Performance Optimization in Transactional Business Intelligence Applications’ Query-Generation Development Cycle

Arjun K Sirohi and Vidushi Sharma
Department Computer Science and Engineering, School of Information and Communication Technology, Gautam Buddha University, Yamuna Expressway, Gautam Budh Nagar, Near Pari Chowk, Greater Noida – 201312, Uttar Pradesh, India; Asirohi@yahoo.com, Vidushi@gbu.ac.in

Abstract

Objectives: To develop optimization techniques that can apply to any Transactional Business Intelligence (TBI) Applications development process leading to significant improvements in the performance and scalability of TBI queries. Methods/Statistical analysis: Benchmark experiments were conducted using Oracle RDBMS 11gR2 using representative SQL queries from Oracle’s Fusion TBI Applications. All four measures of SQL performance viz. SQL Response Time (RT), Input-Output (IO) Buffer Gets, Hard-Parse-Time and Shared Memory utilization were recorded with and without our proposed optimizations. The four performance measures were then compared to record actual improvements. Findings: Oracle TBI Applications are the result of a convergence of many technologies like the Application Development Framework (ADF), Web Logic Server (WLS) and Oracle Business Intelligence Enterprise Edition (OBI EE). TBI queries are generated at runtime by the OBI EE and ADF layers based on the transactional application schema, the ADF model and the logical, business and physical model layers of OBI EE metadata and optimizing this end to end process was the focus of our research. The benchmark experiments established very promising results. We recorded repeatable, significant gains in not across all measures of SQL performance. Our findings and recommendations can optimize both query and application performance. Adoption of our proposed solutions by OBI EE and TBI Fusion applications has provided significant performance and scalability improvements in enterprise OTBI applications. Even though our findings and recommendations were derived during a multi-year ADF-OBIEE application development environment, we are confident that these lessons would provide good guidance for developers embarking on architecting any new TBI applications. Application/Improvements: Our proposed recommendations can improve performance of all TBI applications that use an abstracted model and metadata of the transactional data model with query-generation engines to generate TBI SQL queries.

Keywords: OTBI, Oracle, Performance Optimization, Query Generation, SQL Performance, Transactional Business Intelligence Queries

1. Introduction

TBI applications use three-tier architecture to provide business users the capability of real-time business intelligence from transactional business applications. Oracle has developed the next generation of enterprise applications named Fusion Applications by combining the best of breed applications like Siebel, Peoplesoft, E-Business Suite and JD Edwards. This has been done by integrating these with Oracle Fusion Middleware like WLS and ADF. On top of these, Oracle uses Business Intelligence technology platforms like Business Intelligence Server Enterprise Edition (OBI EE), Business Intelligence Presentation Server, Business Intelligence Publisher and Essbase for creating business analytics and reporting solutions. Each of the Oracle application and the technology is targeted for a specific set of reporting needs based on the strengths of the underlying tools and architecture.
Developing such reporting solutions has been a challenge as it involves multiple and complex technologies and many layers of modeling and abstraction that create an enormous amount of complexity not only in the architecture but also in the way SQL queries are generated at run time.

We analyzed hundreds of poorly performing Fusion Applications TBI SQLs. Our analysis has showed that most performance problems were a result of one or more of the limitations/issues discussed in this paper. Due to the nature of applications development and the ever-evolving technology stacks, this paper attempts to document early findings and recommendations to address the performance and scalability issues affecting TBI queries.

2. Current Architecture, Motivation and Research Objective

Oracle TBI is the real-time operational analytics reporting solution for Fusion Applications that allows the ability to create real-time analysis of business data against the transactional Fusion applications schema to support current state reporting and real-time analysis of their operations. These include Operational Reporting, Embedded Business Intelligence Dashboards and Transactional Dashboards that use only transactional data. The architecture is depicted in Figure 1. Since these TBI applications are built on top of many different technologies, some fitted into the stack retrospectively; the SQL queries generated are very large and complex. The complexity and size of the SQL queries present serious performance and scalability challenges. Since TBI queries are generated at run-time, they cannot be optimized in advance. Many researchers have worked on related topics such as object-relational mapping, software application development project and SQL injection in web applications. Query RT, Database Hard Parse Time, Logical I/O or Buffer Gets, SQL-shared-memory are the four parameters which are generally used in the industry to compare performance of SQL queries Figure 2. Over the period of our research, we aimed at improving the performance of TBI SQL queries using these four parameters. At the end of our paper we provide measurements for all of these four parameters and the improvements recorded by using the techniques proposed by us.

Figure 1. Oracle TBI architecture diagram.

Figure 2. Oracle TBI architecture diagram.

3. Research Methodology, Analysis, Findings and Innovative Solutions

Capturing of TBI SQLs from various sources and their analysis has been an ongoing process for the past few years. The Oracle database version used for this analysis is 11gR2. Through an extensive analysis of TBI SQLs’ text, optimizer execution plans, OBI EE and transactional applications’ data model, a number of generic issues were identified by us. We benchmarked many hundred SQL queries from TBI applications and found that in many cases performance was sub-optimal due to incorrect exe-
cution plan selection by the database optimizer. The root cause analyses of hundreds of such SQLs highlighted one important aspect-- there was a missing synthesis between the SQL query generation mechanism and the optimizer and query execution engine of the database. We present our detailed findings and recommendations in subsequent sections.

3.1 Frequent Hard Parsing of SQLs in Database and Non-sharing of SQL Cursors.

The analysis of a number of TBI SQLs in customer deployments as well as internal testing showed that frequent hard parsing of SQLs by the database was a major factor in poor response times. This problem was further compounded by the fact that TBI SQLs are complex and quite large, thereby incurring a high parse time. The many different reasons for hard parsing and their resolutions are discussed in following sub-sections.

3.2 Lack of Support for Bind Variables by BI Server Results in Hard Parses

During analysis of TBI SQLs' text and the execution plans, we found that the use of literals in the SQLs generated by BI Server was due to the inability of BI Server to support bind variables and this caused multiple performance related problems. For example, in Fusion CRM Lead Management User Interface (UI), the user is presented with a slider to apply a lead filter titled ‘Likelihood to Buy’. When the user moves the slider on the UI, a new physical SQL was generated with a new literal value only for the one column involved. When this SQL reaches the database, the optimizer treats this physical SQL as a new SQL in absence of bind variables and as a result, it hard parses the SQL for every new literal value used by the user. Even for the same literal value, the database can use cardinality feedback feature to produce child cursors a second, third or fourth time causing performance slowness seen in the UI. When the next SQL comes in with same type and number of literals, the optimizer can share the cursor from previous executions. The effect of high parse time and inability to share cursors puts pressure on buffer cache and shared pool resources and directly affects the overall performance of the OLTP database, sometimes causing even the transactional applications' SQLs to inadvertently suffer performance consequences. While the BI Server was initially meant for data warehouse and BI environments and thus lacked support for bind variables, the very wide use of TBI in Fusion Apps product families brings out this big shortcoming that seriously affects the performance and scalability of TBI. Typically, in a transactional database, we expect that application SQLs that are similar except for having some literal values would use bind variables and those SQL cursors would be shared. Since TBI SQLs go against the same transactional database, we expect the use of bind variables. However, we found that SQLs that only varied in one or more literal values were being hard parsed again and again and thus preventing cursor sharing. The reason for this was that OBI EE does not support the use of bind variables. It was designed and created for data warehouse environments where parse time was a relatively small compared to the total query processing time. However, the expectations in a transactional system and database are sub-second response time and cursor sharing. Thus, no matter how well a SQL was tuned, the hard parse time would not allow the performance to reach sub-second response time. The ideal resolution of the problem of cursor sharing is to use parameter markers or bind variables by providing such support in platforms like OBI EE. A workaround and temporary solution to provide some performance relief for use cases where use of literals and hard parse is the primary source of performance problem is setting the database parameter CURSOR_SHARING to the value ‘FORCE’ in the RPD connection pool properties. This setting can significantly improve cursor sharing, resulting in reduced memory usage, faster parses, and reduced latch contention.

3.3 Processing of Big and Varying IN Lists

In TBI SQLs, we see large IN lists that vary in the number of values. Even with the temporary solution until bind variables are supported by OBI EE Server to set SHARED_CURSOR = ‘FORCE’ setting in the RPD connection pool, the problem of hard parse does not completely get resolved because of the varying list of values in these large IN lists. When the database executes such a SQL, it converts the literals to system bind variables. A hard parse occurs at this time. Now if the next SQL comes in with same type and number of literals, the optimizer can share the cursor from previous executions. However, hard parses occur
Performance Optimization in Transactional Business Intelligence Applications' Query-Generation Development Cycle

3.4 Ongoing Discovery of Optimizer Bugs Causing Repeated Hard Parsing

Since data optimizer keeps getting more complex over time, finding bugs was an ongoing issue. Though many optimizer bugs are fixed as soon as they are discovered, some unknown ones cause cursors to be not shared resulting in repeated and frequent hard parsing. A few evaluated by us were: Bug 13648166 - SQLPLAN IS NOT STABILIZED DUE TO OPTIMIZER FEEDBACK, BUG 8608703 - TST&PERF: CHILD CURSOR WITH THE WORST PLAN HAS THE MOST, BUG 9465425 - NEW CURSORS KEEPS GENERATED AFTER CARDINALITY FEEDBACK and BUG 8521689 - PERFORMANCE REGRESSION WHEN RUNNING QUERY MULTIPLE TIMES. As a result, applying RDBMS patches frequently is necessary.

3.5 TBI SQLs' Extensive use of With Clause and Limitations on Cost Based Query Transformation (CBQT) in Database

Analysis of TBI SQLs execution plans revealed a big shortcoming with the Oracle CBO. Though the WITH clause was available as a RDBMS feature for a long time, certain CBO features like Cost Based Query Transformation (CBQT) were not used while evaluating the choices of execution plans for such SQLs. OBI EE query-generation engine generates SQLs which utilize the WITH clause extensively to add clarity as well as to make use of intermediate results. The use of WITH clause or common table expression was one of the major factors contributing to poor execution plans. After this issue was reported, some relief was made available through RDBMS Patch 11740670 which on its own resolves the issue for queries where WITH views are referenced once in the query. However, the remaining queries where WITH views appear more than once can only benefit by setting the parameter '_with_subquery' = 'inline' at the session level or as a hint in the SQLs. Until RDBMS 12.2, it was an all or nothing situation for SQLs using WITH views referenced more than once. For a number of TBI SQLs, the optimal execution plan may require some WITH views to be materialized and some others in-lined. With the current CBO limitation, this is currently not possible, except to manually enter MATERIALIZE hint in the View Objects (VOs) with specific and detailed knowledge of related SQLs and their expected optimal execution plans. To make full use of the database optimizer patch for fixing the performance issues related to the use of WITH clause, especially for use cases that have multiple references to the WITH clause, an additional step is required in the process. The database hidden parameter that controls the CBO behavior needs to be set as '_with_subquery' = 'inline' in the connection pool in order to have the CBO enable CBQT on SQLs that have multiple references to a WITH clause sub-query.

3.6 Blind Aggregation Queries with No Good Filter Predicates Cause Poor Execution Plans and Performance

We found that this was another reason for poor performance that stood out prominently and was the cause of user complaints about slow log-in. The expectation that the BI Server and database would be able to crunch data from a transactional system in seconds, likely led to this cause. For example, in one use case, the user login involved use of aggregation queries with no restrictive filtering criteria in the landing page. In user testing, since the data volume was low, the queries passed performance muster. However, when the data volume increased, the login became extremely slow due to run-time aggregations for every user logging in. In such cases, such aggregation queries in landing page should be avoided. Aggregation queries must have some additional filter criteria like time dimension filters.

3.6 Absence of Table and Column Pruning During SQL Generation

Table and column pruning feature of ADF can significantly help reduce the complexity of SQLs, resulting in lower parse times as well as reduced execution times. However, our analysis showed that there were many instances where such pruning was not happening for various reasons. Some of the identified reasons are discussed here.
3.7 Expert Mode VOs Prevent Pruning, Leading to Poor Execution Plans and Performance

ADF is designed to prune tables and columns at run time based on columns selected in the SELECT clause of the SQL. The pre-requisite for this to happen is that VOs should have been created in declarative mode. However, in a number of cases, for various reasons, TBI VOs were found to have been created in expert mode, thereby preventing table and column pruning, leading to bloated SQLs with unnecessary joins and complexities. For example, one reason for creation of expert mode VOs was related to the BI Server’s limitation in supporting bind variables and ADF only supports use of bind variables or literals as values for conditions while creating View Criteria (VC) for the VO. Thus, for example, SYSDATE cannot be used as a value in a VC. Currently there is a problem to achieve filters like ‘SYSDATE between VO.start_date_attr and VO.end_date_attr’ in declarative mode TBI VOs. Due to these limitations, some TBI VOs were created in expert mode instead of creating them in declarative mode. Workarounds have been suggested for some cases. The recommendation is to enhance ADF to support use of database functions like SYSDATE in View Criteria to enable use of declarative VOs and thus table and column pruning feature of ADF could kick in. Since BI Server does not currently support bind variables, TBI VO view criteria cannot use SYSDATE. Support for bind variables in OBI EE Server may require a significant development effort, though very desirable and needed in the long run. While that project may take time, an enhancement in ADF to support database functions like SYSDATE could allow some of the TBI VOs to switch from expert mode to declarative mode. This small enhancement in ADF would enable such TBI VOs to participate in table and column pruning which are known factors contributing to simplification of SQLs at run time. Once this is achieved, the run time SQLs generated would be simplified and will generally result in lower hard parse time and more stable execution plans in the database. Thus TBI SQLs currently affected by this limitation and ones that currently use expert mode VOs will start to perform better leading to overall improvement in performance and scalability. As an interim solution, it is suggested to add a tag called ‘IsSqlFragment’ in VO.xml and then refer to sydsate in the view criteria definition. The physical query thus generated correctly gets the date filter condition. We recommend that as far as possible, all efforts must be made to use declarative mode VOs for TBI to reduce the complexity of such SQLs. Through table and column pruning on declarative VOs, the optimizer can potentially reduce hard parse time while producing more efficient execution plans. The end result would be improved performance of TBI SQL queries in terms of execution time and buffer gets.

3.8 Incorrect RPD Data Modeling

Our analysis found that some of the table and column pruning problems discovered during the analysis were due to developer’s mistake about the setting of column properties. In one of the use cases identified during OUR analysis, it was determined that the data model in the RPD had been constructed with the cardinality as ‘(0 or 1) to many’. With this type of cardinality, the BI Server cannot remove a table on the ‘(0, 1)’ side since it may incorrectly change the result set. The resolution to this problem was to verify if the backend data is ‘1 to many’ relationship and once verified, make changes in the RPD to change it to ‘1 to many’ join instead of ‘(0 or 1) to many’. This helped resolve the pruning in the particular case.

3.9 Unnecessary Tables/Columns Inadvertently Being Added to Physical SQLs

In some TBI SQLs, we found that unnecessary columns were being added to the physical SQLs even when they were not selected by the user in the report. Analysis showed that this was a result of setting incorrect properties for such columns in TBI VOs definition (isSelected=false). The resolution was simply to make corrections to the properties for these columns in the TBI VOs.

3.10 Use of Language Views (VL) Instead of Base Tables Leads to Unnecessary Joins and Complexity

Even though this finding was not directly related to ADF table/column pruning, the effect of using VL views where it would be sufficient to use base tables directly is the same – it leads to unnecessary tables and columns participating in TBI SQLs, thereby leading to unnecessary joins, complexity, higher parse times and sometimes poor execution plans. We recommend that base tables be used in all TBI VOs as far as possible.
3.11 Large Number of Performance Issues Due to RPD Model Mismatch with Application Schema/Indexes

During our analysis we discovered that many of the issues that affect performance of TBI SQL queries were primarily due to the way RPD modeling was done. The RPD data model presents to the TBI users a number of different attributes in the presentation layer. Some of these map back to logical columns that have transformations enabled on top of them. Transformations on a number of such columns were coded in order to provide the capability of dual use in BI Applications and TBI. The idea behind this dual purpose coding was to make it easy to switch from TBI to BI Applications smoothly. However, for hosted and cloud services, TBI is the primary reporting solution. And the recommended goal should be to ensure that TBI reports and analyses should not involve any columns or attributes that have transformations enabled. Once this is achieved, a number of SQLs that involved use of columns with transformations like use of NVL() function, CASE statements, CAST, TO_CHAR and CONCAT functions etc. would become simplified. The simplified SQLs would perform better because without unnecessary transformation on the columns, the CBO will be able to make use of available indexes on such columns and arrive at better and more efficient execution plans. The most common issues under this category are discussed here.

3.12 Use of NVL() Function on Index Columns Cause Poor Execution Plans Leading to Poor Performance

Our analysis of execution plans revealed that the use of NVL() function on columns that were already indexed in the database was causing the CBO to not use the index, thereby leading to full table scan operations in the execution plans, thereby causing poor performance.

3.13 Lack of Indexes on Filter Predicate Columns Caused Resulted in Bad Execution Plan Choices

In a number of cases, we found that even though columns were being used to filter data from reports, there were no corresponding indexes on those columns in the database. This led to full table scans on some very large tables, a poor execution plan and thus poor SQL performance.

3.14 Multi-Level Nested Views and Joins on Those Views Result in Bad Execution Plans and Performance

Another problem we found was that database views were being used in SQLs including in join conditions with other tables and views. The problem was further aggravated by the fact that some of these view had multiple levels, meaning there were views on top of views. In such cases, the CBO is confronted with very complex access plan decision making and very often led to poor execution plans.

3.15 Unnecessary Checking for Column is Null or Column is Not Null on Columns Defined as Not Null in Database

In many cases, we found that the SQLs contained clauses to check for a column to be null or not, even though the column was already defined in the database as not null.

3.16 Use of NVL Function on Columns Defined as Not Null in Database Schema

Another variation of the use of NVL function was its use on columns that were already defined in the database schema as not null. This was the cause of performance degradation for many such SQLs.

3.17 Flag Columns Expected to be Not Null Defined as Null in the Database Schema

While analyzing the SQLs and execution plans to see why null checks for flag columns were being included in the SQLs, we found that even though such flag columns were expected to always contain data, some of such flag columns were defined in the schema as nullable. This was the reason why the null checks were being added in SQLs, causing performance problems.

3.18 Lack of Indexes on Join Columns Leading to Poor Execution Plans and Performance

From some of the execution plans analyzed, we found that there were many instances where a column was being used in a join condition in the SQL but there was no index on such column. This led to full table scans on the table, which is a costly operation in transactional databases.
3.19 Search Using Upper Function on Columns with No Corresponding Function Based Index

This was a widely used function where searches were being done using UPPER function on columns that were indexed but no function based index was created to support the UPPER function search. This led to full table scans on large tables, thereby resulting in very poor performance.

3.20 Use of CASE Statement on Column to Check for NULL Value Even Though Column Defined as NOT NULL in Database Schema.

This was yet another variation of the check for null or not null in TBI SQLs even though that column is defined as a not nullable column in the database schema. This directly impacts the execution plan choices by the CBO and prevents use of indexes, even though available for the column. An example is shown below.

```sql
WHERE ( CASE
    WHEN T1265466.C51474716 IS NULL
    THEN '__UNASSIGNED__'
    ELSE T1265466.C51474716
END = 'QUALIFIED')
),
SAWITH1 AS
(SELECT D1.c1 AS c2,
 CASE
    WHEN D1.c10 IS NULL
    THEN '__UNASSIGNED__'
    ELSE D1.c10
END

In the SQL, Column T1265466.C51474716 was mapped to V271163669.LOOKUP_CODE which in turn was the Table. Column MKL_LOOKUPS.LOOKUP_CODE.

In the database schema, MKL_LOOKUPS was a view where column LOOKUP_CODE came from table FND_LOOKUP_VALUES_B in which LOOKUP_CODE was defined as a NOT NULL column. As such, above CASE statements were incorrectly used. Further investigation in the RPD revealed that the presentation column 'Lead Status' 'Lead Status Code' was mapped to logical column 'Source Marketing Lead Status Code' in the Logical Table Source (LTS) 'Dim_CrmLeadAM_MarketingLeadStatus' where it was mapped to an expression as under:

`CASE WHEN
    'oracle.apps.crm.model.analytics.applicationModule.CrmAnalyticsAM_CrmAnalytics.AMLocal'"".Dim_CrmLeadAM_MarketingLeadStatus.s".LookupCode'IS NULL
    THEN '__UNASSIGNED__'
    ELSE
    'oracle.apps.crm.model.analytics.applicationModule.CrmAnalyticsAM_CrmAnalytics.AMLocal"".Dim_CrmLeadAM_MarketingLeadStatus.s".LookupCode'END`

This was despite the fact that the LTS 'Dim_CrmLeadAM_MarketingLeadStatus' was mapped to Physical Table 'Dim_CrmLeadAM_MarketingLeadStatus' where column 'LookupCode' was defined as 'Nullable'='False' correctly matching the database schema for FND_LOOKUP_VALUES_B in which LOOKUP_CODE is defined as a NOT NULL column. As seen above, the root cause of this issue is the use of the CASE expression for the logical column 'Source Marketing Lead Status Code' in LTS 'Dim_CrmLeadAM_MarketingLeadStatus', causing poor execution plan and performance.

3.21 Use of NVL on Column that has Default Value Specified in Database Schema

Analysis of execution plans, SQLs and corresponding database schema revealed that in some cases, the database function NVL was being used on columns that had a default value specified for them, meaning that there was no expectation for these columns to store null values. For example, consider the SQL extract 'SELECT NVL (DimMemFlag.EXPLICIT_FLAG, 'N') FROM MOT_TERR_DIM_MEM_FLAGS DimMemFlag'. On checking the database physical schema, we found that the column EXPLICIT_FLAG was defined as 'EXPLICIT_FLAG VARCHAR2 (1 CHAR) DEFAULT 'N', which showed that 'N' were defined as the default value during table creation. So this was incorrect in more than one way. On the schema side, the data model should define it as a NOT NULL column with the default value. Second, even with current schema definition, there is no expectation of this column storing a null value for any row since at the time of committing a row, the database would auto-
3.22 Filter Predicate Columns being Converted to String Type Using CAST and CONCAT Functions Preventing Use of Indexes by CBO.

This was a frequent occurrence in TBI SQLs analyzed. The use of functions like CAST and CONCAT on columns that are otherwise indexed, results in CBO not being able to use the available indexes, resulting in poor execution plans and wasted effort. An example is highlighted below. 'AND concat(CAST(T1265466.C331077375 AS CHARACTER ( 4 ) ), concat(' Q ', CAST(T1265466.C269650446 AS CHARACTER ( 1 ) ))) = '2011 Q 4' '.

The column mapping is as follows: V524003964.FISCAL_YEAR_NUMBER AS C331077375 -> P.PERIOD_YEAR.
FISCAL_YEAR_NUMBER -> GL_PERIODS.
PERIOD_YEAR V524003964.FISCAL_QUARTER_NUMBER AS C269650446 -> P.QUARTER_NUM.
FISCAL_QUARTER_NUMBER -> GL_PERIODS.
QUARTER_NUM. The CAST and CONCAT functions are applied on the columns due to configuration in the BI RPD. Simply changing these filter predicates to 'AND T1265466.C331077375 = 2011 AND T1265466.C269650446 = 4' resolves the issue because the optimizer can then use indexes.

3.23 Language-Translation OR Predicates Added to the SQL Due to RPD Configuration and Unnecessary OUTER JOINS in the SQL

The use of outer joins when not really necessary can add extra costs in the execution plans and I found that there were number of such use cases. An example is discussed below to highlight the issue. 'FROM INV_ORG_PARAMETERS OrganizationParameterPEO, HR_ORGANIZATION_UNITS_F_TL OrganizationUnitTranslationPEO, HR_ORGANIZATION_UNITS_F_TL InvOrgNamePEO WHERE OrganizationParameterPEO. MASTER_ORGANIZATION_ID = InvOrgNamePEO. MASTER_ORGANIZATION_ID ('+) AND OrganizationParameterPEO. ORGANIZATION_ID = InvOrgNamePEO. ORGANIZATION_ID (+)'.

We found that the outer join, denoted by the (+) sign in the SQL extract above was not necessary. It was being added was due to an incorrect understanding on the part of RPD data modelers that this is a functional requirement and that they did not have a choice but to introduce these conditions. Follow up discussion revealed that these OUTER JOINs were not really needed since INV_ORG_PARAMETERS.MASTER_ORGANIZATION_ID column is a required column and will always be populated. In addition, INV_ORG_PARAMETERS.ORGANIZATION_ID is already a NOT NULL column. The revised SQL performed best if we remove the outer joins and the OR predicates 'OR ((V378971821.LANGUAGE IS NULL))' and 'OR ((V378971821.LANGUAGE1 IS NULL))' from the SQL.

3.24 Inability of CBO to Either Merge View or Push Join Predicate Due to Outer Join on Literals

Another problem affecting performance of TBI SQLs was related to the use of OUTER JOIN on literals preventing the CBO to either merge a view or push the join predicate. There may be two resolutions for this issue. One, removing the outer joins to literals ('HZ_PARTIES'), ('Y'), ('A') and ('SITE') as these are already included in the view definition. Two, removing the outer join to ('MOT_INDUSTRY_CLASSCATEGORY') by creating a new Entity Object (EO) association, which is a problem in distributed query processing.

3.25 Joins on Views Cause Full Table Scans on Tables

Joining on views is not a recommended best practice for SQL coding, especially for OLTP applications. The reason is that views mask the complexity of the underlying SQL and then joining View in a SQL further increases the complexity that the CBO needs to deal with while making choices for execution plans. We found that the use of views and joins on those views was not an uncommon practice in TBI VOs. An example is discussed to highlight the problem. QSC_PROD_GRP_ITEMS_TREE_V and EGP_SYSTEM_ITEMS_VL QSC_PROD_GROUP_ITEMS table comes from view QSC_PROD_GRP_ITEMS TREE_V in SQL with no good filter predicates. The
3.26 Very High Hard Parse Times for TBI SQLs Leading to Poor Response Time

By their very nature, TBI SQLs generated are quite complex and huge since they are aggregation queries using the data warehousing concept of Facts and Dimensions but modeled and executed against a transactional database schema. As such, we found that the parse time for such SQLs was often very high. This factor, combined with the fact that SQL cursors were not being shared due to use of literals instead of bind variables, compounded the performance problems and any reasonable expectation of sub-second response time performance was inhibited by these two reasons. Simplification of SQLs by way of VO pruning for declarative mode VOs, removal of unnecessary tables and joins, removal of unnecessary functions on columns etcetera can help in reducing parse times and better selection of access paths.

3.27 Time Dimension Related Performance Issues

To achieve business intelligence and analytics type of analyses, the TBI architecture involves use of complex joins and on-the-fly transformations due to the normalized data model. These types of joins and transformations are normally executed during the Extract, Transform, and Load (ETL) process for creation of a star schema in the data warehouse from the transactional database. For TBI, this means that such joins and transformations must happen at run time to be able to use the same star schema concept over an OLTP database schema. This creates huge complexity in the final SQLs generated at run time. These causes high parse time as well as complex and lengthy execution plans resulting in performance bottlenecks. Time dimension in TBI comes from a VO which has a join of following tables in the transactional database: FND_CAL_DAY, GL_PERIODS, GL_CALENDARS, FND_PROFILE_OPTIONS, and FND_PROFILE_OPTION_VALUES. Another performance issue was caused due to the fact that the queries join on FND_CAL_DAY.REPORT_DATE column while the filter predicate is from GL_PERIODS table in the form: TO_CHAR(P.PERIOD_YEAR) || ' Q ' || TO_CHAR(P.QUARTER_NUM). This causes a poor execution plan to be chosen by the CBO. Yet another problem is the use of TRUNC function on date columns in the SQLs. There could be multiple possible resolutions to this problem. One would be to materialize the definition of time-period VO V524003964 and create indexes on ENTERPRISEQUARTER and ENTERPRISE_YEAR columns. Under the scenario above, creating function based index on MOO_REVN (TRUNC(EFFECTIVE_DATE)) could help improve performance as the optimizer would likely use the indexes on newly created TIME dimension table as well as on MOO_REVN.

3.28 Product Dimension Related Performance Issues

Similar to Time Dimension related issues, the current design in TBI to support Product Dimension is very complex as it requires combining data from multiple OLTP tables and views. This creates huge complexity in the final SQLs generated at run time which causes high parse time as well as complex and lengthy execution plans resulting in performance bottlenecks. By using views and joins as well as using filter predicates within _VL language views in Fusion Applications, the performance problems in down-stream TBI applications were exacerbated. The vast majority of all VL views are simply a join between the B and TL tables along with LANG filter, which is what it should be. However, in some odd cases, we find that Dev has interjected functionally required filters or functions into the VL views. On checking, we find that there is no standard described regarding creation of VL views. In such cases, when the consumer Dev teams uptake and use this VL view in joins, we see serious performance issues. This problem becomes even more in TBI SQLs. One such case is EGP_SYSTEM_ITEMS_VL that we have seen to cause performance issues. One possible resolution for TBI would be to materialize the underlying database objects used in product dimension in order to not execute such complex logic at run time.

3.29 Data Security Related Performance Problems

Through our analysis of hundreds of TBI SQL queries, their execution plans and performance parameters
pointed to the specific implementation of RBAC data security model for row-level data security wherein a UNION of all DSPs’ sub-queries was applied on the main user query to limit the visible rows for the user\textsuperscript{16}. In the case of TBI applications, all VOs were being secured by ensuring that ALL applicable DSP sub-queries for a user got appended to the VO at run time. There are many performance problems that arise from such implementation as discussed below.

3.30 Complexity and Non-Standardization of RBAC Sub-Queries

Here, our recommendation is that the seeded DSP sub-queries should be rewritten in a standardized manner so that the optimizer is presented with sufficient opportunities to make informed decisions about access paths and execution plan selections.

3.31 Lack of Context-Sensitivity Adversely Affects Performance

By appending all DSP sub-queries at run-time to the main query without regard to the context in which the user navigates the application, we add unnecessary baggage to the SQL query and consequently, the database optimizer has to do additional work which is avoidable. The recommendation here is to strip out the unnecessary DSP sub-queries and make the TBI SQL queries lightweight while maintaining functionality.

3.32 Security Predicates Duplicated When Multiple View Objects Involved in TBI SQL

From our analysis of TBI SQLs related to data security, we found that there was an architectural issue related to Data Security Predicates (DSPs) and TBI VOs/RPD modeling. For use cases involving Fact and Dimension VOs where both are secured individually and when used together, in the physical SQLs we find that that DSP sub-queries are appended twice – once on the dimension VO and then again on the Fact VO and then these are being joined. This is extremely inefficient and execution plan analysis showed that there was a very high cost involved in the evaluation of DSP sub-queries twice. For example, in CRM TBI, when TBI VOs for Opportunity and Revenue are used together, they result in this inefficiency and performance bottleneck. In the physical SQLs, we find that all applicable DSPs’ sub-queries for a user get added twice – once on the Opty VO and then again on the Revn VO and then these are joined. This is extremely inefficient as it amounts to running the All Opty and All Revn SQLs combined with aggregation on top. To compound the problem, this query was included as part of the user login landing page, resulting in extremely poor user experience. The resolution to this problem is to create an unsecured version of the TBI VO for the Dimension VO and then ensure that when both the Fact and Dimension participate in the report or analysis, the secured VO for the Fact is used along with the unsecured VO for the Dimension VO. However, when the end user creates a report or analysis only based on the Dimension VO, it should be ensured that only the secured version of the Dimension VO is used. So this solution will need the creation of an unsecured version of the Dimension TBI VO and then RPD modeling needs to ensure that these are used appropriately. For example, in the case of OptyMgmt, the resolution was to use an additional unsecured Opty VO, which would be used when Revn VO is also used in the query, saving the duplication of DSPs. When the user does only dimensional browsing on Opty VO, the secured version of Opty VO would get used.

3.33 Fusion CRM DSPs: Frequent SELECT Calls to HZ_PARTIES which is a Big Table

This was another interesting finding related to DSPs. We found that currently in many Data Security Predicate sub-queries, the usage was in the following form: `<column_name> IN ( SELECT party_id FROM hz_parties WHERE user_guid = FND_Global.USER_GUID). This meant that a join to HZ_PARTIES table happens several times to get the same value. This adds unnecessary complexity to the final SQLs, especially because these are usually placed inside DSPs’ sub-sub-queries, resulting in nested sub-queries. Since HZ_PARTIES table was being accessed to get the mapping of FND_GLOBAL.USER_GUID to PARTY_ID column, this could ideally be replaced with calls to HZ_SESSION_UTIL.GET_USER_PARTYID which caches this value for the user’s session and can be used instead of a SELECT statement using HZ_PARTIES table. Thus the resolution was to replace the `(SELECT party_id FROM hz_parties WHERE user_guid = FND_GLOBAL.USER_GUID)` with `(SELECT HZ_SESSION_UTIL.GET_USER_PARTYID FROM DUAL)`.
3.34 Fusion HCM DSPs: Frequent SELECT Calls to PER_USERS and PER_PESONS Tables

In HCM data security predicate sub-queries, the repeated use of the following query in different forms is a drag on performance: 'SELECT U.PERSON_ID FROM PER_USERS U WHERE U.USER_GUID=FND_GLOBAL.USER_GUID'. This means we are joining to PER_USERS table several times to get the same value. This not only adds unnecessary complexity to the final SQLs, especially because these are usually placed inside data security sub-queries, resulting in nested sub-queries, but also results in unnecessary processing by the database, leading to wasted CPU & I/O. This was similar to the change made for Fusion Apps CRM above. For example, current query: 'SELECT U.PERSON_ID FROM PER_USERS U WHERE U.USER_GUID=FND_GLOBAL.USER_GUID'. The equivalent replacement query: 'SELECT hrc_session_util.get_user_personid from dual'. Similarly, we could use hrc_session_util functions get_userid, get_user_partyid, get_user_personid, get_user_primary_assignmentid, get_user_workterms_assignid and get_enterpriseid as required more efficiently.

3.35 Optimizer Evaluating Too Many Indexes

Our analysis of TBI SQLs over the period of research revealed that TBI SQL queries involved accessing data from a large number of tables in the transactional database due to the nature of creating star-schema model over transactional tables. Another factor was that each of this large number of tables had dozens of indexes created to support the transactional applications and their various modules. During the query evaluation phase, the optimizer evaluates all access paths for each of these tables referenced in the SQL query. Since these tables have a large number of indexes each, the optimizer can take a considerable time in order to evaluate all access paths via the many indexes. In SaaS and Cloud enterprise applications, the OLTP schema can span many applications and thus involve creation of indexes on tables needed by different applications. Evaluation of all available indexes without regard to the application, module or action for which SQL query is generated is wasteful effort. The resolution to this problem is the creation of 'Context-Sensitive Indexes' in the database to limit the number of indexes evaluated by the optimizer
d.

4. Benchmark Results

We conducted our benchmark tests utilizing use cases from Oracle's Fusion CRM OTBI against an 11gR2 Oracle database. Repeated testing recorded significant performance gains when the recommendations highlighted in this paper were implemented. A brief summary of performance improvements is below.

1. Query Response Time (RT) improvements ranging from a low of 5% to a high of 745 times for the queries benchmarked, and in some cases where the queries would hang earlier, by many orders of magnitude.
2. Database hard parse time improvements from 1% to 208 times.
3. Logical I/O or Buffer Gets’ improvement ranging from 43% to 454 times.
4. SQL-shared-memory reduction by up to 52%.

Even though our findings and recommendations were derived during a multi-year ADF-OBIEE-Oracle application development environment and recording of performance improvements from the implementation of our proposed solutions by Oracle's TBI applications, we are confident that these lessons would provide good guidance for developers embarking on architecting any new TBI applications.

5. Conclusion

Capturing of TBI SQLs from various sources and their analysis has been an ongoing process for the past few years. Through an extensive analysis of TBI SQLs’ text, execution plans, RPD and data model, we identified many generic issues, which when resolved resulted in greatly improved performance and scalability of TBI queries.

In this paper, we have presented the findings from our research conducted over multiple development cycles of Oracle's Fusion Applications TBI queries. We have highlighted the limitations in existing technologies or by the unintended consequences of using this state of the art, cutting edge technologies beyond the initial purposes, extending their usage to areas not originally planned. The current TBI architecture and performance cold becomes a barrier for the success of enterprise transactional business applications and their mass adoption. The paper has highlighted many shortcomings and proposed many solutions to
create a synthesis between the query-generation mechanisms of the middleware with the query execution at the backend RDBMS that can result in much improved performance and scalability of TBI applications.

6. References


