The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database

Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, Michael Andrews
MemSQL Inc.
534 4th Street
San Francisco, CA, 94107, USA
{jack, samir, rob, raj, nika, mandrews}@memsql.com

ABSTRACT

Real-time analytics on massive datasets has become a very common need in many enterprises. These applications require not only rapid data ingest, but also quick answers to analytical queries operating on the latest data. MemSQL is a distributed SQL database designed to exploit memory-optimized, scale-out architecture to enable real-time transactional and analytical workloads which are fast, highly concurrent, and extremely scalable. Many analytical queries in MemSQL's customer workloads are complex queries involving joins, aggregations, sub-queries, etc. over star and snowflake schemas, often ad-hoc or produced interactively by business intelligence tools. These queries often require latencies of seconds or less, and therefore require the optimizer to not only produce a high quality distributed execution plan, but also produce it fast enough so that optimization time does not become a bottleneck.

In this paper, we describe the architecture of the MemSQL Query Optimizer and the design choices and innovations which enable it quickly produce highly efficient execution plans for complex distributed queries. We discuss how query rewrite decisions oblivious of distribution cost can lead to poor distributed execution plans, and argue that to choose high-quality plans in a distributed database, the optimizer needs to be distribution-aware in choosing join plans, applying query rewrites, and costing plans. We discuss methods to make join enumeration faster and more effective, such as a rewrite-based approach to exploit bushy joins in queries involving multiple star schemas without sacrificing optimization time. We demonstrate the effectiveness of the MemSQL optimizer over queries from the TPC-H benchmark and a real customer workload.

1. INTRODUCTION

An increasing number of enterprises rely on real-time analytical pipelines for critical business decisions. These pipelines ingest data into a distributed storage system, and run complex analytic queries over the latest data. For many workloads it is critical that the analytical queries are optimized and executed very quickly so that results can be provided for interactive real-time decision making. The ability to store and query huge amounts of data by scaling storage and parallelizing execution across distributed clusters with many nodes enables dramatic performance improvements in execution times for analytical data workloads. Several other industrial database systems such as SAP HANA [3], Teradata/Aster, Netezza [15], SQL Server PDW [14], Oracle Exadata [20], Pivotal GreenPlum [17], Vertica [7], and VectorWise [21] have gained popularity and are designed to run analytical queries very fast.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

1.1 Overview of MemSQL

MemSQL is a distributed memory-optimized SQL database which excels at mixed real-time analytical and transactional processing at scale. MemSQL can store data in two formats: an in-memory row-oriented store and a disk-backed column-oriented store. Tables can be created in either rowstore or columnstore format, and queries can involve any combination of both types of tables. MemSQL takes advantage of in-memory data storage with multiversion concurrency control and novel memory-optimized lock-free data structures to enable reading and writing data highly concurrently, allowing real-time analytics over an operational database. MemSQL's columnstore uses innovative architectural designs to enable real-time streaming analytical workloads with low-latency queries over tables with ongoing writes [16]. Along with its extremely scalable distributed architecture, these innovations enable MemSQL to achieve sub-second query latencies over large volumes of changing 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's distributed architecture is a shared-nothing architecture (nodes in the distributed system do not share memory, disk or CPU) with two tiers of nodes: scheduler nodes (called *aggregator* nodes) and execution nodes (called *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.

User data in MemSQL is distributed across the cluster in two ways, selected on a per-table basis. For *Distributed* tables, rows are hash-partitioned, or sharded, on a given set of columns, called the *shard key*, across the leaf nodes. For *Reference* tables, the table data is replicated across all nodes. Queries may involve any combination of such tables.

In order to execute a query, the aggregator node converts the input user query into a distributed query execution plan (DQEP). The distributed query execution plan is a series of DQEP Steps, operations which are executed on nodes across the cluster which may include local computation and data movement via reading data from remote tables on other leaf nodes. MemSQL represents these DQEP Steps using a SQL-like syntax and framework, using innovative SQL extensions called *RemoteTables* and *ResultTables*. These enable the MemSQL Query Optimizer to represent DQEPs using a SQL-like syntax and interface. We discuss *RemoteTables* and *ResultTables* in more detail later.

Query plans are compiled to machine code and cached to expedite subsequent executions. Rather than cache the *results* of the query, MemSQL caches a *compiled query plan* to provide the most efficient execution path. The compiled query plans do not pre-specify values for the parameters, allowing MemSQL to substitute values upon request, and enabling subsequent queries of the same structure to run quickly, even with different parameter values.

1.2 Query Optimization in MemSQL

The goal of the query optimizer is to find the best query execution plan for a given query by searching a wide space of potential execution paths and then selecting the plan with the least cost. This requires the optimizer to be rich in query rewrites and to be able to determine the best execution plan based on a cost model of query execution in the distributed database. Many of the queries in MemSQL's customer workloads are complex queries from enterprise real-time analytical workloads, involving joins across star and snowflake schemas, sorting, grouping and aggregations, and nested sub-queries. These queries require powerful query optimization to find high-quality execution plans, but query optimization must also be fast enough so that optimization time does not slow down query runtimes too much. Many of the queries run in these workloads are ad-hoc and therefore require query optimization; and even non-ad-hoc queries can often require query optimization, for example due to significant changes in data statistics as new data is ingested. These queries often must be answered within latencies measured in seconds or less, despite being highly complex and resource intensive.

Designing and developing a query optimizer for a distributed query processing system is an extremely challenging task. MemSQL is a high-performance database, built from the ground up with innovative engineering such as memory-optimized lock-free skip-lists and a columnstore engine capable of running real-time streaming analytics. Because of the unique characteristics of the MemSQL query execution engine, and because the real-time workloads MemSQL runs often mean that the time budget for optimizing a query is very limited, we decided to also build the query optimizer from scratch. Reusing any existing optimizer framework would not best address the goals and challenges in the MemSQL context, while it would also mean inheriting all the shortcomings of the framework and integration challenges. Despite the technical and engineering challenges, we developed a query optimizer rich in features and capable of producing high quality query execution plans across a variety of complex enterprise workloads.

The MemSQL Query Optimizer is a modular component 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 Rewriter decides whether to apply the rewrite using heuristics or cost; the cost being the distributed cost of running the query. The Rewriter intelligently applies certain rewrites in a top-down fashion while applying others in a bottom-up manner, and 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 as well as local join order and access path selection. It considers a wide search space of various execution alternatives and selects the best plan, based on the cost models of the database operations and the network data movement operations. The Enumerator is also 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. The Planner uses SQL extensions called *RemoteTables* and *ResultTables* to represent a series of Data Movement Operations and local SQL Operations using a SQL-like syntax and interface, making it easy to understand, flexible, and extensible.

1.3 Contributions

In this paper, we make the following important contributions:

- We claim that if the cost-based query rewrite component is not aware of distribution cost, then the optimizer runs the risk of making poor decisions on query rewrites in a distributed setting. We solve the problem in the MemSQL query optimizer by calling the Enumerator within the Rewriter to cost rewritten queries based on its distributed-aware cost model.
- We enhanced the Enumerator to enumerate very fast by extensively pruning the operator order search space. We implemented new heuristics that are distribution aware and use them to prune out states.
- We propose a new algorithm that analyzes the join graph and discovers bushy patterns; i.e. it identifies parts of the join graph that could be advantageous to run as bushy joins and applies them as a query rewrite mechanism.

The rest of the paper is organized as follows. Section 2 provides an overview of MemSQL query optimization and the structure of DQEPs. In Section 3, we deep dive into the details of the Rewriter. Section 4 introduces a new algorithm to efficiently discover bushy join patterns. Section 5 provides more insights into the Enumerator, and Section 6 describes the Planner. We describe our experimental results in Section 7. In Section 8, we briefly summarize related work done in the area of designing optimizers for distributed databases and also related work on reducing optimization time. Finally, we conclude in Section 9.

2. OVERVIEW OF MEMSQL QUERY OPTIMIZATION

When a user query is sent to MemSQL, the query is parsed to form an operator tree. The operator tree is the input to the query optimizer and it goes through the following steps:

- The Rewriter analyzes the operator tree and applies the relevant query rewrites to the operator tree. If a particular rewrite is beneficial, it will apply it and change the operator tree to reflect the rewritten query. If a rewrite needs to be cost-based, it will cost the original operator tree and the rewritten operator tree and will pick the tree that has a lower cost.

- The operator tree is then sent to the Enumerator. The Enumerator uses a search space exploration algorithm with pruning. It takes into account the table statistics and the cost of the distributed operations such as broadcasting and partitioning to generate the best join order for the input query. The output of the enumerator is an operator tree where the tree nodes are annotated with directives for the Planner.
- The Planner consumes the annotated operator tree that is produced by the Enumerator and generates the distributed query execution plan (DQEP), consisting of a series of *DQEP Steps*, SQL-like steps that can be sent as queries over the network to be executed on nodes across the cluster. The *DQEP Steps* are executed simultaneously on the leaves, streaming data whenever possible. Each step runs in parallel on all partitions of the database.

2.1 DQEP Example

Using the well-known TPC-H schema as an example, let us assume that the *customer* table is a distributed table that has a shard key on *c_custkey* and the *orders* table is also a distributed that has a shard key on *o_orderkey*. The query is a simple join between the two tables with a filter on the orders table.

```
SELECT c_custkey, o_orderdate
FROM orders, customer
WHERE o_custkey = c_custkey
  AND o_totalprice < 1000;
```

The query above is a simple join and filter query and hence, the Rewriter will not be able to apply any query rewrites directly over this query and the operator tree corresponding to the original input query is fed to the Enumerator. It can be seen that the shard keys of the tables do not exactly match with the join keys (*orders* is not sharded on *o_custkey*), and therefore, there needs to be a data movement operation in order to perform the join. The Enumerator will pick a plan based on the statistics of the table, number of nodes in the cluster, etc. One possible plan choice is to repartition *orders* on *o_custkey* to match *customer* sharded on *c_custkey*. The Planner converts this logical plan choice into an execution plan consisting of the following *DQEP Steps*:

```
(1) CREATE RESULT TABLE r0
      PARTITION BY (o_custkey)
    AS
      SELECT orders.o_orderdate as o_orderdate,
             orders.o_custkey as o_custkey
      FROM   orders
      WHERE  orders.o_totalprices < 1000;

(2) SELECT customer.c_custkey as c_custkey,
           r0.o_orderdate as o_orderdate
    FROM   REMOTE(r0(p)) JOIN customer
    WHERE  r0.o_custkey = customer.c_custkey
```

In this DQEP, there are two SQL-like statements which are executed using our ResultTable and RemoteTable SQL extensions. The first of these steps operates locally on each partition of the orders table, filtering and then partitioning the data on the join column, *o_custkey*, and streaming the result into the ResultTable *r0*. It can be seen that the Planner is able to push the predicate associated with the orders table down into the first DQEP step, to be executed before the data is moved.

The second statement in the DQEP draws from a distributed table, indicated by the REMOTE keyword. This is the part of the DQEP that moves the data prepared in the first step across the network. Each partition reads the partitions of *r0* which match the local partition of *customer*. Then, the join between the result of the previous step and the *customer* table is performed across all partitions. Every leaf node returns its result set to the aggregator node, which is responsible for combining and merging the result sets as needed and delivering them back to the client application.

2.2 Query Optimization Example

In this section, we illustrate the steps in the optimization and planning process for an example query. TPC-H Query 17 is an interesting example in that it shows interesting aspects of all three components of the optimizer. In this example, *lineitem* and *part* are distributed rowstore tables hash-partitioned on *l_orderkey* and *p_partkey*, respectively. The query is:

```
SELECT sum(l_extendedprice) / 7.0 as avg_yearly
FROM   lineitem,
       part
WHERE  p_partkey = l_partkey
   AND p_brand = 'Brand#43'
   AND p_container = 'LG PACK'
   AND l_quantity < (
      SELECT 0.2 * avg(l_quantity)
      FROM   lineitem
      WHERE  l_partkey = p_partkey)
```

Rewriter: The Rewriter applies all the query rewrites and comes up with the following rewritten query, in which the scalar subquery has been converted to a join, and we have pushed the join with *part* down into the subquery, past the group by. This is beneficial because it enables more flexible join plan and DQEP. There is no way to efficiently execute the original query without transforming it, because the correlating condition of the subselect does not match the shard key of *lineitem*. Therefore, evaluating the correlated subselect would require doing a remote query for each row of *part*, which is obviously not performant, or first repartitioning *lineitem* on *l_partkey*, which is expensive because *lineitem* is large. In contrast, the transformed query can be executed efficiently by starting with *part*, which has a selective filter, and seeking into *lineitem* for the joins, as determined by the Enumerator.

```
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
```

Enumerator: The Enumerator chooses the cheapest join plan and annotates each join with data movement operations and type. The best plan is to broadcast the filtered rows from *part* and from *sub*, because the best alternative would involve reshuffling the entire *lineitem* table, which is far larger and thus more expensive. The query plan, with some simplifications, is:

```
Project [s2 / 7.0 AS avg_yearly]
Aggregate [SUM(1) AS s2]
Gather partitions:all
Aggregate [SUM(lineitem_1.l_extendedprice) AS s1]
Filter [lineitem_1.l_quantity < s_avg]
NestedLoopJoin
|---IndexRangeScan lineitem AS lineitem_1,
|   KEY (l_partkey) scan:[l_partkey = p_partkey]
Broadcast
HashGroupBy [AVG(l_quantity) AS s_avg]
            groups:[l_partkey]
NestedLoopJoin
|---IndexRangeScan lineitem,
|   KEY (l_partkey) scan:[l_partkey = p_partkey]
Broadcast
Filter [p_container = 'LG PACK' AND
        p_brand = 'Brand#43']
TableScan part, PRIMARY KEY (p_partkey)
```

Planner: The planner creates the DQEP according to the chosen query plan, consisting of a series of SQL statements with *ResultTables* and *RemoteTables*. Playing to the strengths of *ResultTables*, the entire query can be streamed since there are no pipeline-blocking operators. The group-by can also be streamed by taking advantage of the existing index on the *p_partkey* column from the *part* table. For clarity, we show a simplified DQEP, which omits the optimizations for broadcasts described in Section 6.2.1.

```
CREATE RESULT TABLE r0 AS
SELECT p_partkey
FROM   part
WHERE  p_brand = 'Brand#43'
AND p_container = 'LG PACK';

CREATE RESULT TABLE r1 AS
SELECT 0.2 * Avg(l_quantity) AS s_avg,
       l_partkey as s_partkey
FROM   REMOTE(r0),
       lineitem
WHERE  p_partkey = l_partkey
GROUP  BY l_partkey;

SELECT Sum(l_extendedprice) / 7.0 AS avg_yearly
FROM   REMOTE(r1),
       lineitem
WHERE  p_partkey = s_partkey
   AND l_quantity < s_avg
```

3. REWRITER

The MemSQL query optimizer considers a wide variety of query rewrites, which convert a given SQL query to another semantically equivalent SQL query, which may correspond to a better performing plan. The Rewriter locates opportunities to apply a query transformation, decides based on heuristics or cost estimates whether the rewrite is beneficial, and if so applies the transformation to yield a new query operator tree.

3.1 Heuristic and Cost-Based Rewrites

A very simple example of a query transformation performed by the Rewriter is the *Column Elimination* transformation, which removes any projection columns that are never used, thus saving on computation, I/O, and network resources. This transformation is always beneficial, so the Rewriter applies the transformation whenever semantically valid. On the other hand, the *Group-By Pushdown* transformation, which modifies a query by reordering a group by before a join to evaluate the group by earlier, may or may not be advantageous depending on the sizes of the joins and the cardinality of the group by, so deciding whether to apply this transformation requires making cost estimates.

We also make use of heuristics in many rewrite decisions. For example, *Sub-Query Merging* generally merges subselects whenever possible. However, when very large numbers of tables are being joined together under a number of simple views, merging all the subselects would result in a single large join of all these tables, which could be expensive for the Enumerator to effectively optimize. Merging these subselects discards information about the structure of the join graph, which may be helpful for optimizing the join even though it carries no additional semantic information. For example, in a snowstorm query, which includes multiple large fact tables and their associated dimension tables, the input query may contain views corresponding to joins of particular fact tables with their associated dimension tables, which can be efficiently evaluated and then joined together in a bushy join plan. We can use heuristics to detect this type of situation and avoid merging all the views in such cases. Of course, this restricts the space of possible join orders we can consider, which is only acceptable when we expect the join tree structure represented by the subselects to roughly correspond to the optimal join tree. In these situations, we can find a close-to-optimal join tree without needing to pay the high cost of join enumeration over the full, large set of tables including searching for bushy joins.

3.2 Interleaving of Rewrites

The Rewriter applies many query rewrites, many of which have important interactions with each other, so we must order the transformations intelligently, and in some cases interleave them. For example, consider *Outer Join to Inner Join* conversion, which detects outer joins that can be converted to inner joins because a predicate later in the query rejects NULLs of the outer table, and *Predicate Pushdown*, which finds predicates on a derived table which can be pushed down into the sub-select. Pushing a predicate down may enable Outer Join to Inner Join conversion if that predicate rejects NULLs of the outer table. However, *Outer Join to Inner Join* conversion may also enable Predicate Pushdown because a predicate in the ON condition of a left outer join can now potentially be pushed inside the right table, for example. Therefore, to transform the query as much as possible, we interleave the two rewrites: going top-down over each select block, we first apply *Outer Join to Inner Join* conversion, and then *Predicate Pushdown*, before processing any subselects.

On the other hand, some rewrites such as the bushy join rewrite discussed later are done bottom-up, because they are cost-based and their cost can be affected by the rewrites chosen and plans generated for subselects in the subtree.

3.3 Costing Rewrites

We can estimate the cost of a candidate query transformation by calling the Enumerator, to see how the transformation affects the potential execution plans of the query tree, including join orders and group-by execution methods of any affected select blocks. Note that the Enumerator only needs to re-cost those select blocks which are changed, as we can reuse the saved costing annotations for any unchanged select blocks.

It is important that the Enumerator determines the best execution plan taking into account data distribution, including when called by the Rewriter for the purposes of cost-based rewrites, because many query rewrites can potentially alter the distributed plan, including by affecting which operators like joins and groupings can be co-located, and which and how much data needs to be sent across the network. If the Rewriter makes a decision on whether to apply a rewrite based on a model that is not aware of distribution cost, the optimizer can potentially chose inefficient distributed plans.

Let's consider a relatively simple example to illustrate the point. Let us consider two tables *T1 (a, b)* and *T2 (a, b)* which are sharded on the columns *T1.b* and *T2.a,* respectively, and with a unique key on column *a* for *T2*:

```
CREATE TABLE T1 (a int, b int, shard key (b))
CREATE TABLE T2 (a int, b int, shard key (a),
                 unique key (a))
```

Consider the following query Q1:

```
Q1: SELECT sum(T1.b) AS s FROM T1, T2
      WHERE T1.a = T2.a
      GROUP BY T1.a, T1.b
```

This query can be rewritten to with the *Group-By Pushdown* transformation, which reorders the group-by before the join, as shown in the transformed query Q2:

```
Q2: SELECT V.s from T2,
      (SELECT a,
              sum(b) as s
        FROM T1
        GROUP BY T1.a, T1.b
      ) V
      WHERE V.a = T2.a;
```

Let $R_1 = 200{,}000$ be the rowcount of *T1* and $R_2 = 50{,}000$ be the rowcount of *T2*. Let $S_G = \frac{1}{4}$ be the fraction of rows of *T1* left after grouping on *(T1.a, T1.b)*, i.e. $R_1 S_G = 50{,}000$ is the number of distinct tuples of *(T1.a, T1.b*). Let $S_J = \frac{1}{10}$ be the fraction of rows of *T1* left after the join between *T1.a* and *T2.a* (note that each matched row of *T1* produces only one row in the join since *T2.a* is a unique key). Assume the selectivity of the join is independent of the grouping, i.e. any given row has a probability S_J of matching a row of *T2* in the join. So the number of rows after joining *T1* and *T2* on *T1.a = T2.a* is $R_1 S_J = 20{,}000$, and the number of rows after both the join and the group-by of Q1 is $R_1 S_J S_G = 5{,}000$.

Assume seeking into the unique key on *T2.a* has a lookup cost of $C_J = 1$ units, and the group-by is executed using a hash table with an average cost of $C_G = 1$ units per row. Then the costs of the query execution plans for Q1 without the *Group-By Pushdown* transformation, and Q2 with the transformation, without taking distribution into account (i.e. assuming the entire query is executed locally) are:

$$Cost_{Q1} = R_1 C_J + R_1 S_J C_G = 200{,}000 C_J + 20{,}000 C_G = 220{,}000$$

$$Cost_{Q2} = R_1 C_G + R_1 S_G C_J = 200{,}000 C_G + 50{,}000 C_J = 250{,}000$$

For these example values of C_G and C_J as well as many other plausible values, $Cost_{Q1} < Cost_{Q2}$. Therefore, in the context of a non-distributed query or a cost model that does not take distribution into account, the rewrite would be considered disadvantageous and we would execute the plan Q1.

However, if we want to run the query in a distributed setting, we need to move data from at least one of the tables to execute the join. Since *T2* is sharded on *T2.a*, but *T1* is not sharded on *T1.a*, we can best compute this join by reshuffling *T1* or broadcasting *T2*, depending on their sizes. Assuming the size of the cluster is large enough, e.g. 10 nodes, and given that *T2* is not much smaller than *T1*, reshuffling *T1* on *T1.a* is a cheaper plan than broadcasting *T2* for the join.

The group-by can be executed after the join in plan Q1 without any further data movement, since the result of the join is partitioned on *T1.a*, so all rows of each group are located on the same partition. The group-by can also be executed before the join in plan Q2 without any data movement, because *T1* is sharded on *T1.b*, so all groups are also located on the same partition.

In the distributed setting, we would incur an additional cost of shuffling all rows of *T1* for plan Q1. For plan Q2, the plan would be to first execute the group-by locally on each partition, reshuffle the result, and finally join against *T2*, so only $T_1 S_G$ rows must be reshuffled since the group-by reduces the rowset.

The distributed query execution plans in MemSQL are:

```
Q1:
Gather partitions:all
Project [r0.s]
NestedLoopJoin
|---IndexSeek T2, UNIQUE KEY (a) scan:[a = r0.a]
Repartition AS r0 shard_key:[a]
HashGroupBy [SUM(T1.b) AS s] groups:[T1.a, T1.b]
TableScan T1

Q2:
Gather partitions:all
Project [r0.s]
HashGroupBy [SUM(r0.b) AS s] groups:[r0.a, r0.b]
NestedLoopJoin
|---IndexSeek T2, UNIQUE KEY (a) scan:[a = r0.a]
Repartition AS r0 shard_key:[a]
TableScan T1
```

Assuming the average cost of executing a reshuffle, which includes e.g. network and hash evaluation costs, is $C_R = 3$ units per row, the costs are:

$$\begin{aligned} Cost_{Q1} &= R_1 C_R + R_1 C_J + R_1 S_J C_G \\ &= 200{,}000\,(C_R + C_J) + 20{,}000\, C_G \\ &= 620{,}000 \end{aligned}$$

$$\begin{aligned} Cost_{Q2} &= R_1 C_G + R_1 S_G C_R + R_1 S_G C_J \\ &= 200{,}000 C_G + 50{,}000 (C_R + C_J) \\ &= 400{,}000 \end{aligned}$$

For these example parameter values, $Cost_{Q1} > Cost_{Q2}$ because the reshuffle significantly impacts the cost of the plans. This is especially likely to be the case in clusters with slower network where network costs may often dominate the cost of a query. In an Amazon EC cluster, we found that plan Q2 runs around 2x faster than Q1 in MemSQL. A rewrite decision based on a distribution-oblivious cost model would have incorrectly chosen Q1.

The example query Q1 used is a very simple query involving a join and a group-by. Many more complex queries that undergo a series of mutually interacting and interleaved query rewrites would also require the Enumerator to cost plans taking data distribution into account.

Comparison to PDW: Microsoft PDW's Query Optimizer [14] performs distributed costing for join order enumeration, but the query rewrites are all applied in the single-node SQL Server optimizer. The SQL Server optimizer (in a single SQL Server instance) uses the "shell database" that contains the statistical information of the tables, performs the cost-based rewrites and generates the space of execution alternatives (called MEMO) that PDW consumes. Without distributed costing inside the SQL Server optimizer, PDW will produce inefficient distributed execution plans where the query rewrites affect the distributed cost significantly.

4. BUSHY JOINS

As discussed in the literature [8][10], searching all possible join plans, including bushy join plans, as part of the join enumeration makes the problem of finding the optimal join permutation extremely costly and time-consuming. As a result, many database systems do not consider bushy joins, limiting their search to left-deep or right-deep join trees. However, for many query shapes, such as shapes involving multiple star or snowflake schemas, bushy join plans are critical for achieving good execution performance, with massive speedups compared to the best non-bushy join plan.

Our strategy for finding good join plans, which may be bushy in nature, without sacrificing optimization time by paying the cost of searching all bushy join plans, is a heuristic-based approach which considers only promising bushy joins instead of all possible cases. We look for common query shapes that benefit from bushy plans and introduce bushiness via the framework of a query rewrite. In our previous work [12], we demonstrated the effectiveness of this general approach. A direct advantage of generating bushy plans in this way is that we would only consider bushy plans when there is a potential benefit as determined by the heuristics, which allows narrow and targeted exploration of the bushy plan space. As we started analyzing more complex query workloads from real world customers, we realized that while generating bushy join plans via query rewrites was a good idea, the heuristics that we used to generate the candidate plans and the rewrite method itself need to be refined and cover more generic cases. We will discuss our new method for finding bushy join plans which improves on the previous approach.

4.1 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 derived table using the query rewrite mechanism and using the derived table as the right side of the join. The Enumerator works as usual; it optimizes the derived table like any other table in the join. Once a new derived table is introduced as part of the query rewrite, the Rewriter calls the Enumerator to cost the rewritten query, and then based on the cost, determines whether to retain the newly introduced subselect. The Bushy Plan rewrite clearly must make cost-based decisions because comparing two bushy plan options involves considering join execution methods, distribution methods, etc. However, the choices of which plans to consider is heuristic-based to enable this approach to efficiently explore candidate plans which are likely to be beneficial.

4.2 Bushy Plan Heuristics

Using query rewrite mechanism, it is possible to consider promising bushy joins by forming 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 select block. By placing a derived table 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 algorithm to generate bushy join plans traverses the join graph and looks at the graph connections to determine whether any such bushy subselects are possible and what tables may be part of those subselects. For every such subselect that could be potentially formed, it calls the Enumerator to determine the cost in order to decide which candidate option is better. The basic algorithm is as follows:

1. Collect the set of tables in the join and build a graph of the tables in which each table is a vertex and each join predicate between a pair of tables corresponds to an edge between their vertices.
2. Identify candidate *satellite* tables, which are tables with at least one selective predicate on them, such as a predicate of the form `column = constant` or `column IN (constant,…)`.
3. Out of the list of candidate satellite tables, identify the *satellite* tables, which are the tables connected to only other table in the graph (although possibly with multiple join predicates).
4. Identify *seed* tables, which are tables that are connected to at least two distinct tables, at least one of which is a *satellite* table. (Observe that no *satellite* table can be adjacent to more than one *seed* table because of the requirement that *satellite* tables are connected to only one table.)
5. For each *seed* table:
    a) Use the costing mechanism to compute the cost C_1 of the current plan.
    b) Create a derived table containing the *seed* table joined to its adjacent *satellite* tables. Note that some SQL operators may prevent some satellite tables from being moved inside the subselect, in which case move as many as possible.
    c) Apply the *Predicate Pushdown* rewrite followed by the *Column Elimination rewrite* to ensure that any predicate in the outer select which can be evaluated in the inner select is moved inside and that no columns are provided by the inner select which are not needed in the outer select.
    d) Compute the new cost C_2 of the modified plan. If $C_1 < C_2$, discard the changes made in steps (b) and (c), and otherwise keep them.

Our strategy is very generic and does not depend on table cardinalities and/or selectivities to identify possible bushy combinations. In a snowstorm-type query, this will find fact tables, which are often 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. 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 [9] 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 …
ORDER BY …
```

We will focus on the join and ignore the group-by, aggregations, and order-by in our discussion of this example.

The join graph is shown in Figure 1. The tables with filters are colored green. There are three fact tables (*store_sales*, *store_returns*, and *catalog_sales*), each joined against one dimension table with a filter (*date_dim*). All of the joins are on a primary key or another highly selective key.

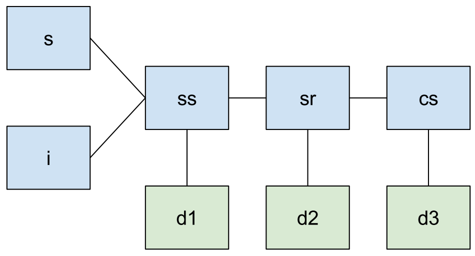
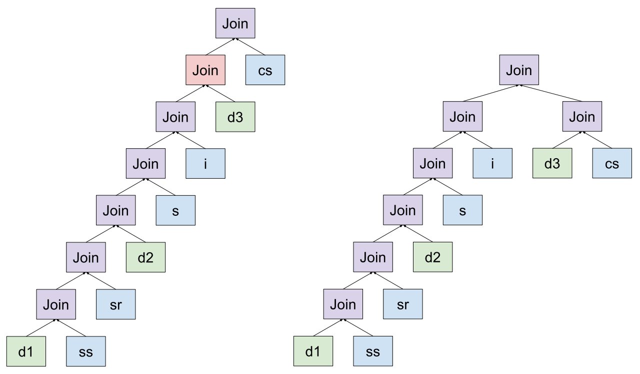


Figure 1: Join graph for TPC-DS q25.

In a distributed setting, the best left-deep join plan chosen by the Enumerator is *(d1, ss, sr, d2, s, i, d3, cs)*, shown in Figure 2a. All of these joins have selective join conditions except for one: the Join node colored red, when we join *d3*, is a Cartesian product join, because *d3* only has join predicates with *cs*. This is expensive, but given the restriction to left-deep join trees it is the better alternative compared to first joining *cs* without having any of the filtering that comes from the single-table filters on *d3*.



(a) Left-deep join **(b) Bushy join**

Figure 2: Join trees for TPC-DS q25.

Our algorithm works as follows. We first build the join graph and then identify the candidate satellite tables, which in this case are *{d1, d2, d3}* since each of them has one selective predicate. We then identify the *satellite* tables, which are connected to more than one table in the join graph; in this example, all the three tables are connected with only table and so the satellite tables are *{d1, d2, d3}*. We now identity the set of *seed* tables, the tables connected to at least two distinct tables, one of which must be a satellite table. Our seed tables are *ss* (connected to satellite *d1* and *sr*), *sr* (connected to satellite *d2* and *ss*), and *cs* (connected to satellite *d3* and *sr*).

The Rewriter tries to cost each combination and uses the Enumerator to cost every rewritten combination. The final bushy join order that is chosen is *(d1, ss, sr, d2, s, i, (d3, cs))*, shown in Figure 2b. It can be seen that out of all candidate *seed* tables, bushiness was introduced only for *cs* and its *satellite* tables. 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 bushy join plan is represented with a derived table as follows:

```
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
```

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 [1]. However, the methods used to achieve the same in [1] and in our framework are totally different from each other. The mechanism in [1] 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 only traverses the join graph and only looks at number of connections in the graph to identity the set of *seed* tables. In [1], the *fact* table always has to have an effective (after applying filters) cardinality that is more than a minimum threshold. Our solution does not have any such restriction; in fact, we never look at cardinalities and it could easily happen that the effective cardinality of the *seed* table is less than that of the *satellite* table. Also in [1], each *fact* table needs a join edge with at least one other *fact* table; in our case, *seed* tables need not necessarily join with another *seed* table. These fundamental differences enable us to cover a lot of more generic cases that might benefit from bushy join plans. We will cover one such case from a real customer workload in our Experiments Section.

5. ENUMERATOR

The Enumerator is the backbone of the MemSQL Query optimizer. It is the component that connects the Rewriter and the Planner; the Rewriter feeds query operator trees into the Enumerator for the Enumerator to determine the execution plan, including distributed data movement decisions and join orders, and annotate the operator tree accordingly. The Rewriter is the component that does a lot of logical optimization leading to the Enumerator; which does the physical optimization of the query. The Enumerator needs to look at cost, table and network statistics, query characteristics etc. to perform the physical optimization. Just like any other industry-strength query optimizers, the Enumerator has a costing model and considers a wide search space of various execution alternatives to select the best join order. The Enumerator is built on the assumption that parallelizing the best serial plan is not good enough for distributed query processing. [14] discusses this claim as well, and we also conducted our own set of experiments over benchmarks like TPC-H and TPC-DS and several customer workloads to find illustrative examples of the need for join choices to be distribution aware and the need for the optimizer, including the enumeration algorithm, to take into account the cost of the data movement operations to come up with the best DQEP. A key focus of the Enumerator is on choosing high-quality distributed query plans, including taking advantage of co-located (bucketed) joins when possible and minimizing data distribution costs. It is also interesting to note that the Enumerator must handle physical optimization for queries involving columnstore and rowstore tables in any combination. This requires searching execution plans appropriate for both storage formats and modeling them in the cost model.

5.1 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. 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 [11] style dynamic programming based join order enumerator with *interesting properties*. System-R style optimizers have the notion of *interesting orders* to help take advantage of physical properties like sort order etc. The MemSQL Optimizer Enumerator employs an *interesting property* of *sharding distribution,* e.g. the set of columns by which data is sharded across the cluster. Interesting shard keys that can be considered are (1) predicate columns of equality joins and (2) grouping columns. In the dynamic programming, we keep track of the best cost for each candidate join set that yields data distributed on each interesting sharding. By examining plans that yield different sharding distributions, we are able to find plans that later take advantage of the sharding distribution. Even though they may be more expensive for an initial part of the join, they may end up cheaper by avoiding the need for a reshuffle or broadcast later.

5.2 Distributed Costing

The cost model for the distributed optimizer consists of the cost model for local SQL relational operations like joins, grouping etc. and the data movement operations. For distributed queries which require non-co-located joins because the shard keys of tables involved do not match the join keys, the data movement processing times are often a dominant component of the query execution time. The cost model for data movement operations assumes that every query runs in isolation; the hardware is homogenous across the cluster; and the data is uniformly distributed across all the nodes. These assumptions are not new and are discussed in [14]; they are clearly not ideal in all cases but are helpful simplifications which work well in most cases.

Data Movement Operations: The data movement operations supported by the distributed query execution engine are:

- Broadcast: Tuples are broadcasted from each leaf node to all other leaf nodes.
- Partition (also called Reshuffle): Tuples are moved from each leaf node to a target leaf node based on a hash of a chosen set of distribution columns.

Data Movement Operation Costs: The costs for data movement operations include the network and computational costs of sending data over the network, as well as other computational costs required for the operations such as hashing costs for reshuffles. The cost is estimated as follows:

- Broadcast: $R\ D$
- Partition: $\frac{1}{N}(R\ D + R\ H)$

where R is the number of rows which need to be moved, D is the average cost per row of moving data (which depends on the row size and network factors), N is the number of nodes, and H is the cost per row of evaluating hashes for partitioning.

5.3 Fast Enumeration

As mentioned earlier, in many real-time analytics workloads, queries need to finish execution within a few seconds or less than a second, and therefore require the optimizer to not only produce the best distributed execution plan but also produce it fast (with low query optimization latency) so that optimization time does not become too expensive a component of query latency. To cost query rewrite combinations, the Rewriter calls the Enumerator and this requires the enumeration to be very fast. This requires the Enumerator to use pruning techniques to filter our plans. In the world of distributed query optimization, any pruning technique that is employed needs to be aware of data distribution; a heuristic technique based on table cardinalities, schema and selectivities is not good enough. The MemSQL Enumerator uses several advanced pruning techniques to enumerate operator orders, thus making the process very fast. A discussion of those techniques is available in [12].

6. PLANNER

The role of the planner is to convert the rewritten and enumerated query into a physical execution plan. The Planner converts the output of the Enumerator to a physical execution plan that can be distributed to all the nodes in a cluster. It consumes the annotated operator tree that is produced by the Enumerator and generates the *DQEP Steps* that are required to execute the query. *DQEP Steps* are SQL-like steps that can be sent as query text over the network. The *DQEP Steps* are executed simultaneously on the leaves, streaming data whenever possible. Each step runs in parallel on all partitions of the database.

6.1 Remote Tables and Result Tables

In MemSQL, each step of a DQEP may consist of data movement and local computation. Because SQL is both easy to reason about and already supported in the engine, all communication between nodes in a MemSQL cluster is done via the SQL interface. This transparently enables features such as node-level optimization and plan extensibility. MemSQL implements and employs two important SQL extensions to support data movement and node-level computation.

6.1.1 Remote Tables

In a simple query, the only inter-node communication required is from the leaf nodes to the aggregator node. Wherever possible, filters and grouping expressions are pushed down into the leaf queries to increase parallelism. When more complex queries are considered, data movement between leaf nodes is required. The SQL extension *RemoteTables* allows leaf nodes to query all partitions in the same way an aggregator would. Consider the following query:

```
SELECT facts.id, facts.value
FROM   REMOTE(facts) as facts
WHERE  facts.value > 4
```

This query can run on any leaf in the MemSQL cluster. The REMOTE keyword indicates to the engine that the relation is comprised of tuples from all partitions of the *facts* table, rather than only the local partition. In the interest of exposing an explicit syntax for the planner to use, the filter is not delegated to each other partition as it would have been in an aggregator query. For the planner to indicate precisely when particular operations should be computed, MemSQL employs an extension called *ResultTables*.

6.1.2 Result Tables

Using *RemoteTables* alone would be enough to evaluate any relational expression on a cluster of nodes, but it has certain drawbacks. In a fully distributed query, each partition of the database will need to query a *RemoteTable*. However, with each partition querying all other partitions, a lot of work will be repeated quadratically. Even table scans can be expensive when repeated for each partition of the database. To share computational work, MemSQL nodes can store local *ResultTables*. A result table is a temporary result set, which stores one partition of a distributed intermediate result. These tables are defined using a SQL SELECT statement, and are read-only after definition. In this way, the planner can delegate work to the partitions with finer granularity. In the example above, the planner could arrange for each partition to run the following query on each partition before computing the final select:

```
CREATE RESULT TABLE facts_filtered
AS SELECT facts.id, facts.value
          FROM facts
          WHERE facts.value > 4
```

The *RemoteTable* can select from this new relation instead of the original base table to avoid running the filter on the receiving end.

6.2 Using Remote Tables and Result Tables in DQEPs

To fully represent a DQEP, the planner must lay out a series of data movement and computational steps in SQL. In a MemSQL plan, this is accomplished by chaining these operations together within *ResultTables*. Each stage of the DQEP is represented by a compute step which optionally pulls rows from another stage of the execution, using *ResultTables* to represent intermediate result sets. In this way, complex data flow and computation can be expressed using only these SQL extensions. However, *ResultTables* need not be materialized as actual tables, and for some query execution plans they are simply an abstraction and the underlying execution can stream rows from the writer to the reader without writing to a physical table.

6.2.1 Broadcasts

Consider the example query

```
SELECT * FROM x JOIN y WHERE x.a = y.a AND x.b < 2
AND y.c > 5
```

where table *x* is sharded on *a* but table *y* is not (if they are both sharded on *a*, then the optimizer would take advantage of that to do a colocated join). In this case, depending on the relative sizes of the tables after applying filters, the best plan may be to either broadcast *x* or reshuffle *y* to match the sharding of *x*. If table *x* is much smaller than table *y* after the relevant filters, the best plan would be to broadcast *x* after the filter. This can be executed with the following DQEP:

```
(1) CREATE RESULT TABLE r1 AS SELECT * FROM x
WHERE x.b < 2 (on every partition)
(2) CREATE RESULT TABLE r2 AS SELECT * FROM
REMOTE(r1) (on every node)
(3) SELECT * FROM r2 JOIN y WHERE y.c > 5 AND r2.a
= y.a (on every partition)
```

(1) is executed on every partition, to apply the filter `x.b < 2` locally prior to the broadcast. Then (2) is executed on every leaf node to bring the filtered rows of *x* to every node. In this case, *r2* would be materialized into a temporary hashtable for the join with *y* in (3). (3) is executed on every leaf node, with results streamed across the network to the aggregator and then to the client. The use of *r2* allows the broadcasted data to be brought to each leaf once, whereas if (3) read directly from `REMOTE(r1)`, the query would produce the same results but every partition would separately read the broadcasted data from across the network and materialize the resulting table.

The flexibility of the *RemoteTables* and *ResultTables* abstraction also easily enables various alternate execution methods for this broadcast. For example, another possible DQEP for this broadcast is:

```
(1) CREATE RESULT TABLE r1 AS SELECT * FROM x
WHERE x.b < 2 (on every partition)
(2) CREATE RESULT TABLE r2 AS SELECT * FROM
REMOTE(r1) (on a single node)
(3) CREATE RESULT TABLE r3 AS SELECT * FROM
REMOTE(r2) (on every node)
(4) SELECT * FROM r3 JOIN y WHERE y.c > 5 AND r3.a
= y.a (on every partition)
```

Here, a single node reads the broadcasted rows from across the cluster, and then distributes them to all other nodes. This is the smallest example of a broadcast tree. Compared to the first plan, only linearly many connections are used across the cluster instead of quadratically many. On the other hand, this introduces slightly more network latency. Which DQEP is better depends on the cluster topology and data size.

6.2.2 *Reshuffles*

ResultTables can also be created with a specified partitioning key to execute reshuffles. Using the same example query and schema, if after applying filters table *x* is larger than or of similar size as table *y*, the best plan would be to reshuffle *y* on *a* to match *x*:

```
(1) CREATE RESULT TABLE r1 PARTITION BY (y.a) AS
SELECT * FROM y WHERE y.c > 5 (on every partition)
(2) SELECT * FROM x JOIN REMOTE(r1(p)) WHERE x.b <
2 AND x.a = r1.a (on every partition)
```

(1) repartitions the rows of *y* from each local partition. Then (2) is executed on each partition on the output side of the reshuffle, reading the data corresponding to one partition of the repartitioned data from across the cluster, the partition *p* which matches the local partition of *x*, and executing the join.

If neither *x* nor *y* are sharded on *a*, then the best plan, if the two tables are similarly sized after filters, may be to reshuffle both sides. This can be done with a similar DQEP:

```
(1) CREATE RESULT TABLE r1 PARTITION BY (x.a) AS
SELECT * FROM x WHERE x.b < 2 (on every partition)
(2) CREATE RESULT TABLE r2 PARTITION BY (y.a) AS
SELECT * FROM y WHERE y.c > 5 (on every partition)
(3) SELECT * FROM REMOTE(r1(p)) JOIN REMOTE(r2(p))
WHERE r1.a = r2.a (on every partition)
```

7. EXPERIMENTS

7.1 TPC-H Benchmark

We used queries from the well-known TPC-H benchmark to investigate the quality of query execution plans generated by the query optimizer, by measuring the performance of queries compared to another database and compared to queries generated by MemSQL with some query optimizations disabled, as well as the time required for query optimization.

We ran and compared MemSQL with another widely used state-of-the-art commercial analytical database, which we will refer to as "A" throughout this section. A is a column-oriented distributed database. We created all tables as disk-backed columnstore tables in MemSQL, matching A, which has only disk-backed column-oriented tables. We used TPC-H at Scale Factor 100.

We ran MemSQL and A on Amazon EC2, on a cluster of 3 virtual machines, each of the m3.2xlarge instance type, with 8 virtual CPU cores (2.5GHz Intel Xeon E5-2670v2), 30 GB RAM, and 160 GB of SSD storage. The network bandwidth was 1 Gbps. The MemSQL cluster configuration was a MemSQL leaf node on each of the three machines, in addition to a MemSQL aggregator node on one of the machines. 3 machines is a relatively small cluster for MemSQL, but the choice of cluster configuration for this experiment was due to limitations on running A.

It is very difficult to get all the tuning right for any database system, and therefore the main aim for this experiment was to provide a rough comparison of MemSQL and A, not to claim that MemSQL is better than A for TPC-H. It is hard to compare execution plans since every database has a different execution engine.

We measured the latency of running each query alone. We used this measurement for simplicity because the focus of this experiment is on the quality of the query execution plans generated by the query optimizer, not on other features of MemSQL such as the technology enabling high-concurrency real-time analytical workloads. Figure 3 shows the execution times (in seconds) for the TPC-H queries. Queries q17 and q21 are omitted because they are currently not efficiently runnable in this cluster configuration of MemSQL on columnstore tables due to execution limitations (on rowstore tables, they are optimized and executed well).

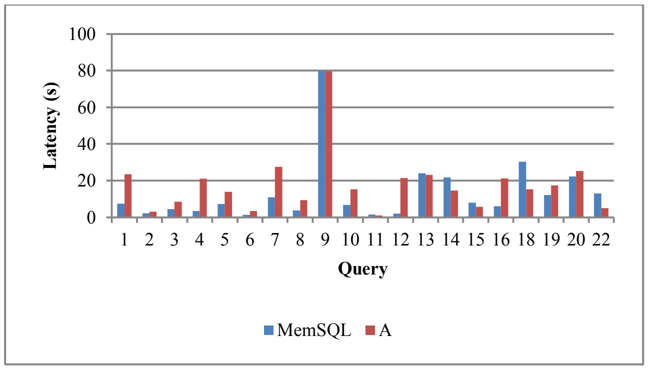


Figure 3: Execution time for TPC-H queries compared to A

MemSQL is significantly faster than A on many of the TPC-H queries (up to 10x faster), and somewhat slower than A on some queries (up to 2.6x slower).

We also compared to MemSQL with most query rewrites disabled. We did not disable some basic query rewrites such as predicate pushdown because that would harm performance too much in an uninteresting way, since predicate pushdown is responsible for moving filters below data movement operations, for example. Without query rewrites, several queries are not efficiently runnable at all (q2, q4, q18, q20, q22). Figure 4 shows the execution times for the remaining queries. As can be seen, query rewrites greatly improve performance on many queries. For example, one rewrite which is critical for performance on several queries is transforming scalar subqueries to joins.

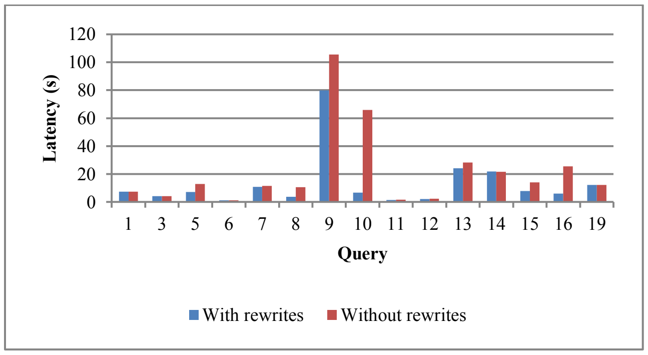


Figure 4: Execution time for TPC-H queries with vs without most query rewrites.

In addition, we measured the time taken to optimize the TPC-H queries in MemSQL, shown in Figure 5. Every query was optimized in less than 100 milliseconds, and most under 20ms. The optimization time for a query is dependent on factors such as the number of tables in the query and the rewrites that were considered for the query. Since we did not have access to the source code of A, there was no way to accurately measure the query optimization times in A.

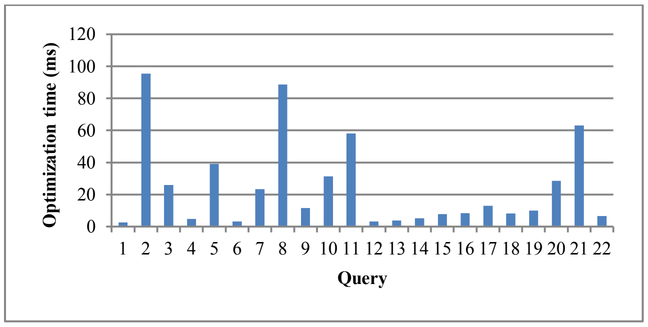


Figure 5: MemSQL query optimization time for TPC-H queries.

7.2 Customer Workload

We now look at a real-world analytical workload from a MemSQL customer. The example query described below is a simplified and anonymized version of the key part of one query from this workload. It features a join involving several tables that form multiple star schemas. The joins do not feature any primary or foreign keys and it was not possible to infer any table as fact or dimension from the schema description. This query is representative of several other queries in the workload, which share the same join pattern.

```
SELECT …
FROM a11, a12, a13, a14, a15, a16, a17, a18, a19
WHERE a11.x = a12.y
  AND a11.y = a13.z
  AND a12.z = a14.x
  AND a11.a = a15.x
  AND a13.a = a16.a
  AND a13.b = a17.b
  AND a14.a = a18.a
  AND a15.a = a19.a
  AND a16.f = 1
  AND a18.c = 2
  AND a19.c = 3
  AND a17 IN (4,5,6)
```

This query runs 10x faster with a bushy join plan compared to a left-deep join plan. In this case, the best join order was a bushy join plan *(a11, a12, (a13, a17, a16), (a14, a18), (a15, a19))*. Producing this join plan was not possible with our previous algorithm in [12] since none of the joins involved primary keys. It is worthwhile to note that the bushy join approach of [1] also would not be able to detect the bushy nature of this query because the *seed* tables *a13, a14* and *a15* had far smaller cardinality than their *satellite* tables and therefore, could not have passed the cardinality threshold of that method to be considered as a *fact* table. With our new algorithm, we are able to detect bushy patterns in many other queries in real customer workloads; a speedup of 5-10x is seen for such customer queries.

8. RELATED WORK

In the recent past, several *Massively Parallel Processing (*MPP) database systems such as SAP HANA [3], Teradata/Aster, Netezza [15], SQL Server PDW [14], Oracle Exadata [20], Pivotal GreenPlum [17], and Vertica [7] have gained popularity. A few systems have implemented and published literature about query optimization and planning for distributed database systems. We briefly summarize some of them in this section.

SQL Server Parallel Data Warehouse (PDW) [14] uses a query optimizer built on top of the Microsoft SQL Server optimizer. The plan search space alternatives that are considered by the SQL Server are sent over to PDW Data Movement Service and these plans are then costed with distributed operator costs to choose a distributed plan.

Orca [17] is the modular query optimizer architecture from Pivotal that is designed for big data. It is top-down query optimizer and can run outside the database system as a stand-alone optimizer, thus providing the capability to support different computing architectures like Hadoop etc. Although the paper mentions about join ordering and rewrites, there is no explicit mention of how rewrites are costed, or any technique that leads to vast pruning of states in the dynamic programming search space. Also, the intermediate language of communication, *Data eXchange Language (DXL)* is based on XML while MemSQL *ResultTables* interface is based on still-popular SQL.

Vertica [7] implements an industry strength optimizer for its column storage data that is organized into projections. The optimizer implements rewrites, cost-based join order selection, optimized for star/snowflake schemas. Vertica's *V2Opt* optimizer improves on existing limitations (if join keys are not co-located) by replicating the pertinent projections to improve performance. Again, there is no explicit mention of how rewrites are costed, join orders are generated, what pruning strategies are used that would potentially overlap with the technical contributions of this paper.

In the past, there have been several attempts to improve query optimization time. Bruno et al. [2] propose several polynomial heuristics that take into account selectivity, intermediate join size etc. Some other previous work [4][18] also propose several heuristics, but these techniques were designed before distributed query processing became widespread and therefore do not take data distribution into consideration. Another area where there have been attempts to improve query optimization time is in parallelizing the join enumeration process. Han et al. in [5] propose several techniques to parallelize parts of the System-R style enumerator and prototyped in PostgreSQL. Waas et al. in [19] propose techniques to parallelize the enumeration process for Cascade style enumerators. Heimel et al. [6] suggest using GPU co-processor to speed up the query optimization process.

9. CONCLUSION

In this paper, we describe the architecture of the MemSQL Query Optimizer, a modern optimizer for a distributed database designed to optimize complex queries effectively and efficiently, in order to produce very efficient distributed query execution plans with fast optimization times. We discuss the problem of query rewrite decisions in a distributed database, argue that the method in existing systems of making these decisions oblivious of distributed costs leads to poor plans, and describe how the MemSQL query optimizer solves that problem. We describe strategies which enable the Enumerator to quickly optimize joins over an extremely large search space, including a new algorithm to efficiently form bushy join plans. Finally, we demonstrate the efficiency of our techniques over several queries from TPC-H benchmark and a real customer workload.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their feedback on this paper. We would also like to thank Ankur Goyal, Wayne Hong, and Jason Israel for their contributions to this work. Finally, we are grateful to the members of the MemSQL team for their collaboration and their contributions to the MemSQL database system.

11. REFERENCES

[1] R. Ahmed, R. Sen, M. Poess, and S. Chakkappen. Of snowstorms and bushy trees. *Proceedings of the VLDB Endowment*, 7(13):1452–1461, 2014.

[2] N. Bruno, C. Galindo-Legaria, and M. Joshi. Polynomial heuristics for query optimization. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 589–600. IEEE, 2010.

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *ACM SIGMOD Record*, 40(4):45–51, 2012.

[4] L. Fegaras. A new heuristic for optimizing large queries. In *Database and Expert Systems Applications*, pages 726–735. Springer, 1998.

[5] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proceedings of the VLDB Endowment*, 1(1):188–200, 2008.

[6] M. Heimel and V. Markl. A first step towards GPU-assisted query optimization. *ADMS@ VLDB*, 2012:33–44, 2012.

[7] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[8] G. Moerkotte and W. Scheufele. Constructing optimal bushy processing trees for join queries is NP-hard. *Technical reports*, 96, 2004.

[9] R. O. Nambiar and M. Poess. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1049–1058. VLDB Endowment, 2006.

[10] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34. ACM, 1979.

[12] R. Sen, J. Chen, and N. Jimsheleishvilli. Query optimization time: The new bottleneck in real-time analytics. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Management and Analytics*, page 8. ACM, 2015.

[13] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex query decorrelation. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 450–458. IEEE, 1996.

[14] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query optimization in Microsoft SQL Server PDW. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 767–776. ACM, 2012.

[15] M. Singh and B. Leonhardi. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 385–386. IBM Corp., 2011.

[16] A. Skidanov, A. Papitto, and A. Prout. A column store engine for real-time streaming analytics. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016.

[17] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2014.

[18] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *ACM SIGMOD Record*, volume 18, pages 367–376. ACM, 1989.

[19] F. M. Waas and J. M. Hellerstein. Parallelizing extensible query optimizers. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 871–878. ACM, 2009.

[20] R. Weiss. A technical overview of the Oracle Exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[21] M. Zukowski, M. Van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical DBMS. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350. IEEE, 201