id = employee.dept_id
END
```

 For more information about procedures and results, see "Returning results from procedures" on page 539

## Variables in Transact-SQL procedures

Adaptive Server Anywhere uses the SET statement to assign values to variables in a procedure. In Transact-SQL, values are assigned using either the SELECT statement with an empty table-list, or the SET statement. The following simple procedure illustrates how the Transact-SQL syntax works:

```
CREATE PROCEDURE multiply
    @mult1 int,
    @mult2 int,
    @result int output
AS
SELECT @result = @mult1 * @mult2
```

This procedure can be called as follows:

```
CREATE VARIABLE @product int
go

EXECUTE multiply 5, 6, @product OUTPUT
go
```

The variable **@product** has a value of 30 after the procedure executes.

☞ For more information on using the SELECT statement to assign variables, see "Writing compatible queries" on page 402. For more information on using the SET statement to assign variables, see "SET statement" on page 531 of the book *ASA SQL Reference Manual*.

## Error handling in Transact-SQL procedures

Default procedure error handling is different in the Watcom-SQL and Transact-SQL dialects. By default, Watcom-SQL dialect procedures exit when they encounter an error, returning SQLSTATE and SQLCODE values to the calling environment.

Explicit error handling can be built into Watcom-SQL stored procedures using the EXCEPTION statement, or you can instruct the procedure to continue execution at the next statement when it encounters an error, using the ON EXCEPTION RESUME statement.

When a Transact-SQL dialect procedure encounters an error, execution continues at the following statement. The global variable @@error holds the error status of the most recently executed statement. You can check this variable following a statement to force return from a procedure. For example, the following statement causes an exit if an error occurs.

```
IF @@error != 0 RETURN
```

When the procedure completes execution, a return value indicates the success or failure of the procedure. This return status is an integer, and can be accessed as follows:

```
DECLARE @status INT
EXECUTE @status = proc_sample
IF @status = 0
    PRINT 'procedure succeeded'
ELSE
    PRINT 'procedure failed'
```

The following table describes the built-in procedure return values and their meanings:

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Data type error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Non-fatal internal problem

Value	Meaning
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

The RETURN statement can be used to return other integers, with their own user-defined meanings.

## Using the RAISERROR statement in procedures

The RAISERROR statement is a Transact-SQL statement for generating user-defined errors. It has a similar function to the SIGNAL statement.

☞ For a description of the RAISERROR statement, see "RAISERROR statement [T-SQL]" on page 501 of the book *ASA SQL Reference Manual*.

By itself, the RAISERROR statement does not cause an exit from the procedure, but it can be combined with a RETURN statement or a test of the @@error global variable to control execution following a user-defined error.

If you set the ON\_TSQL\_ERROR database option to CONTINUE, the RAISERROR statement no longer signals an execution-ending error. Instead, the procedure completes and stores the RAISERROR status code and message, and returns the most recent RAISERROR. If the procedure causing the RAISERROR was called from another procedure, the RAISERROR returns after the outermost calling procedure terminates.

You lose intermediate RAISERROR statuses and codes after the procedure terminates. If, at return time, an error occurs along with the RAISERROR, then the error information is returned and you lose the RAISERROR information. The application can query intermediate RAISERROR statuses by examining @@error global variable at different execution points.

## Transact-SQL-like error handling in the Watcom-SQL dialect

You can make a Watcom-SQL dialect procedure handle errors in a Transact-SQL-like manner by supplying the ON EXCEPTION RESUME clause to the CREATE PROCEDURE statement:

```
CREATE PROCEDURE sample_proc()  
ON EXCEPTION RESUME  
BEGIN  
    . . .  
END
```

The presence of an ON EXCEPTION RESUME clause prevents explicit exception handling code from being executed, so avoid using these two clauses together.



C H A P T E R   1 3

Differences from Other SQL Dialects

About this chapter

Adaptive Server Anywhere complies completely with the SQL-92-based United States Federal Information Processing Standard Publication (FIPS PUB) 127.

Adaptive Server Anywhere is entry-level compliant with the ISO/ANSI SQL-92 standard, and with minor exceptions is compliant with SQL-99 core specifications.

Complete, detailed information about compliance is provided in the reference documentation for each feature of Adaptive Server Anywhere.

This chapter describes those features of Adaptive Server Anywhere that are not commonly found in other SQL implementations.

Contents	<table><tr><th>Topic</th><th>Page</th></tr><tr><td>Adaptive Server Anywhere SQL features</td><td>416</td></tr></table>	Topic	Page	Adaptive Server Anywhere SQL features	416
Topic	Page				
Adaptive Server Anywhere SQL features	416				

## Adaptive Server Anywhere SQL features

The following features of the SQL supported by Adaptive Server Anywhere are not found in many other SQL implementations.

**Type conversions** Full type conversion is implemented. Any data type can be compared with or used in any expression with any other data type.

**Dates** Adaptive Server Anywhere has date, time and timestamp types that includes a year, month and day, hour, minutes, seconds and fraction of a second. For insertions or updates to date fields, or comparisons with date fields, a free format date is supported.

In addition, the following operations are allowed on dates:

- ◆ **date + integer** Add the specified number of days to a date.
- ◆ **date - integer** Subtract the specified number of days from a date.
- ◆ **date - date** Compute the number of days between two dates.
- ◆ **date + time** Make a timestamp out of a date and time.

Also, many functions are provided for manipulating dates and times. See "SQL Functions" on page 93 of the book *ASA SQL Reference Manual* for a description of these.

**Integrity** Adaptive Server Anywhere supports both entity and referential integrity. This has been implemented via the following two extensions to the CREATE TABLE and ALTER TABLE commands.

```
PRIMARY KEY ( column-name, ... )

[NOT NULL] FOREIGN KEY [role-name]
    [(column-name, ...)]
REFERENCES table-name [(column-name, ...)]
    [ CHECK ON COMMIT ]
```

The PRIMARY KEY clause declares the primary key for the relation. Adaptive Server Anywhere will then enforce the uniqueness of the primary key, and ensure that no column in the primary key contains the NULL value.

The FOREIGN KEY clause defines a relationship between this table and another table. This relationship is represented by a column (or columns) in this table which must contain values in the primary key of another table. The system will then ensure referential integrity for these columns - whenever these columns are modified or a row is inserted into this table, these columns will be checked to ensure that either one or more is NULL or the values match the corresponding columns for some row in the primary key of the other table. For more information, see "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

Joins	Adaptive Server Anywhere allows <b>automatic joins</b> between tables. In addition to the NATURAL and OUTER join operators supported in other implementations, Adaptive Server Anywhere allows KEY joins between tables based on foreign key relationships. This reduces the complexity of the WHERE clause when performing joins.
Updates	Adaptive Server Anywhere allows more than one table to be referenced by the UPDATE command. Views defined on more than one table can also be updated. Many SQL implementations will not allow updates on joined tables.
Altering tables	<p>The ALTER TABLE command has been extended. In addition to changes for entity and referential integrity, the following types of alterations are allowed:</p> <pre> ADD column data-type MODIFY column data-type DELETE column RENAME new-table-name RENAME old-column TO new-column </pre> <p>The MODIFY can be used to change the maximum length of a character column as well as converting from one data type to another. For more information, see "ALTER TABLE statement" on page 233 of the book <i>ASA SQL Reference Manual</i>.</p>
Subqueries where expressions are allowed	<p>Adaptive Server Anywhere allows subqueries to appear wherever expressions are allowed. Many SQL implementations only allow subqueries on the right side of a comparison operator. For example, the following command is valid in Adaptive Server Anywhere but not valid in most other SQL implementations.</p> <pre> SELECT    emp_lname,           emp_birthdate,           ( SELECT skill             FROM department             WHERE emp_id = employee.emp_ID             AND dept_id = 200 ) FROM employee </pre>
Additional functions	Adaptive Server Anywhere supports several functions not in the ANSI SQL definition. See "SQL Functions" on page 93 of the book <i>ASA SQL Reference Manual</i> for a full list of available functions.
Cursors	When using Embedded SQL, cursor positions can be moved arbitrarily on the FETCH statement. Cursors can be moved forward or backward relative to the current position or a given number of records from the beginning or end of the cursor.



## PART FOUR

# Accessing and Moving Data

This part describes how to load and unload your database, and how to access remote data.

---

# Importing and Exporting Data

About this chapter      This chapter describes the Adaptive Server Anywhere tools and utilities that help you achieve your importing and exporting goals, including SQL, Interactive SQL, the *dbunload* utility and Sybase Central wizards.

Contents

Topic	Page
Introduction to import and export	422
Importing and exporting data	424
Importing	428
Exporting	433
Rebuilding databases	440
Extracting data	448
Migrating databases to Adaptive Server Anywhere	449
Adaptive Server Enterprise compatibility	454

## Introduction to import and export

Transferring large amounts of data into and from your database may be necessary in several situations. For example,

- ◆ Importing an initial set of data into a new database.
- ◆ Exporting data from your database for use with other applications, such as spreadsheets.
- ◆ Building new copies of a database, perhaps with a modified structure.
- ◆ Creating extractions of a database for replication or synchronization.

## Performance considerations of moving data

The Interactive SQL INPUT and OUTPUT commands are external to the database (client-side). If ISQL is being run on a different machine than the database server, paths to files being read or written are relative to the client. An INPUT is recorded in the transaction log as a separate INSERT statement for each row read. As a result, INPUT is considerably slower than LOAD TABLE. This also means that ON INSERT triggers will fire during an INPUT. Missing values will be inserted as NULL on NULLABLE rows, as 0 (zero) on non-nullable numeric columns, and as an empty string on non-nullable non-numeric columns. The OUTPUT statement is useful when compatibility is an issue since it can write out the result set of a SELECT statement to any one of a number of file formats.

The LOAD TABLE, UNLOAD TABLE and UNLOAD statements, on the other hand, are internal to the database (server-side). Paths to files being written or read are relative to the database server. Only the command travels to the database server, where all processing happens. A LOAD table statement is recorded in the transaction log as a single command. The data file must contain the same number of columns as the table to be loaded. Missing values on columns with a default value will be inserted as NULL, zero or an empty string if the DEFAULTS option is set to OFF (default), or as the default value if the DEFAULTS value is set to ON. Internal importing and exporting only provides access to text and BCP formats, but it is a faster method.

Although loading large volumes of data into a database can be very time consuming, there are a few things you can do to save time:

- ◆ If you use the LOAD TABLE statement, then bulk mode (starting the server with the `-b` option) is not necessary.



- ◆ If you are using the INPUT command, run Interactive SQL or the client application on the same machine as the server. Loading data over the network adds extra communication overhead. This might mean loading new data during off hours.
- ◆ Place data files on a separate physical disk drive from the database. This could avoid excessive disk head movement during the load.
- ◆ If you are using the INPUT command, start the server with the `-b` option for bulk operations mode. In this mode, the server does not keep a rollback log or a transaction log, it does not perform an automatic COMMIT before data definition commands, and it does not lock any records.

The server allows only one connection when you use the `-b` option.

Without a rollback log, you cannot use savepoints and aborting a command always causes transactions to roll back. Without automatic COMMIT, a ROLLBACK undoes everything since the last explicit COMMIT.

Without a transaction log, there is no log of the changes. You should back up the database before and after using bulk operations mode because, in this mode, your database is not protected against media failure. For more information, see "Backup and Data Recovery" on page 299 of the book *ASA Database Administration Guide*.

If you have data that requires many commits, running with the `-b` option may slow database operation. At each COMMIT, the server carries out a checkpoint; this frequent checkpointing can slow the server.

- ◆ Extend the size of the database, as described in "ALTER DBSPACE statement" on page 209 of the book *ASA SQL Reference Manual*. This command allows a database to be extended in large amounts before the space is required, rather than the normal 256 kb at a time when the space is needed. As well as improving performance for loading large amounts of data, it also serves to keep the database more contiguous within the file system.
- ◆ You can use temporary tables to load data. Local or global temporary tables are useful when you need to load a set of data repeatedly, or when you need to merge tables with different structures.

## Importing and exporting data

You can import individual tables or portions of tables from other database file formats, or from ASCII files. Depending on the format of the data you are inserting, there is some flexibility as to whether you create the table before the import or during the import. You may find importing a useful tool if you need to add large amounts of data to your database at a time.

You can export individual tables and query results in ASCII format, or in a variety of formats supported by other database programs. You may find exporting a useful tool if you need to share large portions of your database, or extract portions of your database according to particular criteria.

Although Adaptive Server Anywhere import and export procedures work on one table at a time, you can create scripts that effectively automate the importing or export procedure, allowing you to import and export data into or from a number of tables consecutively.

You can insert (append) data into tables, and you can replace data in tables. In some cases, you can also create new tables at the same time as you import the data. If you are trying to create a whole new database, however, consider loading the data instead of importing it, for performance reasons.

You can export query results, table data, or table schema. If you are trying to export a whole database, however, consider unloading the database instead of exporting data, for performance reasons.

☞ For more information about loading and unloading complete databases, see "Rebuilding databases" on page 440.

You can import and export files between Adaptive Server Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause.

☞ For more information, see "Adaptive Server Enterprise compatibility" on page 454.

## Data formats

Interactive SQL supports the following import and export file formats:

File Format	Description	Available for Importing	Available for Exporting
<b>ASCII</b>	A text file, one row per line, with values separated by a delimiter. String values optionally appear enclosed in apostrophes (single quotes). This is the same as the format used by LOAD TABLE and UNLOAD TABLE.	✓	✓
<b>DBASEII</b>	DBASE II format	✓	✓
<b>DBASEIII</b>	DBASE III format	✓	✓
<b>Excel 2.1</b>	Excel format 2.1	✓	✓
<b>FIXED</b>	Data records appear in fixed format with the width of each column either the same as defined by the column's type or specified as a parameter.	✓	✓
<b>FOXPRO</b>	FoxPro format	✓	✓
<b>HTML</b>	HTML (Hyper Text Markup Language) format		✓
<b>LOTUS</b>	Lotus workspace format	✓	✓
<b>SQL Statements</b>	The SQL statement format. This format can be used as an argument in a READ statement.	Using the READ statement only	✓
<b>XML</b>	The generated XML file is encoded in UTF-8 and contains an embedded DTD. Binary values are encoded in CDATA blocks with the binary data rendered as 2-hex-digit strings.	No	✓

## Table structures for import

The structure of the data you want to load into a table does not always match the structure of the destination table itself, which may present problems during importing. For example, the column data types may be different or in a different order, or there may be extra values in the import data that do not match columns in the destination table.

### Rearranging the table or data

If you know that the structure of the data you want to import does not match the structure of the destination table, you have several options. You can rearrange the columns in your table using the `LOAD TABLE` statement; you can rearrange the import data to fit the table using a variation of the `INSERT` statement and a global temporary table; or you can use the `INPUT` statement to specify a specific set or order of columns.

### Allowing columns to contain NULLs

If the file you are importing contains data for a subset of the columns in a table, or if the columns are in a different order, you can also use the `LOAD TABLE` statement `DEFAULTS` option to fill in the blanks and merge non-matching table structures.

If `DEFAULTS` is `OFF`, any column not present in the column list is assigned `NULL`. If `DEFAULTS` is `OFF` and a non-nullable column is omitted from the column list, the database server attempts to convert the empty string to the column's type. If `DEFAULTS` is `ON` and the column has a default value, that value is used.

For example, to load two columns into the employee table, and set the remaining column values to the default values if there are any, the `LOAD TABLE` statement should look like this:

```
LOAD TABLE employee (emp_lname, emp_fname)
FROM 'new_employees.txt'
DEFAULTS ON
```

### Merging different table structures

You can rearrange the import data to fit the table using a variation of the `INSERT` statement and a global temporary table.

#### ❖ To load data with a different structure using a global temporary table:

- 1 In the SQL Statements pane of the Interactive SQL window, create a global temporary table with a structure matching that of the input file.

You can use the `CREATE TABLE` statement to create the global temporary table.


- 2 Use the `LOAD TABLE` statement to load your data into the global temporary table.

When you close the database connection, the data in the global temporary table disappears. However, the table definition remains. You can use it the next time you connect to the database.

- 3 Use the `INSERT` statement with a `FROM SELECT` clause to extract and summarize data from the temporary table and put it into one or more permanent database tables.

## Conversion errors during import

When you load data from external sources, there may be errors in the data. For example, there may be dates that are not valid dates and numbers that are not valid numbers. The `CONVERSION_ERROR` database option allows you to ignore conversion errors by converting them to NULL values.

 For more information about setting Interactive SQL database options, see "SET OPTION statement" on page 539 of the book *ASA SQL Reference Manual*, or "CONVERSION\_ERROR option" on page 560 of the book *ASA Database Administration Guide*.

## Outputting NULLs


Users often want to extract data for use in other software products. Since the other software package may not understand NULL values, there are two ways of specifying how NULL values are output. You can use either the Interactive SQL NULLS option, or the IFNULL function. Both options allow you to output a specific value in place of a NULL value.

Use the Interactive SQL NULLS option to set the default behavior, or to change the output value for a particular session. Use the IFNULL function to apply the output value to a particular instance or query.

Specifying how NULL values are output provides for greater compatibility with other software packages.

### ❖ To specify NULL value output (Interactive SQL):

- 1 From the Interactive SQL window, choose Tools►Options to display the Options dialog.
- 2 Click the Results tab.
- 3 In the Display Null Values As field, type the value you want to replace null values with.
- 4 Click Make Permanent if you want the changes to become the default, or click OK if you want the changes to be in effect only for this session.

 For more information on setting Interactive SQL options, see "SET OPTION statement" on page 539 of the book *ASA SQL Reference Manual*.

# Importing

Following is a summary of import tools, followed by instructions for importing databases, data, and tables.

## Import tools

There are a variety of tools available to help you import your data.

### Interactive SQL Import wizard

You can access the import wizard by choosing **Data►Import** from the Interactive SQL menu. The wizard provides an interface to allow you to choose a file to import, a file format, and a destination table to place the data in. You can choose to import this data into an existing table, or you can use the wizard to create and configure a completely new table.

Choose the Interactive SQL Import wizard when you prefer using a graphical interface to import data in a format other than text, or when you want to create a table at the same time you import the data.

### INPUT statement

You execute the INPUT statement from the SQL Statements pane of the Interactive SQL window. The INPUT statement allows you to import data in a variety of file formats into one or more tables. You can choose a default input format, or you can specify the file format on each INPUT statement. Interactive SQL can execute a command file containing multiple INPUT statements.

If a data file is in DBASE, DBASEII, DBASEIII, FOXPRO, or LOTUS format and the table does not exist, it will be created. There are performance impacts associated with importing large amounts of data with the INPUT statement, since the INPUT statement writes everything to the Transaction log.

Choose the Interactive SQL INPUT statement when you want to import data into one or more tables, when you want to automate the import process using a command file, or when you want to import data in a format other than text.

☞ For more information, see "INPUT statement [Interactive SQL]" on page 459 of the book *ASA SQL Reference Manual*.

### LOAD TABLE statement

The LOAD TABLE statement allows you to import data only, into a table, in an efficient manner in text/ASCII/FIXED formats. The table must exist and have the same number of columns as the input file has fields, defined on compatible data types. The LOAD TABLE statement imports with one row per line, with values separated by a delimiter.

Use the LOAD TABLE statement when you want to import data in text format. If you have a choice between using the INPUT statement or the LOAD TABLE statement, choose the LOAD TABLE statement for better performance.

🔗 For more information, see "LOAD TABLE statement" on page 472 of the book *ASA SQL Reference Manual*.

#### INSERT statement

Since you include the data you want to place in your table directly in the INSERT statement, it is considered interactive input. File formats are not an issue. You can also use the INSERT statement with remote data access to import data from another database rather than a file.

Choose the INSERT statement when you want to import small amounts of data into a single table.

🔗 For more information, see "INSERT statement" on page 463 of the book *ASA SQL Reference Manual*.

#### Proxy Tables

You can import data directly from another database. Using the Adaptive Server Anywhere remote data access feature, you can create a proxy table, which represents a table from the remote database, and then use an INSERT statement with a SELECT clause to insert data from the remote database into a permanent table in your database.

🔗 For more information about remote data access, see "Accessing Remote Data" on page 455.

## Importing databases

You can use either the Interactive SQL Import wizard or the INPUT statement to create a database by importing one table at a time. You can also create a script that automates this process. However, for more efficient results, consider reloading a database whenever possible.

🔗 For more information about importing a database that was previously unloaded, see "Reloading a Database" on page 444.

## Importing data

### ❖ To import data (Interactive SQL Data Menu):

- 1 From the Interactive SQL window, choose Data ► Import.  
The Open dialog appears.
- 2 Locate the file you want to import and click Open.

You can import data in text, DBASEII, Excel 2.1, FOXPRO, and Lotus formats.

The Import wizard appears.

- 3 Specify how the database values are stored in the file you are importing.
- 4 Select the Use An Existing Table option and then enter the name and location of the existing table. Click Next.

You can click the **Browse** button and locate the table you want to import the data into.

- 5 Follow the remaining instructions in the wizard.

In this case, importing appends the new data to the existing table. If the import is successful, the Messages pane displays the amount of time it took to import the data. If the import is unsuccessful, a message appears indicating the import was unsuccessful. The Results tab in the Results pane displays what execution plan was used.

### ❖ To import data (INSERT statement):

- 1 Ensure that the table you want to place the data in exists.
- 2 Execute an INSERT statement. For example,

```
INSERT INTO t1  
VALUES ( ... )
```

Inserting values appends the new data to the existing table.

### ❖ To import data (Interactive SQL INPUT statement):

- 1 Ensure that the table you want to place the data in exists.
- 2 Enter an INPUT statement in the SQL Statements pane of the Interactive SQL window. For example,

```
INPUT INTO t1  
FROM file1  
FORMAT ASCII;
```

Where *t1* is the name of the table you want to place the data in, and *file1* is the name of the file that holds the data you want to import.

- 3 Execute the statement.

If the import is successful, the Messages pane displays the amount of time it took to import the data. If the import is unsuccessful, a message appears indicating the import was unsuccessful. The Results tab in the Results pane displays what execution plan was used.



☞ For more information about using the INPUT statement to import data, see "INPUT statement [Interactive SQL]" on page 459 of the book *ASA SQL Reference Manual*.

## Importing a table

### ❖ To import a table (Interactive SQL Data Menu):

- 1 Ensure that the table you want to place the data in exists.
- 2 From the Interactive SQL window, choose Data►Import.  
The Open dialog appears.
- 3 Locate the file you want to import and click Open.  
You can import data in text, DBASEII, Excel 2.1, FOXPRO, and Lotus formats.  
The Import wizard appears.
- 4 Select the Create A New Table With The Following Name option and enter a name for the new table in the field.
- 5 Follow the remaining instructions in the wizard.  
If the import is successful, the Messages pane displays the amount of time it took to import the data. If the import is unsuccessful, a message appears indicating the import was unsuccessful. The Results tab in the Results pane displays what execution plan was used.

### ❖ To import a table (Interactive SQL):

- 1 In the SQL Statements pane of the Interactive SQL window, create the table you want to load data into.  
You can use the CREATE TABLE statement to create the table.
- 2 Execute a LOAD TABLE statement. For example,

```
LOAD TABLE department  
FROM 'dept.txt'
```

The LOAD TABLE statement appends the contents of the file to the existing rows of the table; it does not replace the existing rows in the table. You can use the TRUNCATE TABLE statement to remove all the rows from a table.

Neither the TRUNCATE TABLE statement nor the LOAD TABLE statement fires triggers, including referential integrity actions such as cascaded deletes.

The LOAD TABLE statement has an optional STRIP clause. The default setting (STRIP ON) strips trailing blanks from values before inserting them. To keep trailing blanks, use the STRIP OFF clause in your LOAD TABLE statement.

🔗 For more information about the LOAD TABLE statement syntax, see "LOAD TABLE statement" on page 472 of the book *ASA SQL Reference Manual*.

# Exporting

Following is a summary of export tools, followed by instructions for exporting query results, databases, and tables.

## Export tools

Exporting data  
from  
Interactive SQL  
OUTPUT  
statement

There are a variety of tools available to help you export your data.

You can export data from Interactive SQL by choosing Export from the Data menu. This allows you to choose the format of the exported query results.

You can export query results, tables or views from your database using the Interactive SQL OUTPUT statement. The Interactive SQL OUTPUT statement supports several different file formats. You can either specify the default output format, or you can specify the file format on each OUTPUT statement. Interactive SQL can execute a command file containing multiple OUTPUT statements.

There are performance impacts associated with exporting large amounts of data with the OUTPUT statement. As well, you should use the OUTPUT statement on the same machine as the server if possible to avoid sending large amounts of data across the network.

Choose the Interactive SQL OUTPUT statement when you want to export all or part of a table or view in a format other than text, or when you want to automate the export process using a command file.

 For more information, see "OUTPUT statement [Interactive SQL]" on page 488 of the book *ASA SQL Reference Manual*.

UNLOAD TABLE  
statement

You execute the UNLOAD TABLE statement from the SQL Statements pane of the Interactive SQL window. It allows you to export data only, in an efficient manner in text/ASCII/FIXED formats. The UNLOAD TABLE statement exports with one row per line, and values separated by a comma delimiter. The data exports in order by primary key values to make reloading quicker.

Choose the UNLOAD TABLE statement when you want to export entire tables in text format. If you have a choice between using the OUTPUT statement, UNLOAD statement, or UNLOAD TABLE statement, choose the UNLOAD TABLE statement for performance reasons.

 For more information, see "UNLOAD TABLE statement" on page 573 of the book *ASA SQL Reference Manual*.

### UNLOAD statement

The UNLOAD statement is similar to the OUTPUT statement in that they both export query results to a file. The UNLOAD statement, however, allows you to export data in a more efficient manner and in text/ASCII/FIXED formats only. The UNLOAD statement exports with one row per line, with values separated by a comma delimiter.

To use the UNLOAD statement, the user must have ALTER or SELECT permission on the table. For more information about controlling who can use the UNLOAD statement, see "-gl server option" on page 141 of the book *ASA Database Administration Guide*.

Choose the UNLOAD statement when you want to export query results if performance is an issue, and if output in text format is acceptable. The UNLOAD statement is also a good choice when you want to embed an export command in an application.

When unloading and reloading a database that has proxy tables, you must create an external login to map the local user to the remote user, even if the user has the same password on both the local and remote databases. If you do not have an external login, the reload may fail because you cannot connect to the remote server.

For more information, see "UNLOAD statement" on page 571 of the book *ASA SQL Reference Manual*.

### Dbunload utility

The *dbunload* utility and Sybase Central are graphically different, and functionally equivalent. You can use either one interchangeably to produce the same results. These tools are different from Interactive SQL statements in that they can operate on several tables at once. And in addition to exporting table data, both tools can also export table schema.

If you want to rearrange your tables in the database, you can use *dbunload* to create the necessary command files and modify them as needed. Sybase Central provides wizards and a GUI interface for unloading one, many or all of the tables in a database, and dbunload provides command line options for the same activities. Tables can be unloaded with structure only, data only or both structure and data. To unload fewer than all of the tables in a database, a connection must be established beforehand.

You can also extract one or many tables with or without command files. These files can be used to create identical tables in different databases.

Choose Sybase Central or the *dbunload* utility when you want to export in text format, when you need to process large amounts of data quickly, when your file format requirements are flexible, or when your database needs to be rebuilt or extracted.

☞ For more information, see "Unloading a database using the dbunload command-line utility" on page 514 of the book *ASA Database Administration Guide*.

## Exporting query results

You can export queries (including queries on views) to a file from Interactive SQL using the Data menu or the OUTPUT statement.

You can import and export files between Adaptive Server Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause.

☞ For more information, see "Adaptive Server Enterprise compatibility" on page 454.

### ❖ To export query results (Interactive SQL Data menu):

- 1 Enter your query in the SQL Statements pane of the Interactive SQL window.
- 2 Click Execute SQL statement(s) to display the result set.
- 3 Choose Data ► Export.  
The Save As dialog appears.
- 4 Specify a name and location for the exported data.
- 5 Specify the file format and click Save.

If the export is successful, the Messages pane displays the amount of time it took to export the query result set, the filename and path of the exported data, and the number of rows written.

If the export is unsuccessful, a message appears indicating the export was unsuccessful.

### ❖ To export query results (Interactive SQL OUTPUT statement):

- 1 Enter your query in the SQL Statements pane of the Interactive SQL window.
- 2 At the end of the query, type **OUTPUT TO 'c:\filename'**.

For example, to export the entire employee table to the file *employee.dbf*, enter the following query:

```
SELECT *  
FROM employee;  
OUTPUT TO 'c:\employee.dbf'
```

- 3 If you want to export query results and append the results to another file, add the APPEND statement to the end of the OUTPUT statement.

For example,

```
SELECT *  
FROM employee;  
OUTPUT TO 'c:\employee.dbf' APPEND
```

- 4 If you want to export query results and include messages, add the VERBOSE statement to the end of the OUTPUT statement.

For example,

```
SELECT *  
FROM employee;  
OUTPUT TO 'c:\employee.dbf' VERBOSE
```

- 5 If you want to specify a format other than ASCII, add a FORMAT clause to the end of the query.

For example,


```
SELECT *  
FROM employee;  
OUTPUT TO 'c:\employee.dbf'  
FORMAT dbaseiii;
```

where *c:\employee.dbf* is the path, name, and extension of the new file and *dbaseiii* is the file format for this file. You can enclose the string in single or double quotation marks, but they are only required if the path contains embedded spaces.

Where *dbaseiii* is the file format for this file. If you leave the FORMAT option out, the file type defaults to ASCII.

- 6 Execute the statement.

If the export is successful, the Messages pane displays the amount of time it took to export the query result set, the filename and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating the export was unsuccessful.

 For more information about exporting query results using the OUTPUT statement, see "OUTPUT statement [Interactive SQL]" on page 488 of the book *ASA SQL Reference Manual*.

**Tips**

You can combine the APPEND and VERBOSE statements to append both results and messages to an existing file. For example, type **OUTPUT TO 'c:\filename.SQL' APPEND VERBOSE**. For more information about APPEND and VERBOSE, see the "OUTPUT statement [Interactive SQL]" on page 488 of the book *ASA SQL Reference Manual*.

The OUTPUT TO, APPEND, and VERBOSE statements are equivalent to the >#, >>#, >&, and >>& operators of earlier versions of Interactive SQL. You can still use these operators to redirect data, but the new Interactive SQL statements allow for more precise output and easier to read code.

❖ **To export query results (UNLOAD statement):**

- 1 Execute an UNLOAD statement. For example,

```
UNLOAD
SELECT *
FROM employee;
TO 'c:\employee.dbf'
```

If the export is successful, the Messages pane displays the amount of time it took to export the query result set, the filename and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating the export was unsuccessful.

## Exporting a database

❖ **To unload all or part of a database (Sybase Central):**

- 1 From the Sybase Central window, click Utilities in the left pane.  
All of the functions you can perform on a database appear in the right pane.
- 2 Double-click Unload Database in the right pane.  
The Unload Database wizard appears.  
  
You can also open the Unload Database wizard by right-clicking on the database name in the left pane, and choosing Unload Database from the popup menu, or by choosing the Tools►Adaptive Server Anywhere 8►Unload Database command.
- 3 Follow the instructions in the wizard.

### ❖ To unload all or part of a database (command line):

- 1 At a command prompt, enter the *dbunload* command and specify connection parameters using the *-c* option.

For example, the following command unloads the entire database to *c:\temp*:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" c:\temp
```

- 2 If you want to export data only, add the *-d* option.

For example, if you want to export data only, your final command would look like this:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d c:\temp
```

- 3 If you want to export schema only, add the *-n* option instead.

For example, if you want to export schema only, your final command would look like this:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -n c:\temp
```

- 4 Press Enter to execute the command.

☞ For more information about additional command line options you can apply to the *dbunload* utility, see "Unloading a database using the *dbunload* command-line utility" on page 514 of the book *ASA Database Administration Guide*.

## Exporting tables

In addition to the methods described below, you can also export a table by selecting all the data in a table and exporting the query results. For more information, see "Exporting query results" on page 435.

### Tip

You can export views just as you would export tables.

### ❖ To export a table (Command line):

- 1 At a command prompt, enter the following *dbunload* command and press Enter:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t  
employee c:\temp
```



where `-c` specifies the database connection parameters and `-t` specifies the name of the table(s) you want to export. This *dbunload* command unloads the data from the sample database (assumed to be running on the default database server with the default database name) into a set of files in the *c:\temp* directory. A command file to rebuild the database from the data files is created with the default name *reload.SQL* in the current directory.

You can unload more than one table by separating the table names with a comma (,) delimiter.

❖ **To export a table (SQL):**

- 1 Execute an UNLOAD TABLE statement. For example,

```
UNLOAD TABLE department
TO 'dept.txt'
```

This statement unloads the *department* table from the sample database into the file *dept.txt* in the server's current working directory. If you are running against a network server, the command unloads the data into a file on the server machine, not the client machine. Also, the file name passes to the server as a string. Using escape backslash characters in the file name prevents misinterpretation if a directory of file name begins with an n (\n is a newline character) or any other special characters.

Each row of the table is output on a single line of the output file, and no column names are exported. The columns are separated, or delimited, by a comma. The delimiter character can be changed using the *DELIMITED BY* clause. The fields are not fixed-width fields. Only the characters in each entry are exported, not the full width of the column.

🔗 For more information about the UNLOAD TABLE statement syntax, see "UNLOAD TABLE statement" on page 573 of the book *ASA SQL Reference Manual*.

## Rebuilding databases

Rebuilding a database is a specific type of import and export involving unloading and reloading your entire database. Rebuilding your database takes all the information out of your database and puts it back in, in a uniform fashion, thus filling space and improving performance much like defragmenting your disk drive.

It is good practice to make backups of your database before rebuilding.

Loading and unloading are most useful for improving performance, reclaiming fragmented space, or upgrading your database to a newer version of Adaptive Server Anywhere.

Rebuilding is different from exporting in that rebuilding exports and imports table definitions and schema in addition to the data. The unload portion of the rebuild process produces ASCII format data files and a ' *reload.SQL* ' file which contains table and other definitions. Running the *reload.SQL* script recreates the tables and loads the data into them.

You can carry out this operation from Sybase Central or using the *dbunload* command line utility.

When unloading and reloading a database that has proxy tables, you must create an external login to map the local user to the remote user, even if the user has the same password on both the local and remote databases. If you do not have an external login, the reload may fail because you cannot connect to the remote server.

☞ For more information about external logins, see "Working with external logins" on page 465.

Consider rebuilding your database if you want to upgrade your database, reclaim disk space or improve performance. You might consider extracting a database (creating a new database from an old database) if you are using SQL Remote or MobiLink.

If you need to defragment your database, and a full rebuild is not possible due to requirements for continuous access to the database, consider reorganizing the table instead of rebuilding.

☞ For more information about reorganizing tables, see the "REORGANIZE TABLE statement" on page 508 of the book *ASA SQL Reference Manual*.

Rebuilding a  
database involved  
in replication

If a database is participating in replication, particular care needs to be taken if you wish to rebuild the database.

Replication is based on the offsets in the transaction log. When you rebuild a database, the offsets in the old transaction log are different than the offsets in the new log, making the old log unavailable. For this reason, good backup practices are especially important when participating in replication.

There are two ways of rebuilding a database involved in replication. The first method uses the *dbunload* utility *-ar* option to make the unload and reload occur in a way that does not interfere with replication. The second method is a manual method of accomplishing the same task.

The rebuild (load/unload) and extract procedures are used to rebuild databases and to create new databases from part of an old one.

With importing and exporting, the destination of the data is either into your database or out of your database. Importing reads data into your database. Exporting writes data out of your database. Often the information is either coming from or going to another non-Adaptive Server Anywhere database.

Rebuilding, however, combines two functions: loading and unloading. Loading and Unloading takes data and schema out of an Adaptive Anywhere database and then places the data and schema back into an Adaptive Server Anywhere database. The unloading procedure produces fixed format data files and a *reload.SQL* file which contains table definitions required to recreate the table exactly. Running the *reload.SQL* script recreates the tables and loads the data back into them.

Rebuilding a database can be a time consuming operation, and can require a large amount of disk space. As well, the database is unavailable for use while being unloaded and reloaded. For these reasons, rebuilding a database is not advised in a production environment unless you have a definite goal in mind.

#### **Rebuilding a database involved in replication**

The procedure for rebuilding a database depends on whether the database is involved in replication or not. If the database is involved in replication, you must preserve the transaction log offsets across the operation, as the Message Agent and Replication Agent require this information. If the database is not involved in replication, the process is simpler.

## **Rebuild tools**

### **LOAD/UNLOAD TABLE statement**

UNLOAD TABLE allows you to export data only, in an efficient manner in text/ASCII/FIXED formats. The UNLOAD TABLE statement exports with one row per line, with values separated by a comma delimiter. To make reloading quicker, the data exports in order by primary key values.

To use the UNLOAD TABLE statement, the user must have ALTER or SELECT permission on the table.

Choose the UNLOAD TABLE statement when you want to export data in text format or when performance is an issue.

For more information, see "UNLOAD statement" on page 571 of the book *ASA SQL Reference Manual*.

**dbunload/dbisql  
utilities and Sybase  
Central**

The *dbunload/dbisql* utilities and Sybase Central are graphically different, and functionally equivalent. You can use either one interchangeably to produce the same results.

You can use the Sybase Central Unload Database wizard or the *dbunload* utility to unload an entire database in ASCII comma-delimited format and to create the necessary Interactive SQL command files to completely recreate your database. This may be useful for creating SQL Remote extractions or building new copies of your database with the same or a slightly modified structure. The *dbunload* utility and Sybase Central are useful for exporting Adaptive Server Anywhere files intended for reuse within Adaptive Server Anywhere.

Choose Sybase Central or the *dbunload* utility when you want to rebuild your or extract from your database, export in text format, when you need to process large amounts of data quickly, or when your file format requirements are flexible.

For more information, see "Rebuilding a database not involved in replication" on page 444 and "Rebuilding a database involved in replication" on page 445.

## Rebuild file formats

**From one ASA  
database to  
another**

Rebuilding generally takes data out of an Adaptive Server Anywhere database and then places that data back into an Adaptive Server Anywhere database. The unloading and reloading are closely tied together since you usually perform both tasks, rather than just one or the other.

**Rebuilding a  
database**

You might rebuild your database if you wanted to:

- ◆ **Upgrade your database file format** Some new features are made available by applying the Upgrade utility, but others require a database file format upgrade, which is carried out by unloading and reloading the database. The New Features documentation will state if an unload and reload is required to obtain a particular feature.

- ◆ **Reclaim disk space** Databases do not shrink if you delete data. Instead, any empty pages are simply marked as free so they can be used again. They are not removed from the database unless you rebuild it. Rebuilding a database can reclaim disk space if you have deleted a large amount of data from your database and do not anticipate adding more.
- ◆ **Improve performance** Rebuilding databases can improve performance for the following reasons:
  - ◆ If data on pages within the database is fragmented, unloading and reloading can eliminate the fragmentation.
  - ◆ Since the data can be unloaded and reloaded in order by primary keys, access to related information can be faster, as related rows may appear on the same or adjacent pages.

### Upgrading a database

New versions of the Adaptive Server Anywhere database server can be used without upgrading your database. If you want to use features of the new version that require access to new system tables or database options, you must use the upgrade utility to upgrade your database. The upgrade utility does not unload or reload any data.

If you want to use features of the new version that rely on changes in the database file format, you must unload and reload your database. You should back up your database after rebuilding the database.

To upgrade your database file, use the new version of Adaptive Server Anywhere.

☞ For more information about upgrading your database, see "Upgrading Adaptive Server Anywhere" on page 142 of the book *What's New in SQL Anywhere Studio*.

## Exporting table data or table schema

### ❖ To export table data or table schema (Command line):

- 1 At a command prompt, enter the *dbunload* command and specify connection parameters using the *-c* option.
- 2 Specify the table(s) you want to export data or schema for, using the *-t* option.

For example, to export part of the employee table, enter

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -t
employee c:\temp
```

You can unload more than one table by separating the table names with a comma delimiter.

- 3 If you want to export data only, add the `-d` option.

For example, if you want to export data only, your final command would look like this:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -d -t  
employee c:\temp
```

- 4 If you want to export schema only, add the `-n` option instead.

For example, if you want to export schema only, your final command would look like this:

```
dbunload -c "dbn=asademo;uid=DBA;pwd=SQL" -n -t  
employee c:\temp
```

- 5 Press Enter to execute the command.

The *dbunload* commands in these examples unload the data or schema from the sample database table (assumed to be running on the default database server with the default database name) into a file in the *c:\temp* directory. A command file to rebuild the database from the data files is created with the default name *reload.SQL* in the current directory.

## Reloading a Database

### ❖ To reload a database (Command line):

- 1 At a command prompt, execute the *reload.SQL* script.

For example, the following command loads the *reload.SQL* script in the current directory.

```
dbisql -c "dbn=asademo;uid=DBA;pwd=SQL" reload.SQL
```

## Rebuilding a database not involved in replication

The following procedures should be used only if your database is not involved in replication.

☞ For instructions about rebuilding a database not involved in replication from Sybase Central, see "Upgrading the database file format" on page 144 of the book *What's New in SQL Anywhere Studio*.

❖ **To rebuild a database not involved in replication (Command line):**

- 1 At a command prompt, execute the *dbunload* command line utility using one of the following options:

- ◆ The *-an* option rebuilds to a new database.

```
dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -an
asademo.db
```

- ◆ The *-ac* option reloads to an existing database.

```
dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -ac
"uid=DBA;pwd=SQL;dbf=newdemo.db"
```

- ◆ The *-ar* option replaces the existing database.

```
dbunload -c "dbf=asademo.db;uid=DBA;pwd=SQL" -ar
"uid=DBA;pwd=SQL;dbf=newdemo.db"
```

If you use one these options, no interim copy of the data is created on disk, so you do not specify an unload directory on the command line. This provides greater security for your data. The *-ar* and *-an* options should also execute more quickly than Sybase Central, but *-ac* is slower.

- 2 Shut down the database and archive the transaction log, before using the reloaded database.

**Notes**

The *-an* and *-ar* options only apply to connections to a personal server, or connections to a network server over shared memory.

There are additional options available for the *dbunload* utility that allow you to tune the unload, as well as connection parameter options that allow you to specify a running or non-running database and database parameters.

**Rebuilding a database involved in replication**❖ **To rebuild a database involved in replication:**

- 1 Shut down the database.
- 2 Perform a full off-line backup by copying the database and transaction log files to a secure location.
- 3 At a command prompt, run *dbunload* to rebuild the database:

```
dbunload -c connection_string -ar directory
```

where *connection\_string* is a connection with DBA authority, *directory* is the directory used in your replication environment for old transaction logs, and there are no other connections to the database.

The `-ar` option only applies to connections to a personal server, or connections to a network server over shared memory.

🔗 For more information, see "Unload utility options" on page 516 of the book *ASA Database Administration Guide*.

- 4 Shut down the new database. Perform validity checks that you would usually perform after restoring a database.
- 5 Start the database using any production options you need. You can now allow user access to the reloaded database.

### Notes

There are additional options available for the *dbunload* utility that allow you to tune the unload, as well as connection parameter options that allow you to specify a running or non-running database and database parameters.

If the above procedure does not meet your needs, you can manually adjust the transaction log offsets. The following procedure describes how to carry out that operation.

#### ❖ To rebuild a database involved in replication, with manual intervention:

- 1 Shut down the database.
- 2 Perform a full off-line backup by copying the database and transaction log files to a secure location.
- 3 Run the *dbtran* utility to display the starting offset and ending offset of the database's current transaction log file. Note the ending offset for later use.
- 4 Rename the current transaction log file so that it is not modified during the unload process, and place this file in the *dbremote* off-line logs directory.
- 5 Rebuild the database.

🔗 For information on this step, see "Rebuilding databases" on page 440.

- 6 Shut down the new database.
- 7 Erase the current transaction log file for the new database.
- 8 Use *dblog* on the new database with the ending offset noted in step 3 as the `-z` parameter, and also set the relative offset to zero.

```
dblog -x 0 -z 137829 database-name.db
```

- 9 When you run the Message Agent, provide it with the location of the original off-line directory on its command line.



- 10 Start the database. You can now allow user access to the reloaded database.

## Minimizing downtime during rebuilding

The following steps help you rebuild a database while minimizing downtime. This can be especially useful if your database is in operation 24 hours a day.

It's wise to do a practice run of steps 1-4, and determine the times required for each step, prior to beginning the actual backup. You may also want to save copies of your files at various points during the rebuild.

Make sure that no other scheduled backups rename the production database's log. If this happens in error, you will need to apply the transactions from these renamed logs to the rebuilt database in the correct order.

### ❖ To rebuild a database and minimize the downtime:

- 1 Using DBBACKUP -r, create a backup of the database and log, and rename the log.
- 2 Rebuild the backed up database on another machine.
- 3 Do another DBBACKUP -r on the production server to rename the log.
- 4 Run DBTRAN on the log from step 3 and apply the transactions to the rebuilt server.

You now have a rebuilt database that contains all transactions up to the end of the backup in step 3.

- 5 Shut down the production server and make copies of the database and log.
- 6 Copy the rebuilt database onto the production server.
- 7 Run DBTRAN on the log from step 5.  
This should be a relatively small file.
- 8 Start the server on the rebuilt database, but don't allow users to connect.
- 9 Apply the transactions from step 8.
- 10 Allow users to connect.

## Extracting data

Extracting removes a remote Adaptive Server Anywhere database from a consolidated Adaptive Server Enterprise or Adaptive Server Anywhere database.

You can use the Sybase Central Extract Database wizard or the Extraction utility to extract databases. The Extraction utility is the recommended way of creating and synchronizing remote databases from a consolidated database.

For more information about how to perform database extractions, see:





- ◆ "The Database Extraction utility" on page 311 of the book *SQL Remote User's Guide*
- ◆ "Using the extraction utility" on page 193 of the book *SQL Remote User's Guide*
- ◆ "Extraction utility options" on page 314 of the book *SQL Remote User's Guide*
- ◆ "Extracting groups" on page 197 of the book *SQL Remote User's Guide*
- ◆ "Deploying remote databases" on page 148 of the book *MobiLink Synchronization User's Guide*
- ◆ "Extracting a remote database in Sybase Central" on page 311 of the book *SQL Remote User's Guide*

# Migrating databases to Adaptive Server Anywhere

You can import tables from remote Oracle, DB2, Microsoft SQL Server, Sybase Adaptive Server Enterprise, Adaptive Server Anywhere, and Microsoft Access databases into Adaptive Server Anywhere using the *sa\_migrate* set of stored procedures or the Data Migration wizard.

If you do not want to modify the tables in any way, you can use the single step method. Alternatively, if you would like to remove tables or foreign key mappings, you can use the extended method.

When using the *sa\_migrate* set of stored procedures, you must complete the following steps before you can import a remote database:

- ◆ Create a target database.  
 For information about creating a database, see "Creating a database" on page 29.
- ◆ Create a remote server to connect to the remote database.  
 For information about creating a remote server, see "Creating remote servers" on page 460.
- ◆ Create an external login to connect to the remote database. This is only required when the user has a different passwords on the target and remote databases, or when you want to login using a different user ID on the remote database than the one you are using on the target database.  
 For information about creating an external login, see "Creating external logins" on page 465.
- ◆ Create a local user who will own the migrated tables in the target database.  
 For information about creating a user, see "Creating new users" on page 357 of the book *ASA Database Administration Guide*.

If you use the Data Migration wizard, you can perform all these steps, except creating the target database, from the wizard in Sybase Central.

## ❖ To import remote tables (Sybase Central):


- 1 From Sybase Central, connect to the target database.
- 2 In the left pane of the Sybase Central window, click Utilities.  
All the functions you can perform on a database appear in the right pane.
- 3 Double-click Migrate Database in the right pane.

The Database Migration wizard appears.

You can also open the Database Migration wizard by right-clicking on the target database in the left pane, and choosing Migrate Database from the popup menu, or by choosing the Tools►Adaptive Server Anywhere 8►Migrate Database command.


- 4 Select the target database from the list, and then click Next.
- 5 Select the remote server you want to use to connect to the remote database from which you want to migrate data, and then click Next.

If you have not already created a remote server, click Create Remote Server Now to open the Remote Server Creation wizard.

 For more information about creating a remote server, see "Creating remote servers" on page 460.

- 6 Select the tables that you want to migrate, and then click Next.
- 7 Select whether you want to migrate the data and the foreign keys from the remote tables and whether you want to keep the proxy tables that are created for the migration process, and then click Next.

If the target database is version 8.0.0 or earlier, the Migrate the foreign keys option is not enabled. You must upgrade the database to version 8.0.1 or later to use this option.

 For more information about upgrading, see "Upgrading a database" on page 143 of the book *What's New in SQL Anywhere Studio*.

- 8 Select the user that will own the tables on the target database, and then click Finish to complete the migration.

If you have not already created a user, click Create User Now to open the User Creation wizard.

- 9 Click Finish.

The following example uses the *sa\_migrate* stored procedure to import all the tables that belong to one owner on the remote database in one step.

Supplying NULL for both the *table\_name* and *owner\_name* parameters migrates all the tables in the database, including system tables. As well, tables that have the same name, but different owners, in the remote database all belong to one owner in the target database. For these reasons, it is recommended that you migrate tables associated with one owner at a time.

#### ❖ To import remote tables (single step):


- 1 From Interactive SQL, connect to the target database.

- 2 In the Interactive SQL Statements pane, run the *sa\_migrate* stored procedure. For example,


```
CALL sa_migrate( 'local_a', 'ase', NULL, l_smith,  
NULL, 1, 1, 1 )
```

This procedure calls several procedures in turn and migrates all the remote tables belonging to the user *l\_smith* using the specified criteria.

If you do not want all the migrated tables to be owned by the same user on the target database, you must run the *sa\_migrate* procedure for each owner on the target database, specifying the *local\_table\_owner* and *owner\_name* arguments. It is recommended that you migrate tables associated with one owner at a time.

 For more information, see "sa\_migrate system procedure" on page 701 of the book *ASA SQL Reference Manual*.

For target databases that are version 8.0.0 or earlier, foreign keys are migrated automatically. If you do not want to migrate the foreign keys, you must upgrade the database file format to version 8.0.1 or later.

 For more information about upgrading, see "Upgrading a database" on page 143 of the book *What's New in SQL Anywhere Studio*.

### ❖ To import remote tables (with modifications):

- 1 From Interactive SQL, connect to the target database.
- 2 Run the *sa\_migrate\_create\_remote\_table\_list* stored procedure. For example,

```
CALL sa_migrate_create_remote_table_list( 'ase',  
'NULL', 'remote_a', 'mydb' )
```

You must specify a database name for Adaptive Server Enterprise and Microsoft SQL Server databases.

This populates the *dbo.migrate\_remote\_table\_list* table with a list of remote tables to migrate. You can delete rows from this table for remote tables you do not wish to migrate.

Do not supply NULL for both the *table\_name* and *owner\_name* parameters. Doing so migrates all the tables in the database, including system tables. As well, tables that have the same name but different owners in the remote database all belong to one owner in the target database. It is recommended that you migrate tables associated with one owner at a time.

☞ For more information about the *sa\_migrate\_create\_remote\_table\_list* stored procedure, see "sa\_migrate\_create\_remote\_table\_list system procedure" on page 706 of the book *ASA SQL Reference Manual*.

- 3 Run the *sa\_migrate\_create\_tables* stored procedure. For example,

```
CALL sa_migrate_create_tables( 'local_a', )
```

This procedure takes the list of remote tables from *dbo.migrate\_remote\_table\_list* and creates a proxy table and a base table for each remote table listed. This procedure also creates all primary key indexes for the migrated tables.

☞ For more information about the *sa\_migrate\_create\_tables* stored procedure, see "sa\_migrate\_create\_tables system procedure" on page 708 of the book *ASA SQL Reference Manual*.

- 4 If you want to migrate the data from the remote tables into the base tables on the target database, run the *sa\_migrate\_data* stored procedure. For example,

Enter the following stored procedure:

```
CALL sa_migrate_data( 'local_a' )
```

This procedure migrates the data from each remote table into the base table created by the *sa\_migrate\_create\_tables* procedure.

☞ For more information about the *sa\_migrate\_data* stored procedure, see "sa\_migrate\_data system procedure" on page 709 of the book *ASA SQL Reference Manual*.

If you do not want to migrate the foreign keys from the remote database, you can skip to step 7.

- 5 Run the *sa\_migrate\_create\_remote\_fks\_list* stored procedure. For example,

```
CALL sa_migrate_create_remote_fks_list( 'ase' )
```

This procedure populates the table *dbo.migrate\_remote\_fks\_list* with the list of foreign keys associated with each of the remote tables listed in *dbo.migrate\_remote\_table\_list*.

You can remove any foreign key mappings you do not want to recreate on the local base tables.

☞ For more information about the *sa\_migrate\_create\_remote\_fks\_list* stored procedure, see "sa\_migrate\_create\_remote\_fks\_list system procedure" on page 705 of the book *ASA SQL Reference Manual*.

- 6 Run the *sa\_migrate\_create\_fks* stored procedure. For example,

```
CALL sa_migrate_create_fks( 'local_a' )
```

This procedure creates the foreign key mappings defined in *dbo.migrate\_remote\_fks\_list* on the base tables.

For more information about the *sa\_migrate\_create\_fks* stored procedure, see "sa\_migrate\_create\_fks system procedure" on page 703 of the book *ASA SQL Reference Manual*.

- 7 If you want to drop the proxy tables that were created for migration purposes, run the *sa\_drop\_proxy\_tables* stored procedure. For example,

```
CALL sa_migrate_drop_proxy_tables( 'local_a' )
```

This procedure drops all proxy tables created for migration purposes and completes the migration process.

For more information about the *sa\_migrate\_drop\_proxy\_tables* stored procedure, see "sa\_migrate\_drop\_proxy\_tables system procedure" on page 710 of the book *ASA SQL Reference Manual*.

## Adaptive Server Enterprise compatibility

You can import and export files between Adaptive Server Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause. Simply make sure the BCP output is in delimited ASCII format. If you are exporting BLOB data from Adaptive Server Anywhere for use in Adaptive Server Enterprise, use the BCP format clause with the UNLOAD TABLE statement.

☞ For more information about BCP and the FORMAT clause, see "LOAD TABLE statement" on page 472 of the book *ASA SQL Reference Manual* or "UNLOAD TABLE statement" on page 573 of the book *ASA SQL Reference Manual*.



# Accessing Remote Data

About this chapter      Adaptive Server Anywhere can access data located on different servers, both Sybase and non-Sybase, as if the data were stored on the local server.

                                 This chapter describes how to configure Adaptive Server Anywhere to access remote data.

Contents

Topic	Page
Introduction	456
Basic concepts to access remote data	458
Working with remote servers	460
Working with external logins	465
Working with proxy tables	467
Joining remote tables	472
Joining tables from multiple local databases	474
Sending native statements to remote servers	475
Using remote procedure calls (RPCs)	476
Transaction management and remote data	479
Internal operations	481
Troubleshooting remote data access	485

## Introduction

Using Adaptive Server Anywhere you can:

- ◆ Access data in relational databases such as Sybase, Oracle, and DB2.
- ◆ Access desktop data such as Excel spreadsheets, MS-Access databases, FoxPro, and text files.
- ◆ Access any other data source that supports an ODBC interface.
- ◆ Perform joins between local and remote data.
- ◆ Perform joins between tables in separate Adaptive Server Anywhere databases.
- ◆ Use Adaptive Server Anywhere features on data sources that would normally not have that ability. For instance, you could use a Java function against data stored in Oracle, or perform a subquery on spreadsheets. Adaptive Server Anywhere will compensate for features not supported by a remote data source by operating on the data after it is retrieved.
- ◆ Use Adaptive Server Anywhere to move data from one location to another using insert-select.
- ◆ Access remote servers directly using passthrough mode.
- ◆ Execute remote procedure calls to other servers.

Adaptive Server Anywhere allows access to the following external data sources:

- ◆ Adaptive Server Anywhere
- ◆ Adaptive Server Enterprise
- ◆ Oracle
- ◆ IBM DB2
- ◆ Microsoft SQL Server
- ◆ Other ODBC data sources

☞ For platform availability, see "Adaptive Server Anywhere supported operating systems" on page 138 of the book *Introducing SQL Anywhere Studio*.

## Accessing remote data from PowerBuilder DataWindows

You can access remote data from a PowerBuilder DataWindow by setting the DBParm Block parameter to 1 on connect.

- ◆ In the design environment, you can set the Block parameter by accessing the Transaction tab in the Database Profile Setup dialog and setting the Retrieve Blocking Factor to 1.
- ◆ In a connection string, use the following phrase:

`DBParm="Block=1"`

## Basic concepts to access remote data

This section describes the basic concepts required to access remote data.


### Remote table mappings

Adaptive Server Anywhere presents tables to a client application as if all the data in the tables were stored in the database to which the application is connected. Internally, when a query involving remote tables is executed, the storage location is determined, and the remote location is accessed so that data can be retrieved.


To have remote tables appear as local tables to the client, you create local proxy tables that map to the remote data.

#### ❖ To create a proxy table:

- 1 Define the server where the remote data is located. This specifies the type of server and location of the remote server.

 For more information, see "Working with remote servers" on page 460.

- 2 Map the local user login information to the remote server user login information if the logins on the two servers are different.

 For more information, see "Working with external logins" on page 465.

- 3 Create the proxy table definition. This specifies the mapping of a local proxy table to the remote table. This includes the server where the remote table is located, the database name, owner name, table name, and column names of the remote table.

 For more information, see "Working with proxy tables" on page 467.

Administering  
remote table  
mappings

To manage remote table mappings and remote server definitions, you can use Sybase Central or you can use a tool such as Interactive SQL and execute the SQL statements directly.

## Server classes

A **server class** is assigned to each remote server. The server class specifies the access method used to interact with the server. Different types of remote servers require different access methods. The server classes provide Adaptive Server Anywhere detailed server capability information. Adaptive Server Anywhere adjusts its interaction with the remote server based on those capabilities.

There are currently two groups of server classes. The first is JDBC-based; the second is ODBC-based.

The JDBC-based server classes are:

- ◆ **asajdbc** for Adaptive Server Anywhere (version 6 and later)
- ◆ **asejdbc** for Adaptive Server Enterprise and SQL Server (version 10 and later)

The ODBC-based server classes are:

- ◆ **asaodbc** for Adaptive Server Anywhere (version 5.5 and later)
- ◆ **aseodbc** for Adaptive Server Enterprise and SQL Server (version 10 and later)
- ◆ **db2odbc** for IBM DB2
- ◆ **mssodbc** for Microsoft SQL Server
- ◆ **oraodbc** for Oracle servers (version 8.0 and later)
- ◆ **odbc** for all other ODBC data sources

🔗 For a full description of remote server classes, see "Server Classes for Remote Data Access" on page 487.

## Working with remote servers

Before you can map remote objects to a local proxy table, you must define the remote server where the remote object is located. When you define a remote server, an entry is added to the *sys.servers* table for the remote server. This section describes how to create, alter, and delete a remote server definition.

### Creating remote servers

Use the `CREATE SERVER` statement to set up remote server definitions. You can execute the statements directly, or use Sybase Central.

For ODBC connections, each remote server corresponds to an ODBC data source. For some systems, including Adaptive Server Anywhere, each data source describes a database, so a separate remote server definition is needed for each database.

You must have `RESOURCE` authority to create a server.

✍ For a full description of the `CREATE SERVER` statement, see "CREATE SERVER statement" on page 321 of the book *ASA SQL Reference Manual*.

#### Example 1

The following statement creates an entry in the *sys.servers* table for the Adaptive Server Enterprise server called *ASEserver*:

```
CREATE SERVER ASEserver
CLASS 'ASEJDBC'
USING 'rimu:6666'
```

where:

- ◆ **ASEserver** is the name of the remote server
- ◆ **ASEJDBC** is a keyword indicating that the server is Adaptive Server Enterprise and the connection to it is JDBC-based
- ◆ **rimu:6666** is the machine name and the TCP/IP port number where the remote server is located

#### Example 2

The following statement creates an entry in the *sys.servers* table for the ODBC-based Adaptive Server Anywhere server named *testasa*:

```
CREATE SERVER testasa
CLASS 'ASAODBC'
USING 'test4'
```

where:

- ◆ **testasa** is the name by which the remote server is known within this database.
- ◆ **ASAODBC** is a keyword indicating that the server is Adaptive Server Anywhere and the connection to it uses ODBC.
- ◆ **test4** is the ODBC data source name.

## Creating remote servers using Sybase Central

You can create a remote server using a wizard in Sybase Central. For more information, see "Creating remote servers" on page 460.

### ❖ To create a remote server (Sybase Central):

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder for that database.
- 3 Double-click Add Remote Server.
- 4 On the first page of the wizard, enter a name to use for the remote server. This name simply refers to the remote server from within the local database; it does not need to correspond with the name the server supplies. Click Next.
- 5 Select an appropriate type of server and click Next.
- 6 Select a data access method (ODBC or JDBC), and supply connection information.
  - ◆ For ODBC, supply a data source name.
  - ◆ For JDBC, supply a URL in the form *machine-name:port-number*The data access method (JDBC or ODBC) is the method used by Adaptive Server Anywhere to access the remote database. This is not related to the method used by Sybase Central to connect to your database.
- 7 Click Next. Specify whether you want the remote server to be read-only.
- 8 Click Finish to create the remote server definition.

## Deleting remote servers

You can use Sybase Central or a DROP SERVER statement to delete a remote server from the Adaptive Server Anywhere system tables. All remote tables defined on that server must already be dropped for this action to succeed.

You must have DBA authority to delete a remote server.

❖ **To delete a remote server (Sybase Central):**

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder.
- 3 Right-click the remote server you want to delete and choose Delete from the popup menu.

❖ **To delete a remote server (SQL):**

- 1 Connect to the host database from Interactive SQL.
- 2 Execute a DROP SERVER statement.

🔗 For more information, see "DROP SERVER statement" on page 404 of the book *ASA SQL Reference Manual*.

**Example**

The following statement drops the server named testasa:

```
DROP SERVER testasa
```

## Altering remote servers

You can use Sybase Central or an ALTER SERVER statement to modify the attributes of a server. These changes do not take effect until the next connection to the remote server.

You must have RESOURCE authority to alter a server.

❖ **To alter the properties of a remote server (Sybase Central):**

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder for that database.
- 3 Right-click the remote server and choose Properties from the popup menu.
- 4 Configure the various remote server properties.

❖ **To alter the properties of a remote server (SQL):**

- 1 Connect to the host database from Interactive SQL.
- 2 Execute an ALTER SERVER statement.



**Example**

The following statement changes the server class of the server named ASEserver to aseodbc. In this example, the Data Source Name for the server is ASEserver.

```
ALTER SERVER ASEserver
CLASS 'aseodbc'
```

The ALTER SERVER statement can also be used to enable or disable a server's known capabilities.

☞ For more information, see "ALTER SERVER statement" on page 220 of the book *ASA SQL Reference Manual*.

**Listing the remote tables on a server**

It may be helpful when you are configuring Adaptive Server Anywhere to get a list of the remote tables available on a particular server. The *sp\_remote\_tables* procedure returns a list of the tables on a server.

```
sp_remote_tables servername
                  [ , tablename]
                  [ , owner ]
                  [ , database]
```

If *tablename*, *owner*, or *database* is given, the list of tables is limited to only those that match.

For example, to get a list of all of the Microsoft Excel worksheets available from an ODBC data source named *excel*:

```
sp_remote_tables excel
```

Or to get a list of all of the tables in the *production* database in an ASE named *asetest*, owned by 'fred':

```
sp_remote_tables asetest, null, fred, production
```

☞ For more information, see "sp\_remote\_tables system procedure" on page 724 of the book *ASA SQL Reference Manual*.


**Listing remote server capabilities**

The *sp\_servercaps* procedure displays information about a remote server's capabilities. Adaptive Server Anywhere uses this capability information to determine how much of a SQL statement can be passed off to a remote server.

The system tables which contain server capabilities are not populated until after Adaptive Server Anywhere first connects to the remote server. This information comes from the SYSCAPABILITY and SYSCAPABILITYNAME system tables. The servername specified must be the same servername used in the CREATE SERVER statement.


Issue the stored procedure *sp\_servercaps* as follows:

```
sp_servercaps servername
```

 For more information, see "sp\_servercaps system procedure" on page 725 of the book *ASA SQL Reference Manual*.

## Working with external logins

By default, Adaptive Server Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords to be used when communicating with a remote server.

 For more information, see "Using integrated logins" on page 83 of the book *ASA Database Administration Guide*.

### Creating external logins

You can create an external login using either Sybase Central or the CREATE EXTERNLOGIN statement.

Only the login-name and the DBA account can add or modify an external login.

#### ❖ To create an external login (Sybase Central):

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder for that database and select the remote server.
- 3 Right-click the remote server and choose Properties from the popup menu.
- 4 On the External Logins tab of the property sheet, click New and configure the settings in the resulting dialog.
- 5 Click OK to save the changes.

#### ❖ To create an external login (SQL):

- 1 Connect to the host database from Interactive SQL.
- 2 Execute a CREATE EXTERNLOGIN statement.

#### Example

The following statement allows the local user **fred** to gain access to the server **ASEserver**, using the remote login **frederick** with password **banana**.

```
CREATE EXTERNLOGIN fred
TO ASEserver
REMOTE LOGIN frederick
IDENTIFIED BY banana
```

🔗 For more information, see "CREATE EXTERNLOGIN statement" on page 294 of the book *ASA SQL Reference Manual*.

## Dropping external logins

You can use either Sybase Central or a DROP EXTERNLOGIN statement to delete an external login from the Adaptive Server Anywhere system tables.

Only the login-name and the DBA account can delete an external login.

### ❖ To delete an external login (Sybase Central):

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder.
- 3 Right-click the remote server and choose Delete from the popup menu.

### ❖ To delete an external login (SQL):

- 1 Connect to the host database from Interactive SQL.
- 2 Execute a DROP EXTERNLOGIN statement.

### Example

The following statement drops the external login for the local user fred created in the example above:

```
DROP EXTERNLOGIN fred TO ASEserver
```

🔗 See also

- ◆ "DROP EXTERNLOGIN statement" on page 401 of the book *ASA SQL Reference Manual*

## Working with proxy tables

Location transparency of remote data is enabled by creating a local **proxy table** that maps to the remote object. Use one of the following statements to create a proxy table:

- ◆ If the table already exists at the remote storage location, use the `CREATE EXISTING TABLE` statement. This statement defines the proxy table for an existing table on the remote server.
- ◆ If the table does not exist at the remote storage location, use the `CREATE TABLE` statement. This statement creates a new table on the remote server, and also defines the proxy table for that table.

## Specifying proxy table locations

The `AT` keyword is used with both `CREATE TABLE` and `CREATE EXISTING TABLE` to define the location of an existing object. This location string has four components, each separated by either a period or a semicolon. The semicolon delimiter allows filenames and extensions to be used in the database and owner fields.

The syntax of the `AT` clause is

```
... AT 'server.database.owner.table-name'
```

- ◆ **Server** This is the name by which the server is known in the current database, as specified in the `CREATE SERVER` statement. This field is mandatory for all remote data sources.
- ◆ **Database** The meaning of the database field depends on the data source. In some cases this field does not apply and should be left empty. The periods are still required, however.

In Adaptive Server Enterprise, *database* specifies the database where the table exists. For example *master* or *pubs2*.

In Adaptive Server Anywhere, this field does not apply; leave it empty.

In Excel, Lotus Notes, and Access, you must include the name of the file containing the table. If the file name includes a period, use the semicolon delimiter.

- ◆ **Owner** If the database supports the concept of ownership, this field represents the owner name. This field is only required when several owners have tables with the same name.

- ◆ **Table-name** This specifies the name of the table. In the case of an Excel spreadsheet, this is the name of the "sheet" in the workbook. If the table name is left empty, the remote table name is assumed to be the same as the local proxy table name.

Examples:

The following examples illustrate the use of location strings:

- ◆ Adaptive Server Anywhere:  
`'testasa..DBA.employee'`
- ◆ Adaptive Server Enterprise:  
`'ASEServer.pubs2.dbo.publishers'`
- ◆ Excel:  
`'excel;d:\pcdb\quarter3.xls;;sheet1$'`
- ◆ Access:  
`'access;\\server1\production\inventory.mdb;parts'`

## Creating proxy tables (Sybase Central)

You can create a proxy table using either Sybase Central or a CREATE EXISTING TABLE statement.

The CREATE EXISTING TABLE statement creates a proxy table that maps to an existing table on the remote server. Adaptive Server Anywhere derives the column attributes and index information from the object at the remote location.

### ❖ To create a proxy table (Sybase Central):

- 1 Connect to the host database from Sybase Central.
- 2 Do one of the following:
  - ◆ In the Tables folder, double-click Add Proxy Table.
  - ◆ In the Remote Servers folder, right-click a remote server and choose Add Proxy Table from the popup menu.
  - ◆ Select the Tables folder and then choose File►New►Proxy Table.
- 3 Follow the instructions in the wizard.

**Tip**

Proxy tables are displayed under their remote server, inside the Remote Servers folder. They also appear in the Tables folder. They are distinguished from other tables by a letter P on their icon.

## Creating proxy tables with the CREATE EXISTING TABLE statement

The CREATE EXISTING TABLE statement creates a proxy table that maps to an existing table on the remote server. Adaptive Server Anywhere derives the column attributes and index information from the object at the remote location.

❖ **To create a proxy table with the CREATE EXISTING TABLE statement (SQL):**

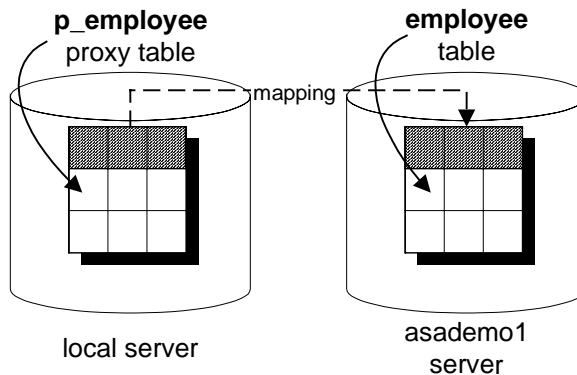
- 1 Connect to the host database.
- 2 Execute a CREATE EXISTING TABLE statement.

🔗 For more information, see the "CREATE EXISTING TABLE statement" on page 291 of the book *ASA SQL Reference Manual*.

### Example 1

To create a proxy table called *p\_employee* on the current server to a remote table named *employee* on the server named *asademo1*, use the following syntax:

```
CREATE EXISTING TABLE p_employee
AT 'asademo1..DBA.employee'
```



Example 2

The following statement maps the proxy table *a1* to the Microsoft Access file *mydbfile.mdb*. In this example, the *AT* keyword uses the semicolon (;) as a delimiter. The server defined for Microsoft Access is named *access*.

```
CREATE EXISTING TABLE a1
AT 'access;d:\mydbfile.mdb;a1'
```

## Creating a proxy table with the CREATE TABLE statement

The *CREATE TABLE* statement creates a new table on the remote server, and defines the proxy table for that table when you use the *AT* option. You enter the *CREATE TABLE* statement using Adaptive Server Anywhere data types. Adaptive Server Anywhere automatically converts the data into the remote server's native types.

If you use the *CREATE TABLE* statement to create both a local and remote table, and then subsequently use the *DROP TABLE* statement to drop the proxy table, then the remote table also gets dropped. You can, however, use the *DROP TABLE* statement to drop a proxy table created using the *CREATE EXISTING TABLE* statement if you do not want to drop the remote table.

☞ For more information, see the "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

❖ **To create a proxy table with the CREATE EXISTING TABLE statement (SQL):**

- 1 Connect to the host database.
- 2 Execute a *CREATE TABLE* statement.

☞ For more information, see the "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

Example

The following statement creates a table named *employee* on the remote server *asademo1*, and creates a proxy table named *members* that maps to the remote location:

```
CREATE TABLE members
( membership_id INTEGER NOT NULL,
  member_name CHAR(30) NOT NULL,
  office_held CHAR( 20 ) NULL)
AT 'asademo1..DBA.employee'
```



## Listing the columns on a remote table

If you are entering a CREATE EXISTING statement and you are specifying a column list, it may be helpful to get a list of the columns that are available on a remote table. The *sp\_remote\_columns* system procedure produces a list of the columns on a remote table and a description of those data types.

```
sp_remote_columns servername, tablename [, owner ]  
[, database]
```

If a table name, owner, or database name is given, the list of columns is limited to only those that match.

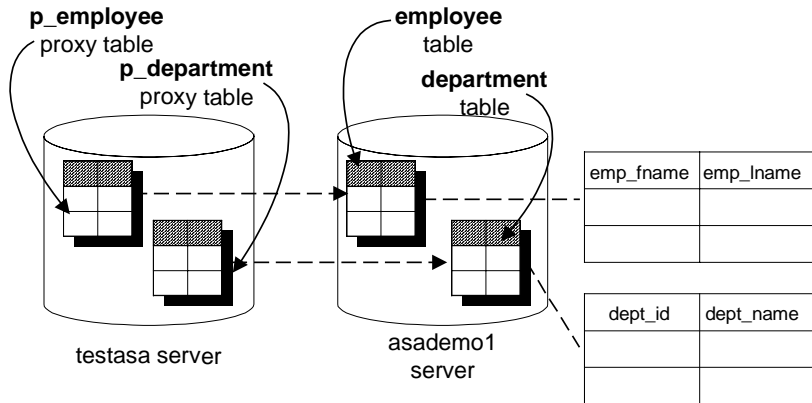
For example, the following returns a list of the columns in the *sysobjects* table in the *production* database on an Adaptive Server Enterprise server named *asetest*:

```
sp_remote_columns asetest, sysobjects, null, production
```

☞ For more information, see "sp\_remote\_columns system procedure" on page 721 of the book *ASA SQL Reference Manual*.

## Joining remote tables

The following figure illustrates the remote Adaptive Server Anywhere tables *employee* and *department* in the sample database mapped to the local server named *testasa*.



In real-world cases, you may use joins between tables on different Adaptive Server Anywhere databases. Here we describe a simple case using just one database to illustrate the principles.

### ❖ To perform a join between two remote tables (SQL):

- 1 Create a new database named *empty.db*.

This database holds no data. We will use it only to define the remote objects, and access the sample database from it.

- 2 Start a database server running both *empty.db* and the sample database. You can do this using the following command line, executed from the installation directory:

```
dbeng8 asademo empty
```

- 3 Connect to *empty.db* from Interactive SQL using the user ID **DBA** and the password **SQL**.
- 4 In the new database, create a remote server named *testasa*. Its server class is *asaodbc*, and the connection information is '**ASA 8.0 Sample**':

```
CREATE SERVER testasa
CLASS 'asaodbc'
USING 'ASA 8.0 Sample'
```

- 5 In this example, we use the same user ID and password on the remote database as on the local database, so no external logins are needed.

- 6 Define the *employee* proxy table:

```
CREATE EXISTING TABLE employee
AT 'testasa..DBA.employee'
```

- 7 Define the *department* proxy table:

```
CREATE EXISTING TABLE department
AT 'testasa..DBA.department'
```

- 8 Use the proxy tables in the SELECT statement to perform the join.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
ON employee.dept_id = department.dept_id
ORDER BY emp_lname
```

## Joining tables from multiple local databases

An Adaptive Server Anywhere server may have several local databases running at one time. By defining tables in other local Adaptive Server Anywhere databases as remote tables, you can perform cross database joins.

✍ For more information about specifying multiple databases, see "USING parameter value in the CREATE SERVER statement" on page 490.

### Example

For example, if you are using database *db1* and you want to access data in tables in database *db2*, you need to set up proxy table definitions that point to the tables in database *db2*. For instance, on an Adaptive Server Anywhere named *testasa*, you might have three databases available, *db1*, *db2*, and *db3*.

- ◆ If using ODBC, create an ODBC data source name for each database you will be accessing.
- ◆ Connect to one of the databases that you will be performing joins from. For example, connect to *db1*.
- ◆ Perform a CREATE SERVER for each other local database you will be accessing. This sets up a **loopback** connection to your Adaptive Server Anywhere server.

```
CREATE SERVER local_db2
CLASS 'asaodbc'
USING 'testasa_db2'

CREATE SERVER local_db3
CLASS 'asaodbc'
USING 'testasa_db3'
```

Alternatively, using JDBC:

```
CREATE SERVER local_db2
CLASS 'asajdbc'
USING 'mypc1:2638/db2'

CREATE SERVER local_db3
CLASS 'asajdbc'
USING 'mypc1:2638/db3'
```

- ◆ Create proxy table definitions using CREATE EXISTING to the tables in the other databases you want to access.

```
CREATE EXISTING TABLE employee
AT 'local_db2...employee'
```

## Sending native statements to remote servers

Use the FORWARD TO statement to send one or more statements to the remote server in its native syntax. This statement can be used in two ways:

- ◆ To send a statement to a remote server.
- ◆ To place Adaptive Server Anywhere into passthrough mode for sending a series of statements to a remote server.

If a connection cannot be made to the specified server, a message is sent to the user explaining why. If a connection is made, any results are converted into a form that can be recognized by the client program.

The FORWARD TO statement can be used to verify that a server is configured correctly. If you send a statement to the remote server and Adaptive Server Anywhere does not return an error message, the remote server is configured correctly.

☞ For more information, see "FORWARD TO statement" on page 431 of the book *ASA SQL Reference Manual*.

### Example 1

The following statement verifies connectivity to the server named *ASEserver* by selecting the version string:

```
FORWARD TO ASEserver {SELECT @@version}
```

### Example 2

The following statements show a passthrough session with the server named *ASEserver*:

```
FORWARD TO ASEserver
select * from titles
select * from authors
FORWARD TO
```

## Using remote procedure calls (RPCs)

Adaptive Server Anywhere users can issue procedure calls to remote servers that support the feature.

This feature is supported by Sybase Adaptive Server Anywhere, Sybase Adaptive Server Enterprise, Oracle, and DB2. Issuing a remote procedure call is similar to using a local procedure call.

### Creating remote procedures

You can issue a remote procedure call using either Sybase Central or the CREATE PROCEDURE statement.

You must have DBA authority to create a remote procedure.

#### ❖ To issue a remote procedure call (Sybase Central):

- 1 Connect to the host database from Sybase Central.
- 2 Open the Remote Servers folder.
- 3 Right-click the remote server for which you want to create a remote procedure and choose Properties from the File menu.
- 4 On the Remote Procedures tab, click New and follow the instructions in the wizard.

#### **Tip**

You can also add a remote procedure by right-clicking the remote server and choosing Add Remote Procedure from the popup menu.

#### ❖ To issue a remote procedure call (SQL):

- 1 First define the procedure to Adaptive Server Anywhere.

The syntax is the same as a local procedure definition except instead of using SQL statements to make up the body of the call, a location string is given defining the location where the procedure resides.

```
CREATE PROCEDURE remotewho ()  
AT 'bostonase.master.dbo.sp_who'
```

- 2 Execute the procedure as follows:

```
call remotewho()
```

For more information, see "CREATE PROCEDURE statement" on page 305 of the book *ASA SQL Reference Manual*.

### Example

Here is an example with a parameter:

```
CREATE PROCEDURE remoteuser (IN uname char(30))
AT 'bostonase.master.dbo.sp_helpuser'

call remoteuser('joe')
```

### Data types for remote procedures

The following data types are allowed for RPC parameters. Other data types are disallowed:

- ◆ [ UNSIGNED ] SMALLINT
- ◆ [ UNSIGNED ] INT
- ◆ [ UNSIGNED ] BIGINT
- ◆ TINYINT
- ◆ REAL
- ◆ DOUBLE
- ◆ CHAR
- ◆ BIT

NUMERIC and DECIMAL data types are allowed for IN parameters, but not for OUT or INOUT parameters.

## Dropping remote procedures

You can delete a remote procedure using either Sybase Central or the DROP PROCEDURE statement.

You must have DBA authority to delete a remote procedure.

### ❖ To delete a remote procedure (Sybase Central):

- 1 Open the Remote Servers folder.
- 2 Right-click the remote server and choose Properties from the File menu.
- 3 On the Remote Procedures tab, select the remote procedure and click Delete.

### ❖ To delete a remote procedure (SQL):

- 1 Connect to a database.
- 2 Execute a DROP PROCEDURE statement.

🔗 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

**Example**

Delete a remote procedure called *remoteproc*.


```
DROP PROCEDURE remoteproc
```



# Transaction management and remote data

Transactions provide a way to group SQL statements so that they are treated as a unit—either all work performed by the statements is committed to the database, or none of it is.

For the most part, transaction management with remote tables is the same as transaction management for local tables in Adaptive Server Anywhere, but there are some differences. They are discussed in the following section.

 For a general discussion of transactions, see "Using Transactions and Isolation Levels" on page 89.

## Remote transaction management overview

The method for managing transactions involving remote servers uses a two-phase commit protocol. Adaptive Server Anywhere implements a strategy that ensures transaction integrity for most scenarios. However, when more than one remote server is invoked in a transaction, there is still a chance that a distributed unit of work will be left in an undetermined state. Even though two-phase commit protocol is used, no recovery process is included.

The general logic for managing a user transaction is as follows:

- 1 Adaptive Server Anywhere prefaces work to a remote server with a `BEGIN TRANSACTION` notification.
- 2 When the transaction is ready to be committed, Adaptive Server Anywhere sends a `PREPARE TRANSACTION` notification to each remote server that has been part of the transaction. This ensures the that remote server is ready to commit the transaction.
- 3 If a `PREPARE TRANSACTION` request fails, all remote servers are told to roll back the current transaction.

If all `PREPARE TRANSACTION` requests are successful, the server sends a `COMMIT TRANSACTION` request to each remote server involved with the transaction.

Any statement preceded by `BEGIN TRANSACTION` can begin a transaction. Other statements are sent to a remote server to be executed as a single, remote unit of work.

## Restrictions on transaction management

Restrictions on transaction management are as follows:

- ◆ Savepoints are not propagated to remote servers.
- ◆ If nested BEGIN TRANSACTION and COMMIT TRANSACTION statements are included in a transaction that involves remote servers, only the outermost set of statements is processed. The innermost set, containing the BEGIN TRANSACTION and COMMIT TRANSACTION statements, is not transmitted to remote servers.

## Internal operations

This section describes the underlying operations on remote servers performed by Adaptive Server Anywhere on behalf of client applications.

### Query parsing

When a statement is received from a client, it is parsed. An error is raised if the statement is not a valid Adaptive Server Anywhere SQL statement.

### Query normalization

The next step is called query normalization. During this step, referenced objects are verified and some data type compatibility is checked.

For example, consider the following query:

```
SELECT *  
FROM t1  
WHERE c1 = 10
```

The query normalization stage verifies that table *t1* with a column *c1* exists in the system tables. It also verifies that the data type of column *c1* is compatible with the value 10. If the column's data type is datetime, for example, this statement is rejected.

### Query preprocessing

Query preprocessing prepares the query for optimization. It may change the representation of a statement so that the SQL statement Adaptive Server Anywhere generates for passing to a remote server will be syntactically different from the original statement.

Preprocessing performs view expansion so that a query can operate on tables referenced by the view. Expressions may be reordered and subqueries may be transformed to improve processing efficiency. For example, some subqueries may be converted into joins.

### Server capabilities

The previous steps are performed on all queries, both local and remote.

The following steps depend on the type of SQL statement and the capabilities of the remote servers involved.

Each remote server defined to Adaptive Server Anywhere has a set of capabilities associated with it. These capabilities are stored in the *syscapabilities* system table. These capabilities are initialized during the first connection to a remote server. The generic server class *odbc* relies strictly on information returned from the ODBC driver to determine these capabilities. Other server classes such as *db2odbc* have more detailed knowledge of the capabilities of a remote server type and use that knowledge to supplement what is returned from the driver.

Once *syscapabilities* is initialized for a server, the capability information is retrieved only from the system table. This allows a user to alter the known capabilities of a server.

Since a remote server may not support all of the features of a given SQL statement, Adaptive Server Anywhere must break the statement into simpler components to the point that the query can be given to the remote server. SQL features not passed off to a remote server must be evaluated by Adaptive Server Anywhere itself.

For example, a query may contain an *ORDER BY* statement. If a remote server cannot perform *ORDER BY*, the statement is sent to the remote server without it and Adaptive Server Anywhere performs the *ORDER BY* on the result returned, before returning the result to the user. The result is that the user can employ the full range of Adaptive Server Anywhere supported SQL without concern for the features of a particular back end.

## Complete passthrough of the statement

The most efficient way to handle a statement is usually to hand as much of the original statement as possible off to the remote server involved. Adaptive Server Anywhere will attempt to pass off as much of the statement as is possible. In many cases this will be the complete statement as originally given to Adaptive Server Anywhere.

Adaptive Server Anywhere will hand off the complete statement when:

- ◆ Every table in the statement resides in the same remote server.
- ◆ The remote server is capable of processing all of the syntax in the statement.

In rare conditions, it may actually be more efficient to let Adaptive Server Anywhere do some of the work instead of passing it off. For example, Adaptive Server Anywhere may have a better sorting algorithm. In this case you may consider altering the capabilities of a remote server using the *ALTER SERVER* statement.

For more information, see "ALTER SERVER statement" on page 220 of the book *ASA SQL Reference Manual*.

## Partial passthrough of the statement

If a statement contains references to multiple servers, or uses SQL features not supported by a remote server, the query is decomposed into simpler parts.

### Select

SELECT statements are broken down by removing portions that cannot be passed on and letting Adaptive Server Anywhere perform the feature. For example, let's say a remote server can not process the `atan2()` function in the following statement:

```
select a,b,c where atan2(b,10) > 3 and c = 10
```

The statement sent to the remote server would be converted to:

```
select a,b,c where c = 10
```

Locally, Adaptive Server Anywhere would apply "`where atan2(b,10) > 3`" to the intermediate result set.

### Joins

When two tables are joined, one table is selected to be the outer table. The outer table is scanned based on the WHERE conditions that apply to it. For every qualifying row found, the other table, known as the inner table is scanned to find a row that matches the join condition.

This same algorithm is used when remote tables are referenced. Since the cost of searching a remote table is usually much higher than a local table (due to network I/O), every effort is made to make the remote table the outermost table in the join.

### Update and delete

If Adaptive Server Anywhere cannot pass off an UPDATE or DELETE statement entirely to a remote server, it must change the statement into a table scan containing as much of the original WHERE clause as possible, followed by positioned UPDATE or DELETE "where current of cursor" when a qualifying row is found.

For example, when the function **atan2** is not supported by a remote server:

```
UPDATE t1
SET a = atan2(b, 10)
WHERE b > 5
```

Would be converted to the following:

```
SELECT a,b
FROM t1
WHERE b > 5
```

Each time a row is found, Adaptive Server Anywhere would calculate the new value of *a* and issue:

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR
```

If *a* already has a value that equals the "new value", a positioned UPDATE would not be necessary and would not be sent remotely.

In order to process an UPDATE or DELETE that requires a table scan, the remote data source must support the ability to perform a positioned UPDATE or DELETE ("where current of cursor"). Some data sources do not support this capability.

**Temporary tables cannot be updated**

In this release of Adaptive Server Anywhere, an UPDATE or DELETE cannot be performed if an intermediate temporary table is required in Adaptive Server Anywhere. This occurs in queries with ORDER BY and some queries with subqueries.

# Troubleshooting remote data access

This section provides some hints for troubleshooting remote servers.

## Features not supported for remote data

The following Adaptive Server Anywhere features are not supported on remote data. Attempts to use these features will therefore run into problems:

- ◆ ALTER TABLE statement against remote tables
- ◆ Triggers defined on proxy tables will not fire
- ◆ SQL Remote
- ◆ Java data types
- ◆ Foreign keys that refer to remote tables are ignored
- ◆ The READTEXT, WRITETEXT, and TEXTPTR functions.
- ◆ Positioned UPDATE and DELETE
- ◆ UPDATE and DELETE requiring an intermediate temporary table.
- ◆ Backwards scrolling on cursors opened against remote data. Fetch statements must be NEXT or RELATIVE 1.
- ◆ If a column on a remote table has a name that is a keyword on the remote server, you cannot access data in that column. Adaptive Server Anywhere cannot know all of the remote server reserved words. You can execute a CREATE EXISTING TABLE statement, and import the definition but you cannot select that column.

## Case sensitivity

The case sensitivity setting of your Adaptive Server Anywhere database should match the settings used by any remote servers accessed.

Adaptive Server Anywhere databases are created case insensitive by default. With this configuration, unpredictable results may occur when selecting from a case sensitive database. Different results will occur depending on whether ORDER BY or string comparisons are pushed off to a remote server or evaluated by the local Adaptive Server Anywhere.

## Connectivity problems

Take the following steps to be sure you can connect to a remote server:

- ◆ Determine that you can connect to a remote server using a client tool such as Interactive SQL before configuring Adaptive Server Anywhere.
- ◆ Perform a simple passthrough statement to a remote server to check your connectivity and remote login configuration. For example:

```
FORWARD TO testasa {select @@version}
```

- ◆ Turn on remote tracing for a trace of the interactions with remote servers.

```
SET OPTION cis_option = 2
```

## General problems with queries

If you are faced with some type of problem with the way Adaptive Server Anywhere is handling a query against a remote table, it is usually helpful to understand how Adaptive Server Anywhere is executing that query. You can display remote tracing as well as a description of the query execution plan:

```
SET OPTION cis_option = 6
```

## Queries blocked on themselves

If you access multiple databases on a single Adaptive Server Anywhere server, you may need to increase the number of threads used by the database server on Windows using the `-gx` command-line switch.

You must have enough threads available to support the individual tasks that are being run by a query. Failure to provide the number of required tasks can lead to a query becoming blocked on itself.



# Server Classes for Remote Data Access

About this chapter	<p>This chapter describes how Adaptive Server Anywhere interfaces with different classes of servers. It describes</p> <ul style="list-style-type: none"><li>◆ Types of servers that each server class supports</li><li>◆ The USING clause value for the CREATE SERVER statement for each server class</li><li>◆ Special configuration requirements</li></ul>
--------------------	--

Contents	<b>Topic</b>	<b>Page</b>
	Overview	488
	JDBC-based server classes	489
	ODBC-based server classes	492

# Overview

The server class you specify in the `CREATE SERVER` statement determines the behavior of a remote connection. The server classes give Adaptive Server Anywhere detailed server capability information. Adaptive Server Anywhere formats SQL statements specific to a server's capabilities.

There are two categories of server classes:

- ◆ JDBC-based server classes
- ◆ ODBC-based server classes

Each server class has a set of unique characteristics that database administrators and programmers need to know to configure the server for remote data access.

When using this chapter, refer both to the section generic to the server class category (JDBC-based or ODBC-based), and to the section specific to the individual server class.

## JDBC-based server classes

JDBC-based server classes are used when Adaptive Server Anywhere internally uses a Java virtual machine and jConnect 4.0 to connect to the remote server. The JDBC-based server classes are:

- ◆ **asajdbc** Adaptive Server Anywhere (version 6 and later).
- ◆ **asejdbc** Adaptive Server Enterprise and SQL Server (version 10 and later).

## Configuration notes for JDBC classes

When you access remote servers defined with JDBC-based classes, consider that:

- ◆ Your local database must be enabled for Java.
  - 🔗 For more information, see "Java-enabling a database" on page 89 of the book *ASA Programming Guide*.
- ◆ The Java virtual machine needs more than the default amount of memory to load and run jConnect. Set these memory options to at least the following values:

```
SET OPTION PUBLIC.JAVA_NAMESPACE_SIZE = 3000000
```

```
SET OPTION PUBLIC.JAVA_HEAP_SIZE = 1000000
```

- ◆ Since jConnect 4.0 is automatically installed with Adaptive Server Anywhere, no additional drivers need to be installed.
- ◆ For optimum performance, Sybase recommends an ODBC-based class (*asaodbc* or *aseodbc*).
- ◆ Any remote server that you access using the *asejdbc* or *asajdbc* server class must be set up to handle a jConnect 4.x based client. The jConnect setup scripts are *SQL\_anywhere.SQL* for Adaptive Server Anywhere or *SQL\_server.SQL* for Adaptive Server Enterprise. Run these against any remote server you will be using.

## Server class asajdbc

A server with server class *asajdbc* is Adaptive Server Anywhere (version 6 and later). No special requirements exist for the configuration of an Adaptive Server Anywhere data source.

**USING parameter value in the CREATE SERVER statement**

You must perform a separate CREATE SERVER for each Adaptive Server Anywhere database you intend to access. For example, if an Adaptive Server Anywhere server named *testasa* is running on the machine 'banana' and owns three databases (*db1*, *db2*, *db3*), you would configure the local Adaptive Server Anywhere similar to this:

```
CREATE SERVER testasadb1
CLASS 'asajdbc'
USING 'banana:2638/db1'

CREATE SERVER testasadb2
CLASS 'asajdbc'
USING 'banana:2638/db2'

CREATE SERVER testasadb2
CLASS 'asajdbc'
USING 'banana:2638/db3'
```

If you do not specify a */databasename* value, the remote connection uses the remote Adaptive Server Anywhere default database.

☞ For more information about the CREATE SERVER statement, see "CREATE SERVER statement" on page 321 of the book *ASA SQL Reference Manual*.

**Server class asejdbc**

A server with server class asejdbc is Adaptive Server Enterprise, SQL Server (version 10 and later). No special requirements exist for the configuration of an Adaptive Server Enterprise data source.

**Data type conversions: JDBC and Adaptive Server Enterprise**

When you issue a CREATE TABLE statement, Adaptive Server Anywhere automatically converts the data types to the corresponding Adaptive Server Enterprise data types. The following table describes the Adaptive Server Anywhere to Adaptive Server Enterprise data type conversions.

Adaptive Server Anywhere data type	ASE default data type
bit	bit
tinyint	tinyint
smallint	smallint
int	int

Adaptive Server Anywhere data type	ASE default data type
integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	not supported
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	not supported
float	real
real	real
double	float
smallmoney	numeric(10,4)
money	numeric(19,4)
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	varchar(n)
character(n)	varchar(n)
varchar(n)	varchar(n)
character varying(n)	varchar(n)
long varchar	text
text	text
binary(n)	binary(n)
long binary	image
image	image
bigint	numeric(20,0)

## ODBC-based server classes

The ODBC-based server classes include:

- ◆ asaodbc
- ◆ aseodbc
- ◆ db2odbc
- ◆ mssodbc
- ◆ oraodbc
- ◆ odbc

### Defining ODBC external servers

The most common way of defining an ODBC-based server bases it on an ODBC data source. To do this, you must create a data source in the ODBC Administrator.

Once you have the data source defined, the USING clause in the CREATE SERVER statement should match the ODBC data source name.

For example, to configure a DB2 server named **mydb2** whose Data Source Name is also **mydb2**, use:

```
CREATE SERVER mydb2
CLASS 'db2odbc'
USING 'mydb2'
```

For more information on creating data sources, see "Creating an ODBC data source" on page 53 of the book *ASA Database Administration Guide*.

Using connection  
strings instead of  
data sources

An alternative, which avoids using data sources, is to supply a connection string in the USING clause of the CREATE SERVER statement. To do this, you must know the connection parameters for the ODBC driver you are using. For example, a connection to an ASA may be as follows:

```
CREATE SERVER testasa
CLASS 'asaodbc'
USING 'driver=adaptive server anywhere
8.0;eng=testasa;dbn=sample;links=tcipip{ }'
```

This defines a connection to an Adaptive Server Anywhere database server named **testasa**, database **sample**, and using the TCP-IP protocol.

See also

For information specific to particular ODBC server classes, see:

- ◆ "Server class asaodbc" on page 493

- ◆ "Server class aseodbc" on page 493
- ◆ "Server class db2odbc" on page 495
- ◆ "Server class oraodbc" on page 497
- ◆ "Server class mssodbc" on page 498
- ◆ "Server class odbc" on page 500

## Server class asaodbc

A server with server class *asaodbc* is Adaptive Server Anywhere version 5.5 or later. No special requirements exist for the configuration of an Adaptive Server Anywhere data source.

To access Adaptive Server Anywhere servers that support multiple databases, create an ODBC data source name defining a connection to each database. Issue a CREATE SERVER statement for each of these ODBC data source names.

## Server class aseodbc

A server with server class *aseodbc* is Adaptive Server Enterprise, SQL Server (version 10 and later). Adaptive Server Anywhere requires the installation of the Adaptive Server Enterprise ODBC driver and Open Client connectivity libraries to connect to a remote Adaptive Server with class *aseodbc*. However, the performance is better than with the *asejdbc* class.

### Notes

- ◆ Open Client should be version 11.1.1, EBF 7886 or above. Install Open Client and verify connectivity to the Adaptive Server before you install ODBC and configure Adaptive Server Anywhere. The Sybase ODBC driver should be version 11.1.1, EBF 7911 or above.
- ◆ Configure a User Data Source in the Configuration Manager with the following attributes:
  - ◆ Under the General tab:

Enter any value for Data Source Name. This value is used in the USING clause of the CREATE SERVER statement.

The server name should match the name of the server in the Sybase interfaces file.
  - ◆ Under the Advanced tab, check the Application Using Threads box and check the Enable Quoted Identifiers box.
  - ◆ Under the Connection tab:

- Set the charset field to match your Adaptive Server Anywhere character set.
- Set the language field to your preferred language for error messages.
- ◆ Under the Performance tab:
  - Set Prepare Method to "2-Full."
  - Set Fetch Array Size as large as possible for best performance. This increases memory requirements since this is the number of rows that must be cached in memory. Sybase recommends using a value of 100.
  - Set Select Method to "0-Cursor."
  - Set Packet Size to as large as possible. Sybase recommends using a value of -1.
  - Set Connection Cache to 1.

**Data type conversions: ODBC and Adaptive Server Enterprise**

When you issue a CREATE TABLE statement, Adaptive Server Anywhere automatically converts the data types to the corresponding Adaptive Server Enterprise data types. The following table describes the Adaptive Server Anywhere to Adaptive Server Enterprise data type conversions.

Adaptive Server Anywhere data type	Adaptive Server Enterprise default data type
Bit	bit
Tinyint	tinyint
Smallint	smallint
Int	int
Integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	not supported
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	not supported
Float	real
Real	real
Double	float
Smallmoney	numeric(10,4)



Adaptive Server Anywhere data type	Adaptive Server Enterprise default data type
Money	numeric(19,4)
Date	datetime
Time	datetime
Timestamp	datetime
Smalldatetime	datetime
Datetime	datetime
char(n)	varchar(n)
Character(n)	varchar(n)
varchar(n)	varchar(n)
Character varying(n)	varchar(n)
long varchar	text
Text	text
binary(n)	binary(n)
long binary	image
Image	image
Bigint	numeric(20,0)

## Server class db2odbc

A server with server class db2odbc is IBM DB2

### Notes

- ◆ Sybase certifies the use of IBM's DB2 Connect version 5, with fix pack WR09044. Configure and test your ODBC configuration using the instructions for that product. Adaptive Server Anywhere has no specific requirements on configuration of DB2 data sources.
- ◆ The following is an example of a CREATE EXISTING TABLE statement for a DB2 server with an ODBC data source named mydb2:

```
CREATE EXISTING TABLE ibmcol  
AT 'mydb2..sysibm.syscolumns'
```

## Data type conversions: DB2

When you issue a CREATE TABLE statement, Adaptive Server Anywhere automatically converts the data types to the corresponding DB2 data types. The following table describes the Adaptive Server Anywhere to DB2 data type conversions.

Adaptive Server Anywhere data type	DB2 default data type
Bit	smallint
Tinyint	smallint
Smallint	smallint
Int	int
Integer	int
Bigint	decimal(20,0)
char(1–254)	varchar(n)
char(255–4000)	varchar(n)
char(4001–32767)	long varchar
Character(1–254)	varchar(n)
Character(255–4000)	varchar(n)
Character(4001–32767)	long varchar
varchar(1–4000)	varchar(n)
varchar(4001–32767)	long varchar
Character varying(1–4000)	varchar(n)
Character varying(4001–32767)	long varchar
long varchar	long varchar
text	long varchar
binary(1–4000)	varchar for bit data
binary(4001–32767)	long varchar for bit data
long binary	long varchar for bit data
image	long varchar for bit data
decimal [defaults p=30, s=6]	decimal(30,6)
numeric [defaults p=30 s=6]	decimal(30,6)
decimal(128, 128)	NOT SUPPORTED
numeric(128, 128)	NOT SUPPORTED

Adaptive Server Anywhere data type	DB2 default data type
real	real
float	float
double	float
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time
smalldatetime	timestamp
datetime	timestamp
timestamp	timestamp

## Server class oraodbc

A server with server class **oraodbc** is Oracle version 8.0 or later.

### Notes

- ◆ Sybase certifies the use of version 8.0.03 of Oracle's ODBC driver. Configure and test your ODBC configuration using the instructions for that product.
- ◆ The following is an example of a CREATE EXISTING TABLE statement for an Oracle server named *myora*:
 

```
CREATE EXISTING TABLE employees
AT 'myora.database.owner.employees'
```
- ◆ Due to Oracle ODBC driver restrictions, you cannot issue a CREATE EXISTING TABLE for system tables. A message returns stating that the table or columns cannot be found.

## Data type conversions: Oracle

When you issue a CREATE TABLE statement, Adaptive Server Anywhere automatically converts the data types to the corresponding Oracle data types. The following table describes the Adaptive Server Anywhere to Oracle data type conversions.

Adaptive Server Anywhere data type	Oracle data type
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)
int	number(11,0)
bigint	number(20,0)
decimal(prec, scale)	number(prec, scale)
numeric(prec, scale)	number(prec, scale)
float	float
real	real
smallmoney	numeric(13,4)
money	number(19,4)
date	date
time	date
timestamp	date
smalldatetime	date
datetime	date
char(n)	if (n > 255) long else varchar(n)
varchar(n)	if (n > 2000) long else varchar(n)
long varchar	long
binary(n)	if (n > 255) long raw else raw(n)
varbinary(n)	if (n > 255) long raw else raw(n)
long binary	long raw

## Server class mssodbc

A server with server class **mssodbc** is Microsoft SQL Server version 6.5, Service Pack 4.

### Notes

- ◆ Sybase certifies the use of version 3.60.0319 of Microsoft SQL Server's ODBC driver (included in MDAC 2.0 release). Configure and test your ODBC configuration using the instructions for that product.

- ◆ The following is an example of a CREATE EXISTING TABLE statement for a Microsoft SQL Server named *mymssql*:

```
CREATE EXISTING TABLE accounts,  
AT 'mymssql.database.owner.accounts'
```

## Data type conversions: Microsoft SQL Server

When you issue a CREATE TABLE statement, Adaptive Server Anywhere automatically converts the data types to the corresponding Microsoft SQL Server data types. The following table describes the Adaptive Server Anywhere to Microsoft SQL Server data type conversions.

Adaptive Server Anywhere data type	Microsoft SQL Server default data type
bit	bit
tinyint	tinyint
smallint	smallint
int	int
bigint	numeric(20,0)
decimal [defaults p=30, s=6]	decimal(prec, scale)
numeric [defaults p=30 s=6]	numeric(prec, scale)
float	if (prec) float(prec) else float
real	real
smallmoney	smallmoney
money	money
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	if (length > 255) text else varchar(length)
character(n)	char(n)

Adaptive Server Anywhere data type	Microsoft SQL Server default data type
varchar(n)	if (length > 255) text else varchar(length)
long varchar	text
binary(n)	if (length > 255) image else binary(length)
long binary	image
double	float

Server class **odbc**

ODBC data sources that do not have their own server class use server class **odbc**. You can use any ODBC driver that complies with ODBC version 2.0 compliance level 1 or higher. Sybase certifies the following ODBC data sources:

- ◆ "Microsoft Excel" on page 500
- ◆ "Microsoft Access" on page 501
- ◆ "Microsoft FoxPro" on page 502
- ◆ "Lotus Notes" on page 502

The latest versions of Microsoft ODBC drivers can be obtained through the Microsoft Data Access Components (MDAC) distribution found at [www.microsoft.com/data/download.htm](http://www.microsoft.com/data/download.htm). The Microsoft driver versions listed below are part of MDAC 2.0.

The following sections provide notes on accessing these data sources.

**Microsoft Excel (Microsoft 3.51.171300)**

With Excel, each Excel workbook is logically considered to be a database holding several tables. Tables are mapped to sheets in a workbook. When you configure an ODBC data source name in the ODBC driver manager, you specify a default workbook name associated with that data source. However, when you issue a CREATE TABLE statement, you can override the default and specify a workbook name in the location string. This allows you to use a single ODBC DSN to access all of your excel workbooks.

In this example, an ODBC data source named **excel** was created. To create a workbook named *work1.xls* with a sheet (table) called mywork:

```
CREATE TABLE mywork (a int, b char(20))
```

```
AT 'excel;d:\work1.xls;mywork'
```

To create a second sheet (or table) execute a statement such as:

```
CREATE TABLE mywork2 (x float, y int)
AT 'excel;d:\work1.xls;mywork2'
```

You can import existing worksheets into Adaptive Server Anywhere using `CREATE EXISTING`, under the assumption that the first row of your spreadsheet contains column names.

```
CREATE EXISTING TABLE mywork
AT 'excel;d:\work1;mywork'
```

If Adaptive Server Anywhere reports that the table is not found, you may need to explicitly state the column and row range you wish to map to. For example:

```
CREATE EXISTING TABLE mywork
AT 'excel;d:\work1;mywork$'
```

Adding the `$` to the sheet name indicates that the entire worksheet should be selected.

Note in the location string specified by `AT` that a semicolon is used instead of a period for field separators. This is because periods occur in the file names. Excel does not support the owner name field so leave this blank.

Deletes are not supported. Also some updates may not be possible since the Excel driver does not support positioned updates.

## Microsoft Access (Microsoft 3.51.171300)

Access databases are stored in a `.mdb` file. Using the ODBC manager, create an ODBC data source and map it to one of these files. A new `.mdb` file can be created through the ODBC manager. This database file becomes the default if you don't specify a different default when you create a table through Adaptive Server Anywhere.

Assuming an ODBC data source named `access`.

```
CREATE TABLE tabl (a int, b char(10))
AT 'access...tabl'
```

or

```
CREATE TABLE tabl (a int, b char(10))
AT 'access;d:\pcdb\data.mdb;tabl'
```

or

```
CREATE EXISTING TABLE tabl
AT 'access;d:\pcdb\data.mdb;tabl'
```

Access does not support the owner name qualification; leave it empty.

## Microsoft FoxPro (Microsoft 3.51.171300)

You can store FoxPro tables together inside a single FoxPro database file (.dbc), or, you can store each table in its own separate .dbf file. When using .dbf files, be sure the file name is filled into the location string; otherwise the directory that Adaptive Server Anywhere was started in is used.

```
CREATE TABLE fox1 (a int, b char(20))
AT 'foxpro;d:\pcdb\;fox1'
```

This statement creates a file named *d:\pcdb\fox1.dbf* when you choose the "free table directory" option in the odbc driver manager.

## Lotus Notes SQL 2.0 (2.04.0203)

You can obtain this driver from the Lotus Web site. Read the documentation that comes with it for an explanation of how Notes data maps to relational tables. You can easily map Adaptive Server Anywhere tables to Notes forms.

Here is how to set up Adaptive Server Anywhere to access the Address sample file.

- ◆ Create an ODBC data source using the NotesSQL driver. The database will be the sample names file: *c:\notes\data\names.nsf*. The Map Special Characters option should be turned on. For this example, the Data Source Name is *my\_notes\_dsn*.

- ◆ Create a server in Adaptive Server Anywhere:

```
CREATE SERVER names
CLASS 'odbc'
USING 'my_notes_dsn'
```

- ◆ Map the Person form into an Adaptive Server Anywhere table:

```
CREATE EXISTING TABLE Person
AT 'names...Person'
```

- ◆ Query the table

```
SELECT * FROM Person
```

### Avoiding password prompts

Lotus Notes does not support sending a user name and password through the ODBC API. If you try to access Lotus notes using a password protected ID, a window appears on the machine where Adaptive Server Anywhere is running, and prompts you for a password. Avoid this behavior in multi-user server environments.



To access Lotus Notes unattended, without ever receiving a password prompt, you must use a non-password-protected ID. You can remove password protection from your ID by clearing it (File►Tools►User ID►Clear Password), unless your Domino administrator required a password when your ID was created. In this case, you will not be able to clear it.



## **Adding Logic to the Database**


This part describes how to build logic into your database using SQL stored procedures and triggers. Storing logic in the database makes it available automatically to all applications, providing consistency, performance, and security benefits. The Stored Procedure debugger is a powerful tool for debugging all kinds of logic.

---

# Using Procedures, Triggers, and Batches

About this chapter

Procedures and triggers store procedural SQL statements in the database for use by all applications. They enhance the security, efficiency, and standardization of databases. User-defined functions are one kind of procedures that return a value to the calling environment for use in queries and other SQL statements. Batches are sets of SQL statements submitted to the database server as a group. Many features available in procedures and triggers, such as control statements, are also available in batches.

 For many purposes, server-side JDBC provides a more flexible way to build logic into the database than SQL stored procedures. For information about JDBC, see "Data Access Using JDBC" on page 129 of the book *ASA Programming Guide*.

Contents

Topic	Page
Procedure and trigger overview	509
Benefits of procedures and triggers	510
Introduction to procedures	511
Introduction to user-defined functions	518
Introduction to triggers	522
Introduction to batches	529
Control statements	531
The structure of procedures and triggers	535
Returning results from procedures	539
Using cursors in procedures and triggers	545
Errors and warnings in procedures and triggers	548
Using the EXECUTE IMMEDIATE statement in procedures	557
Transactions and savepoints in procedures and triggers	558
Tips for writing procedures	559
Statements allowed in batches	561



## Procedure and trigger overview

Procedures and triggers store procedural SQL statements in a database for use by all applications. They can include control statements that allow repetition (LOOP statement) and conditional execution (IF statement and CASE statement) of SQL statements.

Procedures are invoked with a CALL statement, and use parameters to accept values and return values to the calling environment. Procedures can return result sets to the caller, call other procedures, or fire triggers. For example, a user-defined function is a type of stored procedure that returns a single value to the calling environment. User-defined functions do not modify parameters passed to them, but rather, they broaden the scope of functions available to queries and other SQL statements.

Triggers are associated with specific database tables. They fire automatically whenever someone inserts, updates or deletes rows of the associated table. Triggers can call procedures and fire other triggers, but they have no parameters and cannot be invoked by a CALL statement.

### Procedure debugger

You can debug stored procedures and triggers using the Sybase debugger. For more information, see "Debugging Logic in the Database" on page 571.

## Benefits of procedures and triggers

Definitions for procedures and triggers appear in the database, separately from any one database application. This separation provides a number of advantages.

### Standardization

Procedures and triggers standardize actions performed by more than one application program. By coding the action once and storing it in the database for future use, applications need only call the procedure or fire the trigger to achieve the desired result repeatedly. And since changes occur in only one place, all applications using the action automatically acquire the new functionality if the implementation of the action changes.

### Efficiency

Procedures and triggers used in a network database server environment can access data in the database without requiring network communication. This means they execute faster and with less impact on network performance than if they had been implemented in an application on one of the client machines.

When you create a procedure or trigger, it is automatically checked for correct syntax, and then stored in the system tables. The first time any application calls or fires a procedure or trigger, it is compiled from the system tables into the server's virtual memory and executed from there. Since one copy of the procedure or trigger remains in memory after the first execution, repeated executions of the same procedure or trigger happen instantly. As well, several applications can use a procedure or trigger concurrently, or one application can use it recursively.

Procedures are less efficient if they contain simple queries and have many arguments. For complex queries, procedures are more efficient.

### Security

Procedures and triggers provide security by allowing users limited access to data in tables that they cannot directly examine or modify.

Triggers, for example, execute under the table permissions of the owner of the associated table, but any user with permissions to insert, update or delete rows in the table can fire them. Similarly, procedures (including user-defined functions) execute with permissions of the procedure owner, but any user granted permissions can call them. This means that procedures and triggers can (and usually do) have different permissions than the user ID that invoked them.



## Introduction to procedures

To use procedures, you need to understand how to:

- ◆ Create procedures
- ◆ Call procedures from a database application
- ◆ Drop or remove procedures
- ◆ Control who has permissions to use procedures

This section discusses the above aspects of using procedures, as well as some different applications of procedures.

## Creating procedures

Adaptive Server Anywhere provides a number of tools that let you create a new procedure.

In Sybase Central, you can use a wizard to provide necessary information and then complete the code in a generic code editor. Sybase Central also provides procedure templates (located in the Procedures & Functions folder) that you can open and modify.

Alternatively, you use the `CREATE PROCEDURE` statement to create procedures. However, you must have `RESOURCE` authority. Where you enter the statement depends on which tool you use.

### ❖ To create a new procedure (Sybase Central):

- 1 Connect to a database with DBA or Resource authority.
- 2 Open the Procedures & Functions folder of the database.
- 3 In the right pane, double-click Add Procedure (Wizard).
  - ◆ Follow the instructions in the wizard.
  - ◆ When the Code Editor opens, complete the code of the procedure.
  - ◆ To execute and save the procedure, choose File►Save Procedure.
  - ◆ To save the procedure without executing it, choose File►Save File As.

The new procedure appears in the Procedures & Functions folder.

❖ **To create a new remote procedure (Sybase Central):**

- 1 Connect to a database with DBA authority.
- 2 Open the Procedures & Functions folder of the database.
- 3 In the right pane, double-click Add Remote Procedure (Wizard).
- 4 When the Code Editor opens, complete the code of the procedure.
- 5 Follow the instructions in the wizard.

**Tip**

You can also create a remote procedure by right-clicking a remote server in the Remote Servers folder and choosing Add Remote Procedure from the popup menu.

❖ **To create a procedure using a different tool:**

- ◆ Follow the instructions for your tool. You may need to change the command delimiter away from the semicolon before entering the CREATE PROCEDURE statement.

☞ For more information about connecting, see "Connecting to a Database" on page 37 of the book *ASA Database Administration Guide*.

**Example**

The following simple example creates the procedure *new\_dept*, which carries out an INSERT into the department table of the sample database, creating a new department.

```
CREATE PROCEDURE new_dept (  
    IN id INT,  
    IN name CHAR(35),  
    IN head_id INT )  
BEGIN  
    INSERT  
    INTO DBA.department ( dept_id,  
        dept_name,  
        dept_head_id )  
    VALUES ( id, name, head_id );  
END
```

The body of a procedure is a compound statement. The compound statement starts with a BEGIN statement and concludes with an END statement. In the case of *new\_dept*, the compound statement is a single INSERT bracketed by BEGIN and END statements.

Parameters to procedures are marked as one of IN, OUT, or INOUT. All parameters to the *new\_dept* procedure are IN parameters, as they are not changed by the procedure.

For more information, see "CREATE PROCEDURE statement" on page 305 of the book *ASA SQL Reference Manual*, "ALTER PROCEDURE statement" on page 214 of the book *ASA SQL Reference Manual*, and "Using compound statements" on page 533.

## Altering procedures

You can modify an existing procedure using either Sybase Central or Interactive SQL. You must have DBA authority or be the owner of the procedure.

In Sybase Central, you cannot rename an existing procedure directly. Instead, you must create a new procedure with the new name, copy the previous code to it, and then delete the old procedure.

Alternatively, you can use an ALTER PROCEDURE statement to modify an existing procedure. You must include the entire new procedure in this statement (in the same syntax as in the CREATE PROCEDURE statement that created the procedure). You must also reassign user permissions on the procedure.

For more information on altering database object properties, see "Setting properties for database objects" on page 34.

For more information on granting or revoking permissions for procedures, see "Granting permissions on procedures" on page 363 of the book *ASA Database Administration Guide* and "Revoking user permissions" on page 366 of the book *ASA Database Administration Guide*.

### ❖ To alter the code of a procedure (Sybase Central):

- 1 Open the Procedures & Functions folder.
- 2 Right-click the desired procedure.
- 3 From the popup menu, do one of the following:
  - ◆ Choose Open as Watcom-SQL to edit the code in the Watcom-SQL dialect.
  - ◆ Choose Open as Transact-SQL to edit the code in the Transact-SQL dialect.
- 4 In the Code Editor, edit the procedure's code.
- 5 To execute the code in the database, choose File►Save/Execute in Database.

❖ **To alter the code of a procedure (SQL):**

- 1 Connect to the database.
- 2 Execute an ALTER PROCEDURE statement. Include the entire new procedure in this statement.

🔗 For more information, see "ALTER PROCEDURE statement" on page 214 of the book *ASA SQL Reference Manual*, "CREATE PROCEDURE statement" on page 305 of the book *ASA SQL Reference Manual*, and "Creating procedures" on page 511.

## Calling procedures

CALL statements invoke procedures. Procedures can be called by an application program, or by other procedures and triggers.

🔗 For more information, see "CALL statement" on page 254 of the book *ASA SQL Reference Manual*.

The following statement calls the *new\_dept* procedure to insert an Eastern Sales department:

```
CALL new_dept( 210, 'Eastern Sales', 902 );
```

After this call, you may wish to check the department table to see that the new department has been added.

All users who have been granted EXECUTE permissions for the procedure can call the *new\_dept* procedure, even if they have no permissions on the *department* table.

🔗 For more information about EXECUTE permissions, see "EXECUTE statement [ESQL]" on page 414 of the book *ASA SQL Reference Manual*.

## Copying procedures in Sybase Central

In Sybase Central, you can copy procedures between databases. To do so, select the procedures in the right pane of Sybase Central and drag it to the Procedures & Functions folder of another connected database. A new procedure is then created, and the original procedure's code is copied to it.

Note that only the procedure code is copied to the new procedure. The other procedure properties (permissions, etc.) are not copied. A procedure can be copied to the same database, provided it is given a new name.

## Deleting procedures

Once you create a procedure, it remains in the database until someone explicitly removes it. Only the owner of the procedure or a user with DBA authority can drop the procedure from the database.

### ❖ To delete a procedure (Sybase Central):

- 1 Connect to a database with DBA authority or as the owner of the procedure.
- 2 Open the Procedures & Functions folder.
- 3 Right-click the desired procedure and choose Delete from the popup menu.

### ❖ To delete a procedure (SQL):

- 1 Connect to a database with DBA authority or as the owner of the procedure.
- 2 Execute a DROP PROCEDURE statement.

#### Example

The following statement removes the procedure *new\_dept* from the database:

```
DROP PROCEDURE new_dept
```

 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Returning procedure results in parameters

Procedures return results to the calling environment in one of the following ways:

- ◆ Individual values are returned as OUT or INOUT parameters.
- ◆ Result sets can be returned.
- ◆ A single result can be returned using a RETURN statement.

This section describes how to return results from procedures as parameters.

The following procedure on the sample database returns the average salary of employees as an OUT parameter.

```
CREATE PROCEDURE AverageSalary( OUT avgsal  
    NUMERIC (20,3) )  
BEGIN  
    SELECT AVG( salary )  
    INTO avgsal
```

```
        FROM employee;  
END
```

❖ **To run this procedure and display its output (SQL):**

- 1 Connect to the sample database from Interactive SQL with a user ID of **DBA** and a password of **SQL**. For more information about connecting, see "Connecting to a Database" on page 37 of the book *ASA Database Administration Guide*.
- 2 In the SQL Statements pane, type the above procedure code.
- 3 Create a variable to hold the procedure output. In this case, the output variable is numeric, with three decimal places, so create a variable as follows:

```
CREATE VARIABLE Average NUMERIC(20,3)
```

- 4 Call the procedure using the created variable to hold the result:

```
CALL AverageSalary(Average)
```

If the procedure was created and run properly, the Interactive SQL Messages pane does not display any errors.

- 5 Execute the SELECT Average statement to inspect the value of the variable.

Look at the value of the output variable Average. The Results tab in the Results pane displays the value 49988.623 for this variable, the average employee salary.

## Returning procedure results in result sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query. The following procedure returns a result set containing the salary for each employee in a given department:

```
CREATE PROCEDURE SalaryList ( IN department_id INT)  
RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )  
BEGIN  
    SELECT emp_id, salary  
    FROM employee  
    WHERE employee.dept_id = department_id;  
END
```

If Interactive SQL calls this procedure, the names in the RESULT clause are matched to the results of the query and used as column headings in the displayed results.

To test this procedure from Interactive SQL, you can CALL it, specifying one of the departments of the company. In Interactive SQL, the results appear on the Results tab in the Results pane.

**Example**

To list the salaries of employees in the R & D department (department ID 100), type the following:

```
CALL SalaryList (100)
```

Employee ID	Salary
102	45700
105	62000
160	57490
243	72995
...	...

Interactive SQL can only return multiple result sets if you have this option enabled on the Results tab of the Options dialog. Each result set appears on a separate tab in the Results pane.

 For more information, see "Returning multiple result sets from procedures" on page 542.

# Introduction to user-defined functions

User-defined functions are a class of procedures that return a single value to the calling environment. Adaptive Server Anywhere treats all user-defined functions as idempotent: the function returns a consistent result for the same parameters and is free of side effects. That is, the server assumes that two successive calls to the same function with the same parameters will return the same result, and will not have any unwanted side-effects on the query's semantics.

This section introduces creating, using, and dropping user-defined functions.

## Creating user-defined functions

You use the `CREATE FUNCTION` statement to create user-defined functions. However, you must have `RESOURCE` authority.

The following simple example creates a function that concatenates two strings, together with a space, to form a full name from a first name and a last name.

```
CREATE FUNCTION fullname (firstname CHAR(30),
                           lastname CHAR(30))
RETURNS CHAR(61)
BEGIN
    DECLARE name CHAR(61);
    SET name = firstname || ' ' || lastname;
    RETURN ( name );
END
```

### ❖ To create this example using Interactive SQL:

- 1 Connect to the sample database from Interactive SQL with a user ID of **DBA** and a password of **SQL**. For more information about connecting, see "Connecting to a Database" on page 37 of the book *ASA Database Administration Guide*.
- 2 In the SQL Statements pane, type the above function code.

#### **Note**

If you are using a tool other than Interactive SQL or Sybase Central, you may need to change the command delimiter to something other than the semicolon.



☞ For more information, see "CREATE FUNCTION statement" on page 296 of the book *ASA SQL Reference Manual*.

The CREATE FUNCTION syntax differs slightly from that of the CREATE PROCEDURE statement. The following are distinctive differences:

- ◆ No IN, OUT, or INOUT keywords are required, as all parameters are IN parameters.
- ◆ The RETURNS clause is required to specify the data type being returned.
- ◆ The RETURN statement is required to specify the value being returned.

## Calling user-defined functions

A user-defined function can be used, subject to permissions, in any place you would use a built-in non-aggregate function.

The following statement in Interactive SQL returns a full name from two columns containing a first and last name:

```
SELECT fullname (emp_fname, emp_lname)
FROM employee;
```

### **Fullname (emp\_fname, emp\_lname)**

---

Fran Whitney

Matthew Cobb

Philip Chin

...

The following statement in Interactive SQL returns a full name from a supplied first and last name:

```
SELECT fullname ('Jane', 'Smith');
```

### **Fullname ('Jane','Smith')**

---

Jane Smith

Any user who has been granted EXECUTE permissions for the function can use the *fullname* function.

## Example

The following user-defined function illustrates local declarations of variables.

The *customer* table includes some Canadian customers sprinkled among those from the USA, but there is no *country* column. The user-defined function *nationality* uses the fact that the US zip code is numeric while the Canadian postal code begins with a letter to distinguish Canadian and US customers.

```
CREATE FUNCTION nationality( cust_id INT )
RETURNS CHAR( 20 )
BEGIN
    DECLARE natl CHAR(20);
    IF cust_id IN ( SELECT id FROM customer
                    WHERE LEFT(zip,1) > '9') THEN
        SET natl = 'CDN';
    ELSE
        SET natl = 'USA';
    END IF;
    RETURN ( natl );
END
```

This example declares a variable *natl* to hold the nationality string, uses a SET statement to set a value for the variable, and returns the value of the *natl* string to the calling environment.

The following query lists all Canadian customers in the *customer* table:

```
SELECT *
FROM customer
WHERE nationality(id) = 'CDN'
```

Declarations of cursors and exceptions are discussed in later sections.

The same query restated without the function would perform better, especially if an index on zip existed. For example,

```
Select *
FROM customer
WHERE zip > '99999'
```

### Notes

While this function is useful for illustration, it may perform very poorly if used in a SELECT involving many rows. For example, if you used the SELECT query on a table containing 100 000 rows, of which 10 000 are returned, the function will be called 10 000 times. If you use it in the WHERE clause of the same query, it would be called 100 000 times.

## Dropping user-defined functions

Once you create a user-defined function, it remains in the database until someone explicitly removes it. Only the owner of the function or a user with DBA authority can drop a function from the database.

The following statement removes the function *fullname* from the database:

```
DROP FUNCTION fullname
```

## Permissions to execute user-defined functions

Ownership of a user-defined function belongs to the user who created it, and that user can execute it without permission. The owner of a user-defined function can grant permissions to other users with the GRANT EXECUTE command.

For example, the creator of the function *fullname* could allow *another\_user* to use *fullname* with the statement:

```
GRANT EXECUTE ON fullname TO another_user
```

The following statement revokes permissions to use the function:

```
REVOKE EXECUTE ON fullname FROM another_user
```

🔗 For more information on managing user permissions on functions, see "Granting permissions on procedures" on page 363 of the book *ASA Database Administration Guide*.

# Introduction to triggers

A trigger is a special form of stored procedure that is executed automatically when a query that modifies data is executed. You use triggers whenever referential integrity and other declarative constraints are insufficient.

For more information on referential integrity, see "Ensuring Data Integrity" on page 65 and "CREATE TABLE statement" on page 350 of the book *ASA SQL Reference Manual*.

You may want to enforce a more complex form of referential integrity involving more detailed checking, or you may want to enforce checking on new data but allow legacy data to violate constraints. Another use for triggers is in logging the activity on database tables, independent of the applications using the database.

**Trigger execution permissions**

Triggers execute with the permissions of the owner of the associated table, not the user ID whose actions cause the trigger to fire. A trigger can modify rows in a table that a user could not modify directly.

Trigger events

Triggers can be defined on one or more of the following triggering events:

Action	Description
INSERT	Invokes the trigger whenever a new row is inserted into the table associated with the trigger
DELETE	Invokes the trigger whenever a row of the associated table is deleted.
UPDATE	Invokes the trigger whenever a row of the associated table is updated.
UPDATE OF column-list	Invokes the trigger whenever a row of the associated table is updated such that a column in the <i>column-list</i> has been modified

You may write separate triggers for each event that you need to handle or, if you have some shared actions and some actions that depend on the event, you can create a trigger for all events and use an IF statement to distinguish the action taking place.

For more information, see "Trigger operation conditions" on page 30 of the book *ASA SQL Reference Manual*.


Trigger times

Triggers can be either **row-level** or **statement-level**:

- ◆ A row-level trigger executes once for each row that is changed. Row-level triggers execute BEFORE or AFTER the row is changed.
- ◆ A statement-level trigger executes after the entire triggering statement is completed.

Flexibility in trigger execution time is particularly useful for triggers that rely on referential integrity actions such as cascaded updates or deletes being carried out (or not) as they execute.

If an error occurs while a trigger is executing, the operation that fired the trigger fails. INSERT, UPDATE, and DELETE are atomic operations (see "Atomic compound statements" on page 533). When they fail, all effects of the statement (including the effects of triggers and any procedures called by triggers) revert back to their pre-operation state.


 For a full description of trigger syntax, see "CREATE TRIGGER statement" on page 362 of the book *ASA SQL Reference Manual*.

## Creating triggers

You create triggers using either Sybase Central or Interactive SQL. In Sybase Central, you can compose the code in a Code Editor. In Interactive SQL, you can use a CREATE TRIGGER statement. For both tools, you must have DBA or RESOURCE authority to create a trigger and you must have ALTER permissions on the table associated with the trigger.

The body of a trigger consists of a compound statement: a set of semicolon-delimited SQL statements bracketed by a BEGIN and an END statement.

You cannot use COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements within a trigger.

 For more information, see the list of cross-references at the end of this section.

### ❖ To create a new trigger for a given table (Sybase Central):

- 1 Open the Triggers folder of the desired table.
- 2 In the right pane, double-click Add Trigger.
- 3 Follow the instructions in the wizard.
- 4 When the wizard finishes and opens the Code Editor for you, complete the code of the trigger.
- 5 To execute the code in the database, choose File►Save/Execute in Database.

❖ **To create a new trigger for a given table (SQL):**

- 1 Connect to a database.
- 2 Execute a CREATE TRIGGER statement.

**Example 1: A row-level INSERT trigger**

The following trigger is an example of a row-level INSERT trigger. It checks that the birth date entered for a new employee is reasonable:

```
CREATE TRIGGER check_birth_date
  AFTER INSERT ON Employee
  REFERENCING NEW AS new_employee
  FOR EACH ROW
  BEGIN
    DECLARE err_user_error EXCEPTION
    FOR SQLSTATE '99999';
    IF new_employee.birth_date > 'June 6, 2001' THEN
      SIGNAL err_user_error;
    END IF;
  END
```

This trigger fires after any row is inserted into the *employee* table. It detects and disallows any new rows that correspond to birth dates later than June 6, 2001.

The phrase REFERENCING NEW AS *new\_employee* allows statements in the trigger code to refer to the data in the new row using the alias *new\_employee*.

Signaling an error causes the triggering statement, as well as any previous effects of the trigger, to be undone.

For an INSERT statement that adds many rows to the employee table, the *check\_birth\_date* trigger fires once for each new row. If the trigger fails for any of the rows, all effects of the INSERT statement roll back.

You can specify that the trigger fires before the row is inserted rather than after by changing the first line of the example to:

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

The REFERENCING NEW clause refers to the inserted values of the row; it is independent of the timing (BEFORE or AFTER) of the trigger.

You may find it easier in some cases to enforce constraints using declaration referential integrity or CHECK constraints, rather than triggers. For example, implementing the above example with a column check constraint proves more efficient and concise:

```
CHECK (@col <= 'June 6, 2001')
```

**Example 2: A row-level DELETE trigger example**

The following CREATE TRIGGER statement defines a row-level DELETE trigger:

```
CREATE TRIGGER mytrigger BEFORE DELETE ON employee
```

```
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
    . . .
END
```

The `REFERENCING OLD` clause enables the delete trigger code to refer to the values in the row being deleted using the alias *oldtable*.

You can specify that the trigger fires after the row is deleted rather than before, by changing the first line of the example to:

```
CREATE TRIGGER check_birth_date AFTER DELETE ON employee
```

The `REFERENCING OLD` clause is independent of the timing (`BEFORE` or `AFTER`) of the trigger.

### Example 3: A statement-level UPDATE trigger example

The following `CREATE TRIGGER` statement is appropriate for statement-level `UPDATE` triggers:

```
CREATE TRIGGER mytrigger AFTER UPDATE ON employee
REFERENCING NEW AS table_after_update
                OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
    . . .
END
```

The `REFERENCING NEW` and `REFERENCING OLD` clause allows the `UPDATE` trigger code to refer to both the old and new values of the rows being updated. The table alias *table\_after\_update* refers to columns in the new row and the table alias *table\_before\_update* refers to columns in the old row.

The `REFERENCING NEW` and `REFERENCING OLD` clause has a slightly different meaning for statement-level and row-level triggers. For statement-level triggers the `REFERENCING OLD` or `NEW` aliases are table aliases, while in row-level triggers they refer to the row being altered.

☞ For more information, see "CREATE TRIGGER statement" on page 362 of the book *ASA SQL Reference Manual*, and "Using compound statements" on page 533.

## Executing triggers

Triggers execute automatically whenever an `INSERT`, `UPDATE`, or `DELETE` operation is performed on the table named in the trigger. A row-level trigger fires once for each row affected, while a statement-level trigger fires once for the entire statement.

When an INSERT, UPDATE, or DELETE fires a trigger, the order of operation is as follows:

- 1 BEFORE triggers fire.
- 2 Referential actions are performed.
- 3 The operation itself is performed.
- 4 AFTER triggers fire.


If any of the steps encounter an error not handled within a procedure or trigger, the preceding steps are undone, the subsequent steps are not performed, and the operation that fired the trigger fails.

## Altering triggers

You can modify an existing trigger using either Sybase Central or Interactive SQL. You must be the owner of the table on which the trigger is defined, or be DBA, or have ALTER permissions on the table and have RESOURCE authority.

In Sybase Central, you cannot rename an existing trigger directly. Instead, you must create a new trigger with the new name, copy the previous code to it, and then delete the old trigger.

Alternatively, you can use an ALTER TRIGGER statement to modify an existing trigger. You must include the entire new trigger in this statement (in the same syntax as in the CREATE TRIGGER statement that created the trigger).

 For more information on altering database object properties, see "Setting properties for database objects" on page 34.

### ❖ To alter the code of a trigger (Sybase Central):

- 1 Open the Triggers folder of the desired table.
- 2 Right-click the desired trigger.
- 3 From the popup menu, do one of the following:
  - ◆ Choose Open as Watcom SQL to edit the code in the Watcom SQL dialect.
  - ◆ Choose Open as Transact SQL to edit the code in the Transact SQL dialect.
- 4 In the Code Editor, edit the trigger's code.
- 5 To execute the code in the database, choose File►Save/Execute in Database.



❖ **To alter the code of a trigger (SQL):**

- 1 Connect to the database.
- 2 Execute an ALTER TRIGGER statement. Include the entire new trigger in this statement.

🔗 For more information, see "ALTER TRIGGER statement" on page 240 of the book *ASA SQL Reference Manual*.

## Dropping triggers

Once you create a trigger, it remains in the database until someone explicitly removes it. You must have ALTER permissions on the table associated with the trigger to drop the trigger.

❖ **To delete a trigger (Sybase Central):**

- 1 Open the Triggers folder of the desired table.
- 2 Right-click the desired trigger and choose Delete from the popup menu.

❖ **To delete a trigger (SQL):**

- 1 Connect to a database.
- 2 Execute a DROP TRIGGER statement.

**Example**

The following statement removes the trigger *mytrigger* from the database:

```
DROP TRIGGER mytrigger
```

🔗 For more information, see "DROP statement" on page 397 of the book *ASA SQL Reference Manual*.

## Trigger execution permissions

You cannot grant permissions to execute a trigger, since users cannot execute triggers: Adaptive Server Anywhere fires them in response to actions on the database. Nevertheless, a trigger does have permissions associated with it as it executes, defining its right to carry out certain actions.

Triggers execute using the permissions of the owner of the table on which they are defined, not the permissions of the user who caused the trigger to fire, and not the permissions of the user who created the trigger.

When a trigger refers to a table, it uses the group memberships of the table creator to locate tables with no explicit owner name specified. For example, if a trigger on *user\_1.Table\_A* references *Table\_B* and does not specify the owner of *Table\_B*, then either *Table\_B* must have been created by *user\_1* or *user\_1* must be a member of a group (directly or indirectly) that is the owner of *Table\_B*. If neither condition is met, a table not found message results when the trigger fires.

Also, *user\_1* must have permissions to carry out the operations specified in the trigger.

## Introduction to batches

A simple batch consists of a set of SQL statements, separated by semicolons or separated by a separate line with just the word **go** on it. The use of **go** is recommended. For example, the following set of statements form a batch, which creates an Eastern Sales department and transfers all sales reps from Massachusetts to that department.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 220, 'Eastern Sales' )
go
UPDATE employee
SET dept_id = 220
WHERE dept_id = 200
AND state = 'MA'
go
COMMIT
go
```

You can include this set of statements in an application and execute them together.

### Interactive SQL and batches

Interactive SQL parses a list of semicolon-separated statements, such as the above, before sending them to the server. In this case, Interactive SQL sends each statement to the server individually, not as a batch. Unless you have such parsing code in your application, the statements would be sent and treated as a batch. Putting a BEGIN and END around a set of statements causes Interactive SQL to treat them as a batch.

Many statements used in procedures and triggers can also be used in batches. You can use control statements (CASE, IF, LOOP, and so on), including compound statements (BEGIN and END), in batches. Compound statements can include declarations of variables, exceptions, temporary tables, or cursors inside the compound statement.

The following batch creates a table only if a table of that name does not already exist:

```
IF NOT EXISTS (
  SELECT * FROM SYSTABLE
  WHERE table_name = 't1' ) THEN
  CREATE TABLE t1 (
    firstcol INT PRIMARY KEY,
    secondcol CHAR( 30 )
  )
go
```

```
ELSE  
    MESSAGE 'Table t1 already exists' TO CLIENT;  
END IF
```

If you run this batch twice from Interactive SQL, it creates the table the first time you run it and displays the message in the Interactive SQL Messages pane the next time you run it.

## Control statements

There are a number of control statements for logical flow and decision making in the body of the procedure or trigger, or in a batch. Available control statements include:

Control statement	Syntax
Compound statements  ☞ For more information, see "BEGIN statement" on page 248 of the book <i>ASA SQL Reference Manual</i> .	<pre>BEGIN [ ATOMIC ]     Statement-list END</pre>
Conditional execution: IF  ☞ For more information, see "IF statement" on page 454 of the book <i>ASA SQL Reference Manual</i> .	<pre>IF condition THEN     Statement-list ELSEIF condition THEN     Statement-list ELSE     Statement-list END IF</pre>
Conditional execution: CASE  ☞ For more information, see "CASE statement" on page 256 of the book <i>ASA SQL Reference Manual</i> .	<pre>CASE expression WHEN value THEN     Statement-list WHEN value THEN     Statement-list ELSE     Statement-list END CASE</pre>
Repetition: WHILE, LOOP  ☞ For more information, see "LOOP statement" on page 481 of the book <i>ASA SQL Reference Manual</i> .	<pre>WHILE condition LOOP     Statement-list END LOOP</pre>
Repetition: FOR cursor loop  ☞ For more information, see "FOR statement" on page 429 of the book <i>ASA SQL Reference Manual</i> .	<pre>FOR loop-name     AS cursor-name     CURSOR FOR select statement DO     Statement-list END FOR</pre>
Break: LEAVE  ☞ For more information, see "LEAVE statement" on page 469 of the book <i>ASA SQL Reference Manual</i> .	<pre>LEAVE label</pre>
CALL  ☞ For more information, see "CALL statement" on page 254 of the book <i>ASA SQL Reference Manual</i> .	<pre>CALL procname( arg, ... )</pre>

For more information about each statement, see the entries in "SQL Statements" on page 199 of the book *ASA SQL Reference Manual*

## Using compound statements

A compound statement starts with the keyword **BEGIN** and concludes with the keyword **END**. The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in batches. Compound statements can be nested, and combined with other control statements to define execution flow in procedures and triggers or in batches.

A compound statement allows a set of SQL statements to be grouped together and treated as a unit. Delimit SQL statements within a compound statement with semicolons.

For more information about compound statements, see the "BEGIN statement" on page 248 of the book *ASA SQL Reference Manual*.

## Declarations in compound statements

Local declarations in a compound statement immediately follow the **BEGIN** keyword. These local declarations exist only within the compound statement. Within a compound statement you can declare:

- ◆ Variables
- ◆ Cursors
- ◆ Temporary tables
- ◆ Exceptions (error identifiers)

Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures called from the compound statement.

## Atomic compound statements

An **atomic** statement is a statement executed completely or not at all. For example, an **UPDATE** statement that updates thousands of rows might encounter an error after updating many rows. If the statement does not complete, all changes revert back to their original state. The **UPDATE** statement is atomic.

All non-compound SQL statements are atomic. You can make a compound statement atomic by adding the keyword `ATOMIC` after the `BEGIN` keyword.


```
BEGIN ATOMIC
    UPDATE employee
    SET manager_ID = 501
    WHERE emp_ID = 467;
    UPDATE employee
    SET birth_date = 'bad_data';
END
```

In this example, the two update statements are part of an atomic compound statement. They must either succeed or fail as one. The first update statement would succeed. The second one causes a data conversion error since the value being assigned to the *birth\_date* column cannot be converted to a date.

The atomic compound statement fails and the effect of both `UPDATE` statements is undone. Even if the currently executing transaction is eventually committed, neither statement in the atomic compound statement takes effect.

You cannot use `COMMIT` and `ROLLBACK` and some `ROLLBACK TO SAVEPOINT` statements within an atomic compound statement (see "Transactions and savepoints in procedures and triggers" on page 558).

There is a case where some, but not all, of the statements within an atomic compound statement are executed. This happens when an exception handler within the compound statement deals with an error.

 For more information, see "Using exception handlers in procedures and triggers" on page 553.



## The structure of procedures and triggers

The body of a procedure or trigger consists of a compound statement as discussed in "Using compound statements" on page 533. A compound statement consists of a BEGIN and an END, enclosing a set of SQL statements. Semicolons delimit each statement.

### SQL statements allowed in procedures and triggers

You can use almost all SQL statements within procedures and triggers, including the following:

- ◆ SELECT, UPDATE, DELETE, INSERT and SET VARIABLE.
- ◆ The CALL statement to execute other procedures.
- ◆ Control statements (see "Control statements" on page 531).
- ◆ Cursor statements (see "Using cursors in procedures and triggers" on page 545).
- ◆ Exception handling statements (see "Using exception handlers in procedures and triggers" on page 553).
- ◆ The EXECUTE IMMEDIATE statement.

Some SQL statements you cannot use within procedures and triggers include:

- ◆ CONNECT statement
- ◆ DISCONNECT statement.

You can use COMMIT, ROLLBACK and SAVEPOINT statements within procedures and triggers with certain restrictions (see "Transactions and savepoints in procedures and triggers" on page 558).

☞ For more information, see the *Usage* for each SQL statement in the chapter "SQL Statements" on page 199 of the book *ASA SQL Reference Manual*.

## Declaring parameters for procedures

Procedure parameters appear as a list in the CREATE PROCEDURE statement. Parameter names must conform to the rules for other database identifiers such as column names. They must have valid data types (see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*), and must be prefixed with one of the keywords IN, OUT or INOUT. These keywords have the following meanings:

- ◆ **IN** The argument is an expression that provides a value to the procedure.
- ◆ **OUT** The argument is a variable that could be given a value by the procedure.
- ◆ **INOUT** The argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

You can assign default values to procedure parameters in the CREATE PROCEDURE statement. The default value must be a constant, which may be NULL. For example, the following procedure uses the NULL default for an IN parameter to avoid executing a query that would have no meaning:

```
CREATE PROCEDURE
CustomerProducts( IN customer_id
                  INTEGER DEFAULT NULL )
RESULT ( product_id INTEGER,
         quantity_ordered INTEGER )
BEGIN
  IF customer_id IS NULL THEN
    RETURN;
  ELSE
    SELECT product.id,
           sum( sales_order_items.quantity )
    FROM   product,
           sales_order_items,
           sales_order
    WHERE  sales_order.cust_id = customer_id
    AND    sales_order.id = sales_order_items.id
    AND    sales_order_items.prod_id = product.id
    GROUP BY product.id;
  END IF;
END
```

The following statement assigns the DEFAULT NULL, and the procedure RETURNS instead of executing the query.

```
CALL customerproducts( );
```

## Passing parameters to procedures

You can take advantage of default values of stored procedure parameters with either of two forms of the `CALL` statement.

If the optional parameters are at the end of the argument list in the `CREATE PROCEDURE` statement, they may be omitted from the `CALL` statement. As an example, consider a procedure with three `INOUT` parameters:

```
CREATE PROCEDURE SampleProc( INOUT var1 INT
                             DEFAULT 1,
                             INOUT var2 int DEFAULT 2,
                             INOUT var3 int DEFAULT 3 )
...
```

We assume that the calling environment has set up three variables to hold the values passed to the procedure:

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

The procedure *SampleProc* may be called supplying only the first parameter as follows:

```
CALL SampleProc( V1 )
```

in which case the default values are used for *var2* and *var3*.

A more flexible method of calling procedures with optional arguments is to pass the parameters by name. The *SampleProc* procedure may be called as follows:

```
CALL SampleProc( var1 = V1, var3 = V3 )
```

or as follows:

```
CALL SampleProc( var3 = V3, var1 = V1 )
```

## Passing parameters to functions

User-defined functions are not invoked with the `CALL` statement, but are used in the same manner that built-in functions are. For example, the following statement uses the *fullname* function defined in "Creating user-defined functions" on page 518 to retrieve the names of employees:

### ❖ To list the names of all employees:

- ◆ Type the following:

```
SELECT fullname(emp_fname, emp_lname) AS Name
FROM employee
```

**Name**

---

Fran Whitney

Matthew Cobb

Philip Chin

Julie Jordan


...

**Notes**

- ◆ Default parameters can be used in calling functions. However, parameters cannot be passed to functions by name.
- ◆ Parameters are passed by value, not by reference. Even if the function changes the value of the parameter, this change is not returned to the calling environment.
- ◆ Output parameters cannot be used in user-defined functions.
- ◆ User-defined functions cannot return result sets.

## Returning results from procedures

Procedures can return results in the form of a single row of data, or multiple rows. Results consisting of a single row of data can be passed back as arguments to the procedure. Results consisting of multiple rows of data are passed back as result sets. Procedures can also return a single value given in the RETURN statement.

 For simple examples of how to return results from procedures, see "Introduction to procedures" on page 511. For more detailed information, see the following sections.

### Returning a value using the RETURN statement

The RETURN statement returns a single integer value to the calling environment, causing an immediate exit from the procedure. The RETURN statement takes the form:

```
RETURN expression
```

The value of the supplied expression is returned to the calling environment. To save the return value in a variable, use an extension of the CALL statement:

```
CREATE VARIABLE returnval INTEGER ;  
returnval = CALL myproc() ;
```

### Returning results as procedure parameters

Procedures can return results to the calling environment in the parameters to the procedure.

Within a procedure, parameters and variables can be assigned values using:

- ◆ the SET statement.
- ◆ a SELECT statement with an INTO clause.

#### Using the SET statement

The following somewhat artificial procedure returns a value in an OUT parameter assigned using a SET statement:

```
CREATE PROCEDURE greater (    IN a INT,  
                             IN b INT,  
                             OUT c INT)  
  
BEGIN  
    IF a > b THEN  
        SET c = a;
```

```
ELSE
    SET c = b;
END IF ;
END
```

### Using single-row SELECT statements

Single-row queries retrieve at most one row from the database. This type of query uses a **SELECT** statement with an **INTO** clause. The **INTO** clause follows the select list and precedes the **FROM** clause. It contains a list of variables to receive the value for each select list item. There must be the same number of variables as there are select list items.

When a **SELECT** statement executes, the server retrieves the results of the **SELECT** statement and places the results in the variables. If the query results contain more than one row, the server returns an error. For queries returning more than one row, you must use cursors. For information about returning more than one row from a procedure, see "Returning result sets from procedures" on page 541.

If the query results in no rows being selected, a **row not found** warning appears.

The following procedure returns the results of a single-row **SELECT** statement in the procedure parameters.

#### ❖ To return the number of orders placed by a given customer:

- ◆ Type the following:

```
CREATE PROCEDURE OrderCount (IN customer_ID INT,
                             OUT Orders INT)
BEGIN
    SELECT COUNT(DBA.sales_order.id)
        INTO Orders
    FROM DBA.customer
        KEY LEFT OUTER JOIN "DBA".sales_order
        WHERE DBA.customer.id = customer_ID;
END
```

You can test this procedure in Interactive SQL using the following statements, which show the number of orders placed by the customer with ID 102:

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

### Notes

- ◆ The *customer\_ID* parameter is declared as an **IN** parameter. This parameter holds the customer ID passed in to the procedure.
- ◆ The *Orders* parameter is declared as an **OUT** parameter. It holds the value of the orders variable that returned to the calling environment.

- ◆ No DECLARE statement is necessary for the *Orders* variable, as it is declared in the procedure argument list.
- ◆ The SELECT statement returns a single row and places it into the variable *Orders*.

## Returning result sets from procedures

Result sets allow a procedure to return more than one row of results to the calling environment.

The following procedure returns a list of customers who have placed orders, together with the total value of the orders placed. The procedure does not list customers who have not placed orders.

```
CREATE PROCEDURE ListCustomerValue ()
RESULT ("Company" CHAR(36), "Value" INT)
BEGIN
    SELECT company_name,
           CAST( sum(    sales_order_items.quantity *
                        product.unit_price)
                AS INTEGER ) AS value
    FROM customer
      INNER JOIN sales_order
      INNER JOIN sales_order_items
      INNER JOIN product
    GROUP BY company_name
    ORDER BY value DESC;
END
```

- ◆ Type the following:

```
CALL ListCustomerValue ()
```

Company	Value
Chadwicks	8076
Overland Army Navy	8064
Martins Landing	6888
Sterling & Co.	6804
Carmel Industries	6780
...	...

### Notes

- ◆ The number of variables in the RESULT list must match the number of the SELECT list items. Automatic data type conversion is carried out where possible if data types do not match.

- ◆ The **RESULT** clause is part of the **CREATE PROCEDURE** statement, and does not have a command delimiter.
- ◆ The names of the **SELECT** list items do not need to match those of the **RESULT** list.
- ◆ When testing this procedure, Interactive SQL displays only the first result set by default. You can configure Interactive SQL to display more than one result set by setting the **Show multiple result sets** option on the **Results** tab of the **Options** dialog.
- ◆ You can modify procedure result sets, unless they are generated from a view. The user calling the procedure requires the appropriate permissions on the underlying table to modify procedure results. This is different than the usual permissions for procedure execution, where the procedure *owner* must have permissions on the table.

☞ For information about modifying result sets in Interactive SQL, see "Editing table values in Interactive SQL" on page 84 of the book *ASA Getting Started*.

## Returning multiple result sets from procedures

Before Interactive SQL can return multiple result sets, you need to enable this option on the **Results** tab of the **Options** dialog. By default, this option is disabled. If you change the setting, it takes effect in newly created connections (such as new windows).

### ❖ To enable multiple result set functionality:

- 1 Choose **Tools**►**Options**.
- 2 In the resulting **Options** dialog, click the **Results** tab.
- 3 Select the **Show Multiple Result Sets** checkbox.

After you enable this option, a procedure can return more than one result set to the calling environment. If a **RESULT** clause is employed, the result sets must be compatible: they must have the same number of items in the **SELECT** lists, and the data types must all be of types that can be automatically converted to the data types listed in the **RESULT** list.

The following procedure lists the names of all employees, customers, and contacts listed in the database:

```
CREATE PROCEDURE ListPeople()  
RESULT ( lname CHAR(36), fname CHAR(36) )  
BEGIN  
    SELECT emp_lname, emp_fname  
    FROM employee;
```



```
SELECT lname, fname
FROM customer;
SELECT last_name, first_name
FROM contact;
END
```

## Notes

- ◆ To test this procedure in Interactive SQL, enter the following statement in the SQL Statements pane:

```
CALL ListPeople ( )
```

## Returning variable result sets from procedures

The **RESULT** clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

If you do not use the variable result sets feature, you should use a **RESULT** clause for performance reasons.

For example, the following procedure returns two columns if the input variable is **Y**, but only one column otherwise:

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
  IF formal = 'y' THEN
    SELECT emp_lname, emp_fname
    FROM employee
  ELSE
    SELECT emp_fname
    FROM employee
  END IF
END
```

The use of variable result sets in procedures is subject to some limitations, depending on the interface used by the client application.

- ◆ **Embedded SQL** You must **DESCRIBE** the procedure call after the cursor for the result set is opened, but before any rows are returned, in order to get the proper shape of result set.

☞ For more information about the **DESCRIBE** statement, see "DESCRIBE statement [ESQL]" on page 392 of the book *ASA SQL Reference Manual*.

- ◆ **ODBC** Variable result set procedures can be used by ODBC applications. The Adaptive Server Anywhere ODBC driver carries out the proper description of the variable result sets.

- ◆ **Open Client applications** Open Client applications can use variable result set procedures. Adaptive Server Anywhere carries out the proper description of the variable result sets.

## Using cursors in procedures and triggers


Cursors retrieve rows one at a time from a query or stored procedure with multiple rows in its result set. A cursor is a handle or an identifier for the query or procedure, and for a current position within the result set.

### Cursor management overview

Managing a cursor is similar to managing a file in a programming language. The following steps manage cursors:

- 1 Declare a cursor for a particular `SELECT` statement or procedure using the `DECLARE` statement.
- 2 Open the cursor using the `OPEN` statement.
- 3 Use the `FETCH` statement to retrieve results one row at a time from the cursor.
- 4 The warning `Row Not Found` signals the end of the result set.
- 5 Close the cursor using the `CLOSE` statement.

By default, cursors are automatically closed at the end of a transaction (on `COMMIT` or `ROLLBACK` statements). Cursors are opened using the `WITH HOLD` clause will stay open for subsequent transactions until someone explicitly closes them.

 For more information on positioning cursors, see "Cursor positioning" on page 19 of the book *ASA Programming Guide*.

### Using cursors on `SELECT` statements in procedures

The following procedure uses a cursor on a `SELECT` statement. Based on the same query used in the *ListCustomerValue* procedure described in "Returning result sets from procedures" on page 541, it illustrates several features of the stored procedure language.

```
CREATE PROCEDURE TopCustomerValue
(   OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
  -- 1. Declare the "error not found" exception
  DECLARE err_notfound
    EXCEPTION FOR SQLSTATE '02000';
  -- 2. Declare variables to hold
  --     each company name and its value
```

```
DECLARE ThisName CHAR(36);
DECLARE ThisValue INT;
-- 3. Declare the cursor ThisCompany
--     for the query
DECLARE ThisCompany CURSOR FOR
SELECT company_name,
       CAST( sum( sales_order_items.quantity *
                  product.unit_price ) AS INTEGER )
       AS value
FROM customer
   INNER JOIN sales_order
   INNER JOIN sales_order_items
   INNER JOIN product
GROUP BY company_name;
-- 4. Initialize the values of TopValue
SET TopValue = 0;
-- 5. Open the cursor
OPEN ThisCompany;
-- 6. Loop over the rows of the query
CompanyLoop:
LOOP
    FETCH NEXT ThisCompany
    INTO ThisName, ThisValue;
    IF SQLSTATE = err_notfound THEN
        LEAVE CompanyLoop;
    END IF;
    IF ThisValue > TopValue THEN
        SET TopCompany = ThisName;
        SET TopValue = ThisValue;
    END IF;
END LOOP CompanyLoop;
-- 7. Close the cursor
CLOSE ThisCompany;
END
```

## Notes

The *TopCustomerValue* procedure has the following notable features:

- ◆ The "error not found" exception is declared. This exception signals, later in the procedure, when a loop over the results of a query completes.

☞ For more information about exceptions, see "Errors and warnings in procedures and triggers" on page 548.

- ◆ Two local variables *ThisName* and *ThisValue* are declared to hold the results from each row of the query.
- ◆ The cursor *ThisCompany* is declared. The SELECT statement produces a list of company names and the total value of the orders placed by that company.
- ◆ The value of *TopValue* is set to an initial value of 0, for later use in the loop.

- ◆ The *ThisCompany* cursor opens.
- ◆ The LOOP statement loops over each row of the query, placing each company name in turn into the variables *ThisName* and *ThisValue*. If *ThisValue* is greater than the current top value, *TopCompany* and *TopValue* are reset to *ThisName* and *ThisValue*.
- ◆ The cursor closes at the end of the procedure.
- ◆ You can also write this procedure without a loop by adding an ORDER BY value DESC clause to the SELECT statement. Then, only the first row of the cursor needs to be fetched.

The LOOP construct in the *TopCompanyValue* procedure is a standard form, exiting after the last row processes. You can rewrite this procedure in a more compact form using a FOR loop. The FOR statement combines several aspects of the above procedure into a single statement.

```
CREATE PROCEDURE TopCustomerValue2(
    OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
    -- Initialize the TopValue variable
    SET TopValue = 0;
    -- Do the For Loop
    FOR CompanyFor AS ThisCompany
    CURSOR FOR
        SELECT company_name AS ThisName ,
            CAST( sum( sales_order_items.quantity *
                product.unit_price ) AS INTEGER )
            AS ThisValue
        FROM customer
            INNER JOIN sales_order
            INNER JOIN sales_order_items
            INNER JOIN product
        GROUP BY ThisName
    DO
        IF ThisValue > TopValue THEN
            SET TopCompany = ThisName;
            SET TopValue = ThisValue;
        END IF;
    END FOR;
END
```

## Errors and warnings in procedures and triggers

After an application program executes a SQL statement, it can examine a **status code**. This status code (or return code) indicates whether the statement executed successfully or failed and gives the reason for the failure. You can use the same mechanism to indicate the success or failure of a CALL statement to a procedure.

Error reporting uses either the SQLCODE or SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see "Database Error Messages" on page 1 of the book *ASA Errors Manual*. Whenever a SQL statement executes, a value appears in special procedure variables called SQLSTATE and SQLCODE. That value indicates whether or not there were any unusual conditions encountered while the statement was being performed. You can check the value of SQLSTATE or SQLCODE in an IF statement following a SQL statement, and take actions depending on whether the statement succeeded or failed.

For example, the SQLSTATE variable can be used to indicate if a row is successfully fetched. The *TopCustomerValue* procedure presented in section "Using cursors on SELECT statements in procedures" on page 545 used the SQLSTATE test to detect that all rows of a SELECT statement had been processed.

## Default error handling in procedures and triggers

This section describes how Adaptive Server Anywhere handles errors that occur during a procedure execution, if you have no error handling built in to the procedure.

✍ For different behavior, you can use exception handlers, described in "Using exception handlers in procedures and triggers" on page 553. Warnings are handled in a slightly different manner from errors: for a description, see "Default handling of warnings in procedures and triggers" on page 552.

There are two ways of handling errors without using explicit error handling:

- ◆ **Default error handling** The procedure or trigger fails and returns an error code to the calling environment.
- ◆ **ON EXCEPTION RESUME** If the ON EXCEPTION RESUME clause appears in the CREATE PROCEDURE statement, the procedure carries on executing after an error, resuming at the statement following the one causing the error.

✍ The precise behavior for procedures that use `ON EXCEPTION RESUME` is dictated by the `ON_TSQL_ERROR` option setting. For more information, see "ON\_TSQL\_ERROR option" on page 587 of the book *ASA Database Administration Guide*.

## Default error handling

Generally, if a SQL statement in a procedure or trigger fails, the procedure or trigger terminates execution and control returns to the application program with an appropriate setting for the `SQLSTATE` and `SQLCODE` values. This is true even if the error occurred in a procedure or trigger invoked directly or indirectly from the first one. In the case of a trigger, the operation causing the trigger is also undone and the error is returned to the application.

The following demonstration procedures show what happens when an application calls the procedure *OuterProc*, and *OuterProc* in turn calls the procedure *InnerProc*, which then encounters an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.' TO CLIENT;
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in OuterProc.' TO CLIENT
END
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in InnerProc.' TO CLIENT;
END
```

## Notes

- ◆ The `DECLARE` statement in *InnerProc* declares a symbolic name for one of the predefined `SQLSTATE` values associated with error conditions already known to the server.
- ◆ The `MESSAGE` statement sends a message to the Interactive SQL Messages pane.
- ◆ The `SIGNAL` statement generates an error condition from within the *InnerProc* procedure.

The following statement executes the *OuterProc* procedure:

```
CALL OuterProc();
```

The Interactive SQL Messages pane displays the following:

Hello from OuterProc.

Hello from InnerProc.

None of the statements following the SIGNAL statement in *InnerProc* execute: *InnerProc* immediately passes control back to the calling environment, which in this case is the procedure *OuterProc*. None of the statements following the CALL statement in *OuterProc* execute. The error condition returns to the calling environment to be handled there. For example, Interactive SQL handles the error by displaying a message window describing the error.

The TRACEBACK function provides a list of the statements that were executing when the error occurred. You can use the TRACEBACK function from Interactive SQL by typing the following statement:

```
SELECT TRACEBACK( * )
```

## Error handling with ON EXCEPTION RESUME

If the ON EXCEPTION RESUME clause appears in the CREATE PROCEDURE statement, the procedure checks the following statement when an error occurs. If the statement handles the error, then the procedure continues executing, resuming at the statement after the one causing the error. It does not return control to the calling environment when an error occurred.

☞ The behavior for procedures that use ON EXCEPTION RESUME can be modified by the ON\_TSQL\_ERROR option setting. For more information, see "ON\_TSQL\_ERROR option" on page 587 of the book *ASA Database Administration Guide*.

Error-handling statements include the following:

- ◆ IF
- ◆ SELECT @variable =
- ◆ CASE
- ◆ LOOP
- ◆ LEAVE
- ◆ CONTINUE
- ◆ CALL
- ◆ EXECUTE
- ◆ SIGNAL
- ◆ RESIGNAL
- ◆ DECLARE



## ◆ SET VARIABLE

The following example illustrates how this works.

Drop the  
procedures

Remember to drop both the *InnerProc* and *OuterProc* procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc
```

The following demonstration procedures show what happens when an application calls the procedure *OuterProc*; and *OuterProc* in turn calls the procedure *InnerProc*, which then encounters an error. These demonstration procedures are based on those used earlier in this section:

```
CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
    DECLARE res CHAR(5);
    MESSAGE 'Hello from OuterProc.' TO CLIENT;
    CALL InnerProc();
    SELECT @res=SQLSTATE;
    IF res='52003' THEN
        MESSAGE 'SQLSTATE set to ',
            res, ' in OuterProc.' TO CLIENT;
    END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in InnerProc.' TO CLIENT;
END
```

The following statement executes the *OuterProc* procedure:

```
CALL OuterProc();
```

The Interactive SQL Messages pane then displays the following:

Hello from OuterProc.

Hello from InnerProc.

SQLSTATE set to 52003 in OuterProc.

The execution path is as follows:

- 1 OuterProc executes and calls InnerProc.

- 2 In *InnerProc*, the `SIGNAL` statement signals an error.
- 3 The `MESSAGE` statement is not an error-handling statement, so control is passed back to *OuterProc* and the message is not displayed.
- 4 In *OuterProc*, the statement following the error assigns the `SQLSTATE` value to the variable named **res**. This is an error-handling statement, and so execution continues and the *OuterProc* message appears.

## Default handling of warnings in procedures and triggers

Errors and warnings are handled differently. While the default action for errors is to set a value for the `SQLSTATE` and `SQLCODE` variables, and return control to the calling environment in the event of an error, the default action for warnings is to set the `SQLSTATE` and `SQLCODE` values and continue execution of the procedure.

Drop the  
procedures

Remember to drop both the *InnerProc* and *OuterProc* procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc
```

The following demonstration procedures illustrate default handling of warnings. These demonstration procedures are based on those used in "Default error handling in procedures and triggers" on page 548. In this case, the `SIGNAL` statement generates a row not found condition, which is a warning rather than an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.' TO CLIENT;
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in OuterProc.' TO CLIENT;
END
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE row_not_found
        EXCEPTION FOR SQLSTATE '02000';
    MESSAGE 'Hello from InnerProc.' TO CLIENT;
    SIGNAL row_not_found;
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in InnerProc.' TO CLIENT;
END
```

The following statement executes the *OuterProc* procedure:

```
CALL OuterProc();
```

The Interactive SQL Messages pane then displays the following:

```
Hello from OuterProc.  
Hello from InnerProc.  
SQLSTATE set to 02000 in InnerProc.  
SQLSTATE set to 00000 in OuterProc.
```

The procedures both continued executing after the warning was generated, with SQLSTATE set by the warning (02000).

Execution of the second MESSAGE statement in InnerProc resets the warning. Successful execution of any SQL statement resets SQLSTATE to 00000 and SQLCODE to 0. If a procedure needs to save the error status, it must do an assignment of the value immediately after execution of the statement which caused the error warning.

## Using exception handlers in procedures and triggers

It is often desirable to intercept certain types of errors and handle them within a procedure or trigger, rather than pass the error back to the calling environment. This is done through the use of an **exception handler**.

You define an exception handler with the EXCEPTION part of a compound statement (see "Using compound statements" on page 533). Whenever an error occurs in the compound statement, the exception handler executes. Unlike errors, warnings do not cause exception handling code to be executed. Exception handling code also executes if an error appears in a nested compound statement or in a procedure or trigger invoked anywhere within the compound statement.

Drop the  
procedures

Remember to drop both the *InnerProc* and *OuterProc* procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;  
DROP PROCEDURE InnerProc
```

The demonstration procedures used to illustrate exception handling are based on those used in "Default error handling in procedures and triggers" on page 548. In this case, additional code handles the column not found error in the *InnerProc* procedure.

```
CREATE PROCEDURE OuterProc()  
BEGIN  
    MESSAGE 'Hello from OuterProc.' TO CLIENT;  
    CALL InnerProc();  
    MESSAGE 'SQLSTATE set to ',  
        SQLSTATE, ' in OuterProc.' TO CLIENT  
END  
CREATE PROCEDURE InnerProc()
```

```
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL.' TO CLIENT;
    EXCEPTION
        WHEN column_not_found THEN
            MESSAGE 'Column not found handling.' TO
CLIENT;
        WHEN OTHERS THEN
            RESIGNAL ;
END
```

The `EXCEPTION` statement declares the exception handler itself. The lines following the `EXCEPTION` statement do not execute unless an error occurs. Each `WHEN` clause specifies an exception name (declared with a `DECLARE` statement) and the statement or statements to be executed in the event of that exception. The `WHEN OTHERS THEN` clause specifies the statement(s) to be executed when the exception that occurred does not appear in the preceding `WHEN` clauses.

In this example, the statement `RESIGNAL` passes the exception on to a higher-level exception handler. `RESIGNAL` is the default action if `WHEN OTHERS THEN` is not specified in an exception handler.

The following statement executes the *OuterProc* procedure:

```
CALL OuterProc();
```

The Interactive SQL Messages pane then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

### Notes

- ◆ The `EXCEPTION` statements execute, rather than the lines following the `SIGNAL` statement in *InnerProc*.
- ◆ As the error encountered was a column not found error, the `MESSAGE` statement included to handle the error executes, and `SQLSTATE` resets to zero (indicating no errors).
- ◆ After the exception handling code executes, control passes back to *OuterProc*, which proceeds as if no error was encountered.
- ◆ You should not use `ON EXCEPTION RESUME` together with explicit exception handling. The exception handling code is not executed if `ON EXCEPTION RESUME` is included.

- ◆ If the error handling code for the column not found exception is simply a `RESIGNAL` statement, control passes back to the *OuterProc* procedure with `SQLSTATE` still set at the value 52003. This is just as if there were no error handling code in *InnerProc*. Since there is no error handling code in *OuterProc*, the procedure fails.

#### Exception handling and atomic compound statements

When an exception is handled inside a compound statement, the compound statement completes without an active exception and the changes before the exception are not reversed. This is true even for atomic compound statements. If an error occurs within an atomic compound statement and is explicitly handled, some but not all of the statements in the atomic compound statement are executed.

## Nested compound statements and exception handlers

The code following a statement that causes an error executes only if an `ON EXCEPTION RESUME` clause appears in a procedure definition.

You can use nested compound statements to give you more control over which statements execute following an error and which do not.

#### Drop the procedures

Remember to drop both the *InnerProc* and *OuterProc* procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc
```

The following demonstration procedure illustrates how nested compound statements can be used to control flow. The procedure is based on that used as an example in "Default error handling in procedures and triggers" on page 548.

```
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE column_not_found
    EXCEPTION FOR SQLSTATE VALUE '52003';
    MESSAGE 'Hello from InnerProc' TO CLIENT;
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL' TO CLIENT
    EXCEPTION
    WHEN column_not_found THEN
        MESSAGE 'Column not found handling' TO
        CLIENT;
    WHEN OTHERS THEN
        RESIGNAL;
END;
MESSAGE 'Outer compound statement' TO CLIENT;
```

END

The following statement executes the *InnerProc* procedure:

```
CALL InnerProc();
```

The Interactive SQL Messages pane then displays the following:

```
Hello from InnerProc  
Column not found handling  
Outer compound statement
```

When the `SIGNAL` statement that causes the error is encountered, control passes to the exception handler for the compound statement, and the `Column not found handling` message prints. Control then passes back to the outer compound statement and the `Outer compound statement` message prints.

If an error other than `column not found` is encountered in the inner compound statement, the exception handler executes the `RESIGNAL` statement. The `RESIGNAL` statement passes control directly back to the calling environment, and the remainder of the outer compound statement is not executed.

## Using the EXECUTE IMMEDIATE statement in procedures


The EXECUTE IMMEDIATE statement allows statements to be constructed inside procedures using a combination of literal strings (in quotes) and variables.

For example, the following procedure includes an EXECUTE IMMEDIATE statement that creates a table.

```
CREATE PROCEDURE CreateTableProc(  
    IN tablename char(30) )  
BEGIN  
    EXECUTE IMMEDIATE 'CREATE TABLE ' || tablename ||  
    '(column1 INT PRIMARY KEY)'  
END
```

In ATOMIC compound statements, you cannot use an EXECUTE IMMEDIATE statement that causes a COMMIT, as COMMITs are not allowed in that context.

The EXECUTE IMMEDIATE statement does not support statements or queries that return result sets.

 For more information about the EXECUTE IMMEDIATE statement, see "EXECUTE statement [SP]" on page 416 of the book *ASA SQL Reference Manual*.

## Transactions and savepoints in procedures and triggers

SQL statements in a procedure or trigger are part of the current transaction (see "Using Transactions and Isolation Levels" on page 89). You can call several procedures within one transaction or have several transactions in one procedure.

COMMIT and ROLLBACK are not allowed within any atomic statement (see "Atomic compound statements" on page 533). Note that triggers are fired due to an INSERT, UPDATE, or DELETE which are atomic statements. COMMIT and ROLLBACK are not allowed in a trigger or in any procedures called by a trigger.

Savepoints (see "Savepoints within transactions" on page 93) can be used within a procedure or trigger, but a ROLLBACK TO SAVEPOINT statement can never refer to a savepoint before the atomic operation started. Also, all savepoints within an atomic operation are released when the atomic operation completes.



## Tips for writing procedures

This section provides some pointers for developing procedures.

### Check if you need to change the command delimiter

You do not need to change the command delimiter in Interactive SQL or Sybase Central when you write procedures. However, if you create and test procedures and triggers from some other browsing tool, you may need to change the command delimiter from the semicolon to another character.

Each statement within the procedure ends with a semicolon. For some browsing applications to parse the CREATE PROCEDURE statement itself, you need the command delimiter to be something other than a semicolon.

If you are using an application that requires changing the command delimiter, a good choice is to use two semicolons as the command delimiter (;;) or a question mark (?) if the system does not permit a multicharacter delimiter.

### Remember to delimit statements within your procedure

You should terminate each statement within the procedure with a semicolon. Although you can leave off semicolons for the last statement in a statement list, it is good practice to use semicolons after each statement.

The CREATE PROCEDURE statement itself contains both the RESULT specification and the compound statement that forms its body. No semicolon is needed after the BEGIN or END keywords, or after the RESULT clause.

### Use fully-qualified names for tables in procedures

If a procedure has references to tables in it, you should always preface the table name with the name of the owner (creator) of the table.

When a procedure refers to a table, it uses the group memberships of the procedure creator to locate tables with no explicit owner name specified. For example, if a procedure created by *user\_1* references *Table\_B* and does not specify the owner of *Table\_B*, then either *Table\_B* must have been created by *user\_1* or *user\_1* must be a member of a group (directly or indirectly) that is the owner of *Table\_B*. If neither condition is met, a table not found message results when the procedure is called.

You can minimize the inconvenience of long fully qualified names by using a correlation name to provide a convenient name to use for the table within a statement. Correlation names are described in "FROM clause" on page 433 of the book *ASA SQL Reference Manual*.

## Specifying dates and times in procedures

When dates and times are sent to the database from procedures, they are sent as strings. The date part of the string is interpreted according to the current setting of the DATE\_ORDER database option. As different connections may set this option to different values, some strings may be converted incorrectly to dates, or the database may not be able to convert the string to a date.

You should use the unambiguous date format *yyyy-mm-dd* or *yyyy/mm/dd* when using data strings within procedures. The server interprets these strings unambiguously as dates, regardless of the DATE\_ORDER database option setting.

☞ For more information on dates and times, see "Date and time data types" on page 65 of the book *ASA SQL Reference Manual*.

## Verifying that procedure input arguments are passed correctly

One way to verify input arguments is to display the value of the parameter in the Interactive SQL Messages pane using the MESSAGE statement. For example, the following procedure simply displays the value of the input parameter *var*:

```
CREATE PROCEDURE message_test (IN var char(40))
BEGIN
    MESSAGE var TO CLIENT;
END
```

You can also use the stored procedure debugger.

## Statements allowed in batches

All SQL statements are acceptable in batches (including data definition statements such as CREATE TABLE, ALTER TABLE, and so on), with the exception of the following:

- ◆ CONNECT or DISCONNECT statement.
- ◆ ALTER PROCEDURE or ALTER FUNCTION statement.
- ◆ CREATE TRIGGER statement.
- ◆ Interactive SQL commands such as INPUT or OUTPUT.
- ◆ You cannot use host variables in batches.

The CREATE PROCEDURE statement is allowed, but must be the final statement of the batch. Therefore a batch can contain only a single CREATE PROCEDURE statement.

## Using SELECT statements in batches

You can include one or more SELECT statements in a batch.

The following is a valid batch:

```
IF EXISTS(  SELECT *
            FROM SYSTABLE
            WHERE table_name='employee' )
THEN
    SELECT emp_lname AS LastName,
           emp_fname AS FirstName
    FROM employee;
    SELECT lname, fname
    FROM customer;
    SELECT last_name, first_name
    FROM contact;
END IF
```

The alias for the result set is necessary only in the first SELECT statement, as the server uses the first SELECT statement in the batch to describe the result set.

A RESUME statement is necessary following each query to retrieve the next result set.

## Calling external libraries from procedures

You can call a function in an external library from a stored procedure or user-defined function. You can call functions in a DLL under Windows operating systems, in an NLM under NetWare, and in a shared object on UNIX. You cannot call external functions on Windows CE.

This section describes how to use the external library calls in procedures.

External libraries called from procedures share the memory of the server. If you call a DLL from a procedure and the DLL contains memory-handling errors, you can crash the server or corrupt your database. Ensure that you thoroughly test your libraries before deploying them on production databases.

The API described in this section replaces an older API. Libraries written to the older API, used in versions before version 7.0, are still supported, but in new development you should use the new API.

Adaptive Server Anywhere includes a set of system procedures that make use of this capability, for example to send MAPI e-mail messages.

✍ For more information on system procedures, see "System Procedures and Functions" on page 683 of the book *ASA SQL Reference Manual*.

## Creating procedures and functions with external calls

This section presents some examples of procedures and functions with external calls.

### **DBA authority required**

You must have DBA authority to create procedures or functions that reference external libraries. This requirement is more strict than the RESOURCE authority required for creating other procedures or functions.

### Syntax

You can create a procedure that calls a function *function\_name* in DLL *library.dll* as follows:

```
CREATE PROCEDURE dll_proc ( parameter-list )  
EXTERNAL NAME 'function_name@library.dll'
```

If you call an external DLL from a procedure, the procedure cannot carry out any other tasks; it just forms a wrapper around the DLL.

An analogous CREATE FUNCTION statement is as follows:

```
CREATE FUNCTION dll_func ( parameter-list )  
RETURNS data-type  
EXTERNAL NAME 'function_name@library.dll'
```

In these statements, *function\_name* is the exported name of a function in the dynamic link library, and *library.dll* is the name of the library. The arguments in *parameter-list* must correspond in type and order to the argument expected by the library function. The library function accesses the procedure argument using an API described in "External function prototypes" on page 564.

Any value returned by the external function is in turn returned by the procedure to the calling environment.

No other  
statements  
permitted

A procedure that references an external function can include no other statements: its sole purpose is to take arguments for a function, call the function, and return any value and returned arguments from the function to the calling environment. You can use IN, INOUT, or OUT parameters in the procedure call in the same way as for other procedures: the input values get passed to the external function, and any parameters modified by the function are returned to the calling environment in OUT or INOUT parameters.

System-dependent  
calls

You can specify operating-system dependent calls, so that a procedure calls one function when run on one operating system, and another function (presumably analogous) on another operating system. The syntax for such calls involves prefixing the function name with the operating system name. For example:

```
CREATE PROCEDURE dll_proc ( parameter-list )  
EXTERNAL NAME  
'Windows95:95_fn@95_lib.dll;WindowsNT:nt_fn@nt_lib.dll'
```

The operating system identifier must be one of **WindowsNT**, **Windows95**, **UNIX**, or **NetWare**.

If the list of functions does not contain an entry for the operating system on which the server is running, but the list does contain an entry without an operating system specified, the database server calls the function in that entry.

NetWare calls have a slightly different format than the other operating systems. All symbols are globally known under NetWare, so any symbol (such as a function name) exported must be unique to all NLMs on the system. Consequently, the NLM name is not necessary in the call, and the call has the following syntax:

```
CREATE PROCEDURE dll_proc ( parameter-list )  
EXTERNAL NAME 'NetWare:nw_fn'
```

There is no need to provide a library name. If you do provide one, it is ignored.

🔗 For more information about the CREATE PROCEDURE statement syntax, see "CREATE PROCEDURE statement" on page 305 of the book *ASA SQL Reference Manual*.

🔗 For more information about the CREATE FUNCTION statement syntax, see "CREATE FUNCTION statement" on page 296 of the book *ASA SQL Reference Manual*.

## External function prototypes

This section describes the API for functions in external libraries.

The API is defined by a header file named *extfnapi.h*, in the *h* subdirectory of your SQL Anywhere Studio installation directory. This header file handles the platform-dependent features of external function prototypes.

### Declaring the API version

To notify the database server that the library is not written using the old API, you must provide a function as follows:

```
uint32 extfn_use_new_api( )
```

The function returns an unsigned 32-bit integer. If the return value is non-zero, the database server assumes that you are not using the old API.

If the function is not exported by the DLL, the database server assumes that the old API is in use. When using the new API, the returned value must be the API version number defined in *extfnapi.h*.

### Function prototypes

The name of the function must match that referenced in the CREATE PROCEDURE or CREATE FUNCTION statement. The function declaration must be as follows:

```
void function-name( an_extfn_api *api, void *argument-handle )
```

The function must return void, and must take as arguments a structure used to pass the arguments, and a handle to the arguments provided by the SQL procedure.

The **an\_extfn\_api** structure has the following form:

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
        void *      arg_handle,
        a_SQL_uint32 arg_num,
        an_extfn_value *value
    );
    short (SQL_CALLBACK *get_piece)(
        void *      arg_handle,
        a_SQL_uint32 arg_num,
        an_extfn_value *value,
        a_SQL_uint32 offset
    );
    short (SQL_CALLBACK *set_value)(
        void *      arg_handle,
        a_SQL_uint32 arg_num,
        an_extfn_value *value
        short        append
    );
    void (SQL_CALLBACK *set_cancel)(
        void *      arg_handle,
        void *      cancel_handle
    );
} an_extfn_api;
```

The **an\_extfn\_value** structure has the following form:

```
typedef struct an_extfn_value {
    void *      data;
    a_SQL_uint32 piece_len;
    union {
        a_SQL_uint32 total_len;
        a_SQL_uint32 remain_len;
    } len;
    a_SQL_data_type type;
} an_extfn_value;
```

## Notes

Calling **get\_value** on an OUT parameter returns the data type of the argument, and returns data as NULL.

The **get\_piece** function for any given argument can only be called immediately after the **get\_value** function for the same argument,

To return NULL, set **data** to NULL in **an\_extfn\_value**.

The **append** field of **set\_value** determines whether the supplied data replaces (false) or appends to (true) the existing data. You must call **set\_value** with **append=FALSE** before calling it with **append=TRUE** for the same argument. The **append** field is ignored for fixed length data types.

The header file itself contains some additional notes.

The following table shows the conditions under which the functions defined in **an\_extfn\_api** return false:

Function	Returns 0 when the following is true; else returns 1
<b>get_value()</b>	<ul style="list-style-type: none"><li>- <b>arg_num</b> is invalid; for example, <b>arg_num</b> is greater than the number of arguments in <b>ext_fn</b></li><li>- It is called before the external function call has been properly initialized</li></ul>
<b>get_piece()</b>	<ul style="list-style-type: none"><li>- <b>arg_num</b> is invalid; for example, <b>arg_num</b> does not correspond to the last <b>get_value</b>.</li><li>- The offset is greater than the total length of the value for the <b>arg_num</b> argument.</li><li>- It is called before the external function call has been properly initialized.</li></ul>
<b>set_value()</b>	<ul style="list-style-type: none"><li>- <b>arg_num</b> is invalid; for example, <b>arg_num</b> is greater than the number of arguments in <b>ext_fn</b>.</li><li>- <b>arg_num</b> argument is input only.</li><li>- The type of value supplied does not match that of the <b>arg_num</b> argument.</li><li>- It is called before the external function call has been properly initialized.</li></ul>

For more information about the values you can enter in the **a\_sql\_data\_type** field, see "Embedded SQL data types" on page 177 of the book *ASA Programming Guide*.

For more information about passing parameters to external functions, see "Passing parameters to external functions" on page 567.

Implementing  
cancel processing

An external function that expects to be canceled must inform the database server by calling the **set\_cancel** API function. You must export a special function to enable external operations to be canceled. This function must have the following form:

```
void an_extfn_cancel( void * cancel_handle )
```



If the DLL does not export this function, the database server ignores any user interrupts for functions in the DLL. In this function, *cancel\_handle* is a pointer provided by the function being cancelled to the database server upon each call to the external function by the *set\_cancel* API function listed in the **an\_extfn\_api** structure, above.

## Passing parameters to external functions

### Data types

The following SQL data types can be passed to an external library:

SQL data type	C type
CHAR	Character data, with a specified length
VARCHAR	Character data, with a specified length
LONG VARCHAR	Character data, with a specified length
BINARY	Binary data, with a specified length
LONG BINARY	Character data, with a specified length
TINYINT	1-byte integer
[ UNSIGNED ] SMALLINT	[ Unsigned ] 2-byte integer
[ UNSIGNED ] INT	[ Unsigned ] 4-byte integer
[ UNSIGNED ] BIGINT	[ Unsigned ] 8-byte integer
VARBINARY	Binary data, with a specified length
REAL	Single precision floating point number
DOUBLE	Double precision floating point number

You cannot use date or time data types, and you cannot use exact numeric data types.

To provide values for INOUT or OUT parameters, use the **set\_value** API function. To read IN and INOUT parameters, use the **get\_value** API function.

### Passing NULL

You can pass NULL as a valid value for all arguments. Functions in external libraries can supply NULL as a return type for any data type.

### External function return types

The following table lists the supported return types, and how they map to the return type of the SQL function or procedure.

C data type	SQL data type
void	Used for external procedures.
char *	function returning CHAR().
long	function returning INTEGER
float	function returning FLOAT
double	function returning DOUBLE.

If a function in the external library returns NULL, and the SQL external function was declared to return CHAR(), then the return value of the SQL extended function is NULL.

## Hiding the contents of procedures, functions, triggers and views

In some cases, you may want to distribute an application and a database without disclosing the logic contained within procedures, functions, triggers and views. As an added security measure, you can obscure the contents of these objects using the SET HIDDEN clause of the ALTER PROCEDURE, ALTER FUNCTION, ALTER TRIGGER, and ALTER VIEW statements.

The SET HIDDEN clause scrambles the contents of the associated objects and makes them unreadable, while still allowing the objects to be used. You can also unload and reload the objects into another database.

The modification is irreversible, and for databases created using version 8.0 or higher, deletes the original text of the object. Preserving the original source for the object outside the database is required.

Debugging using the stored procedure debugger will not show the procedure definition, nor will procedure profiling display the source.

Running one of the above statements on an object that is already hidden has no effect.

To hide the text for all objects of a particular type, you can use a loop similar to the following:

```
begin
  for hide_lp as hide_cr cursor for
    select proc_name,user_name
    from SYS.SYSPROCEDURE p, SYS.SYSUSERPERM u
    where p.creator = u.user_id
    and p.creator not in (0,1,3)
  do
    message 'altering ' || proc_name;
    execute immediate 'alter procedure "' ||
      user_name || '."' || proc_name
      || '" set hidden'
  end for
end
```

☞ For more information, see the "ALTER FUNCTION statement" on page 213 of the book *ASA SQL Reference Manual*, the "ALTER PROCEDURE statement" on page 214 of the book *ASA SQL Reference Manual*, the "ALTER TRIGGER statement" on page 240 of the book *ASA SQL Reference Manual*, and the "ALTER VIEW statement" on page 241 of the book *ASA SQL Reference Manual*.



C H A P T E R   1 8

Debugging Logic in the Database

About this chapter      This chapter describes how to use the Sybase debugger to assist in developing Java classes, SQL stored procedures, triggers, and event handlers.

Contents	<table><tr><th>Topic</th><th>Page</th></tr><tr><td>Introduction to debugging in the database</td><td>572</td></tr><tr><td>Tutorial: Getting started with the debugger</td><td>574</td></tr><tr><td>Common debugger tasks</td><td>585</td></tr><tr><td>Starting the debugger</td><td>586</td></tr><tr><td>Working with breakpoints</td><td>589</td></tr><tr><td>Examining variables</td><td>593</td></tr><tr><td>Configuring the debugger</td><td>595</td></tr></table>	Topic	Page	Introduction to debugging in the database	572	Tutorial: Getting started with the debugger	574	Common debugger tasks	585	Starting the debugger	586	Working with breakpoints	589	Examining variables	593	Configuring the debugger	595
Topic	Page																
Introduction to debugging in the database	572																
Tutorial: Getting started with the debugger	574																
Common debugger tasks	585																
Starting the debugger	586																
Working with breakpoints	589																
Examining variables	593																
Configuring the debugger	595																

# Introduction to debugging in the database

You can use the debugger during the development of the following objects:

- ◆ SQL stored procedures, triggers, event handlers, and user-defined functions.
- ◆ Java classes in the database.

This chapter describes how to set up and use the debugger.

Separately-  
licensable  
component

Java in the database is a separately licensable component and must be ordered before you can install it. To order this component, see the card in your SQL Anywhere Studio package or see <http://www.sybase.com/detail?id=1015780>.

## Debugger features

You can carry out many tasks with the debugger, including the following:

- ◆ **Debug procedures and triggers** You can debug SQL stored procedures and triggers.
- ◆ **Debug event handlers** Event handlers are an extension of SQL stored procedures. The material in this chapter about debugging stored procedures applies equally to debugging event handlers.
- ◆ **Browse stored procedures and classes** You can browse through the source code of installed classes and SQL procedures.
- ◆ **Debug Java classes** You can debug Java classes that are stored in the database.
- ◆ **Trace execution** Step line by line through the code of a stored procedure or Java class running in the database. You can also look up and down the stack of functions that have been called.
- ◆ **Set breakpoints** Run the code until you hit a breakpoint, and stop at that point in the code.
- ◆ **Set break conditions** Breakpoints include lines of code, but you can also specify conditions when the code is to break. For example, you can stop at a line the tenth time it is executed, or only if a variable has a particular value. You can also stop whenever a particular exception is thrown in a Java application.
- ◆ **Inspect and modify local variables** When execution is stopped at a breakpoint, you can inspect the values of local variables and alter their value.

- ◆ **Inspect and break on expressions** When execution is stopped at a breakpoint, you can inspect the value of a wide variety of expressions.
- ◆ **Inspect and modify row variables** Row variables are the OLD and NEW values of row-level triggers. You can inspect and set these values.
- ◆ **Execute queries** When execution is stopped at a breakpoint in a SQL procedure, you can execute queries. This permits you to look at intermediate results held in temporary tables, as well as checking values in base tables.

## Requirements for using the debugger

The debugger runs on a client machine, and connects to the database using JDBC.

You need the following in order to use the debugger:

- ◆ **Permissions** In order to use the debugger, you must either have DBA authority or be granted permissions in the SA\_DEBUG group. This group is added to all databases when they are created.
- ◆ **jConnect Support** The database that you want to connect to must have jConnect support installed.
- ◆ **Source code** The source code for your application must be available to the debugger. For Java classes, the source code is held on a directory on your hard disk. For stored procedures, the source code is held in the database.
- ◆ **Compilation options** To debug Java classes, they must be compiled so that they contain debugging information. For example, if you are using the Sun Microsystems JDK compiler *javac.exe*, they must be compiled using the *-g* command-line option.

## Debugger information

This chapter contains a tutorial to help you get started using the debugger. Task-based help and information about each window is available in the debugger online Help.

# Tutorial: Getting started with the debugger

This tutorial describes how to start the debugger, how to connect to a database, how to debug a simple stored procedure, and how to debug a Java class.

## Lesson 1: Connect to a database

This tutorial shows you how to start the debugger, connect to a database, and attach to a connection for debugging.

### Start the debugger

The debugger runs on your client machine.

You can start the debugger from the command line or from Sybase Central.

#### ❖ To start the debugger:

- 1 For the purposes of this tutorial, shut down all local servers and then start the sample database.

Choose Start ► Programs ► Sybase SQL Anywhere 8 ► Adaptive Server Anywhere ► Personal Server Sample.

- 2 Start the debugger:

Choose Start ► Programs ► Sybase SQL Anywhere 8 ► Adaptive Server Anywhere ► Database Object Debugger.

*or*

At a command prompt, enter the command **dbprdbg**.

The Connect window appears:



**Connect**

Identification | Database | Advanced

The following values are used to identify yourself to the database

User:

Password:

User to debug:

---

You can use default connection values stored in a profile.

☒ None

☐ ODBC Data Source \_name

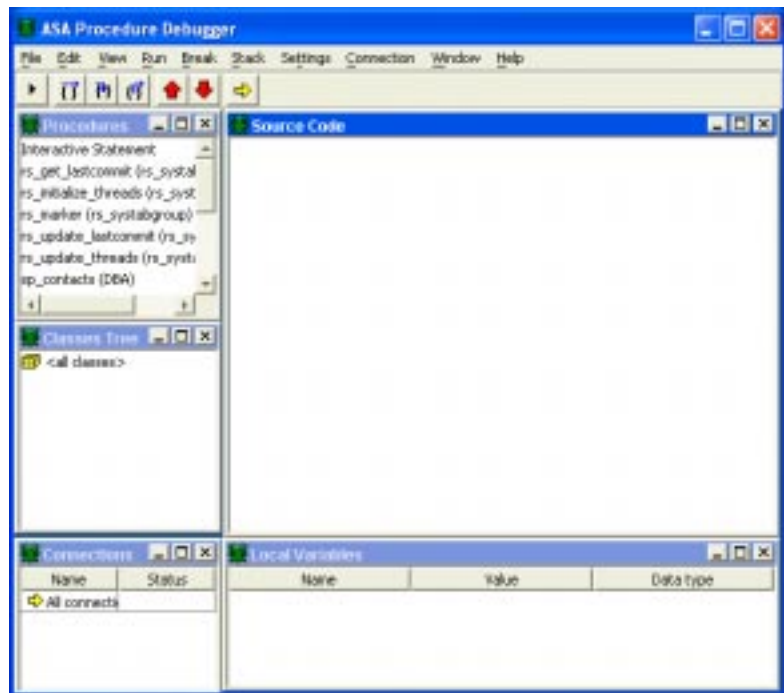
ASA 8.0 Sample

☐ ODBC Data Source \_file

OK Cancel Help

You can now connect to a database from the debugger.

- 3 In the debugger Login window, enter the following information:
  - ◆ **ODBC Data Source Name** Select **ASA 8.0 Sample**. If you have not used this data source before, you may have to click Browse to locate it.
  - ◆ **User to debug** Leave this field empty to indicate that you wish to debug connections for all users.
- 4 Click OK to connect to the database. The debugger interface appears, displaying a list of stored procedures and triggers, and a list of Java classes installed in the database.



## Lesson 2: Debug a stored procedure

This tutorial describes a sample session for debugging a stored procedure. It is a continuation of "Lesson 1: Connect to a database" on page 574.

In this tutorial, you call the stored procedure `sp_customer_products`, which is part of the sample database.

The `sp_customer_products` procedure executes the following query against the sample database:

```
SELECT product.id,sum(sales_order_items.quantity)
FROM product,sales_order_items,sales_order
WHERE sales_order.cust_id = customer_id
AND sales_order.id = sales_order_items.id
AND sales_order_items.prod_id = product.id
GROUP BY product.id
```

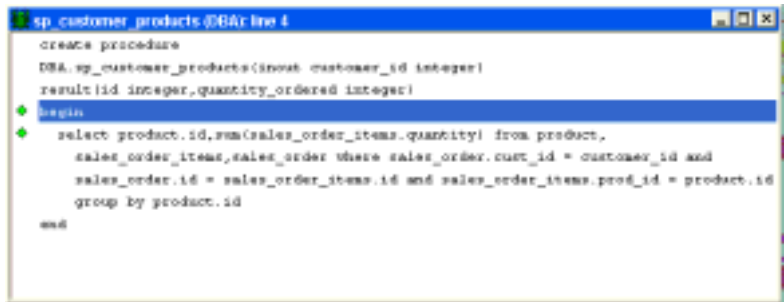
It takes a customer ID as input, and returns as a result set a list of product IDs and the number ordered by that customer.

## Display stored procedure source code in the debugger

The stored procedure source code is stored in the database. You can display the stored procedure source code in the Source Code window.

### ❖ To display stored procedure source code in the debugger:

- ◆ In the Procedures window, double-click the *sp\_customer\_products* stored procedure. The source code for the procedure appears in the Source Code window.



## Set a breakpoint

You can set a breakpoint in the body of the *sp\_customer\_products* procedure. When the procedure is executed, execution stops at the breakpoint.

### ❖ To set a breakpoint in a stored procedure:

- 1 In the source code window, locate the line with the query:  

```
select product.id,...
```
- 2 Click the green diamond to the left of the line until it is red.  
 Repeatedly clicking the indicator toggles its status.

## Run the procedure

You can call the stored procedure from Interactive SQL, and see its execution interrupted at the breakpoint.

### ❖ To call the procedure from Interactive SQL:

- 1 Start Interactive SQL and connect to the sample database with a user ID of **DBA** and a password of **SQL**.

The connection appears in the debugger Connections window.

- 2 Execute the following command in Interactive SQL to call the procedure using the customer with ID 122:

```
CALL sp_customer_products( 122 )
```

The query does not complete. Instead, execution is stopped in the debugger at the breakpoint. In Interactive SQL, the Interrupt the SQL Statement button is enabled. In the debugger Source window, a yellow arrow indicates the current line.

- 3 Step to the next line by choosing Run►Step Over. You can alternatively press F7.

For longer procedures, you can use other methods of stepping through code. For example:

- ◆ **Run to a selected line** Select the following line using the mouse, and choose Run►Run To Selected, or press F6 to run to that line and break:

```
max_price = price;
```

The red arrow moves to the line.

- ◆ **Set a breakpoint and execute to it** Select the following line (line 292) and press F9 to set a breakpoint on that line:

```
return max_price;
```

An asterisk appears in the left-hand column to mark the breakpoint. Press F5 to execute to that breakpoint.

For the next lesson, complete execution of the procedure (press F5) and re-execute step 2 above to leave the debugger stopped at the SELECT line.

## Inspect and modify variables

You can inspect the values of variables in the debugger.

### Inspecting local variables

You can inspect the values of local variables in a procedure as you step through the code, to better understand what is happening.

#### ❖ To inspect and modify the value of a variable:

- 1 If the Local Variables window is not displayed, choose View►Variables►Local Variables to display it.

The Local Variables window displays two local variables: the stored procedure itself (which does not have a return value, and so is listed as NULL) and the **customer\_id** passed in to the procedure.

- 2 In the Local Variables window, double-click the Value column entry for **customer\_id**, type in 125 and press ENTER to change the customer ID value used in the query.
  - 3 In the source code window, press F5 to complete the execution of the query and finish the tutorial.
- The Results tab in the Results pane in Interactive SQL displays the list of product IDs and quantities for customer 125.

Inspecting trigger  
row variables

In addition to local variables, you can display other variables, such as row-level trigger OLD and NEW values in the debugger Row Variables window.

## Lesson 3: Debug a Java class

This lesson describes a sample session for debugging a Java class.

This lesson requires that you have the Java in the database component. Java in the database is a separately licensable component and must be ordered before you can install it. To order this component, see the card in your SQL Anywhere Studio package or see <http://www.sybase.com/detail?id=1015780>.

In this lesson, you call **JDBCExamples.Query** from Interactive SQL, interrupt the execution in the debugger, and trace through the source code for this method.

The **JDBCExamples.Query** method executes the following query against the sample database:

```
SELECT id, unit_price
FROM product
```

It then loops through all the rows of the result set, and returns the one with the highest unit price.

Compiling Java  
classes for  
debugging

You must compile classes with the *javac -g* option in order to debug them. The sample classes are already compiled for debugging.

### Prepare the database

To work through this tutorial, you must enable the sample database to use Java, and install the Java sample classes into the sample database.

#### ❖ To Java-enable the sample database:

- 1 Close the debugger.
- 2 Start Interactive SQL and connect to the sample database.

- 3 In the SQL Statements pane of Interactive SQL, type the following statement:

```
ALTER DATABASE UPGRADE JAVA JDK '1.3'
```

- 4 Shut down Interactive SQL and the sample database.

The asademo.db database must be shut down before you can use Java features.

❖ **To add Java classes and tables to the sample database:**

- 1 Start Interactive SQL and connect to the sample database.
- 2 In the SQL Statements pane of Interactive SQL, type the following statement:

```
READ "path\\Samples\\ASA\\Java\\jdemo.sql"
```

where *path* is your SQL Anywhere directory. This runs the instructions in the *jdemo.sql* command file. The instructions may take some time to complete.

🔗 For information about installing the Java examples, see "Setting up the Java samples" on page 86 of the book *ASA Programming Guide*.

🔗 For information about the **JDBCExamples** class and its methods, see "Data Access Using JDBC" on page 129 of the book *ASA Programming Guide*.

## Display Java source code into the debugger

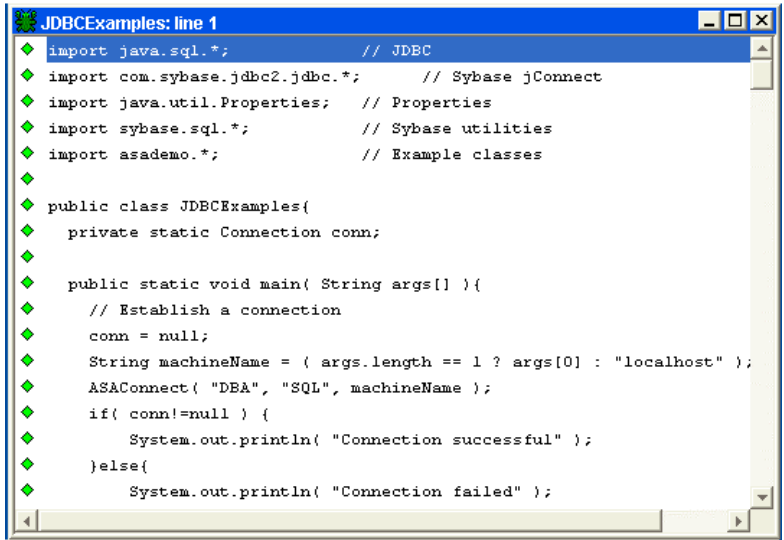
The debugger looks in a set of locations for source code files (with *.java* extension). You need to add the *Samples\ASA\Java* subdirectory of your installation directory to the list of locations, so that the code for the class currently being executed in the database is available to the debugger.

❖ **To display Java source code in the debugger:**

- 1 Start the debugger and connect to the ASA 8.0 Sample database.
- 2 From the debugger interface, select File►Edit Source Path. The Source path window appears.
- 3 Click Browse Folder, and navigate to the *Samples\ASA\Java* subdirectory of your SQL Anywhere installation directory. For example, if you installed SQL Anywhere in *c:\program files\sybase\sql anywhere 8*, you would enter the following:

```
c:\program files\sybase\sql anywhere 8\Samples\ASA\Java
```

- 4 Click OK to add the folder to the list and close the Source Path window.
- 5 In the Classes Tree window, double-click **JDBCExamples**. The source code for the **JDBCExamples** class appears in the Source window.



```

JDBCExamples: line 1
import java.sql.*;           // JDBC
import com.sybase.jdbc2.jdbc.*; // Sybase jConnect
import java.util.Properties; // Properties
import sybase.sql.*;         // Sybase utilities
import asademo.*;            // Example classes

public class JDBCExamples{
    private static Connection conn;

    public static void main( String args[] ){
        // Establish a connection
        conn = null;
        String machineName = ( args.length == 1 ? args[0] : "localhost" );
        ASACONNECT( "DEA", "SQL", machineName );
        if( conn!=null ) {
            System.out.println( "Connection successful" );
        }else{
            System.out.println( "Connection failed" );
        }
    }
}

```

Notes on locating  
Java source code

The Source Path window holds a list of directories in which the debugger looks for Java source code. The debugger also searches the current CLASSPATH for source code.

## Set a breakpoint

### ❖ Set a breakpoint in a Java class:

- 1 In the source code window, page down until you see the beginning of the **Query** method. This method is near the end of the class, and starts with the following line:

```
public static int Query() {
```

- 2 Click the green indicator to the left of the first line of the method, until it is red. The first line of the method is:

```
int max_price = 0;
```

Repeatedly clicking the indicator toggles its status.

## Run the method

### ❖ Invoke the method from Interactive SQL:

- 1 Start Interactive SQL. Connect to the sample database as used ID **DBA** and password **SQL**.
- 2 Enter the following command in Interactive SQL to invoke the method:

```
SELECT JDBCExamples.Query( )
```

The query does not complete. Instead, execution is stopped in the debugger at the breakpoint. In Interactive SQL, the Interrupt the SQL Statement button is enabled. In the debugger Source window, the yellow arrow indicates the current line.

You can now step through source code and carry out debugging activities in the debugger.

## Step through source code

This section illustrates some of the ways you can step through code in the debugger.

Following the previous section, the debugger should have stopped execution of **JDBCExamples.Query** at the first statement in the method:

### Examples

Here are some example steps you can try:

- 1 **Step to the next line** Choose Run►Step Over, or press F7 to step to the next line in the current method. Try this two or three times.
- 2 **Run to a selected line** Select the following line using the mouse, and choose Run►Run To Selected, or press F6 to run to that line and break:

```
max_price = price;
```

The yellow arrow moves to the line.

- 3 **Set a breakpoint and execute to it** Select the following line (line 292) and press F9 to set a breakpoint on that line:

```
return max_price;
```

A red stop sign appears in the left-hand column to mark the breakpoint. Press F5 to execute to that breakpoint.

- 4 **Experiment** Try different methods of stepping through the code. End with F5 to complete the execution.

The complete set of options for stepping through source code is available from the **Run** menu.



When you have completed the execution, the Interactive SQL Results pane in the Results tab displays the value 24.

## Inspect and modify variables

In this lesson you inspect the values of both local variables (declared in a method) and class static variables in the debugger.

### Inspecting local variables

You can inspect the values of local variables in a method as you step through the code, to better understand what is happening. You must have compiled the class with the *javac -g* option to do this.

#### ❖ To inspect and modify the value of a variable:

- 1 Set a breakpoint at the first line of the **JDBCExamples.Query** method. This line is as follows:

```
int max_price = 0
```

- 2 In Interactive SQL, enter the following statement again to execute the method:

```
SELECT JDBCExamples.Query()
```

The query executes only as far as the breakpoint.

- 3 Press F7 to step to the next line. The **max\_price** variable has now been declared and initialized to zero.
- 4 If the Local Variables window is not displayed, choose View ► Variables ► Local Variables to display it.

The Local Variables window displays several local variables. The **max\_price** variable has a value of zero. All others are listed as variable not in scope, which means they are not yet initialized.

- 5 In the Local Variables window, double-click the Value column entry for **max\_price**, and type in 45 to change the value of **max\_price** to 45.

The value 45 is larger than any other price. Instead of returning 24, the query will now return 45 as the maximum price.

- 6 In the Source window, press F7 repeatedly to step through the code. As you do so, the values of the variables appear in the Local Variables window. Step through until the **stmt** and **result** variables have values.
- 7 Expand the **result** object by clicking the icon next to it, or setting the cursor on the line and pressing ENTER. This displays the values of the fields in the object.
- 8 When you have experimented with inspecting and modifying variables, press F5 to complete the execution of the query and finish the tutorial.

## Inspecting static variables

In addition to local variables, you can display class-level variables (static variables) in the debugger Statics window, and inspect their values in the Inspect window. For more information, see the debugger online Help.

## Common debugger tasks

If you want to...	Then choose...
Display caller's context	Stack►Down
Display caller's context	Stack►Up
Exit the debugger	File►Exit
Find a string in the selected window	Edit►Find
Ignore the case of a word when searching	Edit►Ignore Case
Load the procedure debugger's settings from a file	Settings►Load From File
Login to the database as a new connection	Connection►Connect
Log out of the database	Connection►Disconnect
Run a program, line by line, going through procedures and triggers line by line as well	Run►Step Into
Resume a program's execution from a breakpoint	Run►Go
Run a program until the current procedure/method returns.	Run►Step Out
Specify the path containing the Java source file	File►Edit Source Path
View the source code for a procedure or class	File►Open

## Starting the debugger

You can start the debugger from the command line, from the Windows Start menu, or from Sybase Central.

❖ **To start the debugger (Windows):**

- ◆ Choose Start ► Programs ► Sybase SQL Anywhere 8 ► Adaptive Server Anywhere ► Database Object Debugger.

❖ **To start the debugger (Command line):**

- ◆ At a command prompt, execute the following command:

`dbprdbg`

If you have set a `SQLCONNECT` environment variable, the debugger uses it to establish the database connection.

❖ **To start the debugger (Sybase Central):**

- 1 Start Sybase Central.
- 2 Open the Utilities folder of the Adaptive Server Anywhere plug-in.
- 3 Double-click Debug Database Objects in the right pane.

## Connecting to a database

The debugger connects to a database like any other client application. The procedures and Java classes it debugs are executed from other connections, not from the debugger itself.

❖ **To connect to a database:**

- 1 Start the database you wish to debug.
- 2 Start the debugger. If you do not have the `SQLCONNECT` environment variable set, or if you choose Connection►Connect, the debugger displays a Connection dialog at startup.

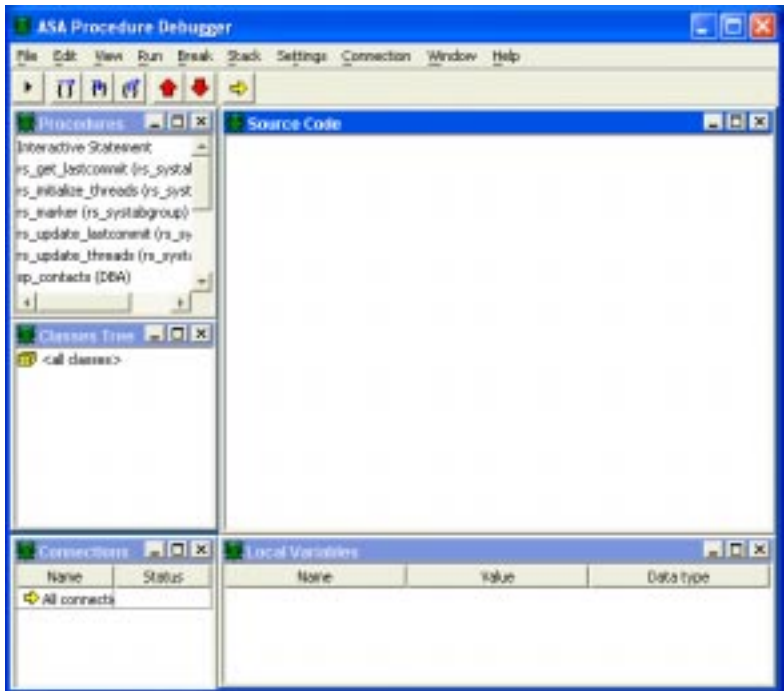
 For more information, see "Starting the debugger" on page 586.

- 3 In the Connect dialog, enter the usual connection information, as if you were connecting from Interactive SQL or Sybase Central.

For more information, see "Connecting from Sybase Central or Interactive SQL" on page 42 of the book *ASA Database Administration Guide*.

In addition, enter the user ID for the connection you wish to debug. To debug all connections to the database, you can leave the field empty or enter an asterisk (\*).

- 4 Click OK to connect. The debugger interface appears, displaying a list of stored procedures and triggers, and a list of Java classes installed in the database.



## Choosing a connection to debug

The Connections window shows all connections to the database, and is updated automatically. Debugger breakpoints apply only to code running in a given connection, called the **active connection**.

### ❖ To set the active connection:

- 1 Open the Connections window.
- 2 Double-click the connection you wish to make active.

A yellow arrow indicates the current active connection.

The special setting **All connections** is treated like a separate connection. If you choose this setting, breakpoints apply to all connections to the database.


# Working with breakpoints

This section describes how to use breakpoints to control when the debugger interrupts execution of your source code.

## Setting breakpoints

A breakpoint instructs the debugger to interrupt execution at a specified line.

When you set a breakpoint, it applies to the active connection only. The active connection is indicated in the Connections window by a yellow arrow.

 For information, see "Choosing a connection to debug" on page 587.

### ❖ To set a breakpoint (source code window):

- 1 Make the connection that you wish to debug the active connection.
- 2 Open the source code window and display the code where you wish to set a breakpoint.
- 3 Double-click a line, or click a line and press F9 to set the breakpoint. A red circle indicates each line with a breakpoint.

### ❖ To set a breakpoint (Break menu):

- 1 Make the connection you wish to debug the active connection.
- 2 Choose Break ► New. Enter a breakpoint address in the dialog.

For a stored procedure, enter an address of the form *procedure:line* where *procedure* includes the procedure name, followed by the procedure owner's user ID in parentheses. For example, **sp\_customer\_list (DBA)**.

For a Java class, enter an address of the form *class:line* or *class.method*.

## Disabling and enabling breakpoints

You can change the status of a breakpoint from the source code window or from the Break menu.

### ❖ To change the status of a breakpoint (source code window):

- 1 Make the connection you wish to debug the active connection.

- 2 Open the source code window and display the procedure that contains the breakpoint whose status you wish to change.
- 3 Click the breakpoint indicator on the left of the line you wish to edit. The status of the line switches from being a breakpoint for the active connection only, to being a breakpoint for all connections, being a disabled breakpoint, to not being a breakpoint.

❖ **To change the status of a breakpoint (Breakpoints window):**

- 1 Highlight the connection you wish to debug to make it the active connection.
- 2 Open the Breakpoints window.
- 3 Click the breakpoint indicator on the left. The status of the breakpoint changes from enabled to disabled.

Alternatively, you can insert a breakpoint by pressing **INSERT** and specifying the breakpoint's address. You can delete a breakpoint by selecting the breakpoint and pressing **DELETE**.

❖ **To change the status of breakpoints (Break menu):**

- 1 Make the connection you wish to debug the active connection.
- 2 From the Break menu, choose the desired behavior:
  - ◆ **Clear All** Clear all breakpoints.
  - ◆ **Disable All** Disable all breakpoints.
  - ◆ **Enable All** Enable all breakpoints.

## Editing breakpoint conditions

You can add conditions to breakpoints, to instruct the debugger to interrupt execution at that breakpoint only when a certain condition or count is satisfied.

❖ **To add a condition or count to a breakpoint:**

- 1 Make the connection you wish to debug the active connection.
- 2 Open the Breakpoints window. The breakpoints for this connection are displayed in a list.

For a Java class, the condition must be a Java boolean expression. For procedures and triggers, it must be a SQL search condition. The condition is evaluated in the context of the active connection.



## Examples

In the procedure *sp\_contacts*, you might use the breakpoint

```
contact.id = contact.old_id
```

on the line

```
DELETE FROM contact WHERE contact.id = contact.old_id
```

In the Java method **JDBCExamples.Query**, you might use the breakpoint condition

```
(price < 10)
```

on the line

```
if (max.price == price) or (price == 10)
```

## Break on exception

Like breakpoints, Break on Exception applies to the active connection. Instead of breaking at a particular line, you can cause the debugger to interrupt execution when an exception is thrown from a Java class.

### ❖ To set a break condition:

- 1 Make the connection you wish to debug the active connection.

For information on setting the active connection, see "Choosing a connection to debug" on page 587.

- 2 Set a break condition from the Break menu by choosing one of the following items:


- ◆ **When Exception Thrown** Break when the selected class is thrown.
- ◆ **On Any Exception** Break when any Java exception is thrown.

## Interrupting connections

It can be useful to interrupt a connection immediately, or more precisely at the next Java or stored procedure instruction it executes. You can interrupt one connection at a time.

### ❖ To interrupt a connection:

- 1 Make the connection you wish to debug the active connection. If you want to interrupt a connection that is not yet present in the Connection window, you must choose All connections.

 For information, see "Choosing a connection to debug" on page 587.


- 2 Choose **Run ► Interrupt**. The debugger interrupts execution on the active connection the next time it executes a statement.

## Examining variables

The procedure debugger lets you view and edit the behavior of your variables while stepping through your code. The debugger provides a set of variables windows to display the different kinds of variables used in stored procedures and Java classes. You display the variable windows by choosing View ► Variables, and click the appropriate window in the list.


### Local variables

#### ❖ To watch the values of your variables:


- 1 Set a breakpoint in the procedure whose variables you wish to examine.  
 For information on setting breakpoints, see "Setting breakpoints" on page 589.
- 2 Open the Local Variables window.
- 3 Run the procedure. The variables, along with their values, appear in the Local Variables window.

### Other variables

Global variables are defined by Adaptive Server Anywhere and hold information about the current connection, database, and other settings. They are displayed in the Globals window.

 For a list of global variables, see "Global variables" on page 40 of the book *ASA SQL Reference Manual*.

Row variables are used to hold the values used in triggers. They are displayed in the Row Variables window.

 For more information on triggers, see "Introduction to triggers" on page 522.

Static variables are used in Java classes. They are displayed in the Statics window.

### The call stack

Sometimes it is useful to examine the context of your variables when they are being used in nested procedures. You can view a listing of the procedures that are using your variables in the Calls window.

#### ❖ To display the context of variables:

- 1 Set a breakpoint in the procedure whose variables you wish to examine.
- 2 Click View ► Other ► Calls to open the Calls Window.

The names of the procedures using the variables appear in the Calls window. The current procedure is shown at the top of the list. The procedure that called it is immediately below, and so forth.

- 3 Double-click a context in the Calls window to change the contents of the variables' windows.

## Using full variable names when expanding objects and arrays

By default, the procedure debugger displays the short form of variable names when expanding objects and arrays. You can modify this setting as follows:

❖ **To use long variable names for objects and arrays:**

- ◆ Choose Settings ► Use long variable names.

## Changing the format in which values of variables are displayed

The procedure debugger allows you to specify the format in which variable values are displayed.

❖ **To change the format of variable values:**

- ◆ In the Settings menu, select the format in which you want your variables displayed. The following formats are available:
  - ◆ Display 'char' in hex
  - ◆ Display 'char' as int
  - ◆ Display 'byte' in hex
  - ◆ Display 'short' in hex
  - ◆ Display 'int' in hex
  - ◆ Display 'long' in hex

# Configuring the debugger

The Adaptive Server Anywhere Procedure debugger allows you to adjust settings to make the debugger easier to use. You can specify the way you want to use your windows, and determine how you want to use breakpoints and display characters by using the Settings menu.

## Saving the positions of your windows

You can save the positions of the windows in the procedure debugger.

### ❖ To save the positions of your windows:

- 1 Choose Settings ► Remember windows positions.
- 2 Choose Settings ► Save.

## Enabling VI keys in text-entry windows

By default, the keys in the windows in which you can view or edit text, such as the Source Code window, have the same meanings that they have in Windows Notepad. The debugger allows you to choose to edit the text in these windows in a way that is similar to the way you edit text in VI.

### ❖ To enable some VI keys in the text windows:

- ◆ Choose Settings ► VI.

The following VI keys are enabled when the VI setting is checked:

- ◆ / search
- ◆ n find next
- ◆ N find previous
- ◆ j cursor up
- ◆ k cursor down
- ◆ CTRL-b page up
- ◆ CTRL-f page down

## Automatically saving all setting changes upon exiting the debugger

By default, all changes to settings are saved when you exit the debugger.

### ❖ To save settings changes upon exiting the debugger:

- ◆ Choose Settings ► Save on exit.

Your settings are saved in the file *ProcDebugDefaults.rc*, in the directory that contains the procedure debugger. When you next open the debugger, your settings will be as they are in *ProcDebugDefaults.rc*.

## Saving your settings to a file

The procedure debugger allows you to save your settings to the file *ProcDebug.rc*.

### ❖ To save your settings to a file:

- ◆ Choose Settings ► Save to file.

## Loading saved settings from a file

You can retrieve saved settings from the file *ProcDebug.rc* as follows:

### ❖ To load your settings from a file:

- ◆ Choose Settings ► Load from file.

# Index

**\***

\* (asterisk)  
select statement, 187

\*=  
Transact-SQL outer joins, 245

**@**

@@identity global variable, 400

**<**

<, 196

**=**

=\*  
Transact-SQL outer joins, 245

**>**

>, 196

**A**

abbreviations used in access plans, 364

access plans  
abbreviations, 364  
caching of, 322

context sensitive help, 373  
customizing, 373  
explanation of statistics, 365  
graphical plans, 373  
Interactive SQL, 378  
long text plans, 372  
reading, 364  
short text plans, 371  
SQL functions, 378

access plans caching, 322

accessing remote data, 455  
basic concepts, 458

accessing remote data from PowerBuilder  
DataWindows, 457

accessing tables  
index scan and sequential scan, 324

actions  
CASCADE, 86  
RESTRICT, 86  
SET DEFAULT, 86  
SET NULL, 86

active connection  
defined, 587  
setting, 587

Adaptive Server Anywhere SQL features, 416

Adaptive Server architectures, 387

Adaptive Server Enterprise  
compatibility, 384  
compatibility in data import/export, 454  
GROUP BY compatibility, 225  
migrating to Adaptive Server Anywhere, 449

add foreign key wizard  
using, 47

adding and removing statistics, 164

- adding data
  - about, 301
  - adding NULL, 304
  - BLOBs, 306
  - column data INSERT statement, 304
  - constraints, 304
  - defaults, 304
  - into all columns, 303
  - using INSERT, 303
  - with SELECT, 305
- adding new rows with SELECT, 305
- adding, changing, and deleting data
  - about, 301
- administrator role
  - Adaptive Server Enterprise, 389
- advanced table properties dialog
  - using, 39
- aggregate functions
  - about, 208
  - Adaptive Server Enterprise compatibility, 225
  - ALL keyword, 208
  - data types, 210
  - DISTINCT keyword and, 208
  - GROUP BY clause, 213
  - NULL, 212
  - order by and group by, 222
  - outer references, 209
  - scalar aggregates, 209
  - vector aggregates, 213
- aggregates
  - item in access plans, 369
- algorithms
  - for query execution, 324
- aliases
  - about, 189
  - correlation names, 194
- ALL
  - keyword and aggregate functions, 208
  - keyword and UNION clause, 223
  - subquery tests, 280, 281
- ALL operator
  - about, 280
  - notes, 281
- ALLOW\_NULLS\_BY\_DEFAULT option
  - setting for Transact-SQL compatibility, 395
- ALTER TABLE statement
  - and concurrency, 136
  - CHECK conditions, 76
  - examples, 42
  - foreign keys, 48
  - primary keys, 46
- altering
  - columns, 42
  - procedures, 513
  - tables, 41, 42
  - triggers, 526
  - views, 55
- altering remote servers, 462
- always use a transaction log, 144
- ANSI
  - non-ANSI joins, 234
  - SQL/92 standard and inconsistencies, 94
- ANSI compliance. *See* SQL standards
- ANSI update constraints
  - in access plans, 368
- anti-insert
  - locks, 122, 130
- anti-semijoins
  - query execution algorithms, 328, 331
- ANY
  - subquery tests, 279
- ANY operator
  - about, 279
  - problems, 280
- apostrophes
  - character strings, 201
- architecture
  - Adaptive Server, 387
- arithmetic
  - expressions and operator precedence, 192
  - operations, 208
- AS keyword
  - aliases, 189
- asademo.db file
  - about, xvi
  - schema, 228
- asajdbc server class, 489



asaodbc server class, 493

ascending order  
    ORDER BY clause, 220

ASCII  
    format for importing and exporting, 424

asejdbc server class, 490

aseodbc server class, 493

assigning  
    data types to columns, 80  
    domains to columns, 80

assumptions affecting optimization, 318

asterisk (\*)  
    select statement, 187

AT clause  
    CREATE EXISTING TABLE statement, 467

atomic compound statements, 533

atomic transactions, 90

attributes  
    choosing, 14  
    definition of, 5  
    SQLCA.lock, 97

autocommit  
    performance, 149  
    transactions, 91

autoincrement  
    IDENTITY column, 399

AUTOINCREMENT  
    default, 72  
    negative numbers, 73  
    signed data types, 73  
    UltraLite applications, 73  
    when to use, 136

automatic joins  
    and foreign keys, 417

automatic performance tuning, 318

automatic translation of stored procedures, 409

AUTOMATIC\_TIMESTAMP option  
    setting for Transact-SQL compatibility, 395

automatically saving all setting changes upon exiting  
    the debugger [dbprdbg] utility, 596

automation  
    generating unique keys, 135

AvgDiskReads  
    estimate in access plans, 367

AvgDiskReadTime  
    estimate in access plans, 367

AvgDiskWrites  
    estimate in access plans, 367

AvgRowCount  
    estimate in access plans, 367

AvgRunTime  
    estimate in access plans, 367

## B

basic concepts to access remote data, 458

batches  
    about, 529  
    control statements, 529  
    data definition statements, 529  
    SQL statements allowed, 561  
    statements allowed, 561  
    Transact-SQL overview, 407  
    using SELECT statements, 561  
    writing, 407

BEGIN TRANSACTION statement  
    remote data access, 479

benefits of procedures and triggers, 510

BETWEEN keyword  
    range queries, 197

bi-directional replication, 138

binary large objects  
    about, 25  
    inserting, 306

bitmaps, 25  
    scanning, 338

BLOBs  
    about, 25  
    inserting, 306

block nested loops joins  
    query execution algorithms, 328

- blocking, 100, 113
  - deadlock, 101
  - transactions, 100
  - troubleshooting, 101

- BLOCKING option
  - using, 100

- break conditions
  - setting, 589, 591

- Break menu
  - on any exception, 591
  - when exception thrown, 591
  - when value changes, 591

- break on exception
  - debugger [dbprdbg] utility, 591

- breakpoints
  - about, 589
  - conditions, 590
  - counts, 590
  - debugging, 582
  - disabling, 589
  - enabling, 589
  - setting, 589
  - status, 589

- browsing
  - table data, 44
  - views, 57

- browsing databases
  - and isolation levels, 103

- B-tree indexes
  - about, 346

- bulk loading
  - performance, 422

- bulk operations
  - performance, 150

## C

- cache
  - about, 152
  - access plans, 322
  - dynamic sizing, 152
  - encrypted databases require larger cache, 144
  - initial, min and max size, 152
  - Java applications on UNIX, 155

- monitoring size, 155
  - performance, 144
  - read-hit ratio, 376
  - UNIX, 154
  - Windows 95/98, 154
  - Windows NT, 154

- cache size
  - and page size, 338
  - initial, min and max size, 152
  - Java applications on UNIX, 155
  - monitoring, 155
  - UNIX, 154
  - Windows 95/98, 154
  - Windows NT, 154

- CacheHits
  - statistic in access plans, 365

- CacheRead
  - statistic in access plans, 365

- CacheReadIndLeaf
  - statistic in access plans, 365

- CacheReadTable
  - statistic in access plans, 365

- caching
  - access plans, 322
  - subqueries, 363
  - user-defined functions, 363

- call stack
  - debugger [dbprdbg] utility, 593

- CALL statement
  - about, 509
  - examples, 514
  - parameters, 537
  - syntax, 531

- calling external libraries from procedures, 562

- calling procedures, 514

- calling user-defined functions, 519

- cancel
  - external functions, 566

- cardinality
  - item in access plans, 369
  - relationships and, 7

- Cartesian products, 239

- 
- CASCADE action
    - about, 86
  - case sensitivity
    - creating ASE-compatible databases, 394
    - data, 397
    - databases, 396
    - domains, 396, 397
    - identifiers, 185, 397
    - passwords, 397
    - remote access, 485
    - sort order, 221
    - Transact-SQL compatibility, 396
    - user IDs, 397
  - CASE statement
    - syntax, 531
  - catalog
    - Adaptive Server Enterprise compatibility, 389
  - changing data
    - about, 301
    - permissions, 302
    - UPDATE statement, 308
    - updating data using more than one table, 309
  - changing isolation levels within transactions, 98
  - changing many-to-many relationships into entities, 9
  - changing the format in which values of variables are displayed
    - debugger [dbprdbg] utility, 594
  - changing the isolation level, 96
  - character data
    - searching for, 201
  - character strings
    - about, 201
    - quotes, 201
    - select list using, 191
  - character strings and quotation marks, 201
  - character strings in query results, 190
  - CHECK conditions
    - columns, 76
    - deleting, 78
    - domains, 77
    - modifying, 78
    - tables, 78
    - Transact-SQL, 388
  - check constraints
    - choosing, 26
    - using in domains, 81
  - check if you need to change the command delimiter procedures, 559
  - check your file, table, and index fragmentation, 150
  - checking referential integrity at commit, 128
  - choosing column names, 25
  - choosing constraints, 26
  - choosing data types for columns, 25
  - choosing isolation levels, 102
  - classes
    - remote servers, 487
  - clauses
    - about, 184
    - COMPUTE, 403
    - FOR BROWSE, 403
    - FOR READ ONLY, 403
    - FOR UPDATE, 403
    - GROUP BY ALL, 403
    - INTO, 540
    - ON EXCEPTION RESUME, 413, 550, 554
  - clearing procedure profiling
    - SQL, 173
    - Sybase Central, 173
  - client side loading, 422
  - CLOSE statement
    - procedures, 545
  - colons separate join strategies, 370
  - column attributes
    - AUTOINCREMENT, 136
    - generating default values, 136
    - NEWID, 135
  - column CHECK conditions from domains, 77
  - column statistics
    - about, 315
    - updating, 317
  - columns
    - allowing NULL values, 26
    - altering, 42
    - assigning data types and domains, 80

- constraints, 26
  - data types, 25
  - defaults, 70
  - GROUP BY clause, 213
  - IDENTITY, 399
  - naming, 25
  - properties, 25
  - select list, 188
  - select statements, 188
  - timestamp, 398
- command delimiter
- setting, 559
- commas
- in star joins, 250
  - table expression lists, 239
  - when joining table expressions, 262
- COMMIT statement
- compound statements, 533
  - procedures and triggers, 558
  - verify referential integrity, 128
- COMMMIT statement
- remote data access, 479
- common estimates used in the plan, 367
- common statistics used in the plan, 365
- comparison operators
- NULL values, 203
  - subqueries, 293
  - symbols, 196
- comparison test
- subqueries, 278
- comparisons
- NULL values, 203
  - trailing blanks, 196
- compatibility
- Adaptive Server Enterprise, 384
  - case-sensitivity, 396
  - GROUP BY clause, 225
  - import/export with Adaptive Server Enterprise, 454
  - non-ANSI joins, 234
  - outputting nulls, 427
  - servers and databases, 387
  - setting options for Transact-SQL compatibility, 395
- compatibility of joins, 404
- complete passthrough of the statement
- remote data access, 482
- completing transactions, 91
- compliance with SQL standards. *See* SQL standards
- composite indexes
- about, 342
  - effect of column order, 343
  - hash values, 347
- compound statements
- atomic, 533
  - declarations, 533
  - using, 533
- compressed B-tree indexes
- about, 348
- compression
- performance, 151
- COMPUTE clause
- unsupported, 403
- computed columns
- making queries using functions sargable, 351
- computing values in the SELECT list, 191
- concatenating strings
- NULL, 204
- conceptual data modeling
- about, 3
- conceptual database models
- definition of, 5
- concurrency, 92
- about, 94, 135
  - and data definition statements, 136
  - benefits of, 92
  - consistency, 94
  - how locking works, 121
  - improving, 104
  - improving and indexes, 133
  - improving using indexes, 105
  - inconsistency, 94
  - ISO SQL/92 standard, 94
  - performance, 92
  - primary keys, 135
  - replication, 138
  - types of locks, 122

- concurrent transactions
  - blocking, 100, 113
- conditions
  - connecting with logical operators, 205
- configuration
  - debugger [dbprdbg] utility, 595
  - loading from file, 596
  - saving to file, 596
  - variable name display in debugger [dbprdbg] utility, 594
  - variable names in the debugger [dbprdbg] utility, 594
  - VI keys, 595
- configuration notes for JDBC classes, 489
- configuring
  - the Sybase Central Performance Monitor, 165
- configuring databases for Transact-SQL
  - compatibility, 393
- configuring the Sybase Central Performance Monitor, 165
- conflicts
  - cyclical blocking, 101
  - locking, 100
  - transaction blocking, 100, 113
- conflicts between locks, 124
- conformance with SQL standards. *See* SQL standards
- connecting
  - starting a database without connecting, 36
- connecting conditions with logical operators, 205
- CONNECTION\_PROPERTY function
  - about, 162
- connections
  - active, 587
  - database, 586
  - debugging, 574, 586, 587
  - interrupting in the debugger [dbprdbg] utility, 591
  - remote, 479
- connectivity problems
  - remote data access, 486
- consistency
  - about, 90
  - assuring using locks, 121
  - correctness and scheduling, 102
  - dirty reads, 94, 125
  - dirty reads tutorial, 106
  - during transactions, 94
  - effects of unserializable schedules, 103
  - example of non-repeatable read, 110
  - ISO SQL/92 standard, 94
  - isolation levels, 94
  - phantom rows, 94, 113, 125, 132
  - practical locking implications, 116
  - repeatable reads, 94, 109, 125
  - two-phase locking, 131
  - versus isolation levels, 95, 113, 116, 132
  - versus typical transactions, 103
- consolidated databases
  - setting, 35
- constant expression defaults, 75
- constraints, 77
  - columns and tables, 26
  - unique constraints, 77
- contiguous storage of rows, 337
- control statements
  - list, 531
- conventions
  - documentation, xiii
- conversion errors during import, 427
- conversion of outer joins to inner joins, 359
- converting subqueries in the WHERE clause to joins, 292
- copying
  - data with INSERT, 306
  - procedures, 514
  - tables, 49
  - views, 53
- copying databases
  - replicating data and concurrency, 138
- correlated subqueries
  - about, 285, 291
  - outer references, 286

- correlation names
  - about, 260
  - in self-joins, 248
  - in star joins, 250
  - table names, 194
- cost model
  - about, 315
- cost-based optimization, 314
- COUNT function
  - about, 211
  - NULL, 212
- CREATE DATABASE statement
  - Adaptive Server Enterprise, 388
  - using, 31, 32
- create database wizard
  - creating Transact-SQL compatible databases, 393
  - using, 30
- CREATE DEFAULT statement
  - unsupported, 388
- CREATE DOMAIN statement
  - Transact-SQL compatibility, 388
  - using, 80
- CREATE EXISTING TABLE statement
  - using, 469
- CREATE FUNCTION statement
  - about, 518
- CREATE INDEX statement
  - and concurrency, 136
- CREATE PROCEDURE statement
  - examples, 511
  - parameters, 536
- CREATE RULE statement
  - unsupported, 388
- CREATE SERVER statement
  - JDBC and Adaptive Server Enterprise, 490
  - ODBC and Adaptive Server Enterprise, 494
  - remote servers, 490
- CREATE TABLE statement
  - about, 40
  - and concurrency, 136
  - foreign keys, 48
  - primary keys, 46
  - proxy tables, 470
  - Transact-SQL, 401
- CREATE TRIGGER statement
  - about, 523
- CREATE VIEW statement
  - WITH CHECK OPTION clause, 53
- creating
  - column defaults, 70
  - data types, 80, 81
  - database for Windows CE, 30
  - database from SQL, 31, 32
  - database from the command line, 31
  - domains, 80, 81
  - indexes, 60
  - procedures, 511
  - procedures and functions with external calls, 562
  - remote procedures, 511
  - tables, 40
  - triggers, 523
  - user-defined functions, 518
  - views, 51
- creating a database, 29
- creating a proxy table with the CREATE TABLE statement, 470
- creating a Transact-SQL-compatible database, 393
- creating compatible tables, 401
- creating databases
  - Sybase Central, 30
- creating external logins, 465
- creating proxy tables in SQL, 469
- creating proxy tables in Sybase Central, 468
- creating remote procedures, 476
- creating remote servers, 460
- cross joins, 239
- cross products, 239
- current date and time defaults, 71
- cursor instability, 95
- cursor management
  - overview, 545

- cursors
  - and LOOP statement, 545
  - in procedures, 545
  - instability, 95
  - on SELECT statements, 545
  - procedures and triggers, 545
  - stability, 95
  - updating in joins, 234

- customizing graphical plans, 373

- cyclical blocking conflict, 101

## D

- data
  - adding, changing, and deleting, 301
  - case sensitivity, 397
  - consistency, 94
  - exporting, 433
  - formats for importing and exporting, 424
  - importing, 424, 429
  - importing and exporting, 422
  - integrity and correctness, 102
  - invalid, 66
  - permissions required to modify data, 302
  - viewing, 44

- data consistency
  - assuring using locks, 121
  - correctness, 102
  - dirty reads, 94, 125
  - dirty reads tutorial, 106
  - ISO SQL/92 standard, 94
  - phantom rows, 94, 113, 125, 132
  - practical locking implications, 116
  - repeatable reads, 94, 109, 125
  - two-phase locking, 131

- data definition
  - concurrency, 135

- data definition language
  - about, 28

- data definition statements
  - and concurrency, 136

- data entry
  - and isolation levels, 103

- data formats
  - for importing and exporting, 424

- data integrity
  - about, 66
  - checking, 87
  - column constraints, 26
  - column defaults, 70
  - constraints, 68, 76
  - effects of unserializable schedules on, 103
  - enforcing, 83
  - information in the system tables, 88
  - losing, 85
  - tools, 67

- data migration wizard
  - about, 449
  - using, 449

- data model normalization, 16

- data modeling
  - about, 3

- data modification statements
  - about, 302

- data organization
  - physical, 337

- data sources
  - external servers, 492

- data types, 80
  - aggregate functions, 210
  - assigning columns, 80
  - choosing, 25
  - creating, 80, 81
  - deleting, 82
  - remote procedures, 477
  - SQL and C, 567
  - timestamp, 398
  - UNION operation, 223

- database administrator
  - roles, 390

- database design concepts, 5

- database files
  - fragmentation, 150, 168
  - growing, 443
  - growing after deletes, 443
  - performance, 147

- database objects
  - editing properties, 34

database options  
    setting for Transact-SQL compatibility, 395

database procedures  
    viewing profiling data, 172

database statistics  
    about, 166

database threads  
    blocked, 101

databases  
    case sensitivity, 394, 396  
    connections, 586  
    creating, 29  
    creating for Windows CE, 30  
    creating from SQL, 31, 32  
    creating from Sybase Central, 30  
    creating from the command line, 31  
    deleting, 32  
    design concepts, 5  
    designing, 3  
    disconnecting from databases, 33  
    displaying system objects, 35  
    displaying system tables, 50  
    erasing, 32  
    erasing from the command line, 32  
    exporting, 437  
    file compatibility, 29  
    importing, 429  
    initializing, 29  
    initializing from SQL, 31, 32  
    initializing from Sybase Central, 30  
    installing jConnect metadata support, 37  
    Java classes, 4  
    migrating to Adaptive Server Anywhere, 449  
    normalizing, 16  
    rebuilding, 444  
    reloading, 444  
    setting a consolidated database, 35  
    setting options, 34  
    starting without connecting, 36  
    transaction log, 29  
    Transact-SQL compatibility, 393  
    unloading, 437  
    unloading and reloading, 440, 444, 445, 447  
    upgrading database file format, 442  
    verifying design, 23  
    viewing and editing properties, 34  
    working with, 29  
    working with objects, 27

DataWindows  
    remote data access, 457

dates  
    entry rules, 201  
    procedures and triggers, 560  
    searching for, 201

DB\_PROPERTY function  
    about, 162

DB2 remote data access, 495

db2odbc server class, 495

DBASE format for importing and exporting, 424

dberase utility, 32  
    using, 32

dbinit utility  
    using, 31

dbisql utility  
    rebuilding databases, 441

dbo user ID  
    Adaptive Server Enterprise, 389

dbspaces  
    managing, 388

dbunload utility, 441  
    exporting data, 433

DDL  
    about, 28

deadlocks  
    about, 100  
    reasons for, 101  
    transaction blocking, 101

debugger [dbprdbg] utility  
    common tasks, 585  
    configuring, 595  
    connecting to a database, 586  
    debugging Java classes, 579  
    debugging stored procedures, 576  
    displaying source code, 580  
    examining variables, 593  
    features, 572  
    requirements, 573  
    starting, 574, 586  
    tutorial, 574  
    working with breakpoints, 589



- debugger features, 572
- debugging
  - about, 571
  - breakpoints, 582
  - choosing connections, 587
  - compiling classes, 579
  - connecting, 574, 586
  - features, 572
  - getting started, 574
  - introduction, 572
  - Java, 579
  - local variables, 578, 583
  - permissions, 573
  - requirements, 573
  - stored procedures, 576
  - tutorial, 576, 579
- debugging logic in the database, 571
- decision support
  - and isolation levels, 103
- declarations in compound statements, 533
- DECLARE statement
  - compound statements, 533
  - procedures, 545, 549
- declaring parameters for procedures, 536
- default error handling in procedures and triggers, 548
- default handling of warnings in procedures and triggers, 552
- defaults
  - AUTOINCREMENT, 72
  - column, 70
  - constant expressions, 75
  - creating, 70
  - creating in Sybase Central, 71
  - current date and time, 71
  - INSERT statement and, 304
  - NEWID, 73
  - NULL, 74
  - string and number, 74
  - Transact-SQL, 388
  - user ID, 72
  - using in domains, 81
  - with transactions and locks, 136
- defragmenting
  - about, 168
  - all tables in a database, 169
  - hard disk, 168
  - individual tables in a database, 170
- delaying referential integrity checks, 128
- DELETE statement
  - locking during, 129
  - using, 311
- deleting
  - column defaults, 71
  - data types, 82
  - database files, 32
  - domains, 82
  - indexes, 61
  - procedures, 515
  - tables, 43
  - triggers, 527
  - user-defined data types, 82
  - views, 56
- deleting all rows from a table, 312
- deleting and deleting CHECK conditions, 78
- deleting data
  - about, 301
  - DELETE statement, 311
  - TRUNCATE TABLE statement, 312
- deleting remote servers, 461
- delimit statements within your procedure, 559
- depth
  - item in access plans, 369
- derived tables
  - in joins, 254
  - in key joins, 268
  - in natural joins, 257
  - in outer joins, 245
- descending order
  - ORDER BY clause, 220
- design process, 11
- designing databases
  - about, 3
  - concepts, 5
  - procedure, 11
- designing the database table properties, 25

designing your database, 3

devices

managing, 388

difference between FALSE and UNKNOWN, 204

differences from other SQL dialects, 415

direction

item in access plans, 369

dirty reads, 94, 125

tutorial, 106

versus isolation levels, 95

disabling breakpoints, 589

disabling procedure profiling

SQL, 173

Sybase Central, 173

DISCONNECT statement

using, 33

disconnecting

from databases, 33

other users from a database, 33

discovery of exploitable conditions, 360

disk access cost model

about, 315

disk allocation for inserted rows, 337

disk space

reclaiming, 443

DISK statements

unsupported, 388

DiskRead

statistic in access plans, 365

DiskReadIndInt

statistic in access plans, 365

DiskReadIndLeaf

statistic in access plans, 365

DiskReadTable

statistic in access plans, 365

DiskWrite

statistic in access plans, 365

displaying system objects in a database, 35

displaying system tables, 50

DISTINCT clause

SELECT statement, 193

unnecessary distinct elimination, 352

distinct elimination

about, 352

DISTINCT keyword

aggregate functions, 208, 211

distinct list

item in access plans, 370

DLLs

calling from procedures, 562

DML

about, 302

documentation

conventions, xiii

SQL Anywhere Studio, x

documents

inserting, 306

domain creation wizard

using, 80

domains

assigning columns, 80

case-sensitivity, 397

CHECK conditions, 77

creating, 80, 81

deleting, 82

examples of uses, 80

using, 80

double quotes

character strings, 201

DROP CONNECTION statement

using, 33

DROP DATABASE statement

Adaptive Server Enterprise, 388

using, 32

DROP statement

and concurrency, 136

DROP TABLE statement

example, 43

DROP TRIGGER statement

about, 527

DROP VIEW statement  
example, 56

dropping  
domains, 82  
indexes, 61  
procedures, 515  
tables, 43  
triggers, 527  
views, 56

dropping external logins, 466

dropping remote procedures, 477

DUMP DATABASE statement  
unsupported, 388

DUMP TRANSACTION statement  
unsupported, 388

duplicate correlation names in joins (star joins), 250

duplicate elimination  
query execution algorithms, 332

duplicate results  
eliminating, 193

duplicate rows  
removing with UNION, 223

dynamic cache sizing  
about, 152  
about UNIX, 154  
about Windows, 154

## E

early release of locks, 104, 131  
an exception, 132

editing  
properties of database objects, 34  
table data, 44

editing breakpoint conditions, 590

effects of  
transaction scheduling, 103  
unserializable transaction scheduling, 103

efficiency  
improving and locks, 105  
improving using indexes, 133

eliminating duplicate query results, 193

elimination of unnecessary case translation, 362

enabling breakpoints, 589

enabling procedure profiling  
SQL, 172  
Sybase Central, 172

enabling VI keys in entry-text windows  
debugger [dbprdbg] utility, 595

encryption  
cache size, 144  
hiding objects, 568

ending transactions, 91

enforcing column uniqueness, 345

enforcing entity and referential integrity, 83

enforcing referential integrity, 84

ensuring compatible object names, 397

ensuring data integrity, 65

entities  
about, 5  
attributes, 5  
choosing, 11  
definition of, 5  
forcing integrity, 83

entity integrity, 416  
breached by client application, 83

entity-relationship diagrams  
about, 5  
reading, 8

equals operator  
comparison operator, 196

equijoins  
about, 237

erase database wizard  
using, 32

erase utility  
using, 32

erasing databases, 32

- error handling
  - ON EXCEPTION RESUME, 550
  - procedures and triggers, 548
- error handling in Transact-SQL procedures, 412
- errors
  - conversion, 427
  - procedures and triggers, 548
  - Transact-SQL, 412, 413
- errors and warnings in procedures and triggers, 548
- EstCpuTime
  - estimate in access plans, 367
- EstDiskReads
  - estimate in access plans, 367
- EstDiskReadTime
  - estimate in access plans, 367
- EstDiskWrites
  - estimate in access plans, 367
- estimated leaf pages
  - item in access plans, 369
- estimated pages
  - item in access plans, 368
- estimated row size
  - item in access plans, 368
- estimated rows
  - item in access plans, 368
- EstRowCount
  - estimate in access plans, 367
- EstRunTime
  - estimate in access plans, 367
- events
  - viewing individual profiling information, 177
  - viewing summary profiling data, 176
- examining variables
  - debugger [dbprdbg] utility, 593
- examples
  - dirty reads, 106
  - implications of locking, 116
  - non-repeatable read, 110
  - non-repeatable reads, 109
  - phantom locks, 116
  - phantom rows, 113
- Excel and remote access, 500
- exception handlers
  - nested compound statements, 555
  - procedures and triggers, 553
- exceptions
  - declaring, 549
- exclusive locks, 122
- exclusive versus shared locks, 123
- EXECUTE IMMEDIATE statement
  - procedures, 557
- executing triggers, 525
- existence test
  - about, 284
  - negation of, 285
- EXISTS operator, 284
- EXISTS predicates
  - rewriting subqueries as, 320
- explicit join conditions, 231
- explicit join conditions (the ON phrase), 236
- export tools, 433
- exporting
  - Adaptive Server Enterprise compatibility, 454
  - file formats, 424
  - introduction, 424
  - NULL values, 427
  - query results, 435
  - schema, 443
  - tables, 438
- exporting data
  - about, 422
  - schema, 443
  - tools, 433
- exporting databases
  - using, 437
- exporting tables
  - schema, 443
- expression SQL
  - item in access plans, 370
- expressions
  - NULL values, 204

external calls  
     creating procedures and functions, 562

external functions  
     canceling, 566  
     passing parameters, 567  
     prototypes, 564  
     return types, 567

external loading, 422

external logins  
     about, 465  
     creating, 465  
     dropping, 466

external servers  
     ODBC, 492

extract database wizard  
     about, 448

extracting  
     databases for SQL Remote, 448

## F

FALSE conditions  
     and NULL, 204

features not supported for remote data, 485

Federal Information Processing Standard Publication  
     compliance. See SQL standards

feedback  
     documentation, xvii  
     providing, xvii

FETCH statement  
     procedures, 545

fetchtst, 170

file formats  
     for importing and exporting, 424  
     rebuilding databases, 442

file fragmentation  
     about, 168

file types  
     for importing and exporting, 424

files  
     fragmentation, 150  
     performance, 147

filter and pre-filter, 336

filters  
     query execution algorithms, 336

finishing transactions, 91

FIPS compliance. See SQL standards

FIRST clause  
     about, 221  
     when not to use, 221

FIXED format for importing and exporting, 424

FOR BROWSE clause  
     unsupported, 403

FOR READ ONLY clause  
     ignored, 403

FOR statement  
     syntax, 531

FOR UPDATE clause  
     unsupported, 403

foreign keys  
     and integrity, 416  
     creating, 47, 48  
     deleting, 47  
     displaying in Sybase Central, 47  
     in key joins, 259  
     managing, 47  
     mandatory/optional, 85  
     modifying, 48  
     performance, 157  
     referential integrity, 85  
     role name, 260

formats  
     for importing and exporting, 424

FORWARD TO statement, 475

FoxPro  
     format for importing and exporting, 424  
     remote data access, 502

fragmentation  
     about, 168  
     file, 168  
     indexes, 170

- of files, tables, and indexes, 150
- tables, 168
- FROM clause
  - explanation of joins, 230
  - introduction, 194
- full compares
  - about, 341
  - statistic in access plans, 365
- full outer joins
  - about, 241
- FullCompare
  - statistic in access plans, 365
- functions
  - external, 562
  - TRACEBACK, 549
  - tsequal, 399
  - user-defined, 518
  - viewing individual profiling information, 177
  - viewing summary profiling data, 175
- G**
- general guidelines for writing portable SQL, 401
- general problems with queries
  - remote data access, 486
- generated join conditions, 231
- generated joins and the ON phrase, 236
- generating
  - physical data model, 20
- generating unique keys, 135
- global temporary tables, 63
- global variables
  - debugger [dbprdbg] utility, 593
- go
  - batch statement delimiter, 529
- GRANT statement
  - and concurrency, 136
  - Transact-SQL, 392
- graph type
  - configuring the Performance Monitor, 165
- graphical plan with statistics
  - about, 373
  - Interactive SQL, 378
  - SQL functions, 378
- graphical plans
  - about, 373
  - context sensitive help, 373
  - customizing, 373
  - Interactive SQL, 378
  - predicate, 377
  - SQL functions, 378
- graphing
  - using the Performance Monitor, 163
- greater than
  - comparison operator, 196
  - range specification, 197
- greater than or equal to
  - comparison operator, 196
- GROUP BY ALL clause
  - unsupported, 403
- GROUP BY clause
  - about, 213
  - Adaptive Server Enterprise compatibility, 225
  - aggregate functions, 213
  - execution, 214
  - order by and, 222
  - SQL standard compliance, 225
  - WHERE clause, 216
- group by list
  - item in access plans, 369
- group by with multiple columns, 216
- group reads
  - tables, 338
- grouping algorithms
  - query execution algorithms, 333
- groups
  - Adaptive Server Enterprise, 390
- GUIDs
  - default column value, 73
  - generating, 135
- gx command-line option
  - threads, 486

## H

- hash anti-semijoins
  - query execution algorithms, 331
- hash B-tree indexes
  - about, 346
- hash distinct
  - query execution algorithms, 332
- hash group by
  - query execution algorithms, 334
- hash joins
  - query execution algorithms, 329, 330, 331
- hash not exists
  - query execution algorithms, 331
- hash semijoins
  - query execution algorithms, 330
- hash values
  - indexes, 346
- HAVING clause
  - GROUP BY and, 218
  - logical operators, 219
  - performance, 350
  - selecting groups of data, 218
  - subqueries, 276
  - with and without aggregates, 218
- histograms
  - about, 315
  - updating, 317
- HOLDLOCK keyword
  - Transact-SQL, 404
- how data can become invalid, 66
- how database contents change, 67
- how joins work, 229
- how locking is implemented, 132
- how locking works, 121
- how NULL affects Transact-SQL outer joins, 247
- how queries with group by are executed, 214
- how subqueries work, 291
- how the optimizer works, 315

## I

- I/O
  - scanning bitmaps, 338
- IBM
  - remote data access to DB2, 495
- IBM DB2
  - migrating to Adaptive Server Anywhere, 449
- icons
  - used in manuals, xiv
- identifiers
  - case sensitivity, 397
  - qualifying, 185
  - uniqueness, 397
  - using in domains, 81
- identifying entities and relationships, 11
- IDENTITY column, 399
  - retrieving values, 400
- if a client application breaches entity integrity, 83
- if a client application breaches referential integrity, 85
- IF statement
  - syntax, 531
- images
  - inserting, 306
- implementation of locking, 132
- import tools, 428
- import wizard
  - about, 428
  - using, 429
- importing
  - Adaptive Server Enterprise compatibility, 454
  - databases, 429
  - file formats, 424
  - introduction, 424
  - non-matching table structures, 425
  - NULL values, 425
  - tools, 428
  - using temporary tables, 63
- importing and exporting data, 421

- ul style="list-style-type: none;">
- importing data
  - about, 422
  - conversion errors, 427
  - DEFAULTS option, 426
  - from other databases, 429
  - interactively, 429
  - LOAD TABLE statement, 431
  - performance, 422
  - proxy tables, 429
  - temporary tables, 425, 426
  - tools, 428
- improving concurrency at isolation levels 2 and 3, 104
- improving index performance, 341
- improving performance, 144
- IN conditions
  - subqueries, 282
- IN keyword
  - matching lists, 198
- IN list
  - algorithm, 326
  - item in access plans, 370
  - optimization, 357
- IN parameters
  - defined, 536
- inconsistencies
  - avoiding using locks, 121
  - dirty reads, 94, 125
  - dirty reads tutorial, 106
  - effects of unserializable schedules, 103
  - ISO SQL/92 standard, 94
  - phantom rows, 94, 113, 125, 132
  - practical locking implications, 116
- inconsistencies non-repeatable reads, 94, 109, 125
- inconsistency
  - example of non-repeatable read, 110
- increase the cache size, 144
- IndAdd
  - statistic in access plans, 365
- index creation wizard
  - using, 60
- index fragmentation, 170
- index scans
  - about, 324
- index selectivity
  - about, 341
- indexed distinct
  - query execution algorithms, 333
- indexed group by
  - query execution algorithms, 334
- indexes
  - about, 340
  - benefits and locking, 105
  - B-tree, 346
  - can usually be found to satisfy a predicate, 319
  - composite, 342, 347
  - compressed B-tree, 348
  - creating, 60
  - deleting, 61
  - effect of column order, 343
  - fan-out and page sizes, 348
  - fragmentation, 170
  - hash B-tree, 346
  - hash values, 346
  - HAVING clause performance, 350
  - improving concurrency, 133
  - inspecting, 62
  - leaf pages, 342
  - optimization and, 340
  - performance, 145, 146
  - predicate analysis, 350
  - recommended page sizes, 348
  - sargable predicates, 350
  - structure, 342
  - temporary tables, 341
  - Transact-SQL, 397
  - types of index, 346
  - validating, 60
  - WHERE clause performance, 350
  - working with, 58
- indexes in the system tables, 62
- IndLookup
  - statistic in access plans, 365
- initial cache size, 152
- initialization utility
  - using, 31
- inner and outer joins, 241



- inner joins
  - about, 241
- INOUT parameters
  - defined, 536
- INPUT statement
  - about, 428
  - using, 429
- INSERT statement
  - about, 303, 428
  - locking during, 127
  - SELECT, 303
  - using, 429
- inserting data
  - adding NULL, 304
  - BLOBs, 306
  - column data INSERT statement, 304
  - constraints, 304
  - defaults, 304
  - into all columns, 303
  - using INSERT, 303
  - with SELECT, 305
- inserting documents and images, 306
- inserting values into all columns of a row, 303
- inserting values into specific columns, 304
- installing
  - jConnect metadata support, 37
- instantiate, 170
- integrity
  - about, 66
  - checking, 87
  - column defaults, 70
  - constraints, 68, 76
  - enforcing, 83
  - information in the system tables, 88
  - losing, 85
  - tools, 67
- integrity constraints belong in the database, 66
- integrity rules in the system tables, 88
- Interactive SQL
  - command delimiter, 559
  - exporting query results, 435
  - rebuilding databases, 441
- Interactive SQL import wizard
  - about, 428
- interference between transactions, 100, 113
- interleaving transactions, 102
- internal loading, 422
- internal operations
  - remote data access, 481
- interrupt
  - Run menu, 589
  - setting, 587
- interrupting connections
  - debugger [dbprdbg] utility, 591
- interrupting execution
  - debugger [dbprdbg] utility, 591
- INTO clause
  - using, 540
- invalid data, 66
- invocations
  - statistic in access plans, 365
- IS NULL keyword, 204
- ISNULL function
  - about, 204
- ISO compliance. See SQL standards
- ISO SQL/92 standard
  - concurrency, 94
  - typical inconsistencies and, 94
- isolation level 0
  - example, 106
  - SELECT statement locking, 125
- isolation level 1
  - example, 109
  - SELECT statement locking, 125
- isolation level 2
  - example, 113, 116
  - SELECT statement locking, 125
- isolation level 3
  - example, 115
  - SELECT statement locking, 125

- isolation levels
  - about, 94
  - changing within a transaction, 98
  - choosing, 102
  - choosing types of locking tutorial, 112
  - implementation at level 0, 125
  - implementation at level 1, 125
  - implementation at level 2, 125
  - implementation at level 3, 125
  - improving concurrency at levels 2 and 3, 104
  - ODBC, 97
  - setting, 96
  - tutorials, 106
  - typical transactions for each, 104
  - versus typical inconsistencies, 95, 113, 116, 132
  - versus typical transactions, 103
  - viewing, 99

- isolation levels and consistency, 94

## J

- Java
  - about debugging, 572
  - debugging, 579

- Java classes
  - database design, 4

- Java debugger
  - about, 572
  - common tasks, 585
  - configuring, 595
  - connecting to a database, 586
  - debugging Java classes, 579
  - debugging stored procedures, 576
  - displaying source code, 580
  - examining variables, 593
  - requirements, 573
  - starting, 574, 586
  - tutorial, 574
  - working with breakpoints, 589

- jConnect metadata support
  - installing, 37

- JDBC-based server classes, 489

- join algorithms
  - about, 326

- join compatible data types, 234

- join conditions
  - types, 237

- join elimination rewrite optimization, 356

- join operators
  - Transact-SQL, 404

- joining more than two tables, 233

- joining remote tables, 472

- joining tables from multiple local databases, 474

- joining two tables, 232

- joins
  - about, 230
  - automatic, 417
  - Cartesian product, 239
  - commas, 239
  - conversion of outer joins to inner joins, 359
  - converting subqueries into, 287
  - converting subqueries to joins, 292
  - cross join, 239
  - data type conversion, 234
  - default is KEY JOIN, 231
  - derived tables, 254
  - duplicate correlation names, 250
  - equijoins, 237
  - FROM clause, 230
  - how an inner join is computed, 232
  - in delete, update and insert statements, 234
  - inner, 241
  - inner and outer, 241
  - join conditions, 231
  - join elimination rewrite optimization, 356
  - joined tables, 231
  - key, 417
  - key joins, 259
  - natural, 417
  - natural joins, 255
  - nesting, 233
  - non-ANSI joins, 234
  - null-supplying tables, 241
  - of more than two tables, 233
  - of table expressions, 233
  - of tables in different databases, 474
  - of two tables, 232
  - ON phrase, 236
  - outer, 241
  - preserved tables, 241
  - query execution algorithms, 326
  - remote tables, 472

- search conditions, 237
- self-joins, 248
- star joins, 250
- Transact-SQL, 404
- Transact-SQL outer, null values and, 247
- Transact-SQL outer, restrictions on, 246
- Transact-SQL outer, views and, 247
- updating cursors, 234
- WHERE clause, 238

## K

- key joins
  - about, 259
  - if more than one foreign key, 260
  - of lists and table expressions that do not contain commas, 266
  - of table expression lists, 264
  - of table expressions, 262
  - of table expressions that do not contain commas, 263
  - of views and derived tables, 268
  - rules, 270
  - the ON phrase, 236
  - with an ON phrase, 259
- key type
  - item in access plans, 369
- keys
  - assigning, 17
  - performance, 157
- keywords
  - HOLDLOCK, 404
  - NOHOLDLOCK, 404
  - remote servers, 485

## L

- laptop computers
  - and transactions, 138
- launching
  - the debugger [dbprdbg] utility, 586
- leaf pages, 342
- LEAVE statement
  - syntax, 531

- left outer joins
  - about, 241
- less than
  - comparison operator, 196
  - range specification, 197
- less than or equal to
  - comparison operator, 196
- levels of isolation
  - about, 94
  - changing, 96
  - changing within a transaction, 98
  - setting default, 96
- LIKE conditions
  - LIKE optimizations, 358
- LIKE operator
  - wildcards, 200
- LIKE optimizations, 358
- limiting the memory used by the cache, 152
- line breaks
  - SQL, 185
- listing remote server capabilities, 463
- listing the columns on a remote table, 471
- listing the remote tables on a server, 463
- literal values
  - NULL, 204
- LOAD DATABASE statement
  - unsupported, 388
- LOAD TABLE statement
  - about, 428
  - using, 431
- LOAD TRANSACTION statement
  - unsupported, 388
- loading saved settings from a file
  - debugger [dbprdbg] utility, 596
- local temporary tables, 63
- local variables
  - debugger [dbprdbg] utility, 593
- locked tables
  - item in access plans, 368

- locking
  - reducing through indexes, 345
- locking during deletes, 129
- locking during inserts, 127
- locking during queries, 125
- locking during updates, 128
- locks
  - about, 121
  - anti-insert, 122, 130
  - blocking, 100, 113
  - choosing isolation levels tutorial, 112
  - conflict handling, 100, 113
  - conflicting types, 124
  - deadlock, 101
  - early release of, 104, 131
  - early release of, an exception, 132
  - exclusive, 122
  - implementation at level 0, 125
  - implementation at level 1, 125
  - implementation at level 2, 125
  - implementation at level 3, 125
  - inconsistencies versus typical isolation levels, 95, 132
  - insert, 122
  - isolation levels, 94
  - nonexclusive, 122
  - objects that can be locked, 121
  - orphans and referential integrity, 128
  - phantom rows versus isolation levels, 113, 116
  - procedure for deletes, 129
  - procedure for inserts, 127
  - procedure for updates, 128
  - query execution algorithms, 336
  - read, 122, 130
  - reducing the impact through indexes, 105
  - shared versus exclusive, 123
  - transaction blocking and deadlock, 100
  - two-phase locking, 131
  - types of, 122
  - typical transactions versus isolation levels, 103
  - uses, 123
  - viewing in Sybase Central, 121
  - write, 122
- logging SQL statements, 36
- logical operators
  - connecting conditions, 205
  - HAVING clauses, 219

- logs
  - rollback log, 93
- long plans
  - about, 372
  - Interactive SQL, 378
  - SQL functions, 378
- LOOP statement
  - in procedures, 545
  - syntax, 531
- losing referential integrity, 85
- Lotus
  - format for importing and exporting, 424
- Lotus Notes
  - passwords, 502
  - remote data access, 502

## M

- maintenance
  - performance, 144
- managing
  - transactions, 479
- managing foreign keys, 47
- managing primary keys, 44
- mandatory
  - foreign keys, 85
- mandatory relationships, 8
- many-to-many relationships
  - definition of, 7
  - resolving, 20, 22
- master database
  - unsupported, 387
- matching character strings in the WHERE clause, 199
- materializing result sets
  - query processing, 160
- MAX function
  - rewrite optimization, 356
- maximum cache size, 152

- merge joins
  - query execution algorithms, 331
- merge sort
  - query execution algorithms, 335
- MESSAGE statement
  - procedures, 549
- metadata support
  - installing for jConnect, 37
- Microsoft Access
  - migrating to Adaptive Server Anywhere, 449
  - remote data access, 501
- Microsoft Excel
  - remote data access, 500
- Microsoft FoxPro
  - remote data access, 502
- Microsoft SQL Server and remote access, 498
- Microsoft SQL Server databases
  - migrating to Adaptive Server Anywhere, 449
- migrating databases
  - about, 449
- MIN function
  - rewrite optimization, 356
- minimal administration work
  - optimizer, 318
- minimizing downtime during rebuilding, 447
- minimum cache size, 152
- modifying
  - CHECK conditions, 78
  - column defaults, 71
  - views, 55
- modifying and deleting CHECK conditions, 78
- modifying and deleting column defaults, 71
- monitor
  - configuring, 165
  - opening the Sybase Central Performance Monitor, 164
  - Performance Monitor overview, 163
- monitoring and improving performance, 143
- monitoring cache size, 155

- monitoring database performance, 162
- monitoring database statistics from Sybase Central, 163
- monitoring database statistics from Windows Performance Monitor, 165
- monitoring performance
  - abbreviations used in access plans, 364
  - reading access plans, 364
  - tools to measure queries, 170
- monitoring query performance, 170
- more than one transaction at once, 92
- msodbc server class, 498
- multiple databases
  - joins, 474
- multiple transactions
  - concurrency, 92

## N

- name spaces
  - indexes, 397
  - triggers, 397
- naming and nesting savepoints, 93
- natural joins
  - about, 255
  - of table expressions, 256
  - of views and derived tables, 257
  - with an ON phrase, 256
- nested block join and sorted block, 328
- nested block joins
  - query execution algorithms, 328
- nested compound statements and exception handlers, 555
- nested loops anti-semijoins
  - query execution algorithms, 328
- nested loops joins
  - query execution algorithms, 326
- nested loops semijoins
  - query execution algorithms, 327

nested subqueries  
about, 289

nesting  
derived tables in joins, 254  
in joins, 233  
in outer joins, 244

nesting and naming savepoints, 93

NEWID  
when to use, 135

NEWID function  
default column value, 73

newsgroups  
technical support, xvii

NLMs  
calling from procedures, 562

NOHOLDLOCK keyword  
ignored, 404

non-ANSI joins, 234

non-dirty reads  
tutorial, 106

nonexclusive locks, 122

non-repeatable reads  
about, 94  
example, 110  
isolation levels, 95, 132  
tutorial, 109

normal forms  
first normal form, 18  
normalizing database designs, 16  
second normal form, 19  
third normal form, 19

normalization  
about, 16  
performance benefits, 145

normalize your table structure, 145

not equal to  
comparison operator, 196

not greater than  
comparison operator, 196

NOT keyword  
search conditions, 197  
using, 205

not less than  
comparison operator, 196

Notes and remote access, 502

NULL  
about, 202  
aggregate functions, 212  
allowing in columns, 26  
column default, 395  
column definition, 204  
comparing, 203  
default, 74  
default parameters, 203  
DISTINCT clause, 193  
output, 427  
properties, 204  
sort order, 221  
Transact-SQL compatibility, 402  
Transact-SQL outer joins, 247

null-supplying tables  
in outer joins, 241

## O

objects  
hiding, 568

objects that can be locked, 121

obtaining database statistics from a client  
application, 162

occasionally connected users  
replicating data and concurrency, 138

ODBC  
applications, 97  
applications, and locking, 97  
external servers, 492

ODBC server classes, 492, 500

odbcfet, 170

ON clause  
joins, 236

- ON EXCEPTION RESUME clause
  - about, 550
  - not with exception handling, 554
  - stored procedures, 548
  - Transact-SQL, 413
- ON phrase
  - joins, 236
- one-to-many relationships
  - definition of, 7
  - resolving, 20
- one-to-one relationships
  - definition of, 7
  - resolving, 20
- OPEN statement
  - procedures, 545
- opening the Sybase Central Performance Monitor, 164
- operators
  - arithmetic, 192
  - connecting conditions, 205
  - NOT keyword, 197
  - precedence, 192
- optimization
  - cost based, 314
  - reading access plans, 364
- optimization for minimum or maximum functions, 356
- optimization goal
  - in access plans, 368
- optimization of queries
  - about, 314
  - assumptions, 318
  - reading access plans, 364
  - rewriting subqueries as EXISTS predicates, 320
  - steps in, 322
- optimizations
  - using indexes, 133
- optimize for first rows or for entire result set, 319
- optimizer
  - about, 314, 315
  - assumptions, 318
  - predicate analysis, 350
  - role of, 314
  - semantic subquery transformations, 349
  - semantic transformations, 352
- optimizer estimates
  - about, 315
- optional foreign keys, 85
- optional relationships, 8
- options
  - BLOCKING, 100
  - DEFAULTS, 426
  - ISOLATION\_LEVEL, 96
  - setting database options, 34
- or (!) bitwise operator
  - using, 205
- Oracle and remote access, 497
- Oracle databases
  - migrating to Adaptive Server Anywhere, 449
- oraodbc server class, 497
- ORDER BY and GROUP BY, 222
- ORDER BY clause
  - GROUP BY, 222
  - limiting results, 221
  - performance, 159
  - sorting query results, 220
- order-by
  - item in access plans, 370
- ordered distinct
  - query execution algorithms, 332
- ordered group by
  - query execution algorithms, 334
- ordering of transactions, 102
- organization
  - of data, physical, 337
- organizing query results into groups, 213
- orphan and referential integrity, 128
- other uses for indexes, 345
- OUT parameters
  - defined, 536

- outer joins
  - about, 241
  - and join conditions, 243
  - complex, 244
  - in Transact-SQL, 245
  - of views and derived tables, 245
  - restrictions, 246
  - star join example, 252
  - Transact-SQL, 404
  - Transact-SQL, restrictions on, 246
  - Transact-SQL, views and, 247

- outer references
  - about, 286
  - aggregate functions, 209
  - HAVING clause, 276

- output redirection, 435

- OUTPUT statement, 435
  - about, 433

- outputting nulls, 427

## P

- page map
  - item in access plans, 368

- page maps
  - scanning, 338

- page size
  - about, 338
  - and indexes, 339
  - disk allocation for inserted rows, 337
  - performance, 146

- page sizes
  - and indexes, 348

- pages
  - disk allocation for inserted rows, 337

- parentheses
  - in arithmetic statements, 192
  - UNION operators, 223

- partial index scan
  - about, 345

- partial passthrough of the statement
  - remote data access, 483

- particular concurrency issues, 135

- passing parameters to external functions, 567

- passing parameters to functions, 537

- passing parameters to procedures, 537

- passwords
  - case sensitivity, 397
  - Lotus Notes, 502

- performance
  - about, 144
  - autocommit, 149
  - automatic tuning, 318
  - bulk loading, 422
  - bulk operations, 150
  - cache, 144
  - cache read-hit ratio, 376
  - comparing optimizer estimates and actual statistics, 375
  - compression, 151
  - database design, 145
  - estimate source, 376
  - file fragmentation, 150, 168
  - file management, 147
  - improving versus locks, 105
  - index fragmentation, 170
  - indexes, 58, 146
  - keys, 157
  - measuring query speed, 170
  - monitoring, 162
  - monitoring in Windows, 165
  - page size, 146
  - predicate analysis, 350
  - reading access plans, 364
  - recommended page sizes, 348
  - reducing requests, 151
  - runtime actuals and estimates, 375
  - scattered reads, 147
  - selectivity, 375
  - statistics in Windows Performance Monitor, 165
  - table and page size, 338
  - table fragmentation, 168
  - tips, 144
  - transaction log, 144
  - WITH EXPRESS CHECK, 150
  - work tables, 160

- performance considerations for moving data, 422

- Performance Monitor
  - adding and removing statistics, 164
  - configuring, 165
  - opening Sybase Central, 164



- overview, 163
  - running multiple copies, 166
  - setting the interval time, 165
  - starting, 166
  - Sybase Central, 163
  - Windows, 165
- PerformanceFetch, 170
- PerformanceInsert, 170
- PerformanceTraceTime, 170
- PerformanceTransaction, 170
- permissions
- Adaptive Server Enterprise, 390
  - data modification, 302
  - debugging, 573, 586
  - procedure result sets, 541
  - procedures calling external functions, 562
  - triggers, 527
  - user-defined functions, 521
- permissions for data modification, 302
- phantom
- rows, 94, 125, 132
- phantom locks, 116
- phantom rows
- tutorial, 113
  - versus isolation levels, 95, 116, 132
- physical data model
- generating, 20
- physical data organization, 337
- place different files on different devices, 147
- plans
- abbreviations used in, 364
  - caching of, 322
  - context sensitive help, 373
  - customizing, 373
  - graphical plans, 373
  - Interactive SQL, 378
  - long text plans, 372
  - reading, 364
  - short text plans, 371
  - SQL functions, 378
- plus operator
- NULL values, 204
- portable computers
- replicating databases, 138
- portable SQL, 401
- PowerBuilder
- remote data access, 457
- practical locking implications tutorial, 116
- predicate analysis
- about, 350
- predicate pushdown into grouped or unioned views, 355
- predicates
- about, 350
  - item in access plans, 369
  - performance, 350
  - reading in access plans, 377
- PREPARE statement
- remote data access, 479
- preserved tables
- in outer joins, 241
- primary keys
- and integrity, 416
  - AUTOINCREMENT, 72
  - concurrency, 135
  - creating, 45, 46
  - entity integrity, 84
  - generation, 135
  - managing, 44
  - modifying, 45, 46
  - performance, 157
  - using NEWID to create UUIDs, 73
- primary keys enforce entity integrity, 84
- procedure creation wizard
- about, 511
  - using, 511
- procedure language
- overview, 406
- procedure profiling
- clearing in Sybase Central, 173
  - clearing with SQL, 173
  - disabling in Sybase Central, 173
  - disabling with SQL, 173
  - enabling in Sybase Central, 172
  - enabling with SQL, 172

- events, 176, 177
  - information for individual procedures, 176, 179
  - resetting in Sybase Central, 173
  - resetting with SQL, 173
  - stored procedures and functions, 175, 177
  - summary data, 175
  - summary of procedures, 178
  - triggers, 176, 177
  - viewing data in Interactive SQL, 178
  - viewing data in Sybase Central, 172, 175
- procedures
- about, 507
  - adding remote procedures, 476
  - altering, 513
  - benefits of, 510
  - calling, 514
  - command delimiter, 559
  - copying, 514
  - creating, 511
  - cursors, 545
  - dates, 560
  - default error handling, 548
  - deleting, 515
  - deleting remote procedures, 477
  - error handling, 412, 413, 548
  - exception handlers, 553
  - EXECUTE IMMEDIATE statement, 557
  - external functions, 562
  - multiple result sets from, 542
  - overview, 509
  - parameters, 536, 537
  - permissions for result sets, 541
  - result sets, 516, 541
  - return values, 412
  - returning results, 539
  - returning results from, 515
  - savepoints, 558
  - security, 510
  - SQL statements allowed, 535
  - structure, 535
  - table names, 559
  - times, 560
  - tips for writing, 559
  - Transact-SQL, 409
  - Transact-SQL overview, 406
  - translation, 409
  - using, 511
  - using cursors in, 545
  - variable result sets from, 543
  - verifying input, 560
  - warnings, 552
- profiling database procedures, 172
- profiling information
- events, 177
  - stored procedures and functions, 177
  - triggers, 177
- properties
- setting all database object properties, 34
- properties of NULL, 204
- property
- definition of, 5
- PROPERTY function
- about, 162
- property sheets, 34
- protocols
- two-phase locking, 131
- prototypes
- external functions, 564
- proxy table creation wizard
- using, 468
- proxy tables
- about, 458, 467
  - creating, 458, 468, 469, 470
  - location, 467
- publications
- data replication and concurrency, 138

## Q

- qualifications
- about, 195
- qualified names
- database objects, 185
- quantified comparison test, 293
- subqueries, 279
- queries
- about, 184
  - access plans, 364
  - optimization, 314, 315

- selecting data from a table, 183
    - Transact-SQL, 402
  - queries blocked on themselves
    - remote data access, 486
  - query execution algorithms, 324
    - abbreviations used in access plans, 364
    - anti-semijoins, 328, 331
    - block nested loops joins, 328
    - duplicate elimination, 332
    - filter and pre-filter, 336
    - grouping algorithms, 333
    - hash anti-semijoins, 331
    - hash distinct, 332
    - hash group by, 334
    - hash joins, 329, 330, 331
    - hash not exists, 331
    - hash semijoins, 330
    - IN list, 326
    - index scans, 324
    - indexed distinct, 333
    - indexed group by, 334
    - joins, 326
    - locks, 336
    - merge joins, 331
    - merge sort, 335
    - nested block joins, 328
    - nested loops anti-semijoins, 328
    - nested loops joins, 326
    - nested loops semijoins, 327
    - ordered distinct, 332
    - ordered group by, 334
    - row limits, 336
    - semijoins, 326, 327, 330
    - sequential table scans, 325
    - single group by, 334
    - sorted block joins, 328
    - sorting and unions, 335
    - union all, 335
  - query normalization
    - remote data access, 481
  - query optimization and execution, 313
  - query parsing
    - remote data access, 481
  - query performance
    - reading access plans, 364
  - query preprocessing
    - remote data access, 481
  - query results
    - exporting, 435
  - quotation marks
    - Adaptive Server Enterprise, 201
    - character strings, 201
  - QUOTED\_IDENTIFIER option
    - about, 201
    - setting for Transact-SQL compatibility, 395
- ## R
- RAISERROR statement
    - ON EXCEPTION RESUME, 413
    - Transact-SQL, 413
  - range queries, 197
  - read locks, 122, 130
  - reading access plans, 364
  - reading entity-relationship diagrams, 8
  - reading the access plan
    - key statistics, 375
  - rebuild file formats, 442
  - rebuild tools, 441
  - rebuilding, 444
    - databases, 444
    - minimizing downtime, 447
    - purpose, 442
    - replicating databases, 445
    - tools, 441
  - rebuilding databases
    - about, 440
  - recommended page sizes, 348
  - redirecting
    - output to files, 435
  - reduce the number of requests between client and server, 151
  - reducing the impact of locking, 105
  - references
    - displaying references from other tables, 47

- ul style="list-style-type: none;">
- referential integrity, 416
  - about, 66
  - actions, 86
  - breached by client application, 85
  - checking, 87
  - column defaults, 70
  - constraints, 68, 76
  - enforcing, 83
  - information in the system tables, 88
  - losing, 85
  - orphans, 128
  - tools, 67
  - verification at commit, 128
- referential integrity actions
  - implemented by system triggers, 86
- reflexive relationships, 9
- relationships
  - about, 6
  - cardinality of, 7
  - changing to an entity, 9
  - choosing, 11
  - definition of, 5
  - mandatory versus optional, 8
  - many-to-many, 7
  - one-to-many, 7
  - one-to-one, 7
  - reflexive, 9
  - resolving, 20
  - roles, 7
- reloading
  - databases, 444
- reloading databases
  - about, 440
- remember to delimit statements within your
  - procedure, 559
- remote data
  - location, 467
- remote data access
  - about, 455
  - case sensitivity, 485
  - internal operations, 481
  - introduction, 456
  - passthrough mode, 475
  - PowerBuilder DataWindows, 457
  - remote servers, 460
  - SQL Remote unsupported, 485
  - troubleshooting, 485
  - unsupported features, 485
- remote databases
  - replication, 138
- remote procedure calls
  - about, 476
- remote procedure creation wizard
  - using, 476, 512
- remote procedures
  - adding, 476
  - calls, 476
  - creating, 511
  - data types, 477
  - deleting, 477
- remote server creation wizard
  - using, 461
- remote servers
  - about, 460
  - altering, 462
  - classes, 487, 488
  - creating, 460
  - deleting, 461
  - external logins, 465
  - listing properties, 463
  - transaction management, 479
- remote table mappings, 458
- remote tables
  - about, 458
  - accessing, 455
  - listing, 463
  - listing columns, 471
- remote transaction management overview, 479
- renaming columns in query results, 189
- repeatable reads, 94, 109, 125
- replication
  - concurrency, 138
  - concurrency issues, 135
  - rebuilding databases, 440, 445
- replication and concurrency, 138
- requests
  - reducing number of, 151
- requirements for using the debugger, 573

- reserved words
  - remote servers, 485
- resetting procedure profiling
  - SQL, 173
  - Sybase Central, 173
- RESIGNAL statement
  - about, 554
- resolving
  - relationships, 20
- RESTRICT action
  - about, 86
- restrictions
  - remote data access, 485
- restrictions on transaction management, 479
- result sets
  - limiting the number of rows, 221
  - multiple, 542
  - permissions, 541
  - procedures, 516, 541
  - Transact-SQL, 410
  - variable, 543
- retrieving the first few rows of a query, 221
- RETURN statement
  - about, 539
- return types
  - external functions, 567
- return values
  - procedures, 412
- returning a value using the RETURN statement, 539
- returning multiple result sets from procedures, 542
- returning procedure results in parameters, 515
- returning procedure results in result sets, 516
- returning result sets from procedures, 541
- returning result sets from Transact-SQL procedures, 410
- returning results as procedure parameters, 539
- returning results from procedures, 539
- returning variable result sets from procedures, 543
- REVOKE statement
  - and concurrency, 136
  - Transact-SQL, 392
- rewrite optimization
  - about, 352
- rewriting subqueries as EXISTS predicates, 320
- right outer joins
  - about, 241
- role names
  - about, 260
- role of the optimizer, 314
- roles
  - Adaptive Server Enterprise, 389
  - definition of, 7
- rollback logs
  - savepoints, 93
- ROLLBACK statement
  - compound statements, 533
  - procedures and triggers, 558
  - triggers, 407
- rolling back transactions, 91
- row limit count
  - item in access plans, 370
- row limits
  - query execution algorithms, 336
- rows
  - copying with INSERT, 306
  - deleting, 338
  - selecting, 195
- RowsReturned
  - statistic in access plans, 365
- rules
  - Transact-SQL, 388
- rules describing the operation of key joins, 270
- RunTime
  - statistic in access plans, 365

**S**

- SA\_DEBUG group, 586
  - debugger [dbprdbg] utility, 573
- sa\_migrate system procedure
  - about, 449
  - using, 450
- sa\_migrate\_create\_fks system procedure
  - using, 451
- sa\_migrate\_create\_remote\_fks\_list system procedure
  - using, 451
- sa\_migrate\_create\_remote\_table\_list system procedure
  - using, 451
- sa\_migrate\_create\_tables system procedure
  - using, 451
- sa\_migrate\_data system procedure
  - using, 451
- sa\_migrate\_drop\_proxy\_tables system procedure
  - using, 451
- sample database
  - about asademo.db, xvi
  - schema for asademo.db, 228
- sargable predicates
  - about, 350
- savepoints
  - nesting and naming, 93
  - procedures and triggers, 558
  - within transactions, 93
- saving the positions of your windows
  - debugger [dbprdbg] utility, 595
- saving transaction results, 91
- saving your settings to a file
  - debugger [dbprdbg] utility, 596
- scalar aggregates, 209
- scattered reads
  - performance, 147
- schedules
  - effects of serializability, 103
  - effects of unserializable, 103
  - serializable, 102
  - serializable versus early release of locks, 132
  - two-phase locking, 131
- scheduling of transactions, 102
- schema
  - exporting, 443
- security
  - hiding objects, 568
  - procedures, 510
- select list
  - about, 187
  - aliases, 189
  - calculated columns, 191
  - in access plans, 368
  - UNION operation, 223
  - UNION statements, 223
- SELECT statement
  - about, 184
  - aliases, 189
  - character data, 201
  - column headings, 189
  - column order, 188
  - cursors, 545
  - INSERT from, 303
  - INTO clause, 540
  - keys and query access, 157
  - specifying rows, 195
  - strings in display, 190
  - Transact-SQL, 402
  - variables, 404
- selecting all columns from a table, 187
- selecting specific columns from a table, 188
- selectivity
  - item in access plans, 369
  - reading in access plans, 377
- selectivity estimates
  - reading in access plans, 377
  - using a partial index scan, 345
- SELF\_RECURSION option
  - Adaptive Server Enterprise, 407
- self-joins, 248
- semantic query transformations, 349
- semantic transformations
  - about, 352

- semicolon
  - command delimiter, 559
- semijoins
  - query execution algorithms, 326, 327, 330
- sending native statements to remote servers, 475
- separate primary and foreign key indexes, 158
- sequential scans
  - about, 325
  - disk allocation and performance, 337
- sequential table scans
  - about, 325
  - disk allocation and performance, 337
- serializable schedules
  - about, 102
  - effect of, 103
  - two-phase locking, 131
  - versus early release of locks, 132
- server capabilities
  - remote data access, 481
- server classes
  - about, 459
  - asajdbc, 489
  - asaodbc, 493
  - asejdbc, 490
  - aseodbc, 493
  - db2odbc, 495
  - defining, 458
  - msodbc, 498
  - ODBC, 492, 500
  - oraodbc, 497
- server classes for remote data access, 487
- server side loading, 422
- servers
  - graphing with the Performance Monitor, 163
  - starting a database without connecting, 36
- servers and databases
  - compatibility, 387
- SET clause
  - UPDATE statement, 308
- SET DEFAULT action
  - about, 86
- set membership test, 297
  - =ANY, 282
  - negation of, 283
- SET NULL action
  - about, 86
- SET OPTION statement
  - Transact-SQL, 395
- setting breakpoints
  - debugger [dbprdbg] utility, 589
- setting database options, 34
- setting options for Transact-SQL compatibility, 395
- setting properties for database objects, 34
- setting the isolation level, 96
- setting the isolation level from an ODBC-enabled application, 97
- settings
  - debugger [dbprdbg] utility, 595
- shared objects
  - calling from procedures, 562
- shared versus exclusive locks, 123
- short plans
  - about, 371
  - Interactive SQL, 378
  - SQL functions, 378
- SIGNAL statement
  - procedures, 549
  - Transact-SQL, 413
- single group by
  - query execution algorithms, 334
- sort order
  - ORDER BY clause, 220
- sorted block joins
  - query execution algorithms, 328
- sorting
  - query execution algorithms, 335
  - with an index, 159
- sorting and unions, 335
- sorting query results, 159

- source code window
  - setting breakpoints, 589
- sp\_addgroup system procedure
  - Transact-SQL, 391
- sp\_addlogin system procedure
  - support, 387
  - Transact-SQL, 391
- sp\_adduser system procedure
  - Transact-SQL, 391
- sp\_bindefault procedure
  - Transact-SQL, 388
- sp\_bindrule procedure
  - Transact-SQL, 389
- sp\_changegroup system procedure
  - Transact-SQL, 391
- sp\_dboption system procedure
  - Transact-SQL, 395
- sp\_dropgroup system procedure
  - Transact-SQL, 391
- sp\_droplogin system procedure
  - Transact-SQL, 391
- sp\_dropuser system procedure
  - Transact-SQL, 391
- special IDENTITY column, 399
- special Transact-SQL timestamp column and data type, 398
- specialized joins, 248
- specifying a consolidated database, 35
- specifying dates and times in procedures, 560
- specifying proxy table locations, 467
- SQL
  - entering, 185
- SQL 3 compliance. See SQL standards
- SQL 92 compliance. See SQL standards
- SQL 99 compliance. See SQL standards
- SQL Anywhere Studio
  - documentation, x
- SQL queries, 185
- SQL Remote
  - remote data access, 485
  - replicating and concurrent transactions, 138
- SQL Server and remote access, 498
- SQL standards, 3
  - compliance, 415
  - GROUP BY clause, 225
  - non-ANSI joins, 234
- SQL statements
  - logging in Sybase Central, 36
  - writing compatible SQL statements, 401
- SQL statements allowed in procedures and triggers, 535
- SQL statements for implementing integrity constraints, 68
- SQL/3 compliance. See SQL standards
- SQL/92 compliance. See SQL standards
- SQL/99 compliance. See SQL standards
- SQL3 compliance. See SQL standards
- SQL-3 compliance. See SQL standards
- SQL-92 compliance. See SQL standards
- SQL-99 compliance. See SQL standards
- SQLCA.lock
  - selecting isolation levels, 97
  - versus isolation levels, 95
- SQLCODE variable
  - introduction, 548
- SQLSTATE variable
  - introduction, 548
- standard output
  - redirecting to files, 435
- standards. See SQL standards
- standards and compatibility. See SQL standards
- star joins, 250
- starting
  - the debugger [dbprdbg] utility, 586
- starting a database without connecting, 36
- statement-level triggers, 406



## statements

- CALL, 509, 514, 531, 537
- CASE, 531
- CLOSE, 545
- COMMIT, 533, 558
- compound, 533
- CREAT DEFAULT, 388
- CREATE DATABASE, 388
- CREATE DOMAIN, 388
- CREATE PROCEDURE, 511, 536
- CREATE RULE, 388
- CREATE TABLE, 401
- CREATE TRIGGER, 523
- DECLARE, 533, 545, 549
- DISK, 388
- DROP DATABASE, 388
- DUMP DATABASE, 388
- DUMP TRANSACTION, 388
- EXECUTE IMMEDIATE, 557
- FETCH, 545
- FOR, 531
- GRANT, 392
- IF, 531
- LEAVE, 531
- LOAD DATABASE, 388
- LOAD TRANSACTION, 388
- logging, 36
- LOOP, 531, 545
- MESSAGE, 549
- OPEN, 545
- optimization, 314
- OUTPUT, 435
- RAISERROR, 413
- RETURN, 539
- REVOKE, 392
- ROLLBACK, 407, 558
- SELECT, 402, 404, 540
- SIGNAL, 413, 549
- WHILE, 531

## statements allowed in batches, 561

## statistics

- access plans, 364, 365
- adding to the Performance Monitor, 164
- available, 166
- column statistics, 315
- displaying, 166
- monitoring, 162
- performance, 165
- removing to the Performance Monitor, 164
- updating column statistics, 317

- statistics are present and correct, 319

- steps in optimization, 322

- stored procedure language
  - overview, 406

- stored procedures
  - debugging, 576
  - Transact-SQL stored procedure overview, 406
  - viewing individual profiling information, 177
  - viewing profiling data, 172
  - viewing summary profiling data, 175

- string and number defaults, 74

## strings

- matching, 199
- quotation marks, 201

- structure of procedures and triggers, 535

## subqueries

- about, 273, 274
- ALL test, 280
- ANY operator, 280
- ANY test, 279
- caching of, 363
- comparison operators, 293
- comparison test, 277, 278
- converting to joins, 287, 292
- correlated, 286, 291
- existence test, 277, 284
- GROUP BY, 276
- HAVING clause, 276
- IN keyword, 199
- nested, 289
- optimizer internals, 291
- outer references, 276
- quantified comparison test, 277, 279
- rewriting as EXISTS predicates, 320
- rewriting as joins, 287
- row group selection, 276
- row selection, 275
- set membership test, 277, 282
- types of operators, 277
- WHERE clause, 275, 292

- subqueries and joins, 287

- subqueries in the HAVING clause, 276

- subquery and function caching, 363

- subquery comparison test, 278

- subquery set membership test
  - about, 282
- subquery tests, 277
- subquery transformations during optimization, 349
- subquery unnesting, 353
- subscriptions
  - data replication and concurrency, 138
- substituting a value for NULL, 204
- subtransactions
  - procedures and triggers, 558
- sub-transactions
  - and savepoints, 93
- summarizing query results using aggregate functions, 208
- summarizing, grouping and sorting query results, 207
- summary profiling data
  - events, 176
  - stored procedures and functions, 175
  - triggers, 176
- summary values, 213
  - about, 208
- support
  - newsgroups, xvii
- swap space
  - database cache, 154
- Sybase Central
  - altering tables, 42
  - and column defaults, 71
  - column constraints, 77
  - copying tables, 49
  - creating databases, 30
  - creating indexes, 60
  - creating tables, 40
  - deleting databases, 32
  - deleting tables, 43
  - displaying system objects, 35
  - displaying system tables, 50
  - dropping views, 56
  - editing tables, 38
  - erasing databases, 32
  - installing jConnect metadata support, 37
  - logging SQL statements, 36
  - managing foreign keys, 47
  - managing primary keys, 45
  - modifying views, 55
  - rebuilding databases, 441
  - setting a consolidated database, 35
  - setting database options, 34
  - starting a database without connecting, 36
  - translating procedures, 409
  - validating indexes, 60
- Sybase Central Performance Monitor, 163
- symbols
  - string comparisons, 199
- syntax-independent optimization, 314
- SYSCOLSTAT
  - updating column statistics, 317
- SYSCOLUMNS view
  - conflicting name, 394
- SYSINDEX table
  - index information, 62
- SYSINDEXES view
  - conflicting name, 394
  - index information, 62
- SYSIXCOL table
  - index information, 62
- sys.servers system table
  - remote servers, 460
- SYSTABLE system table
  - view information, 57
- system administrator
  - Adaptive Server Enterprise, 389
- system catalog
  - Adaptive Server Enterprise compatibility, 389
- system functions
  - tsequal, 399
- system objects, 35
  - displaying, 50
- system security officer
  - Adaptive Server Enterprise, 389
- system tables
  - Adaptive Server Enterprise compatibility, 389
  - and indexes, 62
  - displaying, 50

- information about referential integrity, 88
- owner, 389
- Transact-SQL name conflicts, 394
- views, 57

- system triggers
  - implementing referential integrity actions, 86

- system views
  - and indexes, 62
  - information about referential integrity, 88

- SYSVIEWS view
  - view information, 57

## T

- table and page sizes, 338

- Table Editor
  - about, 38
  - Advanced Table Properties dialog, 39
  - setting the table type, 39
  - using, 40

- table expressions
  - how they are joined, 233
  - in key joins, 262
  - syntax, 230

- table fragmentation
  - about, 168

- table names
  - fully qualified in procedures, 559
  - identifying, 185
  - procedures and triggers, 559

- table scans
  - about, 325
  - disk allocation and performance, 337

- table size
  - about, 338

- table structures for import, 425

- tables
  - adding keys to, 45, 46, 47, 48
  - advanced table properties, 39
  - altering, 41, 42
  - bitmaps, 338
  - browsing data, 44
  - column names, 25

- constraints, 26
- copying, 49
- copying rows, 306
- correlation names, 194
- creating, 40
- creating proxy tables in SQL, 469, 470
- creating proxy tables in Sybase Central, 468
- defragmenting all tables in a database, 169
- defragmenting individual tables in a database, 170
- deleting, 43
- displaying primary keys in Sybase Central, 45
- displaying references from other tables, 47
- dropping, 43
- editing data, 44
- exporting, 438
- fragmentation, 168
- group reads, 338
- importing, 431
- listing remote, 463
- managing foreign keys, 47, 48
- managing primary keys, 44, 45, 46
- managing table constraints, 77
- naming in queries, 194
- properties, 25
- proxy, 467
- remote access, 458
- structure, 41
- system tables, 50
- Transact-SQL, 401
- work tables, 160
- working with, 38

- technical support
  - newsgroups, xvii

- temporary files
  - work tables, 147

- temporary tables
  - about, 63
  - importing data, 425, 426
  - indexes, 341
  - local and global, 63
  - Transact-SQL, 402
  - work tables in query processing, 160

- testing
  - database design, 16, 23

- testing a column for NULL, 203

- testing set membership with IN conditions, 282

- text plans
  - reading access plans, 370
- theorems
  - two-phase locking, 131
- threads
  - deadlock when none available, 101
- times
  - procedures and triggers, 560
- TIMESTAMP data type
  - Transact-SQL, 398
- tips
  - performance, 144
- tips for writing procedures, 559
- tools
  - rebuilding databases, 441
- TOP clause
  - about, 221
  - when not to use, 221
- top performance tips, 144
- TRACEBACK function, 549
- trailing blanks
  - comparisons, 196
  - creating databases, 394
  - Transact-SQL, 394
- transaction blocking
  - about, 100
- transaction log
  - performance, 144
  - role in data replication, 139
- transaction management
  - and remote data, 479
- transaction processing
  - effects of scheduling, 103
  - performance, 92
  - scheduling, 102
  - serializable scheduling, 102
  - transaction log based replication, 139
  - two-phase locking, 131
- transaction scheduling
  - effects of, 103
- transactions
  - about, 90
  - blocking, 100, 101, 113
  - completing, 91
  - concurrency, 92
  - data modification, 302
  - deadlock, 101
  - interference between, 100, 113
  - managing, 479
  - more than one at once, 92
  - procedures and triggers, 558
  - remote data access, 479
  - replicating concurrent, 138
  - savepoints, 93
  - starting, 91
  - sub-transactions and savepoints, 93
  - using, 91
- transactions and data modification, 302
- transactions and isolation levels, 89
- transactions and savepoints in procedures and triggers, 558
- transactions processing
  - blocking, 100, 113
- Transact-SQL
  - about, 384
  - batches, 407
  - creating databases, 393
  - IDENTITY column, 399
  - joins, 404
  - NULL, 402
  - outer join limitations, 246
  - outer joins, 245
  - overview, 384
  - procedures, 406
  - result sets, 410
  - timestamp column, 398
  - trailing blanks, 394
  - triggers, 406
  - variables, 411
  - writing compatible SQL statements, 401
  - writing portable SQL, 401
- Transact-SQL batch overview, 407
- Transact-SQL compatibility, 383
  - databases, 396
  - setting database options, 395
- Transact-SQL procedure language overview, 406

Transact-SQL stored procedure overview, 406

Transact-SQL trigger overview, 406

transformations

rewrite optimization, 352

trantest, 170

trigger creation wizard

using, 523

triggers

about, 507

altering, 526

benefits of, 510

command delimiter, 559

creating, 523

cursors, 545

dates, 560

deleting, 527

error handling, 548

exception handlers, 553

executing, 525

execution permissions, 527

overview, 509

recursion, 407

ROLLBACK statement, 407

savepoints, 558

SQL statements allowed in, 535

statement-level, 406

structure, 535

times, 560

Transact-SQL, 397, 406

using, 522

viewing individual profiling information, 177

viewing summary profiling data, 176

warnings, 552

troubleshooting

ANY operator, 280

deadlocks, 101

debugging classes, 579

performance, 144

remote data access, 485

TRUNCATE TABLE statement

about, 312

try using Adaptive Server Anywhere's compression

features, 151

tsequal function, 399

turn off autocommit mode, 149

tutorials

dirty reads, 106

implications of locking, 116

isolation levels, 106

non-repeatable reads, 109

phantom locks, 116

phantom rows, 113

two-phase locking, 131

two-phase locking theorem, 131

types of explicit join conditions, 237

types of index, 346

types of locks, 122

types of semantic transformations, 352

typical transactions at various isolation levels, 103

typical types of inconsistency, 94

## U

uncorrelated subqueries

about, 291

underlying assumptions

optimizer, 318

understanding complex outer joins, 244

understanding group by, 214

union all

query execution algorithms, 335

UNION operation

combining queries, 223

unions

query execution algorithms, 335

unique keys

generating and concurrency, 135

unique results

limiting, 193

uniqueness

enforcing with an index, 345

UNKNOWN

and NULL, 204

unknown values. about, 202

- unload database wizard
  - about, 442
  - using, 437
- UNLOAD statement
  - about, 433
- UNLOAD TABLE statement
  - about, 433
- unloading and reloading
  - databases, 444, 445, 447
- unloading databases
  - about, 440
  - using, 437
- unnecessary distinct elimination, 352
- unserializable transaction scheduling
  - effects of, 103
- UPDATE statement
  - locking during, 128
  - using, 308
- updating column statistics, 317
- upgrade database wizard
  - installing jConnect metadata support, 37
- upgrading databases, 442
- use and appropriate page size, 146
- use bulk operations methods, 150
- use fully-qualified names for tables in procedures, 559
- use indexes effectively, 146
- use of work tables in query processing, 160
- use the WITH EXPRESS CHECK option when validating tables, 150
- user IDs
  - Adaptive Server Enterprise, 390
  - case-sensitivity, 397
  - default, 72
- user-defined data types, 80
  - CHECK conditions, 77
  - creating, 80, 81
  - deleting, 82
- user-defined functions
  - about, 518
  - caching, 363
  - calling, 519
  - creating, 518
  - dropping, 520
  - execution permissions, 521
  - external functions, 562
  - parameters, 537
- users
  - occasionally connected, 138
- uses for locks, 123
- using aggregate functions with distinct, 211
- using CHECK conditions on columns, 76
- using CHECK conditions on tables, 78
- using column defaults, 70
- using comparison operators in the WHERE clause, 196
- using compound statements, 533
- using count (\*), 211
- using cursors in procedures and triggers, 545
- using cursors on SELECT statements in procedures, 545
- using domains, 80
- using exception handlers in procedures and triggers, 553
- using foreign keys to improve query performance, 157
- using full names when expanding objects and arrays
  - debugger [dbprdbg] utility, 594
- using group by with aggregate functions, 213
- using joins in delete, update, and insert statements, 234
- using keys to improve query performance, 157
- using lists in the WHERE clause, 198
- USING parameter value in the CREATE SERVER statement, 490
- using primary keys to improve query performance, 157

- using procedures, triggers, and batches, 507
- using ranges (between and not between) in the WHERE clause, 197
- using remote procedure calls, 476
- using SELECT statements in batches, 561
- using subqueries, 273
- using subqueries in the WHERE clause, 275
- using Sybase Central to translate stored procedures, 409
- using table and column constraints, 76
- using the cache to improve performance, 152
- using the EXECUTE IMMEDIATE statement in procedures, 557
- using the RAISERROR statement in procedures, 413
- using the Sybase Central Table Editor, 38
- using the WHERE clause for join conditions, 238
- using the WITH CHECK OPTION clause, 53
- using transaction and isolation levels, 89
- using transactions, 91
- using views, 53
- using views with Transact-SQL outer joins, 247
- UUIDs
  - default column value, 73
  - generating, 135

## V

- validating
  - indexes, 60
- validating tables
  - WITH EXPRESS CHECK, 150
- validation
  - column constraints, 26
- variables
  - assigning, 404
  - debugging, 578, 583
  - local, 404
  - SELECT statement, 404
  - SET statement, 404
  - Transact-SQL, 411
- vector aggregates, 213
- verifying
  - database design, 23
- verifying that procedure input arguments are passed correctly, 560
- VI
  - editing text with, 595
  - enabling VI keys in entry-text windows in the debugger [dbprdbg] utility, 595
- view creation wizard
  - using, 52
- viewing
  - table data, 44
  - view data, 57
- viewing procedure profiling data
  - Sybase Central, 175
- viewing procedure profiling information in Interactive SQL, 178
- viewing profiling information for a specific procedure in Interactive SQL, 179
- viewing the isolation level, 99
- views
  - browsing data, 57
  - check option, 53
  - copying, 53
  - creating, 51
  - deleting, 56
  - FROM clause, 194
  - in key joins, 268
  - in natural joins, 257
  - in outer joins, 245
  - modifying, 55
  - SYSCOLUMNS, 394
  - SYSINDEXES, 394
  - updating, 53
  - using, 53
  - working with, 51
- views in the system tables, 57

virtual memory  
a scarce resource, 320

## W

waiting  
to access locked rows, 113  
to verify referential integrity, 128

waiting to access locked rows  
deadlock, 100

warnings  
procedures and triggers, 552

Watcom-SQL  
about, 384  
writing compatible SQL statements, 401

when to create an index, 340

when to use indexes, 58

WHERE clause  
about, 195  
compared to HAVING, 218  
GROUP BY clause, 216  
joins, 238  
NULL values, 203  
performance, 159, 350  
string comparisons, 199  
subqueries, 275  
UPDATE statement, 309

where you can use aggregate functions, 209

WHILE statement  
syntax, 531

wildcards  
LIKE operator, 200  
string comparisons, 199

Windows CE  
creating databases, 30

Windows Performance Monitor  
about, 165

WITH CHECK OPTION clause  
about, 53

WITH EXPRESS CHECK  
performance, 150

wizards  
add foreign key, 47  
create database, 30  
data migration, 449  
domain creation, 80  
erase database, 32  
import, 428  
index creation, 60  
procedure creation, 511  
proxy table creation, 468  
remote procedure creation, 476, 512  
remote server creation, 461  
trigger creation, 523  
upgrade database, 37  
view creation, 52

work tables  
about, 160  
performance tips, 147  
query processing, 160

working with breakpoints, 589

working with column defaults in Sybase Central, 71

working with database objects, 27

working with databases, 29

working with external logs, 465

working with indexes, 58

working with proxy tables, 467

working with remote servers, 460

working with table and column constraints in Sybase Central, 77

working with tables, 38

working with views, 51

write locks, 122

writing compatible queries, 402

writing compatible SQL statements, 401

## X

XML  
format, 424





