SQL Sucks! – Part II

by Iggy Fernandez

Iggy Fernandez

Man is timid and apologetic; he is no longer upright; he dares not say "I think," "I am," but quotes some saint or sage . . . We are like children who repeat by rote the sentences of grandames and tutors, and, as they grow older, of the men of talents and character they chance to see,—painfully recollecting the exact words they spoke; afterwards, when they come into the point of view which those had who uttered these sayings, they understand them and are willing to let the words go; for at any time they can use words as good when occasion comes.
—Ralph Waldo Emerson

Summary

In the first part of this multi-part article, we explained that SQL is what is called a "non-procedural" language; i.e., an SQL query simply identifies a subset of data in the database without specifying *how* to go about extracting that data. This has led to the pervasive belief that application programmers have no responsibility for SQL performance. In this installment, we dive into the theoretical underpinnings of the SQL language—an understanding of which is crucial to taking responsibility for SQL performance. In the next installment, we will put everything together.

Dangerous Beliefs

The non-procedural nature of the SQL language has led to many dangerous beliefs centered around the theme that application programmers have no responsibility for the performance of the SQL statements they write. Here is a representative list.

- **Dangerous Belief #1:** DBAs bear chief responsibility for the performance of SQL statements.
- **Dangerous Belief #2**: Applications should be designed without reference to the way data is stored, e.g., index-organized tables, hash clusters, partitions, etc.
- **Dangerous Belief #3**: Application programmers should not tailor their SQL statements to make use of existing indexes. DBAs should instead create traps to catch badly performing SQL at runtime and create new indexes as necessary to make them perform better.
- **Dangerous Belief #4**: It is not necessary to review the Query Execution Plan of an SQL statement before releasing it into a production environment. It is further not necessary to *freeze* the Query Execution Plan of an SQL statement before releasing it into a production environment. It is desirable that Query Execution Plans change in response to changes in the statistical information that the query optimizer relies upon. Such changes are always for the better.
- **Dangerous Belief #5**: The most common cause of poorly performing SQL is the failure of the DBA to collect statistical information on the distribution of data for the use of the query optimizer.[1] This statistical information should be refreshed frequently.[2]

A True Story

I grub for my living in a dusty corner of a mighty telecommunications company, babysitting a brood of databases and feeding them whenever they are hungry for disk space. One day an irate developer submitted a high-priority request that we find out why Oracle was "not responding to simple queries."

We found that that the developer had submitted a seven-way join without any joining criteria whatsoever, i.e., a query of the form "SELECT . . . FROM Table#1, Table#2, Table#3, Table#4, Table#5, Table#6, Table#7!" Poor Oracle was gamely trying to perform a seven-way Cartesian product of the tables but probably needed 100 years to complete the task since the estimated query cost recorded in the V$sqlplan view was 9,275,840,000,000,000!

When we asked the developer why he had not specified any joining criteria, he said that he first wanted to determine if Oracle could handle a "simple" query before submitting a complex query. We offered to send him the Query Execution Plan for his "simple" query but he said that he did not know

[1] Consider, for example, the following statement found in an article published in a recent issue of the journal of the IOUG: *One of the greatest problems with the Oracle cost-based optimizer was the failure of the Oracle DBA to gather accurate schema statistics. . . The issue of stale statistics and the requirement for manual analysis resulted in a "bum rap" for Oracle's cost-based optimizer, and beginner DBAs often falsely accused the CBO of failing to generate optimal execution plans when the real cause of the sub-optimal execution plan was the DBA's failure to collect complete schema statistics*—**www.ingentaconnect.com/content/ioug/sj/2006/00000013/00000001/art00003**

[2] The following statement by Donald Burleson puts the finger on the dangers of collecting fresh statistical information for the use of the query optimizer: *It astonishes me how many shops prohibit any un-approved production changes and yet re-analyze schema stats weekly. Evidently, they do not understand that the purpose of schema re-analysis is to change their production SQL execution plans, and they act surprised when performance changes!*—**www.dba-oracle.com/art_orafaq_cbo_stats.htm**

how to interpret Query Execution Plans. He probably did not know much about SQL either![3]

Relational Algebra

SQL is largely based on the "algebra of relations," i.e., the ways in which relations (tables) can be combined with each other to form new relations. An SQL statement then is an algebraic expression in which the "operands" are tables instead of numbers. Here are five examples of relational operations.

"Selection"	Form another relation by extracting a subset of the rows of a relation of interest using some criteria.
"Projection"	Form another relation by extracting a subset of the columns of a relation of interest.[4]
"Union"	Form another relation by selecting all rows from two relations of interest. If the first relation has 10 rows and the second relation has 20 rows, then the resulting relation will have at most 30 rows.[5]
"Difference"	Form another relation by extracting only those rows from one relation of interest that do not occur in a second relation.
"Join"	Form another relation by concatenating records from two relations of interest. For example, if the first relation has 10 rows and the second relation has 20 rows, then the resulting relation will have 200 rows—and if the first relation has 10 columns and the second relation has 20 columns, then the resulting relation will have 30 columns.

It is possible to create new operations by combining the "primitive" operations in the above table. For example, "Natural Join" is a combination of Join and Selection.

We illustrate the five operations defined in the above table with an example. Consider the following table definitions.

- Suppliers is a table that contains SupplierName as its only column.
- Parts is a table that contains PartName as its only column.
- SuppliedParts is a table that contains SupplierName and PartName as its two columns. The occurrence of a certain combination of SupplierName and PartName in

[3] Steve Feuerstein, the guru of PL/SQL, said, in an interview for the *NoCOUG Journal*: *Java and .Net and VB programmers should never write SQL. They generally don't have much respect for the language and don't do a very good job writing it.*

[4] Duplicates are eliminated from the result.

[5] Duplicates are eliminated from the result.

Lies, Damn Lies, and SQL!

Sumit Sengupta from Columbus, OH, sent us this puzzle. Describe a combination of circumstances in which the COUNT function could produce the anomalous results seen in the example below.

```
SQL> DESCRIBE employees;
 Name              Null?     Type
 ----------------- -------- ---------------
 NAME               NOT NULL VARCHAR2(25)
 SALARY             NOT NULL NUMBER(8,2)
 COMMISSION_PCT     NOT NULL NUMBER(4,2)
SQL> SELECT COUNT(name) FROM employees;
COUNT(NAME)
-----------
          3
SQL> SELECT COUNT(salary) FROM employees;
COUNT(SALARY)
-------------
          2
SQL> SELECT COUNT(commission_pct) FROM
employees;
COUNT(COMMISSION_PCT)
---------------------
          1
SQL> SELECT COUNT(1) FROM employees;
 COUNT(1)
----------
          0
```

The prize offered for the most thorough answer is a SanDisk Sansa M240 1 GB MP3 Player. The SanDisk Sansa M240 is the most popular flash-based MP3 player sold by Amazon.com, ahead of the iPod Nano, and can be used with music subscription services such as Napster and Rhapsody.

The contest is open to all NoCOUG members. Send your answers to journal@nocoug.org by August 30. The decision of the judges is final. ▲

this table indicates that the supplier in question supplies the indicated part. Here is some sample data.

Suppliers

SupplierName
Ashley
Bertram
Carlton

Parts

PartName
Hammer
Nail
Screw

SuppliedParts

SupplierName	PartName
Ashley	Hammer
Ashley	Nail
Ashley	Screw
Bertram	Hammer
Bertram	Nail
Carlton	Screw

The question we try to answer is "Which suppliers supply all parts?" The answer is that only Ashley supplies all parts. Here is how we can formally obtain this answer with the help of the five relational operations defined previously. We create several intermediate result tables along the way.

1. First we use the Join operation and form an intermediate result table by concatenating records from the Suppliers table and the Parts table. All combinations of SupplierName and PartName occur in this table.

SupplierName	PartName
Ashley	Hammer
Ashley	Nail
Ashley	Screw
Bertram	Hammer
Bertram	Nail
Bertram	Screw
Carlton	Hammer
Carlton	Nail
Carlton	Screw

2. Next we use the Difference operation and form a second intermediate result table by extracting only those rows from the table obtained in the previous step that do not occur in the SuppliedParts table. The occurrence of a certain combination of SupplierName and PartName in this new intermediate table indicates that the supplier in question does not supply the indicated part.

SupplierName	PartName
Bertram	Screw
Carlton	Hammer
Carlton	Nail

3. Next we use the Projection operation and form yet another intermediate result table by extracting only the first column from the table obtained in the previous step. This is the list of suppliers who do not supply at least one part.

SupplierName
Bertram
Carlton

4. Finally we use the Difference operation again and obtain the final result we were seeking by extracting only those rows from the Suppliers table that do not occur in the intermediate result table of the previous step. This is the required list of suppliers who do supply all parts!

SupplierName
Ashley

SQL Solution

Here is the SQL language solution of the question, "Which suppliers supply all parts?" Once again we derive the result in stages to aid understanding. At each stage, we highlight the partial formulation of the previous stage.

1. First we "join" the Suppliers and Parts tables to obtain all combinations of SupplierName and PartName. We use the following language.

```
select SupplierName, PartName
from Suppliers, Parts
```

2. We next eliminate all combinations of SupplierName and PartName that do not occur in the SuppliedParts table.

```
select SupplierName, PartName
from Suppliers, Parts
minus
select SupplierName, PartName
from SuppliedParts
```

3. We then extract supplier names from the intermediate result obtained in the previous step, thus yielding the list of suppliers who do *not* supply *at least one* part.

```
select SupplierName from
(
 select SupplierName, PartName
 from Suppliers, Parts
 minus
 select SupplierName, PartName
 from SuppliedParts
)
```

4. Finally, we remove suppliers obtained in the previous stage from the list of suppliers in the Suppliers table, thus yielding the list of suppliers who *do* supply *all* parts!

```
select SupplierName from Suppliers
minus
select SupplierName from
(
 select SupplierName, PartName
 from Suppliers, Parts
 minus
 select SupplierName, PartName
 from SuppliedParts
)
```

The author can be reached at **iggy_fernandez@hotmail.com**.