BY DAVID McGOVERAN

The performance of on-line, distributed systems hinges on the optimizer's intelligence in handling dissimilar queries and remote data

Evaluating Optimizers

UNDERSTANDING how to evaluate a relational DBMS query optimizer can be a big help in estimating the DBMS's performance capabilities. It can also tell you a good deal about functionality: whether the DBMS is appropriate for ad hoc query processing, distributed database management, concurrent or parallel processor exploitation, batch and on-line transaction processing (OLTP), on-line complex processing (OLCP), disk cache management, parallel disk I/O, and so on. For each of these capabilities, the optimizer can be the functional bottleneck. If it is unable to select an access strategy that uses these capabilities, their benefits won't be fully realized.

Using the Ingres optimizer from Ingres Corp. (formerly Relational Technology Inc.) as a model, we'll discuss the concepts and terminology necessary to question a vendor about optimizer functionality and to determine whether the optimizer will meet your needs. Not only does Ingres have a common structure, it also implements a number of techniques not found in other products. While it doesn't use all the techniques described here, it's a good model for comparing and learning about optimizers.

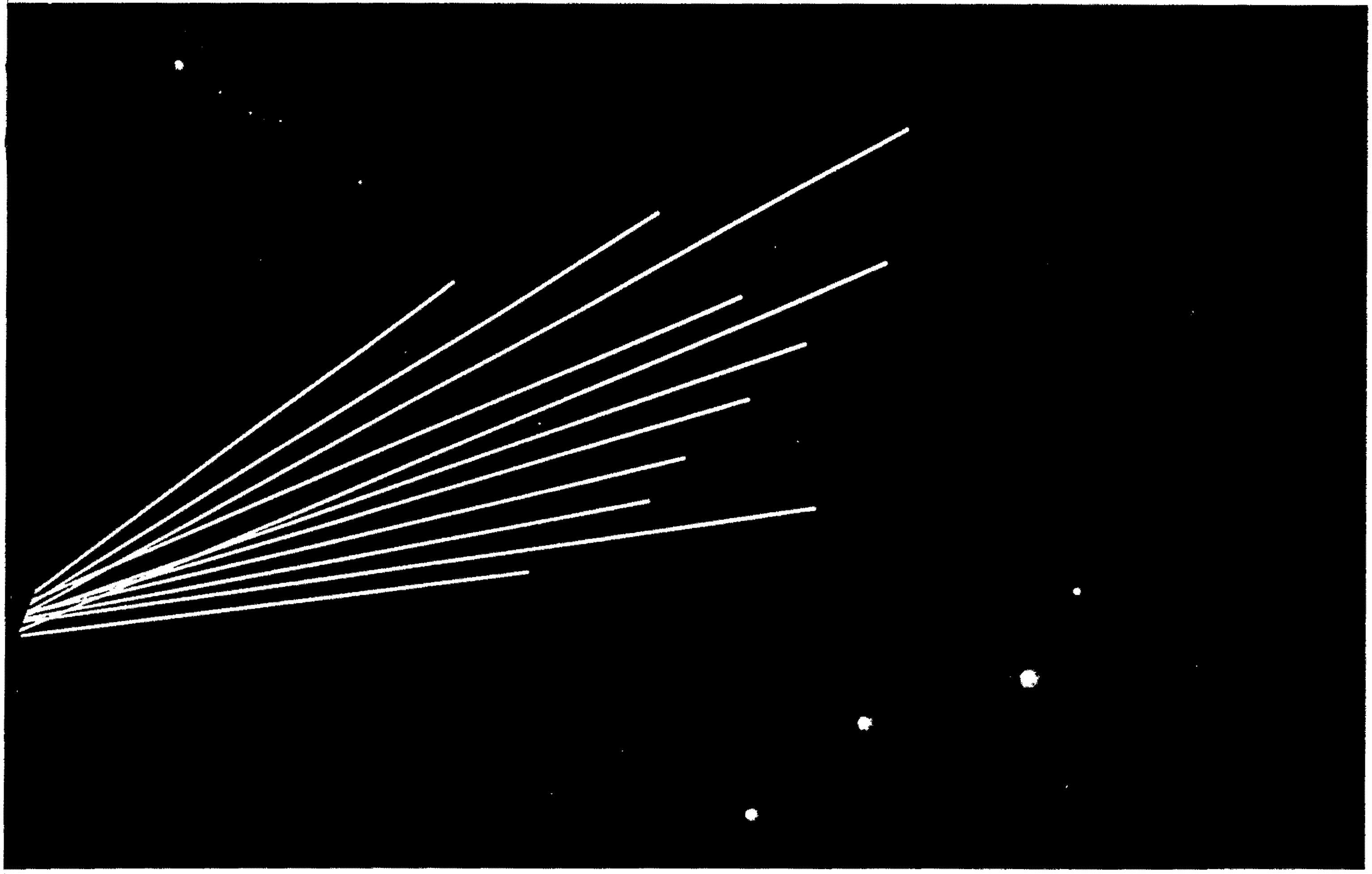
WHY QUERY PROCESSING?

If you've never wondered what happens between the time you enter an SQL statement and the time the results are returned, you may be surprised to discover how much processing takes place before the first data access. A naive approach would avoid such processing, but it takes only a little reflection to recognize how unrealistic that approach would be.

Suppose you enter the following query:

```
SELECT TABLE_A.COLUMN_1,
       TABLE_B.COLUMN_2
  FROM TABLE_A, TABLE_B
  WHERE TABLE_A.COLUMN_1 =
  TABLE_B.COLUMN_1
```

In principle, this SELECT can be processed in three steps. First, the Cartesian product of TABLE_A with TABLE_B (the concatenation of all possible ordered pairs of rows, the first from TABLE_A and the second from TABLE_B) is formed. This creates an intermediate result table—let's call it TABLE_C—that won't be saved when



processing is completed. The resulting table is then restricted to those rows that match the WHERE clause condition, and the desired columns are projected.

It is useful to be able to estimate the size of TABLE_C quickly. Suppose TABLE_A has m rows and TABLE_B has n rows, with m greater than or equal to n. C, the total number of rows in this table (regardless of any restrictions that may come up later), is bounded by $m^2 >= c >= n^2$.

If both m and n are small, this naive approach to query processing can work. But if both are large, TABLE_C will be extremely large. The width w (number of bytes in a row) of TABLE_C is the sum of the widths of TABLE_A and TABLE_B. The total storage required for TABLE_C is w*m*n. For m and n equal to a mere 1,000 and each table 100 bytes wide, TABLE_C requires 200 megabytes of temporary storage. Modern production databases can have tables with millions of rows, making 200 terabytes of temporary storage possible!

Clearly, selecting a better processing strategy is desirable, especially when you realize the restriction step could throw away all these temporary rows. Not only does the naive approach use a great deal of temporary storage, it can require large amounts of disk I/O. It's one of the most inefficient ways to use your resources.

PLANS AND STRATEGIES

The basic steps in producing an optimal processing strategy, sometimes called a query execution plan, are shown in Figure 1.[3] Selecting the strategy involves parsing and compiling the query into an internal representation. This abstract view of the high-level operations (join, projection, restriction, sort, and so on) required to execute the query usually takes the form of a tree. The leaf nodes represent tables; nonleaf nodes represent operations (Figure 2).

Some optimizers (like Ingres) convert the query or parse tree into canonical form. This form is a unique representation of all equivalent queries regardless of how the query was written. Converting to this form lets the optimizer consider fewer logical operations and allows the queries to be compared systematically. As we will see, Ingres's strategy makes this an important feature.

ARTWORK: KEN SCALLON

Step 1: Convert the query into an internal representation.

For example, "SELECT col1, col2 FROM table_A WHERE col1 NOT = 1234 OR col2 NOT = 4321" is converted to tokens for the following:

```
                    Project-restrict (col1, col2)
                         /                  \
           table_A                              table_A
     (col1 NOT = 1234)                      (col1 NOT = 4321)
```

Step 2: Convert the representation into a canonical form.

In conjunctive normal form, the tree has only one child node.

```
                    Project-restrict (col1, col2)
                         /
              table_A
   NOT (col1 = 1234 AND col2 = 4321)
```

Step 3: Choose possible low-level processing methods.

Suppose the project-restrict operation can be performed in one of two ways (PR1 or PR2) and table_A can be accessed in one of two ways (AM1 or AM2) to produce the desired result.

Step 4: Generate all possible plans.

The search space then consists of four possible access plans, represented schematically as follows:

```
      PR2           PR1           PR2           PR1
     /             /             /             /
  AM1           AM1           AM2           AM2
```

Step 5: Evaluate and choose the optimal plan.

Again, schematically, if the evaluated costs (shown in parentheses) are 5 for PR1, 10 for PR2, 7 for AM1, and 4 for AM2, the total costs are shown in square brackets. For the purposes of this simplified illustration, the effects of the amount of data processed at each step have not been taken into account. Clearly, the plan involving PR2 and AM1 incurs the least cost and would be the query execution plan.

```
     [12]          [17]           [9]          [14]
     PR2           PR1           PR2           PR1
     (5)           (10)          (5)           (10)
     /             /             /             /
  AM1           AM1           AM2           AM2
  (7)           (7)           (4)           (4)
```

FIGURE 1. *Optimization steps.*

A common practice is to use conjunctive normal form, meaning that ORs in the WHERE clause contain only ANDs and ORs and that the ORs occur only in Boolean subexpressions (for example, (A OR B) AND (C OR D) AND () . . .). This form allows the optimizer to recognize joins easily. An alternative is to use disjunctive normal form, which replaces ANDs with ORs and NOTs in the WHERE clause. This form helps the optimizer recognize efficient means for keying into a relation.

Because SQL allows subqueries to be nested, Ingres may also flatten the query to a form that uses joins instead of subqueries wherever possible. This approach eliminates the need to optimize subqueries as if they were additional queries.

Another step in the conversion to canonical form is decomposition, the process of turning one query into two or more simpler queries. This reduction is continued until irreducible parts are produced. Tuple substitution and detachment are two decomposition methods. This divide-and-conquer approach fine-tunes the optimizer to work on a class of simple queries. The assumption (which has been proven for nondistributed-database processing) is that optimizing these simple queries and then summing the resource costs from the bottom up is an optimal strategy. Whether this theorem is true for a distributed optimizer remains to be seen.

Once a query has been reduced to canonical form, the optimizer chooses one or more ways to perform the operations specified at each node of the query tree. The tree is generally read from the bottom up and left to right (Figure 3). Each possible set of assignments of one method to each node results in an access plan, represented as a tree (or *query plan tree*).

Two adjacent nodes can usually be switched. Indeed, there should never be an order dependence between the nodes of the query plan tree. If the database allows for duplicates or does not consistently eliminate them from intermediate results, the results will be order-dependent and not all node orderings on the tree will be allowed.[1,2]

The order of the nodes (operations) is generally selected to reduce the size of the lower node's result. This is the optimizer's attempt to make intermediate results small enough to be processed in the cache. Unfortunately, it also means that the number of possible access plans the optimizer must evaluate (called the *search space*) becomes even larger.

PRUNING THE SEARCH SPACE

This explosion of the number of access plans must be controlled. The process of eliminating (or never even generating) access plans that are likely to be costly is referred to as "pruning the search space." The optimizer may use heuristics to estimate the processing cost of the plans quickly. Plans that will clearly result in costly processing can be eliminated from more detailed analysis. In fact, optimizers have been known to rely entirely on such estimates when selecting an access plan.

When the search space has

Parent
Left Child
Right Child

Ingres represents query execution plans as binary trees. The tree is read in post-order sequence (left child, right child, and parent, recursively). Leaf nodes have no children and contain details about tables (base data sources). Nonleaf nodes represent an operation to be carried out on the data resulting from the child operation (or on base data if the child is a leaf node). The root or top node represents either a join or a sort operation.

Before asserting the kind of join, projection-restriction, or sorting to be done, the tree is a query tree—it refers only to the generic operations to be performed, not how they are to be performed. Each tree in the forest that can be formed by assigning specific access methods to each node is an access plan. When one is determined to be optimal, it is called a query execution plan (Ingres) or, more generically, query plan.

FIGURE 2. *Tree representations.*

The following query execution plan says the SQL statement will execute by accessing table_A via a hashed index on column 1, sorting on column 1 and project-restricting (discarding all columns except A.col1 and A.col2), accessing table_B via an ISAM index on column 1, sorting the result and project-restricting (discarding all columns except B.col1 and B.col2), and performing a full sort-merge join (A.col1 = B.col1) on the resulting heap.

SELECT A.col2, B.col2 FROM table_A A, table_B B WHERE A.col1 = B.col1

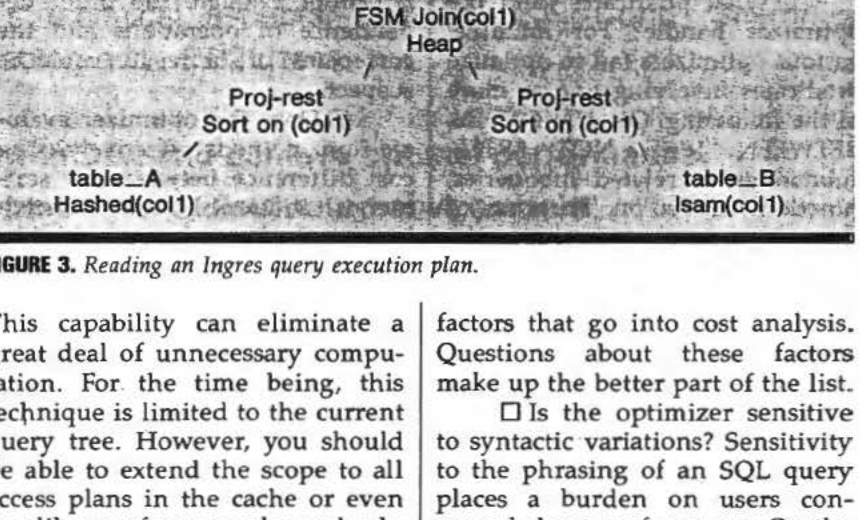


FIGURE 3. *Reading an Ingres query execution plan.*

been reduced as much as possible, the access plans can be evaluated. Evaluation involves calculating a cost function for each node. The idea is to estimate the resources required to perform a particular operation. Cost functions typically depend on the depth (number of rows) of input tables (tables to be accessed or on which a relational operation is to be performed) and on resource factors determined by the algorithm. For example, if an entire table must be read sequentially (a relation scan), the number of disk I/Os can be estimated from the number of rows in the table, the size of each row, and the average number of rows stored per disk page. Given the time required for each disk I/O and an estimate of the CPU time required to process each row, the optimizer can estimate the total processing time.

The costs of each node in an access plan are computed from the bottom up, left to right. This allows Ingres to estimate the size of the result tables from each operation so they can be rolled up into the cost computation for the next higher node. Ingres keeps track of the estimated sizes of the intermediate result tables. If the cost function is linear in the number of result rows or size, the cost can be scaled for circumstances in which the input results differ.

The computational effort involved in evaluating all possible access plans can be enormous. For this reason, optimizers such as Ingres's don't ordinarily perform an exhaustive search even on the pruned search space. Ingres allows the user to set a limit on the amount of time spent on this effort. The optimizer normally stops searching once it has spent an amount of time equivalent to a fraction of the lowest time cost found for the plans evaluated so far. Other heuristics for stopping the search may be used as well.

Various techniques may be used to minimize the cost of optimization. For example, frequently used queries can be compiled and a reusable access plan cached or stored for them. Perhaps even more important is how the Ingres optimizer recognizes portions of an access plan for which it has already computed the cost functions. This capability can eliminate a great deal of unnecessary computation. For the time being, this technique is limited to the current query tree. However, you should be able to extend the scope to all access plans in the cache or even to a library of commonly used subtrees. Of course, you would have to be careful to avoid an undesirably large search of the library.

GENERAL FEATURES

The best way to find out about an optimizer, short of extensive testing, is to ask the vendor. Characteristics pertain to the optimizer's distributed capabilities and its ability to reduce a query to canonical form, flatten subqueries, and manage optimization. While some of these appear in the questions that follow, a key way to characterize an optimizer is to look at the factors that go into cost analysis. Questions about these factors make up the better part of the list.

☐ Is the optimizer sensitive to syntactic variations? Sensitivity to the phrasing of an SQL query places a burden on users concerned about performance. On the other hand, it also allows the sophisticated user to optimize the query manually. If your application uses a fourth-generation language that generates SQL, more complex SQL is unlikely to be optimal. Syntax sensitivity is removed by converting to canonical form or by flattening so that phrases that are logically but not syntactically identical are optimized the same way.

☐ Can the optimizer perform a semantic transformation? If a relational DBMS supports integrity constraints, a query that is semantically (as opposed to syntactically) equivalent to another query can be transformed automatically into the other. This can simplify the query in ways that canonical form and flattening cannot, thus improving query performance and allowing more consistent performance for queries that have the same meaning regardless of their phrasing.

☐ Does the optimizer recognize and use transitivity on equijoins and other logical laws? If the WHERE clause contains the equijoins a=b and b=c, and primary indexes exist on a and c but not on b, the join a=c is implied and can be performed in the index. This is likely to reduce the number of rows that must be joined to b. Other useful laws include DeMorgan's (NOT(A AND B) = NOT A OR NOT B and NOT(A OR B) = NOT A AND NOT B) and the equivalence of NOT GREATER THAN and LESS THAN OR EQUAL TO. Ideally, the optimizer uses these laws so that users don't have to be logicians.

☐ What predicates can't the optimizer handle? For example, various optimizers fail to optimize predicates involving one or more of the following: OR, UNION, IN, BETWEEN, LIKE, NOT, NULL, subqueries, correlated subqueries, functions, and so on. The more of these the optimizer supports, the greater the power of the language and its utility in mission-critical applications.

☐ Does the optimizer have an EXPLAIN facility? EXPLAIN allows the user to obtain a description of the access plan the optimizer has selected for the query. This can be extremely useful in performance optimization if the user has some way to influence the optimizer. Some DBMSs provide a means of doing this directly (for example, sending an access plan to the optimizer); others provide an indirect means (modifying the syntax of the query or modifying indexes). Some means of influence is necessary in production MIS shops.

☐ At what point does the complexity of the query prove too much for the optimizer? For example, it's not uncommon to see the number of tables allowed in a query restricted to 16. However, some optimizers give up when the number of tables in a join is as few as five or six. For OLCP and ad hoc decision support, this shortcoming is unacceptable.

Optimizers use logical laws so that users need not be logicians

ACCESS-METHOD SUPPORT

Optimizers are designed to eliminate obviously expensive access paths. But to provide true access-method support, they should also be able to reconstitute queries for better performance.

☐ Can the optimizer manipulate the order of operations? As noted earlier, the order of operations can be used to produce smaller intermediate results. If the optimizer doesn't attempt to compute or track intermediate results, it cannot evaluate the order dependence of operations and the correctness of the results could be suspect.

☐ Does the optimizer evaluate join methods effectively? The cost difference between the sort-merge, index-only, and nested-loop methods can be considerable. If the optimizer treats all joins equally or fails to evaluate a method properly, anomalous performance behavior can result.

☐ Does the optimizer select an appropriate sorting algorithm? Each algorithm has advantages and disadvantages. For example, a Quicksort becomes embarrassingly slow when faced with data that has already been sorted. If the optimizer uses a single sorting algorithm, it should be one for which such anomalous results do not occur, or the conditions that cause anomalous results should be recognized so that the algorithm isn't applied inappropriately.

☐ Can the optimizer take advantage of inter- and intratable clustering? It should recognize the cost advantage of intratable clustering (optimizing physical storage order for a single table) for sorted retrievals or range queries. Looking up only the minimum and maximum values in the index is then sufficient to determine the disk pages that must be read. Similarly, the optimizer should recognize that intertable clustering (storing rows from two or more tables together) may be advantageous when a primary-key join on the clustered tables is required. It is disadvantageous when only one of the clustered tables is desired.

☐ Does the optimizer take I/O bandwidth into account? To put it simply, not all disk drives are created equal. Some means of factoring in disk-drive performance, preferably at configuration time, is advantageous. The administrator might put rotational speed, latency, and mean transfer rate in a table for use by the optimizer whenever a drive or other storage medium is added to the system. Alternatively, the system might monitor drive performance automatically. An extreme example is the ineffectual use of a RAM disk drive if treated as a standard drive with access times measured in milliseconds.

☐ Does the optimizer evaluate the advantage or cost of buffering? All systems have limited buffer space. If the optimizer assumes effectively unlimited buffer space, the cost estimate may be too low due to paging. On the other hand, if it ignores the possibility of caching and buffer management, the estimate will be too high. The cost function should represent the available buffer-management algorithms.

☐ Does the optimizer take into account the costs of transaction management, journaling, consistency enforcement, and concurrency? Resource waits and setting and resetting locks should contribute to the overall costs. Statistics such as the probability of deadlocks and average resource wait based on the number of concurrent users (important in OLTP and OLCP applications) should be factored in.

INDEX SUPPORT

Database designers create indexes to save I/O and processor time. Because indexes are implemented by the optimizer, the degree to which the optimizer meshes with various indexing strategies is crucial.

□ Does the optimizer use different kinds of indexes effectively? Various indexing methods exist, each with a different utility (possibly implying a different storage structure). For example, a hash index is better for single-record access, while a B-tree is better for finding ranges of values. Indexes can be made ineffectual by poor optimizer evaluation, necessitating considerable care in designing the index to compensate for it.

□ How well does the optimizer recognize indexes? A restriction can be processed concurrently with data access using indexes if recognizable indexes exist on the columns mentioned in the WHERE clause. An optimizer might not always recognize the usefulness of an index. For example, if the index is composed of three columns and only the first two are mentioned, the optimizer may not recognize that the columns form part of an index. Optimizers may not recognize a column as belonging to an index if it's used in a computation or function or if the column occurs in certain kinds of comparisons.

□ Can the optimizer use or at least deal with multiple indexes? If more than one index is potentially useful, the optimizer may not use any. An effective optimizer not only uses multiple indexes but attempts to perform an index join where possible. The idea is to access only those pages referenced by all relevant indexes. When the optimizer cannot take advantage of yet another relevant index, it's time to stop creating indexes.

□ Can the optimizer automatically create useful indexes? When a potentially useful index does not exist, some optimizers create it assuming that the cost of creation is lower than the cost of processing alternate plans. This is typically a temporary index that disappears after the statement is processed. However, knowledge of usage patterns may be used to determine the cost of creating a permanent index. Heavily used stored procedures or repeated queries in a read-intensive environment, for example, may benefit from such indexes. Similarly, sorting or creating temporary indexes on intermediate results can sometimes be beneficial. This feature can be important in processing very large base tables and in OLCP, batch, and decision support applications, where large result tables are more common.

□ Can the optimizer use multitable indexes? Some systems can create a single index on the keys in multiple tables. This mechanism speeds joins and can be used to enforce referential integrity. The optimizer should not only recognize when these indexes are useful but be able to use the index to look up keys from either table. If it can't, additional indexes may be required, increasing the cost of updates.

□ Can the optimizer respond to user-created index and access methods? A few relational DBMSs let the user specify these methods as user exits. They are useful in creating gateways or dealing with applications with special data (computer-aided design, for example). If the optimizer doesn't recognize these methods, they are of little use where performance is important.

□ Are nulls supported? Some optimizers refuse to use any index (not just the primary-key index) on a column that can contain nulls. Others cannot optimize a restriction that involves nulls (in other words, IS NULL and IS NOT NULL).

STATISTICS

The optimizer uses statistics to estimate the number of I/O operations required for each possible access path. The statistics describe the tables to be searched in formulas specific to the type of search required.

Classifying the Optimizer

IT IS CONVENIENT when evaluating an optimizer to classify it according to the features available. A scheme that I find useful is as follows:

□ Class 0—There is no optimizer.

□ Class 1—The optimizer is syntax-based and therefore syntax-sensitive. It does not use statistics but does use either cost indexes or heuristics to determine the access plan.

□ Class 2—The optimizer is syntax-based but does use cost functions. It does not use statistics (except perhaps an estimate of the table size).

□ Class 3—The optimizer is syntax-based and uses cost functions and minimal statistics.

□ Class 4—The optimizer is not syntax-based and uses cost functions and statistics. Facilities to influence the optimizer manually may be available.

□ Class 5—The optimizer is not syntax-based and uses cost functions and extensive statistics, including those about the distribution of data values. The user may be able to limit the search cost or force exhaustive search.

□ Class 6—The optimizer is not syntax-based, uses cost functions and extensive statistics, and performs global optimization in a distributed database environment.

These seven classes are presented only as a guide. The classification assumes that performance is not influenced by syntax, that cost functions are better than cost indexes or simple heuristics, that statistics are useful, and that user facilities for controlling the optimizer are beneficial.

It is important to understand that an optimizer in one class may have some characteristics found in a higher class. Similarly, an optimizer in a higher class may still use the techniques found in a lower-class optimizer but should do so judiciously.

It may be that certain characteristics, while not rated highly by the classification, will nonetheless be acceptable. For example, a DBMS that optimizes only at compile time (as opposed to execution or run-time) can justify an exhaustive search. If the optimizer is then able to reuse access plans or partial plans, this technique may not be particularly detrimental, even in an ad hoc environment. Similarly, if the users of the DBMS are sophisticated and willing to risk possible performance penalties resulting from poorly phrased SQL, syntax-based optimizers may be acceptable.

☐ Is the optimizer sensitive to table and index statistics? A table's cardinality (number of distinct rows) and maximum, minimum, and average column values are likewise useful for heuristic measures. Other factors include the average number of rows per page, number of pages per relation, percent of total pages, and number of index pages. For an index, selectivity is important. This is the number of entries in the table for each entry in the index. Some statistical optimizers rely entirely on these statistics.

☐ Does the optimizer keep track of data value distributions? Ingres maintains fairly sophisticated statistics about the distribution of data values in a table, allowing the optimizer to analyze the number of disk I/Os required to meet a restriction. If the optimizer assumes a flat distribution of values across all pages and the distribution is skewed or multimodal, the number of pages required to access values can be quite different from the estimate.

☐ Can the optimizer estimate the number of disk pages that must be accessed? If not, the cost of disk I/O cannot be computed. Such optimizers typically use a cost index rather than a cost function. A cost index is a numeric value given to a particular operation regardless of the amount of data that must be processed. The sum may then be scaled according to the amount of data. The result is optimization based on gross statistical or theoretical assumptions. This is similar to measuring the health of the U.S. economy based on the Dow Jones industrials instead of computing the GNP, the trade deficit, and inflation.

☐ How does the optimizer update statistics? Updating can be continuous, automatic, manual, on a scheduled interval, or by trigger (for example, when new extents are allocated and new indexes are created). The most desirable approach depends on usage patterns and data value distribution.

☐ How costly is it to obtain statistics from the database? When a database is very large, the cost of updating the statistics can become exorbitant. One solution is to produce the statistics by sampling the data. A variation is to generate the statistics from a sample or test database. Similarly, statistics can be updated continuously, either for all activity or on a random-sampling basis. Up-to-date statistics are critical for performance in a distributed DBMS.

Up-to-date statistics are critical for performance in a distributed DBMS

EFFICIENCY FEATURES

The optimizer must use storage intelligently to prevent the DBMS from monopolizing computer resources. These features can also improve flexibility when handling different kinds of procedures.

☐ Can access plans be saved or cached? Caching eliminates the overhead of generating an access plan when an application uses a query repeatedly. Similarly, database procedures generate access plans that are stored in the database on the first invocation. The optimizer should provide a way to check the validity of stored or cached plans and be able to regenerate a plan automatically.

☐ Can the optimizer recognize and use parts of an existing plan? It might maintain a cache of the plans it recently executed. If a new plan needs to be selected, the optimizer can try to find an existing equivalent plan. Failing this, it should look for parts of the plan that have already been computed and optimized.

☐ Does the optimizer recognize invalid plans or partial plans? Changes to the statistics or the database schema can invalidate a plan (or partial plan). Likewise, a minor variation on a plan or partial plan can be invalid (for example, substituting a wildcard for a constant in a Boolean comparison predicate).

☐ Can both statements and transactions be optimized? If the optimizer can look ahead and see all the statements that make up a transaction (as may be the case in a database or stored procedure), it may be able to cache intermediate results for use later in the transaction. It may also be possible to minimize the amount of disk I/O by scanning a clustered table even though some of the data will be used only by later statements in the transaction.

AI FEATURES

Optimizers are often called the "intelligence" of the DBMS, but not all of them take full advantage of the latest artificial-intelligence technology.

☐ Does the optimizer use heuristics to eliminate plans? During cost-function evaluation, heuristics can be used to terminate evaluation of the entire plan. These techniques allow optimizer time to be spent examining potentially more useful plans.

☐ Can the optimizer learn? In some sense, an optimizer learns if it collects and responds to statistics. However, additional forms of learning are possible. For example, usage patterns can be used to dictate the spread of data across devices or even across data pages. The frequency and quantity of updates can dictate the optimal extent size for allocating new pages. Frequent access of a column can lead to automatic creation of indexes. Automatic reorganization of data is possible when a particular order is common or fragmentation exceeds a cost threshold.

DISTRIBUTED SUPPORT

Less centralized data, network considerations, and proliferating access paths for incoming queries will all become factors in the performance of a DBMS's optimizer. Distributed systems will tax the optimizer's ability to determine (and allocate) the costs of accessing data.

☐ Can the optimizer compute distributed cost functions? When data is distributed, factors such as routing of the retrieved data, network bandwidth, node CPU speed, and possible concurrent or parallel processing of the decomposed query all become important in computing costs. These factors as well as the normal local statistics must be available to the global optimizer. How the factors are migrated from the local database to the global optimizer is also important (for example, by manual or automatic update, on demand, or on an event such as storage-space allocation).

☐ Is the optimization local, global, or both? Even if the optimizer performs global optimization in a distributed environment, it should still handle local queries and not depend on an access plan dictated by the global phase.

☐ Can the user determine a global optimizer's access to local statistics? For example, can the propagation of statistics from various nodes be managed so that they occur manually, on demand, or on schedule? Remember, global optimization requires information not only on where data resides and how best to route the necessary data to the user, but also about how to prepare that data for distributed access in the first place.

☐ Can the optimizer evaluate parallel I/O? If the relational DBMS supports fragmentation and replication, the optimizer might be able to use multiple disk drives and controllers to process the results in parallel. Both Tandem Non-Stop SQL and Teradata's Teradata DBC/1012 can access data this way.

☐ Can the optimizer take advantage of parallel processing? In a distributed system, the portions of a query that are processed in parallel and those that must be processed sequentially can significantly affect processing cost. If the optimizer doesn't model the system properly, parallel processing can be more costly than sequential processing on a uniprocessor machine.

☐ Does the optimizer compute the cost of semijoins? The semijoin is an effective means of joining distributed data when network costs are significant. Semijoins should not have the same cost evaluation as joins.

THE INGRES OPTIMIZER

As noted earlier, the Ingres optimizer uses a number of the techniques presented here. Considerable effort has gone into ensuring that the query tree is represented in canonical form and that SQL is flattened. This is not as simple as it might appear. The point at which the optimizer decides to remove duplicates can introduce an order dependence on operations, so going to a flattened or normal form of the query can affect the query result.[1,2,4] According to Ingres, all such flattening issues have been resolved in the current release.

Ingres attempts to reduce a query to irreducible, simpler queries by decomposition (not to be confused with the less sophisticated optimization-by-decomposition method used by the University of California, Berkeley, version of Ingres[10]). It does this by using tuple substitution and detachment before the access methods are selected and the search space is defined.

Ingres uses various techniques to limit the primitive operations and access methods used to define possible access plans; these include eliminating redundant nodes, recognizing redundant subtrees in the search space, and eliminating plans involving subtrees that (heuristically) are estimated to be too costly. For example, if a plan involves a Cartesian product of very large tables and alternate plans are available that do not, these can generally be eliminated; if a relation contains fewer than five pages, Ingres will select a relation scan rather than use any primary or secondary indexes.

A set of heuristics constrains the kinds of plans generated based on the operations in the query tree and the indexes available. Among the access methods considered are indirect join, semijoin, full and partial sort-merge joins, index join, hashed join, Cartesian product, projection-restriction, primary key lookup, secondary index lookup, subquery join, check-only join, relation scan, sort and ISAM, hash, or B-tree lookup. Whether or not journaling is in effect on the tables, the cost function takes into account the storage structure being used (B-tree, ISAM, hashed, or heap), disk I/O, CPU usage, and a CPU merge factor that can differ from CPU to CPU. These help the optimizer decide between sort-merge and lookup.

The time spent evaluating the search space is controlled so that no further search will occur if the optimizer "believes" the plan it has found would take less time to execute than the time it has taken to look for the plan. The user may also turn off this feature and enable an exhaustive search.

Many statistics are available to the optimizer for computing cost functions. The statistics are collected by running Optimizedb. This utility maintains information on data values such as minimum, maximum, and average as well as number of distinct values, rows, disk pages, rows per page, and so on. It also allows the user to establish, via a histogram, the distribution of values.

The histogram consists of partitions, each with a width determined by the range of values. The height of each partition is simply the number of rows containing values in the range. The user may set both the number of partitions and the width before collecting the statistics. In the current release, Ingres can collect this information by sampling the data in a table rather than scanning it. This can significantly reduce the cost of collecting statistics on deep tables.

The Ingres optimizer does not create indexes on intermediate results, nor does it create a histogram. It does follow certain heuristic rules that allow it to make statistical assumptions about intermediate results given the input tables and the operations involved. Ingres also does not fully optimize UNION. However, its optimization of OR compares favorably to most other optimizers.

Recent improvements to the optimizer include removing unnecessary nodes in processing sort-merge joins, optimizing functions, adding multiple-attribute joins, and taking into account improvements to the buffer manager.

Ingres refers to its product as an "AI optimizer" because of its extensive use of heuristics and statistics. It is, in a sense, an expert system. Its recognition of optimized subtrees might be considered a form of pattern recognition. However, it is not an artificial-intelligence system in the sense that it learns or creates new rules based on an inference engine.

All the optimal subtrees and

plans produced are cached using a least-recently-used algorithm. They may be kicked out of the cache if they no longer constitute a valid plan. For example, dropping an index might well invalidate a plan in the cache. Database procedures are stored as a linked query execution plan and can be invalidated. Ordinarily, a partial plan analysis is performed to determine whether database changes or the parameters being passed might invalidate the plan. For example, passing a wildcard rather than a constant value into a Boolean comparison can mean that an index is no longer viable. In this case, the invalidation is temporary.

The Ingres optimizer is capable of performing global as well as local optimization. The task of global optimization is to determine on what node a partial plan will be executed. Therefore, network costs must be taken into account when the cost functions for accessing base tables or indexes are computed and when the results of an operation on one node are supplied as input to an operation on another node.

The Optimizedb utility is used to update statistics locally. Global optimization is performed by the Ingres Star node. The Star node obtains these statistics automatically from the local node on demand, caching them until Optimizedb is run again. The Star node sends partial SQL plans to the local nodes for processing, where they undergo local optimization.

RETURN ON INVESTMENT

The selection of a relational DBMS may in part be contingent on the optimizer. Balancing the overall capabilities of the optimizer, the need for performance, and other desirable features of the DBMS is a technical problem. And the more you know about how databases work, the more difficult DBMS selection can be. Still, the effort is worthwhile if it leads to a successful installation that provides a continuing return on investment over the years and doesn't frustrate users and developers. ■

The author would like to acknowledge the help of Dave Kellogg, Eric Lundblatt, and Ed Horst of Ingres Corp.

REFERENCES

1. Codd, E. F. "Fatal Flaws in SQL (Part 1)," *Datamation* 34(16): 45-48, Aug. 1988.
2. Codd, E. F. "Fatal Flaws in SQL (Part 2)," *Datamation* 34(17): 71-74, Sept. 1988.
3. Date, C. J. *An Introduction to Database Systems.* Reading, Mass.: Addison-Wesley, 1986.
4. Date, C. J. "Be Careful with SQL EXISTS!" *DATABASE PROGRAMMING & DESIGN* 2(9): 50-52, Sept. 1989.
5. Kellogg, D. *Understanding Query Optimizers.* Relational Technology Technical Report, July 1989.
6. Kim, W., D. Reiner, and D. Batory, eds. *Query Processing in Database Systems.* W. Berlin, W. Germany: Springer Verlag, 1985.
7. Kooi, R. "The Optimization of Queries in Relational Databases." Ph.D. dissertation, Case Western Reserve University, Sept. 1980.
8. Stonebraker, M., ed. *Reading in Database Systems.* San Mateo, Calif.: Morgan Kauffman Publishers, 1988.
9. Wiorkowski, G., and D. Kull. "The Optimizer: Invisible Hand of the DBMS," *DATABASE PROGRAMMING & DESIGN* 1(9): 26-33, Sept. 1988.
10. Wong, E., and K. Youssefi. "Decomposition—A Strategy for Query Processing," *ACM TODS* 1(3), Sept. 1976.

David McGoveran is president of Alternative Technologies in Santa Cruz, Calif., a consulting firm specializing in relational database applications.

"DB/EXPO promises to be the most important database event of the year."

Philip Chapnick, Editorial Director, *Database Programming and Design*

SEE THE LATEST ADVANCES IN DATABASE TECHNOLOGY, END USER TOOLS AND CONNECTIVITY PRODUCTS.

FEATURING:

- Mainframe, Mini, Micro & DBMSs
- Database Machines and Fault-Tolerant Computers
- Object Oriented Tools
- CASE Products and 4GLs
- Expert Systems and Natural Languages
- Gateways, Connectivity and Communications Products

FREE ADMISSION TO THE EXHIBITS!

More than 130,000 square feet of exhibit space, with over 250 exhibits, including: **Oracle, IBM, H-P, DEC, Ingres, Boole & Babbage, Goal Systems, Candle, Computer Associates, Gupta Technologies, On-Line Software, Knowledgeware, Bachman, Teradata, Platinum, Tandem, Sybase,** and many others.

OUTSTANDING TECHNICAL CONFERENCE.

Speakers and topics include:

- **Colin J. White, Conference Director**
  Database: Yesterday, Today & Tomorrow
- "What to Look for in a Relational DBMS"
  - C.J. Date, Codd & Date International
- "Database Trends: How will Relational Systems Become Object-Oriented?"
  - Dr. Michael Stonebraker, U.C. Berkeley
- "Using the IBM Information Repository"
  - Paul Winsberg, Consultant
- "Defining an Application Architecture for SAA."
  - Jeff Tash, Database Associates
- "Getting the Most Out of CASE"
  - Vaughan Merlyn, CASE Research
- "Getting the Most Out of DB2."
  - Prem Mehra, IBM

DBEXPO90

Yes, I want to attend. Please send me information and registration materials for DB/EXPO '90 to be held March 27-29, 1990 at the Moscone Center in San Francisco.

- ❑ Send me ________ Free Exhibit Passes.
- ❑ Please send me information on exhibiting.
- ❑ Please send me information on the conference sessions and details on early registration DISCOUNTS.

Name & Title ____________________

Company ____________________

Street ____________________

City, State, Zip ____________________

Telephone ____________ FAX ____________

Norm De Nardi Enterprises, 289 S, San Antonio Rd., # 204
Los Altos, CA 94022. (415) 941-8440 FAX (415) 941-2066
1-800 2DB EXPO outside CA

THE NATIONAL DATABASE EXPOSITION AND CONFERENCE

DATABASES, TOOLS AND CONNECTIVITY

MARCH 27-29, 1990 • MOSCONE CONVENTION CENTER • SAN FRANCISCO, CA

CIRCLE 19 ON READER SERVICE CARD