Optimizing SQL

THERE IS NO SET of rules for writing code that will take the best advantage of every query optimizer on every SQL product. The query optimizers depend on the underlying architecture and are simply too different for universal rules; however, we can make some general statements. Just remember that you have to test code. What would improve performance in one SQL implementation might have no effect in another or make the performance worse.

There are two kinds of optimizers: cost-based and rule-based. A rule-based optimizer (such as Oracle before version 7.0) looks at the syntax of the query and plans how to execute the query without considering the size of the tables or the statistical distribution of the data. It will parse a query and execute it in the order in which it was written, perhaps doing some reorganization of the query into an equivalent form using some syntax rules. Basically, it is no optimizer at all.

A cost-based optimizer looks at both the query and the statistical data about the database itself before deciding the best way to execute the query. These decisions involve whether to use indexes, whether to use hashing, which tables to bring into main storage, what sorting technique to use, and so forth. Most of the time (but not all!), it will make better decisions than a human programmer would have, simply because it has more information.

CA-Ingres has one of the best optimizers, which extensively reorders a query before executing it. It is one of the few products that can find most semantically identical queries and reduce them to the same internal form.

Rdb, a DEC product that now belongs to Oracle, uses a searching method taken from an AI (artificial intelligence) game-playing program to inspect the costs of several different approaches before making a decision. DB2 has a system table with a statistical profile of the base tables.

In short, no two products use exactly the same optimization techniques.

The fact that each SQL engine uses a different internal storage scheme and access methods for its data makes some optimizations nonportable. Likewise, some optimizations depend on the hardware configuration, and a technique that was excellent for one product on a single hardware configuration could be a disaster in another product, or on another hardware configuration with the same product.

33.1 Access Methods

For this discussion, let us assume that there are four basic methods of getting to data: table scans or sequential reads of all the rows in the table, access via some kind of index, hashing, and bit vector indexes.

33.1.1 Sequential Access

The table scan is a sequential read of all the data in the order in which it appears in physical storage, grabbing one page of memory at a time. Most databases do not physically remove deleted rows, so a table can use a lot of physical space and yet hold little data. Depending on just how dynamic the database is, you may want to run a utility program to reclaim storage and compress the database. Performance can improve suddenly and drastically after database reorganization.

33.1.2 Indexed Access

Indexed access returns one row at a time. The index is probably going to be a B-Tree of some sort, but it could be a hashed index, inverted file structures, or another format. Obviously, if you do not have an index on a table, then you cannot use indexed access on it.

An index can be clustered or unclustered. A clustered index has a table that is in sorted order in the physical storage. Obviously, there can

be only one clustered index on a table. Clustered indexes keep the table in sorted order, so a table scan will often produce results in that order. A clustered index will also tend to put duplicates of the indexed column values on the same page of physical memory, which may speed up aggregate functions. (A side note: "clustered" in this sense is a Sybase/SQL Server term; Oracle uses the same word to mean a single data page that contains matching rows from multiple tables.)

33.1.3 Hashed Indexes

Writing hashing functions is not easy. The idea is that, given input values, the hashing function will return a physical storage address. If two or more values have the same hash value ("hash clash" or "collision"), then they are put into the same "bucket" in the hash table, or they are run through a second hashing function.

If the index is on a unique column, the ideal situation is a "minimal perfect" hashing function—each value hashes to a unique physical storage address, and there are no empty spaces in the hash table. The next best situation for a unique column is a "perfect" hashing function—every value hashes to one physical storage address without collisions, but there are some empty spaces in the physical hash table storage.

A hashing function for a nonunique column should hash to a bucket small enough to fit into main storage. In the Teradata SQL engine, which is based on hashing, any row can be found in at most two probes, and 90% or more of the accesses require only one probe.

33.1.4 Bit Vector Indexes

The fact that a particular occurrence of an entity has a particular value for a particular attribute is represented as a single bit in a vector or array. Predicates are handled by doing Boolean bit operations on the arrays. These techniques are very fast for large amounts of data and are used by the Nucleus database engine from Sand Technology and Foxpro's Rushmore indexes.

33.2 Expressions and Unnested Queries

Despite the fact that this book is devoted to fancy queries and programming tricks, the truth is that most real work is done with very simple logic. The better the design of the database schema, the easier the queries will be to write.

Here are some tips for keeping your query as simple as possible. Like all general statements, these tips will not be valid for all products in all situations, but they are how the smart money bets. In fairness, most optimizers are smart enough to do many of these things internally today.

33.2.1 Use Simple Expressions

Where possible, avoid JOIN conditions in favor of simple search arguments, called SARGs in the jargon. For example, let's match up students with rides back to Atlanta from a student ride share database.

```
SELECT *
  FROM Students AS S1, Rides AS R1
 WHERE S1.town = R1.town
   AND S1.town = 'Atlanta';
```

Clearly, a little algebra shows you that this is true:

```
SELECT *
  FROM Students AS S1, Rides AS R1
 WHERE R1.town = 'Atlanta'
   AND S1.town = 'Atlanta';
```

However, the second version will guarantee that the two tables involved will be projected to the smallest size, then the CROSS JOIN will be done. Since each of these projections should be fairly small, the JOIN will not be expensive.

Assume that there are ten students out of one hundred going to Atlanta, and five out of one hundred people offering rides to Atlanta. If the JOIN is done first, you would have (100 * 100) = 10,000 rows in the CROSS JOIN to prune with the predicates. This is why no product does the CROSS JOIN first. Instead, many products would do the (S1.town = 'Atlanta') predicate first and get a working table of ten rows to JOIN to the Rides table, which would give us (10 * 100) = 1,000 rows for the CROSS JOIN to prune.

But in the second version, we would have a working table of ten students and another working table of five rides to CROSS JOIN, or merely (5 * 10) rows in the result set.

Another rule of thumb is that, when given a chain of ANDed predicates that test for constant values, the most restrictive ones should be put first. For example,

```
SELECT *
  FROM Students
 WHERE sex = 'female'
   AND grade = 'A';
```

That query will probably run slower than the following:

```
SELECT *
  FROM Students
 WHERE grade = 'A'
   AND sex = 'female';
```

because there are fewer 'A' students than number of female students. There are several ways that this query will be executed:

1. Assuming an index on grades, fetch a row from the Students table where grade = 'A'; if sex = 'female' then put it into the final results. The index on grades is called the driving index of the loop through the Students table.
2. Assuming an index on sex, fetch a row from the Students table where sex = 'female'; if grade = 'A' then put it into the final results. The index on sex is now the driving index of the loop through the Students table.
3. Assuming indexing on both, scan the index on sex and put pointers to the rows where sex = 'female' into results working file R1. Scan the index on grades and put pointers to the rows where grade = 'A' into results file R2. Sort and merge R1 and R2, keeping the pointers that appear twice. Use this result to fetch the rows into the final result.

If the hardware can support parallel access, this can be quite fast.

Another application of the same principle is a trick with predicates that involves two columns to force the choice of the index that will be used. Place the table with the smallest number of rows last in the FROM clause, and place the expression that uses that table first in the WHERE clause. For example, consider two tables, a larger one for orders and a smaller one that translates a code number into English, each with an index on the JOIN column:

```
SELECT *
  FROM Orders AS O1, Codes AS C1
 WHERE C1.code = O1.code;
```

This query will probably use a strategy of merging the index values. However, if you add a dummy expression, you can force a loop over the index on the smaller table. For example, assume that all the order type codes are greater than or equal to '00' in our code translation example, so that the first predicate of this query is always TRUE:

```
SELECT *
  FROM Orders AS O1, Codes AS C1
 WHERE O1.ordertype >= '00'
   AND C1.somecode = O1.ordertype;
```

The dummy predicate will force the SQL engine to use an index on Orders. This same trick can also be used to force the sorting in an ORDER BY clause of a cursor to be done with an index.

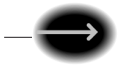
Since SQL is not a computational language, implementations do not tend to do even simple algebra:

```
SELECT *
  FROM Sales
 WHERE quantity = 500 + 1/2;
```

This query is the same thing as quantity = 500.50, but some dynamic SQLs will take a little extra time to compute and add a half as they check each row of the Sales table. The extra time adds up when the expression involves complex math and/or type conversions. However, this can have another effect that we will discuss in Section 33.8 on expressions that contain indexed columns.

The <> comparison has some unique problems. Most optimizers assume that this comparison will return more rows than it rejects, so they prefer a sequential scan and will not use an index on a column involved in such a comparison. This is not always true, however. For example, to find someone in Ireland who is not a Catholic, you would normally write:

```
SELECT *
  FROM Ireland
 WHERE religion <> 'Catholic';
```



The way around this is to break up the inequality and force the use of an index:

```
SELECT *
  FROM Ireland
 WHERE religion < 'Catholic'
    OR religion > 'Catholic';
```

However, without an index on religion, the ORed version of the predicate could take longer to run.

Another trick is to avoid the x IS NOT NULL predicate and use x >= <minimal constant> instead. The NULLs are kept in different ways in different implementations, but almost never in the same physical storage area as their columns. As a result, the SQL engine has to do extra searching. For example, if we have a CHAR(3) column that holds a NULL or three letters, we could look for missing data with:

```
SELECT *
  FROM Sales
 WHERE alphacode IS NOT NULL;
```

However, it would be better written as:

```
SELECT *
  FROM Sales
 WHERE alphacode >= 'AAA';
```

That syntax avoids the extra reads.

Another trick that often works is to use an index to get a COUNT(), since the index itself may have the number of rows already worked out. For example,

```
SELECT COUNT(*)
  FROM Sales;
```

might not be as fast as:

```
SELECT COUNT(invoice_nbr)
  FROM Sales;
```

where invoice_nbr is the PRIMARY KEY (or any other unique non-**NULL** column) of the Sales table. Being the PRIMARY KEY means that there is a unique index on invoice_nbr. A smart optimizer knows to look for indexed columns automatically when it sees a COUNT(*), but it is worth testing on your product.

33.2.2 String Expressions

Likewise, string expressions can be recalculated each time. A particular problem for strings is that the optimizer will often stop at the '%' or '_' in the pattern of a LIKE predicate, resulting in a string it cannot use with an index. For example, consider this table with a fixed length CHAR(5) column:

```
SELECT *
  FROM Students
 WHERE homeroom LIKE 'A-1__'; -- two underscores in pattern
```

This query may or may not use an index on the homeroom column. However, if we know that the last two positions are always numerals, we can replace this query with:

```
SELECT *
  FROM Students
 WHERE homeroom BETWEEN 'A-100' AND 'A-199';
```

This query can use an index on the homeroom column. Notice that this trick assumes that the homeroom column is CHAR(5), and not a VARCHAR(5) column. If it were VARCHAR(5), then the second query would pick 'A-1', while the original LIKE predicate would not. String equality and BETWEEN predicates pad the shorter string with blanks on the right before comparing them; the LIKE predicate does not pad either the string or the pattern.

33.3 Give Extra Join Information in Queries

Optimizers are not always able to draw conclusions that a human being can draw. The more information contained in the query, the better the chance that the optimizer will be able to find an improved execution plan. For example, to JOIN three tables together on a common column, you might write:

```
SELECT *
  FROM Table1, Table2, Table3
 WHERE Table2.common = Table3.common
   AND Table3.common = Table1.common;
```

Alternately, you might write:

```
SELECT *
  FROM Table1, Table2, Table3
 WHERE Table1.common = Table2.common
   AND Table1.common = Table3.common;
```

Some optimizers will JOIN pairs of tables based on the equi-JOIN conditions in the WHERE clause in the order in which they appear. Let us assume that Table1 is a very small table and that Table2 and Table3 are large. In the first query, doing the Table2–Table3 JOIN first will return a large result set, which is then pruned by the Table1–Table3 JOIN. In the second query, doing the Table1–Table2 JOIN first will return a small result set, which is then matched to the small Table1–Table3 JOIN result set.

The best bet, however, is to provide all the information so that the optimizer can decide when the table sizes change.

This leads to redundancy in the WHERE clause:

```
SELECT *
  FROM Table1, Table2, Table3
 WHERE Table1.common = Table2.common
   AND Table2.common = Table3.common
   AND Table3.common = Table1.common;
```

Do not confuse this redundancy with needless logical expressions that will be recalculated and can be expensive. For example,

```
SELECT *
  FROM Sales
 WHERE alphacode BETWEEN 'AAA' AND 'ZZZ'
   AND alphacode LIKE 'A_C';
```

will redo the BETWEEN predicate for every row. It does not provide any information that can be used for a JOIN, and, very clearly, if the LIKE predicate is TRUE, then the BETWEEN predicate also has to be TRUE.

A final tip, which is not always true, is to order the tables with the fewest rows in the result set last in the FROM clause. This is helpful because as the number of tables increases, many optimizers do not try all the combinations of possible JOIN orderings; the number of combinations is factorial. So the optimizer falls back on the order in the FROM clause.

33.4 Index Tables Carefully

You should create indexes on the tables of your database to optimize your query search time, but do not create any more indexes than are absolutely needed. Indexes have to be updated and possibly reorganized when you INSERT, UPDATE, or DELETE a row in a table.

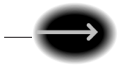
Too many indexes can result in extra time spent tending indexes that are seldom used. But even worse, the presence of an index can fool the optimizer into using it when it should not. For example, let's look at the following simple query:

```
SELECT *
  FROM Warehouse
 WHERE quantity = 500
   AND color = 'Purply Green';
```

With an index on color, but not on quantity, most optimizers will first search for rows with `color = 'Purply Green'` via the index, then apply the `quantity = 500` test. However, if you were to add an index on quantity, the optimizer would likely take the tests in order, doing the quantity test first. I assume that very few items are 'Purply Green', so it would have been better to test for color first. A smart optimizer with detailed statistics would do this right, but to play it safe, order the predicates from the most restricting (i.e., the smallest number of qualifying rows in the final result) to the least.

An index will not be used if the column is in an expression. If you want to avoid an index, then put the column in a "do nothing" expression, such as the following examples:

```
SELECT *
  FROM Warehouse
 WHERE quantity = 500 + 0
   AND color = 'Purply Green';
```



or

```
SELECT *
  FROM Warehouse
 WHERE quantity + 0 = 500
   AND color = 'Purply Green';
```

This will stop the optimizer from using an index on quantity. Likewise, the expression (`color || = 'Purply Green'`) will avoid the index on color.

Consider an actual example of indexes making trouble, in a database for a small club membership list that was indexed on the members' names as the `PRIMARY KEY`. There was a column in the table that had one of five status codes (paid member, free membership, expired, exchange newsletter, and miscellaneous).

The report query on the number of people by status was:

```
SELECT M1.status, C1.code_text, COUNT(*)
  FROM Members AS M1, Codes AS C1
 WHERE M1.status = C1.status
 GROUP BY M1.status, C1.code_text;
```

In an early PC SQL database product, it ran an order of magnitude slower with an index on the status column than without one. The optimizer saw the index on the Members table and used it to search for each status code text. Without the index, the much smaller Codes table was brought into main storage and five buckets were set up for the `COUNT(*)`; then the Members table was read once in sequence. An index used to ensure uniqueness on a column or set of columns is called a primary index; those used to speed up queries on nonunique column(s) are called secondary. SQL implementations automatically create a primary index on a `PRIMARY KEY` or `UNIQUE` constraint. Implementations may or may not create indexes that link `FOREIGN KEY`s within the table to their targets in the referenced table. This link can be very important, since a lot of `JOIN`s are done from `FOREIGN KEY` to `PRIMARY KEY`.

You also need to know something about the queries to run against the schema. Obviously, if all queries are asked on only one column, then that is all you need to index. The query information is usually given as a statistical model of the expected inputs. For example, you might be told

that 80% of the queries will use the PRIMARY KEY and 20% will use another (near-random) column.

This is pretty much what you would know in a real-world situation, since most of the accessing will be done by production programs with embedded SQL in them; only a small percentage will be *ad hoc* queries.

Without giving you a computer science lecture, a computer problem is called NP-complete if it gets so big, so fast, that it is not practical to solve it for a reasonable-sized set of input values.

Usually this means that you have to try all possible combinations to find the answer. Finding the optimal indexing arrangement is known to be NP-complete (Comer 1978; Paitetsky-Shapiro 1983). This does not mean that you cannot optimize indexing for a particular database schema and set of input queries, but it does mean that you cannot write a program that will do it for all possible relational databases and query sets.

33.5 Watch the IN Predicate

The IN predicate is really shorthand for a series of ORed equality tests. There are two forms: either an explicit list of values is given, or a subquery is used to make such a list of values.

The database engine has no statistics about the relative frequency of the values in a list of constants, so it will assume that the list is in the order in which the values are to be used. People like to order lists alphabetically or by magnitude, but it would be better to order the list from most frequently occurring values to least frequently occurring. It is also pointless to have duplicate values in the constant list, since the predicate will return TRUE if it matches the first duplicate it finds and will never get to the second occurrence. Likewise, if the predicate is FALSE for that value, the program wastes computer time traversing a needlessly long list.

Many SQL engines perform an IN predicate with a subquery by building the result set of the subquery first as a temporary working table, then scanning that result table from left to right. This can be expensive in many cases. For example, the following query:

```
SELECT P1.*
  FROM Personnel AS P1, BowlingTeam AS B1
 WHERE P1.last_name IN (SELECT last_name
                          FROM BowlingTeam AS B1
                         WHERE P1.emp_nbr = B1.emp_nbr)
   AND P1.last_name IN (SELECT last_name
```

```
                          FROM BowlingTeam AS B2
                         WHERE P1.emp_nbr = B2.emp_nbr);
```

will not run as fast as:

```
SELECT *
  FROM Personnel AS P1
 WHERE first_name || last_name IN
       (SELECT first_name || last_name
          FROM BowlingTeam AS B1
         WHERE P1.emp_nbr = B1.emp_nbr);
```

which can be further simplified to:

```
SELECT P1.*
  FROM Personnel AS P1
 WHERE first_name || last_name IN
       (SELECT first_name || last_name
          FROM BowlingTeam);
```

or, using Standard SQL row constructors, can be simplified to:

```
SELECT P1.*
  FROM Personnel AS P1
 WHERE (first_name, last_name) IN
       (SELECT first_name, last_name
          FROM BowlingTeam);
```

since there can be only one row with a complete name in it.

The first version of the query may make two passes through the Bowling Team table to construct two separate result tables. The second version makes only one pass to construct the concatenation of the names in its result table.

The optimizer is supposed to figure out when two queries are the same, and it will not be fooled by two queries with the same meaning and different syntax. For example, the SQL standard defines the following two queries as identical:

```
SELECT *
  FROM Warehouse AS W1
 WHERE quantity IN (SELECT quantity FROM Sales);
```

```
SELECT *
  FROM Warehouse
 WHERE quantity = ANY (SELECT quantity FROM Sales);
```

However, you will find that some older SQL engines prefer the first version to the second, because they do not convert the expressions into a common internal form. Very often, things like the choice of operators and their order make a large performance difference.

The first query can be converted to this "flattened" JOIN query:

```
SELECT W1.*
  FROM Warehouse AS W1, Sales AS S1
 WHERE W1.qty_on_hand = S1.qty_sold;
```

This form will often be faster if there are indexes to help with the JOIN.

33.6 Avoid UNIONs

A UNION is often implemented by constructing the two result sets, then merge-sorting them together. The optimizer works only within a single SELECT statement or subquery. For example:

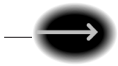
```
SELECT *
  FROM Personnel
 WHERE work = 'New York'
UNION
SELECT *
  FROM Personnel
 WHERE home = 'Chicago';
```

is the same as:

```
SELECT DISTINCT *
  FROM Personnel
 WHERE work = 'New York'
    OR home = 'Chicago';
```

The second will run faster.

Another trick is to use UNION ALL in place of UNION whenever duplicates are not a problem. The UNION ALL is implemented as an append operation, without the need for a sort to aid duplicate removal.



33.7 Prefer Joins over Nested Queries

A nested query is hard to optimize. Optimizers try to "flatten" nested queries so they can be expressed as JOINs and the best order of execution can be determined. Consider the database:

```
CREATE TABLE Authors
(author_nbr INTEGER NOT NULL PRIMARY KEY,
 authorname CHAR(50) NOT NULL);

CREATE TABLE Titles
(isbn CHAR(10)NOT NULL PRIMARY KEY,
 title CHAR(50) NOT NULL
 advance_amt DECIMAL(8,2) NOT NULL);

CREATE TABLE TitleAuthors
(author_nbr INTEGER NOT NULL REFERENCES Authors(author_nbr),
 isbn CHAR(10)NOT NULL REFERENCES Titles(isbn),
 royalty_rate DECIMAL(5,4) NOT NULL,
 PRIMARY KEY (author_nbr, isbn));
```

This query finds authors who are getting less than 50% royalties:

```
SELECT author_nbr
  FROM Authors
 WHERE author_nbr
       IN (SELECT author_nbr
             FROM TitleAuthors
             WHERE royalty < 0.50)
```

This query could also be expressed as:

```
SELECT DISTINCT Authors.author_nbr
  FROM Authors, TitleAuthors
 WHERE (Authors.author_nbr = TitleAuthors.author_nbr)
   AND (royalty_rate < 0.50);
```

The SELECT DISTINCT is important. Each author's name will occur only once in the Authors table. Therefore, the IN predicate query should return one occurrence of O'Leary. Assume that O'Leary wrote two books;

with just a SELECT, the second query would return two O'Leary rows, one for each book.

33.8 Avoid Expressions on Indexed Columns

If a column appears in a mathematical or string expression, then the optimizer cannot use its indexes. For example, given a table of tasks and their start and finish dates, to find the tasks that took three days to complete in 1994 we could write:

```
SELECT task_nbr
  FROM Tasks
 WHERE (finish_date - start_date) = INTERVAL '3' DAY
   AND start_date >= CAST ('2005-01-01' AS DATE);
```

But since most of the reports deal with the finish dates, we have an index on that column. This means that the query will run faster if it is rewritten as:

```
SELECT task_nbr
  FROM Tasks
 WHERE finish_date = (start_date + INTERVAL '3' DAY)
  AND start_date >= ('2005-01-01' AS DATE);
```

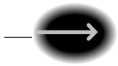
This same principle applies to columns in string functions and, very often, to LIKE predicates.

However, this can be a good thing for queries with small tables, since it will force those tables to be loaded into main storage instead of being searched by index.

33.9 Avoid Sorting

The SELECT DISTINCT and ORDER BY clauses usually cause a sort in most SQL products, so avoid them unless you really need them. Use them if you need to remove duplicates or if you need to guarantee a particular result set order explicitly. In the case of a small result set, the time to sort it can be longer than the time to process redundant duplicates.

The UNION, INTERSECT, and EXCEPT clauses can do sorts to remove duplicates; the exception is when an index exists that can be used to eliminate the duplicates without sorting. In particular, the UNION ALL will tend to be faster than the plain UNION, so if you have no duplicates



or do not mind having them, then use it instead. There are not enough implementations of INTERSECT ALL and EXCEPT ALL to make a generalization yet.

The GROUP BY often uses a sort to cluster groups together, does the aggregate functions, and then reduces each group to a single row based on duplicates in the grouping columns. Each sort will cost you (n*log2(n)) operations. That is a lot of extra computer time that you can save if you do not need to use these clauses.

If a SELECT DISTINCT clause includes a set of key columns in it, then all the rows are already known to be unique. Since you can declare a set of columns to be a PRIMARY KEY in the table declaration, an optimizer can spot such a query and automatically change SELECT DISTINCT to just SELECT.

You can often replace a SELECT DISTINCT clause with an EXIST() subquery, in violation of another rule of thumb that says to prefer unnested queries to nested queries. For example, a query to find the students who are majoring in the sciences would be:

```
SELECT DISTINCT S1.name
  FROM Students AS S1, ScienceDepts AS D1
 WHERE S1.dept = D1.dept;
```

This query can be better replaced with:

```
SELECT S1.name
  FROM Students AS S1
 WHERE EXISTS
       (SELECT *
          FROM ScienceDepts AS D1
         WHERE S1.dept = D1.dept);
```

Another problem is that the DBA might not declare all candidate keys or might declare superkeys instead. Consider a table for a school schedule:

```
CREATE TABLE Schedule
(room_nbr INTEGER NOT NULL,
 course_name CHAR(7) NOT NULL,
 teacher_name CHAR(20) NOT NULL,
 period_nbr INTEGER NOT NULL,
 PRIMARY KEY (room_nbr, period_nbr));
```

This says that if I know the room and the period, I can find a unique teacher and course—"Third-period Freshman English in Room 101 is taught by Ms. Jones." However, I might have also added the constraint `UNIQUE (teacher, period)`, since Ms. Jones can be in only one room and teach only one class during a given period. If the table was not declared with this extra constraint, the optimizer could not use it in parsing a query. Likewise, if the DBA decided to declare `PRIMARY KEY (room_nbr, course_name, teacher_name, period_nbr)`, the optimizer could not break down this superkey into candidate keys.

Avoid using a `HAVING` or a `GROUP BY` clause if the `SELECT` or `WHERE` clause can do all the needed work. One way to avoid grouping is in situations where you know the group criterion in advance and then make it a constant. This example is a bit extreme, but you can convert:

```
SELECT project, AVG(cost)
  FROM Tasks
 GROUP BY project
HAVING project = 'bricklaying';
```

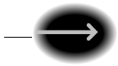
to the simpler and faster:

```
SELECT 'bricklaying', AVG(cost)
  FROM Tasks
 WHERE project = 'bricklaying';
```

Both queries have to scan the entire table to inspect values in the project column. The first query will simply throw each row into a bucket based on its project code, then look at the `HAVING` clause to throw away all but one of the buckets before computing the average. The second query rejects those unneeded rows and arrives at one subset of projects when it scans.

Standard SQL has ways of removing `GROUP BY` clauses, because it can use a subquery in a `SELECT` statement. This is easier to show with an example in which you are now in charge of the Widget-Only Company inventory. You get requisitions that tell how many widgets people are putting into or taking out of the warehouse on a given date. Sometimes that quantity is positive (returns); sometimes it is negative (withdrawals).

The table of requisitions looks like this:



```
CREATE TABLE Requisitions
(req_date DATE NOT NULL,
 rteq_qty INTEGER NOT NULL
     CONSTRAINT non_zero_qty
     CHECK (req_qty <> 0));
```

Your job is to provide a running balance on the quantity on hand with a query. We want something like:

```
RESULT
 req_date   req_qty   qty_on_hand
==============================
'2005-07-01'    100    100
'2005-07-02'    120    220
'2005-07-03'   -150     70
'2005-07-04'     50    120
'2005-07-05'    -35     85
```

The classic SQL solution would be:

```
SELECT R1.reqdate, R1.qty, SUM(R2.qty) AS qty_on_hand
  FROM Requisitions AS R1, Requisitions AS R2
 WHERE R2.reqdate <= R1.reqdate
 GROUP BY R1.reqdate, R1.qty;
```

Standard SQL can use a subquery in the SELECT list, even a correlated query. The rule is that the result must be a single value, hence the name scalar subquery; if the query results are an empty table, the result is a NULL.

In this problem, we need to do a summation of all the requisitions posted up to and including the date we are looking at. The query is a nested self-JOIN, thus:

```
SELECT R1.reqdate, R1.qty,
      (SELECT SUM(R2.qty)
         FROM Requisitions AS R2
        WHERE R2.reqdate <= R1.reqdate) AS qty_on_hand
  FROM Requisitions AS R1;
```

Frankly, both solutions are going to run slowly compared to a procedural solution that could build the current quantity on hand from

the previous quantity on hand, using a sorted file of records. Both queries will have to build the subquery from the self-joined table based on dates. However, the first query will also probably sort rows for each group it has to build. The earliest date will have one row to sort, the second earliest date will have two rows, and so forth until the most recent date will sort all the rows. The second query has no grouping, so it just proceeds to the summation without the sorting.

33.10 Avoid CROSS JOINs

Consider a three-table JOIN like this.

```
SELECT P1.paint_color
  FROM Paints AS P1, Warehouse AS W1, Sales AS S1
 WHERE W1.qty_on_hand + S1.qty_sold =
       P1.gallons/2.5;
```

Because all of the columns involved in the JOIN are in a single expression, their indexes cannot be used. The SQL engine will construct the CROSS JOIN of all three tables first and then prune that temporary working table to get the final answer. In Standard SQL, you can first do a subquery with a CROSS JOIN to get one side of the equation:

```
(SELECT (W1.qty_on_hand + S1.qty_sold) AS stuff
  FROM Warehouse AS W1 CROSS JOIN Sales AS S1)
```

and then push it into the WHERE clause, like this:

```
SELECT color
  FROM Paints AS P1
 WHERE EXISTS ((SELECT (W1.qty_on_hand + S1.qty_sold)
                  FROM Warehouse AS W1 CROSS JOIN Sales AS S1)
            = (P1.gallons/2.5));
```

The SQL engine, we hope, will do the two-table CROSS JOIN subquery and put the results into a temporary table. That temporary table will then be filtered using the Paints table, but without generating a three-table CROSS JOIN as the first form of the query did.

With a little algebra, the original equation can be changed around and different versions of this query built with other combinations of tables.

A good rule of thumb is that the FROM clause should only have those tables that provide columns to its matching SELECT clause.

33.11 Learn to Use Indexes Carefully

By way of review, most indexes are tree structures. They consist of a page or node that has values from the columns of the table from which the index is built, and pointers. The pointers point to other nodes of the tree and eventually point to rows in the table that has been indexed. The idea is that searching the index is much faster than searching the table itself in a sequential fashion (called a table scan).

The index is also ordered on the columns used to construct it; the rows of the table may or may not be in that order. When the index and the table are sorted on the same columns, the index is called a clustered index. The best example of this in the physical world is a large dictionary with a thumb-notch index—the index and the words in the dictionary are both in alphabetical order.

For obvious physical reasons, you can use only one clustered index on a table. The decision as to which columns to use in the index can be important to performance. There is a superstition among older DBAs who have worked with ISAM files and network and hierarchical databases that the primary key must be done with a clustered index. This stems from the fact that in the older file systems, files had to be sorted or hashed on their keys. All searching and navigation was based on this.

This is not true in SQL systems. The primary key's uniqueness will probably be preserved by a unique index, but it does not have to be a clustered unique index. Consider a table of employees keyed by a unique employee identification number. Updates are done with the employee ID number, of course, but very few queries use it. Updating individual rows in a table will actually be about as fast with a clustered or a nonclustered index. Both tree structures will be the same, except for the final physical position to which they point.

However, it might be that the most important corporate unit for reporting purposes is the department, not the employee. A clustered index on the employee ID number would sort the table in employee-ID order. There is no inherent meaning in that ordering; in fact, I would be more likely to sort a list of employees by their last names than by their ID numbers.

However, a clustered index on the (nonunique) department code would sort the table in department order and put employees in the same

department on the same physical page of storage. The result would be that fewer pages would be read to answer queries.

33.12 Order Indexes Carefully

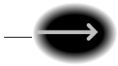
Consider the Personnel table again. There may be a difference among these CREATE INDEX statements:

```
1.  CREATE INDEX XDeptDiv
    ON Personnel (dept, division);

2.  CREATE INDEX XDivDept
    ON Personnel (division, dept);

3.  CREATE CLUSTERED INDEX XCDeptDiv
    ON Personnel (dept, division);

4.  CREATE CLUSTERED INDEX XCDivDept
    ON Personnel (division, dept);
```

In cases 1 and 2, some products build an index only on the first column and ignore the second column. This is because their SQL engine is based on an older product that allows only single-column indexing, and the parser is throwing out the columns it cannot handle. Other products use the first column to build the index tree structure and the secondary columns in such a way that they are searched much more slowly. Both types of SQL engine are like an alphabetic accordion file. Each pocket of an accordion file locates a letter of the alphabet, but within each pocket you have to do a manual search for a particular paper.

If your implementation suffers from this problem, the best thing to do is to order the columns by their granularity; that is, put the column with the most values first and the column with the fewest values last. In our example, assume that we have a few divisions located in major cities, and within each division we have lots of departments. An indexed search that stops at the division will leave us with a scan over the many departments. An indexed search that stops at the department will leave us with a scan over the few divisions.

Most SQL products will concatenate the columns listed in the CREATE INDEX statement. But you have to know in what order they will be concatenated. A telephone book is order by (last_name, first_name), so it is easy to look up someone if you know his or her last name. If you



only know the first name, you have to read the whole darn telephone book until you find the right person. You can spot this problem when you have a slow query on a column with an index on it that is using a table scan.

If you are lucky, or have planned things carefully, you can get a covering index for your query. This means that the index has all of the columns needed to answer the query, so the base table upon which it is built never has to be accessed.

In some products, you may find that the ordering will not matter, or that separate nonunique indexes will do as well as or better than a unique compound index. The reason is that they use hashing or bit-map indexes. Foxpro and Nucleus are two examples of products that use different bit-map schemes, but they have some basic features in common. Imagine an array with table row numbers or pointers on its columns, and values for that column on its rows. If a table row has that value in that position, then the bit is set; if not, the bit is zeroed. A search is done by doing bitwise ANDs, ORs, and NOTs on the bit vectors.

This might be easier to explain with an example of the technique. Assume we have a table of Parts, which has columns for the attributes color and weight.

```
Parts
part_nbr part_name  color  weight city
=======================================
'p1'     'Nut'     'Red'    12   'London  -- Physical row # 3
'p2      'Bolt'    'Green'  17   'Paris'  -- Physical row # 4
'p3'     'Cam'     'Blue'   12   'Paris'  -- Physical row # 7
'p4'     'Screw'   'Red'    14   'London' -- Physical row # 9
'p5'     'Cam'     'Blue'   12   'Paris'  -- Physical row # 11
'p6'     'Cog'     'Red'    19   'London' -- Physical row # 10
```

The bit indexes are built by using the physical row and the values of the attributes in an array, thus:

```
INDEX Parts(color)
 Rows   1 2 3 4 5 6 7 8 9   10 11
      ========================
Blue  | 0 0 0 0 0 0 1 0 0   0  1
Green | 0 0 0 1 0 0 0 0 0   0  0
Red   | 0 0 1 0 0 0 0 0 1   1  0
```

```
INDEX Parts(weight)
 Rows  1 2 3 4 5 6 7 8 9 10 11
      ========================
  12  | 0 0 1 0 0 0 1 0 0  0  1
  17  | 0 0 0 1 0 0 0 0 0  0  0
  14  | 0 0 0 0 0 0 0 0 1  0  0
  19  | 0 0 0 0 0 0 0 0 0  1  0
```

To find a part that weighs 12 units and is colored red, you would perform a bitwise AND, and get a new bit vector as the answer:

```
Red    | 0 0 1 0 0 0 0 0 1 1  0
AND
  12   | 0 0 1 0 0 0 1 0 0  0  1
       ========================
answer| 0 0 1 0 0 0 0 0 0  0  0
```

To find a part that weighs 12 units or is colored red, you would perform a bitwise OR, and get a new bit vector as the answer:

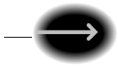
```
Red    | 0 0 1 0 0 0 0 0 1 1 0
OR
  12   | 0 0 1 0 0 0 1 0 0 0 1
       ======================
answer| 0 0 1 0 0 0 1 0 1 1 0
```

Searches become a combination of bitwise operators on the indexes before any physical access to the table is done.

33.13 Know Your Optimizer

One of the best tricks is to know what your optimizer favors. It is often the case that one query construction will have special code written for it that an equivalent query construction does not. Consider this simple adjacency list model of a tree:

```
CREATE TABLE Tree
(node_id CHAR(2) NOT NULL,
 parent_node_id CHAR(2), -- null is root node
 creation_date DATE DEFAULT CURRENT_TIMESTAMP NOT NULL);
```



Let's try to group all the parent_node_ids and display the creation_date of the most recent subordinate under that parent_node_id. This is a straightforward query that can be done with the following statement:

```
SELECT node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
 WHERE NOT EXSTS
      (SELECT *
         FROM Tree AS T2
        WHERE T2.parent_node_id = T1.parent_node_id
          AND T2.creation_date > T1.creation_date);
```

The `EXISTS()` predicate says that there is no sibling younger than the one we picked. Or, as an alternative, you can use an `OUTER JOIN` to do the same logic.

```
SELECT T1.node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
       LEFT OUTER JOIN
       Tree AS T2
       ON T2.parent_node_id = T1.parent_node_id
         AND T2.creation_date > T1.creation_date
 WHERE T2.node_id IS NULL;
```

However, this query was run in SQL Server 2000. Lee Tudor pointed out that SQL Server looks for a join to an aggregated self-view on the group condition and aggregate. Performance is far superior to the alternates, due to the query optimizer having a special way of dealing with it.

```
SELECT T1.node_id, T1.parent_node_id, T1.creation_date
  FROM Tree AS T1
       INNER JOIN
       (SELECT parent_node_id, MAX(creation_date)
          FROM Tree
         GROUP BY parent_node_id)
       AS T2 (parent_node_id, creation_date)
       ON T2.parent_node_id = T1.parent_node_id
          AND T2.creation_date = T1.creation_date;
```

The optimizer will change from release to release, but it is a good general statement that once a trick is coded into it, the trick will stay there until there is a major change in the product.

In 1988, Fabian Pascal published an article in *Database Programming and Design* on the PC database systems available at the time (Pascal 1998), in which he wrote seven logically equivalent queries as a test suite. These tests revealed vastly uneven performance for all of the RDBMS products except Ingres. Ingres's timings were approximately the same for all seven queries, and Ingres had the best average time. The other products showed wide variations across the seven queries, with the worst timing more than an order of magnitude longer than the best. In the case of Oracle, the worst timing was more than 600 times the best.

While optimizers have gotten better over the years, there are still "hot spots" in specific optimizers that favor particular constructions.

33.14 Recompile Static SQL after Schema Changes

In most implementations, static SQL is compiled in a host program with a fixed execution plan. If a database schema object is altered, execution plans based on that object have to be changed.

In older SQL implementations, if a schema object was dropped, the programmer had to recompile the queries that referred to it. The SQL engine was not required to do any checking, and most implementations did not. Instead, you could get a runtime error.

Even worse, you could have a scenario like this:

1. Create table A
2. Create view VA on table A
3. Use view VA
4. Drop table A
5. Create a new table A
6. Use view VA

What happens in step 6? That depended on your SQL product, but the results were not good. The worst result was that the entire schema could be hopelessly messed up. The best result was that the VA in step 3 was not the VA in step 6, but was still usable.

Standard SQL added the option of specifying the behavior of any of the `DROP` statements as either `CASCADE` or `RESTRICT`. The `RESTRICT`

option is the default in Standard SQL, and it will disallow the dropping of any schema object that is being used to define another object. For example, if RESTRICT is used, you cannot drop a base table that has VIEWs defined on it or is part of a referential integrity constraint. The CASCADE option will drop any of the dependent objects from the schema when the original object is removed.

Be careful with this! The X/Open transaction model assumes a default action of CASCADE, and some products may allow it to be set as a system parameter. The moral to the story is that you should never use an implicit default on statements that can destroy your schema when an explicit clause is available.

Some products will automatically recompile static SQL when an index is dropped; some will not. However, few products automatically recompile static SQL when an index is added. Furthermore, few products automatically recompile static SQL when the statistical distribution within the data has changed. Oracle's Rdb is an exception to this, since it investigates possible execution paths when each query is invoked.

The DBA usually has to update the statistical information explicitly and ask for a recompilation. What usually happens is that one person adds an index and then compiles his program. The new index could either hinder or help other queries when they are recompiled, so it is hard to say whether the new index is a good or a bad thing for the overall performance of the system.

However, it is always bad is when two programmers build indexes that are identical in all but name and never tell each other. Most SQL implementations will not detect this. The duplication will waste both time and space. Whenever one index is updated, the other one will have to be updated also. This is one reason that only the DBA should be allowed to create schema objects.

33.15 Temporary Tables Are Sometimes Handy

Another trick is to use temporary tables to hold intermediate results to avoid CROSS JOINs and excessive recalculations. A materialized VIEW is also a form of temporary table, but you cannot index it. In this problem, we want to find the total amount of the latest balances in all our accounts.

Assume that the Payments table holds the details of each payment and that the payment numbers are increasing over time. The Accounts

table shows the account identification number and the balance after each payment is made. The query might be done like this:

```
SELECT SUM(A1.balance)
  FROM Accounts AS A1, Payments AS P1
 WHERE P1.acct_nbr = A1.acct_nbr
       AND P1.payment_nbr = (SELECT MAX(payment_nbr)
                               FROM Payments AS P2
                              WHERE P2.acct_nbr = A1.acct_nbr);
```

Since this uses a correlated subquery with an aggregate function, it will take a little time to run for each row in the answer. It would be faster to create a temporary working table or `VIEW` like this:

```
BEGIN
CREATE TABLE LastPayments
(acct INTEGER NOT NULL,
 last_payment_nbr INTEGER NOT NULL,
 payment_amt DECIMAL(8,2) NOT NULL,
 payment_date DATE NOT NULL,
 ...  );

CREATE INDEX LPX ON LastPayment(acct, payment_nbr);

INSERT INTO LastPayments
SELECT acct, payment_nbr, MAX(payment_nbr)
  FROM Payments
 GROUP BY acct, payment_nbr;

SELECT SUM(A1.balance) -- final answer
  FROM Accounts AS A1, LastPayments AS LP1
 WHERE LP1.acct_nbr = A1.acct_nbr
   AND LP1.payment_nbr = A1.payment_nbr;

DROP TABLE LastPayments;

END;
```

Consider this three-table query that creates a list of combinations of items and all the different packages for which the selling price (price and

box cost) is 10% of the warranty plan cost. Assume that any item can fit into any box we have and that any item can be put on any warranty plan.

```
SELECT I1.item
  FROM Inventory AS I1, Packages AS P1, Warranty AS W1
 WHERE I1.price + P1.box = W1.plancost * 10;
```

Since all the columns appear in an expression, the engine cannot use indexes, so the query will become a large CROSS JOIN in most SQL implementations. This query can be broken down into a temporary table that has an index on the calculations, thus:

```
BEGIN
CREATE TABLE SellingPrices
(item_name CHAR (15) NOT NULL,
 sell_price DECIMAL (8,2) NOT NULL);

-- optional index on the calculation
CREATE INDEX SPX ON SellingPrices (sell_price);

-- do algebra and get everything on one side of an equation
INSERT INTO SellingPrices (item_name, sell_price)
SELECT DISTINCT I1.item_name, (P1.box + I1.price) * 0.1
  FROM Inventory AS I1, packages AS P1;

-- do the last JOIN
SELECT DISTINCT SP1.item_name
  FROM SellingPrices AS SP1, Warranty AS W1
 WHERE SP1.sell_price = W1.plancost;
END;
```

The Sybase/SQL Server family allows a programmer to create temporary tables on the fly. This is a totally different model of temporary table than the Standard SQL model. The standard model does not allow a user to create any schema objects.

In the Sybase/SQL Server model such *ad hoc* creations have no indexes or constraints, and therefore act like "scratch paper" with its own name. However, they do not give much help to the optimizer. This is a holdover from the days when we allocated scratch tapes in file systems to hold the results from one step to another in a procedural process.

You can often use derived tables in the query in place of these temporary tables. The derived table definition is put into the parse tree, and the execution plan can take advantage of constraints, indexing, and self-joins.

33.16 Update Statistics

This is going to sound obvious, but the optimizer cannot work without valid statistics. Some signs that your statistics need to be updated are:

1. *Two queries with the same basic structure have different execution times*. This usually means that the statistical distribution has changed, so one of the queries is being executed under old assumptions.
2. *You have just loaded a lot of new data*. This is often a good time to do the update to the statistics, because some products can get them as part of the loading operation.
3. *You have just deleted a lot of old data*. Unfortunately, deletion operations do not change statistics like insertions.
4. *You have changed the query mix*. For example, the end-of-the-month reports depend on shared views to aggregate data. In the earlier SQL products, you could put these `VIEW`s into temporary tables and index those tables. This would allow the optimizer to gather statistics. Today, the better SQL products will do the same job under the covers by materializing these `VIEW`s. You need to find out if you need to compute indexes and/or do the statistics.