Integrating the Orca Optimizer into MySQL

Arunprasad P. Marathe, Shu Lin, Weidong Yu, Kareem El Gebaly, Per-Åke Larson, Calvin Sun
Huawei Technologies Canada Co., Ltd.
{arun.marathe,shu.lin,weidong.yu,kareem.el.gebaly1,calvin.sun3}@huawei.com,gpalarson@outlook.com

ABSTRACT

The MySQL query optimizer was designed for relatively simple, OLTP-type queries; for more complex queries its limitations quickly become apparent. Join order optimization, for example, considers only left-deep plans, and selects the join order using a greedy algorithm. Instead of continuing to patch the MySQL optimizer, why not delegate optimization of more complex queries to another more capable optimizer? This paper reports on our experience with integrating the Orca optimizer into MySQL. Orca is an extensible open-source query optimizer—originally used by Pivotal's Greenplum DBMS—specifically designed for demanding analytical workloads. Queries submitted to MySQL are routed to Orca for optimization, and the resulting plans are returned to MySQL for execution. Metadata and statistical information needed during optimization is retrieved from MySQL's data dictionary. Experimental results show substantial performance gains. On the TPC-DS benchmark, Orca's plans were over 10X faster on 10 of the 99 queries, and over 100X faster on 3 queries.

1 INTRODUCTION

Huawei's cloud platform includes multiple database offerings under the unifying brand GaussDB. GaussDB for MySQL—also marketed with the name 'Taurus'—is a cloud-native database service, fully compatible with MySQL. Taurus separates compute and storage. Data is divided into slices that are distributed among a pool of Page Stores. The DBMS frontend is a slightly modified version of MySQL 8.0.

Traditionally, MySQL has been used for transactional workloads, but often such workloads also contain some fraction of more complex analytical queries—on which users expect reasonably good performance. Cloud service providers such as Amazon, Alibaba, and Huawei are observing this trend in cloud deployments [2, 4].

In Huawei's case, Taurus currently relies on the query optimizer in MySQL 8.0. The optimizer does a competent job on OLTP-type workloads consisting of relatively simple queries with a few joins, but falls short on more complex queries, producing plans that are sometimes orders of magnitude slower than the optimal plans. In particular, it optimizes one `SELECT` block at a time whose plan follows a somewhat rigid pattern: first a sequence of joins; followed by grouping and aggregation; then a filter operation if a `HAVING` clause is present; and finally an optional sort to satisfy an `ORDER BY` clause. The MySQL optimizer may produce sub-optimal plans for several reasons.

(1) It generates only left-deep join plans; bushy plans are not supported.
(2) It computes the join order using a greedy algorithm, which does not guarantee optimality.
(3) It does not consistently refactor predicates with `OR` to pull out common terms. This may, for example, prevent using a hash join.
(4) It does not consider pushing aggregation below joins, or doing partial aggregations followed by final ones.
(5) It can only push predicates below `GROUP BY` for derived tables, but not for subqueries.

Faced with these optimizer shortcomings, we considered two options: either extend the MySQL optimizer or attempt to integrate into MySQL an existing query optimizer capable of handling complex queries.

We rejected the first approach because the MySQL optimizer is not designed to be extensible [18], and does not include a framework for rule-based query rewrites or query transformations. This makes it difficult to extend the optimizer with new transformation rules—for example, pushing aggregation below join, doing partial aggregations, or decorrelating subqueries—and ensure that the rules are applied consistently whenever possible during the optimization process.

Orca, on the other hand, is a relatively feature-rich, open-source query optimizer, specifically designed to optimize analytical queries [22], especially on MPP systems. We decided to explore whether and how optimization could be delegated to Orca, and the resulting plans executed by MySQL. How well Orca integrates with a non-MPP system such as MySQL is an interesting question by itself. This paper describes how the integration was done, issues found, and reports the improvements in query performance observed on the TPC-H and TPC-DS queries.

The rest of the paper is organized as follows. To set the stage, we first give an overview of the Taurus system, and outline the architecture of the MySQL optimizer in Section 2. A high-level description of the chosen integration between MySQL and Orca is provided in Section 3. In order for the integration to work, query parse trees and plan trees need to be translated back and forth between MySQL and Orca as described in Section 4. Orca's extensibility API requires a metadata provider to be written for each target system. Such a provider for MySQL metadata is described in Section 5. Experimental evaluation appears in Section 6. The lessons learned from the integration exercise—in considerable detail—are gathered in Section 7. The related work is surveyed in Section 8. Finally, conclusions and future work are described in Section 9.

2 OVERVIEW OF TAURUS AND MySQL OPTIMIZER

2.1 Taurus Architecture

A brief overview of the Taurus design appears below; a more detailed description can be found in [6].

Taurus separates compute and storage, and relies only on append-only storage. As illustrated in Fig. 1, a Taurus database system consists of four major logical components: database frontends, a Storage Abstraction Layer (SAL), Log Stores, and Page Stores. These components are distributed between two physical layers: a compute layer and a storage layer. The database is

divided into fixed-size (10 GB) segments called *slices* that are distributed among multiple Page Stores. Log Stores and Page Stores are multi-tenant services shared by many database servers.

Taurus is designed to work with different database frontends: MySQL, PostgreSQL, and openGauss. The frontend layer consists of one master that can serve both read and write queries, and up to 15 read replicas that execute read queries only. A frontend server is responsible for accepting incoming connections, optimizing and executing queries, and managing transactions. All updates are handled by the master. The master makes modifications to database pages persistent by synchronously writing log records, in triplicate, to the durable storage in Log Stores. The master also periodically communicates the location of the latest log records to all of the read-only replicas so that they can read the latest log entries, and update any affected pages in their buffer pools.

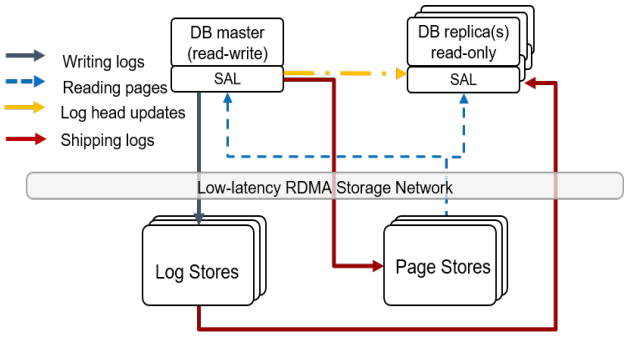

Figure 1: Taurus architecture.

The Storage Abstraction Layer (SAL) is an independent component running on the frontend servers. It isolates the frontends from the underlying complexity of remote storage, slicing of the database, recovery, and read replica synchronization. SAL is responsible for writing log records to Log Stores, distributing them to Page Stores, and reading pages from Page Stores. SAL is also responsible for creating, managing, and destroying slices in Page Stores and routing page read requests to Page Stores.

A Log Store is a service executing in the storage layer responsible for storing log records durably. Once all log records belonging to a transaction have been made durable, transaction completion can be acknowledged to the client. Log Stores have two main functions. First and foremost, they ensure the durability of log records. Second, they also serve log records to read replicas so that the replicas can apply the log records to pages in their buffer pools.

Page Store servers are also located in the storage layer. A Page Store server hosts slices from multiple database frontends (tenants), but each slice contains table and index data from only one database, thus ensuring tenant-level data separation. The main function of a Page Store is to keep pages up-to-date, and serve read requests from masters and replicas. A Page Store receives log records from multiple masters for the pages that it hosts and applies them to bring pages up-to-date so they are ready to be served.

2.2 MySQL Query Optimization

As mentioned in Section 2.1, MySQL is one of the frontend DBMS's that Taurus supports, and in that configuration, MySQL's query optimizer and its execution engine compile and execute the submitted queries.

Fig. 2 illustrates the high-level architecture of MySQL query optimization and execution. The Parser and Resolver layers generate the initial parse tree of a query, and handle such issues as syntax checking; name resolution; access control; data types; and string collations. During the Prepare phase, the parse tree may undergo several logical transformations: derived tables may be merged into their parent query blocks; predicates may be pushed down from a query block into its descendent tables or derived tables; subqueries may be converted into semi-joins or derived tables; join-nests may be flattened; scalar expressions simplified using rules of transitivity, tautologies, and contradictions; etc.

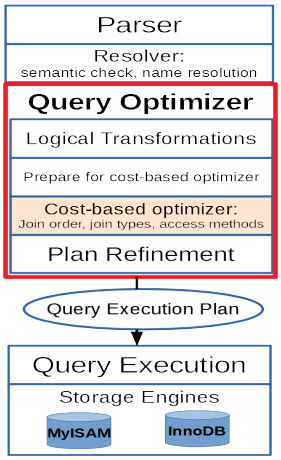

Figure 2: MySQL query optimizer layers. Diagram adapted from: http://www.unofficialmysqlguide.com

The next step is Cost-based Optimization which proceeds one `SELECT` block at a time, and is limited to determining the best join order; join method (nested loop or hash join); and table access method (table scan, index scan, or index lookup and which index). The optimizer only considers left deep plans, and aggregation is always done after all of the tables in a block have been joined. In the integration work described in this paper, this phase's work is delegated to Orca, and therefore, is shown shaded in orange in Fig. 2. The result of the cost-based optimization is a *skeleton plan* in which join orders, join methods, and the tree structure have been finalized.

Although the name may suggest otherwise, Plan Refinement is an important phase of MySQL query optimization. During this phase, selection conditions are placed and pushed down into tables and indexes, where possible. A sort is avoided if an index scan already delivers rows in the expected sorted order. Aggregations, group-level filtering, and row limit enforcement are added at this step too. After this phase, the plan is ready for execution.

3 INTEGRATION OVERVIEW

The schematic diagram in Fig. 3 illustrates how Orca is integrated into MySQL. Three components (shown in blue) implement the interface between them: Parse Tree Converter; Metadata Provider; and Orca Plan Converter. The left and right converters correspond to the Query2DXL an DXL2Plan components mentioned in [21], although unlike those components, the exchange formats are not DXL.

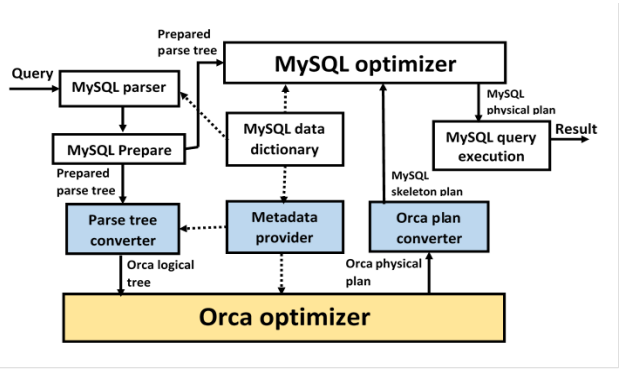

Figure 3: Overview of MySQL's integration with Orca. The middle layer components (in blue) implement the interface between the two.

A query submitted to MySQL arrives at its parser. After the usual syntactic, semantic, and access control checks—during which the parser consults MySQL's data dictionary for required metadata information—a parse tree is created.

The MySQL Prepare phase then performs a number of standard rewrite transformations on the parse tree, creating what is sometimes called an abstract syntax tree (AST). The prepared AST then proceeds to cost-based optimization either by MySQL's query optimizer, or by Orca as described in this paper.

The Parse Tree Converter takes as input a prepared MySQL parse tree, and converts it to an equivalent Orca logical operator tree. The converter creates Orca-compatible trees directly, and does not use Orca's XML-based DXL format as intermediary. When resolving types and attributes of various metadata objects referred to in a parse tree—for example, tables; table columns; data types; and type attributes—the parse tree converter invokes the necessary methods of the Metadata Provider. The parse tree converter is described in more detail in Section 4.1.

The Metadata Provider provides API access to the various MySQL object types—for example, tables; columns; data types; statistics; and histograms—to Orca. A DBMS-specific metadata provider needs to be written to integrate with Orca, and accordingly, we wrote one for MySQL. Unlike the Parse Tree Converter, the metadata provider's interface with Orca is in DXL format. The provider obtains the necessary information about the various objects from MySQL's data dictionary, and is described in detail in Section 5.

Orca query optimization converts an Orca logical plan to an Orca physical plan. The Orca Plan Converter converts that physical plan to the corresponding MySQL *skeleton plan*. As the name suggests, a skeleton plan has the most important plan elements, but also omits many details. Only the table join order, join methods, and table access methods are retained in the skeleton plan, whereas join predicates, non-join predicates, groupings, orderings, projections, and so on are omitted. As already mentioned in Section 2.2, the MySQL query optimizer has the notion of a skeleton plan, and knows how to convert it to a physical plan during the optimizer's plan refinement phase. Therefore, Orca integration uses plan skeletons as the intermediary format. Once again, Orca's DXL data format is not used during this conversion either.

Plan refinement, which converts a skeleton plan to an executable physical plan, accomplishes four things: predicate placement; aggregation; row ordering; and row limit enforcement. Predicates are attached to either leaf nodes (tables) as filters, or to intermediate nodes as join conditions or filters. Aggregations are performed after all of the tables are joined in a query block. Row ordering and limit enforcement are determined last.

Finally, the resulting MySQL physical plan is executed by MySQL's execution engine to produce the query results.

3.1 Orca Query Optimization Benefits

The TPC-DS query 72 shown in Listing 1 can illustrate the gains obtained by delegating query optimization to Orca. It is a simple snowflake query that joins a large fact table (`catalog_sales`) with 10 smaller tables, and performs a group-by aggregation. The essential structures of the plans produced by the MySQL and Orca optimizers are shown in Fig. 4 and Fig. 5, respectively.

Both optimizers return plans that first join all of the tables, and do the group-by last. The execution times for the MySQL and Orca plans are 288 sec and 34 sec, respectively—an 8.5X improvement.

Listing 1: TPC-DS Query 72

```
SELECT i_item_desc, w_warehouse_name, d1.d_week_seq,
  SUM(CASE WHEN p_promo_sk IS NULL THEN 1 ELSE 0 END)
      no_promo,
  SUM(CASE WHEN p_promo_sk IS NOT NULL THEN 1 ELSE 0
      END) promo, COUNT(*) total_cnt
FROM catalog_sales
 JOIN inventory ON (cs_item_sk = inv_item_sk)
 JOIN warehouse ON (w_warehouse_sk=inv_warehouse_sk)
 JOIN item ON (i_item_sk = cs_item_sk)
 JOIN customer_demographics ON (cs_bill_cdemo_sk =
     cd_demo_sk)
 JOIN household_demographics ON (cs_bill_hdemo_sk =
     hd_demo_sk)
 JOIN date_dim d1 ON (cs_sold_date_sk = d1.d_date_sk)
 JOIN date_dim d2 ON (inv_date_sk = d2.d_date_sk)
 JOIN date_dim d3 ON (cs_ship_date_sk = d3.d_date_sk)
 LEFT OUTER JOIN promotion ON (cs_promo_sk=p_promo_sk)
 LEFT OUTER JOIN catalog_returns ON
   (cr_item_sk = cs_item_sk AND cr_order_number =
     cs_order_number)
WHERE d1.d_week_seq = d2.d_week_seq
   AND inv_quantity_on_hand < cs_quantity
   AND d3.d_date > (CAST(d1.d_date AS DATE) + INTERVAL
     '5' DAY)
   AND hd_buy_potential = '501-1000'
   AND d1.d_year = 1999 AND cd_marital_status = 'D'
GROUP BY i_item_desc,w_warehouse_name,d1.d_week_seq
ORDER BY total_cnt DESC, i_item_desc, w_warehouse_name,
   d1.d_week_seq
LIMIT 100
```

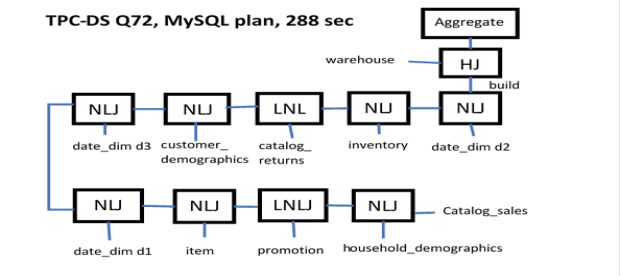

Figure 4: Query 72 plan generated by the MySQL optimizer.

The plan selected by the MySQL optimizer first joins ten tables together using a sequence of nested-loop joins, with the largest table (`catalog_sales`) as the driving table. The eleventh table is then joined in using a hash join. The result is finally sorted and aggregated using streaming aggregation. Only one of the

ten joins is a hash join which is typical: the MySQL optimizer favors nested loop joins because currently, hash join selection is not cost-based.

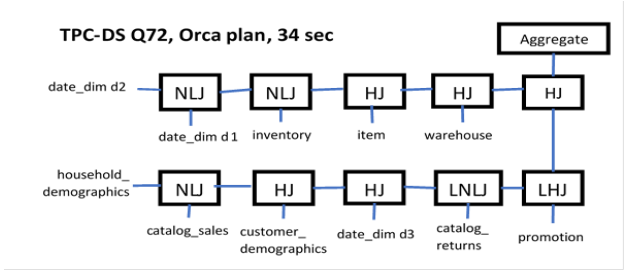

Figure 5: Query 72 plan generated by the Orca optimizer.

Orca selects a bushy plan where six of the ten joins are hash joins, and only four are nested loop joins. The result is then sorted and aggregated using streaming aggregation. The execution time is reduced to 34 sec thanks to better choices of join order and join methods.

4 QUERY TREE TRANSLATIONS AND PLAN REFINEMENT

Orca provides three integration points for interfacing with a database system: queries as input; exchange of metadata about those queries; and execution plans as output. Orca uses an XML-based data format called DXL for the three information exchanges [21], and indeed, a DXL object can encode queries, metadata, and execution plans. The Orca research reported in [21] contains two sample DXL documents.

In the MySQL integration work reported here, the metadata provider communicates with the other components using DXL, but the two tree converters take a more direct approach by exchanging in-memory trees. The descriptions of the two tree conversions and the plan refinement phase follow; Section 5 then explains how the MySQL metadata provider works.

4.1 MySQL to Orca Tree Converter

MySQL to Orca tree converter takes as input a prepared abstract syntax tree (AST), and outputs an Orca logical operator tree. Conversion from an AST-like parse tree to an operator tree is a necessary step in any SQL query optimizer.

MySQL starts with an initial AST; performs name resolution and access checks; and applies logical rewrites to incrementally change that AST. The MySQL way is to continue making such gradual changes by attaching more data structures to the AST until an execution plan results. Indeed, the boundary between a logical plan and a physical plan in MySQL is somewhat blurred.

MySQL sometimes decides to rewrite a `EXISTS/NOT EXISTS` query to semi/anti-semi join structure depending on column nullability. In such a case, the translator needs to rearrange predicates to divide them among the two sides of the semi join and the `ON` condition of the semi join itself for an interesting reason. At the place where the converted Orca logical tree joins Orca's optimization pipeline, selection pushdown has already happened. Thus, without such predicate segregation, Orca plans would not benefit from selection pushdown—a simple yet effective SQL query optimization technique.

MySQL to Orca tree conversion itself is largely clause-wise, and the approximate translation order is:

- `FROM`
- `WHERE (1)`
- window functions (1)
- `WHERE (2)`
- `SELECT (1)`
- `GROUP BY`
- `SELECT (2)`
- `HAVING`
- window functions (2)
- `ORDER BY`
- `SELECT (3)`
- `LIMIT`

In the above list, the conventions '`SELECT (1)`' and '`SELECT (2)`' mean that a part of a `SELECT` clause (projection) is translated before `GROUP BY`, and a part after. '`SELECT (3)`' refers to a case when a projection is not based on any table columns, as in: '`SELECT 'Canada' as Country .. `'. Similarly, window functions are processed twice. If a query has no aggregations, window functions are handled before projection; otherwise, they are handled after aggregations.

The TPC-H query 4 appearing in Listing 2 can suggest some of the complexities involved.

Listing 2: TPC-H Query 4

```
SELECT o_orderpriority, COUNT(*) AS order_count
FROM orders
WHERE o_orderdate >= DATE '1995-01-01' AND
  o_orderdate < DATE '1995-01-01' + INTERVAL '3' MONTH
  AND EXISTS (SELECT *
              FROM lineitem
              WHERE l_orderkey = o_orderkey AND
                    l_commitdate < l_receiptdate)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

MySQL to Orca tree converter receives a rewritten AST version of the query. For easy depiction, we have converted that AST using a SQL-like syntax in Listing 3. Notice that the subquery has been converted into a semi-join, and all of the conditions appear in the `WHERE` clause.

Listing 3: The AST for TPC-H Query 4 in SQL-like syntax

```
SELECT orders.o_orderpriority AS o_orderpriority,count
      (0) AS order_count
FROM orders SEMI JOIN lineitem
WHERE ((orders.o_orderdate >= DATE '1993-11-01')
  AND (orders.o_orderdate < (DATE '1993-11-01' +
       INTERVAL 3 MONTH))
  AND (lineitem.l_commitdate < lineitem.l_receiptdate)
  AND (orders.o_orderkey = lineitem.l_orderkey))
GROUP BY orders.o_orderpriority
ORDER BY orders.o_orderpriority;
```

The converter divides the `WHERE` clauses among the 'orders' table, the 'lineitem' table, and the semi-join's `ON` condition, and the resulting Orca logical tree appears in Listing 4. Notice that predicate pushdown has been accomplished.

Listing 4: The Orca logical tree for TPC-H Query 4 in SQL-like syntax

```
SELECT orders.o_orderpriority AS o_orderpriority,count
      (0) AS order_count
FROM
  (SELECT * FROM orders
   WHERE (orders.o_orderdate >= DATE '1993-11-01')
     AND (orders.o_orderdate < (DATE '1993-11-01' +
          INTERVAL 3 MONTH)))
  SEMI JOIN
  (SELECT * FROM lineitem
   WHERE lineitem.l_commitdate<lineitem.l_receiptdate)
  ON (orders.o_orderkey = lineitem.l_orderkey)
GROUP BY orders.o_orderpriority
ORDER BY orders.o_orderpriority;
```

Although Listing 4 does not make it apparent, the projection of 'o_orderpriority' happens before the GROUP BY (the SELECT (1) case), and the 'COUNT(*)' projection happens after it (the SELECT (2) case).

While determining how to translate a MySQL parse tree to an Orca logical tree, it was helpful to submit the same query text to GPDB (Greenplum DBMS), and consult the logical trees that GPDB submitted to Orca.

During the MySQL to Orca translation, Orca's logical 'table-descriptor' objects are enhanced by adding to them pointers to the *TABLE_LIST* data structure that MySQL maintains to keep track of the tables in a query. Table descriptor objects are subsequently carried into Orca's physical plan operators, and with them the corresponding *TABLE_LIST* pointers get copied too. When the time comes to translate an Orca physical plan to MySQL (as explained in Section 4.2), this mapping comes in handy. Without it, the mapping would have to be reestablished by searching MySQL parse tree which is not only slow, but also error-prone.

The MySQL to Orca tree converter then consults the MySQL metadata provider, and embellishes the Orca logical tree nodes with the necessary OID's. A typical interaction of the converter is to send the schema-qualified name of a table, say 'tpch.lineitem', and receive that table's unique OID. OID's for such objects as tables, column types, expressions, functions, and so on are established. Later on, when Orca needs to consult the metadata provider to extract more information—for example, the histograms of a table's columns—it uses the table's pre-established OID.

Currently, the parse tree converter only sends SELECT queries to Orca for optimization; INSERT, UPDATE, and DELETE statements, and queries posed on MySQL's system tables are not sent. Furthermore, only 'complex' queries are sent to Orca. Query complexity is defined to be the total number of table references in a query, and the resulting 'complex query threshold' is set to three. However, as mentioned in Section 9, we plan to implement a more sophisticated approach in future. Also, we plan to remove the following two limitations of the converter: (1) only non-recursive common table expressions (CTEs) are allowed; and (2) GROUPING functions can only have one column. (Orca does not support GROUPING functions, but we implemented single-column versions to run the TPC-DS workload.)

4.2 Orca to MySQL Plan Converter

The Orca optimizer receives an Orca logical tree generated by the converter described in Section 4.1; accesses the MySQL metadata via the MySQL metadata provider; and optimizes the logical tree to create a physical plan. The task of converting that plan to a MySQL executable plan is accomplished in two steps:

(1) Orca to MySQL plan converter converts an Orca physical plan to a MySQL skeleton plan.
(2) MySQL plan refinement then converts a skeleton plan to an executable plan as described in Section 4.3.

A skeleton plan essentially encodes the best join position and the best join method for each table appearing in a query.[1] In our integration, Orca is used to determine exactly those details, and hence, skeleton plan is an ideal intermediary. The information in a skeleton plan also has the most impact on query performance.

Orca to MySQL plan translation is illustrated using the TPC-H Q17 appearing in Listing 5. The most relevant physical operators in Orca's plan for Q17 appear in Fig. 6; the actual plan is much more complicated. The plan sketch shows that the three leaf nodes are 'part', 'lineitem', and 'lineitem' accessed using table scan, index scan, and index scan, respectively. The numbers after the physical operator names are the 'memo' group ID's.

Orca to MySQL plan translation is accomplished by traversing such an Orca physical plan twice; the two passes are described next. The section ends with a discussion of two special cases.

Listing 5: TPC-H Query 17

```
SELECT SUM(l_extendedprice) / 7.0 AS avg_yearly
FROM lineitem , part
WHERE p_partkey = l_partkey
  AND p_brand = 'Brand#14' AND p_container = 'SM␣PKG'
  AND l_quantity < (SELECT 0.2 * AVG(l_quantity)
                    FROM lineitem
                    WHERE l_partkey = p_partkey)
LIMIT 1;
```

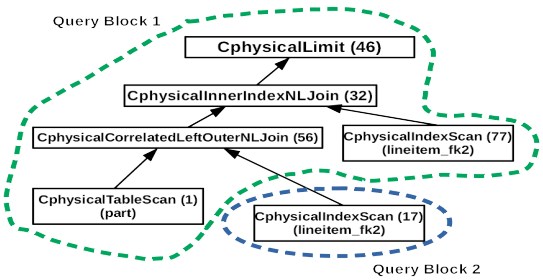

Figure 6: The important physical operators in Orca's TPC-H Q17 plan.

4.2.1 First pass. Orca to MySQL plan converter first makes a pre-order traversal of the tree shown in Fig. 6. The first leaf node encountered is 1, and the path leading up to it (46, 32, 56, 1) is put in Query Block 1 which may acquire more nodes because the tree traversal has not yet completed.

The traversal next visits the leaf 17. Each leaf node contains a *TABLE_LIST* object (mentioned in Section 4.1) which contains—among other things—a link to the leaf's containing query block. For 17, the query block changes, and therefore, a new query block (Query Block 2) is initiated containing (17). Furthermore, the subquery rooted at 17 is converted to a derived table because the subquery is attached to the containing query via the 'l_quantity < (subquery)' predicate, and 'l_quantity', belonging to the containing query's 'lineitem' table, has not been seen by the scan yet.

Had the subquery not been converted to a derived table, during plan refinement (explained in Section 4.3), when MySQL optimizer attaches the 'l_quantity < (subquery)' predicate, it would have been attached to the node 77. That, however, is not what Orca wants: the intention is to compute the subquery before 77 is joined in.

The traversal next encounters 77 which is determined to belong to Query Block 1. At the end of the tree traversal, the two query blocks 1 and 2 are determined to be (46, 32, 56, 1, 77) and (17), respectively, and are demarcated using different colors in Fig. 6.

During the traversal, if the first pass discovers that Orca has changed the query block structure altogether, Orca optimization is aborted, and the system resorts to the usual MySQL query optimization.

[1] A 'table' in this context can also be a temporary derived table or a CTE.

4.2.2 Second pass. Orca to MySQL plan converter then makes a second pass through the tree shown in Fig. 6 to fill in MySQL's 'best-position' arrays. Each query block has its own such array, and the array entries are filled with the leaf nodes as they appear left-to-right in the pre-order traversal. In addition to the table name, an array entry also contains the access method chosen, its cost, and the number of output records.

For the query block 1 in Fig. 6, the best-position array contains [part, derived_1_2, lineitem], whereas for query block 2, it (trivially) contains [lineitem]. The name 'derived_1_2' is that of a temporary table that is materialized once for each outer row. During this second pass, the *TABLE_LIST* pointers mentioned in Section 4.1 are utilized. The two best-position arrays for the TPC-H query 17 are depicted in Fig. 7 using the same color scheme as in Fig. 6.

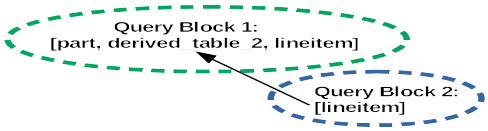

Figure 7: The two MySQL 'best-position' arrays and their relationship for TPC-H Query 17.

Cost and cardinality estimations are important for users to understand why a certain plan is chosen. During the second pass, those values from an Orca plan are copied over to MySQL side. When MySQL optimizer produces the physical plan during plan refinement (explained further in Section 4.3), the cost and row estimations are copied to the iterators, and show up in the MySQL plan (the EXPLAIN output) as usual.

4.2.3 Special cases. Together, the two passes accomplished Orca to MySQL plan conversion, but two complexities were omitted for simplicity's sake. They are briefly described in this section.

The first involves handling of common table expressions (CTE's). Orca has the 'one-producer-plan multiple-consumers' model for CTEs [7], whereas MySQL has the 'multiple-producer-plans multiple-consumers' model (although only one producer plan executes). Orca's single producer is mapped to MySQL's multiple consumers during the first pass.

Specifically, in case of Orca, a CTE sub-tree is a single instance which the multiple consumers point to without ambiguity. During MySQL compilation, multiple identical CTE sub-tree instances exist, and although only one of them is chosen for execution, the choice is not known at plan translation time. Accordingly, Orca's single CTE instance is translated to *n* MySQL copies—one for each consumer—so that any one of the copies can execute.

The second is about subqueries and derived tables. During the tree translation, subqueries are preferred because translation to derived tables involves structural changes to the parse tree. (MySQL can translate subqueries to derived tables, but that functionality is off by default.) There are cases when Orca to MySQL converter must convert a subquery to a derived table. For example, Orca might produce a non-correlated execution plan for a correlated subquery, requiring the derived table approach. There are also cases when the converter overrides Orca's derived table conversion. For example, TPC-DS Q9's `CASE` structure is illustrated in Listing 6. The successive `CASE` clauses are formed using consecutive bucket boundaries: [1, 20], [21, 40], and so on. The corresponding `WHEN`, `THEN`, and `ELSE` queries are similar otherwise. In such a case, the derived table approach requires redundant expression evaluations, whereas the subquery approach only evaluates one of the `CASE` clauses specific to a bucket.

Listing 6: TPC-H Query 9's `CASE` structure

```
SELECT
  -- [1, 20] bucket
  CASE WHEN (SELECT .. FROM .. WHERE ..)
       THEN (SELECT .. FROM .. WHERE ..)
       ELSE (SELECT .. FROM .. WHERE ..)
  -- [21, 40] bucket
  CASE WHEN (SELECT .. FROM .. WHERE ..)
       THEN (SELECT .. FROM .. WHERE ..)
       ELSE (SELECT .. FROM .. WHERE ..)
  ...
```

4.3 MySQL Plan Refinement

The Orca optimization has determined the best join orders and join methods, and the Orca to MySQL plan converter has translated those choices to a MySQL skeleton plan. MySQL plan refinement—which is oblivious of this Orca detour—begins by handling of the scalar expressions, represented in a MySQL parse tree as 'Item' objects. Those 'Item' objects are attached to the appropriate tables or intermediate nodes.

Plan refinement then handles aggregations and window functions, and chooses between stream and hash aggregates. Tuple orderings and row limits are also addressed. If join orders and access methods do not yield desired sorted orders, the necessary sorts are inserted.

In the integration work, MySQL's plan refinement code was essentially unchanged. The only change was to always yield to Orca's hash-join decisions. (In some cases, plan refinement overrides some hash-join decisions made during the previous greedy join-order search, and that overriding was disabled.)

In principle, tighter integration with Orca may enable MySQL to delegate handling of aggregation, windows functions, sorts, and limits too; and we plan to investigate that in future.

For TPC-H Q17, the resulting MySQL physical plan is reproduced in Listing 7 using MySQL's EXPLAIN format (slightly edited for brevity).

Listing 7: TPC-H Query 17's Orca-assisted MySQL query execution plan

```
EXPLAIN (ORCA)
-> Limit: 1 row(s)
  -> Aggregate: sum(lineitem.l_extendedprice)
    -> Nested loop inner join  (cost=1548.90 rows
        =209683)
      -> Nested loop inner join (cost=1292.27 rows=7045)
        -> Invalidate materialized tables (row from part)
            (cost=1051.53 rows=7044)
          -> Filter: ((part.p_container = 'SM␣PKG') and
              (part.p_brand = 'Brand#14'))  (cost
              =1051.53 rows=7044)
            -> Table scan on part  (cost=1051.53 rows
                =3961757)
        -> Table scan on derived_1_2 (cost=0.03 rows=1)
          -> Materialize (invalidate on row from part)
            -> Aggregate: avg(lineitem.l_quantity)
              -> Index lookup on lineitem using
                  lineitem_fk2 (l_partkey=part.
                  p_partkey)  (cost=0.03 rows=30)
      -> Filter: (lineitem.l_quantity < derived_1_2.
          Name_exp_1)  (cost=0.04 rows=1)
        -> Index lookup on lineitem using lineitem_fk2
            (l_partkey=part.p_partkey) (cost=0.04 rows
            =30)
```

Several things are worth noticing in the EXPLAIN output. First, the first line indicates that the plan was Orca-assisted. Second, the estimated execution costs and cardinalities are coming from Orca. Third, the left outer nested loop join of Fig. 6 has been replaced

with an inner nested loop join (typeset in blue) because the predicate 'lineitem.l_quantity < derived_1_2.Name_exp_1' implies that 'derived_1_2.Name_exp_1' is not NULL. Fourth, the best-position array shown in Fig. 7 contains the materialized table as one of the leaves, and its name, 'derived_1_2' matches with the table scan operator's target. Fifth, because correlation is involved, each 'part' row from the outer side causes a separate materialization of the 'derived_1_2' table, and that fact is indicated using the two matching 'invalidate' annotations (typeset in red) on the outer and inner sides of the bolded nested loop join. (In EXPLAIN outputs, an outer table precedes an inner table underneath a join.)

5 Orca METADATA PROVIDER FOR MySQL

Orca's integration with a target DBMS uses the plug-in approach of a DBMS-specific metadata provider, and this section describes the MySQL metadata provider. It differs from a similar provider for PostgreSQL in one important aspect. When integrating with PostgreSQL, Orca handles both metadata objects and, in some cases, function pointers. Therefore, PostgreSQL metadata provider needs to return function pointers for expression evaluation, type casts, and so on—something that the MySQL metadata provider avoids because a query executes inside MySQL. This difference simplifies the MySQL metadata provider, but it still has to fulfil all of the Orca API contracts—even if sometimes by providing stubs.

In PostgreSQL—Orca's original target DBMS—all types of objects (tables, columns, types, expressions, histograms, etc.) are identified by ID's. Accordingly, the communication between Orca and the MySQL metadata provider is heavily object ID-based, and uses the DXL format [21]: the object ID's eventually get inserted into DXL instances. How the metadata provider computes the ID's for the various types of objects such as types; expressions (and their rewritten forms); and functions is described next.

5.1 Types and Type Categories

MySQL has 31 types—for example, various types of integers; various types or reals; various types of dates and datetimes; JSON; geometry; and so on. Conceptually, they can be combined arbitrarily to form various arithmetic, comparison, and aggregation expressions. Not all of the combinations are valid of course, but given MySQL's rich type promotions, many of them are.

The MySQL metadata provider needs to provide metadata ID's for data types and expressions composed from types. To manage complexity, types are composed into *type categories*. The 31 types are divided into 12 type categories. For example, DECIMAL, FLOAT, DOUBLE, and NEWDECIMAL are put into the "NUM" (numeric) type category; four types of BLOB's are put into the consolidated "BLB" type category; and so on. Doing so drastically reduces the number of expression combinations. Less obviously, it avoids handling of MySQL's rich type promotions mentioned earlier.

For each data type, the metadata provider provides the following information to Orca: name; length; whether it is pass-by-value; whether it is text-related; its type modifier (lengths for CHAR, VARCHAR, and other variable-length types); and so on.

5.2 Expressions

Orca expressions are of three types: arithmetic, comparison, and aggregation.

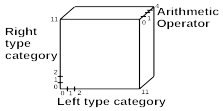

Figure 8: Arithmetic expressions enumerated.

Arithmetic expressions are formed using the usual five operators $\{+, -, *, /, \%\}$. The left and right operands are the 12 type categories mentioned in Section 5.1. The total number of arithmetic expressions is therefore $12 \times 12 \times 5 = 720$. As illustrated in Fig. 8, they can be thought of as the integral vertices contained within a 3-dimensional cube whose X, Y, and Z axes index the left type category, the right type category, and the arithmetic operator, respectively. The metadata provider creates a unique linear order for the 720 indexable points, and provides functions that can translate an (i, j, k) index tuple to a point in that linear order, and vice versa.

Comparison expressions are handled identically except that there are 6 comparison operators $\{<, \leq, >, \geq, =, <>\}$, and therefore, the cube shape is $12 \times 12 \times 6$.

Aggregation expressions are also handled similarly, but two details vary. First, the standard six SQL aggregation operations COUNT, MIN, MAX, SUM, AVG, and STDDEV are unary, and therefore, the cube shown in Fig. 8 is two-dimensional. Second, SQL has two flavors of COUNT: COUNT(*) and COUNT(column/expression), with slightly different semantics. The latter is handled similarly for all of the data types. To handle those two cases, two special type categories called 'STAR' and 'ANY' were created—only for aggregations—for a total of 14 type categories. Thus, the shape of the two-dimensional array is 14×6.

5.3 Commutator and Inverse Expressions

SQL query optimizers may benefit from expression rewrites because rewritten expressions may be further simplified, or may find index access paths more easily. Orca is able to commute and inverse expressions as follows.

For arithmetic expressions, commutation means that $(a + b)$ and $(a * b)$ may be rewritten to $(b + a)$ and $(b * a)$, respectively. The operators '−', '/', and '%' do not commute. For comparison expressions, $(a \leq b)$ and $(10 > c)$ may be rewritten to $(b \geq a)$ and $(c < 10)$, respectively.

The generic expression enumeration scheme described in Section 5.2 can commute and inverse expressions seamlessly as described below. Given an OID k of an expression, the OID of its commutator expression is computed as follows.

(1) From k, the expression type can be determined because various types of expressions are laid out in the OID space in specific slots as will be explained in Section 5.6. For concreteness, and without loss of generality, suppose that k corresponds to a comparison expression.
(2) k is converted to its 0-based enumeration ID, $enum_k$. Because 864 comparison expressions exist, $enum_k \in [0, 863]$.
(3) From $enum_k$, the type-category-based expression is determined, say $INT8 > NUM$.
(4) Metadata provider rewrites it to $NUM < INT8$.
(5) The provider computes the enumeration ID for $NUM < INT8$; converts that ID to an OID; and returns the OID to Orca.

Inverse expressions exist only for comparison expressions, and enable query rewrites by introducing or eliminating SQL's NOT

operator. For example, the inverse of $(a < b)$ is $(a \geq b)$. The six comparison operators $\{=, <>, <, \leq, >, \geq\}$ have the inverses $\{<>, =, \geq, >, \leq, <\}$, respectively. The metadata provider computes the inverses similarly to the way it handles commutators.

For expressions without commutators or inverses, a special invalid OID is returned, informing Orca that certain rewrites for those expressions are not possible.

5.4 Functions

The Orca optimizer considers two major types of functions: *mapped functions* and *regular functions*. Each mapped function corresponds to an expression (arithmetic, comparison, and aggregate) mentioned in Section 5.2. When Orca acts as PostgreSQL's query optimizer, such mapped functions provide executable code to evaluate the expressions. Regular functions are the ones that SQL provides: `EXTRACT`, `SUBSTRING`, `CAST`, `ROUND`, `UPPER`, `CONCAT`, `ABS`, and so on.

In the Orca integration described in this paper, function execution (and SQL physical operator execution) happens inside the Taurus MySQL query executor, but still, to fulfil the Orca API, the MySQL metadata provider generates the metadata IDs for both mapped and regular functions.

5.5 Relations, Statistics, and Histograms

For each relation, MySQL metadata provider provides such information as its name; columns; column types; cardinality; number of null values (column-wise); number of distinct values (column-wise); column histograms, indexes, and so on.

Histogram translation was straightforward because both of the histogram types in Orca—singleton and equi-height—are supported in MySQL, and histograms themselves contain similar information. For columns with `UNIQUE` constraints, MySQL does not maintain histograms, and that restriction was lifted so that such histograms could be fed to Orca.

For histograms on string data types, the two optimizers differ. Orca only supports singleton histograms, whereas MySQL can produce equi-height histograms too. To convert MySQL string histograms to Orca, We had two choices: (1) only convert singleton histograms; and (2) modify Orca to support equi-height histograms. We chose the latter approach; more details on our solution appear in Section 7.

5.6 Metadata OID Layout

As mentioned in the preceding sub-sections, MySQL metadata provider needs to provide OID-based access to various types of SQL objects: from types and expressions to relations and histograms. The metadata provider uses an internal object layout scheme such that the OID space is used judiciously and unambiguously.

A high-level overview of the layout of the various types of objects is suggested in Fig. 9. (Only a subset of the object types are shown although the scheme applies to all of them.) The OID's for the 720 arithmetic expressions, for example, are assigned by first generating their enumeration (the range $[0, 719]$), and then by adding those to a 'base' value suggested by the *arith_base* pointer in Fig. 9. This 'base + enumeration ID' scheme is used for all of the object types.

At the bottom of Fig. 9, OID's for relations and their associated objects are suggested. Such objects are distinguished from the objects above them in that their counts are not known in advance. For relations, columns, and so on, MySQL generates internal

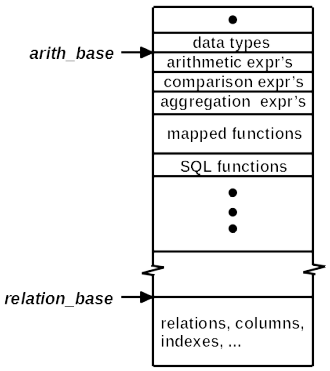

Figure 9: Layout of the various metadata OID's.

object ID's that are added to a different 'base' value (indicated as *relation_base* in Fig. 9) to compute their corresponding Orca OID's. Such relational objects are placed sufficiently apart from the first group of objects so that collisions are avoided. In summary, such a layout scheme ensures that the various OID's are laid out consecutively when appropriate, and with the necessary large gaps in between when not.

5.7 Sample Interaction for TPC-H Q17

For the TPC-H Q17 appearing in Section 4.2, MySQL metadata provider's first interaction is with the MySQL to Orca tree converter shown in Fig. 3, and explained in Section 4.1. For example, OID's for 'lineitem' and 'part' and their indexes are returned to the converter.

Once an OID-embellished Orca logical tree is ready, the metadata provider's remaining interactions are with the Orca optimizer itself. For example, Orca requests and obtains statistical information about the 'part' and 'lineitem' tables—including column histograms. Orca maintains an internal metadata cache—not explicitly shown in Fig. 3—and if the required information preexists there, the metadata provider is not queried again. Expression OID's are also sent as requested. For example, for "`p_container = 'SM_PKG'`", the OID for `STR_EQ_STR` is returned. For `STR_EQ_STR`, the commutator and inverse expressions exists, and their OID's are returned too.

6 EXPERIMENTAL EVALUATION

So far we have shown how Orca can be integrated into MySQL, but the key question is whether this results in better plans. To answer this question, we did performance comparisons between the plans produced by Orca with those by the MySQL optimizer for TPC-H and TPC-DS queries.

The experiments were run on a small Taurus test cluster with four Page Store nodes. Each node was running on Intel® Xeon® Gold 6161 CPU @ 2.20 GHz with 44 cores and 250 GB memory, and had a Huawei Hi1822 network card rated at 25 Gbps. The SQL node had 360 GB of memory, but was otherwise identical to the Page Store nodes.

6.1 TPC-H Workload

TPC-H is a well-known decision support benchmark with 22 queries [23]. We created a database at the scale factor 20 (20 GB), and ran the 22 queries sequentially from Q1 to Q22 with 'complex query threshold' (mentioned at the end of Section 4.1) set to its

default value of 3. Orca's join-order search algorithm was set to EXHAUSTIVE2—its most thorough setting. The run times with MySQL plans and with Orca plans are plotted in Fig. 10.

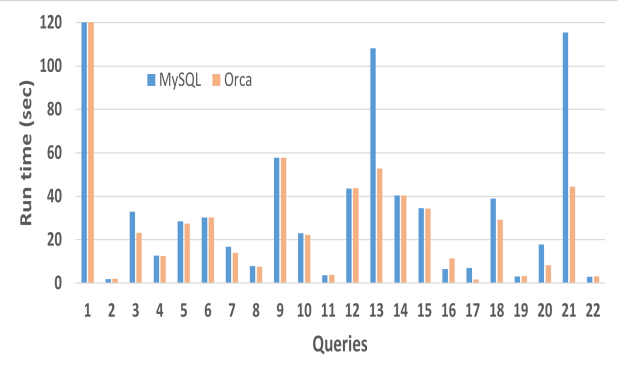

Figure 10: Execution time for the TPC-H queries

Although the total run time for the 22 queries reduces by a modest 16%, the reduction is much higher for some queries. Queries 21 and 13 show the largest improvements.

Query 21 run time decreased from 115 sec to 45 sec, a 2.6X improvement. The query joins four tables, and has two subqueries. The Orca plan has a different join order—the table that occurs first in the MySQL plan is last in the Orca plan. The two subqueries are applied last in both plans but in different order.

Query 13 run time decreased from 108 sec to 53 sec, a 2X improvement. The query consists of a left outer join followed by two rounds of `GROUP BY`. The only plan difference is the choice of the join method for the left outer join: MySQL uses a left outer nested loop join, whereas Orca uses a left outer hash join.

However, for query 16 appearing in Listing 8, the Orca plan is 2X slower then the MySQL plan (12.6 sec versus 6.5 sec).

Listing 8: TPC-H Query 16

```
SELECT p_brand, p_type, p_size,
  COUNT(distinct ps_suppkey) AS supplier_cnt
FROM partsupp, part
WHERE p_partkey = ps_partkey AND
  p_brand <> 'Brand#34' AND
  p_type NOT LIKE 'LARGE␣BRUSHED%' AND
  p_size IN (48, 19, 12, 4, 41, 7, 21, 39) AND
  ps_suppkey NOT IN (
    SELECT s_suppkey FROM supplier
    WHERE s_comment LIKE '%Customer%Complaints%')
GROUP BY p_brand, p_type, p_size
ORDER BY supplier_cnt DESC, p_brand, p_type, p_size;
```

In both plans, the subquery is placed on the inner sides of nested loop joins. The MySQL plan scans the 'supplier' table; applies the `LIKE` filter; materializes a temporary table; de-duplicates the rows; and creates an index on 's_suppkey' on-the-fly. Only 106 rows survive the `LIKE` predicate, and this strategy proves effective, especially because the initial table scan benefits from sequential prefetch. Notice that, in general, table materialization on the inner side of a nested loop join is risky because the actual cardinality could be much larger than the estimated.

Orca, on the other hand, does not seem to create indexes on-the-fly on the inner side, and takes a more conservative approach by accessing the inner side using a primary key index lookup. Eventually, that choice results in a slower plan.

Out of curiosity, we forced Orca to scan the 'supplier' table on the inner side, and the driving join changed to a hash join. Because of the hash join though, a sort was not pushed in to the outer side, and the resulting plan ran in about 8 sec—much faster than before, but still a bit slower than the MySQL plan.

Incidentally, the `LIKE` predicate in query 16 contains a regular expression, and in such a case, a cardinality estimator based purely on histograms may find it difficult to estimate row counts anyway. In summary, MySQL takes a somewhat riskier approach, but it works out.

6.2 TPC-DS Workload

TPC-DS [22] is a more modern decision support benchmark with 99 queries that are considerably more varied and complex than those in the TPC-H benchmark. The scale factor was 100 (100 GB), and the 'complex query threshold' was set to 2. Orca's join-order search algorithm was set to EXHAUSTIVE2. We were able to execute all of the queries, but to do so we had to rewrite queries with `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, and `EXCEPT ALL` because MySQL does not support those four set operators.

Fig. 11 compares the total query run times—including optimization time—of plans selected by the MySQL optimizer and plans selected by Orca. Note the different scales on the two Y-axes.[2] Orca produces better plans for two-thirds of the 99 queries, and the total run time for the 99 queries was reduced by 62%. For the following 10 queries the Orca plan is at least 10X faster: {1, 6, 17, 24, 31, 32, 41, 58, 81, 92}. Among those, three obtain at least 100X speedups: {Q1: 198X, Q6: 123X, Q41: 222X}.

Q1 and Q6 benefit from Orca's choice of hash joins in selected places in the plans instead of MySQL's preference for nested loop joins.

The reason behind Q41's improved performance is instructive. Q41 contains two occurrences of the same 'item' table, one in the outer `FROM` clause, and the other in a subquery embedded within the `WHERE` clause. The inner query in the `WHERE` clause has a complex nest of predicates of the form:

```
((item.i_manufact = i1.i_manufact) AND x) OR
((item.i_manufact = i1.i_manufact) AND y)
```

in which x and y themselves are multi-clause composite predicates combined using `AND`'s and `OR`'s.

Orca is able to factor out the self-join condition, and rewrite the expression to:

```
(item.i_manufact = i1.i_manufact) AND (x OR y)
```

whereas MySQL is unable to do so. The two plans are identical otherwise.

In the original form adopted by MySQL, the 'false' rows need to be processed twice—once on each side of the `OR` clause—whereas in Orca, the 'false' row bailouts need to be determined only once. The 'item' table has 28000 rows, but only 999 distinct 'i_manufact' values, and hence the rewrite is so effective.

Overall, Orca plans tend to be slower than MySQL plans only on short queries; on longer queries Orca plans are almost always faster. This trend is clearly visible in Fig. 12. The X-axis shows query run time using the MySQL plan; the Y-axis shows the Orca plan's run time divided by the MySQL plan's run time. For example, the point just below 6 on the Y-axis represents query 56 which ran in 0.66 sec using the MySQL plan, and 5.6 times slower (3.7 sec) with the Orca plan. The slowness can be attributed to two factors. First, an Orca-routed query undergoes a complete Orca optimization, and a partial MySQL query optimization; and for short-duration queries, that overhead plays a role. Second, short-duration queries also tend to be simple, and for them, MySQL is

[2]During the MySQL run, Query 1 was cancelled after 600 sec (10 min); queries 4 and 78 took 505 and 331 sec, respectively.

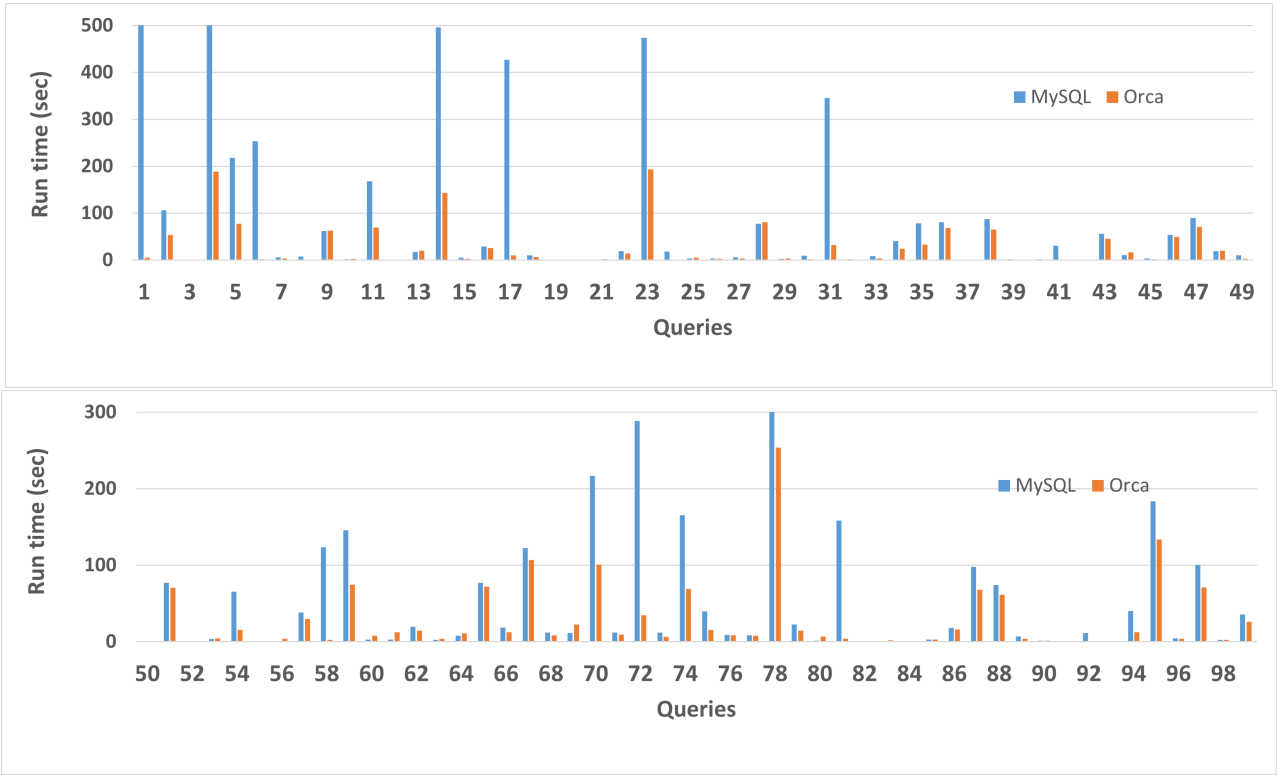

Figure 11: Execution time for the TPC-DS queries.

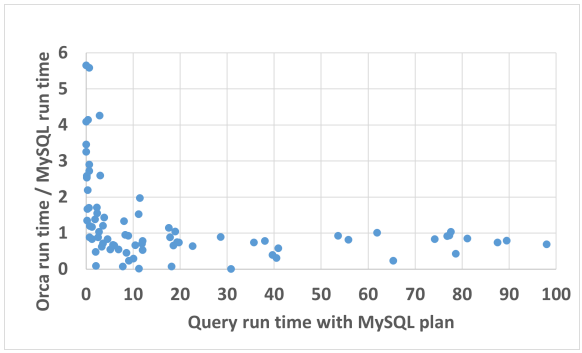

Figure 12: Orca is slower only on short queries.

known to produce reasonably fast plans already. There could be other reasons too, and we continue to investigate to find the root causes. In summary, Orca clearly outperforms the MySQL optimizer on longer queries but sometimes loses out to the MySQL optimizer on short queries.

6.3 Orca Compilation Overhead

In this last experiment, we measure the additional compilation overhead caused by Orca integration. The times to produce the EXPLAIN outputs for the TPC-H and TPC-DS queries are measured with and without Orca's presence, and the 'complex query threshold' is set to 1, so that all of the queries take the Orca detours. Furthermore, Orca is invoked with two possible dynamic programming-based search strategies for join enumeration: EXHAUSTIVE and EXHAUSTIVE2 (the most thorough setting). Table 1 captures the total elapsed (wall-clock) times to compile the entire TPC-H and TPC-DS query suites. Several observations can be made based on the numbers reported in Table 1, and the underlying data.

(1) Orca compilations are significantly slower than MySQL compilations.

(2) For the relatively simple TPC-H queries, EXHAUSTIVE2 does not add additional search overhead, and in fact, is a bit faster than EXHAUSTIVE.

(3) For complex TPC-DS queries, EXHAUSTIVE2 adds a noticeable overhead of 26.1 sec over EXHAUSTIVE. Digging deeper though, the overhead is almost entirely attributable to only two queries, Q14 and Q64, that require 30.0 sec and 2.1 sec, respectively, of additional compilation time under EXHAUSTIVE2.[3] Query 64 contains a CTE with an 18-way join, and the CTE is joined with itself. Query 14 also contains many CTE expressions with multi-way joins, and the query is more than 200 lines long. It is unsurprising that full optimization of such complex queries takes time.

(4) Overall, Orca compilation overhead is worth the trade-off because as mentioned in sections 6.1 and 6.2, TPC-H and TPC-DS benchmarks run 16% and 62% faster, respectively, when executed using Orca-generated plans.

7 LESSONS LEARNED

The fact that we managed to integrate Orca into MySQL query optimizer and obtained better plans for complex queries demonstrates that the approach is feasible and worthwhile. The MySQL optimizer was never designed with complex queries in mind, and continuing to extend it is becoming increasingly difficult [18].

[3] EXHAUSTIVE2 is slightly faster than EXHAUSTIVE for many other queries, which lowers the total difference to 26.1 sec.

Table 1: Orca query compilation overhead

Compiler	Total EXPLAIN time (sec)	
	TPC-H	TPC-DS
MySQL	0.17	1.09
MySQL + Orca—EXHAUSTIVE	2.06	48.08
MySQL + Orca—EXHAUSTIVE2	1.85	74.21

To the best of our knowledge, Orca is currently the only optimizer that was designed to be reused in multiple systems, and has well-defined integration points. As always, there is scope for improvement, and in the following, some of the lessons learned are described.

In the metadata provider, assigning types to type categories proved fruitful. It drastically reduced the number of expression types, and allowed us to fine-tune type categories based on need. For example, in an earlier version of the metadata provider, the following MySQL types were all assigned to the INT type-category: `TINY`, `SHORT`, `LONG`, `LONGLONG`, `INT24`, `YEAR`, `ENUM`, and `SET`. Query performance experimentation, however, revealed that Orca could not determine proper indexes for integer-like columns based on the coarse INT type-category. Therefore, INT was replaced with three more refined type-categories: INT2, INT4, and INT8.

Orca only supports singleton string histograms because it uses a hash function to convert a string to an integer. The hash function is not order-preserving, and so it cannot be used for range predicates, but can be used for equalities and non-equalities, and can handle arbitrary-length strings. MySQL supports both singleton and equi-height string histograms, and bucket boundaries are strings themselves. Because of the more general approach, all of the comparison operators are supported.

We added equi-height string histogram support to Orca by converting string bucket boundaries to 64-bit signed integers, and providing a comparison function for them. The function preserves string order, but because of the fixed length, it cannot distinguish between two strings with a long common prefix. A long-term solution might be call MySQL functions for string comparisons using Orca's proxy mechanism.

A somewhat related topic is that of expression evaluation. Orca treats types other than a handful—for example, various types of integers, OID's, Boolean, and so on—to be generic. Expression evaluation on generics is left to particular DBMS's—to be provided using callbacks, for example. A richer set of non-generic types would permit more effective query optimization out-of-the-box.

In the Orca implementation that we experimented with [9], a multi-process query optimization model is assumed. Indeed, Orca can compile multiple queries in parallel for PostgreSQL because that DBMS is also multi-process (a process per connection). MySQL is multi-threaded (a thread per connection), and therefore, we could not optimize multiple MySQL queries concurrently using Orca.

Some challenges were posed because MySQL was the target system. Although some of them were mentioned earlier in the paper, they are collected here—in no particular order—for the benefit of others.

(1) MySQL does not generate bushy plans, and to be able to execute Orca-optimized bushy plans, additional MySQL 'glue' code needed to be written, although MySQL iterators themselves could be put anywhere in the plan trees. Specifically, because MySQL assumes left-deep plan trees, its best-position array was slightly extended to handle bushy trees.

(2) MySQL does not support `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, and `EXCEPT ALL` operators, and TPC-DS queries having those constructs needed to be rewritten using alternate forms.

(3) Orca can push predicates into common table expressions [7]. For example, if two occurrences of the same CTE have predicates "a = 5" and "a = 6", the two predicates can be `OR`-ed. That functionality had to be added to MySQL.

(4) Orca can refactor predicates with `OR`. For example, it can rewrite "(a = b `AND` c = d) `OR` (a = b `AND` e = f)" to "(a = b) `AND` (c = d `OR` e = f)", and use a hash join with the rewritten predicate (for example). MySQL performs such refactorization only in cases when indexes can be utilized to evaluate "(a = b)", but the rewritten form is desirable, in general. In addition to hash joins, such queries as TPC-DS Q41 mentioned in Section 6.2 also benefit. Accordingly, the scope of the factorization in MySQL was broadened.

(5) MySQL does not collect column histograms for `UNIQUE` columns. That restriction was removed so that such histograms could be provided to Orca.

(6) Orca can push a `HAVING` predicate below `GROUP BY` if the predicate's columns are a subset of the `GROUP BY` columns. MySQL can do that for derived tables, but not when subqueries undergo '`IN`-to-`EXISTS`' transformations. Accordingly, that capability was added to MySQL.

Finally, there were many changes to the Orca optimizer to improve query performance, or to ensure plan appropriateness.

(1) Orca permits one to add additional transformation rules, and one such rule that swapped an inner join with an inner correlated apply was added. That rule is effective when the pushed-down inner join reduces cardinality so that the pushed-up apply has to process fewer rows. For completeness, 11 similar rules were added for the various types of apply and join combinations, although TPC-H queries 20 and 21—whose performance improvements were being targeted—only required 4 of them.

(2) We discovered that MySQL treats the build and probe sides of an inner hash join differently. Everywhere else, the widely-known convention "build-side on the right, and probe-side on the left" is used, but for inner hash joins, the reverse is true. The flip was introduced in the Orca-generated trees for the MySQL target.

(3) We discovered that Orca was using the rebind values incorrectly. The rebind count is simply the number of rows coming from the outer side, and after the operator swapping mentioned in Item 1, outdated rebind values were still in use.

(4) Orca uses an index scan only when there is a predicate that requires it, but an index scan can also supply a required row order. That enhancement to Orca was added.

(5) Orca can push `GROUP BY` below join, but because MySQL query execution cannot execute such plans, that rewrite rule was disabled in Orca. For the same reason, an Orca rule that transforms a left semi-join to inner join below a `GROUP BY` was disabled.

(6) Orca can generate semi-hash-join plans whose build sides contain more than one table. Because MySQL cannot execute such plans, that type of plan generation was disabled.

(7) This work demonstrates that Orca plans—designed for MPP systems—can be made to work on single-node system too. Orca plans (and MPP plans in general) handle data replication

and distribution, but those aspects are not relevant on single-node systems such as MySQL where correlations among inner and outer blocks are one of the predominant features of complex analytical queries. Orca was provided the necessary nudges to guide it towards MySQL-relevant plans. For example, some plans are marked as "'replicated distribution required' *and* 'replication prohibited'"—an invalid combination on MPP systems because replication is required to produce replicated distribution, but perfectly valid on single-node systems. Accordingly, we let Orca produce such plans to not miss out on any good choices.

8 RELATED WORK

A good query optimizer is necessary for efficient and speedy query execution. The relative importance of the various aspects of query optimization, namely, join order selection, cardinality estimation, bushy trees, cost models, heuristics, and so on are somewhat folklore knowledge. A recent work reported in [15] attempts to assign relative importance to such factors. For example, it claims that join order matters more than bushy trees. The Orca evaluation presented in this paper attempted to get the best plans out of Orca; provided the histograms as they existed inside MySQL; and in case of string histograms, extended Orca to use a custom string comparison function. We did also extend MySQL to execute bushy plans.

System R [19] can be considered the original query optimization framework in that it had a large influence on later optimizers. Subsequently, several query optimization frameworks have been proposed: Volcano [10], Cascades [11], Starburst [17], and OPT++ [13]. Orca—whose code we downloaded from its open-source repository [9]—is based on Cascades. Many commercial and open-source query optimizers are based on the ideas from Volcano and Cascades: SQL Server, SCOPE [5], PDW [20], and Greenplum [12].

Reusing an existing query optimizer to do something slightly different has been explored in the past. In particular, by using 'shell' databases, optimizers were utilized to optimize a more general class of queries, and in query tuning. The approach works as follows. A 'shell' of a database is derived from an existing database by only copying from it the metadata and statistics. Because query optimization does not need actual data, it can work on the shell rather than the original one. Such an approach avoids metadata provider altogether. At least two use cases of such shell databases exist. First, SQL Server's Parallel Data Warehouse (PDW) [20] generates plans for a distributed query by first generating a collection of non-distributed plans based on a shell database; then adding data distribution operations to those plans; and finally selecting the lowest-cost distributed plan. Second, SQL Server's Database Tuning Advisor [1] tunes queries by generating a shell database from a production database, and then by asking 'what-if' questions to that shell database in order to avoid tuning overhead on a production server.

9 CONCLUSIONS AND FUTURE WORK

Traditional query optimizers are monolithic, and difficult to extend. In this project, we explored whether and how query optimization can be delegated to Orca: an open-source query optimizer that permits integration with other products. Using Taurus MySQL, we demonstrated the feasibility of this approach, and its performance benefits. In particular, by writing a MySQL metadata provider as a plug-in to Orca, information about MySQL objects pertinent for query optimization (tables, columns, types, cardinalities, histograms, and so on) was made available to Orca. Two additional components were written that translated MySQL prepared parse trees to Orca logical operator trees, and Orca physical plans to MySQL skeleton plans. Performance benefits were observed using two classical decision support benchmarks: TPC-H and TPC-DS.

We took a conservative approach, in essence, delegating only join optimization and access method selection to Orca, being careful to not change the query block structure, and returning Orca plans in the form of skeleton plans already used by MySQL. The Orca optimizer was invoked during MySQL optimizer's 'prepare' phase, but that is not the only possibility. Two alternatives are suggested below; we plan to explore them in future.

First, Orca can be invoked after MySQL's cost-based optimization has been performed, but only if the estimated cost of the MySQL plan is above some threshold. MySQL's greedy join order determination is relatively fast, so this would not add significantly to the query run time. Such an approach would almost certainly be better than our three-table heuristic for deciding which queries to send to Orca.

Second, Orca can be invoked much earlier—say immediately after MySQL's syntactic, semantics, and access-control checks. If so, Orca gets full freedom to do tree transformations, expression rewrites, and so on. This approach has several complexities.

- Any expression evaluation required must be done by MySQL, and the results translated back to Orca.
- MySQL's rigid execution model—query block by query block, and aggregation only after all tables are joined—limits plan choices, sometimes resulting in non-optimal plans. Overcoming this would require extending the MySQL executor to handle a broader class of plans.
- Converting Orca plans to MySQL executable plans may become more difficult; it would probably not be possible to use simple skeleton plans.

Orca's cost model—for example, relatively high index lookup and hash join costs—needs fine-tuning. In general, costing formulas need to be updated to better suit MySQL's InnoDB storage engine. Producing distributed query execution plans from Orca to benefit such plans in MySQL is a possibility. We also plan to handle string histograms in which bucket boundaries have long common prefixes.

To put this work in a larger context, SQL query optimizer architectures require a second look with a view to decomposing them into reusable, API-level components such as cardinality estimator; costing; logical and physical rewrite rules; join enumeration; join order selection; and so on. Orca and such projects as those based on Apache Calcite [3, 8] have taken good first steps, and this work has demonstrated that the topic merits further explorations. The field of compilers, for example, has long benefited from a somewhat reminiscent approach (common front-end, intermediate representation, language runtime, back-end, and so on) [14, 16, 24].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable feedback.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *(e)Proc. of the VLDB 2004 Conference, Toronto, Canada.* 1110–1121. https://doi.org/10.1016/B978-012088469-8.50097-8

[2] Amazon. 2022. *Working with parallel query for Amazon Aurora MySQL.* Retrieved Feb. 8, 2022 from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-mysql-parallel-query.html

[3] Apache Software Foundation 2022. *Apache Calcite.* Apache Software Foundation. Retrieved February 23, 2022 from https://calcite.apache.org/docs/

[4] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA*. USENIX Association, 29–41. https://www.usenix.org/conference/fast20/presentation/cao-wei

[5] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[6] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proc. of the 2020 SIGMOD Conference 2020, online conference [Portland, OR, USA]*. ACM, 1463–1478. https://doi.org/10.1145/3318464.3386129

[7] Amr El-Helw, Venkatesh Raghavan, Mohamed A. Soliman, George C. Caragea, Zhongxian Gu, and Michalis Petropoulos. 2015. Optimization of Common Table Expressions in MPP Database Systems. *Proc. VLDB Endow.* 8, 12 (2015), 1704–1715. https://doi.org/10.14778/2824032.2824068

[8] Lekshmi B. G., Andreas Becher, Klaus Meyer-Wegener, Stefan Wildermann, and Jürgen Teich. 2020. SQL Query Processing Using an Integrated FPGA-based Near-Data Accelerator in ReProVide. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. OpenProceedings.org, 639–642. https://doi.org/10.5441/002/edbt.2020.83

[9] GitHub, Inc. 2021. *gporca.* GitHub, Inc. Retrieved October 7, 2021 from https://github.com/greenplum-db/gporca

[10] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the 1990 ACM SIGMOD Conference, Atlantic City, NJ, USA*. 102–111. https://doi.org/10.1145/93597.98720

[11] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[12] VMWare, Inc. 2021. *Tuning SQL Queries.* VMWare, Inc. Retrieved October 12, 2021 from https://gpdb.docs.pivotal.io/5200/best_practices/tuning_queries.html

[13] Navin Kabra and David J. DeWitt. 1999. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB J.* 8, 1 (1999), 55–78. https://doi.org/10.1007/s007780050074

[14] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[16] Kevin O'Brien, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ronald C. Unrau. 1995. XIL and YIL: The Intermediate Languages of TOBEY. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA*. ACM, 71–82. https://doi.org/10.1145/202529.202537

[17] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of the 1992 ACM SIGMOD Conference, San Diego, CA, USA*. ACM Press, 39–48. https://doi.org/10.1145/130283.130294

[18] Norvald H. Ryeng. 2020. *Refactoring query processing in MySQL.* Carnegie Mellon University. Retrieved February 2, 2022 from https://www.youtube.com/watch?v=u7JOinvbMxc

[19] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proc. of the 1979 ACM SIGMOD Conference, Boston, MA, USA*. ACM, 23–34. https://doi.org/10.1145/582095.582099

[20] Srinath Shankar, Rimma V. Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David J. DeWitt, and César A. Galindo-Legaria. 2012. Query optimization in Microsoft SQL Server PDW. In *Proceedings of the ACM SIGMOD 2012, Scottsdale, AZ, USA*. ACM, 767–776. https://doi.org/10.1145/2213836.2213953

[21] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *Proc. of the SIGMOD 2014, Snowbird, UT, USA*. ACM, 337–348. https://doi.org/10.1145/2588555.2595637

[22] Transaction Processing Performance Council 2021. *TPC-DS Version 2 and Version 3.* Transaction Processing Performance Council. Retrieved October 7, 2021 from http://tpc.org/tpcds/default5.asp

[23] Transaction Processing Performance Council 2021. *TPC-H Vesion 2 and Version 3.* Transaction Processing Performance Council. Retrieved October 7, 2021 from http://www.tpc.org/tpch/

[24] Wikipedia. 2022. *LLVM.* Retrieved Feb. 10, 2022 from https://en.wikipedia.org/wiki/LLVM