



Data Warehousing Tuning Techniques

Technical White Paper

Table of Contents

1	SYNOPSIS / EXECUTIVE SUMMARY	3
2	KEYWORDS / CATEGORY	3
3	PROBLEM STATEMENT / HYPOTHESIS.....	3
4	ANALYSIS OF OBSERVATIONS.....	4
5	PROPOSED SOLUTION	6
5.1	ARCHITECTURAL / DESIGN CONSIDERATIONS.....	7
5.2	ADVANTAGES OF THE PROPOSED APPROACH.....	7
5.3	ADAPTABILITY / PORTABILITY OF THE SOLUTION.....	16
6	CONCLUSIONS.....	16

1 Synopsis / Executive Summary

Environment tuning has been a popular topic of conversation since the beginning of application development: users want faster queries, database administrators want faster backups, and developers want faster processing methods for batch-oriented programs. A huge advantage for efficient tuning is to understand your application requirements and environment. By knowing your application's requirements, you can take the guesswork out of where to place your tuning focus. This simply means addressing the areas that have the highest impact. By knowing your application environment, you have the advantage of knowing the best technique to implement with minimal impact to application architecture.

Data warehousing environments are not an exception to the tuning rule. Data warehousing development projects continue to be on the rise and supporting existing production data warehouses is a major task. Data warehouses need to be tuned just like any OLTP database, however, complexity increases when dealing with the typical large volumes associated with data warehousing databases. Oracle has introduced many significant features that were intended to minimize challenges and increase options when tuning data warehousing environments. With all the tuning techniques available, deciding which to implement can be researched intensively. This paper will help minimize research time by answering the following questions concerning data warehousing environment tuning:

- What areas typically require tuning?
- Which options are best to implement based on minimal complexity with maximum impact?
- How are these tuning techniques implemented?

2 Keywords / Category

- Data warehouse, Tuning

3 Problem Statement / Hypothesis

If you are supporting a production, data warehousing environment is relatively easy to identify the areas that require tuning. Ideally, you should monitor the performance in different areas of the warehouse and compare processing statistics over time. Tuning is an iterative process, but when designing the data warehouse from the beginning it is best to keep the tuning process in mind. Getting involved early in the user requirements phase is the best way to understand the ultimate goal of the warehouse. Understanding the result that fulfills users' needs will help to design the warehouse areas more efficiently. Some of the questions that need to be asked early on from architecture and design perspective is:

- What is the level of data granularity?
- How long will the data reside in the warehouse before it is rolled off?

- What are the expectations for query response time?
- How often does the warehouse need to be loaded with changes (daily, weekly, and monthly) and how much time is available for the data loads?
- Will the warehouse be available for querying while it is being loaded?

These questions will help you design an environment that is flexible to your requirements of data volume, aggregation, and data retention. The issue of “data granularity level” needs to be understood by the design team because incorrect granularity often leads to the biggest challenges when tuning. Capturing data at a more detailed granularity than required will influence data volume and processing expectations. Whether you are tuning a recently implemented production warehouse or one that has been in production for years, the best way to predict success is to understand and implement based on user requirements.

As part of state street project one batch program file loading .ksh has taken 8 hrs to execute in ‘choose’ based optimization environment. So new indexes are created and re organized the queries in the database procedures and resulted in more efficient performance as it is executing within two mins coming down by 7 hrs 58mins.

4 Analysis of Observations

DATA WAREHOUSING TUNING HOTSPOTS

OPERATIONAL SOURCES

The operational sources in a data warehouse environment store all of the data that will be end up in the warehouse. The data might come over exactly as stored in the operational source or it might be used to derive and aggregate data eventually queried by the users. This is where the warehouse process begins. Sources can come from flat files, Oracle databases, and other non-Oracle databases. These sources often come from a mixture of internal company systems that may be well-known as well as external systems, which are not, well understood. The Extract portion of the Extract-Transformation-Load (ETL) process is done from these sources. Usually the tuning biggest concern during this Extraction phase is to ensure that the operational sources are not impacted when the extract occurs.

STAGING

The staging area of a data warehouse environment is usually known as the “work horse”. This is where the majority of the batch processing is performed that changes the operational sources into a format that can be loaded into the target warehouse database. The Extracted operational data is loaded into a staging area and Transformed using Oracle techniques, tools, or vendor ETL packages. The major areas that need to be tuned in the staging area are:

- Loading the data into the staging database
- Transforming the data as quickly as possible during the batch window

Most data warehouses do some daily processing. When first implemented in production, the batch window may be sufficient to support the growing data warehouse, but as subject areas get added and fact and dimension tables grow, the staging process usually expands from its original batch window.

TARGET DATA WAREHOUSE

The target database in the data warehouse environment is the final destination for the data. This is where the user will most likely query and analyze the data. Exceptions to this could be if the enterprise data warehouse has processes for feeding departmental data marts in other databases. This paper approaches data warehouse and data mart targets similarly because they have similar tuning needs. The main areas of concern when tuning the target data warehouse database are:

- Loading data from the staging to the target database
- Query performance for the users
- Scheduled batch reports processing during the batch window

It is very common that once the transformed data has been loaded into the target, a series of scheduled batch reports initiate and are sent to the users to provide their individual view of the data. These reports must be processed efficiently so they are complete before the ad-hoc users start querying the database to analyze the latest data load and its relevance to existing historical data.

5 Proposed Solution

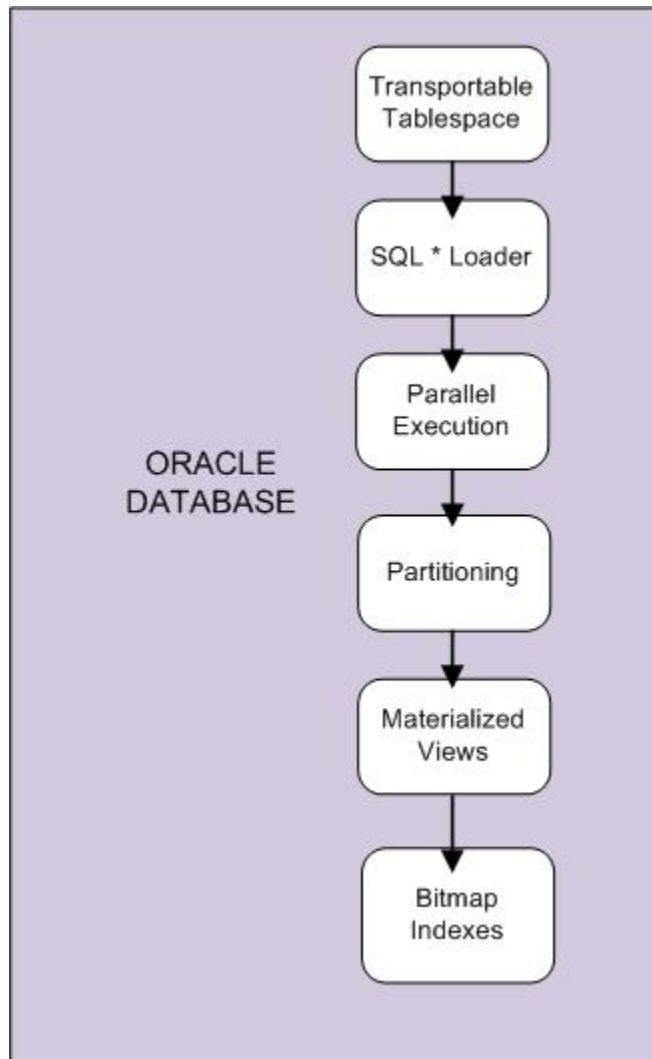
There are many areas that may need to be tuned in a data warehousing environment and Oracle's database features offer different techniques for tuning. This paper focuses on features and techniques that would be common tuning solutions for new and existing data warehouse environments. The topics cover the basic requirements of tuning – designing a solution that can perform in the most efficient manner to complete a task faster, smarter, more easily managed and with minimal resources.

In the ETL phase, the tuning techniques cover the topics of loading data into the staging area and tuning the transformation phase. In the target data warehouse area, the tuning techniques cover the topics of data movement, query, and processing performance. The following tuning techniques were selected because they provide the most significant impact with minimal implementation complexity when planned correctly:

- Transportable Tablespaces – This feature allows data to move between databases quicker than ever before by copying files at the operating system level
- SQL*Loader – This is a very common method for loading data and there are options that can significantly improve its performance
- Parallel Execution – This is one of the best tuning techniques that can be used throughout the entire environment
- Partitioning – This is an essential tuning method for large data warehouse objects
- Materialized Views – This feature can make a big impact on performance with virtually no changes to existing application architecture
- Bitmap Indexes – This indexing strategy allows for more flexible query paths, increases query performance, and is transparent to the reporting queries to implement.

This paper does not go into the details of implementing every technique mentioned. Each technique could be a research paper in itself. The goal is to describe the technique, give an overview of its implementation process, and identify what its tuning benefit is in the data-warehousing environment.

5.1 Architectural / Design Considerations.



5.2 Advantages of the Proposed Approach

Moving large amounts of data on a regular basis is common for data warehousing environments. The following two techniques of Transportable Tablespaces and SQL*Loader Options are easy ways to tune the data movement process for staging and data warehouse databases.

TRANSPORTABLE TABLESPACES

This feature allows data to move between different databases at the operating system level. In a data-warehousing environment, this is very useful when moving data from:

- Operational source databases to a staging database,
- Staging database to data warehouse database,
- Data warehouse database to data mart database, and
- Data warehouse to archive destination.

Transportable tablespaces are moved in “transportable sets” by unplugging the set in the source database and plugging in the set in the target database. The transportable sets must be self-contained, meaning that there must not be any pointers (foreign keys, indexes) from inside the transport set pointing to objects outside the transport set. For partitioned tables, which are commonly used in data warehousing, this means that when partitions are in different tablespaces all tablespaces must be in the transport set. An alternate method for moving partitioned tables is either exchanging the necessary partition into a standalone table or performing a “CREATE TABLE AS SELECT” to move the partitioned data into a stand-alone table.

Moving data using transportable sets is actually very easy to implement. This technique uses a combination of EXPORT/IMPORT and operating system file copy commands. The EXPORT command with the TRANSPORT_TABLESPACE option only extracts the tablespace metadata, which runs in seconds for any size tablespace because it only reads the Oracle Data Dictionary. The tablespace data files are then copied at the operating system level to the target database file location. An IMPORT command with the TRANSPORT_TABLESPACE option inserts the transport set metadata into the Oracle Data Dictionary and the transported data is immediately accessible by the target users. The tablespace sets must stay in READ ONLY mode during the transport process, but can be modified to READ WRITE for updates immediately after transport.

Current limitations of this technique include that both source and target databases must have the same block size, character set, and be on the same hardware platform. The source database where the EXPORT is done must have Oracle Enterprise Edition and the target database cannot have any duplicate tablespace names as a result of the transportable set. This is a very efficient way to move large data sets and can be magnitudes faster than traditional methods of data EXPORT/IMPORT or COPY techniques.

SQL*LOADER OPTIONS

SQL*Loader has been a popular utility to load data from flat files to Oracle databases for many years. It is a straightforward utility, easy to learn and is very dependable. There are several options in SQL*Loader that can significantly improve data loading performance and are equally easy to implement:

- DIRECT
- PARALLEL
- SKIP_INDEX_MAINTENANCE
- UNRECOVERABLE

The basic method for loading data in SQL*Loader is using the CONVENTIONAL path. This is the default option and performs like a typical INSERT statement by updating indexes, firing triggers, and evaluating constraints. Implementing the DIRECT path instead of the CONVENTIONAL path can make significant improvements in performance. The DIRECT path bypasses the SQL processing and writes the data blocks directly to the database files. DIRECT path is a good option for loading staging and data warehouse tables when the loading routine has exclusive access to the object being loaded and the SQL*Loader control file. Limitations of DIRECT option are that no other users can write to the loading table due to its exclusive lock and that no SQL transformations can be made in the control file during the load.

Using the DIRECT option with the PARALLEL option is an efficient tuning technique in data warehousing when loading multiple partitions of the same table concurrently. If the partitions are located on separate disks, the performance impact can reduce the time to that of loading a single partition. This is very useful when loading a data warehouse database that may require multiple days or months to be loaded during the same loading run.

If the CONVENTIONAL path must be used due to transformations being performed during the load, it is a good idea to bypass the index updates by using the SKIP_INDEX_MAINTENANCE option. This will set the indexes to an “unusable” state, but the data load will perform faster if that is the overall goal. The indexes will remain “unusable” until they are rebuilt.

The UNRECOVERABLE option in the control file will bypass any redo log writes during a CONVENTIONAL load. Since recoverability is not an issue when the data load file exists, this is an easy way to increase performance.

Finally, disabling constraints is also a good technique when tuning the SQL*Loader utility. The DIRECT option will automatically disable CHECK and foreign key REFERENCES constraints, but disabling PRIMARY KEY, UNIQUE KEY, and NOT NULL constraints will help the performance of when loading data into a target database. If the data is cleansed in the staging process, there should not be an issue when loading into the data warehouse database.

SQL*Loader is a data loading utility that will be used in many data warehousing environments. Using these tuning options are quick ways to significant performance increases.

TUNING TECHNIQUES FOR QUERY AND PROCESSING PERFORMANCE

Query tuning in a data warehouse is the most visible tuning area to the user community. It is obvious to the users when query response time is slow. Nevertheless, what about transformation and processing time to load the data warehouse from the staging database. These are equally important in order to get the data to the users on time. Query and overall processing tuning considerations are the focus of the following effective techniques: Parallel Execution, Partitioning, Bitmap Indexes, and Materialized Views.

PARALLEL EXECUTION

The Parallel Execution feature allows Oracle to divide operations across processors in a multi-CPU environment. Previously referred to as Parallel Query or Parallelism, this was often confused with Parallel Server, which is dividing users across instances that access the same database.

When a session accesses the database, it uses a sequential server process to fulfill the request. By spreading that single request across multiple processors, the request can be complete that much faster. Parallel execution can be implemented at the SQL statement, database object, or instance level for many SQL operations. As stated earlier, the better you know your requirements and tuning hotspots, the more accurately you can identify the level at which this technique should be implemented.

The degree of parallelism is the number of processors that can get associated with a single SQL action. This degree of parallelism should be identified based on the number of processors and disk drives on the server. The minimum degree would be the number of processors, but an iterative tuning process is usually required to define the optimal degree for an object or SQL statement.

Parallelism is defined at the SQL statement level using statement hints. The following examples demonstrate the ease of implementing at the statement level for tables and indexes:

```
SELECT /*+ PARALLEL(dwh_order_fact, 4) */ ...;
SELECT /*+ PARALLEL_INDEX(dwh_order_fact, dw_order_fact_ix1, 4) */ ...;
SELECT /*+ NOPARALLEL(dwh_order_fact) */ ...;
```

The degree of parallelism can also be defined at the table and index level at object creation or altered afterwards. This example shows changing a table's degree of parallelism to 4 for all eligible SQL statements on this table:

```
ALTER TABLE dwh_order_fact PARALLEL 4;
```

At the instance level, the parallel parameters are set in the init.ora file. The following parameters can be set to control the parallel execution for all eligible statements:

- `parallel_adaptive_multi_user` – allows the degree of parallelism to change based on the number of concurrent users
- `parallel_automatic_tuning` – allows Oracle to manage the parameters for parallel execution
- `parallel_broadcast_enabled` – allows performance improvements for hash and merge joins
- `parallel_execution_message_size` – specifies the size of messages associated with parallel execution
- `parallel_max_servers` – specifies the maximum number of Oracle parallel server processes for an instance
- `parallel_min_percent` – identifies the minimum percent of threads required to ensure parallel execution
- `parallel_min_servers` – specifies the number of parallel server processes created at instance startup
- `parallel_threads_per_cpu` – specifies the default degree of parallelism for the instance by identifying the number of processes a single CPU can handle

Since parallel execution is so versatile it is a very powerful tuning technique to use throughout your data warehousing environment. From data loading to user querying to warehouse maintenance, this technique can be extremely effective when used with the correct hardware and server configuration.

PARTITIONING

Partitioning tables and indexes is an excellent way to increase performance when accessing large amounts of data. Although partitioning is usually discussed when designing a data warehousing environment, it can be implemented effectively with an existing production environment. In Oracle8, physical range partitioning was introduced, and Oracle8i introduced the options of hash and composite range/hash partitioning. The partitioning concept is very simple – by physically organizing large data and index sets into smaller components in the way it will eventually be accessed, it requires less resources to query the data. Partitioned objects can reside in their own tablespaces with data files on separate disks, which minimizes operating system contention when accessing multiple partitions concurrently. Since data warehouses are typically organized by time (sales by day, month, quarter, year), partitioning helps keep the data organized by time for querying and database management tasks.

Creating partitioned tables is as simple as:

```
CREATE TABLE dwh_order_fact (time_key DATE,  
    product_key NUMBER,  
    customer_key NUMBER,
```

```

shipment_key NUMBER,
order_count NUMBER)
PARTITION BY RANGE (time_key)
(PARTITION      order_jan2001      VALUES      LESS      THAN
(TO_DATE('01/31/2001','MM/DD/YYYY'))
TABLESPACE order_jan2001_ts,
PARTITION      order_feb2001      VALUES      LESS      THAN
(TO_DATE('02/28/2001','MM/DD/YYYY'))
TABLESPACE order_feb2001_ts);

```

Adding partitions for new data and moving non-partitioned tables into a partitioned table are performed using an ALTER statement:

```

ALTER TABLE dwh_order_fact ADD
PARTITION      order_mar2001      VALUES      LESS      THAN
(TO_DATE('03/31/2001','MM/DD/YYYY'))
TABLESPACE order_mar2001_ts;
ALTER TABLE dwh_order_fact EXCHANGE
PARTITION order_mar2001 WITH TABLE loaded_order_mar2001;

```

In addition to increasing performance, partitioning also helps tune the administration tasks of backing up new data and rolling off historical data. Since Fact data loaded to the data warehouse database is usually not updated frequently, only the recently loaded partitions need to be backed-up. Exports can extract data by partition, and when partitions are in separate tablespaces, the data files can be backed-up individually by a “hot backup”. For archiving historical data, older partitions can be simply dropped:

```
ALTER TABLE dwh_order_fact DROP PARTITION order_jan1998;
```

This is a significant performance increase over backing up all historical data after each data load.

When implementing partitioned indexes, a combination of local and global indexes is usually required. Local indexes are bitmap or B*Tree indexes on a single data partition. Global indexes are B*Tree indexes across all data partitions for a table. For management purposes and query performance, local indexes are more efficient for any database environment. The following command will create local indexes in both the Jan and Feb partitions.

```
CREATE INDEX dw_order_fact_ix1 ON dwh_order_fact(time_key)
LOCAL (PARTITION order_ix_jan2001 TABLESPACE order_jan2001_ix,
PARTITION order_ix_feb2001 TABLESPACE order_feb2001_ix);
```

Before Oracle8, data partitioning was a consistent struggle to implement efficiently for performance tuning. Various techniques were used, but none was more straightforward or simpler to implement as recent releases of Oracle.

BITMAP INDEXES

Implementing the correct indexing strategy in a data warehouse is critical for expected query response. Data warehouses are commonly designed as star schemas. This design is of a large Fact table surrounded by smaller denormalized Dimension tables benefits query performance. Oracle introduced bitmapped indexes, which changed the way that star query processing could work in future Oracle releases.

Before bitmapped indexes Oracle used composite B*Tree indexes to process a star query. Since the uniqueness of a Fact table is comprised of individual foreign keys to the associated Dimension tables, the composite B*Tree index on the Fact table causes the star query to process by joining all the Dimension tables in a Cartesian product based on the WHERE clause, then join back to the Fact table based on the composite B*Tree index. This worked well when the query requirements always included the first part of the composite key and the volume of a Dimension Cartesian product was relatively low. As volumes increased and user requirements evolved, the bitmap index was used in conjunction with the star transformation query method in Oracle8i to transform, or rewrite, the query to a series of subqueries.

Bitmap indexes are implemented using a bitmap instead of the column value in the index. For low cardinality relationships, common between Fact and Dimension keys, this efficiently stores the index data in significantly less space than required by a B*Tree index. Less space leads to more records per read, which leads to increases in row retrieval performance. By setting the STAR_TRANSFORMATION_ENABLED parameter and creating single column bitmapped indexes on the Fact table foreign key columns, the query transforms to access the Fact table first, followed by the Dimension table joins. This is more efficient because the result set is reduced at the Fact table level instead of performing a Cartesian product of all possible Dimension attributes.

Implementing this index tuning strategy in an existing data warehouse is simple:

- Set the init.ora parameter STAR_TRANSFORMATION_ENABLED = TRUE
- Create single column bitmap indexes on all of the foreign key columns in the fact table.

```
CREATE BITMAP INDEX dw_order_fact_bm1 ON dwh_order_fact(time_key)
LOCAL
(PARTITION order_bm1_jan2001 TABLESPACE order_jan2001_ix,
PARTITION order_bm1_feb2001 TABLESPACE order_feb2001_ix,
PARTITION order_bm1_mar2001 TABLESPACE order_mar2001_ix);
```

```
CREATE BITMAP INDEX dw_order_fact_bm2 ON dwh_order_fact(product_key)
LOCAL
(PARTITION order_bm2_jan2001 TABLESPACE order_jan2001_ix,
PARTITION order_bm2_feb2001 TABLESPACE order_feb2001_ix,
PARTITION order_bm2_mar2001 TABLESPACE order_mar2001_ix);
```

```
CREATE BITMAP INDEX dw_order_fact_bm3 ON dwh_order_fact(customer_key)
LOCAL
(PARTITION order_bm3_jan2001 TABLESPACE order_jan2001_ix,
PARTITION order_bm3_feb2001 TABLESPACE order_feb2001_ix,
PARTITION order_bm3_mar2001 TABLESPACE order_mar2001_ix);
```

```
CREATE BITMAP INDEX dw_order_fact_bm4 ON dwh_order_fact(shipment_key)
LOCAL
(PARTITION order_bm4_jan2001 TABLESPACE order_jan2001_ix,
PARTITION order_bm4_feb2001 TABLESPACE order_feb2001_ix,
PARTITION order_bm4_mar2001 TABLESPACE order_mar2001_ix);
```

The bitmap index tuning technique can be used on new data warehouses or can replace pre-Oracle8i technique for existing data warehouses. When the query path characteristics are unpredictable and query performance is suffering, this technique can quickly increase performance without any changes to the actual query code.

MATERIALIZED VIEWS

The Materialized View feature can be a quick and painless tuning technique to increase the performance of common problem queries. A materialized view is a physically stored summary table that works like a view in the sense that the cost-based optimizer can rewrite other queries to use the summary. This is very helpful with an iterative tuning strategy because the batch reports and user queries do not have to know the materialized view even exists in order to benefit from it. In fact, it's probably best if users and developers do not write queries against the materialized views because if their need in the tuning processes becomes obsolete, then the reporting code does not have to change.

Materialized views are implemented using the new `CREATE MATERIALIZED VIEW` statement, but DBAs will recognize that the underlying structure is that of an intelligent snapshot. The materialized view also has similar refresh capabilities as snapshots by using the `COMPLETE` or `FAST` options.

The following example creates a summary of customer orders by month:

```
CREATE MATERIALIZED VIEW dwh_customer_order_by_month
TABLESPACE summary_ts
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
ENABLE QUERY REWRITE
AS SELECT t.month , t.year, c.name , SUM(o.order_count) sum_orders
FROM dwh_time_dim t, dwh_customer_dim c, dwh_order_fact o
WHERE t.time_key = o.time_key
AND c.customer_key = o.customer_key
GROUP BY t.month , t.year, c.name;
```

Queries that qualify to use the `DWH_CUSTOMER_ORDER_BY_MONTH` materialized view to select monthly or yearly data for a customer will access that physical structure instead of the underlying tables of `DWH_TIME_DIM`, `DW_CUSTOMER_DIM`, and `DWH_ORDER_FACT`. This is where this tuning technique becomes extremely useful because the reporting code does not have to be modified to use the new summary.

The `QUERY_REWRITE_ENABLED` parameter needs be set to `TRUE` at the instance or session level and the `QUERY REWRITE` privilege should be granted for the cost-based optimizer to rewrite incoming SQL statements. When referencing tables outside your schema in a materialized view, the `GLOBAL QUERY REWRITE` privilege must also be granted. In order to create materialized views with the `REWRITE` clause, the `QUERY REWRITE` privilege must be directly granted to the object owner:

```
GRANT QUERY REWRITE TO dwh_schema;
```

If you are not sure which materialized views to create in your environment, the Summary Advisor feature of Oracle can recommend materialized views to create based on defined relationships in the new DIMENSION object. In the CREATE DIMENSION statement you can specify the relationships of attributes within a single or set of physical Dimension tables. The DIMENSION object is contained in the Oracle Data Dictionary and does not have a physical structure like the

MATERIALIZED VIEW. The packaged procedure DBMS_OLAP.RECOMMEND_MV run against a fact table will take the dimension and integrity constraint information into consideration before creating recommended materialized views. The DEMO_SUMADV.PRETTYPRINT_RECOMMENDATIONS packaged procedure or MVIEW\$RECOMMENDATIONS view will return the recommendations from the advisor. These recommendations will also include estimates on storage and percent performance gains. Your actual results will most likely vary.

This tuning technique is encouraged because of its capacity to increase query performance and it is transparent to the users and developers. Once the materialized view is created and parameters are set, the tuning is immediately realized.

5.3 Adaptability / Portability of the Solution

The given solution will help you architect an environment that is flexible to your requirements of data volume, aggregation, and data retention.

6 Conclusions

Tuning a data warehouse is necessary, but effective tuning techniques do not have to be research intensive and painful to implement. The methods described in this paper are proven techniques that have been utilized in new and existing data warehouses. They are by far “the best” to implement based on their high tuning benefit and low overhead