Advanced SQL Database Programmer Handbook

Donald K. Burleson
Joe Celko
John Paul Cook
Peter Gulutzan

RAMPANT
TECHPRESS

Advanced SQL Database Programmers Handbook

By Donald K. Burleson, Joe Celko, John Paul Cook, and Peter Gulutzan

Table of Contents

Conventions Used in this Book

It is critical for any technical publication to follow rigorous standards and employ consistent punctuation conventions to make the text easy to read.

However, this is not an easy task. Within Oracle there are many types of notation that can confuse a reader. Some Oracle utilities such as STATSPACK and TKPROF are always spelled in CAPITAL letters, while Oracle parameters and procedures have varying naming conventions in the Oracle documentation. It is also important to remember that many Oracle commands are case sensitive, and are always left in their original executable form, and never altered with italics or capitalization.

Hence, all Rampant TechPress books follow these conventions:

Parameters - All Oracle parameters will be *lowercase italics*. Exceptions to this rule are parameter arguments that are commonly capitalized (KEEP pool, TKPROF), these will be left in ALL CAPS.

Variables – All PL/SQL program variables and arguments will also remain in lowercase italics (*dbms_job, dbms_utility*).

Tables & dictionary objects – All data dictionary objects are referenced in lowercase italics (*dba_indexes, v$sql*). This includes all v$ and x$ views (*x$kcbcbh, v$parameter*) and dictionary views (*dba_tables, user_indexes*).

SQL – All SQL is formatted for easy use in the code depot, and all SQL is displayed in lowercase. The main SQL terms (select, from, where, group by, order by, having) will always appear on a separate line.

Programs & Products – All products and programs that are known to the author are capitalized according to the vendor specifications (IBM, DBXray, etc). All names known by Rampant TechPress to be trademark names appear in this text as initial caps. References to UNIX are always made in uppercase.

About the Authors

Donald K. Burleson is one of the world's top Oracle Database experts with more than 20 years of full-time DBA experience. He specializes in creating database architectures for very large online databases and he has worked with some of the world's most powerful and complex systems. A former Adjunct Professor, Don Burleson has written 15 books, published more than 100 articles in national magazines, serves as Editor-in-Chief of Oracle Internals and edits for Rampant TechPress. Don is a popular lecturer and teacher and is a frequent speaker at Oracle Openworld and other international database conferences.

Joe Celko was a member of the ANSI X3H2 Database Standards Committee and helped write the SQL-92 standards. He is the author of over 450 magazine columns and four books, the best known of which is *SQL for Smarties* (Morgan-Kaufmann Publishers, 1999). He is the Vice President of RDBMS at Northface University in Salt Lake City.

John Paul Cook is a database and .NET consultant. He also teaches .NET, XML, SQL Server, and Oracle courses at Southern Methodist University's location in Houston, Texas.

Peter Gulutzan is the co-author of one thick book about the SQL Standard (SQL-99 Complete, Really) and one thin book about optimization (*SQL Performance Tuning*). He has written about DB2, Oracle, and SQL Server, emphasizing portability and DBMS internals, in previous dbazine.com articles. Now he has a new job: he works for the "Number Four" DBMS vendor, MySQL AB.

Foreword

SQL programming is more important than ever before. When relational databases were first introduced, the mark of a good SQL programmer was someone who could come up with the right answer to the problems as quickly as possible. However, with the increasing importance of writing efficient code, today the SQL programmer is also charged with writing code quickly that also executes in optimal fashion. This book is dedicated to SQL programming internals, and focuses on challenging SQL problems that are beyond the scope of the ordinary online transaction processing system. This book dives deep into the internals of Oracle programming problems and presents challenging and innovative solutions to complex data access issues.

This book has brought together some of the best SQL experts to address the important issues of writing efficient and cohesive SQL statements. The topics include using advanced SQL constructs and how to write programs that utilize complex SQL queries. Not for the beginner, this book explores complex time-based SQL queries, managing set operations in SQL, and relational algebra with SQL. This is an indispensable handbook for any developer who is challenged with writing complex SQL inside applications.

SQL as a Second Language

Thinking in SQL

Learning to think in terms of SQL is a jump for most programmers. Most of your career is spent writing procedural code and suddenly, you have to deal with non-procedural code. The thought pattern has to change from sequences to sets of data elements.

As an example of what I mean, consider a posting made on 1999 December 22 by *J.R. Wiles* to a Microsoft SQL Server website: "I need help with a statement that will return distinct records for the first three fields where all values in field four are all equal to zero."

What do you notice about this program specification? It is very poorly written. But this is very typical of what people put out on the Internet when they ask for SQL help.

There are no fields in a SQL database; there are columns. The minute that someone calls a column a field, you know that he is not thinking in the right terms.

A field is defined within the application program. A column is defined in the database, independently of the application program. This is why a call to some library routine in a procedural language like "READ a, b, c, d FROM My_File;" is not the same as "READ d, c, b, a FROM My_File;" while

"SELECT a, b, c, d FROM My_Table;" and "SELECT d, c, b, a FROM My_Table;" are the same thing in a different order.

The next problem is that he does not give any DDL (Data Definition Language) for the table he wants us to use for the problem. This means we have to guess what the column datatypes are, what the constraints are and everything else about the table. However, he did give some sample data in the posting which lets us guess that the table looks like this:

```
CREATE TABLE Foobar
(col1 INTEGER NOT NULL,
 col2 INTEGER NOT NULL,
 col3 INTEGER NOT NULL,
 col4 INTEGER NOT NULL);

INSERT INTO Foobar
VALUES (1, 1, 1, 0),
       (1, 1, 1, 0),
       (1, 1, 1, 0),
       (1, 1, 2, 1),
       (1, 1, 2, 0),
       (1, 1, 2, 0),
       (1, 1, 3, 0),
       (1, 1, 3, 0),
       (1, 1, 3, 0);
```

Then he tells us that the query should return these two rows:

```
  (1, 1, 1, 0)
  (1, 1, 3, 0)
```

Did you notice that this table had no name and no key specified? While it is a bad practice not to have a declared PRIMARY KEY on a table, just ignore it for the moment.

At this point, people started sending in possible answers. *Tony Rogerson* at Torver Computer Consultants Ltd came up with this answer:

```
SELECT *
  FROM (SELECT col1, col2, col3, SUM(col4)
```

```
        FROM Foobar
      GROUP BY col1, col2, col3)
      AS F1(col1, col2, col3, col4)
 WHERE F1.col4 = 0;
```

Using the assumption, which is not given anywhere in the specification, Tony decided that col4 has a constraint -- ...

```
      col4 INTEGER NOT NULL CHECK(col4 IN (0, 1)));
```

Notice how doing this INSERT INTO statement would ruin his answer:

```
 INSERT INTO Foobar (col1, col2, col3, col4)
 VALUES (4, 5, 6, 1), (4, 5, 6, 0), (4, 5, 6, -1);
```

But there is another problem. This is a procedural approach to the query, even though it looks like SQL! The innermost query builds groups based on the first three columns and gives you the summation of the fourth column within each group. That result, named F1, is then passed to the containing query which then keeps only groups with all zeros, under his assumption about the data.

Now, students, what do we use to select groups from a grouped table? The HAVING clause! *Mark Soukup* noticed this was a redundant construction and offered this answer:

```
SELECT col1, col2, col3, 0 AS col4zero
  FROM Foobar
 GROUP BY col1, col2, col3
HAVING SUM(col4) = 0;
```

Why is this an improvement? The HAVING clause does not have to wait for the entire subquery to be built before it can go to work. In fact, with a good optimizer, it does not have to wait for an entire group to be built before dropping it from the results.

However, there is still that assumption about the values in col4. *Roy Harvey* came up with answer that gets round that problem:

```
SELECT col1, col2, col3, 0 AS col4zero
  FROM Foobar
 GROUP BY col1, col2, col3
HAVING COUNT(*)
     = SUM(CASE WHEN col4 = 0
                THEN 1 ELSE 0 END);
```

Using the CASE expression inside an aggregation function this way is a handy trick. The idea is that you count the number of rows in each group and count the number of zeros in col4 of each group and if they are the same, then the group is one we want in the answer.

However, when most SQL compilers see an expression inside an aggregate function like SUM(), they have trouble optimizing the code.

I came up with two approaches. Here is the first:

```
SELECT col1, col2, col3
  FROM Foobar
 GROUP BY col1, col2, col3
HAVING MIN(col4) = MAX(col4)  -- one value in table
   AND MIN(col4) = 0;         -- has a zero
```

The first predicate is to guarantee that all values in column four are the same. Think about the characteristics of a group of identical values. Since they are all the same, the extremes will also be the same. The second predicate assures us that col4 is all zeros in each group. This is the same reasoning; if they are all alike and one of them is a zero, then all of them are zeros.

However, these answers make assumptions about how to handle NULLs in col4. The specification said nothing about

NULLs, so we have two choices: (1) discard all NULLs and then see if the known values are all zeros (2)Keep the NULLs in the groups and use them to disqualify the group. To make this easier to see, let's do this statement:

```
INSERT INTO Foobar (col1, col2, col3, col4)
VALUES (7, 8, 9, 0), (7, 8, 9, 0), (7, 8, 9, NULL);
```

Tony Rogerson's answer will drop the last row in this statement from the SUM() and the outermost query will never see it. This group passes the test and gets to the result set.

Roy Harvey's will convert the NULL into a zero in the SUM(), the SUM() will not match COUNT(*) and thus this group is rejected.

My first answer will give the "benefit of the doubt" to the NULLs, but I can add another predicate and reject groups with NULLs in them.

```
SELECT col1, col2, col3
  FROM Foobar
 GROUP BY col1, col2, col3
HAVING MIN(col4) = MAX(col4)
   AND MIN(col4) = 0
   AND COUNT(*) = COUNT(col4);   -- No NULL in the column
```

The advantages of using simple aggregate functions is that SQL engines are tuned to produce them quickly and to optimize code containing them. For example, the MIN(), MAX() and COUNT(*)functions for a base table can often be determined directly from an index or from a statistics table used by the optimizer, without reading the base table itself.

As an exercise, what other predicates can you write with aggregate functions that will give you a group characteristic? I will offer a copy of *SQL FOR SMARTIES* (second edition) for

the longest list. Send me an email at 71062.1056@compuserve.com with your answers.

SQL View Internals

SQL Views Transformed

> "In 1985, Codd published a set of 12 rules to be used as "part of a test to determine whether a product that is claimed to be fully relational is actually so". His Rule No. 6 required that all views that are theoretically updatable also be updatable by the system."
>
> -- C. J. Date, Introduction To Database Systems

IBM DB2 v 8.1, Microsoft SQL Server 2000, and Oracle9i all support views (yawn). More interesting is the fact that they support very similar advanced features (extensions to the SQL-99 Standard), in a very similar manner.

Syntax

As a preliminary definition, let's say that a view is something that you can create with a CREATE VIEW statement, like this:

```
CREATE VIEW <View name>
[ <view column list> ]
AS <query expression>
[ WITH CHECK OPTION ]
```

This is a subset of the SQL-99 syntax for a view definition. It's comforting to know that "The Big Three" DBMSs — DB2, SQL Server, and Oracle — can all handle this syntax without any problem. In this article, I'll discuss just how these DBMSs "do" views: what surprises exist, what happens internally, and what features The Big Three present, beyond the call of duty.

I'll start with two Cheerful Little Facts, which I'm sure will surprise most people below the rank of DBA.

Cheerful Little Fact #1:

The CHECK OPTION clause doesn't work the same way that a CHECK constraint works! Watch this:

```
CREATE TABLE Table1 (column1 INT)
CREATE VIEW View1 AS
   SELECT column1 FROM Table1 WHERE column1 > 0
   WITH CHECK OPTION
INSERT INTO View1 VALUES (NULL) <-- This fails!
CREATE TABLE Table2 (column1 INT, CHECK (column1 > 0))
INSERT INTO Table2 VALUES (NULL) <-- This succeeds!
```

The difference, and the reason that the Insert-Into-View statement fails while the Insert-Into-Table statement succeeds, is that a view's CHECK OPTION must be TRUE while a table's CHECK constraint can be either TRUE or UNKNOWN.

Cheerful Little Fact #2:

Dropping the table doesn't cause dropping of the view! Watch this:

```
CREATE TABLE Table3 (column1 INT)
CREATE VIEW View3 AS SELECT column1 FROM Table3
DROP TABLE Table3
CREATE TABLE Table3 (column0 CHAR(5), column1 SMALLINT)
INSERT INTO Table3 VALUES ('xxxxx', 1)
SELECT * FROM View3 <-- This succeeds!
```

This bizarre behavior is exclusive to Oracle8i and Microsoft SQL Server — when you drop a table, the views on the table are still out there, lurking. If you then create a new table with the same name, the view on the old table becomes valid again! Apart from the fact that this is a potential security flaw and a

violation of the SQL Standard, it illustrates a vital point: The attributes of view View3 were obviously not fixed in stone at the time the view was created. At first, View3 was a view of the first (INT) column, but by the time the SELECT statement was executed, View3 was a view of the second (SMALLINT) column. This is the proof that views are reparsed and executed when needed, not earlier.

View Merge

What precisely is going on when you use a view? Well, there is a module, usually called the Query Rewriter (QR), which is responsible for, um, rewriting queries. Old QR has many wrinkles — for example, it's also responsible for changing some subqueries into joins and eliminating redundant conditions. But here we'll concern ourselves only with what QR does with queries that might contain views.

At CREATE VIEW time, the DBMS makes a view object. The view object contains two things: (a) a column list and (b) the text of the view definition clauses. Each column in the column list has two fields: {column name, base expression}. For example, this statement:

```
CREATE VIEW View1 AS
 SELECT column1+1 AS view_column1, column2+2 AS view_column2
 FROM Table1
 WHERE column1 = 5
results in a view object that contains this column list:
{'view_column1','(column1+1)'} {'view_column2','(column2+2)'}
```

The new view object also contains a list of the tables upon which the view directly depends (which is clear from the FROM clause). In this case, the list looks like this:

Table1

When the QR gets a query on the view, it does these steps, in order:

LOOP:

[0] Search within the query's table references (in a SELECT statement, this is the list of tables after the word FROM). Find the next table reference that refers to a view object instead of a base-table object. If there are none, stop.

[1] In the main query, replace any occurrences of the view name with the name of the table(s) upon which the view directly depends.

Example:

```
SELECT View1.* FROM View1
```

becomes

```
SELECT Table1.* FROM Table1
```

[2] LOOP: For each column name in the main query, do:

```
  If (the column name is in the view definition)
   And (the column has not already been replaced in this pass of the
outer loop)
   Then:
     Replace the column name with the base expression from the column
list
```

Example:

```
SELECT view_column1 FROM View1 WHERE view_column2 = 3
```

Becomes

```
SELECT (column1+1) FROM Table1 WHERE (column2+2) = 3
```

[3] Append the view's WHERE clause to the end of the main query.

Example:

```
SELECT view_column1 FROM View1
```

becomes

```
SELECT (column1+1) FROM Table1 WHERE column1 = 5
```

Detail: If the main query already has a WHERE clause, the view's WHERE clause becomes an AND sub-clause.

Example:

```
SELECT view_column1 FROM View1 WHERE view_column1 = 10
```

Becomes

```
SELECT (column1+1) FROM Table1 WHERE (column1+1) = 10 AND column1 = 5
```

Detail: If the main query has a later clause (GROUP BY, HAVING, or ORDER BY), the view's WHERE clause is appended before the later clause, instead of at the end of the main query.

[4] Append the view's GROUP BY clause to the end of the main query. Details as in [3].

[5] Append the view's HAVING clause to the end of the main query. Details as in [3]

[6] Go back to step [1].

There are two reasons for the loop:

- The FROM clause may contain more than one table and you may only process for one table at a time.

- The table used as a replacer might itself be a view. The loop must repeat till there are no more views in the query.

A final detail: Note that the base expression is "(A)" rather than "A." The reason for the extra parentheses is visible in this example:

```
CREATE VIEW View1 AS
  SELECT table_column1 + 1 AS view_column1
  FROM Table1
SELECT view_column1 * 5 FROM View1
```

When evaluating the SELECT, QR ends up with this query if the extra parentheses are omitted:

```
SELECT table1_column + 1 * 5 FROM Table1
```

... which would be wrong, because the * operator has a higher precedence than the + operator. The correct expression is:

```
SELECT (table1_column + 1) * 5 FROM Table1
```

And voila. The process above is a completely functional "view merge" procedure, for those who wish to go out and write their own DBMS now. I've included all the steps that are sine qua nons.

The Small Problem with View Merge

A sophisticated DBMS performs these additional steps after or during the view merge:

- Eliminate redundant conditions caused by the replacements.

- Invoke the optimizer once for each iteration of the loop.

All three of our DBMSs are sophisticated. But here's an example of a problematic view and query:

```
CREATE TABLE Table1 (column1 INT PRIMARY KEY, column2 INT)
CREATE TABLE Table2 (column1 INT REFERENCES Table1, column2 INT)
CREATE VIEW View1 AS
  SELECT Table1.column1 AS column1, Table2.column2 AS column2
  FROM Table1, Table2
  WHERE Table2.column1 = Table1.column1
SELECT DISTINCT column1 FROM View1 <-- this is slow
SELECT DISTINCT column1 FROM Table2 <-- this is fast

— Source: SQL Performance Tuning, page 209.
```

The selection from the view will return precisely the same result as the selection from the table, but Trudy Pelzer and I tested the example on seven different DBMSs (for our book SQL Performance Tuning, see the References), and in every case the selection-from-the-table was faster. This indicates that the optimizer isn't always ready for the inefficient queries that the Query Rewriter can produce.

Ultimately, the small problem is that the "view merge" is a mechanical simpleton that can produce code that humans would immediately see as silly. But the view-merge process itself is so simple that it should be almost instantaneous. (I say "almost" because there are lookups to be done in the system catalog.)

So much for the small problem. Now for the big one.

Temporary Tables

Here's an example of a view definition:

```
CREATE VIEW View1 AS
    SELECT MAX(column1) AS view_column1
    FROM Table1
```

Now, apply the rules of view merge to this SELECT statement:

```
SELECT MAX(view_column1) FROM View1
```

The view merge result is:

```
SELECT MAX((MAX(column1)) FROM Table1
```

... which is illegal. View merge will always fail if the view definition includes MAX, or indeed any of these constructions:

- GROUP BY, or anything that implies grouping, such as HAVING, AVG, MAX, MIN, SUM, COUNT, or any proprietary aggregate function

- DISTINCT, or anything that implies distinct, such as UNION, EXCEPT, INTERSECT, or any proprietary set operator

So if a DBMS encounters any of these constructions, it won't use view merge. Instead it creates a temporary table to resolve the view. This time the method is:

```
[ at the time the view is referenced ]
CREATE TEMPORARY TABLE Arbitrary_name
 (view_column1 <data type>)
INSERT INTO Arbitrary_name SELECT MAX(column1) FROM Table1
```

That is, the DBMS has to "materialize" the view by making a temporary table and populating it with the expression results.

Then it's just a matter of replacing the view name with the arbitrary name chosen for the temporary table:

```
SELECT MAX(view_column1) FROM View1
```

Becomes

```
SELECT MAX(view_column1) FROM Arbitrary_name
```

And the result is valid. The user doesn't actually see the temporary table, but it's certainly there, and takes up space as long as there is an open cursor for the SELECT.

If a view is materialized, then any data-change (UPDATE, INSERT, or DELETE) statements affect the temporary table, and that is useless — users might want to change Table1, but they don't want to change Arbitrary_name, they don't even know it's there. This is an example of a class of views that is non-updatable. As we'll see, it's not the only example.

So ...

- With view merge alone, it is possible to handle most views.
- With view merge and temporary tables, it is possible to handle all views.

Permanent Materialized Views

Since the mechanism for materializing views has to be there anyway, an enhancement for efficiency is possible. Namely, why not make the temporary table permanent? In other words, instead of throwing the temporary table out after the SELECT is done, keep it around in case anyone wants to do a similar SELECT later. This enhancement is particularly noticeable for views based on groupings, since groupings take a lot of time.

DB2, Oracle, and SQL Server all have a "Permanent Materialized View" feature, although each vendor uses a different terminology. Here are the terms you are likely to encounter:

Vendor Terms that May Refer to Permanent Materialized Views

DBMS	VENDOR TERM
DB2	Automated Summary Table (AST) Materialized Query Table (MQT)
Oracle	Materialized View (MV) summary snapshot
SQL Server	Indexed View

The terms are not perfect synonyms because each vendor's implementation also has some distinguishing features; however, I'd like to emphasize what the three DBMSs have in common, which happens to be what an advanced DBMS ought to have.

- First, permanent materialized views are maintainable. Effectively, this means that if you have a permanent materialized view (say, View1) based on table Table1, then any update to Table1 must cause an update to View1. Since View1 is often a grouping of Table1, this is not an easy matter: either the DBMS must figure out what the change is to be as a delta, or it must recompute the entire grouping from scratch. To save some time on this, a DBMS may defer the change until: (a) it's necessary because someone is doing a select or (b) some arbitrary time interval has gone by. Oracle's term for the deferral is "refresh interval" and can be set by the user. (Oracle also allows the data to get

stale, but let's concentrate on the stuff that's less obviously a compromise.)

(By the way, deferrals work only because the DBMS has a "log" of updates, see my earlier DBAzine.com article, Transaction Logs. It's wonderful how after you make a feature for one purpose, it turns out to be useful for something else.)

- Second, permanent materialized views can be indexed. This is at least the case with SQL Server, and is probably why Microsoft calls them "indexed views". It is also the case with DB2 and Oracle.

- Third, permanent materialized views don't have to be referenced explicitly. For example, if a view definition includes an aggregate function (e.g.: CREATE VIEW View1 AS SELECT MAX(column1) FROM Table1) then the similar query -- SELECT MAX(column1) FROM Table1 -- can just select from the view, even though the SELECT doesn't ask for the view. A DBMS might sometimes fail to realize that the view is usable, though, so occasionally you'll have to check what your DBMS's "explain" facility says. With Oracle you'll then have to use a hint, as in this example:

```
SELECT/*+ rewrite(max_salary) */ max(salary)
  FROM Employees WHERE position = 'Programmer'
```

Permanent materialized views are best for groupings, because for non-grouped calculations (such as one column multiplied by another) you'll usually find that the DBMS has a feature for "indexing computed columns" (or "indexing generated columns") which is more efficient. Also, there are some restrictions on permanent materialized views (for example, views within views are difficult). But in environments where

grouped tables are queried often, permanent materialized views are popular.

UNION ALL Views

In the last few years, The Big Three have worked specifically on enhancing their ability to do UPDATE, DELETE, and INSERT statements on views based on a UNION ALL operator.

Obviously this is good because, as Codd's Rules (quoted at the start of this article) state: Users should expect that views are like base tables. But why specifically are The Big Three working on UNION ALL?

UNION ALL views are important because they work with range partitioning. That is, with a sophisticated DBMS, you can split one large table into n smaller tables, based on a formula. But what will you do when you want to work on all the tables at once again, treating them as a single table for a query? Use a UNION ALL view:

```
CREATEVIEW View1 AS
  SELECT a FROM Partition1
  UNION ALL
  SELECT a FROM Partition2
SELECT a FROM View1
UPDATE View1 SET a = 5
DELETE FROM View1 WHERE a = 5
INSERT INTO View1 VALUES (5)
```

Since View1 brings the partitions together, the SELECT can operate on the conceptual "one big table". And, since the view isn't using a straight UNION (which would imply a DISTINCT operation), the data-change operations are possible too. But there are some issues:

- Where should the new INSERT row end up: in Partition1 or Partition2?

- Where should the changed UPDATE row end up: in Partition1 or Partition2?

The issues arise because a typical partition will be based on some formula, for example: "when a < 5 then use Partition1, when a > 5 use Partition2". So it makes sense for the DBMS to combine UNION ALL view updates with the range partitioning formulas, and position new or changed rows accordingly. Unfortunately, when there are many partitions, this means that each partition's formula has to be checked to ensure that there is one (and only one) place to put the row.

An old "solution" was to disallow changes, including INSERTs, which affected the partitioning (primary) key. Now each DBMS has a reasonably sophisticated way of dealing with the problem; most notably DB2, which has a patented algorithm that, in theory, should handle the job quite efficiently.

Updatable UNION ALL views are useful for federated data, which (as I tend to think of it) is merely an extension of the range partitioning concept to multiple computers.

Alternatives to Views

Think of the typical hierarchy: person, employee, manager.

Each of these items can easily be handled in individual tables if a UNION ALL view is available when you want to deal with attributes that are held in common by all three tables. But in future it might be better to use subtables and supertables, since subtables and supertables were designed to handle hierarchies.

The decision might rest on how well your organization is adjusting to your DBMS's new Object/Relational features.

You cannot create a view with a definition that contains a parameter, so you might have to make a view for each separate situation:

```
CREATE VIEW View1 AS
  SELECT * FROM Table1
  WHERE column1 = 1
  WITH CHECK OPTION
CREATE VIEW View2 AS
  SELECT * FROM Table1
  WHERE column1 = 2
  WITH CHECK OPTION
```

And so on. But in future this too might become obsolete. It is already fairly easy to make stored procedures that handle the job.

If you want to do a materialization but don't want (or don't have the authority) to make a new view, you can do the job within one statement. For example, if this is your view:

```
CREATE VIEW View1 AS
  SELECT MAX(column1) AS view_column1
  FROM Table1
  GROUP BY column2
```

then instead of this:

```
SELECT AVG(view_column1)
  FROM View1
```
do this:
```
SELECT AVG(view_column1)
  FROM (SELECT MAX(column1) AS view_column1
  FROM Table1 GROUP BY column2) AS View1
```

In fact, this is so similar to using a view that many people call it a view —"inline view" is the common term — but in standard

SQL the correct term for [that thing that looks like a subquery in the FROM clause] is: table reference.

Tips

Over time, users of views have developed various "rules" that might make view use easier. The common ones are:

- Use default clauses when you create a table, so that views based on the table will more often be updatable.

- Include the table's primary key in the view's select list.

- Use a naming convention to mark non-updatable columns.

- Use the same naming convention for view names as you use for base table names. Alternatively, view names should begin with the name of the table upon which the view depends.

- [DB2] Document the view's purpose (security, efficiency, complexity hiding, alternate object terminology) in the view's REMARKS metadata.

- [SQL Server] Make an ordered view with a construct like this: CREATE VIEW ... SELECT TOP 100 PERCENT WITH TIES ... ORDER BY".

I would like to end with a recommendation about who has the best implementation of views, but in fact The Big Three are keeping up with each other feature by feature. Besides, I am no longer an unbiased observer.

References

Bello, Randall G., Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. "Materialized Views In Oracle."

(http://www.informatik.uni-trier.de/%7Eley/db/conf/vldb/BelloDDFNSWZ98.html)

- Very complete, for Oracle8.

Bobrowski, Steve. "Creating Updatable Views." http://www.oracle.com/oramag/oracle/01-mar/index.html?o21o8i.html

- An Oracle Magazine article tip set.

Burleson, Donald. "Dynamically create complex objects with Oracle materialized views." (Also at http://www.dba-oracle.com/art_9i_mv.htm.)

- A two-part article on syntax and practical employment.

Gulutzan, Peter and Trudy Pelzer. SQL Performance Tuning. Addison-Wesley 2003

Lewis, Jonathan. "Using in-line view for speed." (http://www.jlcomp.demon.co.uk/inline_1.html)

- An idea that COUNT(DISTINCT) in both the SELECT and the GROUP BY can be more efficient with inline views, on an older version of Oracle.

Mullins, Craig. "A View to a Kill." (http://dbazine.com/mullins_view.html)

- Advice to DBAs.

Rielau, Serge. "INSTEAD OF Triggers: All Views are updatable!"

(http://www7b.software.ibm.com/dmdd/library/techarticle/0
210rielau/0210rielau.html)

- INSTEAD OF triggers are in vogue among all DBMS
  vendors. This is the DB2 take.

"Migrating Oracle Databases to SQL Server 2000."
(http://www.akadia.com/services/sqlsrv2ora.html)

- This article includes a compact description of the
  differences between Oracle and Microsoft with respect to
  views.

"US 6,421,658 B1 - Efficient implementation of typed view
hierarchies for ORDBMS."
(http://www.uspto.gov/web/patents/patog/week29/OG/ht
ml/US06421658-20020716.html)

- An example of an IBM patent relating to views.

"Creating and Optimizing Views in SQL Server."
(http://www.informit.com/isapi/product_id%7E%7B4B34D
DF9-2147-41D0-8BB6-
7101176AD1F0%7D/st%7E%7B340C91CD-6221-4982-8F32-
4A0A9A8CF080%7D/content/index.asp)

- Includes some ideas for using INSTEAD OF triggers

Tip #41: "Restricting query by "ROWNUM" range (Type:
SQL)." (http://www.arrowsent.com/oratip/tip41.htm)

- One of many tip articles about the benefits of ROWNUM
  for limiting a query after the ORDER BY is over.

SQL JOIN

Relational Division

Dr. Codd defined a set of eight basic operators for his relational model. This series of articles looks at those basic operators in Standard SQL. Some are implemented directly, some require particular programming tricks and all of them have to be slightly modified to fit into the SQL language model.

Relational division is one of the eight basic operations in Codd's relational algebra. The idea is that a divisor table is used to partition a dividend table and produce a quotient or results table. The quotient table is made up of those values of one column for which a second column had all of the values in the divisor.

This is easier to explain with an example. We have a table of pilots and the planes they can fly (dividend); we have a table of planes in the hangar (divisor); we want the names of the pilots who can fly every plane (quotient) in the hangar. To get this result, we divide the PilotSkills table by the planes in the hangar.

```
CREATE TABLE PilotSkills
(pilot CHAR(15) NOT NULL,
 plane CHAR(15) NOT NULL,
 PRIMARY KEY (pilot, plane));

PilotSkills
pilot    plane
=========================
'Celko'   'Piper Cub'
'Higgins' 'B-52 Bomber'
'Higgins' 'F-14 Fighter'
'Higgins' 'Piper Cub'
'Jones'    'B-52 Bomber'
'Jones'    'F-14 Fighter'
'Smith'    'B-1 Bomber'
'Smith'    'B-52 Bomber'
'Smith'    'F-14 Fighter'
'Wilson'   'B-1 Bomber'
'Wilson'   'B-52 Bomber'
'Wilson'   'F-14 Fighter'
'Wilson'   'F-17 Fighter'

CREATE TABLE Hangar
(plane CHAR(15) NOT NULL PRIMARY KEY);

Hangar
plane
=============
'B-1 Bomber'
'B-52 Bomber'
'F-14 Fighter'

PilotSkills DIVIDED BY Hangar
pilot
=============================
'Smith'
'Wilson'
```

In this example, Smith and Wilson are the two pilots who can
fly everything in the hangar. Notice that Higgins and Celko
know how to fly a Piper Cub, but we don't have one right now.
In Codd's original definition of relational division, having more
rows than are called for is not a problem.

The important characteristic of a relational division is that the
CROSS JOIN (Cartesian product) of the divisor and the
quotient produces a valid subset of rows from the dividend.

This is where the name comes from, since the CROSS JOIN acts like a multiplication operator.

Relational division can be written as a single query, thus:

```
SELECT DISTINCT pilot
  FROM PilotSkills AS PS1
 WHERE NOT EXISTS
       (SELECT *
          FROM Hangar
         WHERE NOT EXISTS
               (SELECT *
                  FROM PilotSkills AS PS2
                 WHERE (PS1.pilot = PS2.pilot)
                   AND (PS2.plane = Hangar.plane)));
```

The quickest way to explain what is happening in this query is to imagine an old World War II movie where a cocky pilot has just walked into the hangar, looked over the fleet, and announced, "There ain't no plane in this hangar that I can't fly!", which is good logic, but horrible English.

We are finding the pilots for whom there does not exist a plane in the hangar for which they have no skills. The use of the NOT EXISTS() predicates is for speed. Most SQL systems will look up a value in an index rather than scan the whole table. This query for relational division was made popular by Chris Date in his textbooks, but it is not the only method, nor always the fastest. Another version of the division can be written so as to avoid three levels of nesting. While it is not original with me, I have made it popular in my books.

```
  SELECT PS1.pilot
    FROM PilotSkills AS PS1, Hangar AS H1
   WHERE PS1.plane = H1.plane
   GROUP BY PS1.pilot
 HAVING COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar);
```

There is a serious difference in the two methods. Burn down the hangar, so that the divisor is empty. Because of the NOT

EXISTS() predicates in Date's query, all pilots are returned from a division by an empty set. Because of the COUNT() functions in my query, no pilots are returned from a division by an empty set.

In the sixth edition of his book, *Introduction to Database Systems*, Chris Date defined another operator (DIVIDEBY ... PER) which produces the same results as my query, but with more complexity.

Another kind of relational division is exact relational division. The dividend table must match exactly to the values of the divisor without any extra values.

```
SELECT PS1.pilot
  FROM PilotSkills AS PS1
       LEFT OUTER JOIN
       Hangar AS H1
       ON PS1.plane = H1.plane
 GROUP BY PS1.pilot
HAVING COUNT(PS1.plane) = (SELECT COUNT(plane) FROM Hangar)
   AND COUNT(H1.plane) = (SELECT COUNT(plane) FROM Hangar);
```

This says that a pilot must have the same number of certificates as there planes in the hangar and these certificates all match to a plane in the hangar, not something else. The "something else" is shown by a created NULL from the LEFT OUTER JOIN.

Please do not make the mistake of trying to reduce the HAVING clause with a little algebra to:

```
 HAVING COUNT(PS1.plane) = COUNT(H1.plane)
```

because it does not work; it will tell you that the hangar has (n) planes in it and the pilot is certified for (n) planes, but not that those two sets of planes are equal to each other.

The Winter 1996 edition of *DB2 On-Line Magazine* (http://www.db2mag.com/db_area/archives/1996/q4/9601la r.shtml) had an article entitled "Powerful SQL: Beyond the Basics" by Sheryl Larsen that gave the results of testing both methods. Her conclusion for DB2 was that the nested EXISTS() version is better when the quotient has less than 25% of the dividend table's rows and the COUNT(*) version is better when the quotient is more than 25% of the dividend table.

SQL UNION

Set Operations

Introduction

SQL is a language that is supposed to be based on sets. Dr. Codd even defined the classic set operations as part of his eight basic operators for a relational database. Yet we did not have a full collection of basic set operations until the SQL-92 Standard.

By set operations, I mean union, intersection, and set difference -- the basic operators used in elementary set theory, which has been taught in the United States public school systems for decades.

Perhaps the problem in SQL that you did not have in pure set theory is that SQL tables are multisets (also called bags), which means that, unlike sets, they allow duplicate elements (rows or tuples). Dr. Codd's relational model is stricter and uses only true sets. SQL handles these duplicate rows with an ALL or DISTINCT modifier in different places in the language; ALL preserves duplicates and DISTINCT removes them.

Another more subtle problem is that set operations only make sense when the two sets are made up of the same kind of elements. In good database model, each table has one and only one type of elements. That is, you don't have more than one Inventory table, more than one Personnel table, etc.

But when the INCITS H2 (nee ANSI X3) Database Standards Committee added these operators, the model in the SQL-92 standard was to pair off the two tables on a row-per-row basis for set operations.

(note: In SQL-92, we introduced the shorthand TABLE <table name> for the query or subquery SELECT * FROM <table name>, which lets us refer to a table as a whole without referring to its columns. I will use this notation to save space)

Set Operations: Union

Microsoft introduced its ACCESS database product in 1992, after five years and tens of millions of dollars' worth of development work. The first complaints they got on their CompuServe user support forum involved the lack of a UNION operator. UNIONs are supported in SQL-86, SQL-89, and SQL-92, but the other set operations have to be constructed by the programmer in SQL-89. The syntax for the UNION statement is:

```
<query> UNION [ALL] <query>
```

Technically, this BNF is not right, but I will get back to that later. The UNION statement takes two tables and builds a new table from them. The two tables must be "union compatible", which means that they have the same number of columns, and that each column in the first table has the same datatype (or automatically cast to it) as the column in the same position in the second table.

That is, their rows have the same structure, so they can be put in the same final result table. Most implementations will do some datatype conversions to create the result table, but this is

very implementation-dependent and you should check it out for yourself.

What is interesting is that the result of a UNION has no name, and its columns are not named. If you want to have names, then you have to use an AS operator to create those names, thus.

```
((SELECT a, b, c FROM TableA WHERE city = 'Boston')
 UNION
 (SELECT x, y, z FROM TableB WHERE city = 'New York'))
 AS Cities (tom, dick, harry)
```

However, in actual products will find a multitude of other ways of doing this:

- The columns have the names of the first table in the UNION statement.

- The columns have the names of the last table in the UNION statement.

- The columns have the names generated by the SQL engine.

- The columns are referenced by a position number. This was the SQL-89 convention.

There are two forms of the UNION statement: the UNION and the UNION ALL. There was never a UNION DISTINCT option in the language. The UNION is the same operator you had in school; it returns the rows that appear in either or both tables and removes redundant duplicates from the result table.

In most older SQL implementations, this removal is done by merge-sorting the two tables and discarding duplicates during the merge. This has the side effect that the result table is sorted, but you cannot depend on that. This also explains why the ORDER BY clause is a common feature on UNION

operators. As long as the engine is sorting on all the columns anyway, why not let the programmer decide the sort keys?

The UNION ALL preserves the duplicates from both tables in the result table. In most implementations, this statement is done appending one table to the other, giving you a predictable ordering.

The UNION and UNION ALL operators are of the same precedence and are executed from left to right unless parentheses change the order.

In theory, the order of execution of UNIONs is not important, but it can be in practice. Even today, many optimizers generate separate results for each table expression in the UNION [ALL] first, then bring them together. And likewise, few products re-order the execution based on table sizes. Consider this expression:

```
(TABLE SmallTable1)
 UNION
(TABLE BigTable)
 UNION
(TABLE SmallTable2);
```

It will probably merge SmallTable1 into BigTable, then merge SmallTable2 into that first result. If the rows of SmallTable1 are spread out in the first result table, locating duplicates from SmallTable2 will take longer than if we had written the query thus:

```
(TABLE SmallTable1)
 UNION
(TABLE SmallTable2))
 UNION
(TABLE BigTable);
```

There are many reason that products lack UNION optimizations. First, UNIONs are not a common operation, so it is not worth the effort. And secondly, the order of execution becomes important if UNION and UNION ALL are mixed together:

```
 TABLE X
 UNION
 TABLE Y
 UNION ALL
 TABLE Z
```

Is executed as

```
(TABLE X UNION TABLE Y)
 UNION ALL
 TABLE Z
```

and that is not the same as:

```
 TABLE X
 UNION
 (TABLE Y UNION ALL TABLE Z)
```

Optimization of UNIONs is highly product-dependent, so you should experiment with it.

As a general statement, if you know that there are no duplicates, or that duplicates are not a problem in your situation, use the UNION ALL operator instead of UNION for speed.

There is no attempt to merge the table expressions and use OR-ed predicates. For example:

```
 SELECT * FROM Personnel WHERE sex = 'm'
 UNION ALL
 SELECT * FROM Personnel WHERE sex = 'f'
```

can be replaced by:

```
SELECT * FROM Personnel WHERE sex IN ('m', 'f');
```

A useful trick for building the union of different columns from the same table is to use a CROSS JOIN, a table of sequential integers from 1 thru (n) and a CASE expression, thus

```
SELECT employee,
   CASE WHEN S1.seq = 1 THEN P1.primary_lanuage
       WHEN S1.seq = 2 THEN P1.secondary_lanuage
       ELSE NULL END
 FROM Personnel AS F1
   CROSS JOIN
   Sequence AS S1
WHERE S1.seq IN (1,2)
```

This acts like the UNION ALL statement, but change the SELECT to SELECT DISTINCT and you have a UNION. The advantage of this statement over the more obvious UNION is that it makes one pass thru the table. Given a large table, that can be important for good performance.

SQL NULL

Selection

Introduction

Continuing the look at basic relational operators and SQL, we get to an operation with an unfortunate name: Selection. Selection removes rows from a table which do not pass a search condition. It is the counterpart of Projection, which removes columns from tables.

The reason the name is unfortunate is that SQL uses the keyword, SELECT, for the clause in a query that matches to the Projection operator and the keyword where for the clause in a query that matches to the selection operator. It is a little confusing, but just wait: things will get worse.

The search conditions are logical predicates -- things that return **TRUE**, **FALSE** or **UNKNOWN**. But wait a minute, most programming languages work with Boolean logic and have only TRUE and FALSE logical values. SQL and Codd's first relational model have a thing called a NULL and it makes things ... interesting.

The Null of It All

A NULL is not a value; it is a marker for a value that is missing. SQL does not know why the value is missing -- semantics is your job. But SQL does have syntax to handle NULLs. The basic rules are:

- NULLs propagate in calculations. That makes sense; if I don't know what something is, then why would I know what a calculation done with it is?

- NULLs return an UNKNOWN value in a logical expression. In fact, even (NULL = NULL) is UNKNOWN. Again, this makes sense. How can you tell one unknown

- NULLs group together. This property has nothing to do with simple search conditions, so don't worry about it for now; I will cover this point in another article on the GROUP BY clause later.

All of the SQL datatypes can use the basic comparison operators like equal (=), greater than (>), less than (<), not less than (>=), not greater than (<=) and not equal (<>). With the exception of the rules for NULLs, they behave pretty much as in every other programming language.

The logical operators are also familiar looking. They are AND, OR and NOT, and they are found in pretty much every other programming language. The gimmick is that these are three valued logical operators and not two valued ones.

The UNKNOWN value results from using NULLs in comparisons and other predicates, but UNKNOWN is a *logical* value and not the same as a NULL, which is a *data* value.

```
x NOT
==================
TRUE FALSE
UNK UNK
FALSE TRUE
AND | TRUE UNK FALSE
============================
TRUE | TRUE UNK FALSE
UNK | UNK UNK FALSE
FALSE | FALSE FALSE FALSE
OR | TRUE UNK FALSE
==========================
TRUE | TRUE TRUE TRUE
UNK | TRUE UNK UNK
FALSE | TRUE UNK FALSE
```

There is anther predicate of the form (x IS [NOT] NULL) in SQL that exits because you cannot use (x = NULL) to test for a NULL value. Almost all other predicates in SQL resolve themselves to chains of these three operators.

In the WHERE clause, the rows that test FALSE or UNKNOWN are removed from the table. Now, you are probably thinking that if we are going to treat FALSE and UNKNOWN alike, then why go to all the trouble to define a three-valued logic in the first place?

Defining a Three-valued Logic

SQL has three sub-languages: DML, DDL, and DCL. The Data Control Language (DCL) controls user access to the database and does not use predicates. In the Data Manipulation Language (DML), users can ask queries (SELECT statements) or change the data (INSERT INTO, UPDATE, and DELETE FROM statements). The Data Declaration Language (DDL) is where administrators control the schema objects like tables, views, stored procedures and so forth. The FALSE and UNKNOWN remove rows from the results of a query in the DML. In the DDL, a TRUE or UNKNOWN test result in a CHECK() constraint will preserve a row -- give it the benefit of

the doubt, so to speak. Otherwise, no column could be NULL-able.

Wonder Shorthands

SQL also came up with some wonder "shorthands" that improve the readability of the code. The logical operator "x BETWEEN y AND z" means "((y <= x) AND (x <= z))" -- note the order of comparison and the inclusion of the endpoints of the range. Likewise, "x IN (a,b,c,..)" expands out to "((x = a) OR (x = b) OR (x = c) OR ...)" at run time.

Most SQL engines are pretty good about optimizing the predicates and not that good about optimizing calculations. For example, the engine might not change (x + 0) or (x * 1) to (x) when they are compiling the code. This means that you need to write very clear logical expression with the simplest calculations in SQL.

Procedural languages like Fortran or Pascal are very good about optimizing calculations, which only makes sense because all they do is calculations! But SQL is a data retrieval language and the goal is to get back the right set of data as fast as possible from the secondary storage. Calculations are done at the speed of electricity, while data is retrieved by mechanical disk reads. The biggest improvements come from faster retrieval methods, not improved calculations.

Specifying Time

Killing Time

How long is a minute? If you said 60 seconds, you are technically wrong. It can vary from 59 to 61 seconds because of the leap second adjustment. This is the little adjustment that keeps the solar time aligned with the time calculated by an atomic clock. The Earth wobbles a little bit and it is not a precise as the atomic clock.

I am probably one of the few people who sets his wristwatch to the leap second. But a lot of networks, geopositioning satellites and other communications systems really have to worry about it.

Timing is Everything

The United States Naval Observatory sent out a questionnaire concerning the effects of a redefinition of Universal Coordinated Time (UTC) and runs a chat group at http://clockdev.usno.navy.mil/archives/leapsecs.html on the subject.

On 2000 July 2, they issued an "Abstract and Conclusions" on their e-mail survey to find possible adverse effects of a redefinition of UTC. They identified some possibly expensive or unsolvable problems with rewriting or checking software, which I will get to in a minute.

The big problem was the cost of redoing satellite systems software. UTC is commonly confused with the old Greenwich Mean Time and is computed by occasionally adding leap seconds to International Atomic Time (TAI). Since 1972, leap seconds have been added on December 31 or June 30, at the rate of about one every 18 months to keep atomic time in step with the Earth's rotation.

I would recommend that you use only TAI or UTC, since a man with two watches is never sure what time it really is.

But many major navigation systems such as GPS use constant offset from TAI internally. For example, GPs is 19 seconds off of TAI. There is a proposal in the international timing community to redefine UTC to avoid the discontinuities due to leap seconds. A discussion of the reasons for a change and what they might be has been published by McCarthy and Klepczynski in the "Innovations" section of the November 1999 issue of *GPs World* (you can get an abstract of the McCarthy and Klepczynski paper at http://www.findarticles.com/cf_0/m0BPW/11_10/57821998 /p1/article.jhtml).

The major reason they give for wanting to change the current system is to keep spread-spectrum communication systems and satellite navigation systems compatible with each other and with civil times. Another reason is the emerging need in the financial community to keep all computer time-stamps synchronized, which is where us database people need to start worrying about what we are doing on the Internet and communications networks.

If you do not add new leap seconds, solar time and atomic time will diverge at the rate of about 2 seconds every 3 years, and after about a century the difference would exceed 1 minute. Think of it as a Y2K problem on a smaller scale. Most commercial software assumes that UT1 is the same as UTC, or that the difference is always less than some value. If the difference is greater than that value, the software will have overflow problems. This would happen in NIST's WWV, WWVH and WWWB transmissions, which do not allow enough space for the difference to exceed 0.9 sec.

Specifying "Lawful Time"

Another problem is that some countries specify "lawful time" in terms of solar time, or GMT (Greenwich Mean Time, which has not existed for thirty years). Most nations on the Earth have learned to live with daylight savings time and moved from GMT to UTC. If you would like a history of the legal issues raised by past changes in time definition, get a copy of the book *Greenwich Time and Longitude* by Derek Howse.

Along the same lines, we survived Y2K, but nobody talks about what we learned from it. For a lot of companies, this was the first time anyone had looked at their legacy systems in years -- in decades, in fact. I think we can assume that any legacy system that was easy and cheap to replace was replaced. The next class of systems were those that we thought would be easy to patch, and on those systems, the Y2K staff went to work.

There was also a third class of software about which nobody knew anything, but that existed, nonetheless.

The side benefit of inspecting this class of programs was that while the programmers were fixing the date handling code, they

could also fix any other bad code they found. I do not know if anyone collected statistics on how much the non-temporal parts of the legacy systems were rewritten as part of the Y2K efforts.

Avoid Headaches with Preventive Maintenance

I would like to suggest that it would be a good idea to set up regular maintenance policies on legacy systems. After all, you schedule regular maintenance for your automobile. Vendors release new versions of your packaged software. But most companies use the, "If it's not broken, don't fix it!" policy instead.

I appreciate the fact that programmers have to develop new software, and have to try to keep the existing systems up and running by making repairs to the code that's known to be broken.

But how much trouble would be avoided if someone went to the database, looked at trends, and increased or changed things before they broke?

Preventive maintenance could be done to the to the database as well as to the source code. For example, imagine that every month the average length of a VARCHAR(n) column in a table is getting longer. Why not make the column's upper bound greater with an ALTER TABLE now to avoid future problems? On the other hand, could performance be improved by altering a column to a smaller sized datatype, say INTEGER to SMALLINT?

SQL TIMESTAMP datatype

Keeping Time

SQL is the first programming language to have explicit temporal datatypes. I have had the theory that if Cobol had been designed with a TIMESTAMP datatype, we would have avoided all that Y2K trouble. At least now, more people are aware of the ISO 8601 time and date display standards. Who knows? Maybe people will start to use them.

The temporal support in each SQL product can be classified as either a "Unix-style" or "Cobol-style" internal representation.

In the Unix-style representation, each point in time is shown as a very large integer number that represents the number of clock ticks from a base date. This is how the Unix operating system handles its temporal data. The use of clock ticks makes calculations very easy — it becomes simple integer math. However, it is hard to convert the clock ticks into a year-month-day-hour-minute-second format.

In the Cobol-style representation, the database has a separate internal field for the year, month, day, hour, minute, and seconds. This is great for displaying the information, but not for calculations.

One of the debates in the SQL Standards Committee was how to handle intervals of time. The reason that time is tricky is that it is continuous. The defining mathematical property of a

continuum is that any part of it can be further sub-divided forever. Give me any line segment and I can cut it into smaller segments endlessly. But we run into the problem that the defining property of a point is that it cannot be further subdivided. So how can there be points in a continuum?

When you give a year, say 2000, you are really giving me an interval of 365 days. Give me a date, say 2000-01-01, you are not giving me a point; you are identifying an interval of 24 hours. Give me the date and time 2000-01-01 00:00:00 and you are giving me an interval of 60 seconds. It never stops!!

The decision in SQL was to view time as a series of open ended intervals. That is, the segment includes the starting point in time, but never gets to the end point of the interval. This has some nice properties. It prevents you from counting the end of one event and the start of another event as identical moments in time. An open interval minus an open interval gives open intervals as a result and all points are accounted for.

But intervals are hard to work with conceptually. Let me give you an actual example that was posted in a newsgroup. We have a table that catches information about the user activity on a system. It is a very simple "log file" that shows when someone starts and ends a session with the system. We do not even care who the user was, since I am assuming that user_activity_id is a unique number that identifies a session, without identifying individual users. The table looks like this:

```
CREATE TABLE User_Activity
(user_activity_id INTEGER NOT NULL PRIMARY KEY,
 login TIMESTAMP NOT NULL,
 logout TIMESTAMP, -- null means session is still active
 CHECK (login < logout),
 ...);
```

Using a NULL in the logout column to mean that the session is still active adds a little complexity to the problem. I decided to use the current timestamp at the time the query is executed as the logout time.

I would like to be able to report the number of user sessions logged on during each hour of the day. So, if someone began a session at 03:12 Hrs and ended it at 06:45 Hrs, I would like them to be counted as being logged on the system for 03:00 Hrs, 04:00 Hrs, 05:00 Hrs and 06:00 Hrs. This report should work all the hours in several years of data.

One solution proposed in the newsgroup involved using CASE expressions to classify each time extracted from the TIMESTAMP values as to what hourly interval it belongs. The logic got worse from there.

Here is one solution: first, create an auxiliary table like this:

```
CREATE TABLE HourlyReport
(period_nbr INTEGER NOT NULL PRIMARY KEY,
 start_timestamp TIMESTAMP NOT NULL,
 end_timestamp TIMESTAMP NOT NULL,
 CHECK(start_time < end_time));
INSERT INTO HourlyReport
VALUES (1, '1999-01-01 00:00:00.00000',
'1999-01-01 00:59:59.99999');
INSERT INTO HourlyReport
VALUES (2, '1999-01-01 01:00:00.00000',
'1999-01-01 01:59:59.99999');
etc.
```

Before you reject this auxiliary table, notice that it is easy to generate and will be (24 hours per day * 365.25 days per year * 10 years) = 87660 rows in size if you want to handle an entire decade of data.

The query to find the periods in which each activity falls is simply:

```
SELECT DISTINCT A1.user_activity_id, period_nbr
 FROM User_Activity AS A1,
      HourlyReports AS H1
 WHERE H1.start_timestamp BETWEEN A1.login
               AND COALESCE A1.logout,CURRENT_TIMESTAMP)
   OR H1.end_timestamp BETWEEN A1.login
               AND COALESCE A1.logout, CURRENT_TIMESTAMP);
```

Notice the DISTINCT! Without it, you would count both the start and end times of each period. Now, to answer the original question, tally by periods:

```
SELECT A1.period_nbr, A1.start_timestamp,
       COUNT (DISTINCT A1.user_activity_id)
         AS total_sessions
 FROM User_Activity AS A1,
      HourlyReports AS H1
 WHERE H1.start_timestamp BETWEEN A1.login
               AND COALESCE A1.logout, CURRENT_TIMESTAMP)
   OR H1.end_timestamp BETWEEN A1.login
               AND COALESCE A1.logout, CURRENT_TIMESTAMP)
 GROUP BY A1.period_nbr, A1.start_timestamp;
```

It might help if you drew a diagram with a time line, then put in a session as a line segment which crosses the borders between the time periods.

```
session  X------------------X
     -|-----|-----|-----|-----|-----|--
period 2 3 4 5 6
```

Instead of trying to put the session into the periods, this query puts the starts and stops of the periods into the session interval. A period can have a start time, a stop time or both inside the session; this case is why you need to remove the duplicate period numbers.

Internals of IDENTITY datatype Column

The Ghost of Sequential Processing

When we were first creating relational database products, we really did not understand at a fundamental level what we were doing. As a result, we made a lot of mistakes then and have to live with them now. The biggest mistakes come from exposing the physical representation of the logical model to the programmer.

This is a holdover from the early programming language while we were *very* close to the hardware. For example, the fields in a COBOL or FORTRAN program were assumed to be physically located in the order in which they were declared. This meant that you could define a template that overlaid the same physical space and read the representation in several different ways. In COBOL, the command was REDEFINES, EQUIVALENCE in FORTRAN and a union in 'C.'

From a logical viewpoint, this redefinition makes no sense at all. It is confusing the numeral with the number that the numeral represents.

Early SQL and Contiguous Storage

The early SQLs were based on existing file systems. The data was kept in physically contiguous disk pages, in physically contiguous rows, made up of physically contiguous columns — in short, just like a deck of punch cards or a magnetic tape. You

located data by counting its position in the deck, starting at the front.

Physically contiguous storage is only one way of building a relational database and it is not always the best one. But aside from that, the whole idea of a relational database is that user is not supposed to know how things are stored at all, much less write code that depends on the particular physical representation in a particular release of a particular product.

One significant error is the IDENTITY column in the Sybase family (SQL Server and Sybase). If you are not familiar with this "feature," it is assigned to a column as its data type with the limitation that a table can have only one such column. The database engine assigns a sequential integer in this column to every row in the table as it is inserted.

People actually program with this "feature" and even use it as the primary key for the table! Now, let's go into painful details as to why this thing is bad.

IDENTITY Crisis

The practical considerations are that IDENTITY is proprietary and non-portable, so you know that you will have maintenance problems when you change releases or products. It also has some very strange bugs in both Sybase and SQL Server.

But let's look at the logical problems. First, try to create a table with two columns and try to make them both IDENTITY columns. If you cannot declare more than one column to be of a certain datatype, then that thing is not a datatype at all, by definition.

Next, create a table with one column and make it an IDENTITY column. Now try to insert, update and delete different numbers from it. If you cannot insert, update and delete rows from a table, then it is not a table by definition.

Finally, create a simple table with one IDENTITY column and a few other columns. Use a few statements like

```
INSERT INTO Foobar (a, b, c) VALUES ('a1', 'b1', 'c1');
INSERT INTO Foobar (a, b, c) VALUES ('a2', 'b2', 'c2');
INSERT INTO Foobar (a, b, c) VALUES ('a3', 'b3', 'c3');
```

to put a few rows into the table and notice that the IDENTITY column sequentially numbered them in the order in which they were presented. If you delete a row, the gap in the sequence is not filled in, and the sequence continues from the highest number that has ever been used in that column in that particular table.

But now use a statement with a query expression in it, like this:

```
INSERT INTO Foobar (a, b, c)
SELECT x, y, z
FROM Floob;
```

Since a query result is a table, and a table is a set that has no ordering, what should the IDENTITY numbers be? The entire, whole, completed set is presented to Foobar all at once, not a row at a time. There are (n!) ways to number (n) rows, so which one do you pick? The answer has been to use whatever the physical order of the result set happened to be — that non-relational phrase, "physical order" again. But it is actually worse than that. If the same query is executed again, but with new statistics or after an index has been dropped or added, the new execution plan could bring the result set back in a different physical order.

Oh, why did duplicate rows in the second query get different IDENTITY numbers? In the relational model, they should be treated the same if all the values of all the attributes are identical.

There are better ways of creating identifiers, but that is the subject for another column. In the meantime, stop writing bad code, until I can teach you how to write good code.

Keyword Search Queries

Keyword Searches

Here is a short problem that you might like to play with. You are given a table with a document number and a keyword that someone extracted as descriptive of that document. This is the way that many professional organizations access journal articles. We can declare a simple version of this table.

```
CREATE TABLE Documents
(document_id INTEGER NOT NULL,
 key_word VARCHAR(25) NOT NULL,
 PRIMARY KEY (document_id, key_word));
```

Your assignment is to write a general searching query in SQL. You are given a list of words that the document must have and a list of words which the document must NOT have.

We need a table for the list of words which we want to find:

```
CREATE TABLE SearchList
(word VARCHAR(25) NOT NULL PRIMARY KEY);
```

And we need another table for the words that will exclude a document.

```
CREATE TABLE ExcludeList
(word VARCHAR(25) NOT NULL PRIMARY KEY);
```

Breaking the problem down into two parts, excluding a document is easy.

```
 CREATE TABLE ExcludeList
(word VARCHAR(25) NOT NULL PRIMARY KEY);
```

Breaking the problem down into two parts, excluding a document is easy.

```
SELECT DISTINCT document_id
  FROM Documents AS D1
 WHERE NOT EXISTS
       (SELECT *
          FROM ExcludeList AS E1
         WHERE E1.word = D1.key_word);
```

This says that you want only the documents that have no matches in the excluded word list. You might want to make the WHERE clause in the subquery expression more general by using a LIKE predicate or similar expression, like this.

```
WHERE E1.word LIKE D1.key_word || '%'
   OR E1.word LIKE '%' || D1.key_word
   OR D1.key_word LIKE E1.word || '%'
   OR D1.key_word LIKE '%' || E1.word
```

This would give you a very forgiving matching criteria. That is not a good idea when you are excluding documents. When you wanted to get rid "Smith" is does not follow that you also wanted to get rid of "Smithsonian" as well.

For this example, Let you agree that equality is the right matching criteria, to keep the code simple.

Put that solution aside for a minute and move on to the other part of the problem; finding documents that have all the words you have in your search list.

The first attempt to combine both of these queries is:

```
SELECT D1.document_id
  FROM Documents AS D1
 WHERE EXISTS
       (SELECT *
```

```
          FROM SearchList AS S1
        WHERE S1.word = D1.key_word);
    AND NOT EXISTS
        (SELECT *
          FROM ExcludeList AS E1
        WHERE E1.word = D1.key_word);
```

This answer is wrong. It will pick documents with any search word, not all search words. It does remove a document when it finds any of the exclude words. What do you do when a word is in both the search and the exclude lists? This predicate has made the decision that exclusion overrides the search list. The is probably reasonable, but it was not in the specifications. Another thing the specification did not tell us is what happens when a document has all the search words and some extras? Do we look only for an exact match, or can a document have more keywords?

Fortunately, the operation of picking the documents that contain all the search words is known as Relational Division. It was one of the original operators that Ted Codd proposed in his papers on relational database theory. Here is one way to code this operation in SQL.

```
SELECT D1.document_id
  FROM Documents AS D1, SearchList AS S1
 WHERE D1.key_word = S1.word
 GROUP BY D1.document_id
HAVING COUNT(D1.word)
      >= (SELECT COUNT(word) FROM SearchList);
```

What this does is map the search list to the document's key word list and if the search list is the same size as the mapping, you have a match. If you need a mental model of what is happening, imagine that a librarian is sticking Post-It notes on the documents that have each search word. When she has used all of the Post-It notes on one document, it is a match. If you want an exact match, change the >= to = in the HAVING clause.

Now we are ready to combine the two lists into one query. This will remove a document which contains any exclude word and accept a document with all (or more) of the search words.

```
SELECT D1.document_id
  FROM Documents AS D1, SearchList AS S1
 WHERE D1.key_word = S1.word
   AND NOT EXISTS
       (SELECT *
          FROM ExcludeList AS E1
         WHERE E1.word = D1.key_word)
 GROUP BY D1.document_id
HAVING COUNT(D1.word)
       >= (SELECT COUNT(word)
            FROM SearchList);
```

The trick is in seeing that there is an order of execution to the steps in process. If the exclude list is long, then this will filter out a lot of documents before doing the GROUP BY and the relational division.

The Cost of Calculated Columns

Calculated Columns

Introduction

You are not supposed to put a calculated column in a table in a pure SQL database. And as the guardian of pure SQL, I should oppose this practice. Too bad the real world is not as nice as the theoretical world.

There are many types of calculated columns. The first are columns which derive their values from outside the database itself. The most common examples are timestamps, user identifiers, and other values generated by the system or the application program. This type of calculated column is fine and presents no problems for the database.

The second type is values calculated from columns in the same row. In the days when we used punch cards, you would take a deck of cards, run them thru a machine that would do the multiplications and addition, then punch the results in the right hand side of the cards. For example, the total cost of a line in an order could be described as price times quantity.

The reason for this calculation was simple; the machines that processed punch cards had no secondary storage, so the data had to be kept on the cards themselves. There is truly no reason for doing this today; it is much faster to re-calculate the data than it is to read the results from secondary storage.

The third type of calculated data uses data in the same table, but not always in the same row in which it will appear. The fourth type uses data in the same database.

These last two types are used when the cost of the calculation is higher than the cost of a simple read. In particular, data warehouses love to have this type of data in them to save time.

When and how you do something is important in SQL. Here is an example, based on a thread in a SQL Server discussion group. I am changing the table around a bit, and not telling you the names of the guilty parties involved, but the idea still holds. You are given a table that look like this and you need to calculate a column based on the value in another row of the same table.

```
CREATE TABLE StockHistory
 (stock_id CHAR(5) NOT NULL,
  sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
  price DECIMAL (10,4) NOT NULL,
  trend INTEGER NOT NULL DEFAULT 0
        CHECK(trend IN(-1, 0, 1))
  PRIMARY KEY (stock_id, sale_date));
```

It records the final selling price of many different stocks. The trend column is +1 if the price increased from the last reported selling price, 0 if it stayed the same and -1 if it dropped in price. The trend column is the problem, not because it is hard to compute, but because it can be done several different ways. Let's look at the methods for doing this calculation.

Triggers

You can write a trigger which will fire after the new row is inserted. While there is an ISO Standard SQL/PSM language for writing triggers, the truth is that every vendor has a

proprietary trigger language and they are not compatible. In fact, you will find many different features from product to product and totally different underlying data models. If you decide to use triggers, you will be using proprietary, non-relational code and have to deal with several problems.

One problem is what a trigger does with a bulk insertion. Given this statement which inserts two rows at the same time:

```
INSERT INTO StockHistory (stock_id, sale_date, price)
VALUES ('XXX', '2000-04-01',  10.75),
       ('XXX', '2000-04-03', 200.00);
```

Trend will be set to zero in both of these new rows using the DEFAULT clause. But can the trigger see these rows and figure out that the 2000 April 03 row should have a +1 trend or not? Maybe or maybe not, because the new rows are not always committed before the trigger is fired. Also, what should that status of the 2000 April 01 row be? That depends on an already existing row in the table.

But assume that the trigger worked correctly. Now, what if you get this statement?

```
INSERT INTO StockHistory (stock_id, sale_date, price)
VALUES ('XXX', '2000-04-02', 313.25);
```

Did your trigger change the trend in the 2000 April 03 row or not? If I drop a row, does your trigger change the trend in the affected rows? Probably not.

As an exercise, write some trigger code for this problem.

INSERT INTO Statement

I admit I am showing off a bit, but here is one way of inserting data one row at a time. Let me put the statement into a stored procedure.

```
CREATE PROCEDURE NewStockSale
 (new_stock_id CHAR(5) NOT NULL,
  new_sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
  new_price DECIMAL (10,4) NOT NULL)
AS INSERT INTO
   StockHistory (stock_id, sale_date, price, trend)
   VALUES (new_stock_id, new_sale_date, new_price,
          SIGN(new_price -
               (SELECT H1.price
                  FROM StockHistory AS H1
                 WHERE H1.stock_id = StockHistory.stock_id
                   AND H1.sale_date =
                       (SELECT MAX(sale_date)
                          FROM StockHistory AS H2
                         WHERE H2.stock_id = H1.stock_id
                           AND H2.sale_date < H1.sale_date)
                ))) AS trend
          );
```

This is not as bad as you first think. The innermost subquery finds the sale just before the current sale, then returns its price. If the old price minus the new price is positive negative or zero, the SIGN() function can computer the value of TREND. Yes, I was showing off a little bit with this query.

The problem with this is much the same as the triggers. What if I delete a row or add a new row between two existing rows? This statement will not do a thing about changing the other rows.

But there is another problem; this stored procedure is good for only one row at a time. That would mean that at the end of the business day, I would have to write a loop that put one row at a time into the StockHistory table.

Your next exercise is to improve this stored procedure.

UPDATE the Table

You already have a default value of 0 in the trend column, so you could just write an UPDATE statement based on the same logic we have been using.

```
UPDATE StockHistory
   SET trend
       = SIGN(price -
            (SELECT H1.price
               FROM StockHistory AS H1
              WHERE H1.stock_id = StockHistory.stock_id
                AND H1.sale_date =
                    (SELECT MAX(sale_date)
                       FROM StockHistory AS H2
                      WHERE H2.stock_id = H1.stock_id
                        AND H2.sale_date < H1.sale_date)));
```

While this statement does the job, it will re-calculate trend column for the entire table. What if we only looked at the columns that had a zero? Better yet, what if we made the trend column NULL-able and used the NULLs as a way to locate the rows that need the updates?

```
UPDATE StockHistory
   SET trend = ...
 WHERE trend IS NULL;
```

But this does not solve the problem of inserting a row between two existing dates. Fixing that problem is your third exercise.

Use a VIEW

This approach will involve getting rid of the trend column in the StockHistory table and creating a VIEW on the remaining columns:

```
  CREATE TABLE StockHistory
 (stock_id CHAR(5) NOT NULL,
  sale_date DATE NOT NULL DEFAULT CURRENT_DATE,
```

```
  price DECIMAL (10,4) NOT NULL,
  PRIMARY KEY (stock_id, sale_date));

 CREATE VIEW StockTrends (stock_id, sale_date, price, trend)
 AS SELECT H1.stock_id, H1.sale_date, H1.price,
           SIGN(MAX(H2.price) - H1.price)
     FROM StockHistory AS H1 StockHistory AS H2
    WHERE H1.stock_id = H2.stock_id
      AND H2.sale_date < H1.sale_date
    GROUP BY H1.stock_id, H1.sale_date, H1.price;
```

This approach will handle the insertion and deletion of any number of rows, in any order. The trend column will be computed from the existing data each time. The primary key is also a covering index for the query, which helps performance. A covering index is one which contains all of the columns used the WHERE clause of a query.

The major objection to this approach is that the VIEW can be slow to build each time, if StockHistory is a large table.

I will send a free book to the reader who submits the best answers top these exercises. You can contact me at 71062.1056@compuserve.com or you can go to my website at www.celko.com.

Graphs in SQL

Path Finder

I got an email asking me how to find paths in a graph using SQL. The author of the email had seen my chapter on graphs in *SQL for Smarties*, and read that I was not happy with my own answers. What he wanted was a list of paths from any two nodes in a directed graph, and I would assume that he wanted the cheapest path.

After thinking about this for a while, the best way is probably to do the Floyd-Warshall or Johnson algorithm in a procedural language and load a table with the results. But I want to do this in pure SQL as an exercise.

Let's start with a simple graph and represent it as an adjacency list with weights on the edges.

```
CREATE TABLE Graph
(source CHAR(2) NOT NULL,
 destination CHAR(2) NOT NULL,
 cost INTEGER NOT NULL,
 PRIMARY KEY (source, destination));
```

I got data for this table from the book *Introduction to Algorithms* by Cormen, Leiserson and Rivest (ISBN 0-262-03141-8), page 518. This book is very popular in college courses in the United States. I made one decision that will be important later; I added self-traversal edges (i.e., the node is both the source and the destination) with weights of zero.

```
INSERT INTO Graph VALUES ('s', 's', 0);
INSERT INTO Graph VALUES ('s', 'u', 3);
INSERT INTO Graph VALUES ('s', 'x', 5);
INSERT INTO Graph VALUES ('u', 'u', 0);
INSERT INTO Graph VALUES ('u', 'v', 6);
INSERT INTO Graph VALUES ('u', 'x', 2);
INSERT INTO Graph VALUES ('v', 'v', 0);
INSERT INTO Graph VALUES ('v', 'y', 2);
INSERT INTO Graph VALUES ('x', 'u', 1);
INSERT INTO Graph VALUES ('x', 'v', 4);
INSERT INTO Graph VALUES ('x', 'x', 0);
INSERT INTO Graph VALUES ('x', 'y', 6);
INSERT INTO Graph VALUES ('y', 's', 3);
INSERT INTO Graph VALUES ('y', 'v', 7);
INSERT INTO Graph VALUES ('y', 'y', 0);
```

I am not happy about this approach, because I have to decide the maximum number of edges in path before I start looking for an answer. But this will work and I know that a path will have no more than the total number of nodes in the graph. Let's create a table to hold the paths:

```
CREATE TABLE Paths
(step1 CHAR(2) NOT NULL,
 step2 CHAR(2) NOT NULL,
 step3 CHAR(2) NOT NULL,
 step4 CHAR(2) NOT NULL,
 step5 CHAR(2) NOT NULL,
 total_cost INTEGER NOT NULL,
 path_length INTEGER NOT NULL,
 PRIMARY KEY (step1, step2, step3, step4, step5));
```

The step1 node is where I begin the path. The other columns are the second step, third step, fourth step, and so forth. The last step column is the end of the journey. The total_cost column is the total cost, based on the sum of the weights of the edges, on this path. The path length column is harder to explain, but for now, let's just say that it is a count of the nodes visited in the path.

To keep things easier, let's look at all the paths from "s" to "y" in the graph. The INSERT INTO statement for construction that set looks like this:

```
INSERT INTO Paths
SELECT  G1.source, -- it is 's' in this example
        G2.source,
        G3.source,
        G4.source,
        G4.destination, -- it is 'y' in this example
    (G1.cost + G2.cost + G3.cost + G4.cost),
    (CASE WHEN G1.source NOT IN (G2.source, G3.source, G4.source)
          THEN 1 ELSE 0 END
     + CASE WHEN G2.source NOT IN (G1.source, G3.source, G4.source)
            THEN 1 ELSE 0 END
     + CASE WHEN G3.source NOT IN (G1.source, G2.source, G4.source)
            THEN 1 ELSE 0 END
     + CASE WHEN G4.source NOT IN (G1.source, G2.source, G3.source)
            THEN 1 ELSE 0 END)
  FROM Graph AS G1,
       Graph AS G2,
       Graph AS G3,
       Graph AS G4
 WHERE G1.source = 's'
   AND G1.destination = G2.source
   AND G2.destination = G3.source
   AND G3.destination = G4.source
   AND G4.destination = 'y';
```

I put in "s" and "y" as the source and destination of the path, and made sure that the destination of one step in the path was the source of the next step in the path. This is a combinatorial explosion, but it is easy to read and understand.

The sum of the weights is the cost of the path, which is easy to understand. The path_length calculation is a bit harder. This sum of CASE expressions looks at each node in the path. If it is unique within the row, it is assigned a value of one, if it is not unique within the row, it is assigned a value of zero.

All paths will have five steps in them because that is the way the table is declared. But what if a path exists between the two nodes which is shorter than five steps? That is where the self-traversal rows are used! Consecutive pairs of steps in the same row can be repetitions of the same node.

Here is what the rows of the Paths table look like after this INSERT INTO statement, ordered by descending path_length, and then by ascending cost.

```
Paths
step1 step2 step3 step4 step5 total_cost path_length
========================================================
s     s     x     x     y     11          0
s     s     s     x     y     11          1
s     x     x     x     y     11          1
s     x     u     x     y     14          2
s     s     u     v     y     11          2
s     s     u     x     y     11          2
s     s     x     v     y     11          2
s     s     x     y     y     11          2
s     u     u     v     y     11          2
s     u     u     x     y     11          2
s     u     v     v     y     11          2
s     u     x     x     y     11          2
s     x     v     v     y     11          2
s     x     x     v     y     11          2
s     x     x     y     y     11          2
s     x     y     y     y     11          2
s     x     y     v     y     20          4
s     x     u     v     y     14          4
s     u     v     y     y     11          4
s     u     x     v     y     11          4
s     u     x     y     y     11          4
s     x     v     y     y     11          4
```

Clearly, all pairs of nodes could be picked from the original Graph table and the same INSERT INTO run on them with a minor change in the WHERE clause. However, this example is big enough for a short magazine article. And it is too big for most applications. It is safe to assume that people really want the cheapest path. In this example, the total_cost column defines the cost of a path, so we can eliminate some of the paths from the Paths table with this statement.

```
DELETE FROM Paths
WHERE total_cost
    > (SELECT MIN(total_cost)
        FROM Paths);
```

Again, if you had all the paths for all possible pairs of nodes, the subquery expression would have a WHERE clause to correlate it to the subset of paths for each possible pair.

In this example, it got rid of 3 out of 22 possible paths. It is helpful and in some situations we might like having all the options. But these are not distinct options.

As one of many examples, the paths

```
(s, x, v, v, y, 11, 2)
```

and

```
(s, x, x, v, y, 11, 2)
```

are both really the same path, (s, x, v, y). Before we decide to write a statement to handle these equivalent rows, let's consider another cost factor. People do not like to change airplanes or trains. If they can go from Amsterdam to New York City on one plane without changing planes for the same cost, they are happy. This is where that path_length column comes in. It is a quick way to remove the paths that have more edges than they need to get the job done.

```
DELETE FROM Paths
 WHERE path_length
       > (SELECT MIN(path_length)
            FROM Paths);
```

In this case, that last DELETE FROM statement will reduce the table to one row: (s, s, x, x, y, 11, 0) which reduces to (s, x, y). This single remaining row is very convenient for my article, but if you look at the table, you will see that there was also a subset of equivalent rows that had higher path_length numbers.

```
(s, s, s, x, y, 11, 1)
(s, x, x, x, y, 11, 1)
(s, x, x, y, y, 11, 2)
(s, x, y, y, y, 11, 2)
```

Your task is to write code to handle equivalent rows. Hint: the duplicate nodes will always be contiguous across the row.

Finding the Gap in a Range

Filling in the Gaps

As I get older, I am convinced that there really is no such animal as a simple programming problem. Oh, they might look simple when you start but that is just a trick. Under the covers, are all kinds of devils just waiting to get out.

Darren Taft posted what seems like an easy problem on the SQL Server newsgroup in 2000 October. Let me quote him: "I have an ordering system that allocates numbers within predefined ranges. I do this at the moment using this: ..." At this point, he posted a stored procedure written in T-SQL dialect. This procedure had a loop that incremented the request_id number in a loop until it either found a gap in the numbering or failed. Mr. Taft then continued: "This is fine for the first few numbers, but when the ranges are anything up to 10,000 between the minimum and the maximum, it starts to get a little slow. Can anyone think of a better way of doing this?

Basically it needs to find the next number within the range for which there isn't a row in the Requests table (the primary key is the request_id, which is an integer column with a clustered index). Rows can be deleted from within the range, so the next number will not always be the current maximum plus one."

Before you go further, try to write a procedural solution yourself. Now, put down your pencils and start reading again. As an aside, the original stored procedure was wrong because it

did not test for an upper bound. If the range was completely used, the stored procedure would return the upper limit plus one.

Graham Shaw immediately proposed this query:

```
SELECT MIN (R1.request_id        + 1)
  FROM Requests AS R1
       LEFT OUTER JOIN
       Requests AS R2
       ON R1.request_id + 1 = R2.request_id
  WHERE R2.request_id IS NULL;
```

The idea is that there is a leftmost value in the Requests table just before a gap. Therefore, when (request_nbr +1) is not in the table, we have found a gap. This is what the incremental approach in the stored procedure was doing, one row at a time.

Too bad this does not work. First of all, there is no checking for an upper bound. In effect, the flaw in the original stored procedure has become part of the specification! This is like the story about the Englishman who sent a favorite old jacket to a Chinese tailor and told him to make an exact copy of it in heavy silk. The tailor did exactly that, right down to the cigarette burns, stains and frayed elbows. The second problem is that you cannot get the first position in the range if it is the only one vacant.

Umachandar Jayachandranm, another regular to the newsgroup, saw that the OUTER JOIN should be expensive and suggested that Darren try this query:

```
 SELECT MIN(R1.request_id) + 1
  FROM Requests AS R1
WHERE NOT EXISTS
      (SELECT *
         FROM Requests AS R2
        WHERE R2.request_id = R1.request_id + 1
          AND R2.request_id >= {{low range boundary}})
```

```
        AND R1.request_id >= {{low range boundary}}
```

He also proposed a proprietary solution based on the TOP(n) operator in SQL Server, but I will not go into that answer. But again, this answer has the same two flaws as before.

I agreed with Umachandar that the OUTER JOIN solution was needlessly complex. I proposed a more set-oriented solution in the form of a VIEW of the all gaps in the numbering, instead. That query looked like this:

```
CREATE VIEW Gaps (gap_start, gap_end)
AS SELECT DISTINCT R1.request_id + 1, MIN(R2.request_id -1)
     FROM Requests AS R1,
          Requests AS R2
    WHERE R1.request_id <= R2.request_id
      AND R1.request_id + 1
      NOT IN (SELECT request_id FROM Requests)
      AND R2.request_id - 1
      NOT IN (SELECT request_id FROM Requests)
      AND R1.request_id + 1 <= {{high range boundary}}
      AND R2.request_id - 1 >= {{low range boundary}}
    GROUP BY R1.request_id;
```

I was happy with this answer, since it found all the desired numbers and solved the problems at the extremes of the range. By using the plus and minus one, I am finding the gaps from both their left and right sides, so I will catch an open slot in both the high and low range boundaries. The only improvement I found was that you might want to change the NOT IN () predicates to NOT EXISTS() predicates for performance in some SQL products. You can also use this view to get reports on the density of allocated numbers, use it to compress the gaps, to insert new requests in a well distributed manner, and so on.

I was proud of myself until Darren replied, "Interesting response, but it doesn't actually provide the answer. I would need a further query on the view to get what I want. This view

actually runs slower than the OUTER JOIN suggestion, so with a query on top of that, it has to be the slowest answer so far." He did concede that the query is handy for analyzing gaps and that he would keep it for future reference. That helped my wounded ego a little bit.

So it was time to do more thinking about the boundary problems and how to return only one number. I finally came up with this nightmare query:

```
SELECT MIN (X.request_id)
  FROM (SELECT (CASE WHEN (R1.request_id + 1)
                          NOT IN (SELECT request_id
                                    FROM Requests)
                     THEN (R1.request_id + 1)
                     WHEN (R1.request_id - 1)
                          NOT IN (SELECT request_id
                                    FROM Requests)
                     THEN (R1.request_id - 1)
                     ELSE NULL END)
 FROM Requests AS R1
 WHERE R1.request_id + 1
       BETWEEN {low range boundary} AND {high range boundary}
   AND R1.request_id - 1
       BETWEEN {low range boundary} AND {high range boundary}
 GROUP BY R1.request_id) AS X(request_id);
```

The outermost query is simply returning the first number in the derived query. The derived query, X, finds gaps from both the left and the right sides by incrementing and decrementing values in the Requests table. It also does a range check in the WHERE clause. The real trick is in the CASE expression; when a gap exists to the right of a number, return it; when a gap exists to the left of a number, return it; when there are no gaps, return a NULL. This will solve the boundary problem at the extremes of the range. It might be ugly, but at least it works!

There is also a subtle third problem here. All these approaches tend to favor picking a new request_id value in the lower end

of the range. The clustered B-tree index would have to be re-balanced more often than if you were to pick new request_id numbers randomly from the possible values in the gaps. The table will be reorganized more than you would really wish it to be.

For a situation with a great number of transactions, the real trick is to replace the clustered index with an unclustered index.

SQL and the Web

Web Databases

An American thinks that 100 years is a long time; a European thinks that 100 miles is a long trip. How you see the world is relative to your environment and your experience. We are starting to see the same thing happen in databases, too.

The first fight has long since been over and SQL won the battle for a standard database language. However, if you look at the actual figures, only 12 percent of the world's data is in SQL databases. If a few weeks is supposed to be an "Internet Year," then why is it taking so long to convert legacy data to SQL? The simple truth is that you could probably pick any legacy system and move its data to SQL in a week or less. The trouble is that it would require years, maybe decades, to convert the legacy applications code to a language that could use the SQL database. This is not a good way to run a business.

The trend over the past several years is to do new work with an SQL product, and try to interface to the legacy systems for any needed data until you can kill the old system. There are any number of products that will make an IMS, IDMS, TOTAL, or flat file system look like a set of SQL tables (note to younger readers: if you do not know what those products are, look around your shop and ask the programmer who is still using a slide ruler instead of a calculator).

We were comfortable with this situation. In most business reporting programs, you write a preamble to set up the report,

a loop that goes over a cursor, and a post-amble to do the house cleaning. The hard part is getting the query in the cursor just right. What you want is to make the result set from the query look as if it were a very simple sequential file that had all the data required, already sorted in the right order for the report.

Years ago, a co-worker of mine defined the Law of Conservation of Difficulty. Every system has a minimum degree of difficulty, and you cannot put out less effort than is required to overcome that degree of difficulty to solve the problem. You can put out more effort, to be sure, but never less effort. What SQL did was sweep all the difficulty out of the host language and concentrate it in the queries. This situation was fine, and life was good. Then along came the Internet. There are a lot of other trends that are changing the way we look at databases — data warehouses, small machine databases, non-traditional data, and so on — but let's start with the Internet databases first.

Application database builders think that handling 1000 users at one time is scalability; Web database builders think that a Terabyte is a large database.

In a mainframe or client-server database shop, you know in advance the maximum number of terminals or workstations can be attached to your database. And if you don't like that number, you can disconnect some of them until you are finished doing batch processing jobs.

The short-term fear in a mainframe or client-server database shop is of ad hoc queries that can exclude the rest of the company from the database. The long-term fear is that the

database will outgrow the software or the hardware or both before you can do an upgrade.

In a Web database shop, you know in advance what result sets you will be returning to users. If a user is currently on a particular page, then he can only go to the previous page, or one of a (small) set of following pages. It is an old-fashioned tree structure for navigation. When the user does a search, you have control over the complexity of this search. For example, if I get to a Web site that sells antique comic books, I will enter the Web site at the home page 99.98 percent of the time instead of going directly to another page. If I want to look for a particular comic book, I will fill out a search form that forces me to search on certain criteria — I cannot look for "any issue of Donald Duck with a lot of Green on the Cover" on my own if cover colors are not one of the search criteria.

What the Web database fears is a burst of users all at once. There is not really a maximum number of PCs that can be attached to your database. In Larry Niven's science fiction novels, there are cheap teleportation booths all over the planet. You step inside one, put in your credit card, dial the number of your destination and suddenly you are in a receiving booth at your destination. The trouble is that when something interesting happens and it appears on the worldwide television system, you get "flash crowds" — all the people in the world who like to look at car wrecks show up in one place all at once.

If you get too many users trying to get to your Web site at once, the Web server crashes. This is exactly what happened to the Encyclopedia Britannica Web site the first day that they offered free access.

I must point out that virtually every public library on Earth has an encyclopedia set. Yet, you have never seen a crowd form around the reference books and bring the library to a complete halt. Much as I like the Encyclopedia Britannica, they never understood the Web. They first tried to ignore it, then they tried to sell a subscription service, then when they finally decided to make a living off of advertising, they underestimated the demand.

Another difference between an application database and a Web database is that an application database is not altered very often. Once you know the workloads, the indexes are seldom changed, and the tables are not altered very much.

In a Web database, you might suddenly find that one part of the database is all that anyone wants to see. If my Web-enabled comic book shop gets a copy of SUPERMAN #1, puts the cover on the Web, and gets listed as the "Hot Spot of the Day" on Yahoo! or another major search engine, then that one page will get a huge increase in hits.

Another major difference is that the Internet has no SQL-style transaction model. Once a user is connected to an SQL database, the system knows who he is, his privileges, and a history of his session.

The Web site has to confirm who you are with every action you take and has no concept of your identity or history. It is like a bank teller with brain damage who has to ask for your account number and identification for each check you deposit, even though you are standing in front of them. Cookies are a partial answer. These are small files with some identification data in them that can be sent to the Web site along with each request.

In effect, you have put your identification documents in a plastic holder around your neck for the bank teller to read each time. The bad news is that a cookie can be read by virtually anyone else and copied, so it is not very secure.

Right now, we do not have a single consistent model for Web databases. What we are doing is putting a SQL database on the back end, a Web site tool on the front end, and then doing all kinds of things in the middle to make them work together. I am not sure where we will sweep the Difficulty this time, either.

Avoiding SQL Injection

SQL Injection Security Threats

SQL injection is a serious threat to any vendor's SQL database in which applications use dynamic SQL (i.e., SQL compiled while the application is running). A hacker knowledgeable of SQL can exploit weaknesses presented by such applications. Good application design can mitigate the risks. Instead of focusing on satisfying just the literal business requirements, designers must carefully consider how an application can be used by a hacker in ways not intended. Additionally, DBAs must work with developers to grant only the most minimal of permissions to an application.

Creating a Test Application

It is easier to understand how SQL injection works by creating a simple test application. The target database is the SQL Server Northwind database. To simplify creation of the application, the Employees table is used as a list of authorized application users. The LastName column serves as the application's Username and the FirstName column is used as the Password. The user can log in by using either dynamic SQL or a stored procedure. Providing a username and password for authentication by an application is known as "forms-based authentication."

You can download the *ValidateUser.sql* (http://www.dbazine.com/code/ValidateUser.sql) stored procedure and the application code as either *SQLinjection.cs* (http://www.dbazine.com/code/SQLinjection.cs) or *SQLinjection.vb* (http://www.dbazine.com/code/SQLinjection.vb). Entering a first name not matching a last name in the Employees table simulates the effect of entering an invalid password in a real application.

Although this sample application is a Windows application, it could be a Web application or even a Java application. The vulnerability is a direct consequence of using dynamic SQL and completely independent of the operating system, database vendor, and programming language used to write the application.

Understanding the Test Application

Superficially, it appears that the test application works as intended and fulfills the basic business objective. Users who know a valid username and password are authorized by the application. Those who do not know a valid username and password are rejected. In the real world, the application would authenticate the user only by either dynamic SQL or a stored procedure; it would not provide a choice of methods. The test application provides a choice of using either approach to demonstrate both the vulnerability to SQL injection as well as how to protect against it. The dynamic SQL approach to user

authentication opens the SQL injection vulnerability. The stored procedure protects against SQL injection.

To keep the application simple, only a single screen was created. After entering a valid username and password into a real application, the user would of course be taken to another screen, whereas the test application displays "Welcome!" An invalid username and password causes the application to display "Hacker!"

Whether using dynamic SQL or a stored procedure, the SQL statement being executed is logically equivalent to this:

```
select count(*) from Employees where LastName = 'Davolio'
 and FirstName = 'Nancy'
```

An invalid username and password results in a value of zero being returned from the query. A valid username and password returns a value of one (assuming usernames combined with passwords must be unique).

Understanding Dynamic SQL

When a program builds and executes a SQL string at execution time, it is known as "dynamic SQL." Inspecting the application code does not provide an accurate indication of which SQL statement is actually executed. Instead, it only provides an indication as to the intent of what should be executed. Only SQL Profiler indicates what is actually executed. Examine the following application code that constructs the SQL string for the test application:

```
cmd.CommandText = "select count(*) from employees where LastName = '" +
                  username.Text + "' and FirstName = '" + password.Text
& "'"
```

It appears to be logically equivalent to the SQL statement described in the previous statement. If the inputs are Davolio and Lynn for Username and Password, respectively, then the SQL Profiler indicates that the actual SQL executed is:

```
select count(*) from employees where LastName = 'Davolio'
 and FirstName = 'Lynn'
```

This is completely consistent with the intended design of the application. Invalid inputs were detected and the user is denied access.

The Altered Logic Threat

The test application accepts the input Username and Password through simple text boxes. The user is free to enter text other than usernames or passwords. A hacker with basic SQL knowledge can be authenticated without even attempting to guess a valid username and password.

The user entered the following string and was authorized:

```
' or 1=1--
```

By placing a partial SQL statement into the Username textbox, a hacker "injects" the SQL fragment and thus alters the SQL statement that is executed. The injected SQL fragment actually consists of three different fragments, each with a different purpose:

1. The single quote closes out the LastName = ' fragment of the template query. This is done to maintain syntactical correctness of the modified query.

2. The or 1=1 fragment causes the count to always return a count greater than zero (assuming of course that the table has rows).

3. The double dash is a SQL inline comment which causes the entire rest of the dynamically built SQL statement to be ignored. Any input in the Password textbox is ignored.

The SQL Profiler reveals what was actually executed:

```
select count (*) from employees where LastName = ''
  or 1=1 --' and FirstName = ''
```

The application design intends for usernames and passwords to be entered, but SQL injection alters the SQL logic and makes them superfluous.

The Multiple Statement Threat

Unauthorized access to an application has different levels of severity depending on the purpose of the application. Sometimes people incorrectly rationalize the potential harm from security threats. For example, if a Web application

provides fee-based access to publications, unauthorized logins could be dismissed as lost revenue. The rationalization is that the cost to impose additional security features outweighs the cost of lost subscriber revenue. After all, there is a high probability that a person who hacks into a fee-based publication service won't pay to access the site if the hacking attempts fail. Such reasoning is naive and fatally flawed. What if the SQL savvy hacker decides to inject completely new SQL statements?

Consider the following SQL code:

```
' or 1=1;update prices set cost = 0--
```

Once again, the SQL Profiler reveals what was actually executed, which is actually two separate SQL statements:

```
select count (*) from employees where LastName = '' or 1=1
update merchandise set price = 0 --' and FirstName = ''
```

A semicolon is a valid SQL character for separating one SQL statement from another. It is particularly useful when multiple statements are entered on a single line or into a single string. A semicolon tells the SQL parser that the complete string is comprised of individual SQL statements to execute separately.

The hacker is not limited to injecting DML statements (insert, update, delete). How about a drop table statement? Assuming that the application has rights to drop tables, drop table statements could be injected to remove tables from the database. Consider the following input:

```
' or 1=1;update prices set cost = 0;drop table audit_trail;shutdown--
```

Not only would the *audit_trail* table be dropped, but the database would be shutdown immediately afterwards.

Prevention Through Code

To provide the absolutely most effective security, multiple techniques are required to protect your databases. The first line of defense is prevention at the user interface.

Whenever you are working with a database, you must first understand your data so you will better be able to protect it. In the test program, the LastName column of the Employees table is used as if it were a password in a table of usernames. This column has a maximum length of 20 characters, yet the test program does not limit user inputs to 20 characters. This is an egregious oversight: The worst attacks illustrated in this article could easily have been prevented by limiting the input to 20 characters. Not all input fields are short, so input length checking is only part of an overall defense. Additionally, in this example, a length restriction would not prevent this attack:

```
' or 1=1;shutdown--
```

Assuming that characters such as semicolons and double dashes are not valid in a username, then regular expression validation can be used to detect the invalid characters and reject the input. Not only is restricting the set of valid input characters a Procrustean solution, there exists the possibility of a very clever exploit using the SQL char function to provide the value of an individual ASCII character without explicitly having the character in the injected SQL input. Despite the limitations of rejecting input based on certain characters, it should be used when it is appropriate. Visual Studio.NET has a

regular expression validate control that greatly simplifies using regular expressions in ASP.NET Web pages.

Data type checking is helpful in detecting rogue input. User interfaces often accept date, time, and numeric input in text fields. Although users can type whatever they want in a text field, programs can check the input data to see if it is the correct data type. For example, if the Password input box is mapped to the EmployeeID column, then any user input should be checked to see if it is integer data. Only if the input is of the correct data type would the input be passed to the database server for processing. All of the rogue statements shown would fail an integer data type validation check.

The fundamental flaw of dynamic SQL is not that rogue inputs are allowed, but that rogue input can be executed. Dynamic SQL is convenient for developers, but it does not lock down the actual SQL during the application design stage.

Prevention Through Stored Procedures

Stored procedures are compiled when they are created; the SQL statement is frozen at creation time. Using the first rogue SQL fragment of

```
' or 1=1--
```

with the Stored Proc Login button, SQL Profiler reveals what is actually executed:

```
select @NbrRows = count(*) from employees where LastName = @Username
 and FirstName = @Password
```
Understand that @Username contains the following characters: ' or 1=1--

No matter what the inputs for @Username and @Password are, the stored procedure will always execute only the select statement shown. The SQL statement is predefined; it will never change based on the inputs. This stored procedure accepts two inputs, both strings. No matter what those input strings contain, they are always treated as just strings. Even a semicolon is treated as just another character, not as a SQL statement separator.

Although stored procedures overcome the fundamental weakness of dynamic SQL, it comes at a price. A stored procedure must be written in advance for all possible queries, and this is not always practical. For example, a search page for real estate listings does not lend itself to stored procedures. A customer is presented with multiple search criteria (price, number of bedrooms, bathrooms, and so on). Not all search criteria would be used at all times, so the number of stored procedures required to accommodate every possible select string would be unwieldy. Dynamic SQL is required in such cases.

Coding stored procedures in the .NET environment is covered in "Calling Stored Procedures from ADO.NET." (http://www.dbazine.com/cook6.html)

Prevention Through Least Privileges

The most basic security concept of all is the principle of least privileges. Never grant any more privilege to a user or an application than the absolute minimum to accomplish a task. Typical end user applications should not allow application users to execute indiscriminate DML, drop tables, or shut down databases. A hacker who attempts to drop a table but does not have rights to do so will not succeed in the attempt.

Conclusion

Implementing security best practices can prevent unintended access to your database. Forethought and well-designed applications are instrumental in protecting your interests. While dynamic SQL has its uses, a determination should be made early on as to whether or not it would be the best choice. If possible, stored procedures should be considered early in the design stage, as their execution is not dependent on nor changed by user input. Code should also be thoroughly examined to see that it does not lend itself to invasion. Developers must think like a hacker in order to fully evaluate the weaknesses in their applications.

Preventing SQL Worms

Preventing SQL Worms

Most of the damage caused by SQL worms targeting SQL Servers could easily have been prevented by applying service packs to SQL Servers prior to the attacks. Properly configured firewalls could have limited propagation of the worm. SQL worms are a far greater threat than many people realize because there are many SQL Servers out of sight and out of mind. Since SQL 7, the SQL Server database engine has been offered for free as MSDE, **M**icrosoft **D**esktop **E**ngine. MSDE 1.0 is the SQL 7 engine; MSDE 2000 is the SQL 2000 engine. MSDE is effectively limited to five connections, two gigabyte databases, and does not come with any tools such as the Enterprise Manager or the Query Analyzer. Any strategy put in place to protect against SQL worms and other threats must protect both SQL Servers and MSDE installations.

MSDE may be installed as part of an Office XP Developer Edition, Visual Studio .NET, Web Matrix, or other Microsoft product installation. Untold numbers of third-party applications install and use MSDE behind the scenes.

Finding SQL Servers Including MSDE

SQL Servers (for the rest of this article, this term includes MSDE) are applications named **sqlservr.exe** (not **sqlserver.exe**). There can be multiple copies of sqlservr.exe installed on a machine as long as each is in its own directory. You can identify instances of SQL Server by searching for

sqlservr.exe, but keep in mind that by default, XP and Windows 2003 Server do not search all folders as the following screen capture shows:

It is possible that SQL Server could have been installed to a location other than the default. Be sure to check **Search hidden files and folders** before starting your search.

A faster and more convenient way to find SQL Servers on a machine is to use the **Services** applet under either **Administrative Tools** or **Computer Management** (which is itself under Administrative Tools). On XP, **Administrative Tools** is not visible by default. To make it visible, right-click on the **Start** button, select **Properties**, click the **Customize** button, click the **Advanced** tab, scroll to the bottom of the **Start menu items** list and make a selection to **Display** the **System Administrative Tools**. The following screen capture from a Windows 2003 Server shows the **Services** applet. Because of space considerations, only a few services appear in this screen capture.

Name △	Description	Status	Startup Type	Log On As
MSSQL$NetSDK		Started	Automatic	Local System
MSSQL$WEBMATRIX		Started	Automatic	Local System
MSSQLSERVER		Started	Automatic	.\Administ...
MSSQLServerADHel...			Manual	Local System
Net Logon	Maintains a...		Manual	Local System
NetMeeting Remote...	Enables an...		Disabled	Local System
Network Connections	Manages o...	Started	Manual	Local System
Network DDE	Provides n...		Disabled	Local System
Network DDE DSDM	Manages D...		Disabled	Local System
Network Location A...	Collects an...	Started	Manual	Local System
SQLAgent$NetSDK			Manual	Local System
SQLAgent$WEBMA...			Manual	Local System
SQLSERVERAGENT			Manual	.\Administ...

SQL Servers are installed as services and may be installed as either what is known as a default instance or a named instance. A default instance of SQL Server has a service name of **MSSQLSERVER**. Named instances begin with **MSSQL$**. As you can see, the first three entries shown in the preceding screen capture indicate that there are three SQL Servers installed. MSSQLSERVER is the default instance.

MSSQL$NetSDK and MSSQL$WEBMATRIX are named instances. They are intended for use by software developers and may not be as properly secured as a production database should be.

All three SQL Servers are running with elevated privileges. It would be safer to run a SQL Server service under the context of a domain user account instead of a domain administrator account or Local System. The same is true of the SQL Server Agent service accounts. For more information, go to the Microsoft site for SQL Server and download these security whitepapers:

- http://www.microsoft.com/SQL/techinfo/administration/ 70/securityWP.asp

- http://www.microsoft.com/SQL/techinfo/administration/ 2000/securityWP.asp

You can also go to **Control Panel**, **Add/Remove Programs** to find instances of SQL Server installed on a machine. They will not necessarily be grouped together as they are in the Services applet.

Identifying Versions

You can determine if an instance of a SQL Server is the full version or the MSDE version by connecting through osql or the Query Analyzer and running this command:

```
select @@version
Microsoft SQL Server 2000 - 8.00.194 (Intel X86)
Aug 6 2000 00:57:48
Copyright (c) 1988-2000 Microsoft Corporation
Enterprise Edition on Windows NT 5.2 (Build 3768: )
```

As you can see, the last line indicates the version of SQL Server. Here is the output from running the command on an MSDE instance:

```
Microsoft SQL Server 2000 - 8.00.534 (Intel X86)
Nov 19 2001 13:23:50
Copyright (c) 1988-2000 Microsoft Corporation
Desktop Engine on Windows NT 5.2 (Build 3768: )
```

Now look at the output from another MSDE instance:

```
Microsoft SQL Server  2000 - 8.00.760 (Intel X86)
        Dec 17 2002 14:22:05
        Copyright (c) 1988-2003 Microsoft Corporation
        Desktop Engine on Windows NT 5.1 (Build 2600: Service Pack 1)
```

Do not make the mistake of thinking that SQL Server Service Pack 1 was applied to this instance. In this context, Service Pack 1 refers to the operating system only. You determine the SQL Server Service Pack level by looking at the version number on the first line of the output. The version number 8.00.760 is the proof that SQL Server Service Pack 3 was installed. This is explained in **sp3readme.htm**, a document that is included in the Service Pack 3 downloaded files. You should read it carefully before applying any version of SQL Server Service Pack 3.

Another way to determine the service pack level of a SQL instance is to run the following command:

```
select serverproperty('ProductLevel')
```

A single string is returned. If it is **RTM**, no service pack has been applied. If the string is **SP3**, them SQL Server Service Pack 3 has been applied. Do not consider a service pack to be successfully installed until you have used one of these queries to confirm the installation.

SQL Security Tools

Microsoft has tools to help you identify instances of SQL Server that need to be patched. The tools are SQL Scan and SQL Check. You can download them from the Microsoft download center, http://www.microsoft.com/downloads. These are command line tools. You need to read the readme.txt files that come with these tools and choose the appropriate switches. SQL Scan has the ability to check an entire domain or range of IP addresses.

Preventing Worms

First and foremost, you must keep current on service packs. Currently, SQL Server Service Pack 3 is available for download from:

http://www.microsoft.com/sql/downloads/2000/sp3.asp

It is actually three different service packs, one for SQL Server 2000, one for MSDE 2000, and another for SQL Server 2000 Analysis Services. You must download and install the service pack appropriate for which of these components you have installed on the machine. It is important to understand that once the service pack is downloaded, running the service pack executable does NOT install the service pack. It merely unpacks the files needed to install the service pack. You must stop the SQL Server service before a service pack can be applied. You should back up your databases before applying a service pack.

Installing the Service Pack 3 for MSDE 2000 requires that you have administrative rights on the computer. Be sure to read the documentation carefully. The setup.exe is not just for applying

a service pack; it will also install an instance of MSDE 2000. To only install the service pack, you will have to apply command line switches as described in the **sp3readme.htm** help file. You either need to know the instance name or which .msi file was used to install MSDE. As described previously, you can use the Services applet to find the instance names.

The Slammer/Sapphire worm exploits a buffer overrun vulnerability on SQL Server port 1434. Blocking UDP ports 1433 and 1434 at your firewall will protect your server from this worm and many other SQL Server exposures. You can also block your SQL Servers for inbound traffic on UDP port 1434, but this would interfere with name resolution.

MSDE Issues

MSDE requires special attention to the instructions in **sp3readme.htm**. Not all attempts at applying Service Pack 3 to MSDE have been successful. To log installation problems, use the modified syntax shown below:

```
setup /l*v c:\msde.log /upgradesp . . .
```

Additionally, if the MSDE being upgraded has a blank sa password (actually a NULL password, there actually isn't a password), the installation will fail and show the following error message:

If you choose not to take advantage of the opportunity to fix this security vulnerability, use the following syntax:

```
setup /l*v c:\msde.log /upgradesp BLANKSAPWD=1 . . .
```

To change from a NULL password to a real password using osql, use syntax similar to this:

```
C:\>osql -E
1> sp_password NULL, 'Str0ngP@sswOrd', sa
2> go
Password changed.
```

.NET SDK MSDE and Visual Studio .NET

Users of the .NET SDK Version 1.0 must apply a special version of Service Pack 3 which can be found at this location:

http://msdn.microsoft.com/netframework/downloads/update s/sdkfix/default.asp

Additionally, as the following link indicates, if you have both Visual Studio .NET and the .NET SDK MSDE installed, you should apply both the regular MSDE Service Pack 3 as well as the .NET SDK Service Pack 3:

http://support.microsoft.com/default.aspx?scid=kb;en-us;813850

Application Center 2000

Application Center 2000 uses MSDE and has specific requirements for applying Service Pack 3. Details may be found at:

http://support.microsoft.com/?kbid=813115

Deworming

The Slammer/Sapphire worm is memory resident only. Stopping and restarting the SQL Server service will clear the worm from the instance, but will not by itself prevent reinfection. Applying Service Pack 3 stops and restarts the service, so it both clears the worm and prevents reinfection.

Baseline Security Analyzer

It is important not to focus so much attention on widely publicized threats as to overlook other weaknesses. Microsoft provides a free tool to help identify other vulnerabilities your machine may have.

Download the Microsoft Baseline Security Analyzer from:

http://www.microsoft.com/technet/treeview/default.asp?url= /technet/security/tools/Tools/MBSAhome.asp

The following screen capture shows a small portion of the scan performed on the entire machine:

SQL Server Scan Results

Instance (default)

Vulnerabilities

Score	Issue	Result
✗	Service Accounts	SQL Server and/or SQL Server Agent Services accounts are members of the local Administrators group or run as LocalSystem. What was scanned Result details How to correct this
✗	SQL Server Security Mode	SQL Server authentication mode is set to SQL Server and Windows (Mixed Mode). What was scanned How to correct this
✓	Exposed SQL Password	The 'sa' password and SQL service account password are not exposed in text files. What was scanned Result details

The screen capture shows that the scan detected the service account privilege problems mentioned previously in this article. The Baseline Security Analyzer not only points out problems, but also provides hyperlinks to explanations on how to correct the identified problems.

Conclusion

Security best practices can prevent unnecessary down time caused by security threats. Staying current on service packs and hotfixes is essential. By understanding and expecting threats, proper planning can mitigate risks.

Basic SQL Tuning Hints

SQL tuning

Oracle SQL tuning is a phenomenally complex subject, and entire books have been devoted to the nuances of Oracle SQL tuning. However there are some general guidelines that every Oracle DBA follows in order to improve the performance of their systems. The goals of SQL tuning are simple:

- Remove unnecessary large-table full table scans Unnecessary full table scans cause a huge amount of unnecessary I/O, and can drag down an entire database. The tuning expert first evaluates the SQL based on the number of rows returned by the query. If the query returns less and 40 percent of the table rows in an ordered table, or 7 percent of the rows in an unordered table), the query can be tuned to use an index in lieu of the full table scan. The most common tuning for unnecessary full table scans is adding indexes. Standard B-tree indexes can be added to tables, and bitmapped and function-based indexes can also eliminate full table scans. The decision about removing a full table scan should be based on a careful examination of the I/O costs of the index scan vs. the costs of the full table scan, factoring in the multiblock reads and possible parallel execution. In some cases an unnecessary full table scan can be forced to use an index by adding an index hint to the SQL statement.

- Cache small-table full table scans In cases where a full table scan is the fastest access method, the tuning professional should ensure that a dedicated data buffer is available for

the rows. In Oracle7 you can issue alter table xxx cache. In Oracle8 and beyond, the small table can be cached by forcing to into the KEEP pool.

- Verify optimal index usage This is especially important for improving the speed of queries. Oracle sometimes has a choice of indexes, and the tuning professional must examine each index and ensure that Oracle is using the proper index. This also includes the use of bitmapped and function-based indexes.

- Verify optimal JOIN techniques Some queries will perform faster with NESTED LOOP joins, others with HASH joins, while other favor sort-merge joins.

These goals may seem deceptively simple, but these tasks comprise 90 percent of SQL tuning, and they don't require a through understanding of the internals of Oracle SQL.

Index