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

April 24, 2025

1 Introduction

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's distributed architecture is a shared-nothing architecture 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.

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.

1.2 Query Optimization in MemSQL

The optimizer framework is divided into three major modules:

1. **Rewriter**: The Rewriter applies SQL-to-SQL rewrites on the query.
2. **Enumerator**: determines the distributed join order and data movement decisions as well as local join order and access path selection.
3. **Planner**: The Planner converts the chosen logical execution plan to a sequence of distributed query and data movement operations.

2 Overview of MemSQL Query Optimization

2.1 DQEP Examle

Suppose the `customer` table is a distributed table that has a shard key on `c_custkey` and the `orders` table is also a distributed table that has a shard key on `o_orderkey`.

```sql
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.
```sql
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.
```sql
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
```

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 example, lineitem and part are distributed rowstore tables hash-partitioned on `l_orderkey` and `p_partkey`, respectively. The query is:

```sql
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)
```

Rewrite:

```
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
```

Enumerate: 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.

```
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,

```
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

3.1 Heuristic and Cost-Based Rewrites

- **Column Elimination**: remove unsed columns
- **Group-By Pushdown**:
- **Sub-Query Merging**:

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.

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.

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.

Consider two tables $T1(a, b)$ and $T2(a, b)$ which are shared on the columns $T1.b$ and $T2.a$, respectively, and with a unique key on column a for $T2$

```sql
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:

```sql
-- 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:

```sql
-- 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 $S_G = 1/4$ 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 $S_J = 1/10$ 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:

$$Cost_{Q1} = R_1 C_J + R_1 S_J C_G = 200,000 C_G + 20,000 C_G = 220,000$$
$$Cost_{Q2} = R_1 C_G + R_1 S_G C_J = 200,000 C_G + 50,000 C_J = 250,000$$

4 Problems

5 References