OPTIMIZATION OF DYNAMICALLY GENERATED SQL QUERIES FOR TINY-HUGE, HUGE-TINY PROBLEM

Arjun K Sirohi

1 Doctorate Student, Faculty of Technology (Computer Science), CMJ University, Shillong, India
2 Consulting Member Technical Staff, Oracle, Bellevue, WA, USA
      asirohi@yahoo.com

ABSTRACT

In most new commercial business software applications like Customer Relationship Management, the data is stored in the database layer which is usually a Relational Database Management System (RDBMS) like Oracle, DB2 UDB or SQL Server. To access data from these databases, Structured Query Language (SQL) queries are used that are generated dynamically at run time based on defined business models and business rules. One such business rule is visibility-the capability of the application to restrict data access based on the role and responsibility of the user logged in to the application. This is generally achieved by appending security predicates in the form of sub-queries to the main query based on the roles and responsibility of the user. In some cases, the outer query may be more restrictive while in other cases, the security predicates may be more restrictive. This often results in a dilemma for the cost-based optimizer (CBO) of the backend database whether to drive from the outer query or drive from the security predicate sub-queries. This dilemma is sometimes called the “Tiny-Huge, Huge-Tiny” problem and results in serious performance degradation by way of increased response times on the application User Interface (UI). This paper provides a case study of a new approach to vastly reduce this CBO dilemma by a combination of de-normalized columns and re-writing of the security predicates’ sub-queries at run-time, thereby levelling the outer and security sub-queries. This approach results in more stable execution plans in the database and much better performance of such SQLs, effectively leading to higher performance and scalability of the application.

KEYWORDS

SQL Performance, RDBMS, Cost Based Optimizer (CBO), SQL, Plan Cost, Execution Plan, Buffer Gets, Query Optimization, Performance, Scalability, CRM Applications

1. INTRODUCTION

Most commercial business applications use relational databases as the back-end to store business data. Such data is accessed using SQL queries that are dynamically generated by the application framework, using a defined business model and business rules. For example, Oracle Application Development Framework uses its SQL generation engine to do this. The performance of the application is generally gauged by the response times in the UI. The UI response times in turn depends in a large part on the query response time in the database. If a user has to wait to get the results in the UI, they complain about it as the application performing badly. The query response time in the database depends on many factors, including the size and complexity of the SQL text as well as how much processing the database engine must do before it can arrive at the result set to be sent to the application requesting for it. A simple SQL with few tables and joins with good, restrictive filter predicates will generally perform better than a SQL with many tables, views and joins and less restrictive filter predicates. One of the key elements of SQL performance is the
decision by the CBO to arrive at an execution plan that it considers as most optimal. In Oracle database, the CBO is a complex engine and it evaluates many different optimization paths and access methods before finalizing an execution plan. In recent releases, the concept of cardinality feedback and other features also often result in second, third or more execution plans for the same SQL. From the application perspective, a common way to restrict access to the data on a need to know basis is to implement security and visibility through a set of roles and responsibilities that each are defined by way of sub-queries that get appended to the main SQL at run time. This method of adding sub-queries cumulatively through a security framework results in complex SQLs, especially for users who are granted many different roles and responsibilities. The main query is usually referred to as the outer query and the appended security predicates’ sub-queries are termed as the inner query. When such SQLs arrive at the database, the CBO evaluates them for possible access paths and join optimizations to decide on an execution plan based on available statistics. Very often, the CBO is posed with the dilemma of whether to drive from the outer query or drive from the inner query. For some users, the outer query with its filter predicates can be greatly restrictive while for other users, the inner security predicates’ sub-queries can be more restrictive. This if sometimes referred to as the “Tiny-Huge, Huge-Tiny” problem that many application and database designers struggle to manage. Many times, this problem leads to poor choices by the CBO, resulting in sub-optimal execution plans leading to poor query response times and consequently causing performance and scalability issues for the application as well as the database. This paper presents the analysis of SQLs and CBO execution plans from the Opportunity Management module of a CRM application that performed poorly due to the aforesaid “Tiny-Huge, Huge-Tiny” problem. Based on the analysis, the paper then presents a suggested solution incorporating some de-normalized columns and rewrite of the security predicates’ sub-queries that result in vastly improved performance and scalability of such queries and consequently of the application.

2. THE EXISTING SQLS, PLANS AND ISSUES AFFECTING PERFORMANCE

The Opportunity Management Module of any CRM application is a widely used application in any sales department of businesses. Details of sales opportunities are stored in tables of relational databases and accessed through the user interface of the web-based application. The visibility or access-control of which user can access which records of such sales opportunities is controlled through rules defined in terms of SQL sub-queries that get appended to the main SQL at run time based on the roles and responsibilities of the logged in user. For example, roles could be Sales Representative, Sales Manager, Sales Vice President or Sales Administrator. The access could also be defined in terms of teams or sales territories. The access could also come from the user being in a management hierarchy. All these access rules are stored as seeded sub-queries in the application and get appended at run-time to a dynamically generated SQL. As such, the developer does not have much control over how the SQL is written.

2.1. Existing SQLs, Performance and Execution Plans

A sample SQL text, its performance metrics and execution plan are presented below.
Figure 1. Existing SQL Outer Query
Figure 2. Existing Data Security Sub-Queries
A sample SQL is provided in two parts above. The first part is the outer SQL that is generated at run time by the application engine. The second part is the data security predicate sub-queries that are appended to the outer SQL at run time based on the identity of the application user logged in.

The performance of such a SQL against an Oracle 11gR2 database was measured using a benchmarking Java tool for fetching the first 25 rows from the database.

![Table of bind values and SQL benchmark metrics]

Figure 3. Bind values and SQL benchmark metrics

The execution plan chosen by the optimizer is shown in Figure 4 below.
2.1.1 Issues Affecting Existing SQLs’ Performance

In SQL tuning exercises, the first aim is to identify the issues in the execution plan and the reasons for the optimizer to choose the plan. It is not always an easy task but needs careful understanding of the business functionality that the SQL is trying to achieve as well as a deep understanding of the various operations that the optimizer uses. It also needs constant updating of one’s knowledge of the optimizer features and behaviour changes that come with changed and new database parameters. Needless to say, it can become quite complex.
As seen in the performance metrics above in Figure 3, the performance of the SQL is quite poor, both in terms of time taken as well as the number of buffer gets that the optimizer had to process. Even though the plan cost came up low at 145, it is apparent that the optimizer did not do a very good job at estimations. The benchmark metrics show that the database took more than 5 seconds warm time with an extremely high 2.5 million buffer gets to return 25 rows of the result set.

The execution plan shows that even though the outer query processed very few qualifying rows, the optimizer had to evaluate all the security predicates’ sub-queries to find out which rows the user was entitled to see, only to eliminate most of them later, based on the outer query. This is a typical “tiny-huge, huge-tiny” problem. It means that the optimizer has to make a quick decision whether to drive from the outer query or drive from the security predicate sub-queries. Which one of these will result in more restrictive row sets is always a difficult question to answer. This decision making is complicated by the fact that the SQL has many tables and joins and it uses bind variables which make estimating the cardinality that much more complicated in the process of deciding on an execution plan.

The analysis of the SQL, execution plan and schema helped in narrowing down to broadly three issues. The three broad areas affecting the existing SQL’s performance are the security predicates’ sub-queries and the associated “tiny-huge, huge-tiny” problem, the main filter predicate columns coming from different tables and lastly inadequacy of existing indexes.

The two main filtering predicates in the outer SQL are:

\[
\text{AND ( } (\text{OpportunityEO.STATUS_CD} = :\text{BindDefaultStatusCd} ) \\
\text{AND (RevenueEO.EFFECTIVE\_DATE BETWEEN :BindEffectiveBeginDate AND :BindEffectiveEndDate ) } )
\]

As seen above, the Effective\_Date column is from the Revenue table while Status\_Cd column is from the Opportunity table. This makes the optimizer’s job difficult.

The “tiny-huge, huge-tiny” problem is partly due to the fact that the data security sub-queries do not have any of the two filtering predicates that the outer query has i.e. EFFECTIVE\_DATE and STATUS\_Cd.

Finally, there can be additional indexes that may help performance.

3. SUGGESTED APPROACH TO IMPROVE SQL PERFORMANCE

The resolutions for the three issues identified above can now be easily tried and tested using the same Java benchmark tool.

First, let us try to resolve the issue of filter predicate columns coming from different tables. To make the optimizer’s job a little easier, it is sometimes a good idea to de-normalize such filter predicate columns. In the present case, adding EFFECTIVE\_DATE column to MOO\_OPTY table and adding EFFECTIVE\_DATE, STATUS\_Cd to MOO\_OPTY\_RESOURCES table will make the optimizer’s choices less complex and as a result more stable and predictable. In addition, appropriate indexes will also be needed on these columns. Thus, the outer query would now have the following filter predicates from the same Opportunity table:

\[
\text{AND (}(\text{OpportunityEO.STATUS_CD} = :\text{BindDefaultStatusCd})
\text{AND (OpportunityEO.EFFECTIVE\_DATE BETWEEN :BindEffectiveBeginDate AND :BindEffectiveEndDate )})
\]

The resolution for the “tiny-huge, huge-tiny” issue is a little trickier. One approach that has worked consistently well is to create a wrapper around the data security predicates’ sub-queries,
use DISTINCT clause in the wrapper as well as data security sub-queries and also push the outer query’s filter predicates into the data security wrapper code. The changed outer SQL would thus be as in Figure 5 below and the changed data security predicates sub-queries would be rewritten as in Figure 6 below.

Figure 5. Rewritten Outer Query Using De-normalized Columns
Figure 6. Rewritten Data Security Sub-Queries with Wrapper Code & DISTINCT Clause
Lastly, to support the above filter predicate changes and the rewritten data security sub-queries, the following indexes were created:

MOO_OPTY (OWNER_RESOURCE_ID, EFFECTIVE_DATE, STATUS_CD, SUM_REVN_ID)

MOO_OPTY (OPTY_ID, SUM_REVN_ID)

MOO_OPTY.Resources (RESOURCE_ID, EFFECTIVE_DATE, STATUS_CD, OPTY_ID)

MOO_REVN (OPTY_ID, OWNER_RESOURCE_ID, EFFECTIVE_DATE, PRIMARY_FLAG, STATUS_CODE)

3.1. Benchmark Metrics and Execution Plan of Rewritten SQL

The three-pronged strategy described above worked very well and when the SQL was benchmarked using the same Java tool against the same database, results were drastically improved both in terms of warm time as well as the buffer gets. As seen in Figure 7 below, the warm time was only 365 milliseconds and buffer gets were down to 9697.

![Figure 7. SQL Benchmark Metrics for Rewritten SQL](image)

The execution plan for the rewritten SQL is as seen in Figure 8 below. Notice the much reduced A-Rows column as well as reduced buffers in the plan, which shows the levelling of the outer and data security sub-queries’ cardinality achieved using the suggested approach.
<table>
<thead>
<tr>
<th>No.</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>Ends</th>
<th>Cost (CPU)</th>
<th>Amount</th>
<th>Access Time</th>
<th>Suffers</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>UPDATE INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>INSERT INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>DELETE INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>CREATE INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>INDEX RANGE SCAN</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>INDEX SPACER</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>TABLE ACCESS BY INDEX</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>DML</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>DDL</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>VIEW</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>VIEW</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>VIEW</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>VIEW</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>VIEW</td>
<td>STATEMENTS</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

64
3.2. Confirmation of Performance Improvements Using Above Approach

The approach described and detailed above was tried on many different variations of SQLs for different roles, responsibilities and users. The performance metrics and execution plans for the rewritten SQL were always better than the original SQLs.

A summary of the benchmark results for different scenarios is provided below in Figures 9 and 10.
4. CONCLUSION

Most commercial business applications today like Sales Automation are built around technology layers that generate physical SQLs at run time. Access to data in many such applications is restricted on a need to know basis usually by implementing security and visibility through a set of roles and responsibilities that each are defined by way of sub-queries that get appended to the main SQL at run time. This method of adding sub-queries cumulatively through a security framework results in complex SQLs, especially for users who are granted many different roles and responsibilities. The main query is usually referred to as the outer query and the appended security predicates’ sub-queries are termed as the inner query. When such SQLs arrive at the database, the CBO evaluates them for possible access paths and join optimizations to decide on an execution plan based on available statistics. Very often, the CBO is posed with the dilemma of whether to drive from the outer query or drive from the inner query. For some users, the outer query with its filter predicates can be greatly restrictive while for other users, the inner security predicates’ sub-queries can be more restrictive. This if sometimes referred to as the “Tiny-Huge, Huge-Tiny” problem that many application and database designers struggle to manage. Many times, this problem leads to poor choices by the CBO, resulting in sub-optimal execution plans.
leading to poor query response times and consequently causing performance and scalability issues for the application as well as the database.

This paper has presented the analysis of such SQLs and CBO execution plans from the Opportunity Management module of a CRM application that performed poorly due to the aforesaid “Tiny-Huge, Huge-Tiny” problem. Based on the analysis, the paper then presented a suggested solution incorporating some de-normalized columns and rewrite of the security predicates’ sub-queries that result in vastly improved performance and scalability of such queries and consequently of the application.

This approach can be suitably amended and applied to different applications based on the specifics of any such similar SQL performance issues.

REFERENCES


Authors

1. Mr. Arjun Sirohi is currently a Consulting Member of Technical Staff in the PSR Engineering Division (Performance Scalability and Reliability) at Oracle USA located in Bellevue, WA, USA. He holds a MS CS degree from City University of Seattle, from where he graduated with President’s Honours with 3.93 GPA.

2. He has 21 years of work experience in software design, development and implementation in various roles as Developer, Database Consultant, Architecture Specialist and Senior Systems Manager. He has extensive experience in application, SQL, database and system performance tuning, optimizations and testing. Apart from databases, he has deep expertise in data warehousing and business intelligence applications' performance and scalability.