CHAPTER 25

Query Optimization

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 that the optimizer must go through to optimize a query. It begins with the assumptions that underlie the design of the optimizer, then proceeds to discuss selectivity estimation, cost estimation, and the other steps of optimization.

Update, insert, and delete statements must also be optimized, but the focus of this chapter is on queries. The optimization of these other commands follows similar principles.

Contents

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, and the order in which the tables are to be read.

Often, a great number of plans exist that all accomplish the same goal. Other variables may further enlarge the number of possible access plans.

A single statement can contain multiple subqueries. A portion of an access plan that describes how to satisfy a single subquery, including table permutation and access methods, is called a **join strategy**. When a subquery refers to many tables, the number of possible join strategies can become very large. For example, if seven tables must be joined to execute a subquery, then the optimizer must select one of the 7! = 5040 orders in which these tables could be accessed. It must also decide which index, if any, to use when accessing each table.

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 that each will require. 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 Anywhere optimizer devises a suitable access plan based only on the semantics of each statement.

Syntactic differences, although they may appear substantial, usually have no effect. For example, differences in the order of predicates, tables, and attributes in the query syntax have no affect 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 is conscious of the resources it is using. It periodically 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.

The governor limits the optimizer's work

The governor is the part of the optimizer that performs this limiting function. It lets the optimizer run until it has analyzed a minimum number of strategies. Once a reasonable number of strategies have been considered, the governor cuts off further analysis.

In the case of expensive and complicated queries, the optimizer will work longer. In the case of very expensive queries, it may run long enough to cause a discernable delay.

Steps in optimization

The steps that the Anywhere optimizer follows in generating a suitable access plan are as follows.

1. The parser converts the query, expressed in SQL, into an internal representation. In doing so, it may rewrite the query, converting it to a syntactically different, but semantically equivalent, form. These conversions make the statement easier to analyze.
2. Optimization proper commences at OPEN CURSOR. Unlike many other commercial database systems, Anywhere optimizes each statement at the time it executes it.
3. Perform semantic optimization on the statement. The command is rewritten whenever doing so will lead to better, more efficient, access plans.
4. Perform join enumeration and group-by optimization for each subquery.
5. Optimize access order.

Because Anywhere performs just-in time optimization of each statement, the optimizer has access to the value of host variables and stored procedure variables. Hence, it can make better choices because it can perform better selectivity analysis.

Should you execute the same query repeatedly, it is optimized again each time. Because Anywhere saves statistics each time it executes a query, the optimizer also is afforded the opportunity of learning from the experience of executing previous plans and can adjust its choices when appropriate.

Reading access plans

The optimizer can tell you the plan it has chosen in response to any statement. If you are using Interactive SQL, you can simply look to the Statistics window. Otherwise, you can use the PLAN function to ask Anywhere to return a plan

The optimizer can rewrite your query
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.

Commas separate tables within a join strategy

Join strategies in plans appear as a list of correlation names. Each correlation name is followed immediately, in brackets, by the method to be used to locate the required rows. This method is either the word **seq**, which indicates that the table is to be scanned sequentially, or it is the name of an index. The name of a primary index is the name of the table.

The following self-join creates a list of employees and their managers.

```
SELECT e.emp_fname, m.emp_fname
FROM employee AS e JOIN employee AS m
   ON e.manager_id = m.emp_id
```

```
PLAN>  e (seq), m (employee)
```

To compute this result, Adaptive Server Anywhere will first access the employee table sequentially. For each row, it will access the employee table again, but this time using the primary index.

Temporary tables

To execute some query results, Adaptive Server Anywhere must use a temporary table, or may choose to use one to lower the overall cost of computing the result. When a temporary table will be used for a join strategy, the words TEMPORARY TABLE precede the description of that strategy.

```
SELECT DISTINCT quantity
FROM sales_order_items
```

```
PLAN>  TEMPORARY TABLE sales_order_items (seq)
```

A temporary table is necessary in this case to compute the distinct quantities.

Colons separate join strategies

The following command contains two **query blocks**: the outer select statement from the sales_order_items table, 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 )
```

PLAN> o (seq), i (id_fk): p (product)

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 will be executed.

In this case, the optimizer has decided to first access **o**, the **sales_order** table, sequentially and join it to **i**, the **sales_order_items** table, using the foreign key index (contained in the primary index of the product table). At some point, rows from **p**, the **product** table, will be located using the primary index.

The optimizer can rewrite your query

When the optimizer discovers a more efficient means of computing your result, the access plan may not appear to follow the structure of your query. Adding a condition to the subquery in the previous command causes the optimizer to choose a different strategy.

```
SELECT *
FROM sales_order AS o
   KEY JOIN sales_order_items AS i
WHERE EXISTS
    (   SELECT *
        FROM product p
        WHERE p.id = 300
           AND p.id = i.prod_id )
```

PLAN> p (product), i (ky_prod_id), o (sales_order)

The optimizer rewrites this command as a single query block that consists of single join between three tables.

For more information about the rules Adaptive Server Anywhere obeys when rewriting your query, see "Rewriting sub-queries as exists predicates" on page 661 and "Semantic query transformations" on page 671.

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.

Assumptions

The list below summarizes the assumptions upon which the Adaptive Server Enterprise optimizer is based.

Assumption	Implications
Minimal administration work	♦ Self-tuning design that requires fewer performance controls. ♦ No separate statistics-gathering utility
Applications tend to retrieve only the first few rows of a cursor	♦ Indices are used whenever possible ♦ Use of temporary tables is discouraged
Selectivity statistics necessary for optimization are available in the Column Statistics Registry	♦ Optimization decisions are based on prior query execution. ♦ Dropping optimizer statistics makes the optimizer ineffective.
An index can be found to satisfy a join predicate in virtually all cases	♦ Performance is poor if a suitable index cannot be found.
Virtual memory is a scarce resource	♦ Intermediate results are not materialized unless absolutely necessary.

Minimal administration work

Traditionally, high-performance database engines 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.

Anywhere learns and adjusts as the database grows and changes. Each query betters its knowledge of the data distribution in the database. 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 Anywhere has gained through executing another user's queries.

Statistics gathering mechanisms are thus an integral part of the database server. Because of this design, no external mechanism is required. 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.

Only first few rows of a cursor used frequently

Many application programs examine only the first few rows of a cursor, particularly when the cursor is ordered. Select the ordering carefully for best results.

To accommodate this observation, the optimizer avoids materializing cursors whenever possible. Since few rows of the cursor are likely to be fetched, this strategy allows Adaptive Server Anywhere to reduce the time required to pass the first row of the result to the application.

Statistics are present and correct

The optimizer is self-tuning. All the needed information is stored internally. The column statistics registry 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 this registry. In consequence, all subsequent queries gain access to more accurate estimates.

The optimizer relies heavily on these statistics and, because it does so, the quality of the access plans it generates depends heavily on them. If you have recently reloaded your database or inserted a lot of new rows, these statistics may no longer accurately describe the data. You may find that your first subsequent queries execute unusually slowly.

You can assist Anywhere in its efforts to correct its statistical information by executing sample queries. As Anywhere executes these statements, it will learn from its experience. Correct statistical information can dramatically improve the efficiency of subsequent queries.

An index can usually be found to satisfy a predicate

Often, Anywhere can evaluate predicates with the aid of an index. By using an index, the optimizer can speed access to data and reduce the amount of information read. Whenever possible, Anywhere uses indices to satisfy ORDER BY, GROUP BY, and DISTINCT clauses.

When the optimizer cannot find a suitable index, it must resort to a table scan, which can be expensive. An index can improve performance dramatically when joining tables. You should add indices to tables or rewrite queries wherever doing so will facilitate the efficient processing of common requests.

Virtual Memory is a scarce resource

The operating system and a number of applications frequently vie for the memory of a typical computer. Adaptive Server Anywhere treats memory as a scarce resource. Because it uses memory economically, 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 temporary tables.

On the other hand, it is careful to use memory where it will improve performance. For example, caches the results of subqueries when they will be needed repeatedly during the processing of the query.

Rewriting sub-queries as exists predicates

The assumptions which underlie the design of Anywhere require that it conserve memory and that it return the first few results of a cursor quickly as possible. In keeping with these objectives, Adaptive Server Anywhere rewrites all set-operation sub-queries, such as IN, ANY, or SOME predicates, as EXISTS predicates. By doing so, Anywhere avoids creating unnecessary temporary tables and may more easily identify a suitable index through which to access a table.

Non-correlated sub-queries are sub-queries that contain no explicit reference to the table or tables contained in the rest higher-level portions of the tables.

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, 1998.

Non-correlated subquery

```
SELECT *
FROM customer c
WHERE c.id NOT IN
   (  SELECT o.cust_id
      FROM sales_order o
      WHERE o.order_date = '1998-01-01' )
```

PLAN> c (seq): o (ky_so_customer)

One possible access plan is to first read the **sales_order** table and create a temporary table of all the customers who placed orders on January 1, 1998, then, read the **customer** table and extract one row for each customer listed in the temporary table.

However, Adaptive Server Anywhere avoids materializing results. 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 = '1993-01-01'
         AND (  o.cust_id = c.id
            OR  o.cust_id IS NULL
            OR  c.id IS NULL ) )
```

PLAN> c (seq): o (seq)

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.

Anywhere can cache the results of this subquery during processing. This strategy lets Anywhere reuse previously computed results. In the case of query above, caching will not help because customer identification numbers are unique in the customer table.

Further information on subquery caching is located in "Subquery caching" on page 684.

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.

Memory allocation for inserted rows

Anywhere inserts each new row into pages so that, if at all possible, the entire row can be stored contiguously

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, Anywhere will write 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 Anywhere will place the row on a new page at the end of the table.

Anywhere may store rows in any order

The engine locates space on pages and inserts rows in the order that it receives them. It assigns each to a page, but the locations it chooses in the table may not correspond to the order they were inserted. For example, the engine may have to start a new page in order to store a long row contiguously. Should the next row be short, 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 will frequently require the rows of table in a particular order, consider creating an index on those columns. Anywhere always tries to take advantage of indices when processing queries.

Space is not reserved for NULL columns

Whenever Anywhere inserts a row, it reserves only the space necessary to show the row with the values it contains at the time of creation. It reserves no space to store values which are NULL. It reserves no extra space to accommodate fields, such as text strings, which may enlarge.

Once inserted rows are immutable

Once assigned a home position on a page, a row is never moved. If an update changes any of the values in the row so that it will no longer fit in its assigned location, then the row is split and the extra information is inserted on another page.

This characteristic deserves special attention, especially since Anywhere allots no extra space at the time the row is inserted. 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 in the time of insertion. Once inserted, they will then have sufficient room for the data that you expect them to hold.

A database never shrinks

As you insert and delete rows from the database, the space they occupy is automatically reused. Thus, Anywhere may insert a row into space formerly occupied by another row.

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 a number of rows are deleted and no new rows small enough to use the empty space are inserted, the information in the database may become sparse. No utility exists to defragment the database file, as moving even one row might involve updating numerous index entries.

Since Anywhere automatically reuses empty space, the presence of these empty slots rarely affects performance. If necessary, you can reduce disk fragmentation by unloading, then reloading, the database.

Reloading also accomplishes another task. Since you are likely to reload each table in the order you frequently search them, the order the rows are stored in pages during the reload is likely to correspond closely to your preferred order. Hence, it is possible that this operation will improve database performance, much as a defragmentation utility improves disk performance by grouping all the pieces of each file together on the surface of the disk.

Table and page sizes

The page size you choose for your database can affect both the performance of your database. In general, smaller page sizes are likely to benefit operations that retrieve relatively small rows from random locations.

By contrast, larger pages tend to benefit queries that perform sequential scans, particularly when the rows are stored on pages in close to the order that the rows are retrieved via an index. In this situation, reading one page of 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 few large blocks more efficiently than more small ones.

Should you choose a larger page size, such as 4 kb pages, 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 can not hold enough pages, performance will suffer as Anywhere begins swapping frequently-used pages to disk.

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

There are many situations in which creating an index will improve 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 rows to be found quickly. It permits greater concurrency by limiting the number of database pages accessed. An index also affords Anywhere a convenient means of enforcing a uniqueness constraint on the rows in a table.

Hash values

Adaptive Server Anywhere must represent values in an index in order 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, Anywhere creates a corresponding hash value. It stores the hash value in the index, rather than the actual value. Anywhere can perform operations with the hash value, such as 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 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, Anywhere can use hash values that have a one-to-one correspondence to the actual value. 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 that contains 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 can not always compare two strings using only the hash values. If the hash values are equal, 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, Anywhere must retrieve each of the candidate rows and examine the full title.

Composite indexes

A **composite index** is one that is composed of an ordered sequence of columns. However, each index key in these indexes is at most a 9 byte hash value. Hence, the hash value can not necessarily identify the correct row uniquely. When two hash values are equal, Anywhere must retrieve and compare the actual values.

The effect of column order in a composite index

When you create a composite index, the order of the columns affects the suitability of the index to different tasks.

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 indices 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_lname;
```

Suppose you then want search for the first name John. The only index that is of useful is the one that contains the first name in the first column of the index. The index that is 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 indices.

Alternatively, you could make two indices that each index only one of the columns. Remember, however, that Anywhere only uses one index to access any one table while processing a single query. Even if you know both names, it is likely 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 order of the columns is that shown in your command.

Primary indexes and column order

Adaptive Server Anywhere uses a **primary index** to index primary keys. The primary index is a **combined index**: it also contains the entries for all foreign keys that reference this table, whether those foreign keys are located in the same table or in a different table.

The order of the columns in the index of a primary index is always that in which the columns appear in the definition of the primary table. In situations where more than one column appears in a primary key, you should consider the types of searches needed. If appropriate, you should switch the order of the columns in the primary table definition so that the most frequently searched for column appears first, or create separate indices, as required, for the other columns.

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 or HAVING 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*. Both predicates that involve comparisons with constants and those that compare columns from two or more different tables 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
```

```
PLAN>  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
employee.emp_id <> 123
```

```
PLAN>  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.

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* <> 10
x IS NULL	*x* IS NOT NULL
x > 25	*x* = 4 OR *y* = 5
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'

Sometimes it may not be obvious whether a predicate is sargable. In these cases, you may be able to rewrite the predicate so that 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* >= 'pat' and *x* < '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 that is used for indexed retrieval on a table is known as 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.

In other cases, a predicate may not be sargable simply because no suitable index exists. For example, consider the predicate X = Z. This predicate is sargable if these two attributes reside in different tables and at least one of them is the first attribute in an index. Should one of these conditions not be satisfied, the same predicate becomes non-sargable.

Semantic query transformations

In order to operate efficiently, Adaptive Server Anywhere usually rewrites your query. It changes it, possibly in several steps, into a new form. It ensures that the new version computes the same result, even though the query is expressed in a new way. In other words, Anywhere rewrites your queries into semantically equivalent, but syntactically different, forms.

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 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 commands and some of the ways in which it does so, are 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. 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 or HAVING clause.

Types of semantic transformations

The optimizer can perform a number of transformations in search of more efficient and convenient representations of your query. The following are common manipulations. Because the optimizer performs these transformations, the plan may look quite different than a literal interpretation of your original query.

- ♦ 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

The following subsections discuss each of these operations.

Unnecessary DISTINCT elimination

Sometimes a DISTINCT condition is not necessary. For example, the properties of one or more column in your result may contains a UNIQUE condition, either explicitly, or implicitly because it is in fact a primary key.

Examples

1 The distinct keyword in the following command is unnecessary because the product table contains a primary key. This column is part of the result set.

```
SELECT DISTINCT *
FROM product p
```

PLAN> p (seq)

2 Similarly, the result contains the primary keys of both tables so each row in the result must be distinct.

```
SELECT DISTINCT o.id, o.cust_id
FROM sales_order o JOIN customer c
   ON o.cust_id = c.id
WHERE c.state = 'NY'
```

PLAN> c (seq), o (ky_so_customer)

Subquery unnesting

You may express statements as nested queries, given the convenient syntax provided in the SQL language. However, these can often be more efficiently executed and more effectively optimized if rewritten in the form of joins. By doing so, Anywhere can take better advantage of highly selective conditions in a subquery's WHERE clause.

Examples

1 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, Anywhere recognizes that it can convert it to an inner join.

```
SELECT *
FROM sales_order_items s
WHERE EXISTS
    ( SELECT *
      FROM product p
      WHERE s.prod_id = p.id
         AND p.id = 300 )
```

PLAN: p (product): s (ky_prod_id)

Following conversion, this same statement is expressed using join syntax.

```
SELECT *
FROM product p JOIN sales_order_items s
   ON p.id = s.prod_id
WHERE p.id = 300
```

PLAN> p (product), s (prod-id)

2 Similarly, the following query contains a conjunctive EXISTS predicate in the subquery. This subquery can match more than one row.

```
SELECT *
FROM product p
WHERE EXISTS
    ( SELECT *
      FROM sales_order_items s
      WHERE s.prod_id = p.id
         AND s.id = 2001 )
```

PLAN> p (seq): s (ky_prod_id)

Anywhere converts this query to a inner join.

```
SELECT DISTINCT p.*
FROM product p JOIN sales_order_items s
   ON p.id = s.prod_id
WHERE s.id = 2001
```

PLAN> TEMPORARY TABLE s (id_fk), p (product)

3 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 )
```

PLAN> p (seq): s (sales_order_items)

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
```

PLAN> s (sales_order_items), p (product)

Selectivity estimation

Selectivity is a ratio that measures how frequently a predicate is true.

The **selectivity** of a predicate measures how often the predicate evaluates to TRUE. Selectivity is defined as the ratio of the number of times the predicate will evaluate to true, to the total number of possible instances that must be tested. Selectivity is most commonly expressed as a percentage. For example, if 2% of employees have the last name Smith, then the selectivity of the following predicate is 2%.

```
emp_lname = 'Smith'
```

Selectivity is second only to join enumeration in importance to the process of optimization. Hence, the performance of the optimizer relies heavily on the presence of accurate selectivity information.

Adaptive Server Anywhere can obtain estimates of selectivity from four possible sources. It assumes no correlation between columns of a table and so calculates the selectivity of each column independently.

- **Column-statistics registry** Each time Anywhere performs a query, it saves selectivity information about the data in a column for future reference.
- **Partial index scans** The optimizer can examine the upper levels of an index to obtain a selectivity estimate for a condition on an indexed column.
- **User-supplied values** You can supply selectivity estimates in your SQL statement. If you do so, Anywhere will use them in preference to those from other sources.
- **Default values** If no other source is available, Anywhere can fall back on the built-in default values.

The scan factor is the fraction of pages in a table that need to be read.

The **scan factor** queries the fraction of pages in a table that needs to be read to compute the result. It is also usually expressed as a percentage. For example, to find the first name of the employee with employee number 123, Anywhere may have to read two index pages and, finally, the name contained in the appropriate row. If there are 1000 pages in the employee table, then the scan factor for this query would be 0.3%, meaning 3 pages out of 1000.

Although the scan factor is frequently small when the selectivity is small, this is not always the case. Consider a request to find all employees who live on Phillip Street. Less than one percent of employees may live on this street, yet, because street names are not indexed, Anywhere can only find the records by examining every row in the employee table.

Column-statistics registry

Adaptive Server Anywhere caches skewed predicate selectivity values and column distribution statistics. It stores this information in the database. Anywhere stores, logs, and checkpoints this information like other data. Adaptive Server Anywhere updates these statistics automatically during query processing.

The optimizer automatically retrieves and uses these cached statistics when processing subsequent queries. This selectivity information is available to all transactions, regardless of the user or connection.

Adaptive Server Anywhere manages the column-statistics registry on a first-in, first-out basis. It is limited in size to 15,000 entries. Anywhere saves the following types of information.

- ♦ column distribution statistics
- ♦ LIKE predicate selectivity statistics
- ♦ equality predicate statistics

Do not give Anywhere amnesia!

You can reset the optimizer statistics using the DROP OPTIMIZER STATISTICS command. If you do so, you erase all the statistics that Anywhere has accumulated.

Caution
Use the DROP OPTIMIZER STATISTICS command only *when you have made recent wholesale changes that render previous statistical information invalid. Otherwise, you should avoid this command because it can cause the optimizer to choose very inefficient access plans.*

If you erase the statistics, Anywhere must resort to initial guesses about the distribution of your data as though accessing it for the first time. All performance improvements that the statistics could provide will be lost.

Subsequent queries will gradually restore the statistics. In the interim, the performance of many commands can suffer seriously. Consequently, this command rarely improves performance and certainly never provides a long-term solution.

Partial index scans

When cached results are not available to the optimizer, it can decide to probe the directory of an index to estimate the proportion of entries that may satisfy a given predicate. Depending on the predicate and the index, this information may be very accurate.

For example, the optimizer might examine an index of dates to estimate what proportion refer to days before a given date, such as March 3, 1998. To obtain such an estimate, Anywhere examines the upper pages of the index that you have created on that column. It locates the approximate position of the given date, then, from its relative position in the index, estimates the proportion of values that occur before it.

Some cost may be involved in performing such scans because some index pages may need to be retrieved from disk, should they not already be available in the buffer cache. In addition, indices for very large tables, or primary indices for tables pointed to by a large number of foreign keys, may be extremely large. Low fan-out may mean that the optimizer could only obtain specific estimates by examining many pages. To limit this expense, the optimizer examines at most two levels of the index.

Naturally, this method is effective only when the column about which selectivity information is sought is the first column of the index. Should the column comprise the second or more column of the index, the index is of no help because the values will be distributed though out the index.

Similarly, estimates of LIKE selectivity values may be obtained by this method only when the first few letters of the pattern are available. In cases where only the middle or final sections of a word pattern appear, the optimizer must rely on one of the other three sources of selectivity information.

User estimates

Adaptive Server Anywhere allows you, as the user, to supply selectivity estimates of any predicate. These estimates are expressed as a percentage and must be supplied as a floating-point value. You may explicitly state such an estimate for any predicate you choose.

In cases where a user-supplied estimate is available, the optimizer will always use it in preference to an estimate available from any other source. In this situation, it even ignores cached selectivity values for that predicate.

Because the optimizer always uses any explicit estimates you provide, you can use these estimates to guide the optimizer in its choice of access plan.

You should use explicit estimates with care. Estimates in triggers or stored procedures are easily forgotten. Anywhere has no means to update them. For these reasons, all responsibility for their maintenance rests with the author of the procedure or administrator of the database. Should the distribution of data change over time, the values may prove inappropriate and lead the optimizer to choose access plans that are no longer optimal.

Default selectivity estimates

When all else fails and it can obtain estimates from none of the other three sources, the optimizer must fall back on default selectivity estimates.

Anywhere assumes that statistics in the column statistics registry are both present and accurate. For example, if the optimizer is considering a LIKE predicate, it looks in the column-statistics registry. If the registry contains no entry for that predicate, it assumes that none is stored because the selectivity is less than a small threshold value. Since default selectivity estimates are not specific to your data, they can mislead the optimizer into selecting a poor access plan.

When Anywhere executes that plan, it will use the results to save better selectivity estimates in the column statistics registry. If you execute the same query later, it will find these more accurate estimates and adjust the access plan if appropriate. For this reason, performance may be poor the first time or two you execute a particular query on a new database, or after dropping the optimizer statistics.

Anywhere uses the following default selectivities.

Predicate	Default selectivity
Column comparisons: equality to a constant, IS NULL, or LIKE	0.035% (if not stored in registry)
Column comparisons: inequality to a constant	25%
Other LIKE or EXISTS	50%
Other equalities	5%
Other inequalities	25%
Other IS NULL or BETWEEN	6%

Equijoin selectivity estimation

What is an equijoin?

Frequently, you will need to join two or more tables to obtain the results you need. Equijoins join two tables through equality conditions on one or more columns, as in the case of the following query.

```
SELECT *
FROM tablea AS a JOIN tableb AS b
   ON a.x = b.y
```

Join selectivity for equijoins

In the case of equijoins, Anywhere calculates the selectivity of the join based on the cardinality of the individual tables according to the following formula.

selectivity = cardinality(*a* JOIN *b*))/(cardinality(*a*) cardinality(*b*))

If the join condition involves two columns, then the optimizer uses data distribution estimates from the column statistics registry to estimate the cardinality of the result, and hence the selectivity of the join. Otherwise, if the join condition involves a mathematical expression, the join predicate selectivity estimate defaults to 5%.

Key joins—a rare case where syntax matters

The optimizer takes advantage of joins that are based on foreign key relationships. You can identify these to Adaptive Server Anywhere using the KEY JOIN syntax. When you use this syntax, the optimizer can estimate selectivity accurately using special information contained in the primary index. Anywhere only takes full advantage of these relationships when you explicitly use the KEY JOIN syntax. As such, it is a rare exception to the general rule that Anywhere optimizes your commands based on their semantics, not their syntax. When estimating the selectivity of key joins, the Anywhere optimizer assumes a uniform distribution of the values in the table that contains the foreign key.

Diagnosing and solving selectivity problems

Selectivity estimation problems are the root of most optimization problems. The following sources of information are available to help you.

Displaying estimates and their source

Anywhere can tell you the value of a selectivity estimate and the source of that estimate. You have access to this information through the built-in functions ESTIMATE and ESTIMATE-SOURCE.

The following command displays the selectivity estimate that the optimizer will use for queries which select entries from table T which in which column x contains a value greater than 20.

```
SELECT ESTIMATE (quantity, 20, '>')
FROM product
```

Anywhere displays the result of this command as a percentage.

Similarly, the following command displays the source of that estimate. The optimizer can contain estimates from a number of sources, including column statistics registry and user supplied values.

```
SELECT ESTIMATE_SOURCE (quantity, 20, '>')
FROM product
```

Solving selectivity problems

If you find that Anywhere is obtaining an incorrect selectivity value from the registry, you can easily reset the value by issuing any command that will perform a complete scan of the table for that condition. For example, the following SQL statement will cause Anywhere to locate all the entries in **product** table which have quantities equal to 20.

```
SELECT *
FROM product
WHERE quantity = 20
```

```
PLAN> product (seq)
```

Whenever Anywhere completes execution of a statement such as this one, it automatically updates the column statistics registry based on the results.

You can use a similar tactic to load initial selectivity information into a new database. Simply issue commands that contain conditions that will appear in common statements. When the optimizer later prepares to execute a statement, it will generate a better plan because the correct statistics will be available.

Maintain or remove hard-coded selectivity estimates

Users can supply and hard-code selectivity estimates in triggers or stored procedures while developing the database. Once encoded, the database administrator assumes responsibility for their maintenance as Anywhere has no means to update them automatically.

Unfortunately, these hard-coded values are hidden and may become inaccurate as the information in the database grows and changes. For this reason, you should avoid using them, except in very unusual cases where you can encourage the optimizer to choose a better access plan by no other means. Often, you can avoid using them by priming the database using sample queries as described above.

Join enumeration and index selection

Join enumeration, the process of costing each possible join strategy and making a selection, is the heart of any optimizer. Adaptive Server Anywhere uses a proprietary join enumeration algorithm to search for an optimal access plan. This algorithm considers the cost of various strategies and works to find an inexpensive strategy.

When processing any query, Anywhere always accesses any one table by one method. It either scans the table sequentially, or selects one—and only one—index and accesses the rows through it.

Join enumeration

In selecting a join strategy, Anywhere considers the following pieces of information.

- ♦ selectivity estimates of the number of rows in each intermediate result
- ♦ estimates of scan factor for each indexed retrieval
- ♦ the size of the cache—different cache sizes can lead to different join strategies.

Anywhere begins by using selectivity information, as determined in the previous step, to select an access order.

Anywhere derives the estimates of scan factors from estimates of index fan-out. The fan-out of an index can vary greatly depending on the type of index and the page size that you selected when you launched the engine or created the database. Larger fan-out is better, because it allows Anywhere access to locate specific rows using fewer pages and hence fewer resources.

Cache size affects the access plan

The amount of cache space available to Anywhere can affect the outcome of the optimizer's choice of join strategy. The larger the fraction of space consumed by any one query, the more likely that pages will need to be swapped for those on disk. If Anywhere decides that a particular strategy will result in using excessive cache space, it assigns that strategy a higher cost.

The number of possible join strategies can be huge. A join of *n* tables allows *n*! possible join orders. For example, a join of 10 tables may have 10! = 3,628,800 possible orders.

When faced with joins that involve a large number of tables, Anywhere attempts to prune the set of possible strategies. It eliminates those that fall into certain categories, so as to focus effort on investigating possibilities which are more likely to be efficient.

Anywhere chooses plans with fewer Cartesian products

Anywhere always selects plans that minimize the number of Cartesian products required to compute the result. Instead, it favors indexed access.

Index selection

In addition to selecting an order, the optimizer must choose a method of accessing each of these tables. It can choose to either scan a table sequentially, or to access it by through an index. Some tables may have a few indexes, further increasing the number of possible strategies.

The optimizer analyzes each join strategy to determine which type of access—indexed or sequential scan—would best suit each table in that strategy. Although one index may be well suited to one join strategy, it can be a poor choice for another strategy that joins the tables in a different order. By making a custom index selection for each join order, the optimizer gains the opportunity to choose a better access plan.

Anywhere decides to use an index instead of embarking on a sequential scan whenever an index is available and the selectivity is less than 20%.

Cost estimation

The optimizer bases its selection of access plan on the expected cost of each plan. It uses a mix of metrics to estimate the cost of an access plan:

- expected number of rows
- use of temporary tables
- anticipated amount of CPU and I/O for the access plan
- amount of cache utilized

In recognition that disk access is substantially more time-consuming than other operations, Anywhere gives it particular weight.

Associate high cost with temporary tables

In keeping with the assumption that Anywhere is to use both disk and memory efficiently, it avoids using temporary tables. To achieve this goal, the optimizer assigns significant cost to plans that use them.

Anywhere bases its estimate of the cost of temporary table on both the row size and the expected number of rows that the table will contain. The optimizer often pessimistically overestimates the actual cost of using a temporary table. When few queries are competing for cache space, the actual cost of a plan with a temporary table can be significantly less than the estimate.

Costing index access

Anywhere calculates a scan factor for each table accessed. For this calculation, it uses both selectivity estimates and the fan-out of the index.

If the index is a key index, then Anywhere assumes that the entries are uniformly distributed in the corresponding table. However, Anywhere assumes that values in the primary-key index are clustered near similar values. This assumption is usually valid. For example, suppose you use a auto-increment column to generate primary-key values. The rows in the table will lie in roughly the same order in the pages of the table as they do in the primary index.

When the rows in a table are arranged on the database pages in the order you wish to read them, less cache space is required because Anywhere can avoid rereading the same pages from disk.

Subquery caching

New to Adaptive Server Anywhere 6.0 is the ability to cache the result of evaluating a subquery. When Anywhere processes a subquery, it caches the result. Should it need to re-evaluate the subquery for the same value, it can simply be retrieve the result from the cache. In this way, Anywhere can avoid many repetitious and redundant computations.

At the end of each subquery, Anywhere releases the stored values. Since values may change between queries, these values may not be reused to process subsequent queries. For example, another transaction might modify values in a table involved in the subquery.

As the processing of a query progresses, Anywhere monitors the frequency with which cached subquery values are reused. If the values of the correlated variable rarely repeat, then Anywhere needs to compute most values only once. In this situation, Anywhere recognizes that it is more efficient to recompute occasional duplicate values, than to cache numerous entries that occur only once.

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.

As soon as Adaptive Server Anywhere recognizes that few values are repeated, it suspends subquery caching for the remainder of the statement and proceeds to re-evaluate the subquery for each and every row in the outer query block.