


*System thinking and experience, not just technical expertise, are what the relational user needs to tune complex database applications*

# Secrets of Relational Performance Tuning

**T**HE ABILITY TO tune a relational database application for performance is a fine art. If you want throughput, you must be careful to distinguish between tuning the database for performance and tuning the entire system for throughput. The two are not always compatible. In either case, relational performance tuning takes getting used to.

The problem is obvious when you realize that relational databases should protect the user from knowledge of how it handles the requests; requests may come not only from database users but also from production programs and database administrators. A DBMS true to the relational promise should always respond to requests efficiently. The system would create and use indexes as needed and disk allocation would be optimized to minimize storage consumption and access time si-

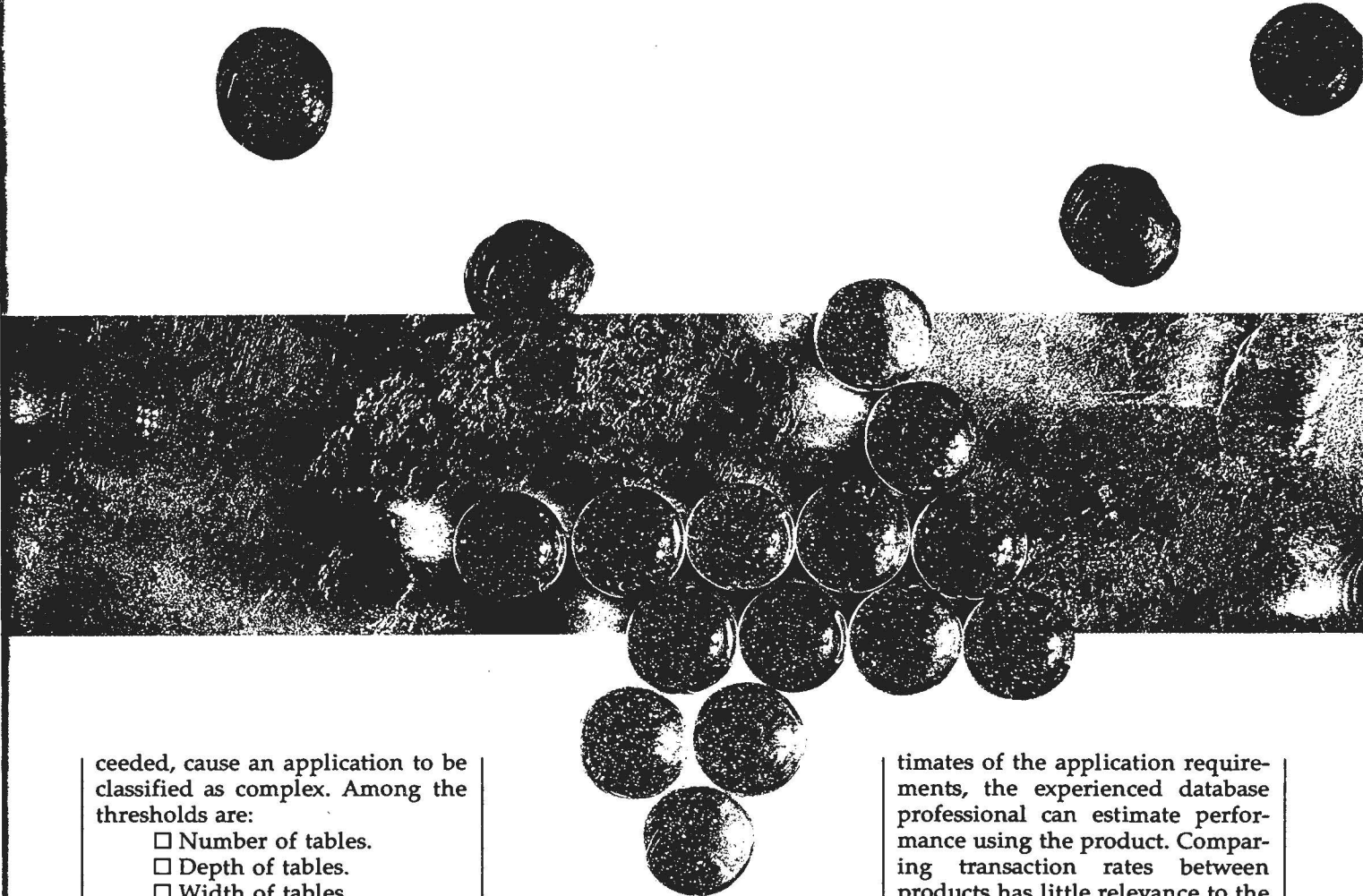


multaneously. Transaction management and recovery would never interfere with concurrency. SQL allowing the same result to be accomplished from syntactically different statements would never affect performance.

Unfortunately, no one has been able to reach this ideal in practice. To be fair, commercial relational database products are improving in this regard. However, such a level of intelligence in a database may be neither achievable nor desirable. We are not likely to solve the inherent problems for quite a while. Until we are able to, database performance tuning will be an issue for many production relational applications.

Some professionals believe that faster and less expensive hardware, more sophisticated relational database products, and distributed database technology will alleviate the need for performance tuning. If the user community's needs typically did not expand to consume available resources, I would agree. But the volume of desired software applications in a production organization tends to grow faster than the acquisition of the hardware required to implement it. For certain production relational applications, hardware requirements frequently are underestimated during design and development. The result is a need for performance tuning to deploy the system or to prolong the life of the hardware platform.

I call the class of production relational applications that may need performance tuning "complex database applications." A number of thresholds, when ex-



ceeded, cause an application to be classified as complex. Among the thresholds are:

- Number of tables.
- Depth of tables.
- Width of tables.
- Number of concurrent users.

Scope of transactions.  
 Number of tables in a typical join.

- Total database size.
- Transaction rate.
- General transaction complexity.

The threshold for each of these parameters depends on the hardware platform, operating system, and database product by version. The thresholds are altered by the mix of other nonrelational work that must be done in the same environment, since these effectively lower the available capacity. It is not sufficient to know that none of the thresholds is exceeded. If enough of the param-

eters are close to their threshold values, the application may qualify as complex.

Clearly, selecting a database product depends on analyzing the requirements and a knowledge of the performance characteristics of available products. In mission-critical and complex applications, any amount of performance tuning may not result in a viable system if the wrong selection has been made.

Transaction rate benchmarks can be quite useful in analyzing the performance characteristics of a relational database, but proper interpretation of the benchmark requires a knowledge of the product. Then, along with judicious es-

timates of the application requirements, the experienced database professional can estimate performance using the product. Comparing transaction rates between products has little relevance to the performance that can be expected in production. If the expertise is not available to perform such an analysis, performance modeling and analysis, in which the application transaction mix is simulated in the target environment, can yield far more useful information.

Assuming you have not been placed in such an untenable position, we will examine some of the knobs that you may turn to tune a relational database application. I use the term relational database application, not relational database management system; in more than eight years of relational database applications consulting, I found most problems of this sort are the result of using the relational database product incompatibly with

ARTWORK: PAUL FARGOHELD

the assumptions of nonprocedural data processing. So, I will briefly discuss some characteristics of applications appropriately using relational databases.

Not all relational database products will allow you to use every knob I discuss. In fact, so many products are available that I am unable to discuss which allow what kind of tuning or the specifics of how to accomplish the tuning for a particular product here. However, the principles are quite clear. The prospective relational database user should ask the vendor which apply if performance is an issue.

Many mathematical techniques for estimating performance exist. By comparing actual performance to the estimate, an educated guess can be made about the nature of the problem. Unfortunately, these depend on an intimate knowledge of the database product internals. I will discuss the empirical concepts behind these techniques.

#### IDENTIFY THE PROBLEM

After you have determined that performance is an issue, the next task is to determine the causes. Though it would be nice if all available products provided a tool to analyze the current performance characteristics of an SQL statement, the available tools generally do not provide high-level information. Usually, you must know what questions to ask and understand the answers.

Sources of difficulty may be:

**User delays:** When a user issues a request to the database, some application systems will allow the user to suspend processing temporarily. Unless the database product allows the administrator to set a time out, this can mean that other users must wait for the database to release locks allocated to the suspended process. The ordinary "coffee break" can introduce major performance problems if not handled properly.

Most operating systems allow the user to request an abort. The mechanism may be disabled or trapped by the programmer. However, applications can handle an abend incorrectly as regards transaction management and re-

## The difficulty is knowing that a problem exists and its causes

covery. A new release of a database product may also fail to detect an abend and then systematically handle the current transaction. The policy may be to either abort or commit pending transactions when the user application exits. Either default policy may be inappropriate for a given application. It is even possible that no default policy is implemented by the vendor, in which case the application developer must take on this task.

This problem typically appears when locks are held by a process for what appears to be undue periods of time.

**User I/O:** The user process may intersperse with database requests and results processing with unnecessary terminal I/O, thus delaying the completion of database processing. It may also make multiple requests of the database when a single request could do the job. In this case, as in the previous, the database software cannot know how to optimize the overall process mix.

This problem, like the last, also appears as locks held by the user process for undue periods of time.

**CPU consumption:** Detecting excessive use of CPU time by a database product can be difficult. Measuring the use is not as difficult, since even if the vendor does not provide a tool for this purpose, most operating systems do. If that fails, numerous third-party products are available for operating system performance monitoring.

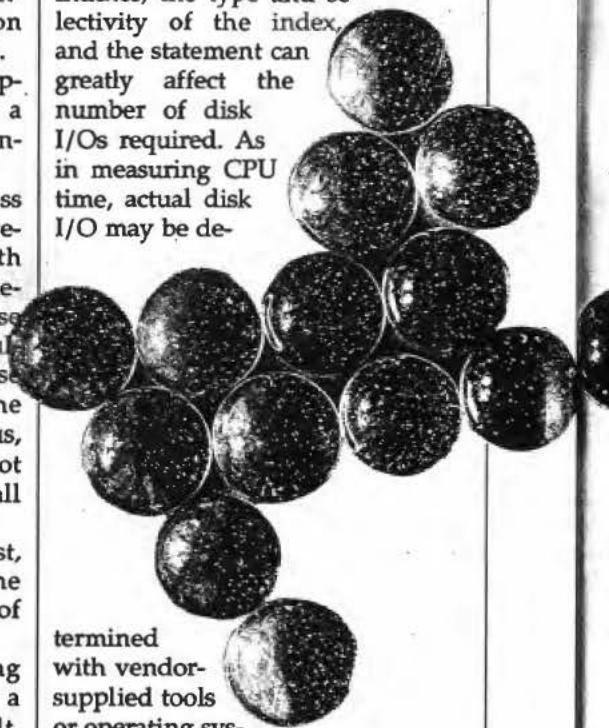
The difficulty is in knowing that a problem exists. The most common practice is not analytical but intuitive. With experience, a database professional learns to expect a correspondence between CPU consumption and the complexity of an SQL statement, given a particular database and environment. Variations on the SQL statement can be run to make comparisons in CPU consumption. Even

this requires some sophistication, since a small change in SQL syntax can result in a large change in performance even when things are going well. If enough is known about the product, an analysis of the necessary CPU consumption may be made. However, this is difficult and results in a rough estimate at best.

**Disk I/O:** Based on the distribution and volume of data affected by an SQL statement, an estimate of the amount of disk I/O required may be made. It is important to consider the disk I/O required in the use of indexes as well as data and data distribution. Of course, the algorithms used in processing indexes, the type and selectivity of the index, and the statement can greatly affect the number of disk I/Os required. As in measuring CPU time, actual disk I/O may be de-

termined with vendor-supplied tools or operating system performance monitoring tools. Excessive disk I/O is usually more sensitive to the amount of data processed than is excessive CPU consumption.

**Optimizer:** Unnecessary disk I/O and CPU consumption may be the optimizer's fault. The database query optimizer should determine which algorithms should be used in processing portions of the SQL statement and in what order. A solution is called a strategy, access plan, or query plan. If more than one strategy can do the job, the optimizer may not make the best choice. Usually, a bad strategy is so bad that something is obviously wrong, considering the amount of



the possibility that the optimizer can select an efficient strategy.

SQL is intended to be a non-procedural language, so you can expect an SQL optimizer to perform better when faced with standard, nonprocedural requests that the vendor has anticipated. It is useful to ask your vendor for specific examples of complex SQL on which their product performs well. The examples can be compared with the level of complexity you expect your application to require, assuming that complexity is not the result of poor database design.

This perspective requires a different view of relational database applications. Although row-at-a-time processing need not suffer unduly in a normalized relational database, the system has clearly not been designed for such procedural processing. But, few applications demand row-at-a-time transaction processing and high transaction rates. Candidate applications for such processing require recoverability and performance for many concurrent users.

In a traditional implementation, transactions are tied to data entry processes. The data entry operator examines one or more records of existing data and makes updates or inserts, requiring feedback of a commit with each transaction. Denormalization can help performance in such cases, but this record-at-a-time processing can be accomplished nonprocedurally. If the operator submits a qualified update or insert, an examination of selected records within a transaction is then not needed. The results of the request can be displayed outside the transaction for confirmation feedback.

In general, denormalization is undesirable. However, it is not always a possibility to circumvent an existing application architecture when migrating to a relational database. Regardless of the strategy's wisdom, pragmatic and political pressures might demand that the database system be changed without initial alterations in the application behavior. Indeed, management often finds it very difficult to permit changes to the application behavior since this has profound and visible oper-

## Political pressures may change a database system

ational effects on the business.

Faced with such a migration, the database designer often must pull out all stops to achieve the necessary performance. Denormalization should be the last of the designer's efforts. If the database is designed initially in third normal form, selective changes can be made to minimize the loss of flexibility, data nonredundancy, and data integrity. Care in isolating the application code from the database schema will allow these compromises to be removed as faster hardware and more sophisticated database products are available.

### SUMMARY TABLES

In some applications, SQL statements need to be qualified, based on the results of complex computation. For example, approval of an ATM withdrawal may depend on the average monthly balance. The cost of computing the average monthly balance with each withdrawal may be too great. An alternative is to recompute a running average with each transaction, which costs less. The numbers to recompute the average are stored in a summary table.

This technique can be extremely beneficial in applications using numeric data, especially financial, scientific, engineering, and manufacturing applications. The result is greater concurrency and better performance. Nonnumeric data can be reduced and stored in summary tables as well, but this is less common and beneficial. The price for summary tables is possible loss of data integrity and additional overhead during writes to the summary tables.

### FOREIGN KEYS

Even in a well-designed and highly tuned database, the cost of joins can be exorbitant. The join keys might not be optimal for the typical joins used by the application. It is important to remember that

joins are used not only in selects but to qualify deletions, insertions, and updates as well. Consider the following tables in third normal form. Parts and suppliers have a many-to-many relationship and cities and suppliers have many-to-one. Table A is quite deep and tables C and D are of moderate size:

TABLE\_A( PART\_NO, SUPPLIER\_NO, PART\_PRICE, QTY\_SUPPLIED)

TABLE\_B( PART\_NO, QTY\_ON\_HAND)

TABLE\_C( SUPPLIER\_NO, CITY)

TABLE\_D( CITY, PERCENT\_TAX)

Suppose we want to know how much city tax was paid on each part. We have to perform a three-table join between tables A, C, and D to find the percent tax to be applied against the total price. Table C contains CITY as a foreign key referencing table D. However, if table A also contained CITY as a foreign key, the required join would only be across two tables.

TABLE\_A'( PART\_NO, SUPPLIER\_NO, CITY, PART\_PRICE, QTY\_SUPPLIED)

The problem is accentuated with the original table A if we are looking for the tax from cities that require tax in a certain percentage range and for a range of part numbers. This requires that the selection of rows from each table be further restricted. The restriction on table A' can be applied before the join. The join can be processed quicker if table A is used. It would be more efficient if the restriction and the second join could be processed simultaneously. This is possible if the restriction is on the new foreign key. An index on the foreign key CITY in table A' can significantly improve performance. Of course, the more foreign keys introduced into an ap-

plication, the greater the concern should be with referential integrity, and thus, maintenance. Nothing is for free.

### SYNTAX SENSITIVITY

Some query optimizers are sensitive to the phrasing of an SQL statement. While vendors can sometimes provide the details of optimizer syntax sensitivity and are sometimes willing to do so, experience is the only certain means of acquiring this information. The details are subject to product version changes, environment, and data distribution.

Sometimes the problem is detected when two supposedly equivalent SQL statements perform dramatically differently. The syntax differences can be as subtle as the order in which tables are referenced:

#### EXAMPLE 1.

```
1. SELECT A.PART_NO, B.SUPPLIER_NO FROM
TABLE_B B, TABLE_A A
2. SELECT A.PART_NO, B.SUPPLIER_NO FROM
TABLE_A A, TABLE_B B
```

It can also be as complex and as obvious as using a join as opposed to a subquery and may actually yield different results. For example, assuming PART\_NO is the primary key in both tables, the following should be equivalent:

#### EXAMPLE 2

```
1. SELECT * FROM TABLE_A WHERE PART_NO
IN (SELECT PART_NO FROM TABLE_B WHERE
SUPPLIER_NO > 10)
2. SELECT * FROM TABLE_A WHERE TABLE_
A.PART_NO = TABLE_B.PART_NO AND TAB
LE_B.SUPPLIER_NO > 10
```

Depending on how the optimizer handles these two SELECTs, the results (for example, because of NULL handling) or the performance (for example, because of index selection) may differ significantly. Often, the sensitivity to syntax is only performance, making it more difficult to observe.

In the absence of further information, several tactics can be used to test for syntax sensitivity. First, rearrange the order of the table and column references. Then rearrange the order in which clauses appear. Next, use a more obvious alternate form of the

## Denormalization should be the last step for the designer

query—a subquery if you are not using one or eliminating a subquery if you are. Finally, rewrite the clauses using DeMorgan's Laws. This frequently yields results, since the cost of processing AND, OR, and NOT may be asymmetrical in otherwise equivalent statements.

### OPTIMIZER INSTRUCTION

Some vendors allow the developer to instruct the optimizer on the order in which to use indexes and access tables, the granularity of locks (table, page, row), and the degree of consistency (when to release or share locks). These considerations can affect performance dramatically. However, such instructions are not part of standard SQL and, thus, are not portable. Furthermore, the appropriate instructions are likely to change as the depth or width of tables or even the data types of columns evolve. As a result, such performance optimization can require ongoing maintenance.

Forcing the use of specific indexes requires an understanding of index selectivity. Selectivity is roughly the (average) number of rows a single index entry references. Using an index with greater selectivity (closer to one) will result in better performance. However, greater selectivity can mean a greater index update cost.

Forcing the access of tables is done by forcing the order in which indexes on the tables are used. You should access tables in an order that will keep the working set of data smallest. Thus, more restricted joins should be performed first, especially if an index is available to accommodate the join.

The cost of acquiring locks can be minimized by forcing the scope of lock acquisition and its order during the course of a transaction (usually consisting of multiple SQL statements). If a signifi-

cant number of the rows in the accessed pages need to be locked, you should probably force page locking. If a significant number of the accessed pages of a table need to be locked, you should probably force table locking. A significant number is the number of lock acquisitions at the lower of the two levels of lock granularity which just exceed the cost of acquiring a single lock at the higher level. The issue is more complex than this, so a test may be in order.

The degree of consistency (zero to four) that a particular application may require determines the amount of lock overhead and serialization. If the product allows you to force just the degree of consistency you need, considerable savings are possible. If the application requires a high degree of concurrency but users typically are not affecting the same data pages, a lower degree of consistency can eliminate unnecessary serializing of application transactions.

### TRICKING THE OPTIMIZER

If the optimizer uses information maintained in the database and that information can be updated, the optimizer can be tricked into selecting a particular processing strategy. For example, when faced with using more than one index on a given table, some optimizers look at the selectivity of indexes to determine which to use. By modifying the stored selectivities of the indexes, the index ranking can be influenced.

Although this tactic should be avoided, it may be necessary at times. Unless the product automatically updates the information it uses in strategy selection frequently, some manual maintenance is required. Some products leave maintenance of particular parameters to the database administrator's manual intervention, in which case the optimizer must be influenced.

### NONPROCEDURALIZATION

The benefits of using less procedural SQL statements include greater concurrency, better caching of data, and less disk I/O. Naturalization is used to combat unnecessary user I/O. It is a common mistake to develop a relational

database application that uses transactions consisting of very simple SQL statements. Rather than relying on the nonprocedurality of SQL and relational databases, third-generation language (3GL) developers may be inclined to treat relational access as one does procedural file access. This does not allow the optimizer or scheduler to do their work, since information about the intended result is not made available and a more restrictive sequence of operations is imposed.

Suppose an application is spending most of its time passing requests and results back and forth to the database. If there is no significant intermediate processing, the series of requests might be written in fewer SQL statements. When possible, an application should issue the highest level SQL request possible to the database. Then, if the product performs poorly, these requests are able to be selectively broken into multiple requests and code written to support them. In other words, relational database requests should replace procedural functions from the top-down, instead of from the bottom-up.

Typically, the level of complexity of an SQL statement will exceed the user's ability to understand it before it will confuse the optimizer. If you happen to be adept with SQL, you can write SQL that you understand, which causes the optimizer to select a poor strategy.

The benefits of making more nonprocedural requests of the database have a practical limit. However, it has been my experience that this limit is rarely approached. Of course, the ability of others to understand and maintain such code must be considered.

#### PROCEDURALIZATION

As noted, sometimes the effect of an SQL statement can be achieved with multiple SQL statements. This should be done as a next-to-last resort in tuning for performance (denormalization is the last-ditch effort to achieve necessary performance). However, if the complexity of the SQL results

## Even in a highly tuned database, the cost of joins can be exorbitant

in intolerable performance, the statement can be broken into multiple statements. Proceduralization helps an ailing optimizer.

More often than not, proceduralizing an SQL statement requires that explicit transaction control be asserted so the new sequence of SQL statements appears as a single transaction to the database. This can improve performance at the cost of concurrency, although I have seen many cases resulting in improved concurrency when the optimizer was already doing the job poorly. Ultimately, if you have a need for both concurrency and performance and have exhausted all other options, the best bet for you is to test the effects of proceduralization.

Sometimes, a temporary table to store intermediate results must be created to proceduralize the statement. Certain products do not allow the creation of temporary tables within a transaction, so the intermediate results can be inserted in an existing table and deleted when no longer needed.

#### STORAGE ALLOCATION

Controlling storage allocation is one technique for improving disk I/O. Inter-table clustering is an example of detailed storage allocation to help performance. By forcing data commonly accessed together into contiguous storage, a single disk I/O can do the job of two or more. Intra-table clustering (physically storing a table in sort order) helps when a table is accessed in a particular order.

Commonly joined tables can be placed on separate disk drives to improve performance, assuming the hardware allows the drives to be controlled in parallel. Similarly, recovery logs can be placed on separate drives.

Some products let tables be

distributed horizontally across available disk drives. This technique can be effective where parallel processing of the data is possible and the application requires that a lot of rows be scanned in each transaction.

Managing the rate at which dynamic allocation occurs can balance performance over time so that each transaction is less likely to require space during peak performance periods.

#### STORED PROCEDURES

More and more relational database vendors are supporting stored procedures (also known as stored commands or precompiled commands). Stored procedures allow the developer to combine multiple SQL statements into a single user-defined and named command, thereby effectively making SQL an extendible language. Unfortunately, current implementations have restrictions on the statements that can be processed in a stored procedure.

Because most implementations perform some preliminary processing of SQL statements, stored procedures can improve performance by reducing the per execution overhead. The disadvantage is that the parser must translate a growing list of stored procedure names. The primary performance advantage is the elimination of redundant parsing and optimization overhead, reducing CPU consumption.

#### TRIGGERS

A database trigger is a set of user-defined conditional actions that occur on update, deletion, or insert to a table that must be scanned (to determine if the action should be triggered) is too large, the per transaction overhead will outweigh the benefits. The number of locks held by a complex trigger can reduce concurrency. As with other potential performance optimizations, triggers should be used judiciously.

#### TRANSACTION MANAGEMENT

A series of SQL statements might be bracketed by explicit transaction control for two reasons. The first is to ensure that the entire series of statements can be rolled

back if any failure occurs. The second is insurance against reading another user's dirty data. Typically, the scope (in time and data) of a transaction is managed under a worst-case rule. Even if rollback of the entire set of transactions is not desired, a COMMIT will not be issued until the entire sequence of SQL statements is completed.

Through careful examination of the relationship between transactions, the sequence transaction requests can be controlled and the scope of transactions reduced. Although this higher level of transaction management must be achieved by 3GL code, insuring that multiple transactions are never submitted that update the same data pages is highly effective.

If the designer or developer has such privileged information, the application can control concurrency in a way that the database scheduler cannot is not able to without excessive serialization. By introducing slight delays in the submission of database transactions and coordinating these across users, conflicts between users can be removed, allowing even more concurrency and greater system throughput.

### LOGGING OVERHEAD

Overhead due to transaction logging can sometimes be reduced by minimizing the amount of data involved in writes to the database. By selecting a dense rather than sparse encoding of data fields, the width of tables can be significantly reduced. This in turn reduces the number of bytes which must be processed per row of data. The trade-off is that users may no longer recognize the data's meaning, although this can usually be handled through views.

Similarly, selecting a data encoding that converts a disjunctive list into a range is appropriate if the typical access pattern involves such a range. Then, by adding an index on the particular column, the SQL statements will access fewer index pages more quickly.

It is useful to encode primary keys as integers, with successive rows as an increment of the highest primary key value. The original primary key becomes a candidate key in the table. The result of

## An application should issue the highest level SQL request possible

this compression is that the primary key index pages can reference the greatest number of rows. In addition, once the key is known in a particular transaction, it can be used to eliminate unnecessary comparisons between compound keys and their component values.

### BUFFER SIZES

The allocation of the memory to be used by the database for query parsing, input buffering, results processing, and output buffering can improve performance. However, it is necessary to know the amount of memory used by the typical transaction, since most products provide this as a system tuning parameter rather than a parameter to tune the individual transaction.

During heavy update and insert periods, the input buffer may become overloaded, requiring a temporary hold on further processing or perhaps flushing the buffer to disk. Both reduce the system throughput, so that a larger input buffer may be desirable.

It is possible to create an SQL statement nested so deep that the parser does not have enough working memory to parse the query. In a similar way, multiple long statements may overload the parse buffer if the parser allows multiple statements to be held in the buffer simultaneously.

Working buffers are used for processing intermediate results. Typical intermediate processing operations include sorting and merging of two or more tables. Again, if the intermediate results are too large to be cached in memory, they must be swapped to disk.

If the final results are available faster than the application can accept them, output buffer sizes become important. This, along with the need to release locks as soon as

possible by committing outstanding transactions, emphasizes the importance of application architecture. The ideal relational application never imposes delays on the relational database, either during input or output.

### OTHER SYSTEM PARAMETERS

Most vendors allow the database administrator to control many system parameters such as the number of concurrent users, the application buffer space per user, open cursors, concurrent processes, open tables per user, and memory allocated for sort-merge operations. In addition, many operating system parameters may affect the performance of the database. For example, the number of files open at any time, the maximum amount of disk space allocated per file, and the maximum number of files in a directory can affect system performance and database administration. These parameters vary from operating system to operating system, as well as from vendor to vendor.

The database vendor should provide guidelines for optimizing both sets of parameters for your installation. However, because of the degree of sensitivity to your application and the hardware and software environment, you should make an applications expert and an operating systems expert available during the tuning process.

Tuning relational database applications for performance requires system thinking. It is not enough to be proficient in SQL or database normalization. While expertise in the technical aspects of relational database systems and performance modeling are important, experience with the environment using the database product of choice is what counts. One last piece of advice: don't ignore how the application uses the database. It affects not only response time, but also that aspect of performance with which production systems are often concerned—throughput. ■

David McGoveran is president of Alternative Technologies in Santa Cruz, Calif., a consulting firm that has been specializing in relational database applications for the last eight years.