Adaptive and Big Data Scale Parallel Execution in Oracle

Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang*, Thierry Cruanes*
Oracle USA
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
*while at Oracle
First.Last@oracle.com

ABSTRACT

This paper showcases some of the newly introduced parallel execution methods in Oracle RDBMS. These methods provide highly scalable and adaptive evaluation for the most commonly used SQL operations – joins, group-by, rollup/cube, grouping sets, and window functions. The novelty of these techniques is their use of multi-stage parallelization models, accommodation of optimizer mistakes, and the runtime parallelization and data distribution decisions. These parallel plans adapt based on the statistics gathered on the real data at query execution time. We realized enormous performance gains from these adaptive parallelization techniques. The paper also discusses our approach to parallelize queries with operations that are inherently serial. We believe all these techniques will make their way into big data analytics and other massively parallel database systems.

1. INTRODUCTION

Parallel execution is the crux of commercial relational database systems, database appliances and Hadoop systems in processing huge volumes of data. While relational database systems and appliances parallelize execution of SQL statements, Hadoop systems parallelize computations specified as map/reduce jobs, and the line is getting blurred by the day. For example, Hive provides SQL interface to the data sitting in a HDFS; Polybase claims to move the data from HDFS to the SQL engine, or the computation (map/reduce job) to Hadoop. Whatever the approach may be, analyzing the vast amounts of data being collected by companies nowadays calls for massively scalable and adaptive parallel execution models. The parallelization models should fully leverage CPU resources, minimize data transmission overheads, and adapt based on the characteristics of the data.

We targeted the heavily used SQL operators for data analysis – joins, aggregations (group-by, rollup, cube, grouping sets), and analytic window functions, and developed scalable parallel execution models for them. These operators play a predominant role in customer workloads, TPC-H and TPC-DS [5] benchmarks, and are commonplace in data mining and graph processing using RDBMSs. The paper is organized like this – in the rest of this section, we briefly introduce SQL analytics and parallel execution in Oracle; Sections 2 and 3 present our adaptive techniques for scaling the computation of group-by and its variants. Techniques for massively scaling analytic window functions are presented in Section 4. In Sections 5 and 6, we describe adaptive distribution methods and massive parallelization of joins. Section 7 shows ways to parallelize queries with operations that are inherently serial. Performance results are presented in Section 8. We discuss related work in Section 9 and conclude in Section 10.

Proceedings of the VLDB Endowment, Vol. 6, No. 11

1.1 SQL Analytics

Data cube [14] computation is an expensive and critical operation in the Data Warehouse environments. To facilitate efficient execution, SQL Group-By clause was extended with ROLLUP, CUBE and GROUPING SETS [4] allowing one to specify aggregations at different levels in a single query block. ROLLUP aggregates (or *rolls up*) data at successively higher levels – ROLLUP (*year, quarter, month*) computes aggregations at (*year, quarter, month*), (*year, quarter*), (*year*), and (<*grand-total*>) levels. CUBE, on the other hand, aggregates the data on all level combinations – CUBE (*region, year*) aggregates on (*region, year*), (*region*), (*year*) and (<*grand-total*>) levels. GROUPING SETS syntax allows users to aggregate the data on arbitrary levels.

These operations attracted research [7][15] by the database community in late 90's and execution schemes were proposed. Commercial database systems have been supporting these operations since then. Oracle RDBMS uses sort-based execution scheme for ROLLUP. Data is sorted and aggregated on all the group-by keys, and higher aggregation levels are computed as the data is read in sorted order – for example, (*year, quarter*), (*year*), and (<*grand-total*>) levels are computed as aggregated data at (*year, quarter, month*) level is read in sorted order. CUBE and Grouping Sets are evaluated by reducing them into one or more ROLLUPs.

Window functions, a part of SQL 2003 [3] standards, enriched SQL with analytic capabilities and have been widely adopted by the user community. Analytic queries expressed with window functions are not only elegant in expression, but execute very well as numerous self-joins and multiple query blocks are avoided. Oracle RDBMS supported window functions since Oracle 8*i*. In the simplest form, the syntax of window functions looks like this:

```
Window_Function ( [ arguments ] ) OVER (
    [ PARTITION BY pk1 [ , pk2, ... ] ]
  [ ORDER BY ok1 [ , ok2, ... ]  [ WINDOW  clause ] ] )
```

Window functions are evaluated on a per partition basis – data is partitioned using the PARTITION BY keys and rows within each partition are ordered on the ORDER BY keys. Then for each row,

the WINDOW clause establishes the *window* (the start and the end boundary) on which a SQL aggregate function like *sum*, *count*, or a ranking function like *rank*, *row_number*, or a reference function like *nth_value*, *lag* is applied to produce the result.

Window functions are evaluated after joins, group-by, and the HAVING clause of the query block. There can be multiple window functions in a query block, each with different partition-by, order-by, and WINDOW specifications. Oracle RDBMS employs sort-based execution of window functions, and this appears as "*window sort*" operator in the query tree. Our query optimizer tries to minimize the number of *window sort* operations needed to evaluate the set of window functions specified in the query. One or more passes over the sorted data will be needed to evaluate the window functions within a *window sort* operator. Three subclasses of window functions – *reporting*, *cumulative*, and *ranking*, are quite ubiquitous in users' queries. They are also employed by database query optimizers [2][9] to remove self-joins and multiple query blocks. Our novel techniques massively parallelize these important classes of window functions.

1.2 Parallel Execution in Oracle

Parallel execution in Oracle RDBMS is based on *producer-consumer* model in which one set of parallel processes produces the data, while the other set consumes it. Producer and the consumer sets have same number of processes and this number forms the "*degree of parallelism*" (DOP) of the statement. There is a *Query Coordinator* (QC) process overseeing the parallel execution of the statement. The QC mostly does logistical work – compilation of the SQL statement, distribution of the work among parallel processes, shipping the results back to the user, and is overloaded to perform computations that are not parallelizable.

The QC groups various operations (table scan, joins, aggregations etc) performed in a sequence into a logical entity called the "*Data Flow Operation*" (DFO). It schedules the execution of the DFOs by the two sets of parallel processes. Data redistribution takes place between producers and consumers, and this happens via shared-memory in the case of single-instance database, or via a network for multi-instance Oracle Real Application Clusters database. The data distribution can be broadcast, hash or range on keys, or random and is determined based on the characteristics of the consuming DFO. Typically, the data distribution type and keys are determined during the compilation of the parallel statement. Once all the producer processes finish executing the DFO and distributing data to consumers, role reversal occurs – QC assigns the next DFO in sequence to the erstwhile producer processes, which now become the consumers; the previously consuming parallel processes now start producing data. In this paper, we show DFOs using ovals and the arrows between DFOs depict the data redistribution.

1.3 Terminology

The following acronyms are frequently used in this paper:

GBY – Group By	DOP – Degree of Parallelism
OBY – Order By	DFO – Data Flow Operation
PBY – Partition By	GPD – GBY Pushdown
QC – Query Coordinator	HBF – Hybrid Batch Flushing
NDV – Number of Distinct Values	

2. GROUP-BY

Group-by Pushdown (GPD) is a well-known technique for GBY parallelization, wherein the data is aggregated before being distributed on the GBY keys for final aggregation. GPD plan (shown in Figure 1) scales well, reduces data distribution costs, and handles skew. However, it comes with the added CPU and IO costs of performing aggregation one extra time (that is at the data production side) and can have an inferior performance when the reduction in the data due to aggregation is low. Due to this benefit vs. risk trade off, Oracle RDBMS has been using a cost-based approach and some heuristics for GBY parallelization.

Oracle query optimizer uses estimates of the number of rows and groups in choosing the GPD plan. As IO overheads are typically much higher, Oracle uses a heuristic of not spilling to disk at the producer side. When a producer process performing GBY runs out of memory, it switches itself off, outputs whatever data it has aggregated thus far, and becomes a *pass-through* operation.

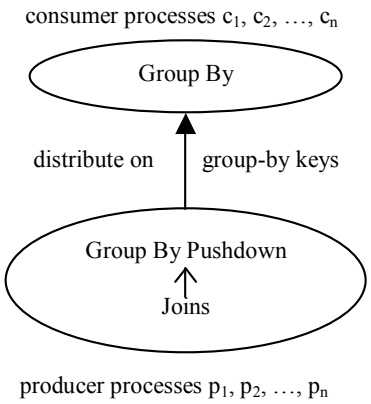


Figure 1. Parallel GroupBy Pushdown

There were several problems with our existing approach – optimizer dependency, untimely shut-off of GPD, and the inability to handle skew. Due to inaccurate optimizer estimates, we weren't choosing GPD plan in cases where it would have been ideal. Our heuristic of stopping aggregation upon running out of memory worked well for systems with limited memory and concurrency support. But for newer generation systems that can support heavy query bursts and can operate almost in-memory, our heuristic needed to be revisited. Because of high memory availability, we might not hit the *memory-full* criterion and would continue to do GPD even though it is detrimental. In the other extreme, memory pressure due to sudden query burst can lead to early shut-off of GPD even though GPD is beneficial. In the former case, we waste CPU resources and in the latter, we incur a lot of row distribution overhead. Skew exacerbates the problem further due to low effective degree of parallelism and the load imbalances.

We made several changes to our GBY parallelization scheme to fix the above-mentioned issues, making it massively scalable and adaptable based on data characteristics. It also incurs low CPU overhead. In the new parallelization model, GPD plan is picked irrespective of optimizer estimates, thereby eliminating the catastrophic scenario (i.e., limited scalability in the presence of skew). We then made GPD operation adaptive (or become *pass-through*) based on the statistics (number of input records and number of groups) gathered at execution time, rather than on the "*memory-full*" heuristic. And unlike the current scheme, we

continue pushdown aggregation if it is found to be effective (1/3rd, or by a configurable ratio) in reducing the data. When memory capacity is reached, two possibilities exist for GBY to free up memory – spill the data to disk, or send it over the network to the consuming parallel processes (c_1, …, c_n of Figure 1) that complete the GBY processing. The former approach, though spills to disk, handles skew and transmits the least amount of data over the network. The latter approach, which we call "*Batch Flushing*" (BF), flushes the batch of rows aggregated thus far, rebuilds a new batch, and repeats the process. BF avoids spilling to disk at the expense of transmitting more rows over the network. It handles skew well provided enough memory is available for aggregation.

To handle high throughput scenarios, we employ a strategy that is a hybrid of the two described above, and we call it "*Hybrid Batch Flushing*" (HBF). In HBF, instead of flushing the batch right away, we use it to aggregate the incoming records. We probe an incoming record against the batch and if we find a match, we aggregate; otherwise we send the record over the network. While doing this, we maintain the efficiency of the batch. When we find that the current batch is not effective in aggregating the data, we flush it to the network, rebuild a new batch and repeat the process. Additionally, we may choose to keep the most frequent records in the batch. HBF betters BF in reducing network traffic when memory is limited due to heavy workload. HBF adapts well to the memory fluctuations – especially, when memory usage by GBY has to be brought down due to a sudden query burst. In such cases, HBF keeps the most-frequent groups in memory and flushes the rest to network.

Another aspect to consider is the timing of the "*adaptive*" decision. Making the decision at the time of spill, as we do today, will not be performant for queries with low or no reduction, and in-memory aggregation. In this case, we effectively consume twice the CPU for the GBY (hash or sort based) operation. If the decision is made too early, we might make a bad decision due to insufficient sample. Our approach is to make the decision when data exceeds L2 cache size, or a multiple of it. If we find that the reduction is good, we try to give more memory to the GBY operation. If not, we enter HBF mode. Having a batch that is L2 resident will not add too much performance penalty, especially when the expensive hash value computation (for group-by hash) is shared across several operations. If we don't encounter an effective batch even after trying several times, we enter "*pass-through*" mode that transmits rows as they are received. As presented in Section 8.1, our HBF strategy gave significant performance gains over the current GBY parallelization.

3. ROLLUP

Rollup operation aggregates the data at successively higher levels and is most commonly applied along a dimensional hierarchy. Users typically rollup the data on one dimension, keeping other dimensions at a particular level – e.g. query Q1 rolls up on *time* dimension, keeping *geography* dimension at *country* level; it computes SUM(sales) for (*c, y,q, m*), (*c, y, q*), (*c, y*) and (*c*) levels.

Currently, Oracle RDBMS parallelizes such queries in two ways, but both the approaches have pitfalls. The first approach, as shown in Figure 2, parallelizes by distributing the data on non-rollup keys (*country* for Q1). This scheme works well when the number of distinct values (NDV) of the non-rollup keys is more than the degree of parallelism (DOP), and there is no/little skew in the distribution keys. Otherwise, performance will be very poor due CPU underutilization.

```
Q1  SELECT country AS c, year AS y,
      quarter AS q, month AS m, SUM(sales)
    FROM time_dim t, geog_dim g, fact f
    WHERE t.time_key = f.time_key AND
          g.geog_key = f.geog_key
    GROUP BY country,
             ROLLUP(year, quarter, month);
```

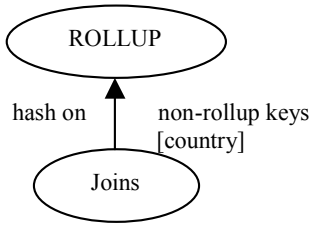


Figure 2. Single Stage (Non-Pushdown) Parallel Plan

The second approach tries to mitigate the scalability and skew issues by computing ROLLUP in two stages, as shown in Figure 3. In the "ROLLUP" stage, each parallel process aggregates the local data and rolls it up. Rows produced by this stage are at multiple levels and are distinguished with a grouping identifier (*grouping_id* function). They are distributed (hash or range) to the second stage on GBY keys and the *grouping_id*. The second stage performs vanilla group-by operation based on all the GBY keys and the *grouping_id*.

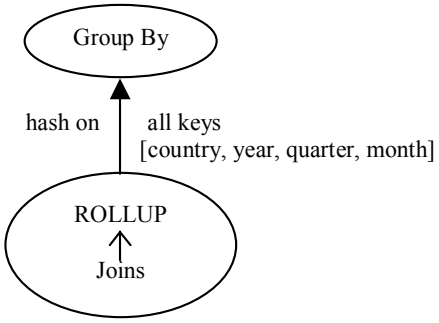


Figure 3. Rollup Pushdown Parallel Plan

This parallel execution plan shows better scalability and performance – skew in the data is handled by the "ROLLUP" stage via aggregation; and enough keys are used to distribute work evenly among parallel processes. However, it falls short when data is sparse, or limited memory is available for the rollup stage. Sparse data [6] is not so uncommon in the real world and as ROLLUP is done before the data distribution, huge explosion in the data traffic can happen. For example, peak holiday sales occur during different months (and say, quarters) in different countries. So the ROLLUP of query Q1 would produce one row at (*year, quarter, month*) level and one at (*year, quarter*) level for every base level (year, quarter, *month, country*) row. When there is not enough memory for the ROLLUP operation, we either have to stop aggregation (like the adaptive group-by of Section 2), or spill the data to disk. The former is like the "*sparse data*" case in that it explodes the data – 4 rows produced for each input row to the query Q2. The latter case is less performant as it might spill the data to disk twice – once each in ROLLUP and GBY stages.

Our new ROLLUP parallelization scheme, shown in Figure 4, overcomes the scalability issues of the above-described parallelization models. It employs two stages namely "*Rollup Distributor*" (RD) and "*Rollup Collector*" (RC) and is adaptive. It makes several decisions at query execution time based on the statistics gathered on the real data. It decides the data distribution keys, whether to aggregate at the base level or now, and the aggregation levels to be computed in the RD vs. the RC during the query execution. Thus, it is immune to optimizer errors and scales massively.

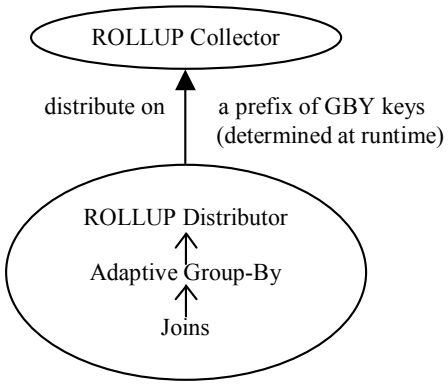
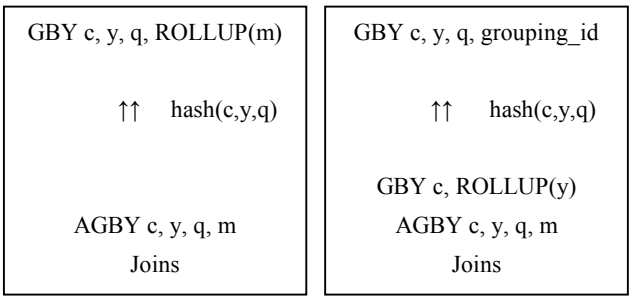


Figure 4. Adaptive Rollup Parallelization

To save space, we use the first letter of the column (*c* for country, *y* for year, *q* for quarter, and *m* for month) in the following description. Adaptive group-by operation (as described in Section 2) aggregates data at the base level and if it finds that there is no reduction due to group-by, it becomes a *pass-through* operation. RD performs several actions in the following sequence. Example query Q1 specific information is given in square brackets:

1. Buffers incoming rows and collects the NDV of potential distribution keys [(*c,y,q,m*), (*c,y,q*), (*c,y*), and (*c*)].
2. Upon finding a candidate distribution key with NDV >> DOP, it informs the Query Coordinator (QC) of its decision [say (*c, y, q*) is the candidate distribution key]. Note that this is a local decision made by a parallel process.
3. QC informs RD of the global choice for the distribution key [say (*c,y,q*)]. QC makes this decision based on the local (and potentially different) choices made by various RD processes. The distribution key implicitly determines the levels RD would be computing.
4. RD now knows the ROLLUP levels it has to compute [(*c,y*) and (*c*)]. It first processes rows from the buffer – each row would be inserted into an access structure (sort/hash) for aggregation [on (*c,y*) level], and would also distributed to a corresponding RC process based on the global distribution key [(*c,y,q*)]. Rows distributed to the RC are tagged with a bit indicating that these are "*base*" rows.
5. RD then reads the remaining rows from input source and processes them like in step 4.
6. Once input is exhausted, RD would roll the data up [level (*c*)] and distributes it to the RC. Rows distributed in this step do not have the "*base*" row tag.

Parallel processes performing the RC step get informed by the QC of the ROLLUP levels they are supposed to compute [(*c,y,q,m*) and (*c,y,q*)]. RC aggregates data on those levels for "*base*" rows. Rows not marked as "*base*" are vanilla aggregated on GBY keys and the *grouping_id* [(*c, y, q, m*), *grouping_id(y, q, m)*]. In the example scenario, non-base rows at level (*c,y*) have NULL value for *q* and *m* columns, and NULLs for *y*, *q*, and *m* for rows at (*c*) level. Grouping_Id is needed to distinguish these NULLs generated by the rollup operation from the NULLs in the real data. Essentially, our new plan is an amalgamation of two independent (pushdown and non-pushdown) plans shown below:



The new ROLLUP parallelization model has yielded significant (up to 25x) performance gains as it leveraged all the CPU resources. As CUBE and GROUPING SETS are evaluated by decomposition into a series of rollups [7], they too become massively scalable.

4. WINDOW FUNCTIONS

In this section, we describe our novel adaptive parallel execution models for the three most popular classes of window functions – *reporting*, *ranking* and *cumulative* window functions. Reporting window functions (also called *reporting aggregates*) report the partition-level aggregate value for each row in the partition. This is so because the window for each row spans the entire partition. They are often used for comparative analysis – e.g. compare each day's (base rows) sale to the yearly sales obtained using "sum(sales) over (partition by year)".

Cumulative window functions, as suggested by their name, produce "*year-to-date*" style aggregations. Unlike reporting aggregates that have a fixed window (that is an entire partition) for each row, cumulative functions have a window that grows, albeit in one direction. For each row in a partition, the window extends from the first row in the partition to the current row, and an aggregate is applied on the window. For example, a window function "max(price) over (partition by stock, year order by date rows between unbounded preceding and current row)" computes the year-to-date maximum price of a stock.

Ranking function is a special case of cumulative function in that the aggregate applied on the window is not a typical SQL aggregate (like *sum*, *count*), but a ranking function – *row_number*, *rank*, *dense_rank* etc. For example, "rank() over (partition by department order by sales desc)" is useful in finding top sales in each department. For convenience, we use short cuts PBY and OBY for "PARTITION BY" and "ORDER BY" respectively.

4.1 Reporting Window Functions

Reporting window functions are used in computing the aggregates at successively higher hierarchical levels for comparative analysis.

Consider this common data warehouse query for example:

```
Q2  SELECT /*y:year q:quarter m:month d:day*/
      y, q, m, d, sales,
      SUM(sales) OVER (PBY y,q,m) msales,
      SUM(sales) OVER (PBY y,q) qsales,
      SUM(sales) OVER (PBY y) ysales
    FROM fact f;
```

As mentioned in Section 1.1, Oracle RDBMS uses sort-based execution of window functions and evaluates the three reporting aggregates in Q2 using a single "*window sort*" operation. Because of this clumping, the parallel plan requires data redistribution (hash or range) on the common PBY key. Figure 5 shows the parallel plan for query Q2, wherein the data distribution is done by "*hash*" on the common PBY key "*year*".

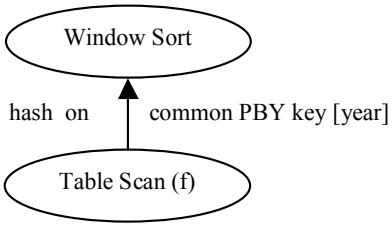


Figure 5. Parallelization on Common PBY Keys

This parallel execution plan works well when the number of partitions created by the common PBY key is equal to or greater than the system supported degree of parallelism (DOP). Otherwise, it fails to leverage system resources fully and executes very poorly. The scaling of this parallel plan depends on the number of distinct values (NDV) of the PBY keys. For example, if "*year*" had only 10 distinct values, the plan cannot use more than 10 parallel processes. Thus, this model is ill suited for big data systems and database appliances that have lots of processing power. So to make the window function computation massively parallelizable, we use extended data redistribution keys, or employ window pushdown. Both these techniques split the window function computation into two stages.

With "*extended distribution keys*", we use more PBY keys than the common PBY key of the window functions involved so that the NDV of the distribution keys is equal to or greater than the desired DOP. With "*window pushdown*" approach, we push the window computation down to the underlying data flow operation (DFO). In both approaches, a second stage is needed to consolidate the local results from various parallel processes. A new operator called "*Window Consolidator*", Figures 6 and 7, performs this consolidation step.

The decision of extending data distribution keys or using window pushdown is based on the optimizer NDV estimation of the PBY keys during query compilation. When none of the PBY key combination has sufficient NDV, window pushdown will be chosen. Otherwise, we will pick a set of PBY keys that has sufficient NDV as the distribution keys. For example, Figure 6 shows "*extended distribution keys*" plan for query Q2, assuming that PBY key combination *(year*, *quarter)* has sufficient NDV for a balanced data distribution and scalability. Figure 7 shows the window function pushdown plan for the same query Q2.

Unlike the traditional parallel plan (Figure 5), window function computation is not completed in the "*window sort*" operation. A parallel process performing "*window sort*" only sees a portion of

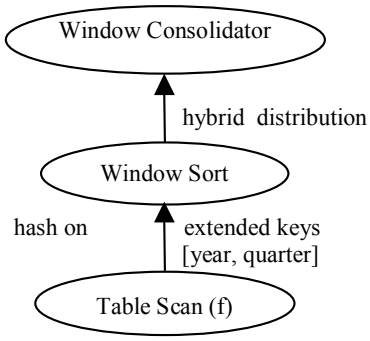


Figure 6. Parallelization on Extended Keys

the data belonging to a partition and needs to know the results from peer parallel processes. In case of "*extended distribution keys*", window functions at a coarser granularity than the distribution keys picked are incomplete. For example, the yearly sales reporting aggregate of query Q2 will not be completed in the "*window sort*" operation of Figure 6. The "*window consolidator*" operation finishes computation of such window functions. For the "*window pushdown*" plan, none of the window functions will be completed in the window sort stage as parallel processes would be working on an arbitrary set of rows. For notational convenience, we use the term "*to-be-consolidated*" window functions to refer to the window functions whose results, as computed in the window sort operation, are not final. We now describe the consolidation phase and then show how we handle optimizer misestimates of the NDV.

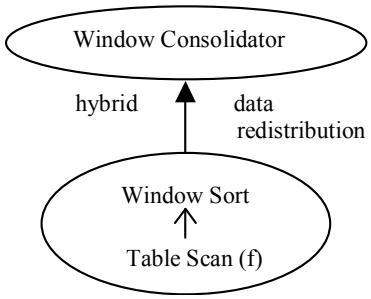


Figure 7. Parallelization with Window Pushdown

4.1.1 Window Consolidation

Parallel processes that perform "*window sort*" would first broadcast the local results of the "*to-be-consolidated*" window functions to the parallel processes performing the "*window consolidator*". Note that this broadcasted data is expected to be small, as these "*to-be-consolidated*" window functions must have low NDV estimates on their PBY keys. Otherwise, their PBY keys would have been chosen to be the extended data re-distribution keys. After they are done with broadcasting, the window sort processes would randomly distribute the actual data and the results of the window functions "completely" processed in that stage. Observe this "*hybrid*" distribution method between window sort and window consolidator in Figures 6 and 7.

At the consuming DFO, each window consolidator process aggregates the partial results it has received via the broadcast, and builds a hash table on the PBY key values. Then the rows that are received via random distribution are probed in the hash table to get the fully aggregated values for the "*to-be-consolidated*"

window functions. Rows are marked with a special bit indicating whether they are "broadcast" or "randomly" distributed. For lack of space, we skipped some finer details such as synchronization among parallel processes and buffering of rows. Note that the window consolidator has an extra *O(n)* lookup operation and in case of "*extended distribution keys*", there is an extra data distribution cost. These additional costs are insignificant and worth spending, considering the massive scalability achieved.

4.1.2 Example

Consider the query Q2 executed on a system with a desired DOP of 20, with the optimizer estimates NDV(*y*)=5, NDV(*y,q*)=20, and NDV(*y,q,m*)=60. Since NDV(*y,q*) is equal to the desired DOP, we pick (*y,q*) for data distribution. This plan is shown in Figure 6 and the "*to-be-consolidated*" window function in this case would be yearly sales "SUM(sales) OVER (PBY y)". There will be 20 parallel processes executing the window sort operation – they will sort input data on (*y, q, m*), compute and broadcast yearly sales to the next set of parallel processes performing window consolidation, and randomly distribute input data along with completed window functions (quarterly and monthly sales) to consolidator processes. For simpler illustration, we describe the example using only two parallel processes.

Table 1. Sample Data for Reporting Aggregates

Parallel Process 1 [WINDOW SORT]					Parallel Process 2 [WINDOW SORT]				
Y	**Q**	**M**	**D**	**S**	**Y**	**Q**	**M**	**D**	**S**
2001	Q1	Jan	1	10	2001	Q1	Jan	3	20
2001	Q1	Feb	8	20	2001	Q2	Apr	20	100
2001	Q2	Apr	15	10	2001	Q2	May	25	35
2001	Q2	Jun	5	8	2001	Q3	Jul	30	8
2001	Q3	Jul	3	2	2001	Q4	Nov	8	20
2001	Q3	Aug	6	20	2001	Q4	Dec	9	20
2001	Q3	Sep	1	5	2002	Q1	Jan	5	50
2001	Q4	Nov	10	10	2002	Q1	Mar	20	30
2002	Q1	Mar	25	30	2002	Q2	Apr	5	30
2002	Q2	Apr	15	20	2002	Q2	Jun	10	25
2002	Q2	May	20	15	2002	Q3	Jul	30	25
2002	Q3	Aug	18	45	2002	Q3	Sep	5	35
2002	Q3	Sep	25	35	2002	Q4	Nov	15	25
2002	Q4	Nov	18	20	2002	Q4	Nov	18	10
2002	Q4	Dec	25	100	2002	Q4	Dec	25	200

This figure shows the sample data received and sorted by the two parallel processes performing the window sort. Here, each parallel process sees two partitions for the "*year*" column. For parallel process 1, partially aggregated yearly sales values for partitions "2001" and "2002" are 85 and 265 respectively. Corresponding yearly sales values for parallel process 2 are 203 and 430. This data gets broadcast to the window consolidator processes.

The window consolidator processes first aggregate the results of "*to-be-consolidated*" window functions. In the example, the fully aggregated value for partition "2001" is 288 and the fully aggregated value for partition "2002" is 695. These values are kept in a hash table based on the PBY key "*year*". Next, the window consolidators probe the incoming regular (non-broadcasted) rows in the hash table to get the final yearly sales.

4.1.3 Handling Optimizer Errors

As stated earlier, the NDV of PBY keys is calculated by the query optimizer and is used in picking the parallelization model. Inaccurate statistics pose a serious performance problem. When NDV is underestimated, we will end up broadcasting too many "*to-be-consolidated*" rows to the window consolidator stage and perform poorly. This may even introduce performance regression compared to the traditional plan of Figure 5. In order to accommodate optimizer's NDV misestimates, we made the new parallel execution plans adapt at query execution time. In particular, the window sort operations compute/monitor the NDV of PBY keys of the "*to-be-consolidated*" window functions. As soon as they discover that the NDV is higher than anticipated, they inform the query coordinator (QC) of their finding. QC then adapts the plan to the traditional parallel plan (Figure 5) by setting the distribution keys to be the common PBY keys. It asks the window sort processes to become *pass-through* operation, and informs window consolidators to perform the entire window function computation. In response, the window sort processes would distribute whatever rows they have processed/buffered thus far and become *pass-through* operations. The consolidators would act as "*window sort"* and compute all the window functions.

As a future work, we want to make window function parallelization completely immune to optimizer errors (over and under estimations). We propose an "*always window pushdown*" parallel execution plan that would be similar to scalable and adaptive rollup computation of Section 3. In particular, we will monitor the NDV of PBY keys and pick the set of PBY keys that give us good scalability and with less broadcasting overhead. Once data distribution keys are decided, window sort and window consolidator operations will be adapted accordingly. Though this plan is truly scalable and adaptable, it may end up sorting/buffering and spilling to disk twice i.e., at window sort and window consolidator. We plan to investigate further and implement this "*always window pushdown*" parallel plan.

4.2 Ranking and Cumulative Functions

In this section, we describe scalable execution models for *ranking* and *cumulative* window functions. Consider the following query computing rank and cumulative sum total. The window functions have the same PBY key, but have different OBY keys:

```
Q3 SELECT prod_id, date, sales,
     SUM(sales) OVER (PBY prod_id OBY date),
     RANK() OVER (PBY prod_id OBY sales)
   FROM fact f;
```

When parallelizing multiple window functions in the query block, Oracle query optimizer currently makes a heuristic decision of combining/clumping multiple window sort operations that have common PBY keys into the same DFO. Data is distributed on the common PBY key and each parallel process will execute the window sort operations independent of the others. As shown in Figure 8, the parallel plan for query Q3 uses distribution on the PBY key *prod_id*. The "*window sort (s)*" operation computes the cumulative total and requires ordering on (*prod_id, date*). The rank function is computed by the "*window sort (r)*" operation and requires data to be sorted on (*prod_id, sales*). The clumping

optimization that combined window sort (*s*) and window sort (*r*) in the same DFO reduces the number of data redistribution steps. It works well when the number of partitions created by PBY keys is equal to or greater than the DOP. Otherwise, this plan has severe scalability limitation.

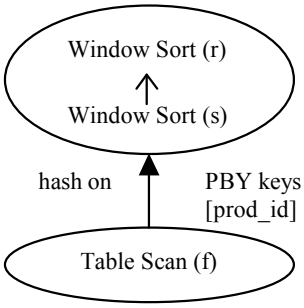


Figure 8. Clumped Plan on Common PBY Keys

Our new strategy to massively scale ranking and cumulative window functions extends data distribution keys to include some or all OBY keys, and can further include a random number such that the number of partitions is equal to or greater than the DOP. We use optimizer estimates of the NDV of PBY and OBY keys in deciding the distribution keys. If the NDV of PBY key is less than the desired DOP, we successively add OBY keys of the window function till the NDV exceeds the DOP. Even if including all OBY keys does not meet the DOP threshold, we include a random number in the distribution keys.

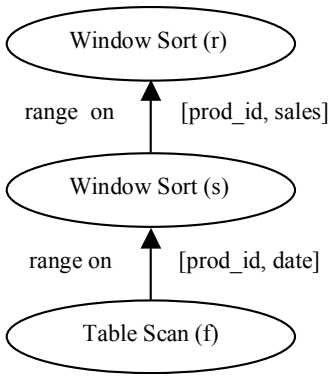


Figure 9. Range Parallelization on Extended Keys

Unlike reporting aggregates, ranking and cumulative window functions require rows to be sorted within each partition. So when additional keys are used for data distribution and multiple parallel processes do the computation of a partition, range distribution needs to be employed. For the example query Q3, assuming NDV(*prod_id, date*) and NDV(*prod_id, sales*) is greater than the DOP, we will have a parallel execution plan like Figure 9. Observe that as a result of including OBY keys for data distribution, we introduced data redistribution from window sort (*s*) to window sort (*r*).

The parallel execution model with range distribution operates in two phases interspersed with a synchronization step. Assuming that the data is range distributed among parallel processes p_1 to p_n in that sequence, a parallel process p_i needs to know about the data processed by parallel processes p_j, $j<i$ that have the same partition as the first partition of p_i. This is to correctly compute the window function results. When random number is used for distribution, process p_i may need information about processes p_k, $k>i$ that have same partition as the last partition of p_i.

In the first phase, each parallel process computes the local window function results for the first row of the first partition and the last row of the last partition. They then send these results, the corresponding PBY key values, and if required, the OBY key values to the query coordinator (QC). OBY key values are required when a random number is included in the distribution keys as rows with the same PBY and OBY key values can end up at different parallel processes, but the window computation ought to treat these rows alike and produce the same result.

Based on the type of window functions being computed, the QC consolidates the information it receives from various window sort operations. The QC then sends the relevant information to the participating parallel processes so that they can produce correct final results. This information includes offsets or replacements to be used by the respective parallel processes for computing the final results. For data partitions that get allocated/distributed in entirety to one parallel process, window computation does not need consolidation. These are the partitions that are not the first and last data partitions within a window sort.

4.2.1 Example

Table 2. Sample Data for Cumulative

Parallel Process 1 [WINDOW SORT]

prod_id	date	sales
P1	D1	10
P1	D1	20
P1	D2	10
P1	D2	20
P1	D3	15
P1	D3	25

Parallel Process 2 [WINDOW SORT]

prod_id	date	sales
P1	D4	10
P1	D5	25
P1	D5	15
P2	D1	20
P2	D2	10
P2	D2	20

Parallel Process 3 [WINDOW SORT]

prod_id	date	sales
P2	D3	10
P2	D3	20
P2	D3	10
P2	D4	20
P2	D5	15
P2	D6	25

Parallel Process 4 [WINDOW SORT]

prod_id	date	sales
P2	D7	10
P2	D8	20
P2	D9	10
P2	D9	20
P2	D10	15
P2	D12	25

Assume that query Q3 is executed with a DOP of 4 using the parallel plan of Figure 9. To save space, we only show how to compute "SUM(sales) OVER (PBY prod_id OBY date". Table 2 shows the data sorted on PBY and OBY keys by each of the four parallel processes performing the window sort.

Each parallel process computes the local cumulative sum for the first row of the first partition and the last row of the last partition. For parallel process 1, there is only one partition "P1". It produces the result "P1, 10" for the first row and "P1, 100" for the last row. Process 2 sees two partitions "P1" and "P2", and produces local results "P1, 10" and "P2, 50". Parallel process 3 sees only one partition "P2" and produces the results "P2, 10" and "P2, 100". Similarly, process 4 produces local results "P2, 10" and "P2, 100". All these local results are sent to the QC. The QC consolidates and sends the following information to each of the parallel processes.

Table 3. Consolidation Information

Parallel Process	Information from QC
Process1	no "offset"
Process2	Use 100 as "offset" for rows in first partition "P1"
Process3	Use 50 as "offset" for rows in first partition "P2"
Process4	Use 150 as "offset" for rows in first partition "P2"

The information in Table 3 is deciphered like this: there is no "offset" for process 1, so its local results are final. For process 2, the offset is 100 for partition "P1". This is because cumulative sum from process 1 for partition "P1" is 100. So process 2 adds 100 to the local results for partition "P1" to produce correct result. It also produces local results for partition "P2". The process 3 needs to offset the local result for partition "P2" by 50, the cumulative sum of partition "P2" in process 2. Likewise, process 4 offsets local results by 150 (50 from process 2 and 100 from process 3).

With the techniques described in this section, window function computation can scale massively. We got a whopping 20x improvement in our performance results (see Section 8).

5. ADAPTIVE DISTRIBUTION METHODS FOR JOINS

Parallel execution plans for an equi-join between two non-partitioned tables involve data distribution on one or both tables involved. We assume hash joins in this discussion as they are most commonly used, but the discussion is applicable to merge join as well. Consider the following query for which the optimizer picks hash join:

```
Q4 SELECT t.year, t.quarter, f.sales
    FROM time_dim t, fact f
    WHERE t.time_key = f.time_key;
```

Different data distribution methods are possible for the parallel plan of this hash join. One approach is hash-hash distribution (Figure 10) in which both tables are distributed by hash on the join key. This works well when the inputs are large. Another approach is broadcast or broadcast-random distribution, in which the smaller table is broadcast to the parallel processes performing join and the larger table is either randomly distributed or accessed in chunks. Broadcast plan, as shown in Figure 11, is ideal when the left input of the join is small. Query optimizer uses cardinality estimates and makes a cost-based decision in choosing the

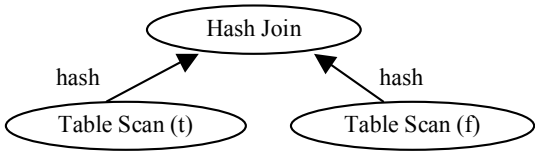


Figure 10. Hash-Hash Distribution Plan

distribution method. It is not uncommon to have the query optimizer make gross over or under estimation of the cardinality of join inputs, and this can result in severe scalability issues. If hash-hash distribution is picked due to overestimation, it can happen that only a few parallel processes would end up doing most of the join execution. With underestimation of the input, broadcast distribution would get picked and can be catastrophic due to huge data transmission and CPU overheads.

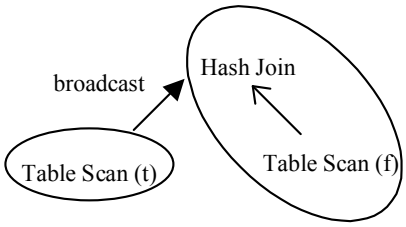


Figure 11. Broadcast Plan

With systems that can support massive parallelism becoming a necessity with big data, what is needed is a distribution method that adapts at query execution time based on the cardinality of the real data. So we have extended our hash-hash distribution to be "*adaptive*" – become "broadcast-random" when the left input is small, otherwise stay as "hash-hash". We call this "*hybrid hash-hash*" distribution method. It is shown in Figure 12.

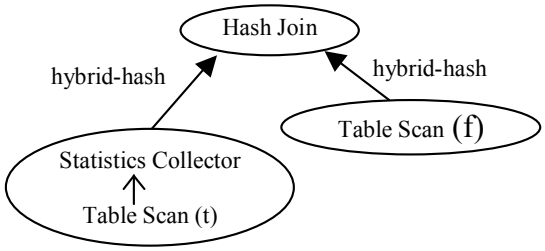


Figure 12. Adaptive Distribution Plan

As can be observed, there is a new operator, called "*statistics collector*", in the query tree. This operator buffers rows and aids in selecting the distribution method at runtime. For hash-hash, we let the statistics collector operator to buffer at most 2*DOP rows. Upon reaching this threshold or seeing end-of-input, it informs the query coordinator (QC) of the local cardinality. The QC aggregates the information sent by all the parallel processes and chooses the distribution method. If the left input cardinality is less than 2*DOP, the data distribution for the left input is set as "*broadcast*" and the right input is distributed randomly. Otherwise, "*hash-hash*" distribution on join keys is used. Once the distribution decision is made, the statistics collector becomes a *pass-through* operator. With this adaptive hash-hash distribution, we can massively scale joins irrespective of the accuracy of the optimizer's estimates. Our performance results (Section 8) show impressive gains from this method.

The hash-hash distribution method is naturally suited to adaptive behavior as the left and the right inputs of the join are in separate DFOs. Broadcast plans pose a difficulty as the right input is in the same DFO as the join. So if this plan were to be adaptive and become hash-hash when the left input is large, we would have to change the shape of the query tree at execution time. This is quite involved and we might consider it in the future. Note that when we place the hash join and its inputs in separate DFOs, we would have to buffer the output of join owing to our producer-consumer[1] model. This buffering can hinder performance if it were to spill to disk. So, we make a conservative choice of picking "*hybrid hash-hash*" distribution method for cases in which the optimizer is not confident about the statistics and choosing broadcast distribution method might result in a bad plan. In other words, we made our traditional "*hash-hash*" plans "*hybrid hash-hash*". In the next section, we describe improvements to the *broadcast* plans.

6. SMALL TABLE REPLICATION

As mentioned in Section 5, broadcast distribution is ideal for smaller inputs as it does not have to distribute the right input and is also resilient to skew. When there is skew, broadcast plan results in better CPU utilization and massive scalability. Broadcast plans are quite common in data warehousing workloads – a bigger fact table is joined with one or more small dimension tables (e.g., *time*, *geography*). Consider this query on star schema:

```
Q5 SELECT country, year, SUM(sales)
   FROM fact f, time_dim t, geog_dim g
   WHERE f.time_key = t.time_key AND
         f.geog_key = g.geog_key
   GROUP BY country, year;
```

Parallel execution plan for this query follows the model in Figure 11 and would broadcast the small dimension tables, *time_dim* and *geog_dim*. The cost of broadcasting is proportional to the number of rows being distributed and the degree of parallelism (DOP). With massive DOP, broadcasting small tables can consume significant network and CPU resources. This can be improved with a new parallelization model called "*small table replication*". A table is a candidate for small table replication if it can be cached in the buffer pool of a database instance. Scans of such tables would be serviced from the database buffer cache instead of expensive disk reads and hence, scanning done by multiple parallel processes should have negligible cost. The name replication comes from the fact that these small tables get replicated in the buffer pools of multiple database instances. Once these small tables are brought in the database buffer pool/cache, they will stay there because of their high frequency of usage. This feature goes extremely well with in-memory parallel query execution where caching of the database objects is maximized.

With small table replication, scan of the small table and the subsequent join operation get combined into a single DFO. Figure 13 shows parallel plan with small table replication. This plan has several benefits – first, there is no data distribution cost. For multi-instance database clusters in which data distribution happens on a network, this saving is significant. Secondly, there will be a reduction in parallel query startup and teardown costs, as only one set of parallel processes is sufficient to execute the statement. Finally, the unused set of parallel processes is available to other concurrent database statements.

[1] Consuming parallel processes cannot produce rows and have to buffer results till they become producers.

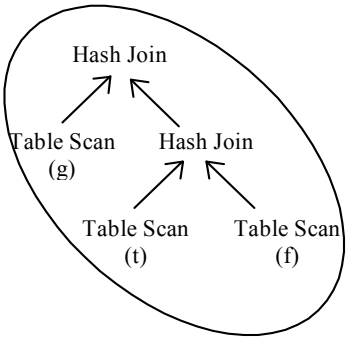


Figure 13. Small Table Replication Plan

7. HANDLING SERIALIZATION POINT

In Oracle parallel execution plans, serialization points can exist due to the presence of operators (e.g. *top-N*) that are inherently serial. Obviously, having a serialization point is not good for massive parallelism. Firstly, any serialization point is executed on the query coordinator (QC) process. This would block the QC from performing its usual tasks of coordinating the query execution among parallel processes. Secondly, any single-input operator (e.g. group-by) above the serialization point also becomes serial, and any multiple-input operator (e.g. join) becomes serial if all its inputs are serial.

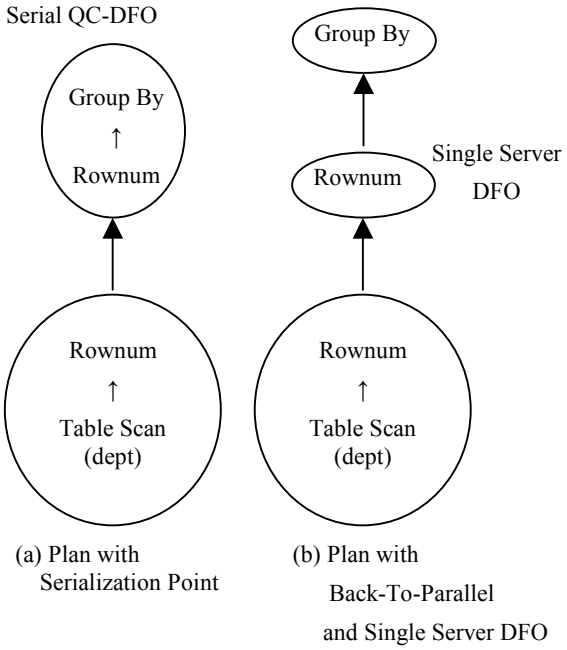


Figure 14. Back-to-Parallel Plan

Two new features, *Back-To-Parallel* and *Single-Server DFO*, are introduced to handle serialization points. The *Back-To-Parallel* feature brings an operator back to parallelism after a serialization point. *Single-Server DFO* feature places the serialization point on a parallel process instead of on the QC, thereby letting the QC do important tasks.

Consider the following *top-N* query containing an inherently serial *rownum* predicate. Figure 14 shows the plan for this query, with original plan on the left and the new plan on the right side.

```
Q6 SELECT time_key, sum(sales)
     FROM (SELECT * FROM fact ORDER BY sales)
     WHERE rownum < 100000
     GROUP BY time_key;
```

In the original plan, the DFO containing the serialized rownum is executed on the QC and is tagged as *serial QC-DFO*. Observe rownum pushdown parallelism and that the serial rownum is needed to produce the correct result. The GBY operator above the serial rownum is serial as well. In the new plan, the GBY operator goes back to parallel and the serial rownum is executed on a single parallel process instead of the QC. Back-To-Parallel feature is always applied as long as the operator above the serialization point has a calculated DOP that is greater than 1. Single-Server DFO feature is applied whenever the operators in a serial DFO can be executed on a parallel process.

8. PERFORMANCE STUDY

Performance experiments were conducted on a 2-node Oracle RAC database, with each node having 40 2.26GHz dual-core CPUs and 200GB of memory. This system supports a *degree of parallelism* (DOP) of 160. To evaluate the effectiveness of our parallelization techniques in adapting to data reduction ratios and skew, and in scaling the computation beyond the traditional parallelization, we had to use synthetic datasets. Some of these synthetic datasets are derived from our customer datasets. We varied the data reduction (due to GBY) ratios, skew and the NDV of distribution keys in the experiments. Statistics were gathered on all the tables involved. Due to the criticality of GBY operation, we validated our new GBY parallelization on the 3TB TPC-H.

8.1 Group-By

We took a GBY query with 50 aggregates (*min*, *max*, *count*, *avg*, and *stddev* on each of the 10 columns) and measured the query performance, varying the input size (16, 64, and 256 million rows) and the GBY reduction ratio (low i.e. unique data and high). Due to the high overhead of processing the aggregates, this query is least favorable to *group-by pushdown* in the low reduction scenario. We chose it to see how much the performance might regress in choosing *hybrid batch flushing*. When GPD becomes adaptive and enters the *pass-through* mode, it needs to marshal the aggregate operator's result to be like a partially aggregated value, and the higher the number of aggregates, the greater the number of CPU cycles spent in marshalling.

In the results shown in Figures 15 and 16, we use labels CB_GPD and HBF for the existing cost-based *group-by pushdown* scheme and the new *hybrid batch-flushing* scheme respectively. With unique data and accurate statistics (Figure 15), query optimizer decision of not choosing group-by pushdown is ideal. As can be observed, HBF performed on par with CB_GPD (a non-pushdown plan) by adapting itself at runtime to a plan that is effectively a non-pushdown plan. The slight improvement can be ignored as a run-to-run variation.

Next, we introduced skew in the data – about 75% of the rows have same value for GBY keys and rest of them are unique. In this case, Oracle query optimizer incorrectly picked the non-pushdown plan i.e. CB_GPD is a non-pushdown plan. With CB_GPD, 75% of rows are processed by one parallel process. In contrast, HBF

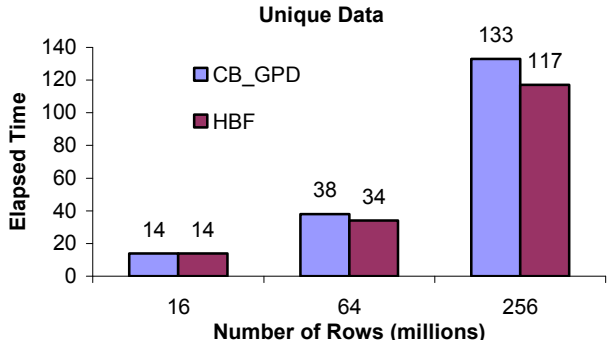


Figure 15. GBY Performance – Unique Data

handled skew and gave 2*x* improvement (Figure 16). HBF performed as expected in other experimental variations as well.

We then tested HBF on query *q18* (that groups *lineitem* table by *orderkey*) of TPC-H [5]. For this query, pushdown will not be beneficial due to low reduction (about 4 *lineitem*s per *order*) and can even lead to performance regression. Our HBF technique adapted at runtime as designed and there was no perceptible degradation in *q18* performance.

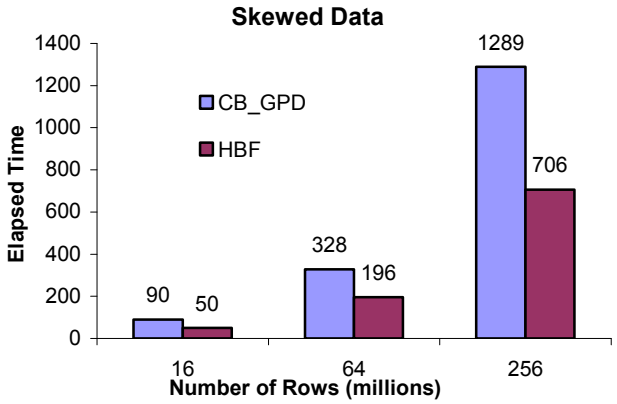


Figure 16. GBY Performance – Skewed Data

8.2 Rollup

To test our ROLLUP parallelization strategy, we took a query with shape similar to Q1 and varied the GBY reduction ratio and the NDV of non-rollup keys (*country* for Q1). Table 4 summarizes these results – it gives the elapsed times when the NDV of non-rollup keys is greater/lesser than the DOP, and for the high (90%) and low (1%) reduction scenarios.

Table 4. Adaptive Rollup Performance

Reduction	NDV>>DOP	Static	Adaptive	Benefit
Low (1%)	Yes	89	102	0.9x
	No	1663	119	14x
High (90%)	Yes	27	23	1.2x
	No	646	26	24.8x

Unlike the "*static*" parallelization model that distributes work based on non-rollup keys, our new "*adaptive*" parallelization model picks the right set of distribution keys at runtime and scales well. When the NDV of non-rollup keys is lower than the DOP, the "*static*" plan used only a few parallel processes, while the

"*adaptive*" plan uses all CPU resources and ran significantly faster (up to 25*x*). As expected, there was no difference in performance when NDV is higher than the DOP. Elapsed times for the low reduction case are high due to GBY spilling to disk.

8.3 Window Functions

In the first experiment, we took a query like Q2 and varied the number of rows in the table from 16 to 1024 million. The NDV of the common PBY key ("*y*" in Q2) is a mere 2 and the naïve parallelization scheme (tagged "*common-pby*") could only use 2 parallel processes. In contrast, our adaptive method (tagged "*adaptive*") finds at query execution time that the NDV is low and pushes the computation of all the window functions to 160 parallel processes. Results, as shown in Figure 17, demonstrate that "*adaptive*" parallelization improved the performance by a whopping 20x for 64m and 256m row cases. One would expect an improvement of 80x with "*adaptive*" as it ran with a DOP of 160. Mimicking our customers' usage of window functions for complex analysis, we used "*create table as select*" statement in our scalability experiments. Hence, the improvement is only 20x as the costs of table scan and load into a destination table are included. In addition, the "*adaptive*" plan incurs an extra cost of performing hash lookup to produce the final results. The benefit is lower for 16m and 1024m row cases as the data was fitting in-memory in the former case and was spilling to disk in the latter.

Figure 17 also shows the performance comparison when the NDV of common PBY key is greater than the DOP. This result is inline with our expectation – that there will be a small (negligible for large datasets) penalty with "*adaptive*" algorithm due to NDV counting and the code overheads.

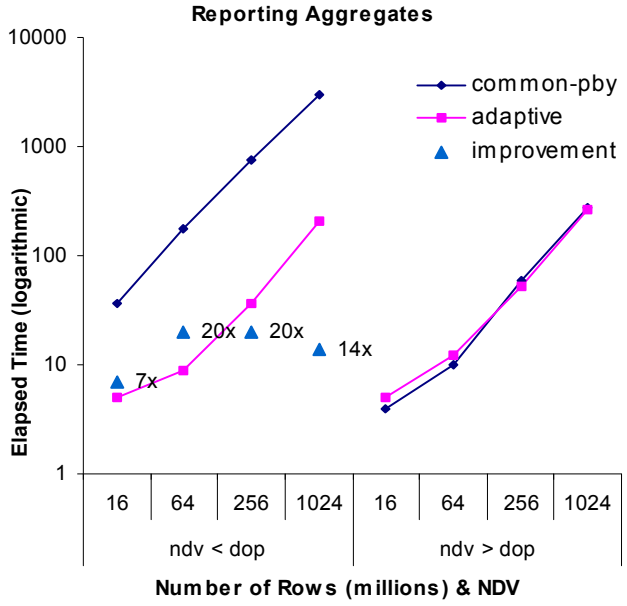


Figure 17. Reporting Aggregate Performance

Next we compared the two alternatives – *extended distribution keys* (Figure 6) and *pushdown* (Figure 7), with the current common PBY-based parallelization of the reporting aggregates. For the 256m rows input, we took the best case for PBY parallelization where in the PBY key has NDV much higher than the DOP and forced our runtime window function parallelization algorithm to pick different distribution keys. The results and explanation is tabulated in Table 5. For the "*default*" case, parallelization based on common PBY key is ideal. When we force the distribution keys to be 2, 3, or 4, performance suffers due to extra data distribution and code overheads. The "*pushdown*" case finds that the NDV of common PBY key is greater than the DOP and adapts to the "*default*" case.

Table 5. Reporting Aggregates Performance

Distribution Keys	Elapsed Time	Explanation
default (1)	36	best-case
2	52	extra data distribution
3	53	
4	44	
pushdown	36	adapts to default

In the final experiment, we compared the PBY key based parallelization of ranking and cumulative window functions with the new parallelization scheme that uses extended keys. We took a query (like Q3) with *rank*, *dense_rank* and *cumulative* sum functions having the same PBY keys, but different orderings (ok_1), (ok_1, ok_2), and (ok_1, ok_2, ok_3). Figure 18 shows the results for low and high NDV PBY keys. For the low NDV case, the current parallelization (tagged "*pby*") could use only few (4 in this case) parallel processes. Our new parallelization scheme (tagged "*extended keys*") uses PBY and OBY keys for distribution, and achieves better scalability; it gives up to 10x improvement. When the NDV of PBY keys is greater than the DOP, "*extended keys*" scheme incurs tiny overhead due to NDV counting.

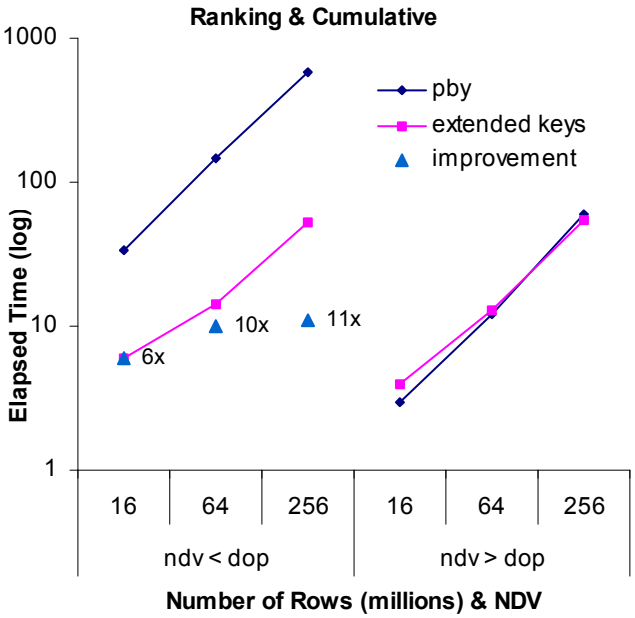


Figure 18. Ranking & Cumulative Performance

8.4 Adaptive Distribution Methods for Joins

We conducted a study comparing ADDM plan (Figure 12) with the traditional *hash-hash* plan (Figure 10). We took the scenario in which the optimizer incorrectly picks the hash-hash plan.

Both tables have two distinct key values, with *time_dim* having just two rows and *fact* having millions of rows. ADDM broadcasts *time_dim* and distributes *fact* in a round robin fashion, thereby using all the 160 parallel processes for the join. The *hash-hash* plan could only use 2 parallel processes for the join. Results as shown in Table 6 show dramatic improvement and could have been 80*x*, if not for the constant costs (parallel query startup/teardown).

Table 6. ADDM Performance

fact size	Hash-Hash	ADDM	Improvement
16m	1.32	0.06	22x
64m	5.37	0.15	36x
256m	20.74	0.45	46x

8.5 Small Table Replication

Our experiments comparing small table replication (Figure 13) with broadcast (Figure 11) yielded expected results. We used query Q5 with 1M row table T1 and a DOP of 160. Once T1 is cached, small table replication was 8*x* faster in handling T1 than scanning and broadcasting the table T1 to hash join.

9. RELATED WORK

Recent years have seen some published research work related to SQL window functions – [9][1] use window functions in query optimization, [10] addresses the performance of a set of window functions in a query but doesn't touch upon massive scalability; [1] describes ways to parallelize window functions with zero or low cardinality PBY keys and is a precursor to the elaborate, scalable and adaptive parallelization schemes of this paper. A novel way of handling of skew in the context of joins was presented in [11], but no prior work has attempted skew handling for window functions. Some of our window parallelization techniques can be extended to handle skew – much like our "*extended distribution key*" method, a skewed partition can be distributed to multiple parallel processes, with the query coordinator consolidating the local results.

Research on cache-friendly hash joins [12] and the performance analysis of our hash algorithms (joins and aggregation) motivated us to use hash tables that are cache resident. In [13], the authors propose sharing of the hash table by the parallel processes performing group-by, and being orthogonal, that strategy can be employed along with our *hybrid batch flushing*. Hive [8] employs "map-join" optimization that is similar to our small table replication method. Unlike [8], we do not require any pre-processing (map-reduce local task) for small table replication. In the steady state, Oracle RDBMS would have small tables cached in the buffer cache. Small table replication benefits us by picking higher parallelism for the query, or makes parallel resources available to other queries in the system.

10. CONCLUSION

Joins, aggregations, and analytics are the key operations used to analyze the already huge, and fast-growing datasets residing in traditional databases or in hadoop-based distributed systems. Scalable and adaptive evaluation of these operations is of paramount importance to the success of any data management system. To that end, we developed several techniques for massive scalability of SQL operations in the Oracle RDBMS. We believe these techniques would be quite applicable to analyses using hadoop map-reduce jobs.

Our parallelization schemes are not only scalable, but adaptable – ADDM chooses a distribution method for joins based on the real input size rather than the optimizer estimate; group-by pushdown with hybrid batch flushing adapts based on the data reduction observed during execution; ROLLUP and window function parallelization models choose work distribution based on data demographics. Performance studies on datasets and database statements similar to what we have seen from customers have yielded exciting results – up to 30x improvement to elapsed times.

11. ACKNOWLEDGMENTS

We thank our colleagues for their inputs, Allison Lee for *statistics collector* work, and Murali Krishna for the performance work.

12. REFERENCES

[1] R. Ahmed, et al, "*Cost-Based Query Transformation in Oracle*", Proceedings of the 32nd VLDB Conference, Seoul, S. Korea, 2006.

[2] S. Bellamkonda, et al, "*Enhanced Subquery Optimizations in Oracle*", Proceedings of the 35th VLDB Conference, Lyon, France, 2009.

[3] A. Eisenberg, K. Kulkarni, et al, "*SQL: 2003 Has Been Published*", SIGMOD Record, March 2004.

[4] SQL – Part2: Foundation, ISO/IEC 9075:1999.

[5] TPC-H (Decision Support), Standard Specification Rev 2.8, TPC-DS Specification Draft, Rev 32, http://www.tpc.org/

[6] OLAP Council, APB-1 OLAP Benchmark, Release II, http://olapcouncil.org/research/APB1R2_spec.pdf

[7] S. Agarwal, et al, "*On the Computation of Multidimensional Aggregates*", Proceedings of the 22nd VLDB Conference, Mumbai (Bombay), India, 1996.

[8] Map Join Optimization, Apache Hive, https://cwiki.apache.org/Hive/joinoptimization.html

[9] C. Zuzarte, et al, "*WinMagic: Subquery Elimination Using Window Aggregation*", Proceedings of the 2003 ACM SIGMOD Conference, San Diego, USA.

[10] Y. Cao, et al, "*Optimization of Analytic Window Functions*", Proceedings of the 38th VLDB Conference, Istanbul, 2012.

[11] Y. Xu, et al, "*Handling Data Skew in Parallel Joins in Shared-Nothing*", Proceedings of the 2008 ACM SIGMOD Conference, Vancouver, BC, Canada.

[12] C. Kim, et al, "*Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs*", Proceedings of the 35th VLDB Conference, Lyon, France, 2009.

[13] Y. Ye, et al, "*Scalable Aggregation on Multicore Processors*", Proceedings of the 7th International Workshop on Data Management on New Hardware, Athens, 2011.

[14] J. Gray, et al, "*Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals*", Data Mining and Knowledge Discovery, 1997.

[15] V. Harinarayan, et al, "*Implementing Data Cubes Efficiently*", Proceedings of the 1996 ACM SIGMOD Conference, New York, USA.