Query Optimization in Oracle 12c Database In-Memory

Dinesh Das[*], Jiaqi Yan[*], Mohamed Zait[*], Satyanarayana R Valluri[†], Nirav Vyas[*], Ramarajan Krishnamachari[*], Prashant Gaharwar[*], Jesse Kamp[*], Niloy Mukherjee[*]

[*] Oracle USA, {firstname.lastname}@oracle.com

[†] EPFL, Switzerland, satya.valluri@epfl.ch (Work done while at Oracle)

ABSTRACT

Traditional on-disk row major tables have been the dominant storage mechanism in relational databases for decades. Over the last decade, however, with explosive growth in data volume and demand for faster analytics, has come the recognition that a different data representation is needed. There is widespread agreement that in-memory column-oriented databases are best suited to meet the realities of this new world.

Oracle 12c Database In-memory, the industry's first dual-format database, allows existing row major on-disk tables to have complementary in-memory columnar representations. The new storage format brings new data processing techniques and query execution algorithms and thus new challenges for the query optimizer. Execution plans that are optimal for one format may be sub-optimal for the other.

In this paper, we describe the changes made in the query optimizer to generate execution plans optimized for the specific format – row major or columnar – that will be scanned during query execution. With enhancements in several areas – statistics, cost model, query transformation, access path and join optimization, parallelism, and cluster-awareness – the query optimizer plays a significant role in unlocking the full promise and performance of Oracle Database In-Memory.

1. INTRODUCTION

A confluence of events is reshaping the data processing world. Rapid advances in hardware are bringing faster processors; faster, cheaper and higher-capacity memory; larger persistent storage devices; and faster interconnect with higher bandwidth. At the same time, the number of data sources and their variety is exploding. To make sense of and monetize the large volume of data, businesses, large and small, need complex analytic queries answered faster than ever.

Column-oriented databases have risen to meet this challenge. Larger memory sizes and high compression ratios allow much columnar data to fit entirely in memory. New algorithms allow evaluating relational operations directly on the compressed columnar data. Every major database vendor, including Oracle, has support for column-oriented databases.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

With a new storage format comes new query processing techniques. While tables stored in memory in columnar form are generally faster to access than those on disk, analytic queries are rarely just simple table scans. They usually involve complex joins and aggregations. In addition, the applications themselves are becoming more complicated, with OLTP and data warehouse workloads expected to run in the same database.

Various vendors have taken different approaches to generating execution plans for in-memory columnar tables compared to row major on-disk tables. Some make no changes to the query optimizer with the expectation that the change in data format itself will make the plans perform better. Other systems have implemented simple heuristics to allow the optimizer to generate different plans. Still others limit their optimizer enhancements to specific workloads like star queries.

In this paper, we argue that a comprehensive optimizer redesign is necessary to handle a variety of workloads on databases with varied schemas and different data formats running on arbitrary hardware configurations with dynamic system constraints (like available memory). An execution plan generated by an optimizer designed for an on-disk row major format may be sub-optimal on an in-memory columnar format. It is imperative that the query optimizer use a holistic approach, taking into account not only all the operations in a query but also the specific storage formats and system configuration. Without this requirement, as we show in this paper, many workloads will see limited benefit or none at all from in-memory columnar tables.

In this paper, we describe the Oracle query optimizer in Oracle 12c. Section 2 provides a brief overview of Oracle 12c Database In-Memory. Section 3 presents the enhancements in the query optimizer for in-memory. Section 4 contains experiments to validate the optimizer changes. Section 5 provides an overview of related work. Finally, we conclude the paper in Section 6.

2. ORACLE DATABASE IN-MEMORY

This section provides a brief introduction to Oracle DBIM; more details are in [13] and [18].

Oracle 12c Database In-Memory (DBIM) is a dual-format database where data from a table can reside in both columnar format in an in-memory column store and in row major format on disk. The in-memory columnar format speeds up analytic queries and the row major format is well-suited for answering OLTP queries. Note that scanning on-disk tables does not necessarily mean disk I/O; some or all of the blocks of the table may be cached in the row major buffer cache [4].

A dedicated in-memory column store called the *In-Memory Area* acts as the storage for columnar data. The in-memory area is a subset of the database shared global area (SGA).

The creation and storage of columnar data in the in-memory area is called *population*. Population is done from the on-disk row major data. The columnar representation consists of contiguously allocated chunks called *In-Memory Compression Units* (IMCUs). Each IMCU contains the contents from a set of rows. Within each IMCU, each column is stored separately and contiguously as a column *Compression Unit* (CU). Partitions and sub-partitions of a partitioned table are organized into IMCUs independently of each other; an IMCU cannot span multiple partitions. A single partition can, however, have multiple IMCUs.

Each column may be compressed at different compression levels and it is even possible for different CUs of the same column to be compressed differently. Different compression levels are suitable for different use cases. There are three classes of compression algorithms optimized for different criteria: for DML performance, for query performance and for space capacity. Figure 1 shows an overview of the storage format of an in-memory table.

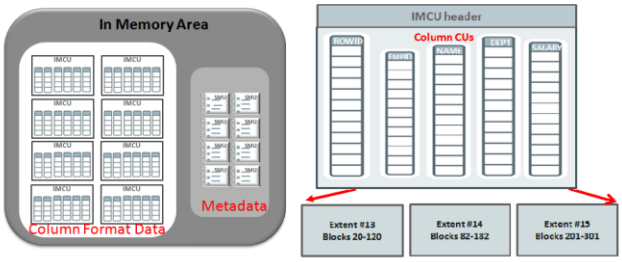

Figure 1: Data Format in Oracle DBIM Column Store

It is not necessary for the entire table to be populated in-memory nor is it necessary for all columns of a table to be populated. Thus, if memory is a constraint, users can decide what tables and columns to enable for in-memory. For partitioned tables, it is possible to enable some of the partitions for in-memory and leave others on disk.

Oracle Real Application Clusters (RAC) [20] is a shared-disk cluster allowing multiple Oracle instances running on different nodes to access a common database. While the on-disk data is shared, each node has its own private in-memory area that is accessible only to that node's instance.

There are two ways to populate tables in-memory in RAC: DUPLICATE and DISTRIBUTE. Under DUPLICATE mode, all the IMCUs are populated on at least two instances. This provides fault tolerance in case one of the instances goes down. The DUPLICATE ALL mode populates all the IMCUs on all instances; this provides full fault tolerance. The DISTRIBUTE mode has three options: distribution by PARTITION, by SUBPARTITION, or by ROWID RANGE. The DISTRIBUTE AUTO option lets Oracle choose the distribution method.

The DISTRIBUTE mode provides a way to efficiently use the combined in-memory areas of all nodes and access the data using parallel queries on the in-memory tables. However, if the query is executed in serial on a single instance, there will be disk I/O since parts of the table will be populated in-memory on other nodes. Running in parallel but with insufficient processes will also result in disk I/O. Thus, even though a table may be fully populated in-memory in RAC, it is possible to incur disk I/O depending on the execution plan.

An in-memory scan starts by locating the IMCUs that contain the required rows. Within each IMCU, it is only necessary to look at the CUs of the columns referenced in the query. All necessary columns must be in-memory-enabled to use an in-memory scan.

Each CU has an in-memory *storage index* that contains the minimum and maximum column values for all rows in that CU. The storage index is used to prune CUs using predicates on the column. Even if a CU is not pruned, certain predicates are much more efficiently evaluated on compressed data instead of the SQL execution engine. For this reason, Oracle pushes down many types of predicates to the in-memory scan.

When rows are deleted from a table, the corresponding entries in the IMCUs for these rows are marked as invalid. When new rows are inserted into an in-memory table, they are first stored in an in-memory row major *transaction journal* until they reach a certain threshold, after which the IMCUs are rebuilt. Thus, when scanning an in-memory table, invalid rows from the CUs are skipped and additional rows in the transaction journal are scanned. Transactional consistency is maintained in all cases.

3. QUERY OPTIMIZATION

The in-memory columnar format of tables fundamentally changes the performance of operations like scans, joins and aggregations. New query execution techniques like vector predicate evaluation and bloom filter pushdown into scans are possible on columnar tables. On Oracle RAC (a shared-disk cluster), in-memory tables can be distributed in the in-memory areas of different nodes, making it shared-nothing with respect to the separate in-memory column stores. A query optimizer designed only for row major tables is unlikely to yield plans that are optimal when some or all of the tables are in-memory.

In this section, we describe the enhancements in the optimizer in Oracle DBIM to be cognizant of in-memory tables. These enhancements ensure that the optimizer generates plans that are optimal regardless of whether tables are fully, partially, or not at all, in-memory and whether queries are executed on a single node or on a cluster.

Oracle DBIM also introduces a new in-memory aggregation feature that accelerates many classes of analytic queries with complex joins and aggregations. We do not describe it in this paper but more details are available in [19].

In the remainder of this paper, *on-disk table* refers to a traditional row major table stored on disk while *in-memory table* refers to an on-disk table which is also populated in-memory. As mentioned in Section 2, a scan of an on-disk table does not mean physical disk I/O.

3.1 In-memory Statistics

It is well known that a query optimizer needs accurate statistics [5] to generate optimal plans. Broadly speaking, there are two kinds of statistics: object statistics and system statistics.

Object statistics on tables can be categorized as logical or physical. Logical statistics are a function only of the data, not the table's storage format. Such statistics include the number of rows, average row length, column histograms, column minimum and maximum values and so forth. (Auxiliary structures like indexes also have logical statistics like number of distinct index keys.) Physical statistics of tables are a function of the data, their storage representation and database settings. For row major on-disk tables in Oracle, they include, among others, the number of blocks and number of chained rows.

System statistics include the number of CPUs, CPU speed, I/O throughput, number of nodes in the cluster, available memory, and so on.

The optimizer uses logical object statistics primarily to estimate cardinalities of various operations like table and index scan, join, and GROUP BY. The estimated cardinalities, along with physical object statistics and system statistics are used to estimate costs for these operations. Accurate costs are necessary to choosing the best plan from among several alternatives; this means that complete and up-to-date statistics are a must for the optimizer.

For in-memory tables, logical object statistics are still necessary since they are used to estimate cardinalities which are independent of the physical storage representation of the data. Physical object statistics, however, depend on the actual data storage format. Since Oracle DBIM allows tables to be declared in-memory, new physical object statistics are needed for such tables. Indexes cannot be declared in-memory and thus no new statistics are needed for them.

Oracle DBIM introduces the following new physical table statistics (which we will call *in-memory statistics* in this paper): number of IMCUs, number of in-memory blocks, number of in-memory rows, number of in-memory transaction journal rows, and the in-memory quotient. For partitioned tables, in-memory statistics, like other table statistics, are maintained at the partition level. Table-level in-memory statistics are derived by aggregating partition-level statistics.

An in-memory block corresponds to a specific disk block of the table. If the table is fully populated in-memory, there will be an equal number of in-memory blocks and on-disk blocks. A table partially populated in-memory will have a smaller number of in-memory blocks compared to the on-disk blocks. The *in-memory quotient* is the ratio of the in-memory blocks to the number of on-disk blocks. Its value ranges from 0 to 1, both inclusive, and it indicates the fraction of the table that is populated in-memory. If an in-memory table is partitioned, the in-memory quotient is computed for each partition, since in-memory statistics are partition-specific.

In-memory statistics are maintained in real-time since parts of a table can be populated in or evicted from the in-memory area at any time. For example, if enough new rows are inserted into a table, then the number of IMCUs might change. If a large number of rows are inserted and there is not enough space in the in-memory area, then some rows will remain on disk only which means an in-memory quotient less than 1. The optimizer takes into account the current in-memory statistics during query compilation to ensure accurate costing and plan selection. Cached query execution plans are invalidated using the same mechanism as when regular table statistics change.

For queries on partitioned tables, the optimizer can sometimes determine the partitions that must be scanned by analyzing the predicates on the table. If this partition pruning [10] is possible at compilation time, the optimizer computes in-memory statistics for the remaining partitions by aggregating the in-memory statistics of each partition.

Consider a table with four partitions, P_1 through P_4, where P_1 and P_2 are on-disk while P_3 and P_4 are in-memory. For these partitions, suppose M_1 through M_4 are the number of in-memory blocks and D_1 through D_4 are the number of disk blocks, $M_i \leq D_i$. Since P_1 and P_2 are on-disk, M_1 and M_2 will be 0. The in-memory quotient of the entire table is $(M_3+ M_4)/(D_1+ D_2+ D_3+ D_4)$, which is a value less than 1.

Consider three possible queries on this table where partition pruning takes place:

- Only partitions P_1 and P_2 must be scanned. The optimizer will compute the aggregate in-memory quotient as 0 since these two partitions are disk-only. In other words, this query will be satisfied purely from a disk scan with no benefit from in-memory.
- Only partitions P_3 and P_4 must be scanned. The aggregate in-memory quotient will be computed as $(M_3+ M_4)/(D_3+ D_4)$. This value will be 1 if these two partitions are entirely in-memory (M_3 and M_4 are equal to D_3 and D_4, respectively) which means that the table scan will be entirely from in-memory with no disk access.
- Partitions P_2 and P_3 must be scanned. The aggregate in-memory quotient will be $M_3/(D_2+ D_3)$ which is a value less than 1. This matches the fact that the scan of P_2 will be from disk and that of P_3 will be from in-memory.

As the above example shows, the aggregation of in-memory statistics after partition pruning allows the optimizer to accurately estimate scan costs that reflect the true cost. Using global in-memory statistics is not appropriate. This is especially important because partitions are often highly skewed with some containing far more data than others.

When executing queries on Oracle RAC, each instance compiles the query separately. The instance where the query is submitted compiles and generates the initial execution plan followed by each of the other instances doing the same. Oracle's parallel query framework requires each instance to reproduce the same execution plan.

On RAC, each instance has its own in-memory area. Suppose a query references an in-memory table. If this table is defined as DUPLICATE ALL, it will be populated in the in-memory areas of each instance. This means that the in-memory statistics of the table are the same on all the instances. Thus the optimizer on each instance will estimate the same cost for the table scan and generate the same plan for the query.

Now suppose the in-memory table is defined as DISTRIBUTE AUTO. The table will be distributed in the in-memory areas of each instance so that no instance has the entire table in its in-memory area. The in-memory statistics on each instance will reflect this, with an in-memory quotient that will be less than 1. A naïve optimizer would assume that some disk scans would be required. However, the Oracle query execution engine allocates parallel processes on each instance in such a way that the entire table scan is satisfied purely with in-memory scans from the instances where the data is populated in the in-memory area.

If a query includes distributed in-memory tables, the optimizer computes the *effective* in-memory statistics. These statistics are computed by treating the multiple RAC nodes as a single node and in-memory areas of all the nodes as one single store. For example, the effective number of in-memory blocks is the sum of the in-memory blocks in each node.

3.2 Cost Model

A query optimizer takes into account object statistics, system statistics, and database settings when evaluating alternative

execution plans. The optimizer is usually comprised of various components. The estimator component computes predicate selectivities (which help determine the resulting cardinalities of tables, joins, and aggregations) and estimates the costs of various database operations including access paths, join methods, aggregations, communication between parallel processes, and many more. A cost-based query transformation component [1] works in conjunction with the estimator to enumerate and compare semantically equivalent forms of the query.

The cost model of an optimizer includes I/O, CPU, and network communication costs. Scans of on-disk row major tables have different I/O and CPU costs than in-memory columnar tables. In Oracle DBIM, we have enhanced the cost model to include new cost formulas for in-memory tables. The in-memory-aware optimizer supports queries with any combination of on-disk row major tables and fully or partially populated in-memory columnar tables. The awareness extends to RAC where the tables may be duplicated or distributed in the in-memory column stores of different instances. Below we describe the enhancements and some of the new cost components in the cost model.

Storage index pruning cost: The optimizer estimates how many IMCUs must be scanned after the in-memory storage index prunes non-matching IMCUs. This is computed by applying the table filter predicates on the minimum and maximum values of the corresponding column CU. Consider the predicate c1 < 10. Suppose the minimum value of one of the column's CU is 15. Then we can safely prune away that CU for scanning since none of its rows will satisfy the predicate. The optimizer determines this for every CU of the column. If there are predicates on multiple columns, say c1 < 10 AND c2 > 5, the pruning is computed for each column. Because these predicates are AND-ed, the optimizer can prune an IMCU if any single predicate prunes its column's CU.

Because storage index pruning requires scanning every IMCU header, the optimizer includes this in its cost for the table scan. There are several other operations that are performed during the scan. These operations are described below. Note that the costs for these operations are included only for the IMCUs remaining after storage index pruning. In the corner case where all IMCUs are pruned, none of the following costs will apply.

Decompression cost: At run-time, column CUs must be decompressed to retrieve the corresponding values. This must be done for all referenced columns for the table. The decompression cost of a CU is a function of the compression method used for that CU. Different CUs, even for the same column, may be compressed differently.

Predicate evaluation cost: Predicate evaluation on in-memory tables takes place during the scan. For example, the predicate c1 < 10 may be evaluated on encoded column values while the CUs for c1 are being scanned. In addition, the evaluation can be done on multiple rows using vector operations on a SIMD system. If there are multiple AND-ed predicates like c1 < 10 AND c2 > 5, any IMCU that is eliminated because of an earlier predicate is skipped for subsequent predicates.

Row stitching cost: This includes stitching all projected columns into rows for the subsequent operation, like join, in the plan.

Transaction journal scan cost: The in-memory transaction journal contains rows that were inserted or updated by DML statements but that have not yet been populated in IMCUs. These rows are in row major format and must be scanned in addition to the data in the IMCUs.

All the above costs apply to the scan of the in-memory portion of a table. Consider a table with D disk blocks and N rows that is partially populated in-memory. The in-memory statistics will reflect the partial population status: the in-memory quotient will be q (< 1) and number of in-memory blocks will be Dq (< B). A scan of this table will require reading both the in-memory and on-disk portions. The in-memory scan cost is computed as described above using the in-memory statistics. The on-disk scan cost is computed using the standard cost model for row major tables but with prorated statistics of D(1-q) blocks and N(1-q) rows (this cost will include both I/O and CPU). These costs are then combined to get the total table scan cost.

For partitioned tables, the optimizer uses aggregated in-memory statistics as described in Section 3.1. This ensures that partition pruning is correctly accounted for in the cost model.

3.3 Predicate Pushdown

Oracle DBIM can evaluate many types of single table filter predicates on columnar formats directly using a bytecode interpreter when scanning CUs. Such evaluations filter out rows early on and avoid the cost of stitching columns of these rows. Also, storage index pruning can make use of in-list and range predicates to prune CUs before the scan takes place.

During query compilation, the optimizer figures out which predicates in the query can be evaluated on compressed formats, splits out these predicates, and pushes them down the query plan to the in-memory scan. Predicates which cannot be evaluated during the in-memory scan are still evaluated in the SQL Execution engine.

The optimizer also generates implied predicates based on the user-specified predicates in the WHERE clause and pushes them down for storage index pruning.

3.4 Join Processing

Oracle supports three join methods: nested loops (where the access is driven by an index), hash join, and sort merge. Each of these join methods is implemented for different join types: inner, outer, semi and anti joins. The latter two are a result of un-nesting EXISTS and NOT EXISTS sub-queries respectively. Oracle also supports several parallelization techniques for processing joins in a multi-core and multi-node hardware configuration. These techniques take into account the attributes of the tables processed by the join (such as the static partitioning of the tables, size of the tables), the join method, the join type, the number of processes assigned to the join, the hardware configuration (number of nodes in a cluster), etc. These techniques have now been enhanced to also take into account how the tables are populated in-memory.

3.4.1 Parallel Join Processing

The most effective way to speed up join processing is to use a larger number of processes. The join is executed in parallel by dividing the joined tables' rows into chunks and sending one or more chunks to the processes performing the join. Dividing the rows is performed by the set of processes (*producers*) producing the rows, which, in the simplest case, scan a table from disk. The producers send chunks to the set of processes (*consumers*) performing the join. The work of dividing and sending the tables' rows (called *data redistribution*) adds to the overall time to complete the join processing. So it is important to pick the most efficient way to redistribute data. Oracle supports several

redistribution techniques from which the optimizer decides based on the table sizes, the join method, the join type, the number of processes performing the join, and the hardware configuration [6]. One such technique, referred to as *partition-wise join* [24], applies when joining two tables on their partitioning key. The benefit of this technique is the absence of any data exchange between the producers and consumers. For example, consider the following query which returns the revenue for all orders in year 2014:

```
SELECT sum(l_extendedprice*(1-l_discount)) as rev
FROM lineitem, orders
WHERE l_orderkey = o_orderkey
  AND o_orderdate between '01-01-2014' and
                          '12-31-2014'
```

Lineitem and *orders* are partitioned using hash on *l_orderkey* and *o_orderkey* columns, respectively. Since the join key is the same as the partitioning key, the parallel processing of the join can be implemented using partition-wise join where each process joins one partition from table *lineitem* to the corresponding partition in *orders*, i.e., the process scans both partitions and processes the join. Assuming P partitions and S processes, each process will end up performing the scan and join P/S times. Tables can also be sub-partitioned and partition-wise join can happen when the join key is the sub-partitioning key. The partition-wise dimension is referred to as either PARTITION or SUB-PARTITION depending on whether the join is on the partitioning or sub-partitioning key.

Figure 2 illustrates the parallel partition-wise join between *lineitem* and *orders* for the above example query using 4 processes. *Lineitem* is partitioned by hash on column *l_orderkey* and sub-partitioned by hash on *l_custkey*. *Orders* is partitioned by hash on *o_orderkey*. Each process (P_i) scans one partition of *lineitem* (comprised of sub-partitions L_{i1}, L_{i2}, L_{i3}, and L_{i4}) and one partition of orders (O_i).

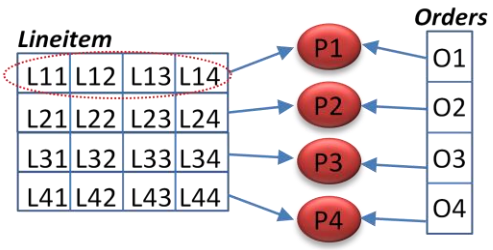

Figure 2: Parallel Partition-wise Join

When joining partitioned in-memory tables in a RAC system, the query optimizer must account for how the rows are populated in-memory. Partitioned tables that are distributed can be populated using three options: row-id range, partition, or sub-partition. The first option divides the table into row-id ranges and populates every range into the in-memory area of one node of a RAC system. The second (resp. third) option populates one partition (resp. sub-partition) into the in-memory area of one node of a RAC system. The populate dimension is referred to as either PARTITION or SUB-PARTITION depending on whether the population is done using the partition or sub-partition option. When the joined tables are populated along a dimension that does not match the partition-wise dimension, then the scan operation will have to read data from disk. Only when both the population and partition-wise dimensions are the same does the scan read the data entirely from in-memory.

Using the same example, assume *lineitem* is hash partitioned on *l_orderkey* column (4 partitions) and hash sub-partitioned on *l_custkey* column (4 sub-partitions) while *orders* is hash partitioned on *o_orderkey* column (4 partitions). Figure 3 illustrates the case when *lineitem* is populated on the sub-partition dimension. When process P_3 scans the third partition of *lineitem* (L_{31} … L_{34}), it will read from in-memory only ¼ of the rows (L_{33} colored in green) with the rest read from disk (L_{31}, L_{32}, L_{34} colored in red). Figure 4 illustrates the case when *lineitem* is populated on the partition dimension. A process P_i scanning the i^{th} partition of *lineitem* (L_{i1} … L_{i4}) will find all its rows in-memory and will perform no disk reads. The optimizer cost model for partition-wise join has been enhanced to be aware of the populate dimension. So for the same query on the same tables, the optimizer may not use partition-wise join when the tables are read from in-memory while it would use partition-wise join when the tables are read from disk. The decision depends on the tradeoff between reading data from memory and doing inter-process communication (not using partition-wise join) vs. reading the data from disk and not doing inter-process communication (using partition-wise join).

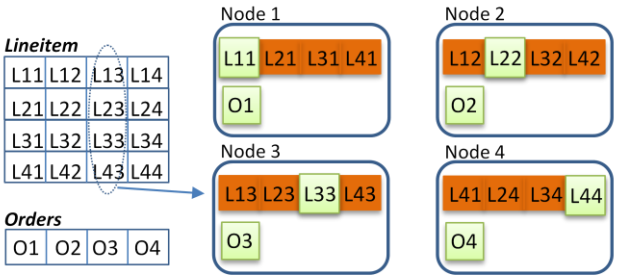

Figure 3: Lineitem Populated on Sub-partition Dimension

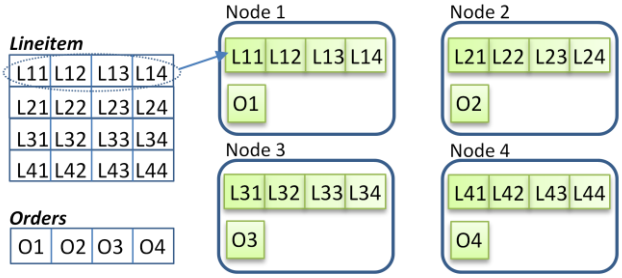

Figure 4: Lineitem Populated on Partition Dimension

3.4.2 Join Filter

As explained earlier in Section 3.3, filter predicates are used by the in-memory scan to discard CUs based on storage index pruning. A join operation also throws away rows from the joined tables that do not satisfy the join condition. The in-memory scan can be made more efficient if the effect of join predicate filtering is pushed down to the scan. This reduces the number of rows that the scan has to construct (stitch), rows that would be just thrown away by the join operation anyway. This is accomplished in Oracle using a *join filter*, similar to the bloom filter concept described in [2]. A join filter is created from the join key values of the tables with which the in-memory table is joined using equality comparison, and pushed down along with the other table filters. Join filters are optimized for in-memory by evaluating them only once per distinct value of the join key instead of once per row. Using join filters reduces the throw away factor in joins and reduces the row stitching cost in the in-memory scan. Join filters provide extremely high performance improvement for certain classes of queries such as star queries where dimension tables

have filter predicates that reduce the rows returned from the scan of the fact table by several orders of magnitude. In the example query in Section 3.4.1, while scanning table *orders*, a join filter is built based on the join key *o_orderkey* and combined with the table filters evaluated during the scan of *lineitem*. This reduces the number of rows constructed during the in-memory scan of *lineitem*.

3.4.3 Partition Join Filter

A partition join filter is similar to a join filter but is used to prune partitions instead of CUs. A partition join filter is generated for a table if it is joined on its partitioning key. The join filter is used to eliminate partitions during the scan and provide a level of pruning above the CU. This optimization is used for disk-based scans as well. In the example query given in Section 3.4.1, *lineitem* is joined on its partitioning key, so a partition join filter is generated when scanning *orders* and used to prune partitions during the scan of *lineitem*.

3.5 Hybrid Data Access

Oracle DBIM allows a table to be partially populated in-memory so a scan operation on such a table must be able to process both on-disk database blocks as well as in-memory CUs. Since the cost and time of reading on-disk data is very different than reading in-memory data, it is possible that an index may perform better than a full table scan for on-disk data but not for in-memory data. In other words, the best way to access data from a partially populated in-memory table may be a hybrid access path: index access path for on-disk data and an in-memory scan for in-memory data. The same principle applies when joining to a partially in-memory table: a nested loops join can be used for the on-disk data and a hash join can be used for the in-memory data.

Hybrid execution plans is an optimization technique introduced in Oracle Database 11.2 for queries involving partially indexed tables. For example, using the Oracle partitioning feature, a table can be partitioned into physically independent partitions and every partition can be indexed independently. Partial indexing is commonly used for tables where most partitions are static, e.g., a date-partitioned table with no data changes for partitions older than a week. Creating indexes on the static partitions results in zero overhead from DML on the table since DML will only affect the non-indexed partitions. Access to the non-indexed partitions will always use sequential scan while access to the indexed partitions can use either sequential scan or index scan. Queries that read data from both indexed and non-indexed partitions may use a hybrid execution plan. The fact that a table is partially indexed leads not only to a different access path for that table but also affects other optimizer decisions, including the join method, the join order, and query transformations (if they are cost-based).

Similar to partially indexed tables, any partition of a partitioned table can be independently populated in-memory. Indeed, a partially populated in-memory table may also be partially indexed. The optimizer has been enhanced to generate hybrid execution plans when the query accesses a mix of on-disk and in-memory table partitions. This is made possible by the changes to the cost model described in Section 3.2 which in turn uses the partition-specific in-memory statistics described in Section 3.1.

The hybrid plan optimization for in-memory tables has been implemented as a new cost-based query transformation which rewrites a query into a semantically equivalent form where the table is replaced by a UNION-ALL view with two branches: one branch represents the access to the in-memory data only and the other branch represents the access to the on-disk data only. Each branch has filter conditions to restrict access to the relevant partitions. Other operations in the query (e.g., join) can be pushed into the UNION-ALL view. A cost-based decision determines which operations to push into the view and therefore includes all factors that are taken into account in the cost model.

For example, consider the following query which returns the average revenue per zip code in outlet stores for year 2014:

```
SELECT stores.zipcode, avg(sales.revenue)
 FROM sales, stores
 WHERE sales.store_id = stores.id
   AND stores.type = 'Outlet'
   AND sales.sales_date between '01-01-2014'
                            AND '12-31-2014'
 GROUP BY stores.zipcode
```

Table SALES has indexes on some of the columns commonly used to join to dimension tables. One such index is on column `store_id`. The table is partitioned monthly on the `sales_date` column, i.e., every partition contains data for a single month of a single year. With most queries accessing data from a subset of the partitions, the table definition is changed to make the October to December partitions in-memory. Since the query accesses data from all of year 2014, the query optimizer has an opportunity to use a different execution plan for the January-September partitions than for the October-December partitions. An example of such an execution plan is shown below. The optimizer generated a hybrid execution plan with the first branch using a nested loops join to fetch rows from SALES using an index on the join key, and the second using a hash join to fetch rows from SALES using an in-memory scan.

```
-----------------------------------------------------
| ID | Operation                      | Name       |
-----------------------------------------------------
|   0| SELECT STATEMENT               |            |
|   1|  HASH GROUP BY                 |            |
|   2|   VIEW                         | VW_TE_5    |
|   3|    UNION-ALL                   |            |
|*  4|     TABLE ACCESS BY INDEX ROWID| SALES      |
|   5|      NESTED LOOPS              |            |
|*  6|       TABLE ACCESS INMEMORY FULL| STORES     |
|   7|       PARTITION RANGE AND      |            |
|*  8|        INDEX RANGE SCAN        | S_STORE_ID |
|*  9|     HASH JOIN                  |            |
|* 10|      TABLE ACCESS INMEMORY FULL| STORES     |
|  11|      PARTITION RANGE AND       |            |
|* 12|       TABLE ACCESS INMEMORY FULL| SALES      |
-----------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter(TIME_ID <  '09-01-2014' AND
              TIME_ID >= '01-01-2014')
   6 - inmemory(TYPE='Outlet')
   8 - access(SALES.STORE_ID = STORES.ID)
   9 - access(SALES.STORE_ID = STORES.ID)
  10 - inmemory(TYPE='Outlet')
  12 - inmemory(TIME_ID >= '09-01-2014' AND
                TIME_ID <  '12-31-2014')
```

3.6 Parallel Execution

Parallelism is an effective way to improve the performance of a SQL statement. During the parallel execution of the statement, every SQL operation in the execution plan is divided into tasks, each assigned to a different process (*slave process*). For example, the scan operation is parallelized by assigning a granule [6] of the scanned table (sequence of database blocks) to a different process. Data produced by a set of processes executing one operation are distributed to the set of processes executing the next operation in

the plan. The number of processes assigned to a SQL operation is called the *degree of parallelism* (DOP).

Oracle supports two modes of parallelization: manual and automatic. In the manual mode, the user specifies the DOP using one of three options: (1) table property, (2) session configuration, (3) hint. The order of precedence is hint, then session, then table property. The automatic mode (*Auto DOP*) was introduced in Oracle 11.2 to make it easier to deploy parallel execution in database applications. At a high level, Auto DOP is a two step process. In the first step (called *serial optimization pass*), the query optimizer estimates the query execution time in serial mode and the optimal DOP for every operation in the execution plan. The estimated time is compared to a time threshold: if the estimated time is less than the threshold, then the query is executed in serial mode, otherwise it goes through the second step. In the second step (called *parallel optimization pass*), the optimizer derives the maximum DOP from all the operations in the execution plan (computed in the first step) and re-optimizes the statement using that DOP. If the ratio of serial execution plan cost to the parallel execution plan cost is greater than *DOP * scalability-factor*, then the optimizer generates a parallel plan using the computed DOP, otherwise it generates a serial execution plan. The scalability factor (a value between 0 and 1) is used to account for the overhead of using parallel execution (creating and assigning processes, various forms of communication, etc.).

Auto DOP applies to statements using on-disk tables or in-memory tables. However, it works differently when applied to in-memory tables as follows:

- The DOP derived for the in-memory scan accounts for the same factors (described in Section 3.2) that the optimizer cost model does: pruning of CUs, no disk reads, rate of processing, etc.
- For a partially in-memory table, using the in-memory quotient, a portion of the scan DOP is computed based on the disk scan and another portion based on the in-memory scan.

In a RAC environment, an in-memory table can be distributed across multiple nodes. The optimizer ensures that the final DOP is adjusted to a multiple of the number of nodes. For example, if the computed DOP is 6 for a statement on tables distributed on 4 nodes, then the final DOP is 8 (4*2).

4. EXPERIMENTAL RESULTS

We evaluated the query optimizer using experiments. The goal of these experiments was two-fold:

- Consider two identical queries, one using on-disk tables and the other in-memory tables. Did the query with in-memory tables perform better than that with the on-disk tables?
- Consider a query with in-memory tables. If we disabled all the in-memory-related enhancements to the optimizer, did the query perform worse than with the enhancements enabled?

In short, the goal was to verify whether the enhanced optimizer generated optimal plans for queries with in-memory tables.

The experiments were performed on a system using Intel Xeon with 16 CPU cores and 252 GB DRAM running Linux and Oracle Database 12c Enterprise Edition. For the RAC experiments, we used a 4-node cluster where each node had this same configuration (for a total of 64 CPU cores and 1 TB DRAM).

In all the experiments that follow, except for the ones in Section 4.3, we cached all on-disk tables and indexes in the buffer cache. This ensured a fair comparison between on-disk tables and in-memory tables: there was no physical I/O for any query and the performance differences were attributable solely to differences in execution plans and data formats (row major in buffer cache vs. columnar in-memory). This is an important point to keep in mind when reviewing the results.

In the experiments in Section 4.3, the on-disk tables were not cached in the buffer cache because the experiments were specifically intended to verify and measure the amount of disk I/O incurred.

4.1 Single Table Query

For this experiment, we created an on-disk table T_100_DISK with 100M rows with 100 columns named C0 through C99. Then we created an in-memory table T_100_IM with the same structure and content as T_100_DISK. T_100_IM was fully populated in-memory. For both tables, we created B-tree indexes on the column C48 which was a unique column with values ranging from 1 to 100M.

We used a single node for this experiment and ran the following query in serial:

```
SELECT <projected columns>
FROM <T_100_DISK | T_100_IM>
WHERE C48 < :bind
```

We ran several experiments by using all the combinations possible by varying the following:

- Use T_100_DISK or T_100_IM.
- Use 12 different values for :bind to obtain predicate selectivities of 0.00001%, 0.0001%, 0.001%, 0.01%, 0.1%, 1%, 5%, 10%, 25%, 50%, 75%, and 100%.
- Use 12 sets of projected columns: 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100.
- Use optimizer hint to force an index access path (IDX) or full table scan (FTS).

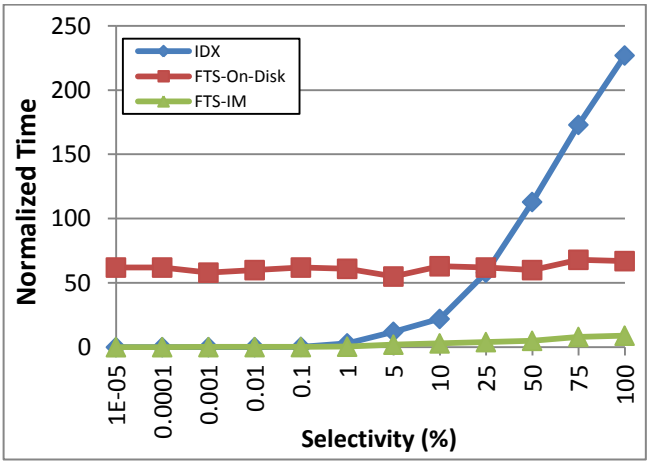

Figure 5: Single Table Query with 5 Projected Columns

Figure 5 shows the performance of the IDX, on-disk full FTS, and in-memory FTS plans for 5 projected columns at various selectivities. As expected, the index plan is good at very low selectivities but then rapidly degrades at higher selectivities. On-disk FTS is worse than in-memory FTS at all selectivities, also as expected. What is also remarkable is that at lower selectivities, in-memory FTS is competitive with the index; only at extremely low selectivities is the index better than in-memory FTS (although this is not visible in the graph). This shows that certain indexes, especially those used for analytic queries, can be dropped for in-memory tables without loss of performance. On the other hand, indexes used for OLTP queries, which often do single-value lookups, may out-perform in-memory table scans and so cannot be dropped.

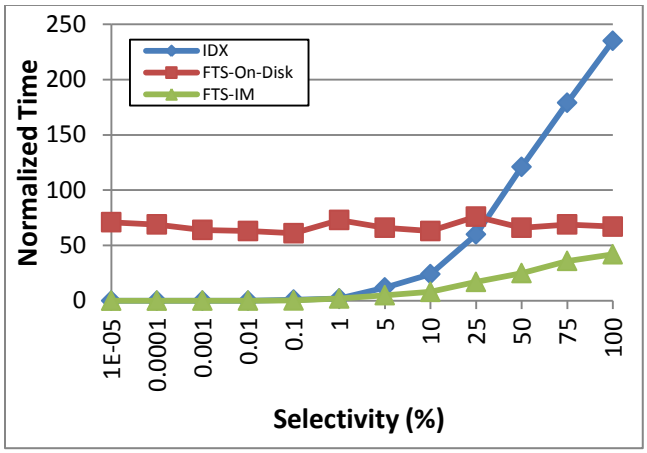

Figure 6: Single Table Query with 20 Projected Columns

Figure 6 shows the same three plans with 20 projected columns. Here too IM FTS is competitive with the index at low selectivities and outperforms on-disk FTS at all selectivities. Comparing the two figures, note the slightly worse performance of in-memory FTS at high selectivities with more projected columns. This is due to the increased cost in decompressing the additional CUs and stitching them together into rows.

It is well known that an index is a good choice when there are low-selectivity predicates that can be used as keys. As the selectivity increases, the index performs worse and a full table scan becomes more attractive. The *inflection point* is the selectivity at which the index performs as well as a full table; the index is better below the inflection point, and the table scan is better above. The *estimated inflection point* is based on the optimizer's estimated cost of the access paths at various selectivities. The *actual inflection point* is based on the run-time performance of the two access paths. The estimated and actual inflection points are generally close for any good optimizer but not necessarily the same because of variations in runtime state and inherent optimizer uncertainties. But one of the more fundamental tests of the optimizer is whether the estimated and actual inflection points for a given query follow a similar trend when varying some aspect of the query, like the number of projected columns. This metric is important because it indicates the optimizer's ability to adapt to various changes in the query or table formats and correctly choose the optimal plan.

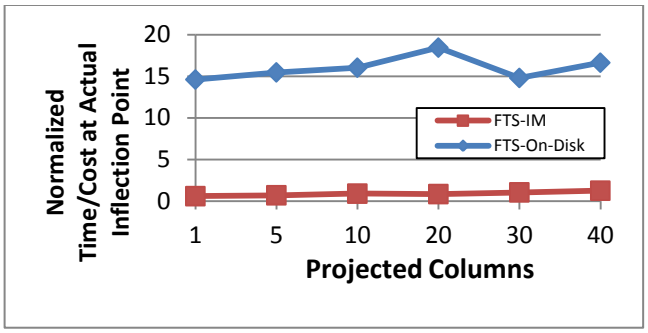

Figure 7: Time/Cost for Single Table Query

Figure 7 shows the normalized time to cost ratio at various projected columns for both on-disk and in-memory FTS. The key observation here is that both lines are more or less flat indicating that the time increases proportionally with the optimizer cost which indicates that the estimated cost is very accurate.

Another interesting question is what would happen if an in-memory table was partially populated in-memory. This scenario can arise in a real system either because of memory constraints or because of the DBA consciously enabling only some partitions (perhaps the most frequently accessed ones) for in-memory.

We created a table T_100_HPART_IM that was identical to T_100_IM except that it was hash partitioned on the column C48 with 16 partitions. Then we ran the following query in serial:

```
SELECT <projected columns>
FROM T_100_HPART_IM
WHERE C49 < :bind
```

We conducted several experiments by using all the combinations possible by varying the following:

- Use 2 different values for :bind to obtain predicate selectivities of 1% and 10%.
- Use 2 sets of projected columns, 5 and 20.
- Start with all 16 partitions in memory (in-memory quotient of 1). Alter one partition at a time to be non-in-memory until all partitions are non-in-memory making the table on-disk only. Assuming the partitions are of equal sizes (the reason why we chose to partition by hash), this will result in 17 different in-memory quotients ranging from 0 to 1 in increments of 1/16.

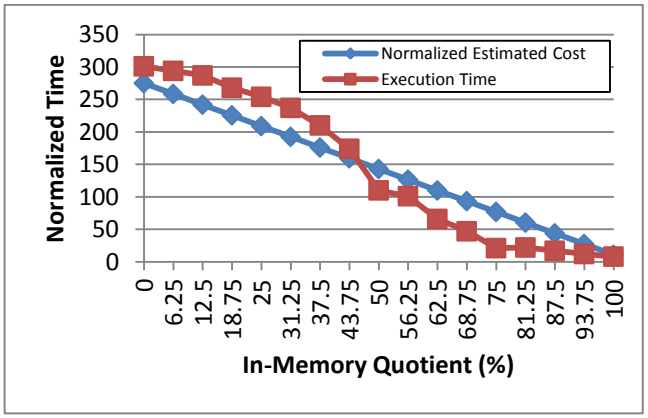

Figure 8: Cost and Time with Varying In-Memory Quotient

If a table is partially in-memory, then some I/O is necessary (in our experiments, this was logical I/O rather than physical disk I/O since we ensured that all table blocks were in the buffer cache). Since buffer cache reads are more expensive than in-memory, we expect that as the in-memory quotient increases from 0 (on-disk) to 1 (fully in-memory), the table scan should become faster. Moreover, the optimizer cost should also decrease in the same manner. Figure 8 (10% selectivity and 20 projected columns) shows that this is indeed true. Using current in-memory statistics that reflect the state of the table, the cost model accounts for the mixed dual format scans accurately enough that the cost closely tracks the elapsed time.

4.2 Join Query

For the join experiment, we used the same tables T_100_DISK and T_100_IM described in Section 4.1. The experiments were run on a single node in serial. The query was:

```
SELECT <projected columns>
FROM T LT, T RT
WHERE LT.C0 < :bind
  AND LT.C49 = RT.C48
```

For simplicity, we used the same table for the left and right sides. This has no effect on the result. The following criteria were varied:

- Use T_100_DISK or T_100_IM for T.
- Use 12 different values for :bind to obtain predicate selectivities for the left table of 0.00001%, 0.0001%, 0.001%, 0.01%, 0.1%, 1%, 5%, 10%, 25%, 50%, 75%, and 100%.
- Use 4 sets of projected columns: 1, 5, 10, and 20, using a mix of columns from both the left and right tables.
- Use optimizer hint to force an index nested loops join or hash join.

Note that we used C48 as the join column on the right side. This was because both T_100_DISK and T_100_IM are indexed on this column and so the index could be used with a nested loops join. For hash join, the access path on the right side was a full table scan. The access path of the left side table was fixed at full table in-memory scan for all experiments.

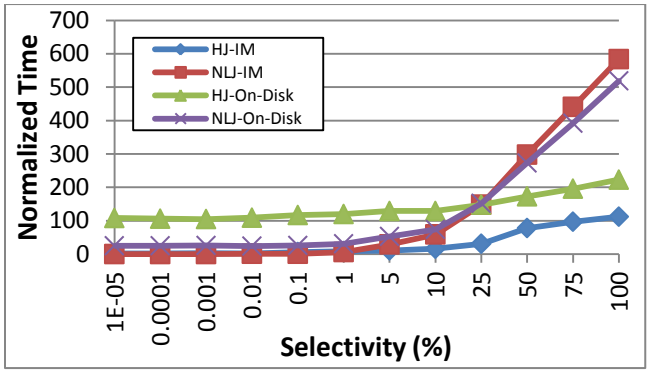

Figure 9: Join Query with 5 Projected Columns

Nested loops join is optimal when the left table is small; larger tables mean more index lookups on the right side which is sub-optimal. Thus, there is an inflection point when nested loops join becomes worse than hash join. For both on-disk and in-memory joins, this inflection point is clearly seen in Figure 9 (with 5 projected columns) and Figure 10 (with 20 projected columns). As expected, the in-memory hash join takes longer with increasing selectivities and projected columns

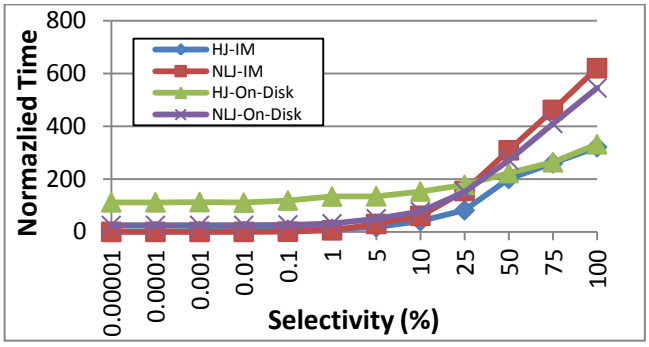

Figure 10: Join Query with 20 Projected Columns

The nested loops results in Figure 9 deserve a closer examination. The right side of the plans in both NLJ-IM and NLJ-On-Disk use the same index access path. However, NLJ-IM uses an in-memory scan for the left side whereas NLJ-On-Disk has a full table scan from the buffer cache. At high selectivities, an in-memory scan is worse than a buffer cache scan because of the extra row stitching costs which explains why NLJ-IM becomes worse than NLJ-On-Disk.

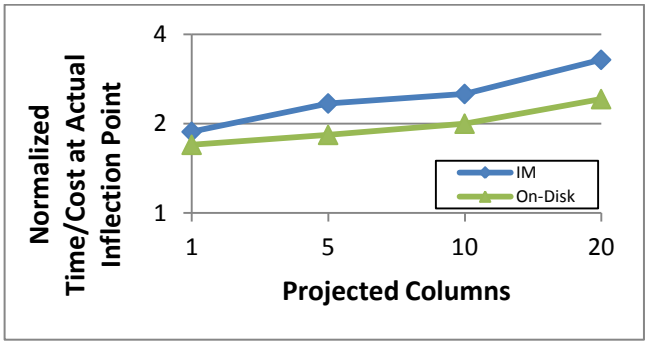

Figure 11: Time/Cost for Join Query

Figure 11 shows the normalized time to cost ratio for the join query at various projected columns for both on-disk and in-memory FTS. Both lines are mostly flat indicating that the time increases proportionally with the optimizer cost. The cost is reasonably accurate but the margin of error is slightly higher than for the single table query.

4.3 Parallel Join Processing

The purpose of this experiment was to demonstrate the importance of making the optimizer aware of the populate dimension on RAC when the join key is the same as the partition key of one or both tables in the join, as explained in Section 3.4.1. We created a partitioned version of the 100M rows table T_100_DISK described in Section 4.1. Table T_100_HH_IM was partitioned by hash on column C48 into 16 partitions and sub-partitioned by hash on column C49 into 16 sub-partitions. The table was populated in-memory on the partition dimension which allows an in-memory full partition-wise join if joined on C48.

We constructed a self join query and forced a hash join executed by 4 processes running on a 4-node RAC system, i.e., each process was assigned to a different node. We varied the following:

- Join key. We generated 4 join conditions using different permutations of C48 and C49 for probe and build tables.
- Number of projected columns. We used 5 and 20 items with equal proportion from both build and probe tables.
- Number of rows used in the build table of the hash join. We ran with 1M rows and 10M rows using a filter condition on the build table.

In all, we ran 8 versions of the following query (B refers to the build table and P to the probe table):

```
SELECT <5 columns | 20 columns>
FROM T_100_HH_IM B, T_100_HH_IM P
WHERE B.C0 < {1000000 | 10000000}
  AND B.<C48 | C49> = P.<C48 | C49>
```

In addition to the optimizer's default plan, we also repeated the execution by forcing a different plan to compare the quality of the query optimizer decision.

We found that the optimizer used partition-wise join only when the partition-wise join dimension matches the distribution dimension, i.e., when the tables are joined on C48 regardless of the number of projected columns and the number of rows in the build table. That plan results in no disk reads and no inter-process communication: every process reads the table rows from the local node's in-memory area. This is shown as the bar labeled "Full PWJ on Distribution Dim" in Figure 12 and Figure 13.

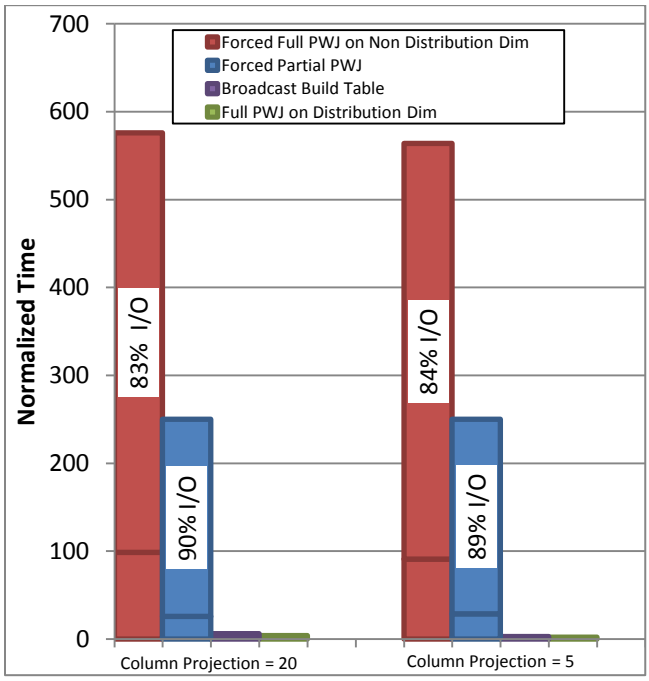

Figure 12: Impact of Distribution Dim. on PWJ (sel=1%)

When the join uses C49 for the build table and C48 for the probe table, then the query optimizer chose to broadcast the build table instead which resulted in inter-process communication. This is the purple bar labeled "Broadcast Build Table" in Figure 12 and Figure 13. The alternatives, partition-wise join and partial partition-wise join, performed much worse. These are shown respectively as the bars labeled "Forced Full PWJ on Non Distribution Dim" and "Forced Partial PWJ" in Figure 12 and Figure 13. The optimizer also chose to broadcast the build table when the join uses C49 for both the build and probe table, and when the join uses C48 for the build table and C49 for the probe table.

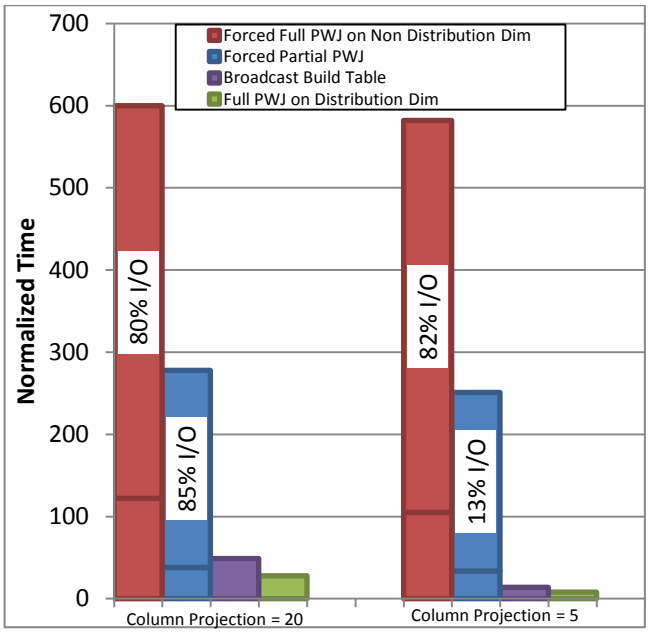

Figure 13: Impact of Distribution Dim. on PWJ (sel=10%)

4.4 Customer Workload

For this experiment, we used a workload from one of Oracle's customers. The database consisted of 31 tables and 35 indexes with almost 3 billion rows in the largest table. 3 of the tables were partitioned. 10 tables were composite-partitioned (partitioned at two levels) and the rest were non-partitioned. The total on-disk size of the database was approximately 1 TB. When all the tables were fully populated in-memory, the total memory footprint was 156 GB, a compression ratio of approximately 7.

The workload had 32 decision support queries, none of which were tuned in any way: there were no hints or parameters to constrain or "help" the optimizer. We performed three different experiments on this workload:

1. **On-disk**. All tables were on-disk only. This run served as our baseline.
2. **IM-**. All tables were in-memory. In-memory awareness was disabled for the optimizer but not for the SQL execution engine. In other words, all plans were the same as **On-disk** above, but all full table scans in the plans were in-memory scans.
3. **IM+**. All tables were in-memory. All optimizer enhancements were enabled to choose the best plans for the workload.

We ran these 3 experiments twice: first on a single node and then again on a 4-node RAC. On RAC, we declared the tables DISTRIBUTE AUTO which meant that Oracle automatically

distributed the tables into the in-memory areas of each of the 4 nodes. All experiments were done using Auto DOP.

The goal, as in earlier experiments, was to verify whether the optimizer enhancements worked as designed, and generated plans optimized for in-memory. There are two comparisons that are interesting:

- Compare On-disk with IM-. This compares the performance of the same execution plan when the only change is making all tables in-memory.
- Compare IM- with IM+. Between these two runs, the execution plans may be different and the goal is to verify whether the optimizer, enhanced for in-memory tables, picks plans that are optimal for in-memory.

Figure 14 shows the cumulative time for the workload in each of the three experiments in both single node and RAC environments. IM+ was significantly better than IM-: 65% on the single instance and 66% on RAC. This is very encouraging because it shows that our enhancements worked as designed: the optimizer chose the plans and DOP that were in-memory-aware and cluster-aware. Another point worth noting is that the On-Disk and IM- performance was extremely close. This is because this workload had queries where often an index was the best choice for certain tables. The table scans that were in the plans became in-memory scans in IM- which accounts for the slight improvement over On-Disk.

The results of this experiment validate one of the key claims in this paper. To get significant performance improvements, it is not enough to just take on-disk row major tables and make them in-memory columnar tables. The optimizer must also be made in-memory-aware so that it can correctly cost and explore alternative plans before choosing the best one.

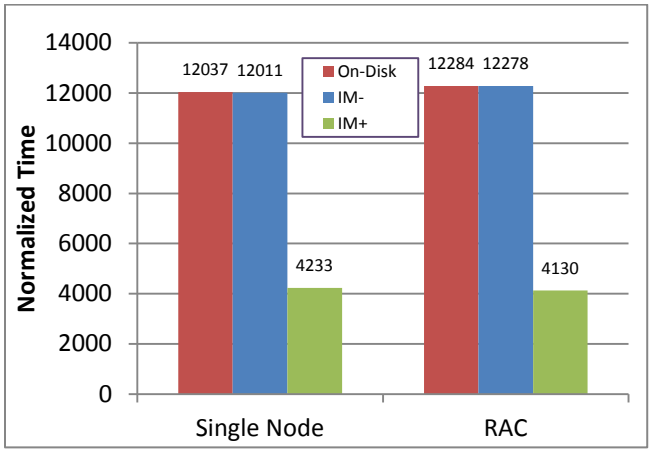

Figure 14: Cumulative Time for Customer Workload

5. RELATED WORK

In recent years, there have been a number of main-memory-only databases optimized for OLTP performances, including both research prototypes such as HYRISE [9], H-store [11], HyPer [12] as well as commercial systems such as solidDB [17], VoltDB [23], SQL Server Hekaton [7] and Oracle TimesTen [14]. These systems usually require the entire database to fit in memory. While this speeds up transaction processing, it is sub-optimal for analytic queries which scan a vast number of rows while projecting only a few columns. In contrast, Oracle DBIM allows the table to be partially populated in-memory depending on the workload requirements, and the data could be present in both row major format (in the buffer cache) and columnar formats. Oracle DBIM provides all the features of a relational database, including full ACID properties, with complete application transparency.

The idea of column major tables dates all the way back to the 1980s. Sybase IQ was the first commercialized column-major storage product, and has been around since 1994. In the mid 2000s, the MonetDB [3] and C-Store [22] research prototypes revived interest in column-oriented databases for analytic workloads. The major contribution of these research prototypes was to directly evaluate queries against the columnar formats. A commercialized version of C-store eventually evolved into Vertica [15], now marketed by HP. One feature that distinguishes Vertica is the ability to define *projections*, which, like indexes in row stores, contain copies of the base data that are in domain order, rather than the default order in which rows are initially loaded. While projections can provide improved performance for queries requiring ordering, they consume additional storage and complicate both database design and the re-assembly of rows during query execution. MonetDB also spawned a commercialized version called Vectorwise [25], which stores each column separately as a vector.

Since the 2010s, major business intelligence and database vendors have also begun to integrate columnar storage into their products. SAP HANA [8] is a columnar in-memory database which supports both OLTP and BI workloads. Microsoft provides column store as an additional index through the SQL Server Column Storage Index [16] and some new query operators that take advantage of these indexes. IBM's DB2 BLU [21] acceleration has a query engine that operates directly on compressed data format for scans, joins and aggregations.

Most of the main memory databases optimized for OLTP workloads have limited enhancements in their query optimizers for columnar tables. For example, Hekaton has no significant changes to SQL Server's optimizer, and H-store uses a simple optimizer which is based on the communication costs across the network.

Query optimizations for columnar databases are mostly focused on leveraging the columnar-specific structures and execution operators. For example, MonetDB has a simple optimization to restrict the join order selection based on the physical properties of the join columns. Similarly, C-store's optimizer is mostly used to pick the projection structure created for column groups. This optimizer is minimalist in that the projections it reaches first are chosen for the query and the join order of the projections are completely random. Vertica's original optimizer was targeted for star schemas, and every query had to be first converted to a star schema. It has since acquired capabilities found in traditional optimizers, including a physical cost model and column histograms. IBM BLU's optimizer generates different types of execution plans, called *evaluator chains*, specifically for columnar compressed tables. The evaluators work only on single table queries, and join queries are restructured into a list of single table queries by the optimizer.

6. CONCLUSION

Oracle Database 12c introduces a revolutionary dual-format in-memory columnar technology. It provides substantial performance benefits with very minimal user effort and no application changes. Oracle DBIM works with all workloads, including OLTP and data warehouse workloads.

In this paper, we described the changes made to Oracle's leading-edge query optimizer to make it in-memory-aware. These changes enable the optimizer to choose plans that are optimal for the specific configuration and system state when a query is executed. This includes queries executed on a single node and on RAC, and queries that involve on-disk and in-memory tables where the latter may be fully or partially populated in-memory.

Using multiple workloads, including a large customer workload, we showed that the optimizer picks the best plans and degree of parallelism for queries with in-memory tables. We also showed that if the optimizer enhancements are disabled, the performance drops to the level of on-disk tables. These experiments confirm that the optimizer enhancements are effective and essential. Without these enhancements, the optimizer may choose sub-optimal plans which can negate the expected benefits of an in-memory columnar database.

The query optimizer plays an important role in realizing the full potential of fast query performance in Oracle Database In-Memory.

7. REFERENCES

[1] Ahmed, R. et al. Cost-Based Query Transformation in Oracle. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 1026-1036, 2006

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

[3] Boncz, P. et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 479-490, 2006

[4] Bridge, W., Joshi, A., Keihl, M., Lahiri, T., Loaiza, J. The Oracle Universal Server Buffer Manager. *Proceedings of the 23rd VLDB Conference*, pp. 590-594, 1997

[5] Chakkappen, S. et al. Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g. In *Proceedings of the 2008 ACM SIGMOD Conference on Management of Data*, pp. 1053-1064, 2008

[6] Cruanes T., Dageville, B., Ghosh, B. Parallel SQL Execution in Oracle 10g. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 850-854, 2004

[7] Diaconu, C. et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1243-1254, 2013

[8] Färber, F. et al. The SAP HANA Database—An Architecture Overview. *IEEE Data Eng. Bull.* 35(1), pp. 28-33, 2012

[9] Grund, M. et al. HYRISE: A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, 4(2), pp. 105-116, 2010

[10] Jakobsson, H., Zait, M., Dageville, B. Method and Mechanism for Partition Pruning. *U.S. Patent No. 6,965,891*, 2005

[11] Kallman R. et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2), pp. 1496-1499, 2008

[12] Kemper, A., Neumann T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pp. 195-206, 2011

[13] Lahiri, T. et al. Oracle Database In-Memory: A Dual Format In-Memory Database. *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering*, pp. 1253-1258, 2015

[14] Lahiri, T., Neimat, M., Folkman, S. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36(2), pp. 6-13, 2013

[15] Lamb, A. et al. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12), pp 1790-1801, 2012

[16] Larson, P., Hanson, E., Price, S. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1): pp. 15-20, 2012

[17] Lindstrom J., Raatikka, V., Ruuth, J., Soini, P., Vakkila, K. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Eng. Bull.* 36(2), pp. 14-20, 2013

[18] Oracle Database In-Memory. *Oracle White Paper*, Oracle 2014

[19] Oracle Database In-Memory: In-Memory Aggregation. *Oracle White Paper*, Oracle, 2015

[20] Oracle Real Application Clusters (RAC). *Oracle White Paper*, Oracle, 2013

[21] Raman, V. et al. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proceedings of the VLDB Endowment*, 6(11), pp. 1080-1091, 2013

[22] Stonebraker, M. et al. C-Store: A Column-oriented DBMS. *Proceedings of the the 31st International Conference on Very Large Data Bases (VLDB)*, pp. 553-564, 2005

[23] Stonebraker, M., Weisberg, A. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36(2), pp. 21-27, 2013

[24] Zait, M., Dageville, B. Parallel Partition-wise Joins. *U.S. Patent No. 6,609,131*, 2003

[25] Zukowski, M., Boncz, P. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 35(1): 21-27, 2012