

## CHAPTER 24

# Monitoring and Improving Performance

### About this chapter

This chapter describes some of the methods available to monitor and improve the performance of your database.

### Contents

Topic	Page
Top performance tips	624
Using keys to improve query performance	630
Using indexes to improve query performance	634
Search strategies for queries from more than one table	636
Sorting query results	639
Temporary tables used in query processing	640
How the optimizer works	641
Monitoring database performance	644

## Top performance tips

Adaptive Server Anywhere is designed to provide excellent performance and to do so automatically. However, some tips help you achieve the most from the product. The following suggestions are a good starting point.

### Always use a transaction log

You might think that Adaptive Server Anywhere would run faster without a transaction log because it has to maintain less information on disk, yet the opposite is the case. Not only does a transaction log provide a large amount of protection, it can dramatically improve performance.

Without a transaction log, Adaptive Server Anywhere must always be sure that any changes to your database are written to the disk at the end of every transaction. Writing these changes can consume considerable resources.

With a transaction log, Adaptive Server Anywhere need only write notes detailing the changes. It can choose to write the new database pages more efficiently, later.

The process of writing information to the database file to make it consistent and up to date is called a **checkpoint**. Without a transaction log, Adaptive Server Anywhere must perform a checkpoint at the end of every transaction.

#### 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, rather than the one that contains the main database file, you can further improve performance. With this arrangement, you will reduce contention for the hard drives.

### Increase the cache size

Adaptive Server Anywhere stores pages that it has used recently in a cache. The size of the cache is set on the command line when you launch the database server. Should another connect require the same page, it may find it already in memory and hence avoid reading information from disk.

If your cache is too small, Adaptive Server Anywhere is unable to keep pages in memory long enough to reap these benefits. When you launch the server, allocate as much memory to the database cache as is feasible, given the requirements of the other applications and processes that will run concurrently.

**Tip**

Increase the cache size. Increasing the cache size can often improve performance dramatically because retrieving information from memory is many times faster than reading it from disk. You may even find it worthwhile to purchase more RAM to allow a larger cache.

In particular, databases that make use of Java objects benefit greatly from larger cache sizes. If you are using Java in your database, consider using a cache of at least 8 Mb.

## **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 many copies of the same information.

For example, suppose the people in your company work at a number of offices. You might better place information about the office, 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 any one statement, Adaptive Server Anywhere must choose one index to access each table. When it cannot find a suitable index, it must instead resort to scanning the table sequentially—a process that can take a long time.

For example, if suppose you need to search for people, but may know only their first or only their last name. Create two indexes, one that contains the last names first, and a second that contains the first names first.

Examine the plans generated in response to your common statements. Considering adding an index when it will allow Adaptive Server Anywhere to access data more efficiently. In particular, add an index when it will eliminate unnecessary sequential access of a large table.

Although indexes let Adaptive Server Anywhere locate information very efficiently, you should exercise some caution when adding them. Each index creates extra work whenever you insert a row because Adaptive Server Anywhere must, in addition to adding the row, update all affected indexes. The same applies when you delete rows or update information in an indexed column.

If you need better performance when you add rows to a table and do not need to find information quickly, use as few indexes as possible.

## Use a larger page size

Large page sizes can help Adaptive Server Anywhere to read databases more efficiently when the database is large or when you access information sequentially. If either of these criteria apply, try using 4 or 2 kb pages instead of 1 kb pages.

Larger pages also bring other benefits. They improve the fan-out of your indexes and reduce the number of index levels. Large pages also let you create tables with more columns.

You cannot change the page size of a database. 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 4 kb pages.

```
dbinit -p 4096 new.db
```

If you use larger pages, you must specify a large page size when you start a database server. A database server cannot open a database that uses pages larger than the size you chose when you started it. You specify the maximum page size using the `-gp` flag. You should increase your cache size. A cache of the same size will accommodate only a fraction of the number of the larger pages, leaving less flexibility in arranging the space.

The following command starts a server that reserves an 8 Mb cache and can accommodate databases of page sizes up to 4096 bytes.

```
dbsrv6 -gp 4096 -c 8M -x tcpip -n myserver
```

In contrast, a small page size sometimes allows Adaptive Server Anywhere to run with less resources because it can store more pages in a cache of the same size. They are thus useful if your database must run on small machines with limited memory. Small pages can also help in situations when you use your database primarily to retrieve small pieces of information from random locations.

The benefits of smaller pages are not always realized. 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.

## Place different files on different devices

Disk drives operate much more slowly than modern processors or RAM. Much of what slows a database server is waiting for the disk to read or write pages.

You almost always improve database performance when you put different physical database file on different physical devices. For example, while one disk drive is busy writing out swapping database pages to and from the cache, another device can be writing to the log file.

Notice that to gain these benefits, the two or more devices involved 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 (.tmp)

The first is your **database file**. It holds the entire contents of your database. A single database is contained in a single file. You choose a location for it appropriate to your needs.

The second is the **transaction log file**. Effective recovery of the information in your database in the event of a failure depends most on the transaction log file. 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**

Locate 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 will run faster because it can efficiently write to the log and log mirror files simultaneously.

You can use the **dblog** transaction log utility to specify the location of the transaction log and transaction log mirror files.

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**

Direct Adaptive Server Anywhere to place its temporary file on a fast device, physically separate from that holding the database file. Adaptive Server Anywhere runs faster because many of the operations that necessitate using the temporary file also require retrieving a lot of information from the database.

Adaptive Server Anywhere examines the following environment variables, in the order shown, to determine a directory in which to place the temporary file.

- ◆ TMP
- ◆ TMPDIR
- ◆ 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 in performance.

If your machine has 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 data spaces, located on separate devices. In such a case, group tables in the separate data spaces so that common join operations will read information from different files.

Another similar strategy is to place the temporary and database files on a RAID device. 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 information about data recovery, see "Backup and Data Recovery" on page 553.

☞ For information about transaction log and transaction logs and the dbcc utility, see "Administration utilities overview" on page 65 of the book *Adaptive Server Anywhere Reference Manual*.

## Turn off autocommit mode

If your application is running 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.

Each application interface has its own way of setting autocommit behavior. For the Open Client, ODBC, and JDBC interfaces, Autocommit is the default behavior.

Instead of running in autocommit mode, you should consider grouping your commands so that 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, and a isolation level of one or greater is used, blocking and deadlock can result.

The cost of using autocommit mode is particularly high if you are also not using a transaction log file. Every statement forces a checkpoint—an operation that can involve writing numerous pages of information to disk.

☞ For more information about autocommit, see "Setting autocommit or manual commit mode" on page 218.

## Defragment your drives

Your hard disk is excessively fragmented. This becomes more important as your database increases in size. In particular, the Windows 3.x server cannot do direct (fast) reading and writing when the database file is very fragmented. There are several utilities available to defragment your hard disk. One of these should be run periodically. You could put the database on a disk partition by itself to eliminate fragmentation problems.

## Use bulk operations methods

If you are loading huge amounts of information into your database, you can benefit from the special tools provided for these tasks.

☞ See "Tuning bulk loading of data" on page 287 for methods to improve performance during bulk operations.

## Using keys to improve query performance

The **foreign key** and the **primary key** are used for validation purposes. However, these keys are also used to improve performance where possible.

### Example

The following example illustrates how keys are used to make commands execute faster.

```
SELECT *  
FROM employee  
WHERE emp_id = 390
```

The simplest way for the server to perform 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 the search can take a long time.

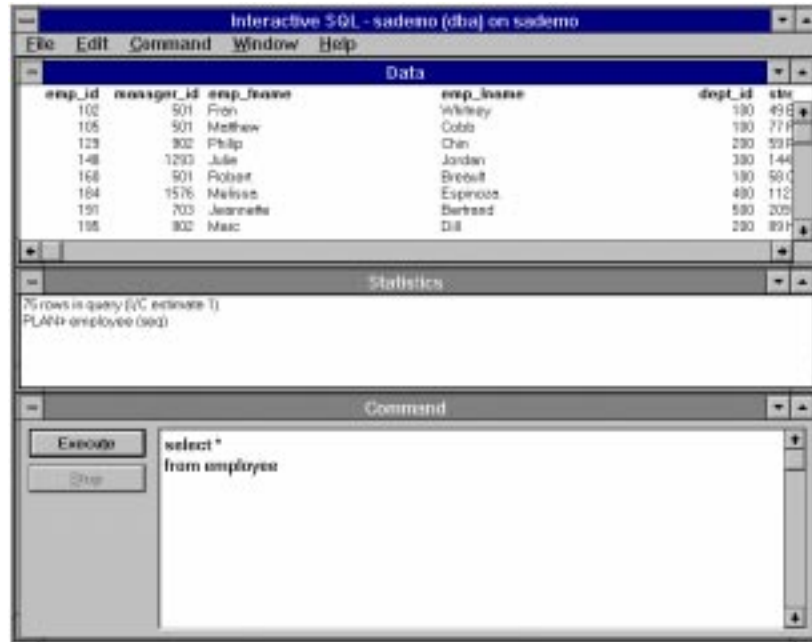
The **emp\_id** column is the **primary key** for the **employee** table. There is a built-in index mechanism for finding primary and foreign key values quickly. (This mechanism is used for the validation you saw in "Validity checking" on page 267 of the book *First Guide to SQL Anywhere Studio*.)

The same mechanism is used automatically to find the 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 table.

## Using Interactive SQL to examine query performance

The Interactive SQL Statistics window tells you when keys are being used to improve performance.





Information in the statistics window

If you execute a query to look at every row in the **employee** table:

```
SELECT *
FROM employee
```

two lines appear in the Statistics window:

```
75 rows in query (I/O estimate 14)
PLAN> employee (seq)
```

The first line indicates the number of rows in the query. Sometimes the database knows exactly, as in this case where there are 75 rows; other times it estimates the number of rows. The first line also indicates an internal I/O estimate of how many times the server will have to look at the database on your hard disk to examine the entire **employee** table.

The second line summarizes the execution plan for the query: the tables that are searched, any indexes used to search through a table. This plan says that the server will look at the **employee** table sequentially (that is, one page at a time, in the order that the rows appear on the pages). The letters **seq** inside parentheses mean that all the rows of the table need to be examined. This makes sense, since the query fetches the entire table.

### Resetting statistics

You may notice, when working through the tutorial yourself, that the statistics window contains estimates that are different from what is given here. This may happen because the optimizer has decided to optimize a query differently. The optimizer maintains statistics as it evaluates queries and uses these statistics to optimize subsequent queries. These statistics can be reset by executing the following statement:

```
DROP OPTIMIZER STATISTICS
```

Note that you must have DBA authority to execute this statement. In production environments, dropping the optimizer statistics can cause queries to execute slower, as the optimizer has less information about the actual distribution of data in the database tables.

## Using primary keys to improve query performance

A primary key is used to improve performance on the following statement:

```
SELECT *  
FROM employee  
WHERE emp_id = 390
```

### Statistic window information

The statistics window contains the following two lines:

```
Estimated 1 rows in query (I/O estimate 2)  
PLAN> employee (employee)
```

Whenever the name inside the parentheses in the Statistics window 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. Also, the Statistics window shows that the database optimizer estimates that there will be one row in the query and that it will have to go to the disk twice.

## Using foreign keys to improve query performance

The following query lists the orders from customer with customer ID 113:

```
SELECT *  
FROM sales_order  
WHERE cust_id = 113
```

### Statistic window information

The statistics window contains the following information:

```
Estimated 2 rows in query (I/O estimate 2)  
PLAN> sales_order (ky_so_customer)
```

Here **ky\_so\_customer** refers to the **foreign key** that the **sales\_order** table has for the **customer** table.

Primary and foreign keys are just special indexes that also maintain entity and referential integrity. The integrity is maintained by extra information that is placed in the indexes.

## Using indexes to improve query performance

Sometimes you need to search for something which is not in a primary or foreign key. In this case, a key cannot be used to improve performance. Creating **indexes** speeds up searches on particular columns. For example, suppose you wanted to look up all the employees with a last name beginning with M.

A query for this is as follows:

```
SELECT *
FROM employee
WHERE emp_lname LIKE 'M%'
```

If you execute this command, the plan description in the Interactive SQL Statistics window shows that the table is searched sequentially.

### Creating an index

If a search by employee last names is common, you may wish to create an **index** on the **emp\_lname** column in order to speed up the queries. You can do this with a CREATE INDEX statement.

```
CREATE INDEX lname
ON employee ( emp_lname )
```

The column name **emp\_lname** indicates the column that is indexed. An index can contain one, two, or more columns. However, if you create a multiple-column index, and then do a search with a condition using only the second column in the index, the index cannot be used to speed up the search.

An index is similar to a telephone book, which first sorts people by their last name, and then all the people with the same last name by their first name. A telephone book is useful if you know the last name, even more useful if you know both the first name and last name, but worthless if you only know the first name and not the last name.

Once you have created the index, rerunning the query produces the following plan description in the Statistics window:

```
PLAN> employee (lname)
```

### How indexes are used

Indexes are used automatically. Once an index is created, it is automatically kept up to date and used to improve performance whenever it can.

You could create an index for every column of every table in the database. But that would make data modifications slow, since all indexes affected by the change have to be updated. Further, each index requires space in the database. For these reasons, you should only create indexes that are used frequently.

Since you will not be using this index again, you should delete it by entering the following statement:

DROP INDEX lname

## How indexes work

This section provides a technical description of how the server uses indexes when searching databases.

### Index page structure

The Adaptive Server Anywhere query processor uses modified B+ trees. Each index page is a node in the tree and each node has many index entries. Leaf page index entries have a reference to a row of the indexed table. Indexes are kept balanced (uniform depth) and pages are kept close to full.

### An index lookup

An index lookup starts with the root page. The index entries on a nonleaf page determine which child page has the correct range of values. The index lookup moves down to the appropriate child page. This continues until a leaf page is reached. An index with N levels will require N reads for index pages and 1 read for the data page that contains the actual row. Index pages tend to be cached due to the frequency of use.

### Recommended page sizes

About the first 10 bytes of data for each index entry are stored in the index pages. This allows for a fan-out of roughly 200 using 4K pages, meaning that 200 rows can be indexed on one page, and 40,000 rows can be indexed 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. 4K pages are recommended for large databases.

The leaf nodes of the index are linked together. Once a row has been looked up, the rows of the table can be scanned in index order. Scanning all rows with a given value requires only one index lookup, followed by scanning the leaf nodes of the index until the value changes. This occurs when you have a WHERE clause that filters out rows with a certain value or a range of values. It also occurs when joining rows in a one-to-many relationship.

## Search strategies for queries from more than one table

This section uses sample queries to illustrate how the server selects an optimal processing route for each query. If you execute each of the commands in this section in Interactive SQL, the Statistics window display shows you the execution plan chosen to process each query.

### Using a key join

The following simple query uses a **key join** to search more than one table:

```
SELECT customer.company_name, sales_order.id
FROM sales_order
KEY JOIN customer
```

The Statistics window displays the following:

Estimated 711 rows in query (I/O estimate 2)

PLAN> customer(seq), sales\_order(ky\_so\_customer)

When this query is executed, the Interactive SQL Statistics window display indicates that Adaptive Server Anywhere first examines each row in the **customer** table, then finds the corresponding sales order numbers in the **sales\_order** table using the **ky\_so\_customer** foreign key joining the **sales\_order** and **customer** tables.

The order that the tables are listed in the Statistics window is the order that the tables are accessed by the database.

### Adding a WHERE clause

If you modify the query by adding a WHERE clause, as follows, the search is carried out in a different order:

```
SELECT customer.company_name, sales_order.id
FROM sales_order
KEY JOIN customer
WHERE sales_order.id = 2583
```

The Statistics windows displays the following plan:

PLAN> sales\_order(sales\_order), customer(customer)

Now, Adaptive Server Anywhere looks in the **sales\_order** table first, using the primary key index. Then, for each sales order numbered 2583 (there is only one), it looks up the **company\_name** in the **customer** table using the customer table primary key to identify the row. The primary key can be used here because the row in the **sales\_order** table is linked to the rows of the **customer** table by the customer id number, which is the primary key of the **customer** table.

The tables are examined in a different order depending on the query. The Adaptive Server Anywhere built-in query optimizer estimates the cost of different possible execution plans, and chooses the plan with the least estimated cost.

For some more complicated examples, try the following commands which each join four tables. The Interactive SQL statistics window shows that each query is processed in a different order.

#### Example 1

##### ❖ To list the customers and the sales reps they have dealt with.

###### ◆ Type the following:

```
SELECT customer.lname, employee.emp_lname
FROM customer
KEY JOIN sales_order
KEY JOIN sales_order_items
KEY JOIN employee
```

lname	emp_lname
Colburn	Chin
Smith	Chin
Sinnot	Chin
Piper	Chin
Phipps	Chin

The plan for this query is as follows:

```
PLAN> employee (seq), sales_order (ky_so_employee_id),
customer (customer), sales_order_items (id_fk)
```

#### Example 2

The following command restricts the results to list all sales reps that the customer named **Piper** has dealt with:

```
SELECT customer.lname, employee.emp_lname
FROM customer
KEY JOIN sales_order
KEY JOIN sales_order_items
KEY JOIN employee
WHERE customer.lname = 'Piper'
```

The plan for this query is as follows:

```
PLAN> customer (ix_cust_name), sales_order (ky_so_customer),
employee (employee), sales_order_items (id_fk)
```


#### Example 3

The third example shows all customers who have dealt with a sales representative who has the same name that they have:

```
SELECT customer.lname, employee.emp_lname
FROM customer
  KEY JOIN sales_order
  KEY JOIN sales_order_items
  KEY JOIN employee
WHERE customer.lname = employee.emp_lname
```

The plan for this query is as follows:

```
PLAN> employee (seq), customer (ix_cust_name),
       sales_order (ky_so_employee_id), sales_order_items (id_fk)
```

 For information on how the optimizer selects a strategy for each search, see "How the optimizer works" on page 641.



## Sorting query results

Many queries against a database have an `ORDER BY` clause so that the rows come out in a predictable order. Indexes are used to accomplish the ordering 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 are scanned and quickly sorted. If most of the **id** values are greater than 300, the first strategy is much better because no sorting is necessary.

### Solving the problem

The example above could be solved by creating a two-column index on **id** and **company\_name**. (The order of the two columns is important.) The server could then use this index to select rows from the table and have them in the correct order. However, keep in mind that indexes take up space in the database file and involve some overhead to keep up to date. Do not create indexes indiscriminately.


## Temporary tables used in query processing

When temporary tables occur	<p>Sometimes Adaptive Server Anywhere needs to make a <b>temporary table</b> for a query. This occurs in the following cases:</p> <ul style="list-style-type: none"><li>◆ When a query has an ORDER BY or a GROUP BY clause and Adaptive Server Anywhere does not use an index for sorting the rows, no suitable index exists.</li><li>◆ When a multiple-row UPDATE is being performed and the column being updated is used in the WHERE clause of the update or in an index that is being used for the update.</li><li>◆ When a multiple-row UPDATE or DELETE has a subquery in the WHERE clause that references the table that is being modified.</li><li>◆ When an INSERT from a SELECT statement is being performed and the SELECT statement references the insert table.</li><li>◆ When a multiple row INSERT, UPDATE, or DELETE is performed, and there are triggers defined on the table that the operation causes to fire.</li></ul>
	<p>In these cases, Adaptive Server Anywhere makes a temporary table before the operation begins. The records affected by the operation are put into the temporary table and a temporary index is built on the temporary table. This operation of extracting the required records into a temporary table can take a significant amount of time before any rows at all are retrieved from the query. Thus, creating indexes that can be used to do the sorting in first case, above, will improve the performance of these queries since it will not be necessary to build a temporary table.</p>
Notes	<p>The INSERT, UPDATE and DELETE cases above are usually not a performance problem since they are usually one-time operation. However, if problems occur, the only thing that can be done to avoid building a temporary table is to rephrase the command to avoid the conflict. This is not always possible.</p> <p>In Interactive SQL, the Statistics window displays "TEMPORARY TABLE" before the optimization strategy is listed if a temporary table is created by Adaptive Server Anywhere in carrying out the search.</p>

## How the optimizer works

Adaptive Server Anywhere has an optimizer that attempts to pick the best strategy for executing each query. The best strategy is the one that gets the results in the shortest period of time. The optimizer determines the **cost** of each strategy by estimating the number of disk reads and writes required. The strategy with the lowest cost is chosen.

The optimizer must decide which order to access the tables in a query, and whether or not to use an index for each table. If a query joins *N* tables, there are *N* factorial possible ways to access the tables. The optimizer will estimate the cost of executing the query in the different ways and use the ordering with the lowest cost estimate. The query execution plan in the Interactive SQL statistics window shows the table ordering for the current query and indicates in parentheses the index that was used for each table.

 This section provides an introduction to the optimizer. For more information, see "Query Optimization" on page 653.

### Optimizer estimates

The optimizer uses heuristics (educated guesses) to help decide the best strategy.

For each table in a potential execution plan, the optimizer must estimate the number of rows that will be part of the results. The number of rows will depend 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, a default estimate for equality is only used in cases where no better statistics are available.

The optimizer makes use of **indexes** and **keys** to improve its guess of the number of rows. Here are a few single-column examples:

#### 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: use the index to estimate the percentage of rows that will 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 rows for each primary row.

## Self tuning of the query optimizer

One of the most common constraints in a query is equality with a column value. For example,

```
SELECT *
FROM employee
WHERE sex = 'f'
```

tests for equality of the **sex** column. For this type of constraint, the Adaptive Server Anywhere optimizer learns from experience. A query will not always be optimized the same way the second time it is executed. The estimate for an equality constraint will be modified for columns that have an unusual distribution of values. This information is stored permanently in the database. If needed, the statistics can be deleted with the **DROP OPTIMIZER STATISTICS** command.

## Providing estimates to improve query performance

Since the query optimizer is guessing at the number of rows in a result based on the size of tables and particular restrictions used in the **WHERE** clause, it almost always makes inexact guesses. In many cases, the guess that the query optimizer makes is close enough to the real number of rows that the optimizer will have chosen the best search strategy. However, in some cases this does not occur.

The following query displays a list of order items that shipped later than the end of June, 1994:

```
SELECT ship_date
FROM sales_order_items
WHERE ship_date > '1994/06/30'
ORDER BY ship_date DESC
```

The estimated number of rows is 274. However, the actual number of rows returned is only 12. This estimate is wrong because the query optimizer guesses that a test for greater than will succeed 25 percent of the time. In this example, the condition on the **ship\_date** column:

```
ship_date > '1994/06/30'
```

is assumed to choose 25 percent of rows in the **sales\_order\_items** table.

### Supplying an estimate

If you know that a condition has a success rate that differs from the optimizer rule, you can tell the database this information by using an estimate. An estimate is formed by enclosing in brackets the expression followed by a comma and a number. The number represents the percentage of rows that the expression is estimated to select. In this case, you could estimate a success rate of one percent:

```
SELECT ship_date
FROM sales_order_items
WHERE ( ship_date > '1994/06/30', 1 )
ORDER BY ship_date DESC
```

With this estimate, the optimizer estimates ten rows in the query.

**Note**

Incorrect estimates are only a problem if they lead to poorly optimized queries.

☞ For further information about the optimizer and query optimization, see "Query Optimization" on page 653.

## Monitoring database performance


Adaptive Server Anywhere provides a set of statistics that can be used to monitor database performance. These are accessible from Sybase Central, and client applications can access the statistics as functions. In addition, these statistics are made available by the server to the Windows NT performance monitor.

This section describes how to access performance and related statistics from client applications, how to monitor database performance using Sybase Central, and how to monitor database performance using the Windows NT performance monitor.


### 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 engine-wide basis. The kind of information available ranges from static information such as the server name to detailed performance-related statistics concerning disk and memory usage.

The performance-related statistics are also available, along with some other statistics, for the Windows NT engine and server in the Windows NT Performance Monitor.

 For more information on the Performance Monitor, see "Monitoring database statistics from the Windows NT Performance Monitor" on page 646.

This section illustrates how to use the functions.

 A complete list of system functions and of their properties is provided in the section "System functions" on page 276 of the book *Adaptive Server Anywhere Reference Manual*.

Functions that  
retrieve system  
information

The following functions are used to retrieve system information:

- ◆ **property** Provides the value of a given property on an engine-wide basis
- ◆ **connection\_property** Provides the value of a given property for a given connection, or for the current connection by default.
- ◆ **db\_property** Provides the value of a given property for a given database, or for the current database by default.

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.

**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 maximum efficiency, 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.

Many of these statistics are made available in graphical form from the Sybase Central database management tool.

## Monitoring database statistics from Sybase Central

You can monitor database statistics from Sybase Central. The Sybase Central Performance Monitor is a graphing tool that can present database statistics as a line graph or a bar graph.

### ❖ To start the Sybase Central Performance Monitor:

- 1 Click the icon for the server you wish to monitor in the left panel.
- 2 Double-click the Statistics folder underneath the server.
- 3 Select a statistic to graph, and drag it to the Performance Monitor icon to start graphing that statistic.

About the  
performance  
monitor

The Performance Monitor uses the regular Adaptive Server Anywhere communication mechanisms to gather statistics. This means some statistics (most notably Cache Reads) are affected by Sybase Central. For example, graphing Cache Reads/sec in Sybase Central shows a steady rate, even when nothing apart from the monitoring is going on.

If you have a Windows NT client and server, the Windows NT Performance monitor is preferable since it offers more statistics, and is not intrusive: updating the statistics will not affect the measurements. The extra statistics the Windows NT performance monitor offers deal mainly with network communications—packets received, network buffers used, and so on.


## Monitoring database statistics from the Windows NT Performance Monitor

The Windows NT performance monitor is an application for viewing the behavior of objects such as processors, memory, and applications. Adaptive Server Anywhere provides many statistics for the performance monitor to display.

The Windows NT performance monitor allows unintrusive monitoring of statistics: updating the statistics does not affect the measurements.

❖ **To start the Windows NT performance monitor:**

- 1 Open the Administrative Tools program group.
- 2 Double click Performance Monitor.

 For information about Performance Monitor, see the Performance Monitor online Help.

❖ **To display database statistics:**

- 1 With Performance Monitor running, select Add To Chart from the Edit menu, or click the Plus sign on the toolbar.

The Add To Chart dialog appears.

- 2 From the Object list, select Adaptive Server Anywhere.

The Counter list then displays a list of the statistics provided.

- 3 From the Counter list, click a statistic to be displayed.
- 4 For a description of the selected counter, click Explain.
- 5 To display the counter, click Add.



- 6 When you have selected all the counters you wish to display, click Done.

#### Performance Monitor statistics

The statistics made available for Performance Monitor by Adaptive Server Anywhere are as follows:

Statistic	Description
Active Requests	Active Requests is the number of engine threads that are currently handling a request.
Asynchronous Reads/sec	Asynchronous Reads/sec is the rate at which pages are being read asynchronously from disk.
Asynchronous Writes/sec	Asynchronous Writes/sec is the rate at which pages are being written asynchronously to disk.
Bytes Received/sec	Bytes Received/sec is the rate at which network data (in bytes) are being received.
Bytes Transmitted/sec	Bytes Transmitted/sec is the rate at which bytes are being transmitted over the network.
Cache Hits/sec	Cache Hits/sec is the rate at which database page lookups are satisfied by finding the page in the cache.
Cache Index Internal Reads/sec	Cache Index Internal Reads/sec is the rate at which index internal-node pages are being read from the cache.
Cache Index Leaf Reads/sec	Cache Index Leaf Reads/sec is the rate at which index leaf pages are being read from the cache.
Cache Reads/sec	Cache Reads/sec is the rate at which database pages are being looked up in the cache.
Cache Table Reads/sec	Cache Table Reads/sec is the rate at which table pages are being read from the cache.
Cache Writes/sec	Cache Writes/sec is the rate at which pages in the cache are being modified (in pages/sec).
Checkpoint Flushes/sec	Checkpoint Flushes/sec is the rate at which ranges of adjacent pages are being written out during a checkpoint.
Checkpoint Log/sec	Checkpoint Log/sec is the rate at which the transaction log is being checkpointed.
Checkpoint Urgency	Checkpoint Urgency is expressed as a percentage.
Checkpoints/sec	Checkpoints/sec is the rate at which checkpoints are being performed.
Commit files/sec	Commit files/sec is the rate at which the engine is forcing a flush of the disk cache. On Windows NT and NetWare platforms, the disk cache does not need to be flushed because unbuffered (direct) I/O is used.

Statistic	Description
Commits/sec	Commits/sec is the rate at which Commit requests are being handled.
Context Switch Checks/sec	Context Switch Checks/sec is the rate at which the current engine thread is volunteering to give up the CPU to another engine thread.
Context Switches/sec	Context Switches/sec is the rate at which the current engine thread is being changed.
Continue Requests/sec	Continue Requests/sec is the rate at which "CONTINUE" requests are being issued to the engine.
Corrupt Packets/sec	Corrupt Packets/sec is the rate at which corrupt network packets are being received.
Current IO	Current IO is the current number of file I/Os issued by the engine which have not yet completed.
Current Reads	Current Reads is the current number of file reads issued by the engine which have not yet completed.
Current Writes	Current Writes is the current number of file writes issued by the engine which have not yet completed.
Cursor	Cursor is the number of declared cursors that are currently being maintained by the engine.
Dirty Pages	Dirty Pages is the number of pages in the cache which must be written out and which do not belong to temporary files.
Disk Index Internal Reads/sec	Disk Index Internal Reads/sec is the rate at which index internal-node pages are being read from disk.
Disk Index Leaf Reads/sec	Disk Index Leaf Reads/sec is the rate at which index leaf pages are being read from disk.
Disk Reads/sec	Disk Reads/sec is the rate at which pages are being read from file.
Disk SyncReads/sec	Disk SyncReads/sec is the rate at which pages are being read synchronously from disk.
Disk SyncWrite Other/sec	Disk SyncWrite Other/sec is the rate at which pages are being written synchronously to disk for a reason not covered by other "Disk SyncWrites ____/sec" counters.
Disk SyncWrites Checkpoint/sec	Disk SyncWrites Checkpoint/sec is the rate at which pages are being written synchronously to disk for a checkpoint.
Disk SyncWrites Extend/sec	Disk SyncWrites Extend/sec is the rate at which pages are being written synchronously to disk while extending a database file.
Disk SyncWrites Free Current/sec	Disk SyncWrites Free Current/sec is the rate at which pages are being written synchronously to disk to free a page that

Statistic	Description
	cannot remain in the in-memory free list.
Disk SyncWrites Free Push/sec	Disk SyncWrites Free Push/sec is the rate at which pages are being written synchronously to disk to free a page that can remain in the in-memory free list.
Disk SyncWrites Log/sec	Disk SyncWrites Log/sec is the rate at which pages are being written synchronously to the transaction log.
Disk SyncWrites Rollback/sec	Disk SyncWrites Rollback/sec is the rate at which pages are being written synchronously to the rollback log.
Disk SyncWrites/sec	Disk SyncWrites/sec is the rate at which pages are being written synchronously to disk. It is the sum of all the other "Disk SyncWrites ____/sec" counters.
Disk Table Reads/sec	Disk Table Reads/sec is the rate at which table pages are being read from disk.
Disk Waitreads/sec	Disk Waitreads/sec is the rate at which the engine is waiting synchronously for the completion of a read IO operation which was originally issued as an asynchronous read. Waitreads often occur due to cache misses on systems that support asynchronous IO.
Disk Waitwrites/sec	Disk Waitwrites/sec is the rate at which the engine is waiting synchronously for the completion of a write IO operation which was originally issued as an asynchronous write.
Disk Writes/sec	Disk Writes/sec is the rate at which modified pages are being written to disk.
Dropped Packets/sec	Dropped Packets/sec is the rate at which network packets are being dropped due to lack of buffer space.
Extend Database/sec	Extend Database/sec is the rate (in pages/sec) at which the database file is being extended.
Extend Temporary File/sec	Extend Temporary File/sec is the rate (in pages/sec) at which temporary files are being extended.
Free Buffers	Number of free network buffers.
Freelist Write Current/sec	Freelist Write Current/sec is the rate at which pages that cannot remain in the in-memory free list are being freed.
Freelist Write Push/sec	Freelist Write Push/sec is the rate at which pages that can remain in the in-memory free list are being freed.
Full compares/sec	Full compares/sec is the rate at which comparisons beyond the hash value in an index must be performed.
IO to Recover	IO to Recover is the estimated number of IO operations required to recover the database.
Idle Active/sec	Idle Active/sec is the rate at which the engine's idle thread

Statistic	Description
	becomes active to do idle writes, idle checkpoints, etc.
Idle Checkpoints/sec	Idle Checkpoints/sec is the rate at which checkpoints are completed by the engine's idle thread. An idle checkpoint occurs whenever the idle thread writes out the last dirty page in the cache.
Idle Waits/sec	Idle Waits/sec is the number of times per second that the server goes idle waiting for IO completion or a new request.
Idle Writes/sec	Idle Writes/sec is the rate at which disk writes are being issued by the engine's idle thread.
Index Fills	Index Fills is the number of times a new temporary merge index is created.
Index Merges	Index Merges is the number of times a temp index has been merged into a main index
Index adds/sec	Index adds/sec is the rate at which entries are being added to indexes.
Index lookups/sec	Index lookups/sec is the rate at which entries are being looked up in indexes.
Lock Table Pages	Lock Table Pages is the number of pages used to store lock information.
Main Heap Pages	Main Heap Pages is the number of pages used for global engine data structures.
Map Pages	Map Pages is the number of map pages used for accessing the lock table, frequency table, and table layout.
Maximum IO	Maximum IO is the maximum value that "Current IO" has reached.
Maximum Reads	Maximum Reads is the maximum value that "Current Reads" has reached.
Maximum Writes	Maximum Writes is the maximum value that "Current Writes" has reached.
Multi-packets Received/sec	Multi-packets Received/sec is the rate at which multi-packet deliveries are being received.
Multi-packets Transmitted/sec	Multi-packets Transmitted/sec is the rate at which multi-packet deliveries are being transmitted.
Open cursors	Open cursors is the number of open cursors that are currently being maintained by the engine.
Packets Received/sec	Packets Received/sec is the rate at which network packets are being received.
Packets Transmitted/sec	Packets Transmitted/sec is the rate at which network packets are being transmitted.

Statistic	Description
Page Relocations/sec	Page Relocations/sec is the rate at which relocatable heap pages are being read from the temporary file.
Pending requests/sec	Pending requests/sec is the rate at which the engine is detecting the arrival of new requests.
Ping1/sec	Ping1/sec is the rate at which ping requests which go all the way down into the engine are serviced.
Ping2/sec	Ping2/sec is the rate at which ping requests which are turned around at the top of the protocol stack are serviced.
Procedure Pages	Procedure Pages is the number of relocatable heap pages used for procedures.
Read Hints Used/sec	Read Hints Used/sec is the rate at which page-read operations are being satisfied immediately from cache thanks to an earlier read hint.
Read Hints/sec	A read hint is an asynchronous read operation for a page that the server is likely to need soon. Read Hints/sec is the rate at which such read operations are being issued.
Recovery Urgency	Recovery Urgency is expressed as a percentage.
Redo Free Commits/sec	A "Redo Free Commit" occurs when a commit of the transaction (redo) log is requested but the log has already been written (so the commit was done for "free").
Redo Rewrites/sec	Redo Rewrites/sec is the rate at which pages that were previously written to the transaction log (but were not full) are being written to the transaction log again (but with more data added).
Redo Writes/sec	Redo Writes/sec is the rate at which pages are being written to the transaction (redo) log.
Relocatable Heap Pages	Relocatable Heap Pages is the number of pages used for relocatable heaps (cursors, statements, procedures, triggers, views, etc.).
Remoteput Wait/sec	Remoteput Wait/sec is the rate at which the communication link must wait because it does not have buffers available to send information. This statistic is collected for NetBIOS (both sessions and datagrams) and IPX protocols only.
Requests/sec	Requests/sec is the rate at which the engine is being entered to allow it to handle a new request or continue processing an existing request.
Rereads Queued/sec	A reread occurs when a read request for a page is received by the database IO subsystem while an asynchronous read IO operation has been posted to the operating system but has not completed. Rereads Queued/sec is the rate at which this condition is occurring.

Statistic	Description
Rereceived Packets/sec	Rereceived Packets/sec is the rate at which duplicate network packets are being received.
Retransmitted Packets/sec	Retransmitted Packets/sec is the rate at which network packets are being retransmitted.
Rollback Log Pages	Rollback Log Pages is the number of pages in the rollback log.
Rollback/sec	Rollback/sec is the rate at which Rollback requests are being handled.
Adaptive Server Anywhere	The Adaptive Server Anywhere object provides information about the database server.
Sends Failed/sec	Sends Failed/sec is the rate at which the underlying protocol(s) failed to send a packet.
Statement	Statement is the number of prepared statements that are currently being maintained by the engine.
TotalBuffers	TotalBuffers number of network buffers.
Trigger Pages	Trigger Pages is the number of relocatable heap pages used for triggers.
Unscheduled requests	Unscheduled requests is the number of requests that are currently queued up waiting for an available engine thread.
View Pages	View Pages is the number of relocatable heap pages used for views.
Voluntary blocks/sec	Voluntary blocks/sec is the rate at which engine threads voluntarily block on pending disk IO.
Waitread Full Compare/sec	Waitread Full Compare/sec is the rate at which read requests associated with a full comparison (a comparison beyond the hash value in an index) must be satisfied by a synchronous read operation.
Waitread Optimizer/sec	Waitread Optimizer/sec is the rate at which read requests posted by the optimizer must be satisfied by a synchronous read operation.
Waitread Other/sec	Waitread Other/sec is the rate at which read requests from other sources must be satisfied by a synchronous read operation.
Waitread SysConnection/sec	Waitread SysConnection/sec is the rate at which read requests posted from the system connection must be satisfied by a synchronous read operation. The system connection is a special connection that is used as the context before a connection is made and for operations performed outside of a client connection.
Waitread	Waitread Temporary Table/sec is the rate at which read

<b>Statistic</b>	<b>Description</b>
Temporary Table/sec	requests for a temporary table must be satisfied by a synchronous read operation.