Query Optimization Time: The New Bottleneck in Real-time Analytics

Rajkumar Sen
MemSQL Inc.
534 4th Street,
San Francisco,
CA 94107, U.S.A
raj@memsql.com

Jack Chen
MemSQL Inc.
534 4th Street,
San Francisco,
CA 94107, U.S.A
jack@memsql.com

Nika Jimsheleishvilli
MemSQL Inc.
534 4th Street,
San Francisco,
CA 94107, U.S.A
nika@memsql.com

ABSTRACT
In the recent past, in-memory distributed database management systems have become increasingly popular to manage and query huge amounts of data. For an in-memory distributed database like MemSQL, it is imperative that the analytical queries run fast. A huge proportion of MemSQL's customer workloads have ad-hoc analytical queries that need to finish execution within a second or a few seconds. This leaves us with very little time to perform query optimization for complex queries involving several joins, aggregations, sub-queries etc. Even for queries that are not ad-hoc, a change in data statistics can trigger query re-optimization. Query Optimization, if not done intelligently, could very well be the bottleneck for such complex analytical queries that require real-time response. In this paper, we outline some of the early steps that we have taken to reduce the query optimization time without sacrificing plan quality. We optimized the Enumerator (the optimizer component that determines operator order), which takes up bulk of the optimization time. Generating bushy plans inside the Enumerator can be a bottleneck and so we used heuristics to generate bushy plans via query rewrite. We also implemented new distribution aware greedy heuristics to generate a good starting candidate plan that significantly prunes out states during search space analysis inside the Enumerator. We demonstrate the effectiveness of these techniques over several queries in TPC-H and TPC-DS benchmarks.

1. INTRODUCTION
In the past few years, there has been a massive adoption of distributed in-memory databases. The ability to store and query huge amounts of data by parallelizing execution across nodes leads to dramatic performance improvements in execution times for analytical data workloads. A few industrial database systems such as MemSQL [13], SAP HANA [9], Teradata/Aster [10], Netezza [11], SQL Server PDW [6], Oracle Exadata [12], Pivotal GreenPlum [7], Vertica [8], VectorWise [16] etc. have gained popularity and are designed to run queries very fast.

IMDM '15, August 31 2015, Kohala Coast, HI, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3713-7/15/08 $15.00
DOI: http://dx.doi.org/10.1145/2803140.2803148

1.1 Overview of MemSQL
MemSQL [13] is a database for performing real time transactions and analytics. By storing data in memory, MemSQL can concurrently read and write data on a distributed system, therefore enabling real-time analytics over an operational database. Due to its innovative in-memory storage of data with lock-free data structures and its extremely scalable distributed architecture, MemSQL achieves sub-second query latencies across very high volumes of data. MemSQL is designed to scale on commodity hardware and does not require any special hardware or instruction set to demonstrate its raw power. MemSQL has a shared-nothing architecture, which means that no two nodes in the distributed system share memory, disk or CPU. MemSQL has a two-tiered [14] clustered architecture that consists of two types of nodes: aggregator nodes and leaf nodes. Aggregator nodes serve as mediators between the client and the cluster, while leaf nodes provide the data storage and query processing backbone of the system. Users route queries to the aggregator nodes, where they are parsed, optimized, and planned.

1.2 Query Optimization in MemSQL
MemSQL [13] is a database for real-time transactions and analytics, which must support a wide variety of challenging queries. A lot of queries that MemSQL executes are complex queries from enterprise analytical workloads, involving joins across star and snowflake schemas, sorting, grouping and aggregations, and nested sub-queries. A considerable percentage of those queries go through the process of query optimization because they are either ad-hoc or the data statistics have changed enough to trigger re-optimization. These queries often must be answered within latencies measured in seconds or even milliseconds despite being highly resource intensive. The goal of the query optimizer is to find the best query execution plan for a given query by enumerating a wide space of potential execution paths and then selecting the plan with the least cost.

The MemSQL Query Optimizer is a modular layer in the database engine. The optimizer framework is divided into three major modules:

(1) *Rewriter*: The Rewriter applies SQL-to-SQL rewrites on the query. Depending on the characteristics of the query and the rewrite itself, the rewrite decides whether to apply the rewrite using heuristics or cost. The Rewriter is also smart to apply certain rewrites in a top-down fashion while applying others in a bottom-up manner. It also interleaves rewrites that can mutually benefit from each other.

(2) *Enumerator*: The Enumerator is a central component of the optimizer, which determines the distributed join order and data movement decisions. It considers a wide search space of various execution alternatives and selects the best join order, based on the cost models of the database operations and the network data movement operations. The Enumerator is invoked by the Rewriter to cost transformed queries when the Rewriter wants to perform a cost-based query rewrite.

(3) *Planner:* The Planner converts the chosen logical execution plan to a sequence of distributed query and data movement operations.

1.3 Reducing Query Optimization time

It is imperative for a system like MemSQL that the time spent in optimizing a query should not overshadow the benefits of in-memory and distributed query execution. Query Optimization cannot afford to be the bottleneck in a system that is expected to answer real-time analytic queries within a second or few second (sometimes fraction of a second). At the same time, it is also essential that the optimizer generate a good plan for complex queries involving joins, aggregations, sub-queries etc. This poses a new challenge in the process of query optimization. Query Optimization, if not done intelligently, could very well be the bottleneck for such complex analytical queries that require real-time response. At MemSQL, we have taken some initial steps in this direction

(a) Instead of generating bushy join trees inside the Enumerator, we generate bushy join trees via query rewrite using heuristics that are based on schema and query.

(b) Enumerate very fast by extensively pruning the operator order search space. We implemented new data distribution aware greedy heuristics to determine an initial candidate operator order and use that to extensively prune states in the search space analysis phase.

1.4 Related Work

In the past, there have been several attempts to improve query optimization time. Bruno et al. [19] propose several polynomial heuristics that take into account selectivity, intermediate join size etc. Some other previous work [20][21] also propose several heuristics but all these techniques were designed in the days when distributed query processing was not in vogue and therefore, they do not take data distribution into consideration. Another area where there have been attempts to improve query optimization time is in parallelizing the Enumeration process. Han et al. in [23] proposes several techniques to parallelize parts of the System-R style enumerator and prototyped in PostGreSQL. Waas et al. in [22] propose techniques to parallelize the enumeration process for Cascade style enumerators. A very recent work by Heimel et al. [24] suggests using GPU co-processor to speed up the query optimization process.

2. BUSHY PLAN GENERATION

As mentioned in the literature [3][4], generating Bushy Plans as part of the join enumeration makes the problem of finding the optimal join permutation extremely challenging and time-consuming. MemSQL solves this problem by generating Bushy Plans using heuristics and implementing it via query rewrite. A direct advantage of generating bushy plans in this way is that we would only consider bushy plans when there is a potential benefit.

2.1 Bushy Plans via Query Rewrite

Even if the Enumerator considers only left-deep join trees, it is easy to generate a query execution plan that is bushy in nature. This can be done by creating a subselect/view/derived-table using the query rewrite mechanism and using the view as the right side of the join. The Enumerator works as usual; it treats the view/derived-table as another base table.

In particular, we use a query rewrite called *Table Pushdown* to generate bushy plans in MemSQL. *Table Pushdown* is a rewrite mechanism, which transforms a table joined with a subselect by pushing the table inside the subselect. *Table Pushdown* has its origins in Magic-set subquery de-correlation technique proposed in [18]. For *Table Pushdown*, we primarily look for subselects, which are joined with an outer table on its primary key. This ensures that evaluating the join with the outer table does not increase the size of the join, so that the transformed query plan is unlikely to do worse. It also ensures that if the sub-select has any grouping, as long as we have a join between the group-by columns of the subselect and the primary key of the outer table, pushing the table inside will not change the semantics of the grouping. It is still possible to do *Table Pushdown* without a primary key join, but it is less likely to be advantageous, and may be more complex: for example, if the subselect has a group by, we must add the primary key of the outer table to the group by to preserve the correct grouping. The following sample query using tables from TPC-H benchmark would help us understand it better.

```
SELECT Sum(l_extendedprice) / 7.0 AS avg_yearly
FROM   lineitem,
        part,
        (
        SELECT 0.2 * Avg(l_quantity) AS s_avg,
                l_partkey AS s_partkey
        FROM   lineitem
        GROUP BY l_partkey
        ) sub
WHERE  p_partkey = l_partkey
        AND p_brand = 'Brand#43'
        AND p_container = 'LG PACK'
        AND p_partkey = s_partkey
        AND l_quantity < s_avg
```

After applying *Table Pushdown* rewrite, we get the following equivalent query

```
SELECT Sum(l_extendedprice) / 7.0 AS avg_yearly
FROM   lineitem,
        (
        SELECT 0.2 * Avg(l_quantity) AS s_avg,
                l_partkey AS s_partkey
        FROM   lineitem,
                part
        WHERE  p_brand = 'Brand#43'
```

```
              AND p_container = 'LG PACK'
              AND p_partkey = l_partkey
       GROUP  BY l_partkey
       ) sub
WHERE  s_partkey = l_partkey
       AND l_quantity < s_avg
```

Note that the join between the subselect and *part* is on *p_partkey* = *s_partkey*, the primary key of *part* and the group by key of the subselect. So we can easily push the join with *part* inside the subselect, making it far cheaper.

2.2 Bushy Plan Heuristics

Using smart heuristics, it is possible to consider particularly promising bushy joins with relatively little cost. We can form one or more subselects, each of which has an independent left-deep join tree. The Enumerator chooses the best left-deep join tree within each sub-select, as well as the outer select block. By placing a sub-select on the right side of a join, we form a bushy join tree. For example, consider a snowstorm shape query, where there are multiple large fact tables, each joined against its associated dimension table(s), which have single-table filters. The best left-deep join plan generally must join each fact table after the first by either joining it before its associated dimension tables, when its size has not yet been reduced by their filters, or by joining the dimension table first, an expensive Cartesian product join. We may benefit greatly from a bushy join plan where we join the fact table with its dimension tables, benefiting from their filters, before joining it to the previous tables.

Our strategy to generate bushy join plans is to try moving a single table into a subselect, and then apply *Table Pushdown* to push joined tables into that subselect, especially ones with selective table filters. To determine which "*seed*" tables to consider as candidate starting points for forming these subselects, we use heuristics which often allow us to find helpful bushy join plans on snowstorm query shapes, such as trying all tables which are joined against the primary key of another table which has a single-table filter. In a snowstorm-type query, this will find fact tables, which are joined to the primary key of their associated dimension tables where at least one of the dimension tables has a single-table filter. This is exactly the type of situation where we most benefit from generating a bushy join plan through *Table Pushdown*. The Rewriter will generate different candidate bushy join trees using these "*seed*" tables (one bushy view per *seed* table) and it will use the Enumerator to cost each combination and then (based on cost) decide which ones to retain. As an example, consider TPC-DS [2] query 25:

```
SELECT  …….
FROM  store_sales ss,
      store_returns sr,
      catalog_sales cs,
      date_dim d1,
      date_dim d2,
      date_dim d3,
      store s,
      item i
WHERE  d1.d_moy = 4
      AND d1.d_year = 2000
      AND d1.d_date_sk = ss_sold_date_sk
      AND i_item_sk = ss_item_sk
      AND s_store_sk = ss_store_sk
      AND ss_customer_sk = sr_customer_sk
      AND ss_item_sk = sr_item_sk
      AND ss_ticket_number = sr_ticket_number
      AND sr_returned_date_sk = d2.d_date_sk
      AND d2.d_moy BETWEEN 4 AND 10
      AND d2.d_year = 2000
      AND sr_customer_sk = cs_bill_customer_sk
      AND sr_item_sk = cs_item_sk
      AND cs_sold_date_sk = d3.d_date_sk
      AND d3.d_moy BETWEEN 4 AND 10
      AND d3.d_year = 2000
GROUP  BY i_item_id,
          i_item_desc,
          s_store_id,
          s_store_name
ORDER  BY i_item_id,
          i_item_desc,
          s_store_id,
          s_store_name
LIMIT  100;
```

Here, there are three fact tables (*store_sales*, *store_returns*, and *catalog_sales*), each joined against one dimension table with a single-table filter (date_dim). In the cluster setup described in Section 4, the best left-deep join plan chosen by the Enumerator is *(d1, ss, sr, d2, s, i, d3, cs)*. Note that when we join *d3* it is as a Cartesian product join, because *d3* only has join predicates with *cs*, so this is expensive, but given the restriction to left-deep join trees it is the better alternative to first joining *cs* without having any of the filtering that comes from the single-table filters on *d3*. We consider *cs* as the first "*seed*" table for forming a sub-select for a bushy join plan because it is joined against the primary key of *d3*, which has single-table filters. Applying Table Pushdown pushes *d3* into the subselect to join with *cs*, because it is joined on its primary key and has table filters. Now, the Enumerator chooses the best left-deep join plan in each select block, producing the overall bushy join order *(d1, ss, sr, d2, s, i, (d3, cs))*. We also consider *ss* and *sr* as "*seed*" tables, but these bushy views do not improve the cost of the query and are rejected. The bushy join plan runs 10.1 times as fast as the left-deep join plan. The transformed query is

```
SELECT ……
FROM   store_sales,
       store_returns,
       date_dim d1,
       date_dim d2,
       store,
       item,
       (SELECT *
        FROM   catalog_sales,
               date_dim d3
        WHERE  cs_sold_date_sk = d3.d_date_sk
               AND d3.d_moy BETWEEN 4 AND 10
               AND d3.d_year = 2000) sub
WHERE  d1.d_moy = 4
       AND d1.d_year = 2000
       AND d1.d_date_sk = ss_sold_date_sk
       AND i_item_sk = ss_item_sk
       AND s_store_sk = ss_store_sk
       AND ss_customer_sk = sr_customer_sk
       AND ss_item_sk = sr_item_sk
       AND ss_ticket_number = sr_ticket_number
       AND sr_returned_date_sk = d2.d_date_sk
       AND d2.d_moy BETWEEN 4 AND 10
       AND d2.d_year = 2000
       AND sr_customer_sk = cs_bill_customer_sk
       AND sr_item_sk = cs_item_sk
GROUP  BY i_item_id,
          i_item_desc,
          s_store_id,
          s_store_name
ORDER  BY i_item_id,
          i_item_desc,
          s_store_id,
          s_store_name
LIMIT  100;
```

By considering these query shapes in the context of a query transformation pushing tables into a subselect, rather than by considering bushy join trees in the Enumerator, we are able to take advantage of this same transformation for simple bushy join trees as well as join trees with subselects, especially those with a group by. For example, using *Group-by Pushdown* in conjunction with *Table Pushdown*, we can transform the query by moving some of the tables along with a group by into a subselect: we can join a fact table together with its associated dimension tables, then evaluate the group by, then join with the remaining tables. We are now able to separately join the inner and outer tables, and then join the two intermediate relations together, forming a bushy join tree, which also contains a group by.

It is worthwhile to note here that the technique of using a query rewrite mechanism to generate bushy join plans is not new and has already been explored in [5]. However, the methods used to achieve the same in [5] and in our Rewriter are totally different from each other. The mechanism in [5] identifies fact (large), dimension (small) and branch tables using table cardinalities, statistics and join conditions. It then uses a combination of such tables to form a view (sub-select). Instead, the MemSQL Rewriter does not do any categorization of tables based on cardinalities and statistics. It identifies a set of "*seed*" tables as candidate starting points for the sub-selects based on a set of heuristics that the *seed* table joins with a primary key of another table (without taking into account the cardinality or any other statistic of the *seed* table). Using the seed table and our *Table Pushdown* rewrite, we construct the bushy view (subselect).

3. FAST ENUMERATION

Since MemSQL is an in-memory distributed database that satisfies real-time constraints on complex analytical queries, there are several occasions where the optimizer has to come up with the DQEP within very limited time budgets. For readers who are well versed in the area of query optimization, it would not come as a surprise that the enumeration component takes up bulk of the time as it has to implement a search space analysis algorithm using some pruning criteria. The greater the number of tables in the query, the more time consuming the process becomes. Many industrial query optimizers implement a left deep or right deep join order enumeration algorithm to limit the number of states in the search space and avoid a combinatorial explosion. That alone is not enough for MemSQL because the join order is distributed and addition of data movement operations increases the state space further.

3.1 Enumerator Search Space Analysis

The Enumerator optimizes the join plan within each select block, but does not consider optimizations involving moving joins between different select blocks, which is instead done by the Rewriter. The Enumerator processes the select blocks bottom-up, starting by optimizing the smallest expressions (subselects), and then using the annotation information to progressively optimize larger expressions (subselects that are parents of other sub-selects). Eventually, the physical plan for the entire operator tree is determined when the enumerator is done with the outermost SELECT block. Even though a bottom-up approach is used, a top-down enumeration should still be applicable with the same set of pruning heuristics. Figure 1 depicts the pseudo-code for the bottom-up enumerator.

As mentioned before, the set of possible plans is huge and the search space size increases by the introduction of data movement operations. To limit the combinatorial explosion, the Enumerator implements a bottom-up System-R [1] style dynamic programming based join order enumerator with *interesting properties*. System-R style optimizers have the notion of "*interesting orders*" to help optimize for physical properties like sort order etc. MemSQL Optimizer Enumerator employs an *interesting property* of "*sharding distribution*"; i.e. the set of keys over which data is sharded in the distributed system. The interesting shard keys are (1) predicate columns of equality joins and (2) grouping columns.

```
Enumerator(Select)
{
    Apply the Enumerator over all child sub-selects
    Use the heuristics to generate initial candidate join orders
    Select the best cost plan from the candidates
    Determine the best join order for Select, using the best known cost to prune.
    Add optimizer annotations to the operator tree
}
```

Figure 1: Pseudo-code for the Enumerator

3.2 Pruning & Pre-Processing

Pruning is vital to finishing enumeration within a limited time budget. In the ideal situation, we want to generate very good plans using heuristics so that we can eliminate most of the join orders during the enumeration process. The Enumerator, therefore, starts by using a number of heuristics to generate a set of initial candidate join orders. These candidate join orders are then cost estimated, including the size, cost of individual joins and cost of data movement operations. After costing the candidate join orders, the cheapest one amongst them provides an upper bound on the cost we need to consider in the dynamic programming, allowing us to prune the search space. We can easily use multiple cheap heuristics, since the cost of generating these initial join orders is far smaller than the cost of the full enumeration. These heuristics, depending on the schema and the query, can considerably reduce the time spent in enumeration.

The very first candidate join order we consider is the given join order as specified in the query, which often is meaningful, and makes it very easy to give a hint to the optimizer. We also consider a simple greedy order, joining the tables in increasing order of size, preferring co-located joins over ones which require data movement, and preferring primary key-foreign key joins over other joins, and over Cartesian product joins. In addition, we consider join orders from a rule-based greedy approach.

3.2.1 *Rule-based Greedy Candidate Join order*

We use a relatively simple and fast rule-based greedy algorithm to construct a candidate join order, which often is fairly reasonable. The heuristics are based on primary key-foreign key relationships and shard keys of the tables. The greedy algorithm starts with a table that joins with the largest number of tables on their primary keys, which is often a central fact table. (Note that we can later swap the first and second tables to start with a smaller dimension table in the left-deep join tree.) Then, we greedily select the next table in the join using the following rules, in order of precedence:

- Prefer a table that has a join with the current tables. Among these, prefer a join on a primary or unique key over any other join condition.
- If there are any remaining tables which can be joined without data movement (i.e. the join and shard keys match), prefer one of them, or any other table for which the join method will preserve this partitioning (i.e. where we would choose to broadcast or repartition that table). The rationale allowing a next table, which does have a shard/join key match, is that after joining with the ones, which do, we would still in most circumstances join it with the same data movement method. By joining it first, we can still join the shard/join key matching tables without movement later, and we may benefit from a more selective join (the next rule).
- Prefer a table that results in the smallest intermediate join size, taking into account the selectivity of the available join and table filters.
- Prefer the table that requires the least cost to join.

Consider TPC-H query 8 as an example:

```
SELECT …..
       FROM   part,
               supplier,
               lineitem,
               orders,
               customer,
               nation n1,
               nation n2,
               region
       WHERE  p_partkey = l_partkey
               AND s_suppkey = l_suppkey
               AND l_orderkey = o_orderkey
               AND o_custkey = c_custkey
               AND c_nationkey = n1.n_nationkey
               AND n1.n_regionkey = r_regionkey
               AND r_name = 'AMERICA'
               AND s_nationkey = n2.n_nationkey
               AND   o_orderdate   BETWEEN   DATE
('1995-01-01') AND DATE ('1996-12-31')
               AND p_type = 'LARGE BRUSHED BRASS'
       ) AS all_nations
GROUP BY o_year
ORDER BY o_year
```

In this example, we will assume the cluster setup described in Section 4, at scale factor 10. We begin with *lineitem*, because it is joined against three tables on their primary keys: *orders*, *supplier*, and *part.* All three of those tables are tied under the first rule, which favors primary key joins. Of those tables, *orders* table has a join and shard key match, so it can be joined without movement. However, because *part* is small (especially after filters), the best way to join *part* would be to broadcast it, which preserves the partitioning of the join-so-far. Between *orders* and *part*, we prefer *part* because it has a more selective filter, thus yielding a smaller number of rows after the join. After joining *part*, we join with *orders* next, because it has both a shard key match and table filters. Now we have a primary key match with *supplier* and *customer*, both of which we would join by repartitioning the left side. Since supplier is smaller and thus cheaper, we join it next. Next is *nation n2* by broadcasting the right side, then *customer* (by repartitioning the left side), then *nation n1* by broadcasting nation, and finally *region* by broadcasting right. This is in fact the optimal distributed join plan. Finding it therefore allows us to prune the vast majority of the Enumerator's dynamic programming search space. After finding this initial candidate join order, we can prune 96% of the total dynamic programming search space. Even if we already heuristically prune all Cartesian product joins, it still prunes 83% of the remaining search space. Of course, this example includes only eight tables, and with more tables the search space would be exponentially larger, as well as the number and proportion of bad intermediate join plans which could be pruned, making high-quality pruning even more important.

Our Rule-based Greedy Heuristic, driven by data distribution, have very little in common with the ones proposed in [19], [20] and [21]. Primarily, because the aim is to reduce the amount of data that goes over the network and in order to achieve that, we have to necessarily take into account the physical data distribution (shard keys etc.). Our heuristics also differ in the fact that the previous work [19] and [20] employed the heuristics over an Enumerator that supports bushy trees and that created more opportunities for applying their heuristics whereas our Enumerator only accepts left-deep trees and that constrains the applicability of several heuristics proposed in [19] and [20].

4. EXPERIMENTS

For our experimental comparisons, we ran MemSQL on Amazon EC2 [15], using the MemSQL row-store only. For TPC-H and TPC-DS scale factor 10, we used a cluster of 1 aggregator and 4 leaves, while for scale factor 100 we used a cluster of 1 aggregator and 32 leaves. All instances were m3.2xlarges, with 8 virtual CPU cores (on 2.5 GHz Intel Xeon E5-2670 v2 processors), 30 GB RAM, and two 80GB SSDs.

In order to prove the effectiveness of the Rule-Based Greedy Heuristic, we ran queries from the TPC-H benchmark. Figure 2 shows the percentage of states that were pruned for six queries from the benchmark. We also mention the number of tables for each query to indicate the benefit of the heuristic with the increase in the number of tables.

Query	Tables	Pruned %
Q3	3	25.00%
Q5	6	61.46%
Q7	6	72.92%
Q8	8	95.80%
Q9	6	84.90%
Q10	4	62.50%

Figure 2: Percentage of pruned states for Greedy

It can be seen that the heuristic allows us to prune a majority of the states for Q5, Q7, Q8, Q9 and Q10. Also, the pruning percentage is consistently on the higher side as the number of tables in the query increases. In the case of Q3, there were only twelve states and we pruned only three, thus making the pruning percentage 25.00%. But Q3 only had three tables, and it did not matter a lot even if we prune less since the total number of states was only twelve.

It is worthwhile to mention that the aim of the pruning heuristics is to come up with a reasonably good plan that helps us prune a majority of the states; the heuristic plan does not have to be the optimal plan. In case of Q5, the cost of the optimal plan is 4,359,746 while the cost of the heuristic plan is 102,177,223. However, it still led to a good pruning percentage of 61.46%. The percentage numbers for the other queries in TPC-H are not mentioned here but for every query that has more than three tables joined, the Enumerator was able to prune at least 60% of the states and more than 80% in most of the cases.

Another experiment performed was to evaluate the performance gained by introducing bushy join plans and also the overhead of introducing bushy joins. The first metric used in this case was the actual query response time (including optimization time). The second metric was the added overhead of the Bushy Join query rewrite. We ran several queries from the TPC-DS benchmark and our heuristics enabled bushy joins for several queries. Figure 3 (Column 3) mentions the speedup in query response time over the left-deep tree plan chosen by the optimizer when bushy plans were turned off. Figure 3 (Column 2) mentions the overhead in the Rewriter when bushy joins were enabled. In all cases, the Rewriter spent a very minimal amount of time to perform the Bushy Plan generation. It can be seen that the queries get a tremendous speedup in execution at the cost of a very minimum overhead in the Rewriter. This validates our assumption that it is indeed a good idea to generate bushy joins outside of the Enumerator. In total, fourteen queries in TPC-DS benchmark were benefitted by Bushy Joins; speedups ranging from 2.5X to 10.1X.

Query	Overhead	SpeedUp (X)
Q15	13%	5.8
Q25	16%	10.1
Q46	12%	2.85

Figure 3: Bushy Join Speedup for TPC-DS

5. CONCLUSION & FUTURE WORK

In this paper, we described some of the early steps that we have taken to reduce the query optimization time for large analytical queries. We proposed generating bushy plans via query rewrite mechanism the rewrite itself triggered using heuristics that use properties of the schema and the query. We proposed new distribution aware greedy heuristics to prune out states in the distributed join order selection inside the Enumerator. We also demonstrated the effectiveness of these techniques with experimental results. Our Greedy heuristic is able to prune a majority of the states for queries that involve more tables. Our Bushy Join technique gives us huge improvements in execution time for several TPC-DS queries. The next steps for us would be to investigate parallelizing the enumeration process based on ideas in literature. We also want to refine the existing heuristics based on more customer experiences, and run the Enumerator over queries that require a huge number of tables to be joined.

6. REFERENCES

[1] P. Selinger et al "Access Path Selection in a Database Management System", *Proc. ACM SIGMOD*, 1979.

[2] R. Othayoth et al., "The Making of TPC-DS", Proc. of the 32nd International Conf. on VLDB, Seoul, S. Korea, 2006.

[3] K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization", *Proc. Of 16th VLDB, Conf.*, Brisbane, Australia, 1990.

[4] G. Moerkotte and W. Scheufele, "Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard", Technical Report, University of Mannheim 1996.

[5] R. Ahmed, R Sen, M Poess, S Chakkappen "Of Snowstorms and Bushy Trees", Proc. of the 40th *VLDB* Conference, Hangzhao, China, 2014.

[6] S. Shankar et al. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD*, 2012

[7] M A. Soliman et al. : Orca: a modular query optimizer architecture for big data. SIGMOD Conference 2014

[8] Lamb et al. :The Vertica Analytic Database: C-Store 7 Years Later. PVLDB 5(12): 1790-1801 (2012)

[9] Farber et al. SAP HANA database: data management for modern business applications. SIGMOD Record, 2011

[10] Teradata. http://www.teradata.com/Teradata-Aster-Database

[11] M. Singh and B. Leonhardi. Introduction to the IBM Netezza Warehouse Appliance. In CASCON, 2011.

[12] R. Weiss. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server, 2012.

[13] MemSQL. www.memsql.com

[14] MemSQL Two-Tiered Architecture. http://docs.memsql.com/docs/latest/intro.html#two-tiered-architecture

[15] Amazon EC2. http://aws.amazon.com/ec2/

[16] VectorWise. http://www.actian.com

[17] SAP HANA. http://hana.sap.com/abouthana.html

[18] P. Seshadri, H. Pirahesh, T.Y. Leung. "Complex Query Decorrelation", In *ICDE*, 1996

[19] Bruno et al. Polynomial Heuristics for Query Optimization. In *ICDE*, 2010.

[20] A. Swami. "Optimization of large join queries: combining heuristics and combinatorial techniques". In SIGMOD 1989

[21] L. Fegaras. "A new heuristic for optimizing large queries". In DEXA, 1998

[22] F Waas, J Hellerstein. "Parallelizing Extensible Query Optimizers". In SIGMOD 2009.

[23] Han et al. "Parallelizing Query Optimization". VLDB 2008.

[24] Heimel et al. "A First Step Towards GPU-assisted Query Optimization". In ADMS, 2012