Dimensions Based Data Clustering and Zone Maps

Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Dmitry Potapov, Janaki Lahorani, Murali Krishna

Oracle Corporation
500 Oracle Parkway, Redwood Shores, CA 94065

{Mohamed.Ziauddin, Andrew.Witkowski, You.Jung.Kim, Dmitry.Potapov, Janaki.Narasinghanallur, Murali.x.Krishna} @oracle.com

ABSTRACT

In recent years, the data warehouse industry has witnessed decreased use of indexing but increased use of compression and clustering of data facilitating efficient data access and data pruning in the query processing area. A classic example of data pruning is the partition pruning, which is used when table data is range or list partitioned. But lately, techniques have been developed to prune data at a lower granularity than a table partition or sub-partition. A good example is the use of data pruning structure called zone map. A zone map prunes zones of data from a table on which it is defined. Data pruning via zone map is very effective when the table data is clustered by the filtering columns.

The database industry has offered support to cluster data in tables by its local columns, and to define zone maps on clustering columns of such tables. This has helped improve the performance of queries that contain filter predicates on local columns. However, queries in data warehouses are typically based on star/snowflake schema with filter predicates usually on columns of the dimension tables joined to a fact table. Given this, the performance of data warehouse queries can be significantly improved if the fact table data is clustered by columns of dimension tables together with zone maps that maintain min/max value ranges of these clustering columns over zones of fact table data. In recognition of this opportunity of significantly improving the performance of data warehouse queries, Oracle 12c release 1 has introduced the support for dimension based clustering of fact tables together with data pruning of the fact tables via dimension based zone maps.

1. INTRODUCTION

Data warehouses (DWs) typically contain large fact tables connected to various dimension tables in a star or snow flake formation [1]. Naturally, a typical DW query joins a fact table to one or more dimension tables in the form of a star or snowflake with filter predicates usually specified on one or more dimension table columns. Since there are no direct joins between different dimension tables (apart from joins between tables of normalized dimensions in the case of snowflake schema) but only to the fact table, one straightforward strategy would be to join the fact table to a dimension table and then join to the remaining dimension tables. Since the fact table is typically orders of magnitude larger than dimension tables, the size of intermediate results produced by this strategy will usually be large making the subsequent joins quite expensive. An alternate strategy would be to postpone joining the fact table to the end of join order but this leads to the formation of Cartesian products between different dimension tables, again making the intermediate join results usually large.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

1.1 Star Transformation Technique

The database industry has historically used star transformation technique using semi-joins [2] [4] [5] for star or snowflake queries to avoid producing large intermediate results yet joining the fact table toward the end of join order. This technique is used when B-tree or bitmap indexes exist on foreign key columns of the fact table. Since the index size is much smaller compared to the fact table size each generated semi-join subquery that joins a foreign key index to a dimension table (or tables within a normalized dimension) that has filter predicates tends to be efficient to execute. The result of a semi-join subquery is a list or bitmap of qualified fact table rowids. If multiple semi-join subqueries are used the resulting lists or bitmaps are intersected to produce a final list or bitmap, which is then used as a filter when fetching rows from the fact table. The qualified fact table rows are then joined back to appropriate dimension tables to fetch non-key columns referenced by the query. Star transformation strategy is quite efficient when the semi-joins are able to filter out a large majority of the fact table rows thereby making the subsequent join backs to some of the dimension tables relatively inexpensive.

1.2 Right-Deep Joins with Bloom Filters

A Bloom filter [3] built along with hash table by the left (i.e. build) side of a hash join serves as a semi-join filter for the table scanned by the right (i.e. probe) side. Similarly, a Bloom filter built by the left side of a sort-merge join serves as a semi-join filter for the table on the right side. A right-deep joins query tree [6] processes a series of left sides before the table on right side is fetched. Therefore, an optimized plan for a star query could be a series of right-deep hash or sort-merge joins with a dimension table on each left side and the fact table on the right side. When a left side involves a dimension table with filter predicates it creates a Bloom filter on the join key. Thus Bloom filters built by one or more left sides are then used to filter out unqualified fact table rows fetched by the right side thereby boosting the performance of joins. Similar to a transformed star/snowflake query with semi-joins, a right-deep joins query tree accomplishes the goal of joining the fact table at end while producing intermediate results of manageable size.

1.3 Data Clustering and Data Pruning

Recent trend in DWs has been away from indexing and more toward better compression (e.g. columnar format) [20] and clustering of data. Data clustering when properly utilized can lead to significant improvement in the query performance because of the avoidance of unnecessary data access from the disk, and when the data is already cached in memory by avoiding the processing of unnecessary rows or groups of rows. Data clustering can also aid in better compression.

The technique of recognizing certain chunk of data as unnecessary for query processing and thus skipping over such data is called data pruning. A data chunk could be a table partition, or a set of contiguous data blocks or rows, so data pruning can operate at different granularity levels.

Partition pruning is a classic data pruning technique employed when the table data is horizontally partitioned by the ranges or list of partitioning key values [7] [8]. The range or list partitioning of tables represents strict clustering of data because given a partitioning key value the associated table rows can only be located in a designated partition. In contrast, table data could be naturally clustered by some columns such as order_create_date and order_ship_date of a table storing customer orders that are created in chronological order. Obviously, this form of data clustering is not strict but incidental. Thus data in a table can be strictly clustered using a data partitioning scheme, and within each partition data can be either implicitly clustered by columns that are correlated to the partitioning key (e.g. range partitioning by order_create_date and implicit clustering by order_ship_date) or explicitly clustered by some columns (e.g. range partitioning by order_create_date and explicit clustering by order_status). Explicit clustering of table data is cost effective when it is performed during bulk operations such as bulk data loads and bulk data moves.

In the context of the star transformation technique or right-deep joins query tree, data clustering of fact table data can be exploited by applying data pruning in addition to regular row filtering via bitmaps or Bloom filters [3]. For example, a fact table could contain orders that are created and inserted with their order_create_date and order_ship_date values appearing in a rough chronological order. If the fact table is partitioned by ranges of order_create_date, classic partition pruning technique is used when a query contains a filter predicate on the partition key column. What if the query has a filter predicate on non partition key column such as order_ship_date that is correlated to the partition key column? Classic partition pruning is not possible in this situation unless an auxiliary access structure known as zone map [11] or storage index [12] exists for the fact table.

1.4 Zone Maps

A zone map is like a coarse index that maintains min/max value ranges of one or more specified columns over contiguous sets of data blocks or rows called zones of a table. Thus a zone map is a table that contains rows made up of min/max value ranges of specified columns per zone of table data. A zone is an arbitrarily defined unit of data, which can be equal to a table partition but usually it is much smaller. With zones smaller than table partitions, a zone map can still maintain min/max value ranges per zone and per partition of the table data. Now, if a zone map is maintaining min/max values ranges for order_ship_date column of a fact table partitioned by value ranges of order_create_date and a query contains a range predicate on order_ship_date then partition pruning can be performed using the zone map. Specifically, a table partition can be pruned if the order_ship_date value range specified in the query filter predicate doesn't intersect with the min/max value range of order_ship_date present in the zone map for that partition. Similarly, data pruning can be performed at a lower granularity level of zones of table data.

A storage index is like a zone map on a single table. It is an access structure that is built and maintained within the storage sub-system separated from the RDBMS. A good example is the storage sub-system consisting of storage cells in the Oracle Exadata Database Machine [9]. Storage cells are active components that can perform simpler query tasks such as expression evaluation, and filtering and pruning of table rows. For example, storage cells can filter out unqualified table rows by evaluating a Bloom filter pushed to the storage sub-system. Like a zone map a storage index maintains min/max value ranges for a set of table columns over contiguous chunks of the table data called regions. Therefore, storage cells can perform data pruning with the help of a storage index using appropriate pushed-in filter predicates.

The left side of a hash or sort-merge join can easily compute min/max value range of the join key in addition to the Bloom filter. Using the join key min/max value range RDBMS (storage cells) operating on the right side table can perform data pruning provided a zone map (storage index) exists on the join key. This is join key min/max range based pruning or simply join pruning. It is effective when the right side table data is clustered around the join key, and the join key is correlated with the filtering column.

Several commercial database vendors offer data pruning techniques using zone maps and storage indexes. IBM Netezza Data Warehouse Appliance [10] offers zone map based data pruning. Amazon Redshift allows for sorting data upon loads and creating min/max zone maps [21]. Oracle Exadata Database Machine system offers storage index based data pruning by the storage cells [12]. The storage indexes are created and maintained automatically by the storage cells on a set of table columns determined by the monitoring of pushed-in predicates. In Oracle RDBMS 12c Release 2, table data can be cached in Oracle Database In-Memory columnar store [13]. The column data pertaining to contiguous sets of table rows is stored compressed in columnar format in memory units called In-Memory Compression Units (IMCUs) [14]. Thus each column is stored separately in a series of IMCUs. Each IMCU maintains min/max value range of the column data it stores, thus it can be pruned if its range doesn't intersect with the value range represented by a query filter predicate.

DW queries usually contain filter predicates on dimension table columns thus a significant data pruning opportunity is missed when the fact table data is clustered by its local columns and not by the dimension table columns. So we argue and show in this paper that it is quite significant to cluster the fact table data by columns of one or more dimension tables, or by a combination of fact and dimension columns. Realizing this significant data pruning benefit, Oracle 12c release 1 is the first commercial database product to offer dimensions based data clustering of tables and data pruning via dimensions based zone maps.

Rest of the paper is organized as follows. We first introduce dimensions based data clustering in section 2, and dimensions based zone map in section 3. We explain different types of data pruning methods employed using a zone map in section 4. In section 5 we describe the experiments we performed using the Star Schema Benchmark (SSB) [15] to measure the benefit of dimensions based data pruning on query performance. We conclude this paper in section 6 with a summary plus a discussion of possible future enhancements.

2. DIMENSIONS BASED DATA CLUSTERING

A majority of DW queries contain filter predicates on dimension table columns, and a few contain filter predicates on fact and dimension table columns. For this reason, query performance of DW queries can be improved significantly if fact table data can be clustered by dimension columns or when necessary a combination of dimension and fact columns, together with a zone map that maintains min/max values ranges of clustered columns on zones of fact table data.

2.1 Dimensions Based Clustering of Fact Data

Table data can be explicitly clustered by a combination of columns of other related tables. For example, in data warehouses that typically have star schema relationships, fact table data can be clustered by the columns of one or more dimension tables. This can be accomplished by joining the fact table to the dimension tables and ordering the fact table data by the specified dimension columns. In fact, multi-dimensional clustering such as the ordering produced by Z-curve or Hilbert curve fitting [19] can be used to cluster fact table data by columns of multiple dimensions. When the fact table is partitioned, which is usually the case in data warehouses, multi-dimensional clustering applies to data within each fact table partition.

Dimensions usually contain data hierarchies such as (c_region, c_nation, c_city) in the customer dimension and (p_mfgr, p_category, p_brand1) in the part dimension of the Star Schema Benchmark (SSB) [15]. The fact table data can be explicitly clustered in linear manner along the columns of each dimension hierarchy and then in interleaved (i.e. multi-dimensional) manner along hierarchies themselves. For example, we can cluster the lineorder table of SSB in linear manner along customer dimension so that rows belonging to same c_region are grouped together, and within each c_region rows belonging to same c_nation are grouped together, and within each c_nation rows belonging to same c_city are grouped together. Similar linear grouping of lineorder rows can be applied along the part dimension hierarchy. Finally, lineorder rows can be grouped in interleaved manner along customer and part hierarchies. This type of clustering based using a combination of linear and interleaved grouping can be notationally described as INTERLEAVED(LINEAR(c_region, c_nation, c_city), LINEAR(p_mfgr, p_category, p_brand1)).

In Oracle RDBMS release 12c, table data can be cached in the In-Memory Columnar store in compressed columnar format. The data for a contiguous set of table rows is stored separately in memory chunks called In-Memory Compression Units (IMCUs) [14]. Thus the data for each column is stored separately in a series of IMCUs. It is important to note that data cached in IMCUs is in the same order as the order of rows stored on disk, which means any clustering of the table data is also preserved in IMCUs. Therefore, incidental or explicit clustering of table data in IMCUs can be exploited to identify and prune unnecessary IMCUs.

2.2 Data Clustering Operations

We will use the Star Schema Benchmark (SSB) and the published Oracle syntax for table clustering to illustrate the operations performed to define explicit clustering on tables, two forms of data clustering, and conditions under which to perform automatic clustering of table data. Further, we will assume that the lineorder table of SSB is partitioned by yearly ranges of lo_orderdate column. The simplest form of clustering of a table is by its local columns. The following statement defines clustering of lineorder table by its lo_quantity and lo_discount column.

```
CREATE TABLE lineorder ( ... )
  CLUSTERING BY ORDER (lo_quantity, lo_discount);
```

Explicit clustering can be defined for an existing table but its data is not immediately clustered

```
ALTER TABLE lineorder
  ADD CLUSTERING
    BY ORDER (lo_quantity, lo_discount);
```

In Oracle, the clustering of data is performed upon bulk load and bulk data move operations. The latter are typically partition maintenance operations, such as move partition. In fact, the clustering definitions listed above have these options by default. The following statement modifies clustering to turn off clustering of data upon bulk loads.

```
ALTER TABLE lineorder
  MODIFY CLUSTERING
    BY ORDER (lo_quantity, lo_discount)
    NO ON LOAD YES ON DATA MOVEMENT;
```

After defining clustering for an existing table, its data can be clustered by performing bulk data move operations as follows.

```
ALTER TABLE lineorder
  MOVE PARTITION lineorder_part1;
ALTER TABLE lineorder
  MOVE PARTITION lineorder_part2;
. . . .
```

The partition move causes the data at the original location to be read and ordered according to the clustering definition before it is moved to its new location.

If YES ON LOAD option is specified or it is simply omitted then upon each bulk load operation the source data is clustered before it is bulk loaded into the table. The following statement bulk loads data from lo_source_data into lineorder table with defined clustering. In Oracle RDBMS, the hint /*+ APPEND */ indicates to perform bulk insert (i.e. bulk load) instead of normal conventional insert operation.

```
INSERT /*+ APPEND */ INTO lineorder
  SELECT * FROM lo_source_data;
COMMIT;
```

Two forms of data clustering are supported. One is linear and the other is interleaved. Linear clustering means order table rows by specified columns in major to minor column order, similar to the ordering of table rows in a multi-column index. Interleaved clustering means order table rows by specified columns in multi-dimensional manner such as the ordering produced by well known Z-curve or Hilbert curve fitting [19]. Since linear is the implicit default, so interleaved needs to be explicitly specified as illustrated below.

```
ALTER TABLE lineorder
  MODIFY CLUSTERING
  BY INTERLEAVED ORDER (lo_quantity, lo_discount);
```

The statement above modifies the clustering definition but does not re-cluster the table data until it is bulk moved. Interleaved clustering orders the table rows equally well by specified columns so data pruning using filter predicate on one, or other, or both columns will be effective. Oracle RDBMS uses Z-ordering technique for interleaved clustering.

All examples above deal with clustering of a table by its local columns. Clustering of data especially of fact tables can be made dimensions based. In this case, fact table needs to be joined to the dimension tables so its rows can be ordered by dimension columns. For this, the dimension join keys need to be unique. The following statement modifies clustering of lineorder table to be dimensions based.

```
ALTER TABLE lineorder
  MODIFY CLUSTERING
    lineorder LEFT JOIN customer
                ON (lo_custkey = c_custkey)
              LEFT JOIN part
                ON (lo_partkey = p_partkey)
  BY INTERLEAVED ORDER (c_region, p_mfgr);
```

The left outer joins specified in the clustering definition preserve all lineorder rows in the joined result. The joined result is then ordered interleaved by c_region and p_mfgr columns.

In SSB, customer and part dimensions contain (c_region, c_nation, c_city) and (p_mfgr, p_category, p_brand1) data hierarchies respectively. This means, the lineorder table data can be ordered interleaved by these hierarchies. This maximizes the data pruning opportunity since queries will likely have filter predicates on any columns of these hierarchies in different combinations. Dimension hierarchies based clustering can be specified as shown below.

```
ALTER TABLE lineorder
  MODIFY CLUSTERING
    lineorder LEFT JOIN customer
                ON (lo_custkey = c_custkey)
              LEFT JOIN part
                ON (lo_partkey = p_partkey)
    BY INTERLEAVED ORDER
      ( (c_region, c_nation, c_city),
        (p_mfgr, p_category, p_brand1) );
```

Here, lineorder table rows are ordered linearly along the columns of two dimension hierarchies, and then ordered interleaved by these two hierarchies. As stated before, effective data pruning occurs when queries contain filter predicates on any combination of these hierarchy columns, such as (p_category = 'MFGR#12' AND c_region = 'ASIA').

3. DIMENSIONS BASED ZONE MAPS

In Oracle, you can create either a basic or join zone map on a table. The table on which a zone map is defined is called fact table. There are no joins in a basic zone map definition so it has only the fact table. A join zone map is a dimension based zone map. It contains a fact table plus one or more dimension tables joined to the fact table in a star formation. A basic zone map maintains min/max value ranges on columns of the fact table, whereas a join zone map usually maintains min/max value ranges on columns of the dimension tables although it can also maintain min/max value ranges on columns of the fact table. Oracle RDBMS allows only one zone map per fact table, which is adequate for the purpose of data pruning of fact table. However, a table can appear as a dimension in multiple zone maps. This allows for defining zone maps on different fact tables that share same dimensions.

Basic or join zone map is nothing but materialized aggregation of fact table data, wherein minimum and maximum aggregates of certain columns are formed on well defined chunks of the fact table data. Further, join zone map also represents materialization of joins between the fact table and dimension tables. In other words, zone map is a materialized view.

Min/max aggregates are difficult to refresh incrementally using the change delta when deletes or updates to min/max columns occur. Thus classic incremental refresh method using the change data logs of base tables cannot be used. In section 3.2, we explain how zone maps are maintained and refreshed in efficient manner without the use of change data logs.

3.1 Zone Map Operations

The following statement creates a basic zone map that maintains min/max value ranges of lo_quantity and lo_discount columns per zone of lineorder table.

```
CREATE MATERIALIZED ZONEMAP lineorder_zmap AS
  SELECT SYS_OP_ZONE_ID(rowid),
         MIN(lo_quantity), MAX(lo_quantity),
         MIN(lo_discount), MAX(lo_discount)
  FROM lineorder
  GROUP BY SYS_OP_ZONE_ID(rowid);
```

The function SYS_OP_ZONE_ID() maps the lineorder table rows stored in sets of contiguous data blocks on disk into different zones. Given the rowid (i.e tuple id) of a fact table row this function computes a unique zone identifier. This function takes an optional second argument representing zone map scale that determines the size of each zone. The default value of zone map scale is for each zone to encompass up to 1024 contiguous data blocks. The lineorder_zmap will have as many rows as the number of zones of lineorder data. If lineorder table is partitioned, say, with 10 partitions then lineorder_zmap will have additional 10 rows each storing min/max value ranges of lo_quantity and lo_discount for a partition.

If the data in lineorder table is clustered by the customer and part dimension hierarchies then a join zone map should be created. The following statement defines a join zonemap on lineorder table. Since only one zone map can exist on a table we first drop the previously created one before creating a new one.

```
CREATE MATERIALIZED ZONEMAP lineorder_zmap AS
SELECT SYS_OP_ZONE_ID(lineorder.rowid),
       MIN(c_region), MAX(c_region),
       MIN(c_nation), MAX(c_nation),
       MIN(c_city), MAX(c_city),
       MIN(p_mfgr), MAX(p_mfgr),
       MIN(p_category), MAX(p_category),
       MIN(p_brand1), MAX(p_brand1)
FROM lineorder LEFT JOIN customer
                 ON (lo_custkey = c_custkey)
               LEFT JOIN part
                 ON (lo_partkey = p_partkey)
GROUP BY SYS_OP_ZONE_ID(lineorder.rowid);
```

A DESCRIBE command for above join zone map shows the following columns present in lineorder_zmap.

```
DESCRIBE lineorder_zmap;

Name                      Null?     Type
------------------------  --------  --------------
ZONE_ID$                  NOT NULL  NUMBER
MIN(C_REGION)                       VARCHAR2(20)
MAX(C_REGION)                       VARCHAR2(20)
MIN(C_NATION)                       VARCHAR2(20)
MAX(C_NATION)                       VARCHAR2(20)
MIN(C_CITY)                         VARCHAR2(20)
MAX(C_CITY)                         VARCHAR2(20)
MIN(P_MFGR)                         VARCHAR2(20)
MAX(P_MFGR)                         VARCHAR2(20)
MIN(P_CATEGORY)                     VARCHAR2(20)
MAX(P_CATEGORY)                     VARCHAR2(20)
MIN(P_BRAND1)                       VARCHAR2(20)
MAX(P_BRAND1)                       VARCHAR2(20)
ZONE_LEVEL$                         NUMBER
ZONE_STATE$                         NUMBER
ZONE_ROWS$                          NUMBER
ZONE_AJ_ROWS$1_CUSTOMER             NUMBER
ZONE_AJ_ROWS$2_PART                 NUMBER
```

Column ZONE_ID$ stores the zone identifiers computed by SYS_OP_ZONE_ID(). This is followed by pairs of columns created to store min/max value ranges as specified in the SELECT list. In Oracle, additional zone map columns are transparently created to store other zone related information. ZONE_LEVEL$ indicates the granularity (i.e. zone or partition) of min/max value ranges. ZONE_STATE$ is used to store the staleness state of min/max value ranges pertaining to a zone or partition. ZONE_ROWS$ stores the number of fact table rows in that zone or partition. Columns ZONE_AJ_ROWS$1_CUSTOMER and ZONE_AJ_ROWS$2_PART capture the referential integrity state of the set of fact rows belonging to a zone (partition) with respect to dimension tables customer and part respectively. Specifically, these columns store the count of orphan fact table rows within a zone (partition) not joining with any row in the respective dimension table. This information is used for the purpose of staleness tracking of a join zone map upon DML operations to respective dimension tables, which is described in section 3.2.2.2.

The state and properties applicable to entire zone map are stored in the data dictionary (aka database catalog). The properties include zone map type (i.e. basic or join), zone map scale, refresh mode (i.e. when to refresh - upon bulk data load, upon bulk data move, upon DML transaction commit, or on demand). The zone map states include zone map being invalid, stale, and disabled (i.e. zone map use disabled by the user). The following command disables the use of zone map.

```
ALTER MATERIALIZED ZONEMAP lineorder_zmap DISABLE
PRUNING;
```

3.2 Zone Map Maintenance

Zone map maintenance consists of tracking the state of the zone map itself as well as the state of min/max value ranges stored per zone (and per partition) upon changes to the fact table and changes to the dimension tables in the case of a join zone map. A second aspect of zone map maintenance consists of refresh operations using full and incremental refresh methods.

3.2.1 Validity Checking

Certain DDL operations on tables on which a zone map depends necessitate a re-validation of the zone map definition and marking its validity state accordingly. For example, dropping a table or a column on which a zone map depends makes its definition invalid. Changing column data type on which a zone map depends keeps the zone map definition valid but the min/max value ranges corresponding to the changed column require re-computation thus necessitating a full refresh of the zone map. Certain other DDL operations on tables on which a zone map depends do not invalidate the zone map. For example, adding a new column or a constraint to an underlying table leaves the zone map definition and its content intact.

3.2.2 Staleness Tracking of Zone Map

DML operations on tables on which a zone map is defined make either none, or some, or entire content of the zone map stale depending on the nature of change. Thus zone map staleness state is maintained at different granularity levels of zone, partition (if fact table is partitioned) and the entire zone map. Staleness tracking of a basic zone map is straightforward since there are no joins to worry about. Staleness tracking methodology is same when the fact table of a basic or join zone map is changed. In contrast, staleness tracking methodology is more complicated when dimension table of a join zone map is changed. Below we describe separately the staleness tracking actions performed upon fact table DML, and upon dimension table DML.

3.2.2.1 Fact Table DML

Insert of a fact table row belonging to an existing zone (and partition) makes the min/max value ranges of that zone (and partition) stale. Subsequent inserts affecting the same zone (partition) simply skip staleness tracking. Observe that an inserted row may belong to a zone that is currently not present in the zone map, in which case no staleness tracking occurs. Instead of staleness marking, an alternate strategy would be to check the column values of inserted row with corresponding min/max value ranges and update them as necessary. However, using alternate strategy for a join zone map will require joins to the dimension tables thus rendering it impractical.

Delete of a fact table row doesn't affect the validity of min/max value ranges of corresponding zone (and partition) since data pruning will continue to work albeit with potential for reduced efficiency. So on deletes corresponding min/max value ranges are not marked as stale but they are marked as needing refresh.

Fact table row updates that neither modify min/max columns nor modify join key columns are ignored. Update of a fact table row belonging to an existing zone (and partition) that modifies min/max columns (i.e. min/max columns chosen from fact table) or in the case of join zone map update that modifies join keys (i.e. foreign keys joining to dimension tables) makes the min/max value ranges of corresponding zone (and partition) stale. As noted before, an alternative strategy of directly updating the affected min/max value ranges incurs excessive overhead upon join key changes since it requires joining to the corresponding dimension tables.

3.2.2.2 Dimension Table DML

Before we discuss in detail the staleness tracking methodology for DML operations on dimension tables, we need to consider the referential integrity state of the fact table with respect to each dimension table.

A DML operation on a dimension table of a join zone map usually requires joining the dimension table to the fact table to identify all affected zones and partitions. Because joining to the fact table is impractical, so the main challenge here is to identify a set of zones (and partitions) affected by dimension table DML without doing the join. The join can be avoided if referential integrity between the fact and dimension tables is declared and enforced within the RDBMS. Referential integrity guarantees that no orphan rows exist in the fact table except the ones with NULL foreign keys. Our focus here is on fact orphan rows with non NULL foreign keys. So from here on, when we refer to an orphan fact row it means a fact row with non NULL foreign key without a matching key in the corresponding dimension table. Without referential integrity, there is always a possibility that a newly inserted dimension row will join with one or more orphan fact rows. Most of the data warehouses do not declare and enforce referential integrity due to performance reasons. However, referential integrity usually exists and it is maintained in the application layer via data flow logic. This is good news because it implies that fact table will not contain orphan rows. For the purpose of staleness tracking, referential integrity state of the fact table is captured at the time a join zone map is created or refreshed. In Oracle, referential integrity state of a fact table with respect to each of the dimension tables is captured by computing the count of orphan fact rows in each zone (and partition). This is stored in the anti-join columns of a join zone map (see DESCRIBE command in section 3.1).

At first glance, it seems we can ignore dimension table row delete. But it can result in fresh orphan fact rows. So if a dimension row delete is followed by a dimension row insert using the same join key it can potentially affect the min/max values ranges of many zones (and partitions). Can we find the set of zones (and partitions) potentially affected by a dimension row delete without joining to the fact table? Yes, they are the ones with min/max value ranges encompassing the corresponding min/max values being deleted. We refer to them as "potentially stale" zones. It is important to note that because of the clustering of fact table data by min/max columns the set of zones (and partitions) becoming potentially stale is usually small. More importantly, zones (and partitions) marked as potentially stale are handled as follows:

1. They can continue to be used for data pruning.
2. They can become stale on subsequent dimension row inserts.
3. They are candidates for zone map refresh.

A newly inserted dimension row, as observed earlier, can join with orphan fact rows including the ones made orphan by prior delete operations. But also, a newly inserted dimension row with a duplicate join key can potentially join with non-orphan fact rows. Therefore, if dimension key is not declared to be unique then zones (and dimensions) with min/max value ranges not encompassing the corresponding new values are marked as stale. The good news is that join keys of dimension tables are usually declared with unique or primary key constraints in the RDBMS. So when dimension key is unique then zones (and partitions) with min/max value ranges excluding the corresponding new values are marked as stale provided they were previously marked as potentially stale or they have corresponding anti-join count > 0. Because of the latter condition, usually none or small set of zones (and partitions) become stale in unique join key case.

Dimension row updates that neither modify min/max columns nor modify join key columns are ignored. A dimension row update modifying the join key columns is equivalent to a dimension row delete with old key followed by a dimension row insert with new key. The following staleness tracking actions are performed upon dimension row update modifying column(s) of the join key:

1. Mark zones (and partitions) with min/max value ranges encompassing the corresponding values in the dimension row as potentially stale.
2. If the join key is unique, mark zones (and partitions) with min/max value ranges not encompassing the corresponding values in the dimension row as stale if they were previously marked as potentially stale or they have corresponding anti-join count > 0.
3. If the join key is not unique, mark zones (and partitions) with min/max value ranges encompassing the corresponding values in the dimension row as stale.

A dimension row update not modifying the join key but modifying min/max column values has potential to make stale the set of zones (and partitions) with min/max value ranges encompassing the corresponding old values but not encompassing the corresponding new values. This set can be reduced by joining to the fact table but this is not really necessary based on the following observation. Since fact table data is clustered by min/max columns the potential set of zones (and partitions) becoming stale is usually small.

3.2.2.3 Additional Staleness Tracking Notes

The enhanced staleness tracking strategy based on the uniqueness of dimension join keys and the capture of orphan fact row counts works very well in practice because referential integrity mostly exists whether or not it is declared and enforced in the RDBMS. This staleness tracking strategy based on chunks (i.e. zones) of fact table data enables the incremental refresh of zone maps to work without requiring the data change logs of underlying base tables.

It is important to point out that zone maps are automatically refreshed upon bulk operations (data loads, data moves) unless it is explicitly turned off. When refresh upon bulk operations is disabled only the affected portion of the zone map is marked stale or left with missing zones. For example, without automatic refresh fact table data moved from an old partition to a new partition will result in missing zones for the newly moved data. Without automatic refresh bulk data load generally causes minimal number of existing zones to go stale since data is loaded into new extents mostly forming new zones.

With automatic zone map refresh upon bulk operations being the default, it is the conventional DML operations that entail the staleness of zone map data. Since conventional DML operations are typically infrequent in DWs most of the zone map data tends to stay fresh especially when the dimension join keys are unique and referential integrity of the fact table data mostly exists.

3.2.3 Refresh Methods

A basic or join zone map can be refreshed to incorporate changes that may have occurred to the underlying tables since the time it was created or last refreshed. There are two refresh methods: full (aka complete) and incremental (aka fast) [18]. A full refresh is used when the entire zone map is marked as stale or it is explicitly chosen; otherwise an incremental refresh is used. Any queries that were using a zone map before its refresh operation started will continue to use it due to the default isolation level of read consistency (i.e. READ COMMITED) mode in Oracle. In read consistency mode read operations are never blocked by the DML operations and vice versa.

Zone map refresh operations also use the default read consistency mode, which means they read a consistent snapshot of the table data while concurrent DML operations could be committing changes to the tables. This concurrency between refresh and DML operations can lead to scenarios of refresh computing min/max value ranges based on the table data that has already grown stale. This requires identifying the newly computed min/max value ranges impacted by concurrent DML activity and leaving them as stale at the end of refresh. To accomplish this, we can first mark the min/max value ranges to be re-computed as "pending refresh" and commit this change. A concurrent DML operation that commits changes to the underlying tables will clear the “pending refresh” mark of affected zones (and partitions) thus indicating that corresponding re-computed min/max value ranges have already grown stale. The refresh sub-system at the end of re-computation will check for cleared “pending refresh” ones and leave them as stale. An upshot of concurrent DML activity is that the zone map after refresh may still contain stale zones but it will be a lot fresher than before. Also, if the entire zone map was stale before the refresh operation, it will no longer be fully stale after refresh but only parts of it may remain stale. This is an important outcome. In contrast, classic refresh methods [18] such as ones refreshing materialized joins lack DML activity tracking by data chunks (e.g. zones). Instead, they usually track concurrent DML activity at either object or object partition level, which means entire materialized object or object partitions may remain stale at the end of refresh. Such classic refresh methods perform retries to avoid leaving entire or large parts of the materialized result stale.

A zone map being refreshed could have missing zones due to newly added data to its fact table since it was created or last refreshed. Refresh should compute min/max value ranges for the missing zones but concurrent DML activity can occur on newly added data so we first need to insert rows in the zone map corresponding to missing zones and mark them also as “pending refresh”. Missing zones can be found by scanning the fact table and using the zone map itself to prune away all fact table data except new data forming the missing zones. Here is a case of zone map helping in its own refresh cause!

3.2.3.1 Full Refresh

The full refresh method is a straightforward method of re-populating the zone map table with freshly computed min/max value ranges using all of the data from table or tables on which it is defined. In preparatory stage, all rows from the zone map table are deleted. This helps to get rid of obsolete zones that were based on permanently deleted data from the fact table. Next, fact table is fast-scanned and zones are identified and inserted into the zone map with stale and “pending refresh” mark. For fast-scan, it is enough to scan first data block and skip rest of the blocks of each zone. After the preparatory stage, full refresh re-computes the min/max value ranges and merges them into corresponding zones that still remain as "pending refresh" in the zone map, and clearing both stale and “pending refresh” marks.

3.2.3.2 Incremental Refresh

The incremental refresh method involves computing min/max value ranges only for zones marked as stale or “potentially stale” or zones missing from the zone map. Classic incremental refresh method usually requires change logs that record row activity on underlying tables to identify and compute change delta. But incremental zone map refresh requires no such logs because staleness states are maintained per zone (and partition) within the zone map. We can use the staleness states stored in the zone map to prune away the fact table data belonging to fresh zones so that only the fact table data pertaining to stale or missing zones is read and min/max value ranges are computed. This is an example of zone map helping to make its own refresh operation as efficient as possible!

To account for concurrent DML activity during refresh operation, incremental refresh method also uses a preparatory stage. In this stage, fact table is scanned to identify missing zones while pruning away fact table data belonging to zones that currently exist in the zone map. Identified missing zones are added to the zone map. Missing and “potentially stale” zones are marked as stale, and then all stale zones are marked as “pending refresh”. After the preparatory stage, incremental refresh re-computes min/max value ranges for stale zones while pruning away fact table data belonging to fresh zones. Re-computed min/max value ranges are merged into corresponding zones that still remain as "pending refresh" in the zone map, and clearing both stale and “pending refresh” marks.

3.2.3.3 Additional Refresh Notes

For a partitioned fact table, full or incremental refresh will compute partition level min/max value ranges by rolling up the zone level min/max value ranges belonging to each partition. Partition level min/max value ranges inherit the staleness of any zone level min/max value range during the rollup operation.

Under read consistency mode, queries that execute during the time a zone map is being refreshed will have their execution plans built using this zone map unless it is fully stale. When a fully stale zone map is refreshed the query execution plans built without this zone map are invalidated so that they can be rebuilt to take advantage of the newly refreshed zone map.

4. DIMENSION BASED DATA PRUNING

A zone map is used when a query selects from a fact table and contains proper filter predicates on columns for which it is maintaining min/max value ranges by zones (and partitions) of the fact table. A proper filter predicate is one that specifies a value range using <, <=, =, =>, >, or BETWEEN operator, or it specifies an IN list of values or a LIKE operator comparing to a pattern with constant prefix. For such a query, zone map can be used to prune fact table data at appropriate granularity levels of partitions (if fact table is partitioned), extents (i.e. allocation units of contiguous blocks on disk), and data block zones for disk based fact table, or IMCUs for in-memory based fact table. When fact table is accessed via an index then zone map can still be used to prune individual rowids thus avoiding row fetches that entail random IO. Various pruning methods are explained in the following sections. The query shown below is Q1.1 from SSB.

```
SELECT SUM(lo_extendedprice * lo_discount)
FROM lineorder, dwdate
WHERE lo_orderdate = d_datekey AND
      d_year = 1993 AND
      lo_discount BETWEEN 1 AND 3 AND
      lo_quantity < 25;
```

To begin with, assume lineorder is not partitioned and no zone map defined. Following is a possible execution plan for above query.

```
-----------------------------------------
|Id|Operation             |Name      |
-----------------------------------------
| 0|SELECT STATEMENT      |          |
| 1| SORT AGGREGATE       |          |
|*2|  HASH JOIN           |          |
| 3|   JOIN FILTER CREATE|:BF0000   |
|*4|    TABLE ACCESS FULL|DWDATE    |
| 5|   JOIN FILTER USE   |:BF0000   |
|*6|    TABLE ACCESS FULL|LINEORDER|
-----------------------------------------
```

Plan steps 3 and 5 indicate the creation and use of Bloom filter named BF000 based on the filter predicate d_year = 1993 on dwdate table, which is on the left side of the hash join.

4.1 Join Pruning

Pruning of fact table data can be performed based on the join to a dimension table with proper filter predicate provided a zone map is created to maintain min/max value ranges of the join key (i.e. foreign key). The join pruning is made possible by the construction of min/max value range of the join key when Bloom filter is built. Join pruning is a dimension based pruning method even though the zone map is basic. Let us create a basic zone map on lo_orderdate of lineorder, which is a join key to dwdate. The execution plan for Q1.1 changes to the following.

```
-------------------------------------------------------
|Id|Operation                           |Name      |
-------------------------------------------------------
| 0|SELECT STATEMENT                    |          |
| 1| SORT AGGREGATE                     |          |
|*2|  HASH JOIN                         |          |
| 3|   JOIN FILTER CREATE               |:BF0000   |
|*4|    TABLE ACCESS FULL               |DWDATE    |
| 5|   JOIN FILTER USE                  |:BF0000   |
|*6|    TABLE ACCESS FULL WITH ZONEMAP|LINEORDER|
-------------------------------------------------------
```

At plan step 3, the min/max value range created with Bloom filter is passed down to the operation at plan step 6 to perform join pruning. Join pruning is effective if fact data is clustered by the join key lo_orderdate and hence d_datekey, which in turn is correlated to the filter predicate column d_year.

4.2 Block Pruning

Join pruning does not work well in practice because it requires the join key to be correlated to the filter predicate column. We can get much better pruning if fact data is explicitly clustered by the filter predicate column of the dimension, and a join zone map is defined. So, instead of a basic zone map let us define a join zone map on lineorder joined to dwdate with min/max value ranges on dimension column d_year, and re-execute Q1.1. The execution plan remains the same as shown in previous section but the underlying mechanism changes from join pruning to block pruning. In other words, filter predicate d_year = 1993 is directly used to compare with min/max value ranges in the join zone map and perform data block pruning zone by zone.

4.3 Partition Pruning

Partition pruning via zone map applies when the fact table is partitioned. So let us make lineorder table partitioned by yearly ranges of lo_orderdate. The execution plan for Q1.1 will change to the following.

```
-----------------------------------------------------
|Id|Operation            |Name     |Pstart |Pstop  |
-----------------------------------------------------
| 0|SELECT STATEMENT     |         |       |       |
| 1| SORT AGGREGATE      |         |       |       |
|*2|  HASH JOIN          |         |       |       |
| 3|   JOIN FILTER CREATE |:BF0000  |       |       |
|*4|    TABLE ACCESS FULL |DWDATE   |       |       |
| 5|   JOIN FILTER USE    |:BF0000  |       |       |
| 6|    PARTITION RANGE   |         |       |       |
|  |          ITERATOR    |         |KEY(ZM)|KEY(ZM)|
|*7|     TABLE ACCESS FULL|         |       |       |
|  |          WITH ZONEMAP|LINEORDER|KEY(ZM)|KEY(ZM)|
-----------------------------------------------------
```

Plan steps 6 and 7 showing KEY(ZM) indicate that partition pruning is occurring based on the zone map min/max key (i.e. d_year). In this example, all but one partition belonging to 1993 are pruned away. Note that block pruning still applies to partitions that are not pruned away, which is important because predicate value range may partially cover a partition. Thus this example shows the use of zone map to perform partition as well as zone level block pruning.

4.4 IMCU Pruning

When a table is cached in Oracle In-Memory Columnar store, each column is cached in a series of in-memory compression units (IMCUs). Each IMCU has well defined boundary in terms of contiguous data blocks on disk it covers, and it carries a min/max value range computed from the column data it contains. If a query contains a proper filter predicate, IMCUs storing data of that predicate column can be pruned by comparing the filter predicate value range against the min/max value range carried in each IMCU. But what if the proper filter predicate is on a column of the dimension table joined to the in-memory fact table? IMCUs can still be pruned provided a join zone map is created. Using the well defined contiguous block boundaries, each IMCU can be mapped on to zones in the join zone map. If min/max value ranges in mapped zones do not intersect with the filter predicate value range then corresponding IMCUs can be pruned.

Assume lineorder is cached in In-Memory Columnar store. The execution plan for Q1.1 will be as shown below.

```
----------------------------------------------------
|Id|Operation           |Name     |Pstart |Pstop  |
----------------------------------------------------
| 0|SELECT STATEMENT    |         |       |       |
| 1| SORT AGGREGATE     |         |       |       |
|*2|  HASH JOIN         |         |       |       |
| 3|   JOIN FILTER CREATE|:BF0000  |       |       |
|*4|    TABLE ACCESS FULL|DWDATE   |       |       |
| 5|   JOIN FILTER USE   |:BF0000  |       |       |
| 6|    PARTITION RANGE  |         |       |       |
|  |       ITERATOR     |         |KEY(ZM)|KEY(ZM)|
|*7|     TABLE ACCESS    |         |       |       |
|  |       INMEMORY FULL |         |       |       |
|  |       WITH ZONEMAP  |LINEORDER|KEY(ZM)|KEY(ZM)|
----------------------------------------------------
```

Note the keyword INMEMORY shown on plan step 7. Since lineorder is partitioned the zone map is also used to do partition pruning, which means all IMCUs belonging to pruned partitions are skipped. This example illustrates data pruning taking place at different granularity levels of partitions and individual IMCUs.

4.5 Extent Pruning

In Oracle, a table is stored on disk in a series of storage extents, each made up of a contiguous set of blocks allocated together. When a table is scanned its extent map is read and multi-block IO is issued for a fixed number of blocks from the current extent. When all blocks in current extent are fetched then next extent is processed. Similar to IMCU mapping, each extent can be mapped on to zones in the zone map. Therefore, extent pruning works very much like IMCU pruning except that it is used for a fact table stored on disk as opposed to being cached in In-Memory Columnar store. Currently extent pruning using a basic or join zone map is performed during the generation of data granules (i.e. data units of work for parallel processes) by the parallel query plans but extent pruning can be easily extended to serial query plans.

4.6 Index Rowid Pruning

If the query optimizer chooses an index to access rows from fact table on which a basic or join zone map is defined, rowids from index scan can be pruned before they are used to fetch the table rows. The idea is to map each rowid to a zone identifier and compare the value range represented by a proper filter predicate in the query to the corresponding min/max value range in the identified zone and pruning the rowid when no intersection is found. Following is the execution plan for Q1.1 using index to access data from lineorder as shown at plan step 2.

```
--------------------------------------------------------
|Id|Operation                | Name    |Pstart | Pstop |
--------------------------------------------------------
| 0|SELECT STATEMENT         |         |       |       |
| 1| SORT AGGREGATE          |         |       |       |
|*2|  TABLE ACCESS BY        |         |       |       |
|  |     LOCAL INDEX ROWID   |         |       |       |
|  |     BATCHED WITH ZONEMAP|LINEORDER|       |       |
| 3|   NESTED LOOPS          |         |       |       |
|*4|    TABLE ACCESS FULL    |DWDATE   |       |       |
| 5|    PARTITION RANGE      |         |       |       |
|  |       ITERATOR          |         |KEY(ZM)|KEY(ZM)|
|*6|     INDEX RANGE SCAN    |LO_INDEX |KEY(ZM)|KEY(ZM)|
--------------------------------------------------------
```

Zone map is first used at plan steps 5 (indicated by KEY[ZM]) to prune partitions, and later at plan step 2 it is used again to prune index rowids batched from index scans of un-pruned partitions.

5. EXPERIMENTAL RESULTS

We conducted performance experiments to measure the benefit of data pruning resulting from dimension based clustering and the dimension based zone map using Star Schema Benchmark (SSB) [15]. This benchmark contains one fact table (lineorder) and four dimension tables (dwdate, customer, supplier, part), and four sets of related queries called query flights: QF1 (Q1.1, Q1.2, Q1.3), QF2 (Q2.1, Q2.2, Q2.3), QF3 (Q3.1, Q3.2, Q3.3, Q3.4), and QF4 (Q4.1, Q4.2, Q4.3).

We used SSB with scale factor = 1000 (approximately 1 TB size) with lineorder table containing 6 billion rows. It was partitioned on yearly ranges of lo_orderdate. There were no indexes on the lineorder table but unique indexes on the join keys of dimension tables. Our experiments were conducted using Oracle 12c RDBMS running on a single node Exadata system [9] with 64 cores. We performed three different experiments: (1) query performance experiment with lineorder table data clustered by two and three dimensions with corresponding two and three dimensional zone maps, (2) performance cost measurement of bulk loading data into lineorder table with two and three dimensional clustering, and (3) performance cost measurement of creating and refreshing join zone map with two and three dimensions.

All original benchmark tables as well as lineorder tables with two and three dimensional clustering were created using Oracle's patented hybrid columnar compression [22] that is typically used in DWs. Two flavors of query performance experiments were conducted by running the queries in QF2, QF3, and QF4 with all tables on disk, and then with all tables cached in Oracle Database In-Memory Columnar store [13].

5.1 Query Performance

Two Dimensional Clustering: Supplier and customer dimensions appear in QF2, QF3 and QF4. We clustered the lineorder data by the hierarchies of these two dimensions into lineorder_sc as shown below.

```
CREATE TABLE lineorder_sc ( . . . )
  CLUSTERING
    lineorder LEFT JOIN supplier
                ON (lo_suppkey = s_suppkey)
              LEFT JOIN customer
                ON (lo_custkey = c_custkey)
    BY INTERLEAVED ORDER
     ( (s_region, s_nation, s_city),
       (c_region, c_nation, c_city) );
```

We bulk loaded the empty lineorder_sc table partition by partition, and then created a corresponding two dimensional join zonemap as follows.

```
CREATE MATERIALIZED ZONEMAP lineorder_sc_zmap AS
SELECT SYS_OP_ZONE_ID(lo.rowid),
       MIN(s_region), MAX(s_region),
       MIN(s_nation), MAX(s_nation),
       MIN(s_city),   MAX(s_city),
       MIN(c_region), MAX(c_region),
       MIN(c_nation), MAX(c_nation),
       MIN(c_city),   MAX(c_city)
FROM lineorder LEFT JOIN supplier
                 ON (lo_suppkey = s_suppkey)
               LEFT JOIN customer
                 ON (lo_custkey = c_custkey)
GROUP BY SYS_OP_ZONE_ID(lo.rowid);
```

Three Dimensional Clustering: Supplier and customer dimensions appear in QF2, Q3F and QF4, and part dimension appears in QF2 and QF4. We clustered lineorder data by the hierarchies of these three dimensions into lineorder_scp as shown below.

```
CREATE TABLE lineorder_scp
  CLUSTERING
    lineorder LEFT JOIN supplier
                ON (lo_suppkey = s_suppkey)
              LEFT JOIN customer
                ON (lo_custkey = c_custkey)
              LEFT JOIN part
                ON (lo_partkey = p_partkey)
    BY INTERLEAVED ORDER
     ( (s_region, s_nation, s_city),
       (c_region, c_nation, c_city),
       (p_mfgr, p_category, p_brand1) );
```

We bulk loaded data into lineorder_scp partition by partition, and then created a corresponding three dimensional join zone map lineorder_scp_zmap. This zone map is similar to two dimensional zone map but with an additional join to part table and additional min/max aggregates on the part dimension hierarchy columns.

Table 1 shows the performance improvement of QF2, QF3, and QF4 queries due to clustering and join zone map pruning with respect to query performance using original schema tables. In this experiment, both the original schema tables and the clustered tables (lineorder_sc and lineorder_scp) were on disk. All queries were run using a degree of parallelism of 64 matching the number of cores. The x-times performance improvement is shown in term of query run times. The run time units are not disclosed for competitive reasons.

Table 1: X-times query performance improvement using disk resident tables

Query	**2-Dim Clustering**	**3-Dim Clustering**
Q2.1	2.37 x	10.61 x
Q2.2	2.32 x	8.18 x
Q2.3	2.45 x	8.41 x
Q3.1	2.91 x	2.95 x
Q3.2	4.68 x	4.38 x
Q3.3	5.23 x	4.56 x
Q3.4	26.82 x	6.06 x
Q4.1	3.36 x	4.60 x
Q4.2	15.14 x	22.77 x
Q4.3	6.80 x	38.04 x
AVG	**4.27 x**	**6.67 x**

The last line in table 1 shows improvement factors designated as AVG are based on the arithmetic sum of all query run times.

Run time improvements were proportional to the improvements in the CPU time and disk IOs. For example, table 2 shows these numbers for Q3.2 without and with two dimensional clustering and zone map pruning.

Table 2: Improvements in CPU time, IOs, etc. for Q3.2

2-Dimensional Clustering and Join Zone Map	**CPU**	**IO**	**Read Requests**	**Read Bytes**
No	2506	30223	551 K	534 GB
Yes	383	2651	56 K	51 GB

The table 3 shows the performance improvement of QF2, QF3, and QF4 when the original schema tables and the clustered tables (lineorder_sc and lineorder_scp) were cached in Oracle Database In-Memory Columnar store. Because the run times were significantly shorter we had to reduce the degree of parallelism to 16 to control the rounding error in the reported times.

As in table 1, the last line in table 3 shows improvement factors designated as AVG are based on the arithmetic sum of all query run times.

Table 3: X-times query performance improvement using in-memory tables

Query	**2-Dim Clustering**	**3-Dim Clustering**
Q2.1	1.58 x	4.21 x
Q2.2	1.48 x	3.91 x
Q2.3	1.41 x	3.67 x
Q3.1	4.39 x	3.83 x
Q3.2	6.81 x	5.27 x
Q3.3	5.57 x	4.07 x
Q3.4	1.84 x	1.73 x
Q4.1	4.43 x	5.82 x
Q4.2	2.24 x	2.46 x
Q4.3	2.29 x	2.27 x
AVG	**2.63 x**	**3.61 x**

Improvement factors shown in table 1 and table 3 confirm that effective data pruning helps in reducing the query run times whether the table data is on disk or cached in memory. The improvement factors are more dramatic in the case of disk resident tables because data pruning eliminates redundant IO as well as CPU processing.

5.2 Cost of Clustering Fact Data

The cost of clustering of fact data is significant. The costs are due to the following factors.

- Data needs to be sorted and this may involve spilling of sort segments to disk.
- Fact table needs to be joined to dimension tables. Joins will be typically hash outer joins. With parallel query execution, data distribution occurs unless plans are pipelined. However, pipelined plans require broadcast of left side of the join that is suitable for small tables like dwdate. With large dimension tables the data on both sides of hash join is usually hash distributed, and it could spill to disk.

We performed the bulk loading of data into different versions of the lineorder table (original lineorder, lineorder_sc, lineorder_scp) partition by partition. Table 4 shows relative run time performance of loading data into the table with two and three dimensional clustering in comparison to loading data into the original table.

Table 4: Relative cost of clustering compared to simple load

2- Dim Clustering Load	**3-Dim Clustering Load**
2.83 x	3.24 x

For bulk data loading with two and three dimensional clustering the following was percentage of activities.

Table 5: Cost of SQL operations for data loads with clustering

SQL Operations	% of activity for 2-Dim clustering	% of activity for 3-Dim clustering
Scan of original lineorder	5.02%	3.32%
Outerjoin to supplier (+ distribution)	34.12%	30.77%
Outerjoin to customer (+ distribution)	33.97%	20.89%
Outerjoin to part (+ distribution)	NA	22.54%
Sort	12.17%	10.15%
Load to target	1.17%	1.07%

5.3 Cost of Join Zone Map

The cost of creating join zone maps was significantly lower than the cost of clustering. Some optimization techniques, for example, pushing partial GROUP BY below joins [4] were helpful here. Table 6 shows the percentage of run time taken to create two and three dimensional join zone maps compared to two and three dimensional clustering of data respectively.

Table 6: Cost of join zone map creation relative to data clustering

2- Dim Join Zone Map	3-Dim Join Zone Map
9.22 %	13.23 %

We also conducted experiments by making some percentage of join zone map stale by updating some of the data in lineorder_sc and lineorder_scp tables, and then performing incremental refresh of the join zone map. In another scenario we rendered entire join zone map stale and then performed full refresh. Table 7 shows the percentage of two and three dimensional join zone map that was stale and the run time it took to incrementally refresh it relative to the time it took to fully refresh it.

Table 7: Cost of incremental refresh relative to full refresh of join zone map

Percent Stale	2-Dim Join Zone Map	Percent Stale	3-Dim Join Zone Map
6.0%	4..12%	8.66%	5.50%
19.5%	12.57%	21.75%	12.40%

The first two columns show for a two dimensional join zone map the percent stale and the percent time it took for incremental refresh compared to a full refresh, respectively The next two columns show similar information for a three dimensional join zone map. This experiment proves the point of zone map itself being used to skip over the fact data corresponding to fresh zones when performing incremental refresh.

6. CONCLUSION

Zone map or storage index on single tables have been offered in several commercial products, in particular IBM® Netezza® system [10], Amazon Redshift [21], Oracle Exadata system [9], and Oracle Database In-Memory Columnar store [14].

The IBM Netezza system automatically creates zone maps [10] stored in internal system tables, and refreshes existing zone maps upon insert, update, load data into table, or generate statistics. For every column in a user table with date, timestamp, byteint, smallint, integer, bigint data type, it maintains the minimum and maximum values per data extent. Deletions do not have immediate effect on zone maps. We note that Netezza maintains a zone map on every insert, update, and load while we maintain on every bulk data load and bulk data move operation. Zone maps in Netezza are used for extent pruning based on the predicates on the columns on which zone maps are maintained. This is a subset of our pruning methods where in addition to extent pruning we perform IMCU pruning, join pruning, partition pruning, and index rowid pruning.

In Amazon Redshift, user can create a table with one or more columns as the sort keys. When data is loaded, it is stored on disk in sorted order. Further, Redshift stores columnar data in 1MB disk blocks. For each disk block it creates min/max value ranges on sort key columns. If a query uses range-based filter predicates on sort key columns, the query processor uses the min/max value ranges to skip over unqualified disk blocks during the table scan.

Oracle Exadata system [9] consists of RDBMS servers processing general SQL operations (e.g. joins, aggregations), and storage servers processing disk accesses and evaluation of expressions and filter predicates. Exadata storage servers support a construct analogous to zone map on a single table called Storage Index [12]. Storage index maintains min/max value ranges of a column and a flag indicating if any nulls exist there. It is provided for up to 8 different columns of a table within each 1MB disk region. Storage indexes reside in the memory of the storage servers and they eliminate unnecessary I/O by pruning irrelevant data regions based on filter predicates. Storage indexes are automatically created and maintained after a storage server receives disk scan requests with repeated predicates on the same columns. Efficiency of storage index relies on the clustering of data and for its effectiveness it is recommended to order data on loads possibly using the clustering clause presented in this paper. Storage indexes can perform join pruning using min/max value range created along with a Bloom filter on the left side of the hash join and pushed to the storage servers.

Similar to the Oracle Exadata system storage index, Oracle Database In-Memory Column store index maintains min/max value range of the column data held in each IMCU, and performs IMCU pruning.

One of the inspirations for the dimensions based data clustering and dimension based zone maps was the concept of invisible join proposal [16], which is very similar to a semi-join or a late materialized hash join [17] in star schemas. In the proposed scheme, semi-join produces an array (or a bitmap) of fact table join keys obtained after filtering the dimension table. If the join key on fact table was correlated with the filtering column of dimension table then Between Predicate Rewriting [16] could be applied. The authors observed that even though in many cases there is no such correlation, it can be introduced by common dictionary encoding of both the fact join key and dimension join key. It was our observation that this would particularly be the case if dictionary encoding on the dimension table was correlated with dimension hierarchy in the table. We experimented with adding dictionary encoding to the dimension table that was correlated with dimension hierarchy and enforcing the same dictionary encoding on the foreign key of the fact able. The experiments showed significant improvements in pruning in case of semi-join. Following that result, rather than using a coordinated hierarchy based dictionary encoding scheme, we used multi-dimensional clustering based on the joins between dimension tables and the fact table to achieve similar pruning advantage via join zone maps.

The experiments we conducted show significant improvement in the performance of star queries when the fact data is clustered by dimension hierarchies in conjunction with join zone maps on clustered fact data.

Possible future enhancements to the dimension based clustering and zone maps include extending full support for snowflake queries. Observe that snowflake queries can still benefit from the current support for star queries with pruning based on filter predicates on normalized dimension tables that are directly joined to the fact table. Other possible enhancements include adding the support for discovering the clustering of data in existing tables by its local columns as well as by columns in other tables via joins, and support for breaking down bulk data load into a table with defined clustering into a series of loads such that spilling of data during joins and sort used for clustering is minimized yet achieving good clustering effect.

7. ACKNOWLEDGMENT

We acknowledge the contributions of other members of our development team: Rupa Rangaiyengar, Randall Bello, Ananth Raghavan, Sankar Subramanian and Manish Pratap Singh.

8. REFERENCES

[1] Kimball, R., Ross, M. The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (2nd edition), *John Wiley & Sons, Inc.*, New York, 2002.

[2] Weininger, A. Efficient execution of joins in a star schema. *SIGMOD*, pages 542–545, 2002.

[3] Bloom, B. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), pp. 422-426, 1970.

[4] Bellamkonda S., Ahmed R., et al. Enhanced Subquery Optimizations in Oracle. *Proceedings of the VLDB Endowment*, 2009.

[5] Bernstein, P. A., Chiu, D. W. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1), pp 25–40, 1981.

[6] Schneider, D. A., DeWitt, D. J. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. *Proceedings of the VLDB Endowment*, 1990.

[7] Oracle Partitioning with Oracle Database 12c. *Oracle White Paper*, Sep 2014. http://www.oracle.com/technetwork/database/options/partitioning/partitioning-wp-12c-1896137.pdf

[8] Ceri, S., Negri, M., Pelagatti, G. Horizontal Data Partitioning in Database Design, *SIGMOD*, pp 128-136, 1982.

[9] A Technical Overview of Oracle Exadata Database Machine and Exadata Storage Server. *Oracle White Paper*, June 2012. http://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf

[10] IBM Netezza Data Warehouse Appliance. http://www-01.ibm.com/software/data/netezza/

[11] Guido MoerKotte, Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. *Proceedings of the VLDB Endowment 1998.*

[12] Smart Scans Meet Storage Indexes. *Oracle Magazine*, May/June 2011. http://www.oracle.com/technetwork/issue-archive/2011/11-may/o31exadata-354069.html

[13]Oracle Database In-Memory. *Oracle White Paper*, July 2015. http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html

[14] Mukherjee, N., et al. Distributed Architecture of Oracle Database In-memory. *Proceedings of the VLDB Endowment*, 2015.

[15] O'Neil, P., O'Neil, B., Chen, X. Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

[16]] Abadi D. J., Madden S. R., Hachem N., ColumnStores vs. RowStores: How Different Are They Really? *SIGMOD*, 2008.

[17] Abadi D. J., Myers D.S., DeWitt D. J., Madden S.R. Materialization Strategies in a Column-Oriented DBMS. *Proceedings of ICDE*, 2007.

[18] Bello, R. G., et al. Materialized Views in Oracle, *Proceedings of the VLDB Endowment*, 1998.

[19] Warren, H. S. Hacker's Delight, Addison-Wesley. 2002.

[20] Stonebraker, M., et al. C-Store: A Column-oriented DBMS, *Proceedings of the VLDB Endowment*, 2005.

[21] Amazon Redshift. *Database Developer Guide (API Version 2012-12-01)*. Choosing Sort Keys. http://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html

[22] Hybrid Columnar Compression (HCC) on Exadata. *Oracle White Paper*, November 2012. http://www.oracle.com/technetwork/database/exadata/ehcc-twp-131254.pdf