

Optimizing SQL Query Processing

Abstract

Query performance in relational database systems is dependent not only on the database structure, but also on the way in which the query is optimized. We show various classes of syntactically equivalent SQL queries, each of which can exhibit substantial differences in data access depending on the characteristics of the query formulation and the success of the database query optimizer. Simply put, similar looking queries can take significantly different times to execute. We conclude that on-line analytic processing systems must not depend on dynamic user specified SQL queries if consistent overall system performance is required. If SQL queries can be structured dynamically from user input, then system designers will not be able to guarantee performance.

Introduction

SQL query processing requires that the DBMS identify and execute a strategy for retrieving the results of the query. The SQL query determines what data is to be found, but does not define the method by which the data manager searches the database. Hence, query optimization is necessary for high-level relational queries and provides an opportunity for the DBMS to systematically evaluate alternative query execution strategies and to choose an optimal strategy. In some cases the data manager cannot determine the optimal strategy. Assumptions are made which are predicated on the actual structure of the SQL query. These assumptions can significantly affect the query performance. This implies that certain queries can exhibit significantly different response times for relatively innocuous changes in query syntax and structure.

For the purpose of this discussion an example medical database will be used. Figure 1 below illustrates our subject database schema for physicians, patients, and medical services. The Physician table contains one row for every physician in the system. Various attributes describe the physician name, address, provider number and specialty. The Patient table contains one row for every individual in the system. Patients have attributes listing their social security number, name, residence area, age, gender, and doctor. For simplicity, a physician can see many patients, but a patient has only one doctor. A Services table exists which lists all the valid medical procedures which can be performed. When a patient is ill and under the care of a physician, a row exists in the Treatment table describing the prescribed treatment. This table contains one attribute recording the cost of the individual service and a compound key that identifies the patient, physician, and the specific service received.

Patient	<table border="1"><tr><td><u>SSN</u></td><td>Name</td><td>Age</td><td>Gender</td><td>Area</td><td>Doctor</td></tr></table>	<u>SSN</u>	Name	Age	Gender	Area	Doctor	1,000,000
<u>SSN</u>	Name	Age	Gender	Area	Doctor			
Physician	<table border="1"><tr><td><u>Provider</u></td><td>Dr_SSN</td><td>Specialty</td><td>Dr_Name</td><td>Dr_Address</td></tr></table>	<u>Provider</u>	Dr_SSN	Specialty	Dr_Name	Dr_Address	1,000	
<u>Provider</u>	Dr_SSN	Specialty	Dr_Name	Dr_Address				
Service	<table border="1"><tr><td><u>Service</u></td><td>Type</td></tr></table>	<u>Service</u>	Type	10,000				
<u>Service</u>	Type							
Treatment	<table border="1"><tr><td><u>Patient</u></td><td><u>DrNum</u></td><td><u>Srvnum</u></td><td>Cost</td></tr></table>	<u>Patient</u>	<u>DrNum</u>	<u>Srvnum</u>	Cost	10,000,000		
<u>Patient</u>	<u>DrNum</u>	<u>Srvnum</u>	Cost					

Figure 1

Query Processing

The steps necessary for processing an SQL query are shown in Figure 2. The SQL query statement is first parsed into its constituent parts. The basic SELECT statement is formed from the three clauses SELECT, FROM, and WHERE. These parts identify the various tables and columns that participate in the data selection process. The WHERE clause is used to determine the order and precedence of the various attribute comparisons through a conditional expression. An example query to determine the names and addresses of all patients of Doctor 1234 is shown as query Q1 below. The WHERE clause uses a conjunctive clause which combines two attribute comparisons. More complex conditions are possible.

```
Q1:  SELECT  Name, Address, Dr_Name
      FROM    Patient, Physician
      WHERE   Patient.Doctor = Physician.Provider AND Physician.Provider = 1234
```

The query optimizer has the task of determining the optimum query execution plan. The term “optimizer” is actually a misnomer, because in many cases the optimum strategy is not found. The goal is to find a reasonably efficient strategy for executing the query. Finding the perfect strategy is usually too time consuming and can require detailed information on both the data storage structure and the actual data content. Usually this information is simply not available.

Once the execution plan is established the query code is generated. Various techniques such as memory management, disk caching and parallel query execution can be used to improve the query performance. However, if the plan is not correct, then the query performance cannot be optimum.

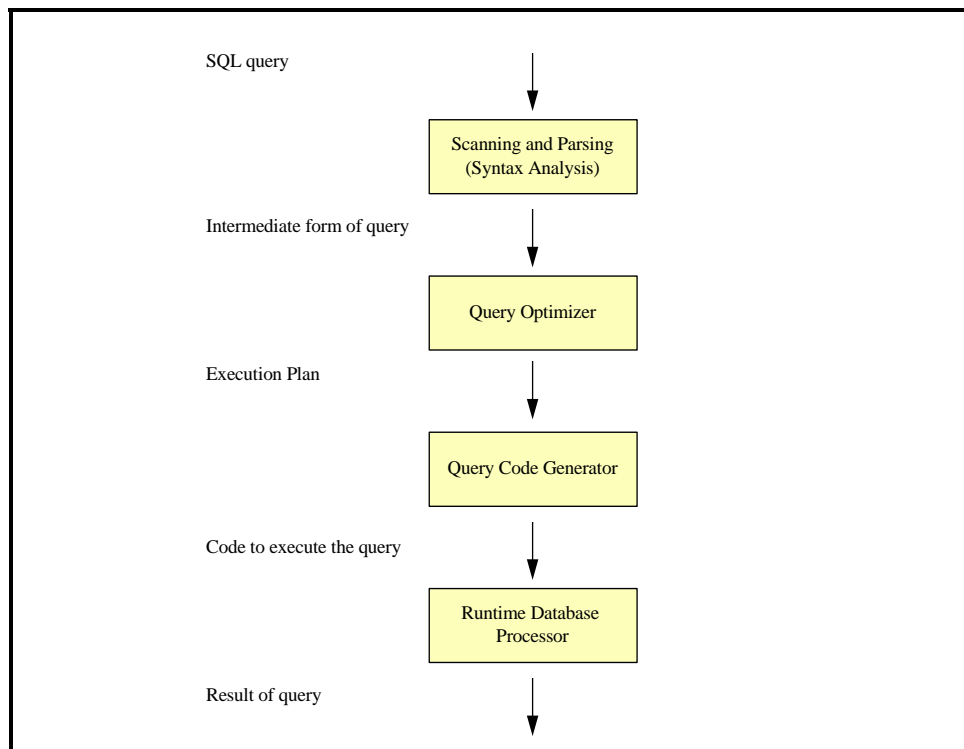


Figure 2

Query Optimizing

There are two main techniques for query optimization. The first approach is to use a rule based or heuristic method for ordering the operations in a query execution strategy. The rules usually state general characteristics for data access, such as it is more efficient to search a table using an index, if available, than a full table scan. The second approach systematically estimates the cost of different execution strategies and chooses the least cost solution. This approach uses simple statistics about the data structure size and organization as arguments to a cost estimating equation. In practice most commercial database systems use a combination of both techniques.

Indexes

Consider, for example, a rule-based technique for query optimization that states that indexed access to data is preferable to a full table scan. Whenever a single condition specifies the selection, it is a simple matter to check whether or not an indexed access path exists for the attribute involved in the condition. Queries Q2 and Q3 are two queries which, from a syntactic structure, are identical. However, query Q2 uses an index on the patient number, and query Q3 does not have an index on the patient name. Assuming a balanced tree based index, query Q2 will at worst case access on the order of $\log_2(n)$ entries to locate the required row in the table. Conversely, query Q3 must search on average $n/2$ rows to find the entry during a full table scan, and n rows if the entry does not exist in the table. When $n = 1,000,000$ this is the difference between accessing 20 rows versus 500,000 rows for a successful search. Clearly, indexing can significantly improve query performance. However, it is not always practical to index every attribute in every table, thus certain types of user queries can respond quite differently from others.

Q2:	SELECT *	FROM Patient	WHERE Patient.SSN = 11111111	In this query, the SSN attribute is the primary key index for the Patient table.
Q3:	SELECT *	FROM Patient	WHERE Patient.Name = "Doe, John Q."	In this query, no index exists on the Name attribute. This requires a full table scan.

Selectivities

A more significant problem occurs when more than one condition is used in a conjunctive selection. In this case the selectivity of each condition must be considered. Selectivity is defined as the ratio between the number of rows that satisfy the condition to the total number of rows in the table. This is the probability that a row satisfies the condition, assuming a uniform distribution. If the selectivity is small, then only a few rows are selected by the condition, and it is desirable to use this condition first when retrieving records. To calculate selectivities, the database manager needs statistics on all table and attribute values. The heuristic rule states that, for multiple conjunctive conditions, the order of application is from smallest selectivity to largest.

Queries Q4 and Q5 illustrate multiple conditions in a conjunctive selection on the Patient table. Consider the case where the selectivity on Age is $10,000/1,000,000 = 0.01$ (Age is assumed to be uniformly distributed between 0 and 100). The selectivity on Gender is $500,000/1,000,000 = 0.5$ (Gender is assumed to be either M or F). It is clear that by using age as the first retrieval condition, 10,000 rows are accessed for testing against the gender condition, versus accessing 500,000 rows if the gender attribute was chosen first. This is a 50 times performance difference. Selectivities can be used only if statistics are maintained by the database manager. If this information is not available, then the order of condition testing often defaults to the order of conditions as specified in the WHERE clause.

- Q4: SELECT *
 FROM Patient
 WHERE Age = 45 AND Gender = M
- In this query, the Age attribute is specified first.
- Q5: SELECT *
 FROM Patient
 WHERE Gender = M AND Age = 45
- This query specifies Gender first.

Uniformity

In many cases the actual data does not follow a uniform distribution. Consider the case where 95% of the patients live in the province of New Brunswick and the remaining 5% live in 199 different states and countries of the world. In this case there are 200 different values for the Area attribute. The selectivity of the Area attribute, assuming a uniform distribution, is $5,000/1,000,000 = 0.005$. Thus, this attribute will be accessed first given any query with a conjunctive clause relating Area and Age. In the example below, query Q6 selects Area based on the province of Ontario. We estimate that $(5\% \text{ of } 1,000,000) / 199$, or 251 patients live in Ontario. These rows are accessed first and then tested against the Age condition. Conversely, query Q7 selects patients in the province of New Brunswick. In this case, 950,000 patient rows are accessed, or more than 3,700 times the number of rows for the Ontario example. The distribution was skewed sufficiently to result in a poor choice by the query optimizer. Clearly, non-uniform data distributions can significantly affect query performance.

- Q6: SELECT *
 FROM Patient
 WHERE Area = "Ontario" AND Age = 45
- A uniform distribution for out of province residents predicts that 251 patients live in Ontario.
- Q7: SELECT *
 FROM Patient
 WHERE Area = "New Brunswick" AND Age = 45
- Actual data has 950,000 patients living in New Brunswick.

Disjunctive Clauses

A disjunctive clause occurs when simple conditions are connected by the OR logical connective rather than AND. These clauses are much harder to process and optimize. For example, consider query Q8, which uses a disjunctive clause relating a specific doctor and the patient area of residence. With such a condition, little optimization can be done because the rows satisfying the query are the union of the rows satisfying each of the individual conditions. If any one of the search conditions does not have an access path, then the query optimizer is compelled to choose a full table scan to satisfy the query. Performance can only be improved if an access path exists on every condition in the disjunctive clause. In this case, row sets can be found satisfying each condition and then combined through applying a union operation across the result sets to eliminate duplicate rows. However, set union operations can also be expensive. The customary way to implement union operations is to sort the relations on the same attributes and then scan the sorted files to eliminate duplicate rows. Superficially, the differences between query Q8 and Q9 appear trivial, yet the queries can have profound differences in performance. In many cases the use of disjunctive clauses in queries results in either a brute force linear search of the table, or a sort of a potentially large amount of data.

Q8:	SELECT * FROM Patient WHERE Doctor = 1234 OR Area = "Ontario"	Group one doctor's patients with Ontario patients.
Q9:	SELECT * FROM Patient WHERE Doctor = 1234 AND Area = "Ontario"	Identify only the Ontario patients of a particular doctor.

Join Selectivities

The JOIN operation is one of the most time consuming operations in query processing. A join operation matches two tables across domain compatible attributes. One common technique for performing a join is a nested (inner-outer) loop or brute force approach. In this case, for every row in the first table a scan of the second table is performed and every record is tested for satisfying the join condition. A second technique is to use an access structure or index to retrieve the matching records. In this case, for every row in the first table an index is used to access the matching records from the second table.

One factor that significantly affects performance of the join is the percentage of rows in one table that will be joined with rows in the other table. This is called the join selection factor. This factor depends not only on the two tables to be joined, but also on the join fields if there are multiple join conditions between the two tables. For example, query Q10 joins each Physician row with the Patient rows. Each physician is expected to exist once in the Patient table (after all, a physician is also a patient), but 999,000 patient rows will not be joined. Suppose indexes exist on each of the join attributes. There are two options for performing the join. The first retrieves each Patient row and then uses the index into the Physician table to find the matching record. In this case, no matching records will be found for those patients who are not also physicians. The second option first retrieves each Physician row and then uses the index into the Patient table to find the matching Patient row. In this case, every physician will have one matching patient row.

It is clear that the second option is more efficient than the first option. This occurs because the join selection factor of Physician with respect to the join condition is 1. Conversely, the Patient selection factor with respect to the same join condition is 1,000/1,000,000. Choosing optimum join methods requires that various table sizes and other statistics be used to compute estimated join selectivities.

Q10:	SELECT * FROM Patient, Physician WHERE Patient.SSN = Physician.Dr_SSN	If join selectivities are not used, then these two queries can exhibit quite different performance.
Q11:	SELECT * FROM Patient, Physician WHERE Physician.Dr_SSN = Patient.SSN	

Views

A view in SQL is a single table that is derived from other tables. A view can be considered as a virtual table or as a stored query. A view is often used to specify a frequently used query. This is of particular benefit if tables must be joined or restricted. One difficulty with views is that a view can hide the query complexity from the user. For example, view V1 describes a virtual table that contains the same number of rows as the Physician table. Query Q12 accesses the Patient, Provider, and Treatment tables through view

V1 to determine the total cost of services that Ophthalmologists have rendered. Conversely, query Q13 accesses only the Physician table to retrieve (different) data on Ophthalmologists. The problem is that both Q12 and Q13 appear to be of the same order of complexity, given that knowledge of the view is hidden, yet each query will clearly have a different performance profile.

V1:	<pre>CREATE VIEW DrService (Dr, Specialty, Age, TotCost) AS SELECT Provider, Specialty, Age, Sum(Cost) FROM Patient, Physician, Treatment WHERE SSN = Dr_SSN AND DrNum = Provider GROUP BY Provider</pre>	<p>This view matches the Physician table to the Treatment table, and then joins the result to the Patient table.</p>
Q12:	<pre>SELECT * FROM DrService WHERE Specialty = "Ophthalmologist"</pre>	<p>This query performs a three-way join, through the view.</p>
Q13:	<pre>SELECT * FROM Physician WHERE Specialty = "Ophthalmologist"</pre>	<p>This query simply scans one table.</p>

Conclusions

For many decision support systems we have observed that clients expect that information can always be retrieved efficiently, assuming that the database is designed properly. We have attempted to show why this is a myth. Queries formulated using an SQL query language provide little predictive information useful for estimating query performance. Internal knowledge of the database structure, data distribution, and query optimizing strategy are necessary to develop effective query statements. This technical knowledge rarely exists in the user community.

This leads us to recommend that enterprise decision support systems remain independent from user developed, unstructured queries. Any request to integrate ineffective or unproven query statements into a management system should be discouraged. The inevitable result is a dissatisfied client.

References

- [1] Date, C. *An Introduction to Database Systems*, Addison-Wesely Publishing Co., 1975
- [2] Knuth, D. *The Art of Computer Programming*, Vol. 3, Searching and Sorting, Addison-Wesely Publishing Co., 1973
- [3] Elmasri, R. And Navathe, S. *Fundamentals of Database Systems*, Benjamin Cummings Publishing Co., 1989