Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håvard Ola Eggen & Kristian Andersen Hole

An evaluation of join-strategies in a distributed MySQL plugin architecture

Master's thesis in Computer Science
Supervisor: Jon Olav Hauglid
June 2019

NTNU
Norwegian University of
Science and Technology

Håvard Ola Eggen & Kristian Andersen Hole

An evaluation of join-strategies in a distributed MySQL plugin architecture

Master's thesis in Computer Science
Supervisor: Jon Olav Hauglid
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Abstract

Join-queries in distributed database systems can be executed using a number of strategies that vary in communication requirements, compute-heaviness, and complexity. An evaluation of several join-strategies in a distributed MySQL system is presented. The design and implementation of 5 methods, inspired by literature and state-of-the-art, and a plugin system enabling distribution in MySQL are detailed. Data-to-query, semi-join, bloom-join, hash redistribution, and sort-merge are tested in a number of scenarios, leading to a discussion about their relative performance and viability. In the results, the benefits of parallelism are displayed through hash redistribution. And semi- and bloom-join show how being clever with resources can lead to great performance in distributed systems, and how distributed joins can benefit from different approaches than centralized joins. The more bottom-up implementation of sort-merge shows the value of using a specialized join algorithm in terms of processor utilization. And the more naïve data-to-query strategy is shown to be consistent, but generally the slowest. A discussion about the viability of the plugin architecture is also provided, with potential avenues for further development laid out in future work.

Sammendrag

Join-spørringer i distribuerte database-systemer kan utføres på forskjellig vis ved bruk av ulike strategier. De ulike strategiene kan variere i kommunikasjons-, prosesserings-behov, og kompleksitet. Vi presenterer i denne rapporten en sammenlikning av flere ulike join-strategier i et distribuert MySQL system. Designet og utviklingen av 5 metoder inspirert av litteratur og aktuelle systemer blir fremlagt, samt et plugin-system for å fasillitere distribusjon i MySQL. Data-to-query, semi-join, bloom-join, hash redistribusjon og sort-merge blir testet gjennom flere scenarioer, med en diskusjon rundt deres relative ytelse og egenskaper. I resultatene vises fordelene med å parallellisere utføringen ved hash redistribusjon. Semi- og bloom-join sin gode ytelse viser hvordan smart ressurshåndtering kan påvirke ytelse i distribuerte systemer, og hvordan distribuerte join-spørringer er tjent med å bruke andre teknikker enn sentraliserte joins. Implementasjonen av sort-merge, som er laget mer fra bunnen opp, viser verdien av å bruke en spesialisert join algoritme når det kommer til prosessor-utnyttelse. Og den naive strategien data-to-query viser seg å være stabil, men er totalt sett den treigeste. En diskusjon rundt plugin-arkitekturen blir også fremlagt, samt tanker rundt fremtidig utvikling.

Acknowledgement

First of all, we would like to give a special thanks to Jon Olav Hauglid for providing guidance and conducive discussions throughout this project. Furthermore, the MySQL team at Oracle for hosting us, providing office space, equipment, and a productive environment. We want to specifically acknowledge a few key players of the Oracle staff. We would like to thank Terje Røsten for technical support and making sure we always had a functional setup. Eivin Hatvik for providing access to and assistance with Oracle Cloud. Norvald Ryeng for helping us in corporate and organizational matters at Oracle. Rafał Somla, Luis Silva, and Bogdan Degtyariov in the Connector/C++ team for always answering our technical questions and for being great sparring partners during the implementation phase. And the rest of the MySQL team in Trondheim for moral support and interesting lunch conversations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The volume of traffic and data on the internet is increasing. Popular applications have to handle millions of concurrent users, who might be located anywhere on the globe. As scaling up single servers to tackle this load requires expensive specialized hardware, or in some cases simply cannot be done with current technology, the focus has shifted towards distributed systems and the potential they have for scalability.

Distributed systems are systems that reside and operate on multiple machines. They are typically built on a shared-nothing architecture in which the only communication between nodes is message-passing over a network. For databases this approach has become very popular, giving rise to new paradigms of database systems. Modern database systems attempt to be distributed, while still retaining the functionality of traditional *Relational Database Management Systems* (RDBMS) [1]. This can be a non-trivial task as it involves upholding the rich functionality of the *Structured Query Language* (SQL), whilst operating on datasets split across a number of nodes. The goal of many such systems is to surpass traditional systems in scalability and performance. They do this by using parallel processing techniques, and co-operation between nodes to benefit from the added compute power.

Common performance problems in distributed database systems are related to data locality, i.e. how close data is to where it is going to be processed. Usually one wants to maximize utilization of all the nodes in the system by having them all work in parallel. In order to do this, all of the nodes need to have a piece of the problem to work on. When talking about databases, this means having tables and rows. This cost of transferring data can vary based on how closely the nodes are linked, i.e. on a local network, or geographically spread out on a WAN.

Joins, in particular, tend to be greedy, with regards to data and can cause trouble in distributed databases. This is due to the fact that joins compare all the rows from two or more tables against each other. With a partitioned data set, spread across multiple nodes, naïve strategies for executing joins are bound to generate a lot of network traffic.

1.1 Problem

Joins in distributed database systems require a plan of execution. The plan needs to dictate the distribution of operators, operands, and data in order to coordinate the cooperative execution of a single join between multiple nodes. We will refer to these plans as ”join strategies”.

Different strategies from literature and state-of-the-art vary in complexity and traits such as parallelism and network requirements. The decision about what strategy to use in a given context is based on a number of factors, everything from the hardware used to what data and queries a system encounters. The goal of this project is to evaluate multiple join strategies in a distributed MySQL system. The distributed system for which we will be evaluating join strategies is based on a system we developed for our specialization project. In that project we presented a plugin for MySQL, extending MySQL with a distribution layer capable of executing a limited set of distributed queries. While in this project we will be looking at ways to implement joins in that architecture and evaluate the different strategies against each other.

Chapter 2

Theory

Before delving into existing systems, design, and methods, we will first present some fundamental theory about distributed database systems, queries, and joins. This is to refresh the reader on what distributed database systems are, how they perform queries in general, and why join queries present a problem for these systems. Further on we will present some previous research done in this area, and investigate different distributed join strategies and how they operate.

2.1 Distributed database systems

A distributed database is a network of logically connected databases. Distributed database management systems manage these databases, providing a common interface making the distribution transparent to the outside. ”Distributed database system” refers to both of these together [2].

Common for distributed database systems is a shared-nothing architecture. This means no nodes overlap in any way with any hardware resource. This has the benefit of simplifying the handling of failures and allowing for higher scalability. It does, however, have its disadvantages, namely requiring data transfers over some a network for most tasks, which is an expensive operation. Data locality in distributed database systems is, therefore, an important topic. As each node only handles a subset of the data, communication between the nodes is key. How the data consequently is partitioned among the nodes determines the cost/complexity of accessing and maintaining it, which is a sought-after cost to minimize.

It is this distribution of data which allows for the scalability. With each node handling only a subset, the computational cost per node decreases as the number of nodes increases.

2.2 Distributed queries

Queries in a distributed database may target data on multiple nodes in the system. To access this data, queries, or parts of queries, need to be conveyed to all the relevant nodes.

The simplest way of executing a distributed query is to request all the data referenced by a query from the nodes, and executing the query once all the data is gathered on one node. We will refer to this as data-to-query. The other class of strategies, called query-to-data, involve the nodes receiving and executing part of the query, and returning the resulting data [3]. Strategies belonging to this category have the potential for greater parallelism where nodes can share the workload. This is the most interesting area as there are a lot of different possible strategies and opportunities for optimization.

When multiple nodes execute different parts of the query you get partial results spread throughout the system. If the system is to be transparently distributed and act as one logical database, the results need to be combined before sending it to the client. Aggregation of the results presents a challenge, in that a divided result needs to come together over the network, but also an opportunity in that the aggregation work itself can be parallelized. The partial results of a query may have a much smaller cardinality compared to the tables they are from, e.g. a partially computed average query wherein the nodes return the sum of their rows.

2.3 Distributed join-queries

Join-queries in a distributed system presents a challenge as the entirety of the data being joined need to be cross-compared somehow. This might result in a lot of network traffic, which can be a great bottleneck of distributed systems. A common goal for distributed query strategies is therefore to reduce the number of network messages needed to complete the join [4].

Types of strategies for distributed join-queries:

1. **Parallel algorithms**
   There exist three different classes of parallel join algorithms: nested-loop joins, hash-joins, and sort-based joins [5]. Some algorithms allow for a greater degree of parallelization than others, e.g. the work of a hash join can be shared between nodes by having them hash their local partitions. This is categorized as intra-operator parallelism. They are often designed with pipelining in mind as well, allowing for different operations to be executed in parallel, also called inter-operator parallelism [6].

2. **Knowledge about data locality**
   Strategies can exploit the properties of tables. Such as the distribution of values based on statistics or a predetermined partitioning scheme. Joining on

the partition column of two tables might not require the transfer of any tuples to complete the join, because the system knows the location of those values. E.g. a Person table partitioned on person ID and a Wallet table partitioned on its foreign key relation Person.ID, placing all the matches on the same nodes.

3. **Redistributing the data**
   Redistributing data before executing joins can be a viable strategy and can result in less overall messaging between nodes. A common way of placing data evenly is to use hashing on the join-columns. The goal of the redistribution operation is to have all matching tuples be placed on the same nodes, making it possible to, in parallel, execute the join locally on the nodes [7].

4. **Distributing the work**
   Distributing the actual work of the queries can be done in several ways. It is possible to do work before sending any or little data, depending on how data resides in the system. A semi-join can achieve this by having join-columns shipped and joined with rows locally on other nodes. Other ways include sorting tables in parallel before doing a merge-join, or hashing tables before shipping them in a hash-join. The order of query operators can play an important role also. Doing selections and projections on each node before sending the operands to the join operators can reduce network traffic when transferring tables used for joins. Join-ordering is a well-known join-optimization as well. When you factor in network messaging in a distributed system the impact can be even larger.

In this project we will be focusing on both intra- and inter-operator parallelization of joins, as well as data (re)distribution strategies for optimization. However, we will be using MySQL's native nested-loop join on the local level at each node. This because nested-loop is the only join algorithm available in MySQL. This means that our focus is mainly on orchestrating the distribution of operators, operands and data, and not on implementing parallel versions of serial join algorithms.

2.4 Previous work

Distributed join methods have been discussed and compared in previous research. Some prominent efforts include:

- *Some experimental results on distributed join algorithms in a local network* [8], by Lu, Hongjun and Carey, Michael J., contains a comprehensive evaluation of different types of optimizations such as pipelining, semi-join strategy, and different local join algorithms. They compared the performance of these techniques to what they call ”traditional” join (joins without the optimizations), and found that, in general, network communication is not a dominating factor in a local network scenario.

- *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment* [9], by Schneider, Donovan A. and DeWitt, David J., looks at parallel variants of popular join algorithms such as hash-join, sort-merge and compares their performance and memory usage when completing joins on various datasets. They found that a parallel hybrid hash-join delivered the strongest performance peaks, while a parallel sort-merge was the most stable throughout the different tests.
- *Advanced Join Strategies for Large-Scale Distributed Computation* [10], by Bruno, Nicolas and Kwon, YongChul and Wu, Ming-Chuan, looks at join graph topologies involving multiple joins to find the best execution order for a parallel system. They also introduce methods to perform ”SkewJoins”, which are join methods that seek to reduce the impact that significant data skew can have.

These research efforts have investigated distributed joins using a top-to-bottom implementation approach where algorithms specialized to handle distribution has been deployed. Our project seeks to provide join strategies on top of a regular MySQL instance, thereby foregoing any local algorithm optimizations and rather exploring the movement of data and operators between the nodes. Although, some aspects of the methods used and results in these papers are still interesting and valuable to our efforts.

2.5 Methods

We will in this section present a few different methods from research on distributed joins. The methods highlighted are the methods we found to be the most interesting and applicable to this project.

2.5.1 Distributed hash-join

A simple way of executing a distributed join is through hashing. A hash-join on one machine works by reading one set, R, building a corresponding hash table, and then passing/looping through another set, S, and continuously joining matches from S in the hash table [7]. When building the hash table it may overflow, meaning there is not enough room in the working space for it. In this case, overflows are written to an interim file to be processed later on. Incoming records from S not matching in the current hash table will also be written to an interim table to be processed later. The distributed algorithm is very similar in its workings, except it also has to distribute data to nodes [9]. This distributed hash-join works by:

1. Hash join-column of table R to route tuples to nodes.
2. Build hash tables of R at each node. Overflows are written to a local file.

3. Route rows from table S using the same hash function.
4. Join immediately on arrival at a node, mismatches written to overflow file.
5. Move on to the next chunk of R from overflow file and repeat join with S overflow.

As hash-joins are not supported in any current version of MySQL, a full parallel hash join implementation is out of scope for this project. Looking at the parallel hash-joins in research is still worth-while as the techniques employed for data redistribution and partitioning are still highly relevant for our project.

2.5.2 Distributed sort-merge join

Sort-merge join works by sorting the tables on the join column, and merging together the ordered sets, creating matches as it goes. One distributed variant, as described in [9], parallelizes the work by having each involved node sort its part of the tables locally. They include an initial data redistribution step, using hashing, to be able to complete the join locally on the nodes. The nodes then perform a simple merge-join locally and can ship its part of the final result.

But it is also feasible without the hashing step. If the data is not aligned by join column values, an n-way merge of the sorted data streams from the nodes can be performed, and matches can be produced continuously by the merging node [11].

”Interesting order” is a shortcut when performing merge-joins. If the system has information about the ordering of the table, and they happen to be sorted by the join columns, the sorting step can be bypassed. This also allows for pipelining, which otherwise is not possible as sorting is a blocking operation [12].

2.5.3 Semi- and Bloom-join

Semi-join is a reduction operator that works by joining two tables on a column, but only leaving the matched result of one table as opposed to retrieving the aggregated result from both. For distributed databases, this reduction is handy as only the join-column needs to be sent from a node to another to retrieve matches. This means smaller message sizes, thus, reducing the network volume. When the result of a semi-join is returned the full join can be finalized with the rest of the columns on the first node [13].

The idea of semi-joins and sending a minimum amount necessary to execute a join has been expanded upon in research. Instead of sending the columns themselves, methods for sending only a bit array representation functioning as a filter has been presented.

A bloom filter is a bit array generated by passing elements of a set through multiple hash functions denoting which bits to flip ”on”. When all the elements of a set have been passed through, we can use the same algorithm to determine if

an element is a member of the original set. If an element is hashed to a position containing a zero by any of the hash functions, it is not part of the original set. False negatives are not possible with bloom filters, but false positives are. Thus, a bloom filter can attribute membership to an element not actually in the set, but will never claim a member is not one [14].

Bloom-join is a variant of semi-join using bloom filters to reduce the opposite table's join-column to what the filter deems potential matches [15]. It works by:

1. Generating a bloom filter of table S's join-column.
2. Shipping the filter to the other node.
3. Pruning the join-column of table T on the second node by applying the filter.
4. Shipping back the potential matches, and joining them with table S.

Semi- and Bloom-joins are very interesting strategies for our project as they are inter-operator optimizations, but still, very much affect the execution of the joins themselves by splitting them into multiple join operations.

Chapter 3

State-of-the-art

In this chapter, we will look at modern distributed database systems supporting SQL and explore the different strategies they employ for executing distributed join queries. Looking at contemporary systems can give us an insight into what strategies are used and tested in the real world, and can impact the choices we make about what methods to pursue in this project.

3.1 ClustrixDB

ClusterixDB is a distributed relational database, based on a shared nothing architecture, aiming for great scalability by using highly parallel querying methods[1].

ClustrixDB optimizes their distributed joins by having tables' columns indexed by means of a hash function. Indices are placed on nodes using the hash function and serve as pointers to the rows. When a primary key is indexed, the index is stored on the same node as the row it points to. This means that when joining two tables, by means of hash-join, the hashed values already exist on the correct nodes and simply need to be paired. This also means that if a join is on two tables' primary keys there is no need to move any data between nodes to complete the join[16].

If a join is between table A's primary key and a non-primary key index of B, the join operator is shipped to the node containing the rows for A. This way A can execute the join based on A's rows and the indices of B. At the end the only data that needs to be sent between the nodes are the rows of B that satisfy the join condition. This is a semi-join, but with the performance improvement of having the join-columns already in place on the nodes. The work can be done in parallel, with minimum messaging, as the hashed indices ensure that each node already has the information to complete its part of the join. In the end, unicast messaging is used to move the required data directly to the designated manager node[17].

[1] http://docs.clustrix.com/display/CLXDOC/Frequently+Asked+Questions

With this strategy, ClustrixDB demonstrates the value of preparation and maintenance of strict and plentiful metadata in a system. Every subsequent query receives the performance payoff, from this initial and ongoing survey of the data.

3.2 MySQL Federated Storage Engine

The MySQL Federated storage engine is a storage engine which ships with MySQL, and can be used as a replacement for e.g. InnoDB. Its purpose is to enable simple remote access to tables residing on other MySQL servers. When enabling the federated engine, users can reference non-local tables in queries. This is achieved by creating a local reference to the table with the network location, database name, and the table schema. When a query targets a federated table, MySQL fetches all the rows from the remote server that the table resides on. In other words, it brings the data to the query [18]. This is a simplistic approach, of the form ”data-to-query”, in which any query, including joins, can be executed on a distributed dataset without modification, but it also has downsides. This method can potentially lead to network overload and/or running out of memory, having to write to disk when the dataset is big.

3.3 MySQL Cluster

MySQL Cluster is a system providing sharding and clustering of MySQL through the use of the *Network Database* (NDB) engine. This is a shared-nothing distribution aware storage engine[2].

To parallelize the work of a distributed join, MySQL Cluster uses something they call *Adaptive Query Localization* (AQL). It works by querying the storage-engine nodes with a specification of the needed data. This specification includes what tables and columns are being accessed, access type (primary key, full table scan, etc.), any relations between the tables. They also push down selection and projection operators [19]. This means much of the work of the join can be done in parallel on the nodes.

MySQL cluster’s distribution capabilities live in the storage engine layer, using their distributed storage engine NDB. Since our project operates on levels above the storage layer, the methods are not directly applicable, but some important aspects to take away is the effect of pushing down the knowledge of the data and query to the distribution aware components.

[2] https://www.mysql.com/products/cluster/faq.html

3.4 CockroachDB

CockroachDB is a distributed SQL database built on top of a strongly consistent key-value store[3]. It focuses on consistency and reliability and aspires to be ultra-resilient against all types of failures.

For its distributed joins, CockroachDB uses a combined hash-redistribution and sort-merge join strategy. To use this method CockroachDB requires both tables of a join to be indexed and sort on the join-columns to be used.

The first step it takes is to redistribute the data of its sorted table partitions using a hash function. Each node then sets up a merge joiner and starts receiving the data streams of its designated hash value range. Merging happens by comparing the rows from both tables until a match is found. Once a match is found it keeps reading rows to find all the rows that match with the currently matched value. When no more matches for the current value is found, the merge joiner returns a cartesian product of all the matching rows and then moves on to the next set of matches. The incoming data streams are thereby continuously merged, joined and streamed to the master node. If a partition is not sorted the receiver node needs to wait for the entire stream to then sort it, thereby blocking the pipeline. This is acceptable for small tables, but for large unsorted partitions, Cockroach uses a hash join instead [20]. This indicates that sort-merge join is a limited strategy, as CockroachDB only uses it in situations where there are heavy constraints on the data.

3.5 Google Spanner

Spanner is a cloud database focused on delivering high scalability and strong consistency throughout Google Cloud[4].

When joining two independently distributed tables, Spanner employs a data redistribution strategy to reduce the overall number of messages between nodes. First, it analyzes the query to figure out what shard key ranges are in the scope of the join, and by extension which nodes need to be sent a part of the left join-operand. It then makes a set of subqueries to retrieve the needed rows in batches from the nodes, called a distributed union. These queries are modified by pushing down operators, like projections and selections, to reduce the size of the result. The distributed union is then broken into shard specific minimal batches, based on shard key ranges, and sent out to the relevant nodes. The nodes now receive a minimal set of rows needed to execute the join with their local set of rows [21].

Spanner's strategy is based on doing preliminary redistribution work in order to align the data before executing the join. It is a simple and general strategy that does not alter the join execution itself, and it aligns well with partitioning schemes and metadata about data-locality. However, it is limited in that it is only a data

[3] https://github.com/cockroachdb/cockroach
[4] https://cloud.google.com/spanner/

redistribution strategy, and is not participating in the work of the join itself, like e.g. a distributed hash-join.

3.6 MemSQL

MemSQL is a distributed main memory SQL database using RAM as its primary storage, and logs for durability[5]. Ir provides row storage tailored towards transactional workloads (OLTP) and column storage for analytical workloads (OLAP).

MemSQL operates with a concept called ”reference tables” to provide efficient distributed joins. The reference tables are fully replicated on every node, meaning that a join between a table and any number of reference tables can be executed locally on the nodes using the replicas [22]. This method, however, requires some planning on the users' part in deciding which tables are reference tables. When joining a table with a non-reference table MemSQL needs to move data about, however, they try to minimize the amount by aligning shard-keys between big tables. This is done by looking at the signatures of the shard keys and aligning them such that as much of the join as possible is done locally on the nodes. Users can influence this as well in choosing shard keys.

Giving users the ability to impact join performance and distribution strategy is smart as it takes advantage of advanced human knowledge about the query patterns and use-cases that need to perform well, especially for an OLTP workload where the set of queries is predictable and rarely change.

3.7 Apache Ignite SQL

Ignite SQL is a distributed SQL database, using tiered storage layers, meaning it prefers to store data in main memory, but also has full support for using disks[6].

Ignite SQL supports distributed joins in three different capacities. Similar to MemSQL, if a table is replicated, it can immediately be joined with any table as one of the operands exists on every node. The next level is referred to as collocated joins. Instead of replicas of tables, here they require that the keys being joined on are collocated. Meaning an index of the keys is replicated on every node. This way the join operation can be completed using the local copy of the index. Then the rest of the requested data and the final join result can be aggregated at the master. This is similar to the way ClusterixDB does it, which is to say it is a pre-shipped semi-join.

The final variety is non-collocated joins. This is when a join between partitioned tables without replicas or collocated keys is requested. In this case, the system requests all the needed data from the other nodes, just like the MySQL Federated

[5] https://docs.memsql.com/introduction/latest/memsql-faq/
[6] https://apacheignite-sql.readme.io/docs

Engine. This is not enabled by default and needs to be activated upon configuring the system. This is because it is considered to have really bad performance due to the network round-trips and traffic it can generate [23].

3.8 Conclusion

Looking at the systems in this section has shown us that processing and keeping knowledge about data is valuable. As well as the fact that no single join strategy is a solution to every problem. Supporting multiple strategies, choosing the best one for a particular query, and/or storing data in a way that suits a particular strategy are common denominators to the way many of these systems operate. We will use the knowledge of these systems when designing our join strategies, partitioning schemes and overall architecture.

Chapter 4

Design

In this chapter, we will present the design of a distributed MySQL solution based on plugins, capable of performing selected JOIN statements using different strategies. First, we will talk about the legacy system from our specialization project and how it works, before presenting the design of the different strategies, and why they are interesting, and lastly discuss how the legacy system must be extended to handle the strategies.

4.1 Legacy system architecture

The system we will make for this project is based on the system we made for our specialization project. In this section, we present a birds-eye-view of the legacy system's design. The system we made then was a MySQL plugin enabling distribution by connecting multiple MySQL instances, keeping metadata about partitioned data and rewriting queries to run distributed on all nodes. It was made using the plugin API exposed by MySQL. This means the plugin can access some MySQL internals, e.g. the parse-tree for a query, through the plugin interface. So it is tighter integrated than an application layer system, but not completely integrated with the DBMS itself either. Communication between nodes is done using the MySQL protocol, and queries target the MySQL instances directly.

The design choices in the rest of this chapter are made in the context of the plugin architecture, and the inherited philosophy of the legacy system design. This means using existing MySQL functionality to do the heavy lifting, avoiding re-inventing functionality and keeping the plugin a lightweight distribution layer.

At the end of the specialization project, the system was capable of handling a limited set of distributed queries, more specifically SELECT, COUNT, SUM, and AVG, but no joins. Thus, we need to extend it with implementations of the different join strategies.

4.1.1 Life of a query

An overview of how the legacy system is put together is best explained with how a query passes through the system. A diagram of this process can be seen in figure 4.1.

Initially, it starts like any other query in a normal MySQL environment. A user opens a connection to a MySQL instance, which in this case is an arbitrary node in the system that has the plugin installed. From here a query is entered and sent to the query parser. An *Abstract Syntax Tree* (AST) is created before triggering and passing the result to the plugin. The normal flow is then interrupted and blocked, and the distributed system takes over.

From here on a multitude of different operations are performed. The plugin starts by checking whether the tables in the query are known to the distributed system. Further on it evaluates which nodes to include, queries the relevant nodes, creates interim tables and stores the results in them, and finally rewrites the original query to instead target these.

After this, the blocking is released and the execution continues as normal, but with the rewritten query instead.

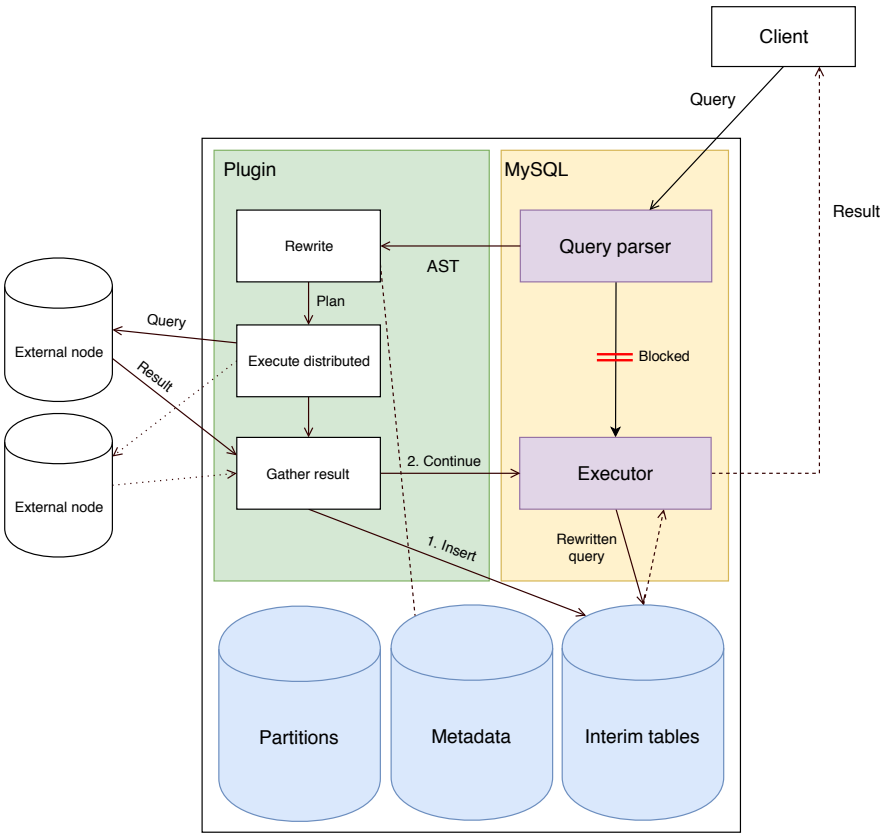


Figure 4.1: Life of a query

4.1.2 Plugin overview

Figure 4.2 shows the overall architecture of the system where the green area is the plugin, consisting of four components: distributed query rewriter, distributed query manager, plugin hooks and MySQL driver. The *Distributed Query Rewriter* (DQR) walks the parse tree and plans the execution of a distributed query. The *Distributed Query Manager* (DQM) is responsible for overseeing the execution of the node-specific queries generated by the distributed query rewriter. The plugin hooks represent the modules interfacing with the API, e.g. internal queries, defining the plugin. The MySQL driver, namely Connector/C++[1], is used to query external nodes. Also shown in the figure are the different databases defined. One for metadata, one for interim tables, and the last one for the user data partitions.

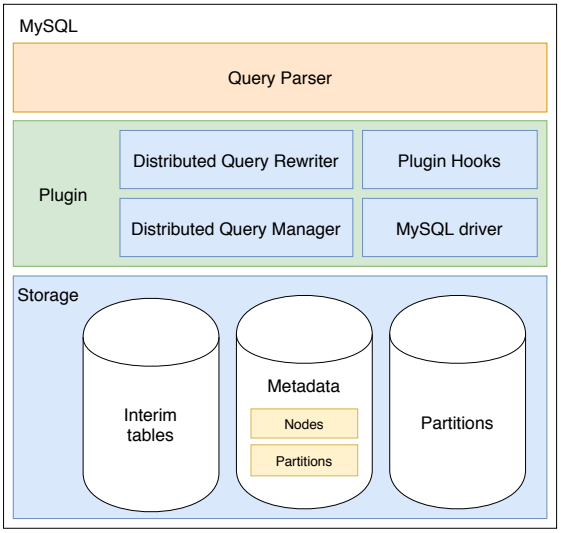


Figure 4.2: Architectural overview of the plugin

4.1.3 Metadata model

The metadata model consists of three entities: nodes, partitions and shard keys. A shard key in our system is defined by a column name, a start value, and an end value. A partition can be any subset of rows in a table, including the complete set of rows, and will have exactly one shard key denominating which ones. Thus, in our metadata, a partition is defined by a shard key id, a table name, and a node ID. A node is defined by a vector of host IP address, port, database name, and user credentials. This addressing scheme is similar to MySQL Federated Engine's (section 3.2), and gives our system the ability to refer to any MySQL table in existence. This opens up for a lot of additional use-cases outside of partitioning a database.

[1] https://dev.mysql.com/doc/connector-cpp/8.0/en/connector-cpp-introduction.html

For example, logically linking two existing databases with no previous relationship and executing queries across the two.

The information about partitions is stored on the MySQL instances as tables, referred to as the metadata tables. The plugin accesses and manages these tables through internal querying. There is one table for nodes that exist in the system, one table for partitions, and one for shard keys. The model for these tables can be seen in figure 4.3. Given that the metadata tables are regular SQL tables we are able to use the full expressiveness and power of SQL when accessing them. This means that we can, based on an incoming query, select only the partitions that are targeted by it.

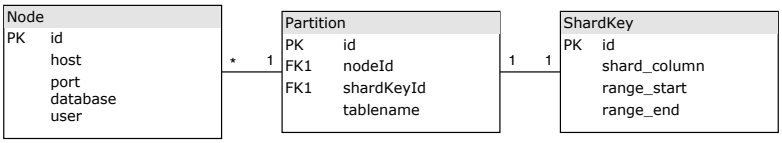


Figure 4.3: Metadata model

This marks the end of the section about the legacy system, serving as a basis for the new system. Further on in this chapter, we present the design of all new developments.

4.2 Join types

There is a multitude of possible join queries available in a typical relational database system. Inner join, left-, right- and full outer join, comparator-based join and so forth[2]. As complete support for SQL is not a goal of this project, we will focus our efforts on the typical and implicitly inferred inner join. In addition to this, we must also evaluate all methods on the same basis, thus, we need to perform joins which are supported by all the methods. E.g. once hashing is involved you are limited to equi-joins, because once you hash a sequence of numbers the transitive relationships of the $<$ and $>$ operators cannot be guaranteed. Therefore we will be limiting our design to inner equi-joins.

4.3 Join strategies

To decide which strategies we will implement and evaluate, we have taken applicable strategies from both the theory and state-of-the-art chapters and categorized them into 5 types of ”pure” strategies. For example, the hybrid strategy of CockroachDB is a combination of two ”pure” strategies; hash redistribution and sort-merge join.

[2] https://dev.mysql.com/doc/refman/8.0/en/join.html

This is to make our research more informative in terms of the specific strengths and weaknesses of a strategy in and of itself.

The 5 main join strategies we have landed on are the following:

1. Data-to-query A simple method serving as a baseline for evaluation.
2. Semi-join A method that is well suited to reducing network in a distributed join.
3. Bloom-join A modified semi-join using Bloom filters to reduce network volume in exchange for generating and processing the filters.
4. Hash redistribution A data-redistribution strategy for parallelizing the work of the joins.
5. Sort-merge join A method making use of the existing data distribution and parallelizing its work by sorting data in place on each node.

We will not evaluate external products, but rather evaluate the methods and strategies they utilize. This is because the project is about comparing strategies against one another in the plugin architecture. E.g. MySQL Federated Engine works similarly to what we will make for our data-to-query strategy, but since it is not using the plugin architecture we will be using for the other strategies we will not be testing it.

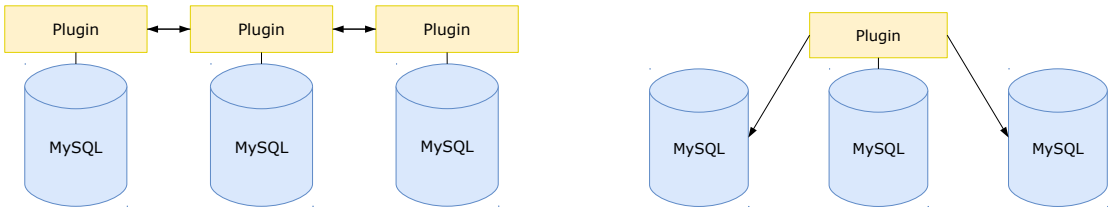
4.4 Node communication

As this is a distributed system it consequently requires some sort of communication between the nodes. The legacy system relied on sending regular SQL queries to the nodes in the system to orchestrate its distributed execution. This worked fine for the limited subset of queries it supported, however, some of the join strategies we will explore in this project have more complex networking requirements.

Some strategies can be purely expressed using SQL syntax and subsequently only need the plugin-to-mysql contact, like in the legacy system. These strategies can be implemented using proper queries between the nodes, communicated through the MySQL protocol. E.g. a semi-join can be easily expressed in SQL as it is built using only projections, joins and unions.

Other strategies' requirements go beyond the functionality offered by SQL. E.g. bloom-join cannot be communicated using only SQL, because of the use of the bloom filter data-structure and functions not present in SQL. To accommodate these strategies we had to create an extra-SQL communication mode, we will refer to this as plugin-to-plugin communication.

An illustration of the two communication modes can be seen in figure 4.4.



(a) Plugin-to-plugin (b) Plugin-to-MySQL

Figure 4.4: Different communication modes

For plugin-to-plugin we considered a couple of options. Namely a custom protocol using TCP or using the SQL comment syntax and piggyback queries between the nodes.

Using TCP would allow us to reliably send any data, at any time, between the plugins, but would likely add a lot of complexity as we would have to keep TCP servers and clients, and by extension the plugin, alive outside of the duration of a query. It would also require us to implement the transportation of result-sets, which are already first-class citizens in the MySQL protocol.

So, keeping by our philosophy of utilizing MySQL as much as possible, taking advantage of the pre-existing client-server communication offered by MySQL makes more sense. The MySQL protocol deals in queries and result sets, which is what our nodes will be sending most of the time. It would also simplify the life cycle of the plugin as it would only be active in the ”call and response”-context of a query. For the extra-SQL communication, we will expand on the MySQL protocol by encoding data into comments in our SQL statements. These comments will then be parsed by the plugin on the receiving end of the query.

4.5 Data-to-query

The most naïve method of performing distributed joins is data-to-query. It works by simply retrieving the relevant data from external nodes, and joining once all the data is collected. MySQL Federated Engine is, as mentioned earlier, one system using this approach. The drawback of this approach is, instead of utilizing the combined potential processing power in the distributed system, it uses it solely as distributed storage space, only requesting data for local processing. One consequence of this is it involves a lot of networking, which can be expensive.

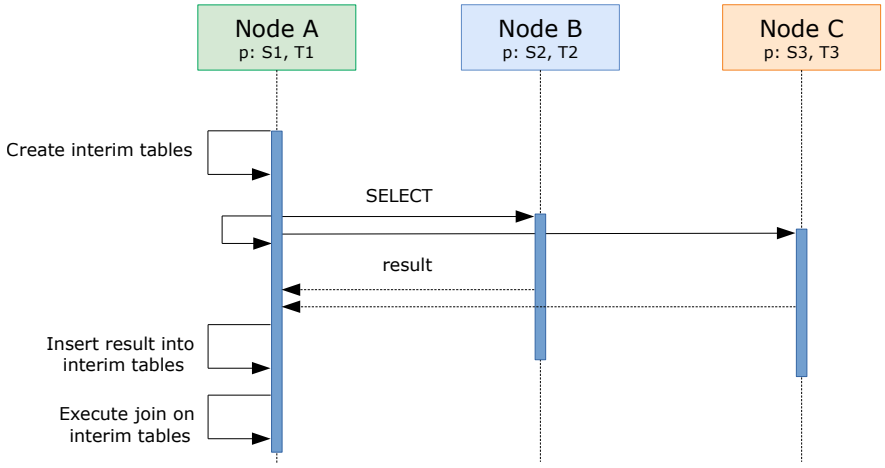


Figure 4.5: Sequence diagram of data-to-query

Figure 4.5 illustrates the sequence of operations and communication for the data-to-query strategy. To handle all returning datasets the master node first needs to create interim table(s) for temporary storage. It can then perform its requests towards nodes containing wanted partitions, and store the results. Only after receiving all requested data can it perform the join query. One optimization our data-to-query will have is projection push-down. By pushing the projections of the original query to all sub-queries the master node will only retrieve the columns needed to complete the query. This is similar to what MySQL Cluster (section 3.3) does, and its a general optimization we will apply to all strategies.

The benefit of the data-to-query strategy, with regards to evaluation, is that it serves as a good benchmark against other methods. This is because of the ease of implementation and the fact that it immediately supports all of SQL, compared to the more intricate methods which will be further discussed in the sections below.

4.6 Semi-join

The idea behind semi-join is to limit data transfer by only sending the join-column, instead of the whole table. This tackles the main disadvantage of Data-To-Query, namely data transfer. To illustrate how semi-join works we will look at it from both a relational algebra and design perspective.

4.6.1 Relational algebra

A semi-join can be expressed in relational algebra, as can be seen in figure 4.6. The diagram shows the distribution of operators between nodes A and B, and which nodes that are responsible for performing the operation. In the depicted scenario

there are two tables, T and S, with T partitioned into T_1 and T_2 and placed on node A and B, and S residing solely on node A.

```
1 SELECT * FROM S JOIN T ON S.j = T.j;
```

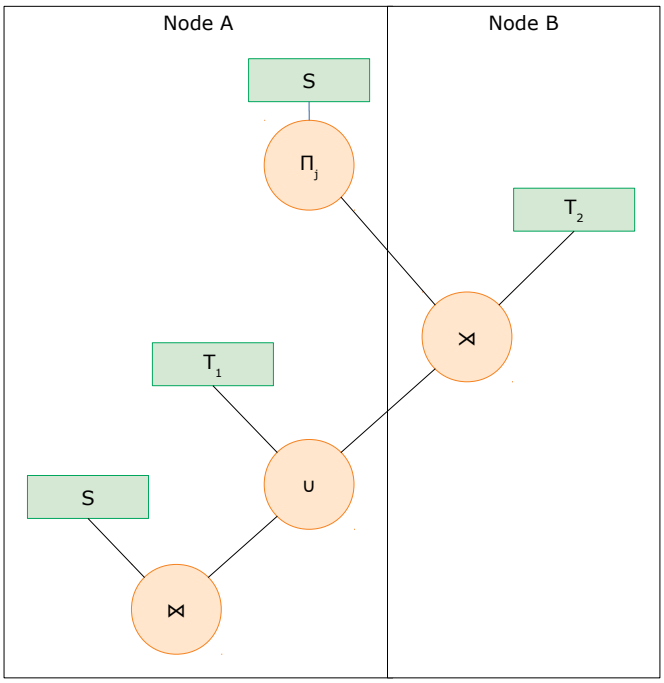


Figure 4.6: Relational algebra of semi-join with 2 nodes

The first operator is a projection of table S returning only the join-column. This result is joined with the partial table T_2 on the join operator on node B, the semi-join operator. The resulting set of rows is the minimum amount of rows that need to be transferred from B to A to complete the join on A. A union of the semi-joined T_2 rows and the T_1 partition on node A is performed and finally is joined with table S.

4.6.2 Our semi-join method

How our semi-join method works is dependant on how the involved tables are partitioned and distributed. There are two cases it must solve. It is the case where one of the two tables in a join is partitioned, and the case of both of the tables being partitioned. Both of these cases will be elaborated upon in the sections to come.

4.6.3 One partitioned table

In the case of only one being partitioned, we have the same scenario as depicted for the relational algebra, and we get the sequence diagram 4.7. All queries are initiated by node A, with node B and C simply fulfilling them and responding as normal MySQL servers.

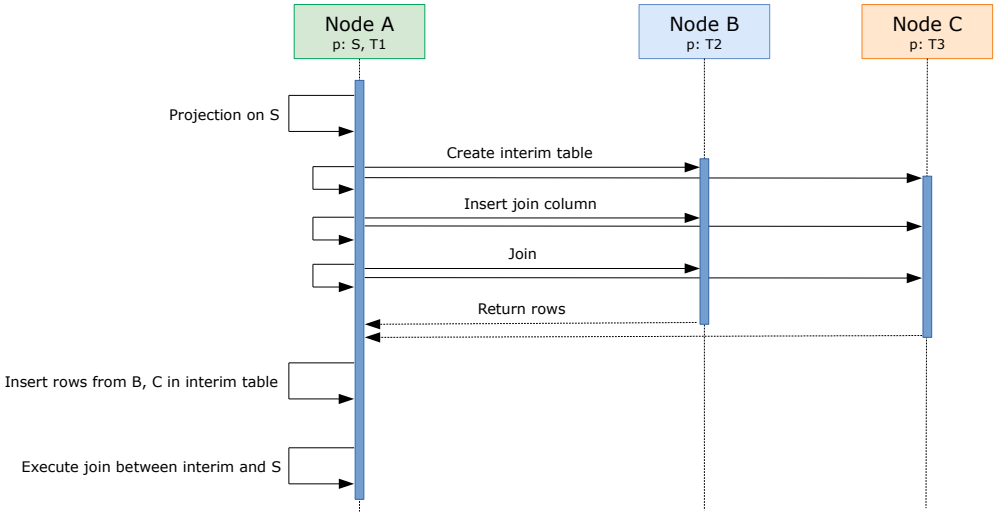


Figure 4.7: Sequence diagram of a semi-join with 3 nodes and 1 partitioned table

Node A will first perform a projection of table S on the join-column specified by the query. Then it connects to the nodes that have a partition of table T, creates an interim table, and inserts the result from the projection of S. When it receives a response from the insert query, it queries the nodes with rewritten variants of the original join query targeting the interim tables and the T partitions. This query will project table T through the selectors of the original join query to reduce the size of the shipment to the bare minimum. Upon receiving these subsets of T, node A inserts them into its interim table. Finally, node A rewrites the original join query to target the interim table instead of T, and lets MySQL return the result to the client.

4.6.4 Multiple partitioned tables

If both tables involved in the join are partitioned and distributed, the problem becomes more complex. This because there is not one single join-column to send out to the nodes for the semi-join operation.

Our solution to this is using recursion, calling the method again, but with one less active partition. Sequence diagram 4.8 is an example of this, with two partitioned tables. Using this method incoming queries accessing n partitioned tables can be broken into m number of $n = n-1$ queries, where n is the number of accessed tables

which are partitioned, and m is the number of nodes a selected partitioned table resides on. In the case of $n = 2$, this means it creates $m = 3$ queries, treating each local S_n partition as a ”complete” S, and performing a semi-join on this as explained in the previous section. After each node completes, the union of results is returned to the initial node A. It is worth noting that to distribute the workload the table which is partitioned across the most nodes is the one chosen for each step to be treated as ”complete”.

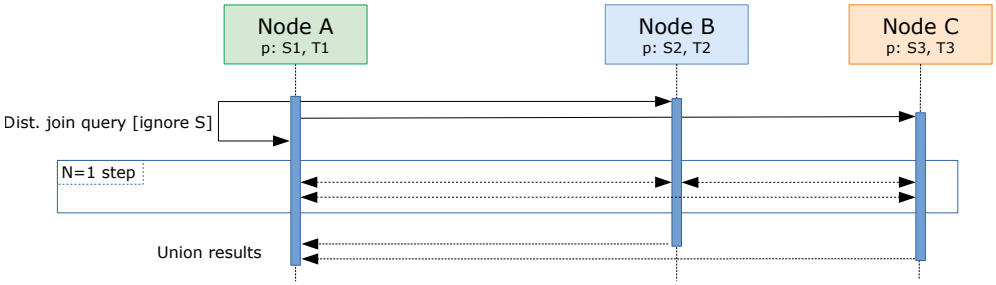


Figure 4.8: Sequence diagram of $n = 2$ semi-join with 3 nodes and 2 partitioned tables

This design utilizes a simple form of plugin-to-plugin communication. Using recursive distributed queries means the only extra communication is a simple argument containing a table name. Therefore, using comments to encode this communication makes sense for this strategy.

To explain how the recursion in this method works we can look at the pseudo code in figure 1.

```
1  semi_join(partitions):
2
3      n = count_non_local(partitions)
4
5      n == 1:
6          return execute_semi_join(partitions)
7
8      n > 1:
9          set_ignore_non_local_flag(partitions[0])
10         result_set = []
11         for node in nodes:
12             result_set += node.semi_join(partitions)
13         return result_set
```

Listing 1: Pseudo code for recursive semi-join

4.7 Bloom-join

Like semi-join, bloom-join's purpose is to reduce data transfer. However, bloom-joins' angle of approach is a bit different. While semi-join sends a join-column, bloom-join instead generates and sends a bloom filter of the join-column. This is more space efficient, but may result in more returned rows because of false positives.

The methods have many similarities in how they work, in fact the $n > 1$ steps are identical for both. They differ however on the $n = 1$ step, where the method-specific operations are performed. We will focus on the $n = 1$ step for the design of bloom-join, as $n > 1$ is already discussed in section 4.6.4.

Since bloom filters are not natively supported by MySQL, plugin-to-plugin communication is required. This means the filtering algorithm will reside in the plugin itself and be remotely invoked by other instances of the plugin. An effect of this is a requirement of a master/slave configuration. This is because some plugins need to perform operations for others, contrary to MySQL supported operations where MySQL can be invoked instead.

Below we will design the processes in this step. We will design the two roles, master and slave, separately.

4.7.1 Bloom-join master

The bloom-join master's responsibility is to coordinate the execution of the bloom-join. These responsibilities can be divided into two main steps: Firstly the creation and distribution of both the bloom filter and interim table names for the results to be stored in. Secondly the retrieval of results from slaves and execution of the join on the retrieved data. The processes can be seen in figure 4.9 from the view of node A.

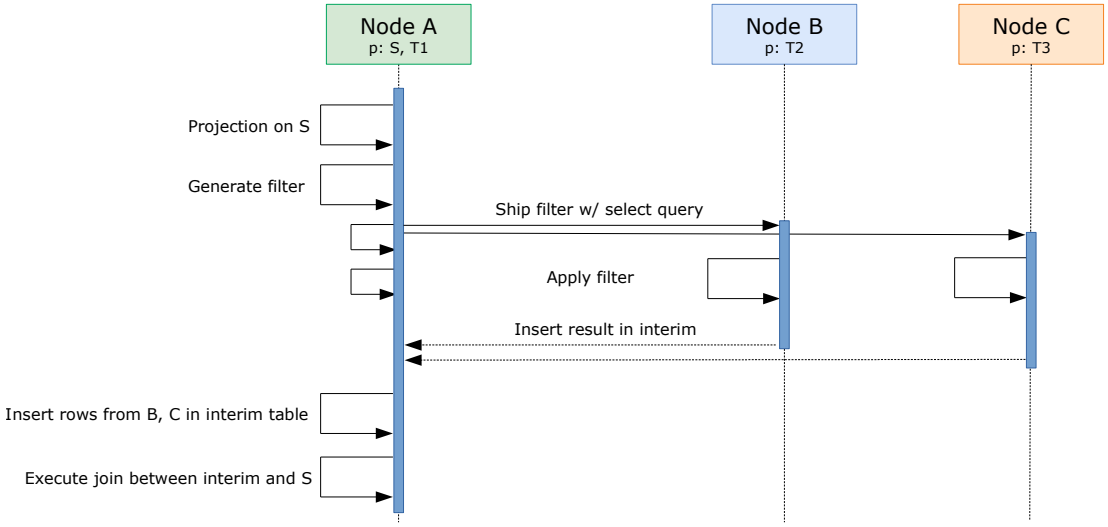


Figure 4.9: Sequence diagram of $n = 1$ bloom-join with 3 nodes and 2 partitioned tables

We will use comments as the mode of communication for this strategy as well, even though we are sending a lot more non-SQL data than in the semi-join strategy. This is for simplicity of implementation, and because of the fact that the extra-SQL communication is done in one request, like the recursive semi-join step, making the call-response context of a query a good fit. This means the master can query the slave, and the slave can perform necessary operations to both create and populate the denoted tables before the master moves on to its join.

4.7.2 Bloom-join slave

The slave is a small remote procedure at the nodes invoked by the master's query. Its task is to read a local partition, filter it with the received bloom filter, and then to place the results in an interim table as specified by the master. This can be seen in figure 4.9 from the view of node B and C.

4.8 Hash redistribution strategy

Hash redistribution works by moving corresponding data to the same node. It is an elegant way of evenly distribute data (as long there is no large skew in values) to all nodes involved in the join, thus, it can utilize the combined power in the distributed system well. It does, however, have the drawback of it potentially requiring the movement of a lot of data around the network before executing the join.

The method is divided into two steps: first performing the redistribution, and then executing the join. It consequently starts off by having every node relevant to the

query hashing its partition(s) and transferring the results to the hash's corresponding node. When everything is redistributed the actual join can be performed. Since everything that could match in the join now is on the same node, each node can work separately, hence the utilization of resources, before returning the result to the initial node. This can be seen in figure 4.10.

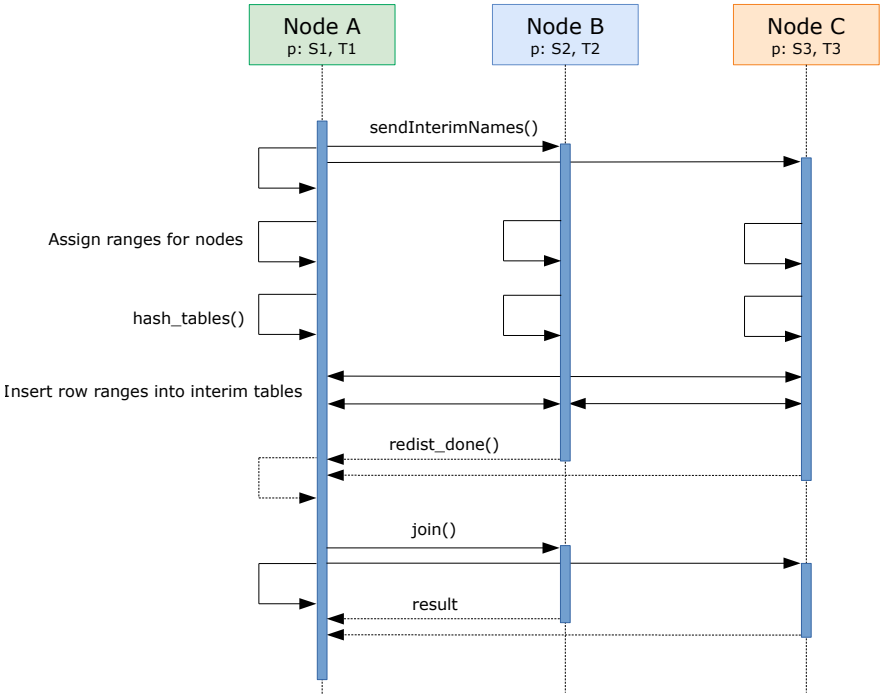


Figure 4.10: Sequence diagram of hash redistribution

Similar to bloom-join, the hash redistribution strategy uses the master/slave pattern for plugin-to-plugin communication. The master is responsible for initiating both the redistribution and the join on each node, with the slaves actually performing the operations.

4.8.1 Hash function

Important for hash-based methods are, of course, the hash algorithm used. These come in different flavors, all depending on the task they are used for. They range from those designed to be secure, to those meant for general usage and to simply be fast. Secure algorithms are mainly used for cryptography applications, which have requirements for algorithms being both computationally heavy and memory heavy to safeguards against attacks. Example of these are bcrypt[24] and scrypt[25]. These are, however, not interesting for this redistribution method, namely because they are designed to be slow. This is not ideal as there will be performed a lot of hashing,

which will cause unnecessary overhead. This leaves the algorithms designed to be fast. Since we will use the hash to nothing more than placing data, these suits the redistribution method well. In section 5.5.3 we will evaluate a few different algorithms and choose a specific one.

4.9 Sort-merge join

Sort-merge is similar to data-to-query in that it retrieves all the data as its first step, however, it does it a bit more cleverly. It performs a sorting operation on all nodes in parallel so it can perform a join contiguously when retrieving the data, not afterward as is the case with data-to-query. Sort-merge also differs from all the other strategies in a very important aspect; it can not use the native MySQL nested-loop join at the local level. Due to the nature of sort-merge, a nested-loop would make it exactly like data-to-query where the order of rows does not matter. So instead, our sort-merge will offer its own merging algorithm to do the join operation. This makes sort-merge more of a top-to-bottom strategy, similar to ones from earlier research (see section 2.4), in comparison with the rest. This is interesting because it offers an insight into the differences between our lightweight orchestration strategies and a more specialized one, in both implementation and evaluation. The design is divided into three parts: Sorting, K-way merge, and finally, merge-join. We will address each of these in the sections to come.

4.9.1 Sorting

The sorting work of the sort-merge strategy will be done in parallel between all the nodes that have active partitions of the join tables. This will be achieved by concurrently sending out `ORDER BY` SQL statements targeting the partitions. For example:

```
1 SELECT {projections} FROM Person ORDER BY Person.homeworld;
2 SELECT {projections} FROM Planet ORDER BY Planet.id;
```

The master sends out these queries and processes the results as they arrive. The sequence of this strategy can be seen in figure 4.11.

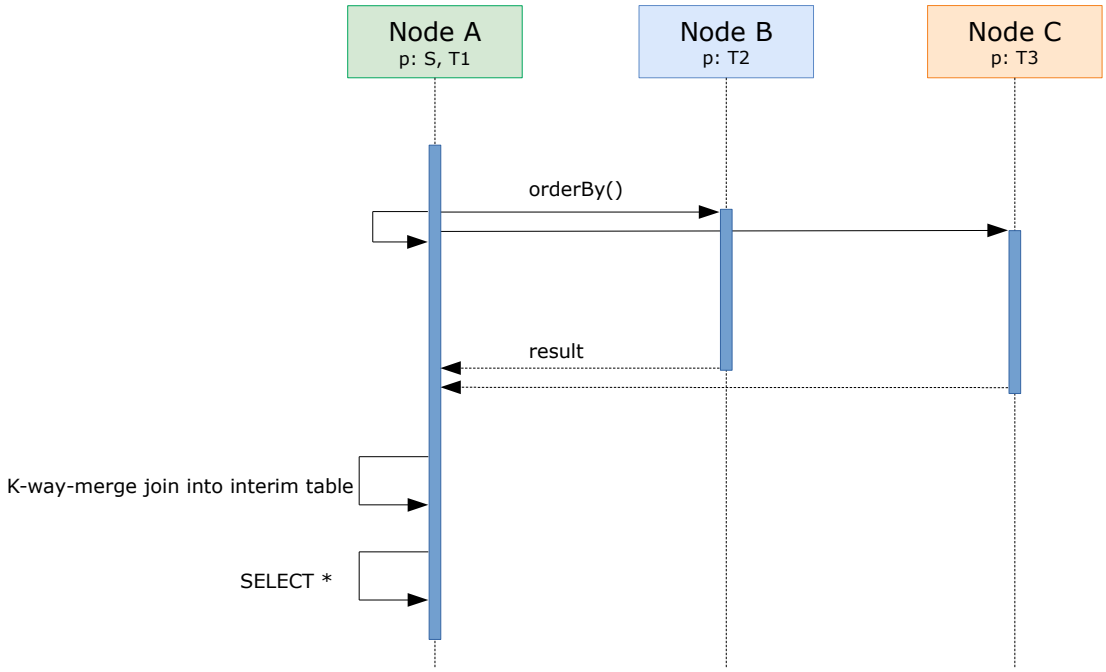


Figure 4.11: Sequence diagram of sort-merge

4.9.2 K-way merge

When querying k partitions with the `ORDER BY` queries we will receive k streams of ordered rows, belonging to either table S or T. In order to join the tables, we first need to unify the streams, making two total result streams, one for each table, where we contiguously can fetch the next row in the correct order. We will do this first step using a k-way merge algorithm.

A k-way merge is described as the process of combining k ordered sequences into a single ordered sequence. The core of the problem is the comparison of k sequences when selecting the next output. The simplest approach is comparing the top of all the sequences each time. Doing this results in a cost of $k - 1$ comparisons to find the next output. However, there is a better way. Using a binary heap reduces the number of comparisons needed by keeping sequences with ”better” next values near the root of the heap. With a binary heap, we need a maximum of $\log_2 k$ number of comparisons for each fetched row to maintain correct heap order[26]. We will use a heap to get an efficient k-way merge.

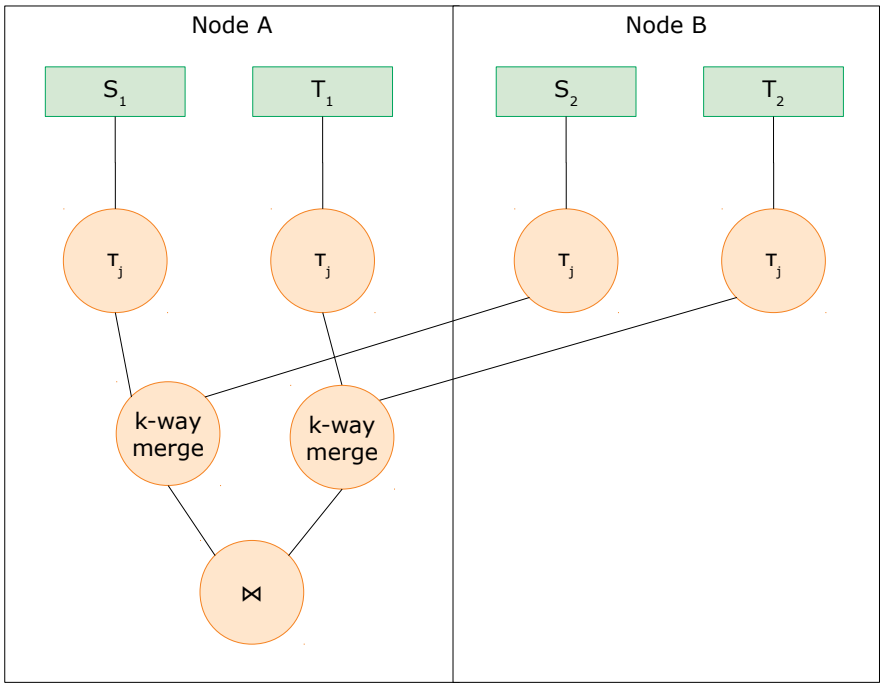


Figure 4.12: Relational algebra of sort-merge join

4.9.3 Merge-join

The input for the merge-join procedure are the two heaps created in the previous step. When called, the procedure will pick the heap with the largest value (in an ascending sort-merge join) and then skip ahead in the other heap until it reaches a value that is greater than or equal to the current value. It will then buffer rows from the heaps as long as the value is equal to the current element. If either of the buffers is empty after this, we empty both buffers, move on to the next value, and run the procedure again. Once both the buffers have values the procedure will return those buffers. Pseudo-code for this procedure can be seen in listing 2.

The sort-merge strategy module will iteratively call this procedure and loop the two output buffers to insert the rows joint into an interim table. And when there are no more matches coming from the merge joiner, it will send off a final `SELECT *` query to return the join result to the client.

4.10 Architectural changes to support the join strategies

To be able to implement the join-strategies in accordance with their designs, the legacy system has to be heavily modified. The strategies rely on functionality beyond

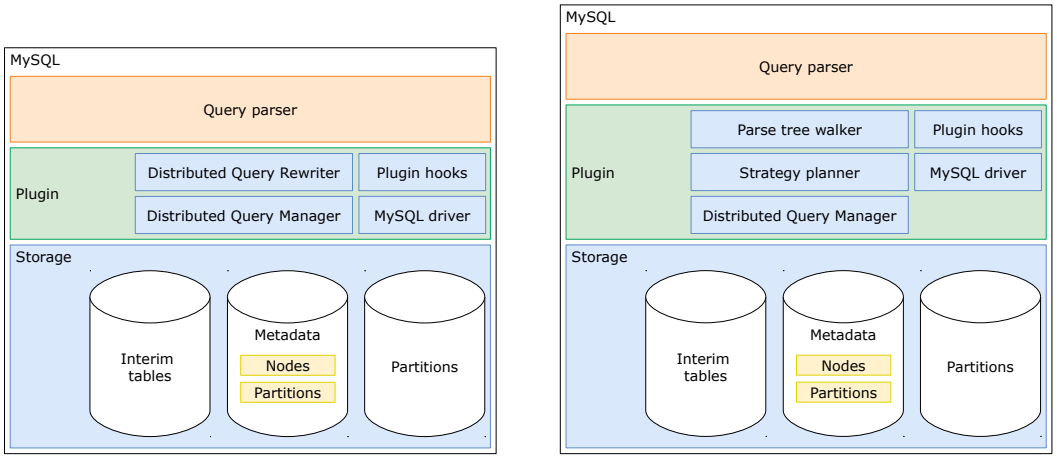
```
lhs_heap = heapify(left_operand_streams)
rhs_heap = heapify(right_operand_streams)

get_next_matches:
    while is_empty(lhs_buffer) or is_empty(rhs_buffer):
        empty_buffers()

        if lhs_heap.peek() >= rhs_heap.peek():
            current_value = lhs_heap.peek()
            while rhs_heap.peek() < current_value:
                rhs_heap.pop()
        else:
            current_value = rhs_heap.peek()
            while lhs_heap.peek() < current_value:
                lhs_heap.pop()

        while lhs_heap.peek() == current_value:
            lhs_buffer.push(lhs_heap.pop())

        while rhs_heap.peek() == current_value:
            rhs_buffer.push(rhs_heap.pop())

    return lhs_buffer, rhs_buffer
```

Listing 2: K-way merge join pseudo code

what was implemented for the specialization project. In this section, we will present some of the larger modifications required.

4.10.1 Modularization

In the legacy system, the DQR module was responsible for both walking the parse-tree to gather data about queries, and making distributed execution plans (generating distributed query sets). As the distributed query plans are going to differ between the strategies it makes sense to separate these responsibilities into distinct modules. Walking the parse tree and gathering data about the tables and queries is now going to be handled by the *parse-tree-walker* module (PTW), and the distributed query plans will be generated in separate modules. These modifications can be seen in figure 4.13.



(a) Original architecture (b) Modified architecture

Figure 4.13: Architectural changes to support multiple strategies

Each of the main strategies will have its own module using the same interface. Namely a function receiving data from the parse-tree-walker as input, and returning a distributed query set containing at minimum a ”final rewritten query” to be executed, and return a result directly to the client. This pluggability will make it simpler to switch between strategies in a running instance of the system. We will also need a strategy entry point, which is just a point in the execution where different strategies will be invoked based on a condition in the system. To simplify, the choice of strategy will be user-specified.

4.10.2 Modifications to the execution model

Sequential stages

Some of our strategies are described as several sequential queries, waiting for the response of a query before sending the next one out. The legacy execution model has no support for this, and currently just executes all the queries of a distributed query set in parallel. This gives rise to the concept of stages in the execution model. We will extend the distributed query model with the ability to define discrete stages of execution, each containing their own set of queries. Stages are to be executed one after the other in the order in which they lie. The queries within a stage are to be executed in parallel as they are not dependent on the completion of each other. For example, in the semi-join strategy, the queries shipping the join-column to external nodes can all be done in parallel. The same applies to the semi-join queries joining the join-column with the remote partitions on the nodes. But, the semi-join queries cannot be executed before shipping of the join-columns has completed. Thus, these two query sets needed to reside in separate stages in the model. An illustration of these stages can be seen in figure 4.14.

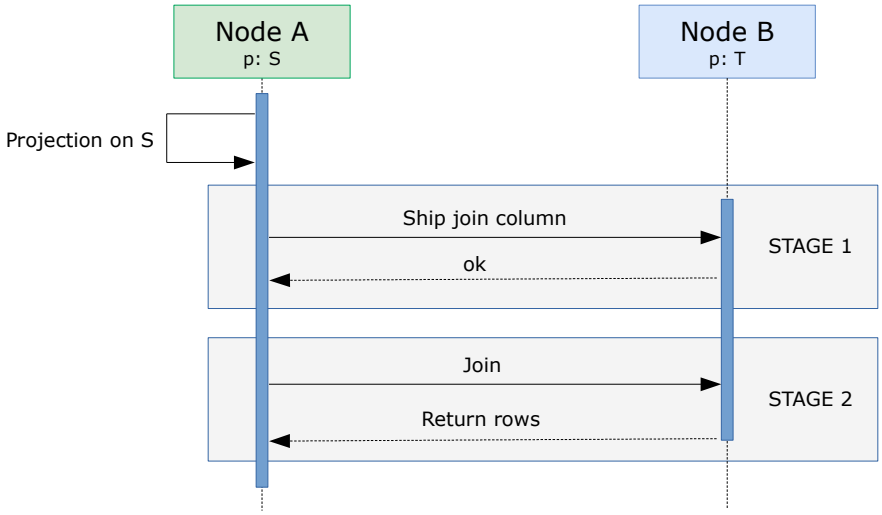


Figure 4.14: Sequential stages of semi-join

Remote interim targets

Another design specification of our strategies requiring modifications of the execution model is the ability to specify remote targets for interim storage of results. E.g. in semi-join, the ability for the join-column projection query to have the relevant remote nodes as destinations, i.e. shipping the join-column, without requiring another set of intermediary queries.

4.11 Hypotheses

Before talking about the implementation of the different methods, we will discuss some predictions about their scalability/performance. In general, we think it is likely that when adding more nodes the strategies will be able to divide up the labor and perform better together, giving near-linear scalability of the system. As each method works differently we suspect they will have strengths and weaknesses depending on the data processed. This can be the results of skew in the dataset, the number of matches, etc. This is important to keep in mind when evaluating each method against one another to better understand and explain the results.

4.11.1 Join selectivity

Join selectivity is a measure of how many of the rows between two tables being joined match, i.e. how many rows are selected by the join. Using join-columns between which there are few matches, we believe semi-join and bloom-join will do well. This because they reduce the amount of data being sent by sending only the

join-column before executing the join, meaning it will be sending a smaller dataset due to there being few matches. Bloom-join uses a compressed representation of the join-column, trading extra processing for a smaller network volume, thus we think that at some size of the dataset bloom will surpass semi-join. Conversely, when using join-columns which generate a lot more matches than operand rows put in, we believe data-to-query or sort-merge will perform best. This because they fetch all the relevant data prior to executing the join.

4.11.2 Value distribution

Using columns with a certain distribution of values can also affect the performance of strategies. We believe using join-columns with a significant number of duplicate values, per table, will boost the performance of semi-join and bloom-join as it means they will ship and process a compressed, smaller join-column set.

On the other hand, we believe duplicates may have a negative impact on hash redistribution. This is due to the probability of a skewed distribution increasing as the number of duplicate values goes up. Duplicates will be hashed to the same nodes, and if several sets of duplicates get assigned the same node, that node might have to do a larger share of the work, causing the strategy to tend towards the data-to-query strategy. Although, for the most part, we believe hash redistribution will perform consistently. This because it shapes the distribution of the dataset and divides up the work, thereby shaping the environment for its execution each time. It doesn't make any assumptions about the properties of the data.

4.11.3 Data distribution skew

Another factor that we believe can have an effect on the strategies is skew in the distribution of partitions themselves. Unevenly dividing data between the nodes we believe can put extra strain on some of the nodes when using hash redistribution and sort-merge, and that these strategies work best when data is evenly spread out. We believe semi- and bloom-join can be served best using an uneven configuration, specifically having a full table on a single node, and the opposing table partitioned. This because they then avoid the recursive step required when having multiple partitioned tables. It simplifies their execution while still retaining the essence of their parallel qualities, the semi-join operation itself.

4.11.4 Slow networks

We believe using a high latency or small bandwidth network between the nodes will make the differences between the network heavy strategies and the less so, much more prominent. Specifically, we think semi-join and bloom-join will be the least affected by the throttling, while data-to-query and hash redistribution will get

a severe performance penalty. This can be tested by using a network with high latency and low bandwidth.

Chapter 5

Implementation

In this chapter, we will discuss the implementation of the different strategies, and how we have evolved the plugin to support them. Each strategy was developed separately, and exist as separate modules in the plugin. Development of the plugin is done in C++. All of the examples in this chapter will be based on variations of this join query:

```
1 SELECT Person.name, Planet.name
2 FROM Person JOIN Planet ON Person.homeworld = Planet.id;
```

5.1 Architectural changes

To implement the architectural changes needed to support the join strategies we had designed, we first split the DQR module in two. The parse-tree walker module was created and we moved all the code concerning parsing and walking the query parse tree to it. We then neatly packaged this information into a data structure to pass on to the strategy modules.

The rest of the DQR, which previously created the distributed query plan was moved to the data-to-query strategy module as this query plan was to be used as a basis for the DTQ join strategy (section 5.2).

To adhere to the design specifications about the pluggability of strategies, and user-specified join strategies, we needed to implement a strategy entry point. To avoid the plugin infinitely invoking itself with every query, the legacy system already had support for an inline comment flag specifying whether or not a query should be ignored: `/*distributed*/ <QUERY GOES HERE >`. We extended this comment flag to include a key-value pair indicating the preferred join strategy. This argument is read by the plugin and based on it the main function of the designated strategy module is called. As some strategies require additional information, the key-value

argument parser is general and able to contain any number of arguments. The chosen data structure is of the form `/*distributed <join_strategy={strategy_id}, key=value,key=value >*/`. All arguments beyond the join-strategy flag are passed to the active join-strategy module.

5.1.1 New execution model

The design specified some changes to the execution model to facilitate some of the more advanced operations of the strategies, e.g. hash redistribution placing data on other nodes and needing to await completion of redistribution before performing a distributed join. To enable this we changed the structure of the distributed query to include remote interim targets and sequential stages.

We then needed to change the DQM to execute this new model. This entailed using Connector/C++ and remote target credentials to connect and insert rows into interim tables. And for the stages, we simply wrapped the entire execution logic of the DQM in a stage iteration.

5.2 Data-to-query

The plugin already supported simple queries of the form: `SELECT projections FROM Table;` using a data-to-query execution strategy. So to support data-to-query joins, we expanded on the existing code.

The first problem we encountered was detecting whether a query contained a join-statement. We did not find any consistent way of telling if something is a join statement using the parse-tree, so we resorted to a simple string search for the keywords involved in the types of joins we support, more specifically ”join”.

The next step was to make the parse-tree walker capable of parsing queries with multiple tables. Previously it had assumed the first table it encountered to be the only one. This made the code simpler for the purpose of supporting single-table query distribution as we could assume every column reference to refer to a column in that table.

We started by making an array of all the tables in the query and ran the partition query generation code for every table. This seemed to work fine, and it returned a set of queries targeting all the existing partitions of both tables. Here is an example of one of the partition queries generated at this stage: `SELECT Person.name, Planet.name FROM Planet;`. Alongside this, we also generated interim table names for every table involved to store the result of the partition queries together. Meaning that the union of the partitions of `Planet` gets stored in one interim table, and `Person` in another.

There was however a problem with the partition queries generated. The previous assumption that any field applies to the table of the partition query was no longer a safe one. We had projections meant for the `Person` table in the partition query

for the `Planet` table, namely `Person.name`. As we quickly learned, this crashes when executed.

To avoid this issue we had to determine which fields belong to which table. The solution we decided to go for was requiring that column names always are referenced using the table name prefix. That makes distinguishing them a trivial pattern matching problem. This, of course, limits the range of queries we support, but because wide query support is not the purpose of this research, this is not an issue. After this, the partition query for `Planet` became simply: `SELECT Planet.name FROM Planet;`.

The next step was to create the final join query between the interim tables. For this, we captured the join-condition from the parse-tree, where it is represented as a where-clause, and replaced the table names with the interim table names. It was then a simple matter of using the information from the parse-step to build a query string of the form:

```
1 SELECT {interim_table1.projections}, {interim_table2.projections}
2 FROM interim_table1
3 JOIN interim_table2
4 ON interim_table1.join_column {=|>|<|...} interim_table2.join_column;
```

The procedure we wrote returned a well-formed query like expected, but when we ran it we received an error from MySQL stating that the columns referenced in the join-condition did not exist. The reason for this then occurred to us. The partition queries had projections that were subsets of the projections from the original join-query. This meant that fields referenced in the join-condition got pruned and were not placed in the interim tables unless they also were part of the projection.

To fix this we needed to modify the parse-step to also take note of all the fields referenced in the join-condition. We grouped these properties by table, like the regular projections, labeling them as ”where-transitive projections”. This allowed us to include them in the generation of the partition queries’ projections, and leave them out of the final query. Here is an example of the partition query for `Planet` with the where-transitive projections added: `SELECT Planet.name, Planet.id FROM Planet;` A nice side effect of adding this was that selection push down for select-queries became as simple as adding one line of code, as the planner now knew which table each selection applied to. At this point, we had a distributed query plan consisting of well-formed queries in accordance with our design.

We then went on to ensure correct execution of the plan. As the data structure used to communicate the distributed query plan to the DQM remained unchanged, there should not have been much of an issue making this work. However, there were some hurdles.

The DQM was, like the original DQR, initially built to perform single table queries, and consequently assumed all queries to access the same interim table. Because of

this, it needed to be extended with support for handling multiple tables to support joins. Instead of retrieving the first interim table name it finds among the incoming queries, it instead reads all interim table names from the incoming queries. These names are further used to build individual queries for both the creation of interim tables and the insertion of result sets from external nodes into these.

5.3 Semi-join

When implementing the semi-join strategy we decided to first implement the non-recursive version, meaning the $n = 1$ step. This is the case where one of the tables exist fully on the master node receiving the query. This is a natural starting point for implementation as it ensures a simple vertical which can be expanded upon.

5.3.1 One partitioned table (n=1)

We started off by implementing the strategy module, whose output was expected to be a distributed query set for the DQM to execute.

The module first needed to do some reconnaissance work to figure out which partitions were located where, and if the $n = 1$ step would apply. We did this by querying the metadata for all the partition information about the tables handed down by the parse-tree-walker. If one of the tables had only one partition locally on the self-node we marked this as the ”stationary” table and sent it along with the partition info about the other table, now dubbed ”the remote table”, to the $n = 1$ planner logic.

The ”stages” feature in the distributed query model added earlier would now come in to play.

The first stage of the strategy we completed was the projection of the stationary table’s join column, where the result of the query was to be placed on the nodes containing partitions of the remote table, including the self-node if need be. We wrote the simple projection query: `SELECT stationary_join_column FROM stationary_table_name` and added the remote targets, together with the intended interim table names, to the distributed query’s interim target list.

We discovered that we were generating duplicates during our semi-join and figured out that the cause was duplicate join-column entries. This was fixed by adding the ”distinct” keyword to the query, resulting in this being the new projection query: `SELECT DISTINCT stationary_join_column FROM stationary_table_name`

We then moved on to the next stage; the semi-join queries. The semi-join queries were written as join-queries between the interim tables at the remote targets and the partition of the remote table residing on that node. It was a simple matter of constructing a join between these, applying the projections and the where-transitive projections concerning the remote table from the original query. We constructed it in this fashion:

```
SELECT {remote_table_projections} {remote_table_where_transitive}
FROM remote_table->name
JOIN stationary_table->interim_name
ON stationary_join_column = remote_join_column
```

We then added an interim target on the self-node to gather all the semi-joined results in an interim table.

Those were the two steps that required sequential execution in the DQM. After that, we wrote the final join-query to be executed locally and return the result to the client. It was generated similarly to the data-to-query one, with the difference being that the stationary table now was referenced as is, and the remote table was replaced by the interim table reference. As opposed to the final join with data-to-query, which is a join between two interim tables.

5.3.2 Recursive distributed queries (n=2)

For the recursive case, i.e. having more than one partitioned table in the join, we went on to generate recursive queries using the query comment flags and arguments implemented earlier.

First, we wrote the logic for deciding which table was to be designated as the stationary table in the next step, making it n=1. We choose to designate the table with the most partitions for this. This made sense to us as this would result in the greatest distribution of work, given a fairly even distribution of that table's partitions. Each of the nodes holding one of these partitions were then to receive a modified version of the original join query, with the "ignore partitions of table"-flag added. The set of distributed queries were generated like this:

```
/*distributed<join_strategy=semi, ignore_table_partitions=Person>*/
SELECT {projections} FROM Person JOIN Planet ON {join_condition}
```

We then had to modify the reconnaissance logic to ignore the partitions of the ignore-partitions table in the argument. When this was in place the new queries were processed as n=1 queries even with partitions of both `Person` and `Planet` present. We still used the execution model of the DQM to execute these queries and gather the results together in a single local interim table.

We did, however, run into an issue with colliding table names. A join result in MySQL can have multiple equal column-names stemming from the different tables in the query. The recursive querying strategy invokes the complete plugin life cycle in multiple steps, this meant that the intermediary results were the user-facing results and that we could not easily mess around with this without breaking the MySQL-conforming results generated by the final query. We solved this by adding

plugin-wide support for aliases, such that the recursive queries could be aliased for the intermediary result.

```
-- alias for intermediary result
SELECT {projections} FROM Person.id AS personId, Planet.id AS planetId JOIN
  ↪  Planet ON {join_condition}
```

5.4 Bloom-join

When implementing the bloom-join strategy, because they are closely related, we made use of as much of the existing semi-join implementation as possible. We went on to identify the bloom-join specific points of the strategy. As described in the design the recursive step of both strategies would be exactly the same, this is natural as the recursive step is purely orchestrational and does not get involved with the actual semi-join logic. The reconnaissance logic of determining the level of recursion and which tables to be labeled stationary and remote would also be shared between the two strategies. This meant we only needed to implement a bloom-join replacement for the actual planning and execution steps of the semi-join itself, the core of the strategy. In the design stage, we determined that we would need to have remote procedures callable by a master of the query. The master is analogous to the semi-join strategy module, and where semi-join relied on the general execution model of the DQM to execute its queries, the bloom-join would require the implementation of a slave procedure on the target nodes to help it complete the requests.

5.4.1 Bloom-filter library

For the bloom filter data structure and algorithms we used an open source C++ library called ”bloom”[1]. The library offers a bloom filter class and a data structure for setting the parameters of the filter. Some of the parameters that can be set are: expected element count, false-positive-probability, the maximum number of hashes, etc. These parameters can impact the size and accuracy of the filter. The library also offers a function `compute_optimal_parameters()` that helps with constructing the optimal parameters based on only false-positive-probability and expected element count. In our implementation, we relied on this function to set the remaining parameters.

[1] https://github.com/ArashPartow/bloom

5.4.2 Shipping the filter

To be able to send the filter from one node to another we needed to be able to serialize it. The filter itself was an object containing a bit-table, some hash-salts and parameters about its size and number of inserted elements. Through some experiments, we were able to recreate a read-only duplicate of a filter by only using the bit-table, and the number of inserted elements. This meant we only needed to serialize the bit-table and send the number of inserted elements as an argument to ship an applicable filter. As stated in the design we wanted to use comments as the method of communicating the filter to other nodes. In other words: strings. With this in mind, we landed on Base64 as the encoding scheme for the bit-table. Base64 is a common compact binary-as-text format using 64 ASCII characters as its index. The MySQL source tree also happens to contain an implementation of Base64 encoding. Using this module we were able to encode the filter into ASCII and append it as an argument in the distributed-query comment.

5.4.3 Bloom-join master

We implemented the bloom-join master strategy as two stages. Structurally it is very similar to the implementation of the non-recursive case in the semi-join strategy, and it shares the reconnaissance logic for the selection of the stationary and remote tables. One important difference is that the first stage of the bloom-join strategy does not fit into the general distributed execution model of the DQM. Therefore, we implemented the first stage as a custom procedure called the bloom-join executor. This procedure is responsible for the functionality of the strategy not available in SQL, namely the generation of the bloom filter. This procedure has a complementary component, the slave procedure, responsible for applying the filter on the receiving end of a bloom filter query. We will describe the implementation of the slave in section 5.4.4.

We set up the entry point to the bloom-join executor by having it take a table name and a join column. We then wrote the setup logic for the bloom filter. We made queries to read the join column, as well as the number of rows.

We then made the bloom filter by using the class from the library and gave it the appropriate parameters, such as how many elements it expects to insert (the row count). The filter library could then optimize for the number of hashes to use, as well as the size of its internal bit-table. Then, with a prepared filter, we iterated over the join column, inserting the elements. After the filter was complete the executor calls upon the Base64 encoder of MySQL to return a Base64-serialized version of the bit-table for the strategy module to use.

We could then move on to the next stage of the bloom-join strategy module, stage 2. We named the second stage the ”bloom filter query”, as all of stage 2 could be achieved using one query.

To generate this query, we started by making interim table names for the target

nodes to place their filtered results in. This interim name was encoded into the comments of the query along with the bloom filter to be applied. The query was generated like this:

```
/*distributed<join_strategy=bloom,
table_name={remote_table_name},
filtered_interim_name={interim_name},
bloom_filter={BASE64_FILTER}>*/
SELECT {projections} FROM {interim_name};
```

Giving the interim name as an argument serves as a command to the bloom slave about where to place its filtered data as to fulfill the query after the comment. We then placed these interim targets in the distributed execution model in the same way as the semi-join strategy, as everything after this point is the same in both strategies.

5.4.4 Bloom-join slave procedure

On the receiving end of the ”bloom filter query”, we needed a custom procedure to apply the filter before responding to the query. We implemented the slave procedure very straightforward as its instructions are given by the arguments generated by the master. We decoded the filter using the Base64 module. Queried the specified table, with a simple `SELECT {projections} FROM {remote_table_name}`. The rows were then passed through the filter’s membership test returning either true or false. The decision of the filter was then used to either insert the row into the specified interim target table on the master or to discard it. When all the rows have been tested, the rows that passed the filter have all been placed on the master’s interim table. So, after this the bloom slave returns a *NO-OP* query, signaling to MySQL that it can continue with the execution, but with a query that does nothing, and returns an empty result to the master signaling that the stage is complete.

5.5 Hash redistribution

The hash redistribution method is made up of two main parts. There is the master that resolves which nodes to include, plans the redistribution and initiates the join, and there are the slaves which actually executes the redistribution and joins.

5.5.1 Master

To start off, the master was implemented. It builds two stages for the strategy: one for the redistribution and one for the join.

Firstly it must figure out which nodes to include. To do this it analyses the tables queried and resolves which partitions exist of the table, and where they are located. This is done through a lookup in the metadata. From this, it builds individual query comments for each node relevant for the query. These comments contain, like in Bloom-join, a slave flag used for plugin-to-plugin communication to trigger the slave, e.g. `/*distributed <join_strategy=hash_redist,hash_redist_slave=true >*/`. Further on, for each slave to do their job they need to know which local table to query, in which interim table to place the result, and which column is to be joined on. This is additional information placed into the query comment in the format:

```
tables=[
    table_1_name:interim_1_name:table_1_join_column,
    table_2_name:interim_2_name:table_2_join_column
]
```

An example of a complete query comment:

```
/*distributed<
    join_strategy=hash_redist,
    hash_redist_slave=true,
    tables=[Person:af9d3:homeworld,Planet:37f4a:id]
>*/
```

It is worth noting that what makes these query comments individual for each node, is that the `tables` parameter will be populated by tables local on that node. Thus, no node will try to query tables not present on itself.

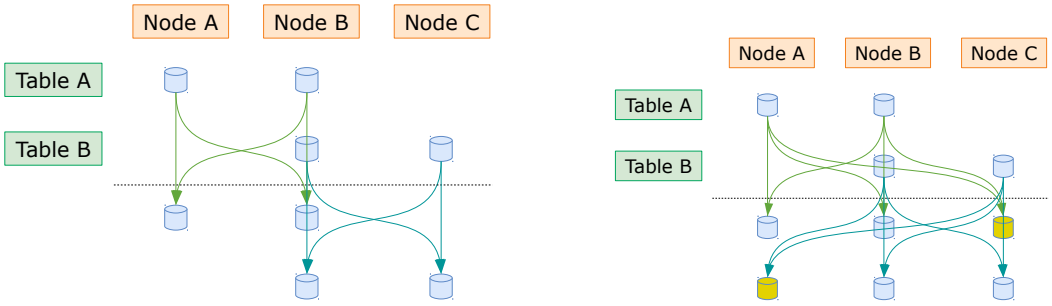
Initially, the query performed with the query comments was a no-op `SELECT 1 WHERE false;`. This was because we thought there was no reason to perform any query, as this step was simply plugin-to-plugin communication to move data around. However, this was changed to passing the actual user query instead. The reason for this is explained in section 5.5.2.

The final join stage is quite simple. Since the interim tables holding the data on each node corresponds, the same join statement can be used on all nodes. The generation of the join query is done in the same way as in both semi-join and bloom-join. The result of the different join queries are placed into an interim table on the master, before rewriting the original user query to a simple `SELECT * FROM interim_table;`. The reason for the simple select-statement is that the returned result already is joined, thus, it needs only be returned to the user.

5.5.2 Slave

The slave starts by parsing the query comment. This involves generating projection strings for each table and query the metadata for nodes to involve. It is the latter which could cause problems as it could potentially lead to an incomplete result, and the reason we had to switch from a query no-op to the original user query.

To understand the cause of the problem we will look at what is passed to the slave. As mentioned earlier the slaves receive a `tables` list in the query comment, containing all the partitions that given node itself holds. We simply queried the metadata for all nodes containing that partition and hashed the data to those nodes, which can be seen in figure 5.1a. The problem with this approach is that there could, and probably would exist node(s) containing only a partition of one of the tables, thus, would not receive data coming from tables it did not previously hold a partition of. When entering the join stage none of these nodes would have any other tables to match results towards, hence the incomplete result to the user. The problem boils down to the set of nodes containing a partition of A is not being the same as for B. The solution for this problem is to ensure that the nodes the data is hashed to are the set of all nodes containing a partition of one of the tables in the query, as seen in figure 5.1b. There are two ways of rectifying this problem: either having the master generate the set and pass it as an additional parameter in the query comment, or passing the original query and leaving it up to each node to generate the set. We chose the latter. This also has the added benefit of simplifying retrieval of projections as well, which comes in handy when generating the hash queries later on.



(a) Node set not equal (b) Node set equal

Figure 5.1: Problems with not including all nodes containing a partition of A, B or both

After the parsing step the slave starts to generate individual query strings for all nodes in the node set for each partition the slave contains. These query strings are of the form:

```
1 SELECT projections
2 FROM table_name
3 WHERE conv(substr(MD5(join_column), 1, 6), 16, 10) % size_of_node_set = node_i;
```

The WHERE clause is the interesting part of the queries generated, as this is where the hash redistribution is actually performed. Initially we used `WHERE MD5(join_column) % size of node set = node_i` to determine the receiving node. This did, however, skew the distribution of data among the nodes quite heavily, with nearly half of the data being placed on one node, and the remainder more evenly divided among the rest. The cause of this problem is rooted in the output of MD5 and the operation done on the output. MD5 outputs a hex value represented as a string. When performing a modulo on this output it is evaluated by its ASCII values, and not as hex. As it only contains the characters 0-9 and A-F only a small subset of all ASCII characters is ever used. These two character ranges are not continuous either. Thus, it will leave large gaps of unused values. The solution to this was to take a substring of the MD5 hash, parse it as a hex-value, before performing the modulo operation. Doing this ensures the use of the whole value range in regards to the modulo function.

5.5.3 Hash function

As mentioned in the design, what we are interested in are fast hash algorithms, not algorithms designed for security. MySQL itself supports multiple hash algorithms[2]

[2] https://dev.mysql.com/doc/refman/8.0/en/encryption-functions.html

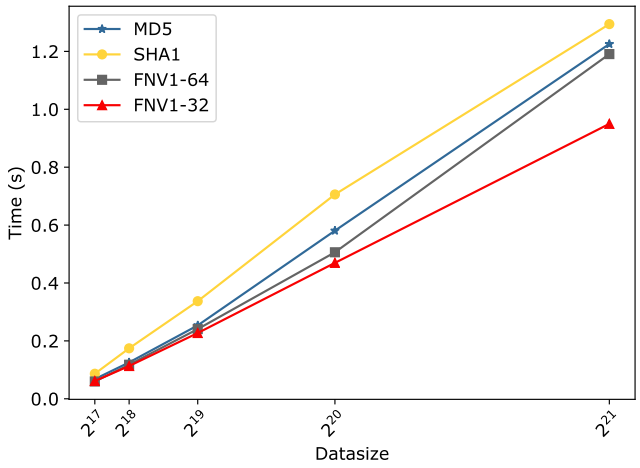


Figure 5.2: Time usage for different hash algorithms

that fit our requirements and are easy to use since they can be used directly in SQL queries. An alternative to using these would be to either implement some hash function directly in the query, such as `WHERE CUSTOM_HASH_FUNCTION(join)%size_of_node_set`, or using *User Defined Functions* (UDF)[3], which would provide the same interface as the built-in hash functions.

To verify that the hash redistribution strategy would not be negatively impacted by a slow hash function, we did some testing. The test involved the two natively supported functions, SHA1 and MD5, and a third one. The third one was chosen on the basis of being considered fast, namely the FNV hash function [27].

We used an existing wrapper for FNV to a UDF[4]. The wrapper uses the C-implementation[5] of FNV by one of the original authors, Landon Curt Noll. The algorithms were tested by running each function against a table of varying size and looking at the time usage. Each hash function was run five times, with the average time as the result, and with an initial warm-up, for a total of six runs. This was performed for each size tested. The query performed was `SELECT rhs_normal, HASH_FUNCTION(rhs_normal) AS hash FROM rhs;`. The results can be seen in figure 5.2. It is worth noting that the different algorithms create different size hashes as can be seen in table 5.1.

[3] https://dev.mysql.com/doc/refman/8.0/en/adding-functions.html
[4] https://github.com/mjradwin/fnv-mysql-udf
[5] http://www.isthe.com/chongo/tech/comp/fnv/index.html

Algorithm	Hash size (bits)
SHA-1	160
MD5	128
FNV1-64	64
FNV1-32	32

Table 5.1: Output size for the different hash algorithms

As expected the hash functions perform nearly linearly. This is because there is no additional cost to hashing e.g. two attributes over just one, other than the time it takes to hash another attribute. An interesting thing, however, is that FNV only has a slight improvement over SHA1 and MD5, which indicates that the built-in methods in MySQL would do fine. The choice of hash method, therefore, falls on MD5, as it is available in MySQL out-of-the-box, slightly faster than SHA1, and that the difference in performance over FNV is small.

5.6 Sort-merge join

For the sort-merge strategy, we built a custom join algorithm and made a separate module for all of its execution. This made sense as it has very little in common with the other strategies after the parse-tree walker has done its reconnaissance.

To begin with, we generated the `ORDER BY` queries for each table read by the parse tree walker. The queries were generated like this: `SELECT {projections} FROM table_name ORDER BY {join_column} ASC` where the join column is the property to be ordered by. We choose ascending order for the queries as primary keys are often generated by incrementing an integer, making an ascending order of IDs. We then iterated over known partitions and sent off the queries to the nodes that held them. We then placed pointers to the result streams in arrays for the heap merge to consume. None of the results were read yet at this stage of the implementation, meaning nothing waited for results to be transferred back yet.

5.6.1 K-way-merging

We implemented a k-way-merger using a heap in accordance with our design. Since we were operating on Connector/C++ result stream iterators and row objects we could not use the standard library heap implementation, as the templating in this is not quite flexible enough to enable join column access and iterator style heap generation. To have these properties for our heap, we implemented our own binary heap construct. Our heap was implemented using an array representing a binary tree, with simple arithmetic addressing to retrieve left-child, right-child, and parent. The heap constructor takes an array of result streams as input and moves them around by using an iterative sift down heap construction method. At this point, only

the first row of the streams was read. The next read occurs when taking a row out of the root of the heap, causing the root stream to fetch the next row. The root node is then sent into the sift-down procedure to correct the heap. This is the extent of functionality we implemented for the heap, as it was all we needed.

5.6.2 Merge joiner

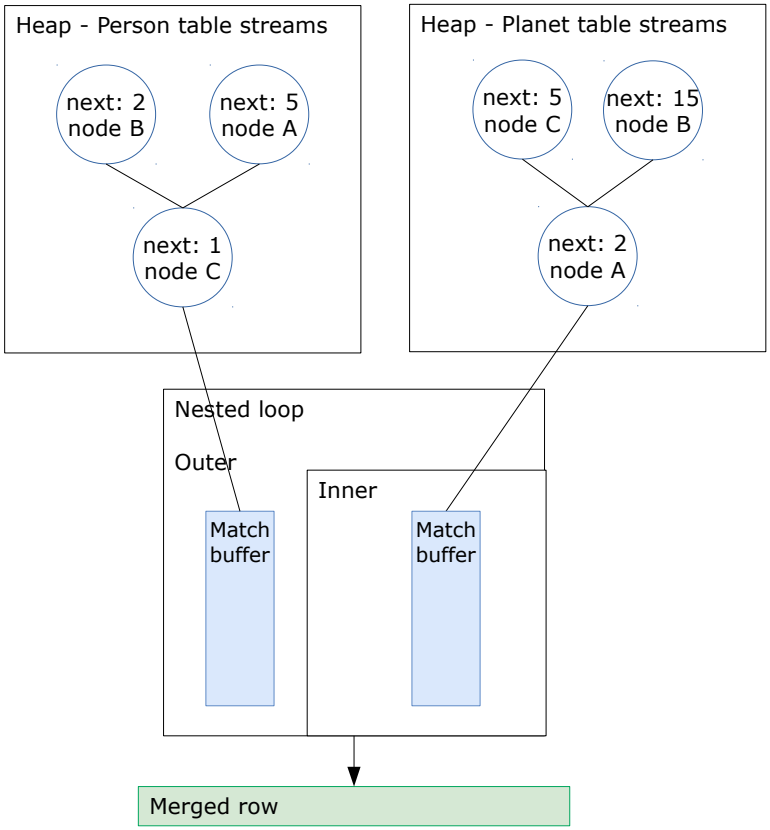


Figure 5.3: Merge joiner diagram

The merge join operation itself was implemented as a class keeping two heaps and two buffers, similar to the pseudo code for it in the design chapter. A diagram of the implementation can be seen in figure 5.3. The current value was assigned by taking the top of the heap with the largest value, as the smaller of the two would not match anything with ascending ordered streams. All the rows, from both heaps, with join columns matching the current value were then buffered. This procedure was then repeated until both buffers had values, meaning some matches were found. This discovered set of matches were then returned to the caller, pausing the merging process until the caller asks for more matches. The caller, which is the strategy

module then proceeded to insert these matches into an interim table by looping both buffers and firing off insert statements containing the combined rows.

To keep a low memory footprint and allow for pipelining, we used iterators all the way through the merging processes. The streams coming into the heap are iterators, so we made the heap like an iterator. The heaps coming into the merge-joining algorithm were then iterators, so we made this process iterative as well, only having to keep currently matching rows in memory. This way we could contiguously generate matches and place them in an interim table, without running out of memory with large result sets. This is illustrated in figure 5.4.

We keep the chain of iterators down to ”Connector/C++” and made the strategy reliant on its buffering mechanisms. The benefit of this is that it keeps the implementation small as well as more robust. The mechanisms of ”Connector/C++” are probably more reliant than anything we would have time to create for this project, which gives the implementation a better chance to keep up with more rigorous testing involving large amounts of data.

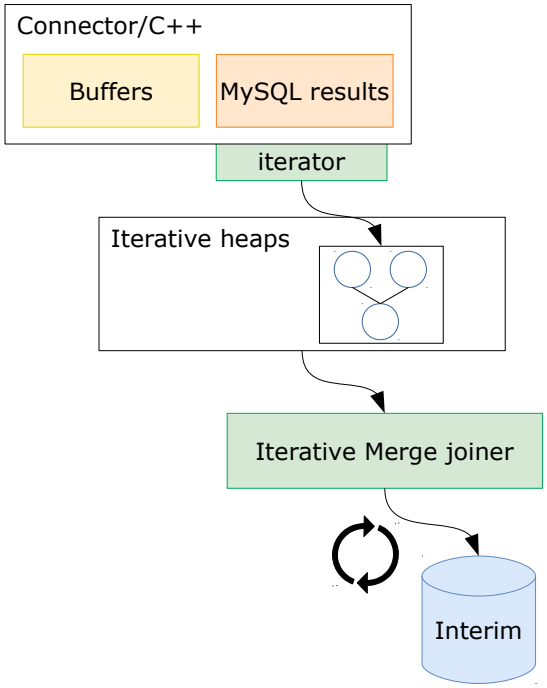


Figure 5.4: Iterative data flow of sort-merge

5.7 Optimization

In preparation for the evaluation of our system, we did some preliminary testing using bigger datasets than before. We tested the robustness of our implementation, looked for anomalies, and validated the cardinality of the join-results returned from all strategies. During this period we uncovered a number of bugs relating to memory-usage, blocking calls, lack of indices for interim tables, and more.

The most significant improvements were; indices for all interim tables involved in joins, switching to Connector/C++ CRUD API for generating insert statements, as well as implementing batch inserts, removing the blocking `fetchAll()` call when handling interim results, replacing it with an iterative generation of batches of `BATCH_SIZE`.

Batch insertion allowed the system to handle much bigger datasets. Previously it attempted to insert all the rows it handled with one insert statement, no matter the size of the dataset, and predictably this led to very high memory pressure and ultimately failure. With the fix of reading and inserting rows in batches, the system does not run out of memory, as only `BATCH_SIZE` number of rows is kept in memory at the same time.

Adding indices to the interim tables before using them in joins was a game changer in terms of performance. With a dataset consisting of two tables, with 1000 rows each, we observed a decline in query duration of up to 46%. The results from this sample can be seen in figure 5.5. As can be seen from the figure, the sort-merge strategy was unaffected by the change, this is because sort-merge does not use interim tables for its joining operation, but rather its own join algorithm, and therefore does not need nor use indices. Using other larger datasets the improvement was even bigger, and it became clear that indices are crucial to join performance in MySQL.

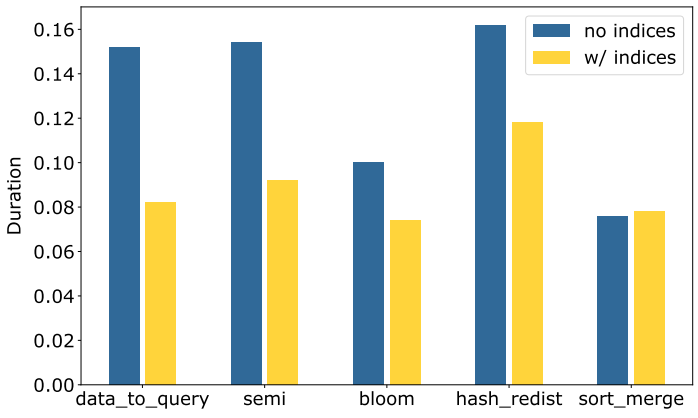


Figure 5.5: Indices improvement with 2 x 1000 rows, average of 5 runs

5.7.1 Parallelization of interim table insertion

Initially, insertions of results into interim tables were done sequentially. After each thread has performed their task, result-sets were returned to the main thread and inserted from there. This was unnecessary because all the infrastructure of multi-threading for the execution of queries was already in place, and MySQL supports concurrent inserts. So instead of the threads just performing the queries, they were extended to also insert the query results into interim tables.

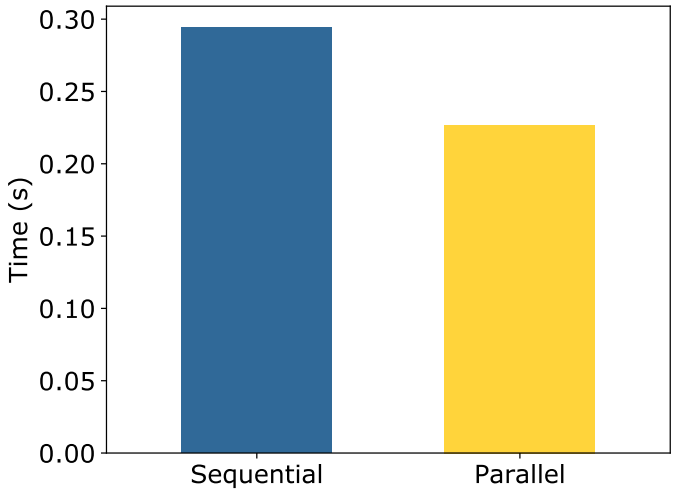


Figure 5.6: Parallel insertion speedup with 2 x 2^{17} rows, average of 5 runs, data-to-query strategy

The effect of this redesign can be seen in figure 5.6. It shows a decrease of ~23% in time usage. This is a notable decrease and confirms the redesign was worth the effort.

Chapter 6

Evaluation

In this chapter, we will evaluate and compare the different strategies we have implemented. The goal of the evaluation is to investigate the strengths and weaknesses the strategies have in different scenarios, and create a basis for a discussion around the viability of the different strategies in a plugin architecture. We will present the test suite created for the project, the datasets used for testing, our results and a comprehensive discussion around them.

To test the hypotheses from section 4.11, we will put the strategies through a number of trials. We will test

- Horizontal scalability of the strategies by adding nodes to the system.
- Vertical scalability by increasing the size of the datasets used.
- One partitioned table vs two partitioned tables
- Datasets with different characteristics to see the effects they have on the strategies.
- Network effects by using a slower network, i.e. adding latency and lowering bandwidth

6.1 Measuring performance

Our main measure of performance will be query completion time, that is to say, the duration of time it takes to perform a query and return the result. More specifically this window of time is defined by two points, illustrated in figure 6.1, where t_1 is the moment right before a query is sent to MySQL, and t_2 is the moment the client starts to receive a result from that query. This is so that we only measure the execution time of the query and not the time it takes to ship the entire result to the client. This way we can increase the data size without polluting our results with uninteresting and increasing shipping times of the end result.

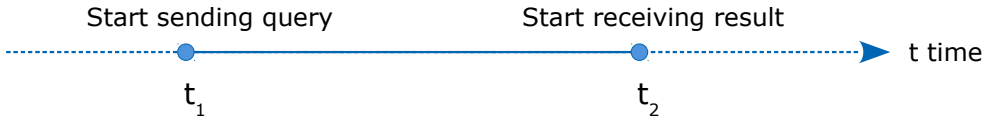


Figure 6.1: Measuring points timeline

All queries will be sent from a client program local to the node receiving the query, so any network delays for sending queries or receiving the start of a result are negligible.

As MySQL has some clever tricks concerning memory usage, such as keeping recently accessed tables in buffer pools in memory[1], every query that we measure is preceded by a warmup-query whose time measurement we discard. We will then run several instances of that query measuring their completion time now that relevant data likely resides in the buffer pools of the nodes. This means that all instances of that query, run after the warmup-query, will run under similar conditions, and produce more consistent results.

To produce our results each query will be run 1+5 times, where the representative duration of a query will be the mean of the duration of the 5 last runs.

6.2 Dataset

The join columns are the pivot points to most of our hypotheses. To ensure coverage of all the scenarios described we needed to have columns with the described properties. Such as selectivity (how many matching rows between two columns) and different distributions of values. We also needed the dataset to be size-adjustable and partitionable. This, so we could generate a variable number of rows and distribute them to the nodes while still retaining the distinct properties of the columns.

With these requirements, as well as a wish to have control over the granularity of the data, we found that traditional database testing tools, such as TPC-H[2] did not quite fit the bill. Therefore, we decided to generate our own dataset using Numpy[3]. Numpy is a numerics library for Python allowing generally well suited for generating sequences of numerical data, as well as matrices and tabular data. We made use of the statistics functionality provided by Numpy to make sure our dataset had the required properties, at every size and partitioning.

The entire dataset consists of two tables, the *left hand side* (lhs) table and the *right hand side* (rhs) table. These represent the operands of the join, one on each side. Columns in each of these tables will be used to provide properties of the individual operand inputs as well as the combined properties of the joined result. For most of the tests, both tables will be split into equal size partitions, one lhs and

[1] https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
[2] http://www.tpc.org/tpch/
[3] http://www.numpy.org/

rhs partition for each node. E.g. if lhs and rhs contain a million rows each, and we are testing with 4 nodes, each node gets an lhs partition containing a quarter million rows and the same for rhs. We will also test having only one partitioned table, in that case, rhs will reside in full on one node.

6.2.1 Selectivity of join

To test how the selectivity of the joins would affect the strategies we generated multiple columns containing a variable number of matches between the tables. We generated selectivity columns going from 10% matching rows to 100%, in steps of 10%. An illustration of the different selectivity levels with a small sample of 10 rows can be seen in figure 6.2.

We did this simply by generating two sequences of integers, one column for each lhs and rhs, with an offset between them determining the selectivity level. These columns have no duplicates within themselves. After generating the sequences we shuffled the order so that the partitions do not have any unwanted or unexpected properties, e.g. sorted order.

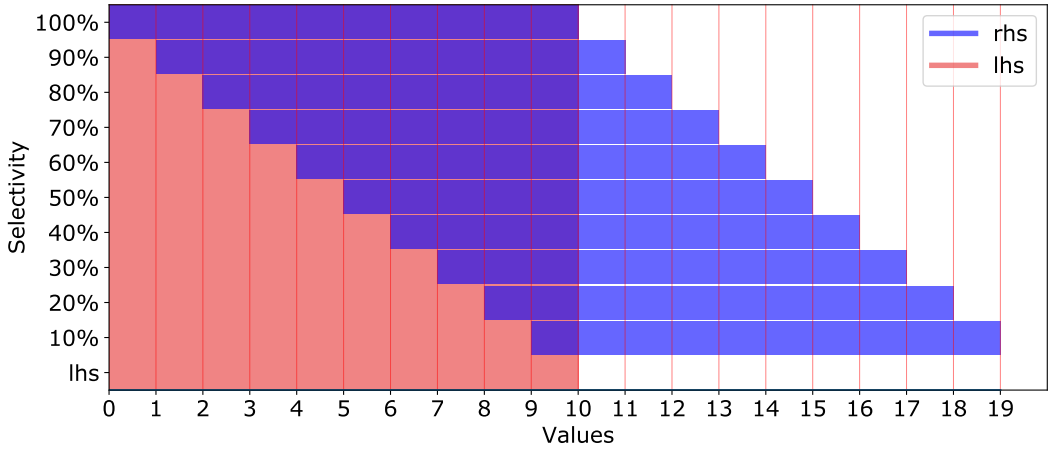


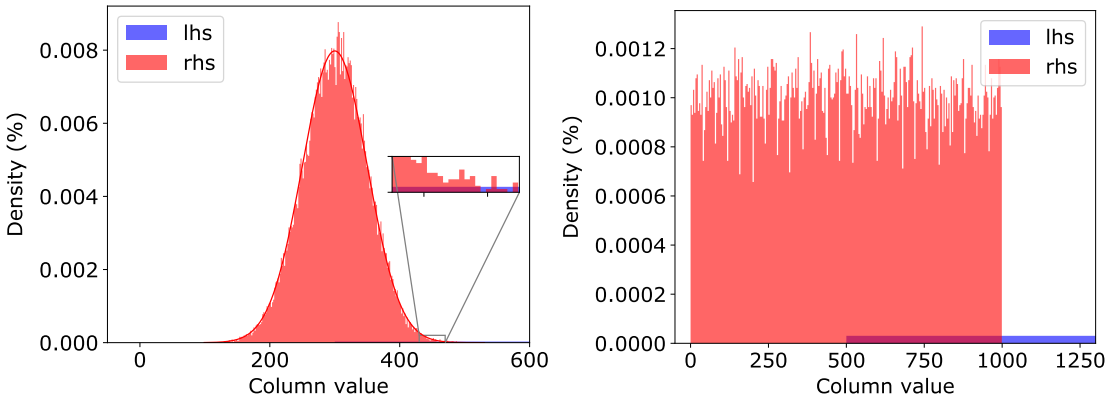
Figure 6.2: Selectivity columns plot with 10 rows

6.2.2 Distribution of values

To test how the distribution of values within a column would affect the performance of the different strategies we generated two different distributions. The motivation behind this being the predictions we made about semi- and bloom-join performing better in the presence of multiple duplicate values, as these need to send less information in order to filter the other table. We also predicted that hash redistribution might suffer from this due to the increased probability of skewed redistribution when there are duplicates.

The two distributions we generated were: Normal distribution, which has a high density of values near the mean, decreasing as it moves away on either side, see figure 6.3a. Uniform distribution, which has duplicate values uniformly distributed across a range, see figure 6.3b.

The columns with the distributions are put only in the rhs table, while the lhs table column remains a sequence of integers. This was so we could control the number of matches generated between them. We decided to go with 50% overlap of the values, as this is the median of our selectivity datasets. Important to note is that rhs is the outer relation in all our test queries. This is significant for semi- and bloom-join as it ships the join column of the outer relation when all else is equal. We do not believe it to have any significant impact on the other strategies.



(a) Normal distribution

(b) Uniform distribution

Figure 6.3: Sample of 2^{15} rows, 50% probability of match

6.3 Test setup

To test the methods we have had access to 16 *virtual machines* (VM) in Oracle Cloud dedicated to this project. The VMs were provisioned with the same template, VM.Standard1.2[4], resulting in a homogeneous system.

Shape	OCPU[5]	CPU Model	Memory (GB)	Network Bandwidth
VM.Standard2.1	1	Intel Xeon 8167M	15	1Gbps

Table 6.1: Specification of the VMs

We then optimized our build of MySQL and the plugin by enabling compiler optimization and setting the limits for buffer-pool size and heap-size for in-memory

[4] https://docs.cloud.oracle.com/iaas/Content/Compute/References/computeshapes.htm

[5] An *Oracle Compute Unit* (https://cloud.oracle.com/compute) is a physical CPU core, including the hyperthreading unit. It is linked to that VM only, and not shared with others.

tables to the maximum allowed values. This was to reduce the probability of the system hitting any of those limits when we scale up the dataset.

6.3.1 Automation of tests

To make testing on 16-machines practical we automated as much of the test process as possible. We used the cloud automation tool Ansible[6] to set up the test environment and run tests across the nodes. This included: installing MySQL with our plugin as well as all dependencies, setting up user-credentials, loading datasets, partitioning data, running tests, as well as a clean teardown of the nodes after the tests.

The tests are broken into steps of number of nodes and size of the dataset. When increasing the size of the dataset, the old data is deleted from the nodes, and the new dataset is uploaded to the node, partitioned according to the number of nodes, and inserted into the MySQL instances. When increasing the number of nodes the MySQL servers are restarted. The result of this was a reproducible test suite that ran autonomously.

6.4 Results

In the following sections, we will present our results using multiple graphical plots. The y-axis for most plots is a measure of performance denoted by either ”Time spent” or ”Throughput”. Time spent is the simple measure of query execution time, while throughput is the measure of rows processed per second. ”Rows” in this case is effectively all rows that have been scanned in the execution of the query. The results are divided into sections for VMs on a local network, simulated ”slow” network, and ”physical machines”.

6.5 Virtual machines on a local network

This section presents the results of the strategies running on an uninhibited local network between the VMs in a data center. To measure the stability of our results we computed the mean and the standard deviation of the series of 5 runs for each unique query and dataset. We did this to gauge the level of confidence in our results in order to be able to make sound conclusions about them.

[6] https://www.ansible.com/

Strategy	Standard deviation (%)
data-to-query	0.77
semi	0.57
bloom	0.63
hash redistribution	1.01
sort-merge	0.59

Table 6.2: Average percentage standard deviation for each strategy in figure 6.4

Table 6.2 shows the average standard deviation of the data points for each strategy in the first plot (figure 6.4). The average is, for the most part, less than 1 percent, meaning the results are representative of the actual performance and allow for valid comparisons of the strategies. The standard deviation for all our results lie around and below these averages.

6.5.1 Horizontal scalability

We have tested the scalability of the methods with regards to various node sizes. The result of variable nodes for 60% selectivity, figure 6.4, and 10% selectivity figure 6.5, show how the performance of the different strategies evolves when adding more nodes.

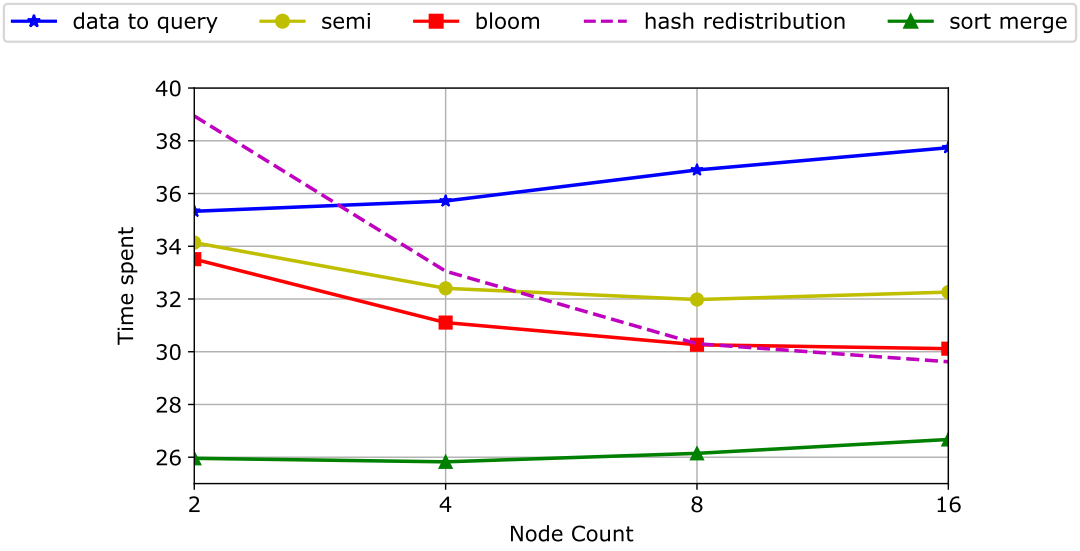


Figure 6.4: Scaling number of nodes with the 60% selectivity column, with 2^{19} rows

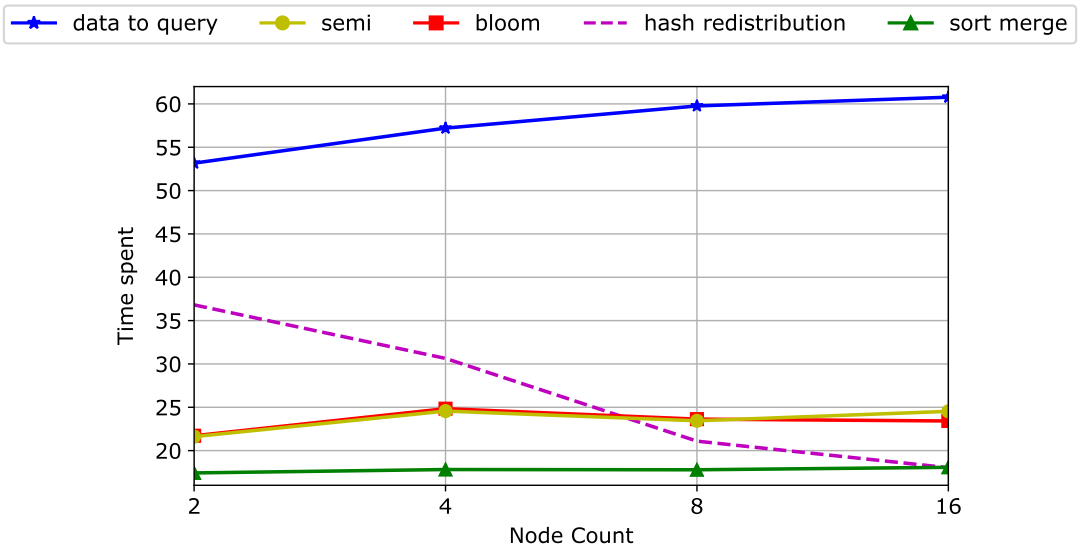


Figure 6.5: Scaling number of nodes with the 10% selectivity column, with 2^{20} rows

In the results, we can see that data-to-query is the slowest strategy. And even though the data-to-query master receives the same amount of data regardless, it displays a performance decrease when adding nodes. This is due to the extra overhead of querying and handling incoming results from an increasing number of nodes.

For bloom and semi, the scalability plot is flat. Adding nodes neither increases nor decreases the time used for each query. Having no performance gain when adding nodes to a parallel strategy indicates that the sequential part of the execution grows inversely to the performance gained by parallelizing the work. This might be due to the added overhead of managing more nodes, the relative complexity of the strategies (e.g. number of messages, sequential steps, etc), or a bottleneck in the implementation. We will look into some of the limitations of the architecture in section 6.5.4.

Sort-merge also does not display any performance boost when adding nodes. This is most likely due to the fact that the parallel part of the strategy, the sorting, is relatively small compared to the sequential merging when using the dataset sizes we have tested.

For hash redistribution, the story is another. With a sufficiently high number of rows and participating nodes hash redistribution begins to overtake every strategy. This is because the overhead of redistributing the data gets relatively smaller to the payoff of doing most of the work in parallel. Important to note here is that the redistribution work itself is also done in parallel. So when adding more nodes the number of messages goes up, but the volume of data handled by each node goes down. Hash redistribution is overall the most parallel strategy we have implemented,

and the benefit of this becomes clear as we scale.

1 partitioned table

To investigate our hypotheses regarding the strategies' ability to handle uneven partition distributions, i.e. one node having a larger share of the data, we tested with only partitioning and distributing one table, the lhs, leaving the rhs table to reside solely on the node receiving the query.

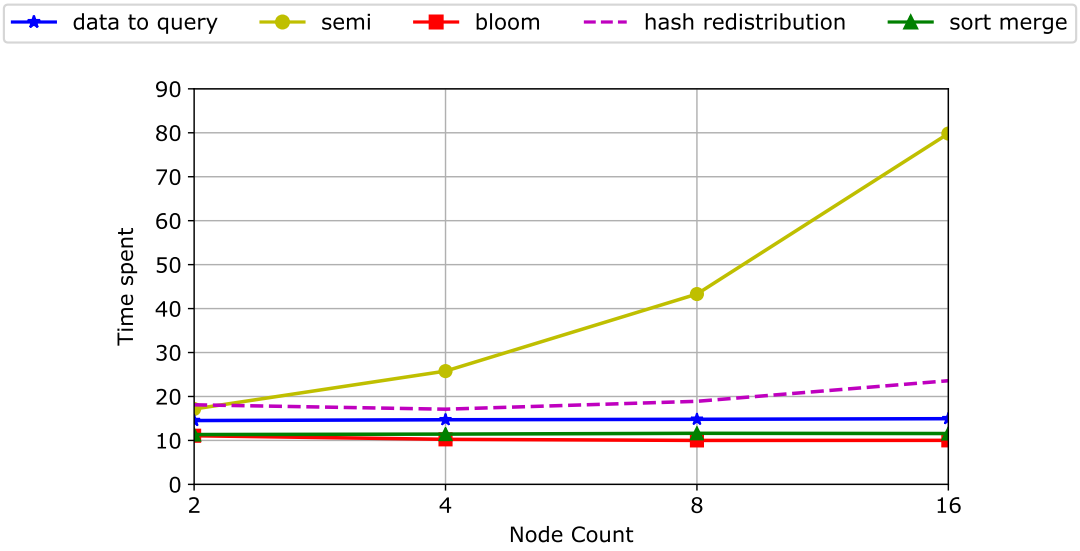


Figure 6.6: One partitioned table, scaling number of nodes with the 50% selectivity column, with 2^{19} rows

The results of this test can be seen in figure 6.6.

In this test case hash redistribution has lost its scalability in comparison to what it showed in figures 6.4 and 6.5. This is because of the skewed distribution of data, making one node responsible for redistributing 50% of the total amount, instead of many nodes performing smaller redistributions like in earlier scenarios.

The workload of the redistribution is skewed to such a degree that the benefits of executing the join in parallel get overshadowed. This is in line with our hypothesis for hash redistribution in this scenario.

The elephant in the room is the divergence between bloom and semi, which otherwise performs similarly given partitioning of all tables. Like with hash redistribution semi suffers when having to deal with one large table. Its initial step is to distribute the join column to all nodes, which in this case is a large dataset. This is the only place where bloom and semi differ. Bloom's filter representation of the join column is far more efficient to transfer than a large column. One would think the culprit of this effect is network, but in these tests, this looks to be caused by the method

of which rows are received and inserted onto nodes in the plugin architecture. We discuss this further in section 6.5.4.

Bloom, however, performs the best of all strategies in this test. This is because, as discussed in our hypotheses, bloom and semi are well suited to this configuration. Their main mode of operation is sending out a representation of the join column to the other nodes so they can perform a halfway join. So this configuration saves them from executing the recursive step ensuring all the nodes' join columns are considered. Bloom capitalizes well on this, and semi would too if not for the architectural limits it encountered in this test.

Sort-merge is still parallel in sorting the partitioned table, but now has become single-node bound in sorting the unpartitioned table. Having this sequential sort stage on one node limits the benefits of parallelism to the time it takes to perform the sequential step. Meaning that no matter how many nodes are working in parallel on the partitioned table, the execution time will converge to the time of the sequential step. This limit is known as Amdahl's law [28]. In our hypotheses, we predicted that this would negatively affect sort-merge, and it has lost its lead to bloom-join, but there is not any big loss in performance. This might be due to the dataset not being sufficiently large to make the sorting operation the bottleneck.

6.5.2 Vertical scalability

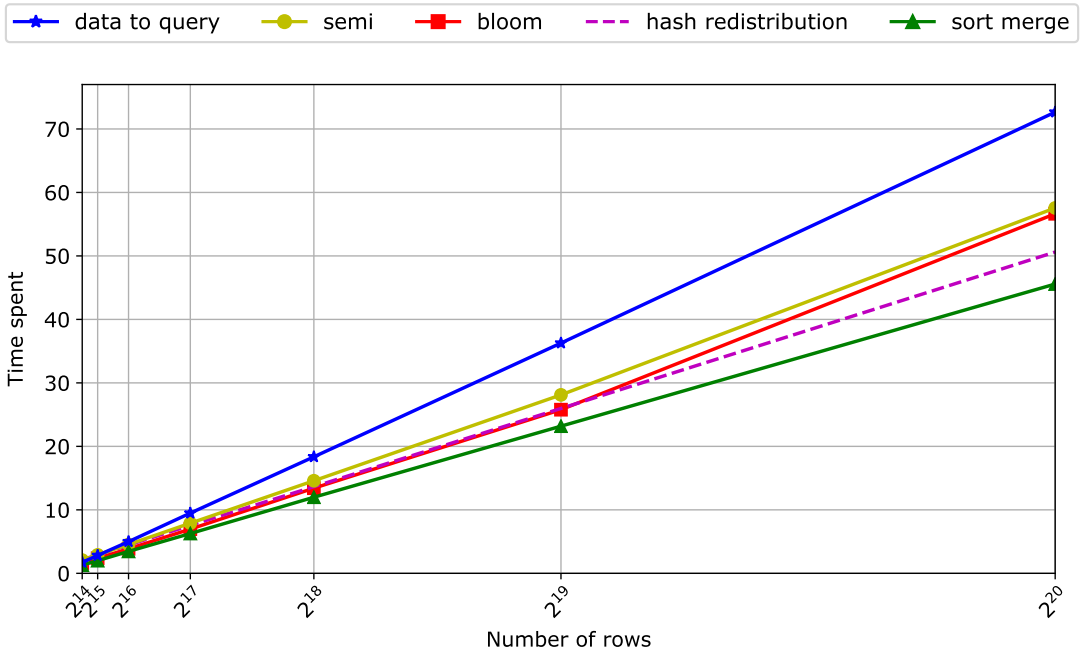
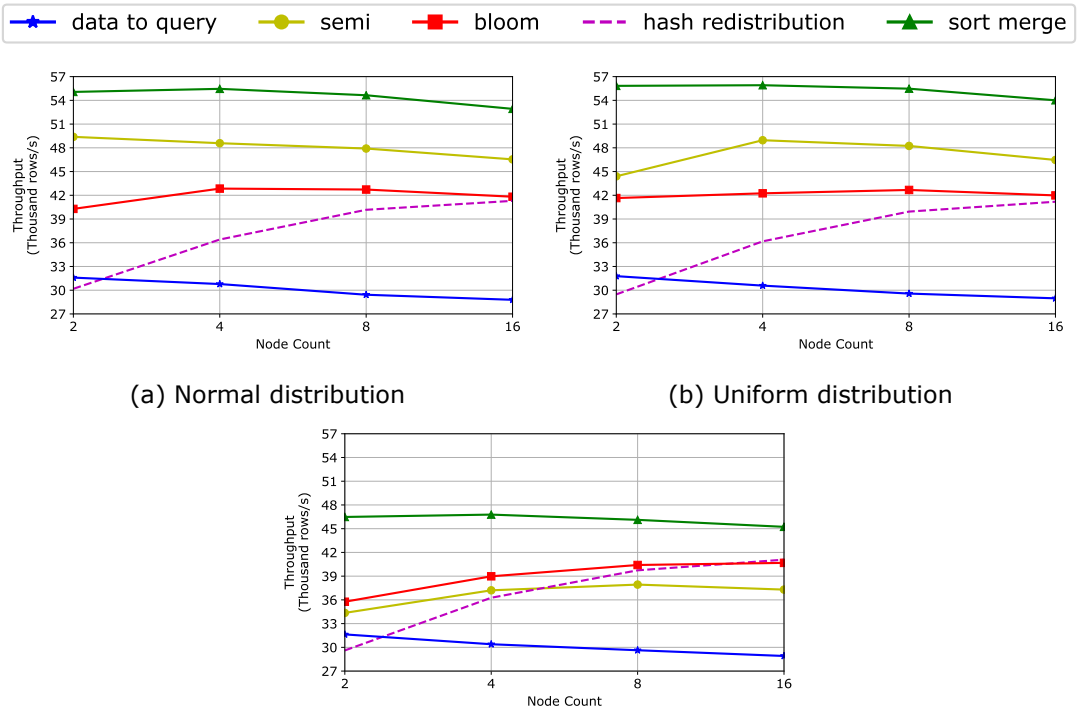


Figure 6.7: Varying datasize with 16 nodes and the 50% selectivity column

Figure 6.7 shows the vertical scalability of the strategies when using the 50% selectivity column. In it, we can see that the strategies scale linearly and that none of the strategies encounter any obstacles up to 2^{20} rows. We also see that the query-to-data strategies and sort-merge showcase better scalability than data-to-query, with sort-merge having the best. This is due to the overhead of the parallelization and data-movement these perform becoming a smaller portion of total time spent as the size of the dataset increases. Thus the payoff of this initial work becomes larger.

6.5.3 Distribution of values



(a) Normal distribution

(b) Uniform distribution

(c) 50% selectivity

Figure 6.8: Results for normal and uniform distribution, as well as 50% selectivity on up to 16 nodes, with 2^{19} rows

In figure 6.8, a comparison of the strategies when using the normal distribution, uniform distribution and 50% selectivity dataset can be seen. It is important to note that the selectivity of the joins performed on all of these datasets is 50%, the difference between them being that the normal and uniform distribution dataset contains a significant number of duplicate values, while the selectivity column contains strictly unique values.

In the results, we see that semi, bloom, and sort-merge significantly improve in the presence of duplicates. While hash redistribution and data-to-query stay mostly the same for each plot.

This means the experiment to produce a skewed distribution in order to negatively affect the performance of hash redistribution has not yielded positive results. From this, we can deduce that hash redistribution has been able to redistribute the data with little to no skew, even with the big groups of duplicate values present in the normal distribution, or that the skew produced was not big enough to significantly impact performance.

Data-to-query was also not affected in any way by the different distributions, this was expected.

Semi- and Bloom-join redundancy reduction

In the results, we see that semi, bloom, and sort-merge significantly improve when duplicates are introduced to the dataset. It is clear that both semi and bloom have expressed the necessary information in a compressed manner, namely, the join columns they ship are significantly smaller and allow for more efficient filtering of the opposing table, and thus have been rewarded with greater performance. This aligns with the hypothesis, from section 4.11, about semi and bloom-join being able to take advantage of duplicate values in the join columns. Another interesting aspect of these results is that semi-join has surpassed bloom-join in the transition from the 50% selectivity column to the normal distribution, we will explore this further in section 6.5.5.

Sort-merge with duplicate values

Sort-merge also shows an increase in performance when using the duplicate value distributions. This is likely due to the merging implementation processing larger buffers of matches at once. In the unique value distributions, sort-merge has to retrieve a set of matches, there only being 1 from each table each round, generate combined rows and insert statements before getting the next set of matches. With the duplicate value distributions, more rows are processed in each one of these stages, reducing the total overhead of building up match buffers and moving back and forth. Sort-merge's overall performance lead for this sample, and in general, is more closely investigated in the section about selectivity 6.5.4.

6.5.4 Selectivity

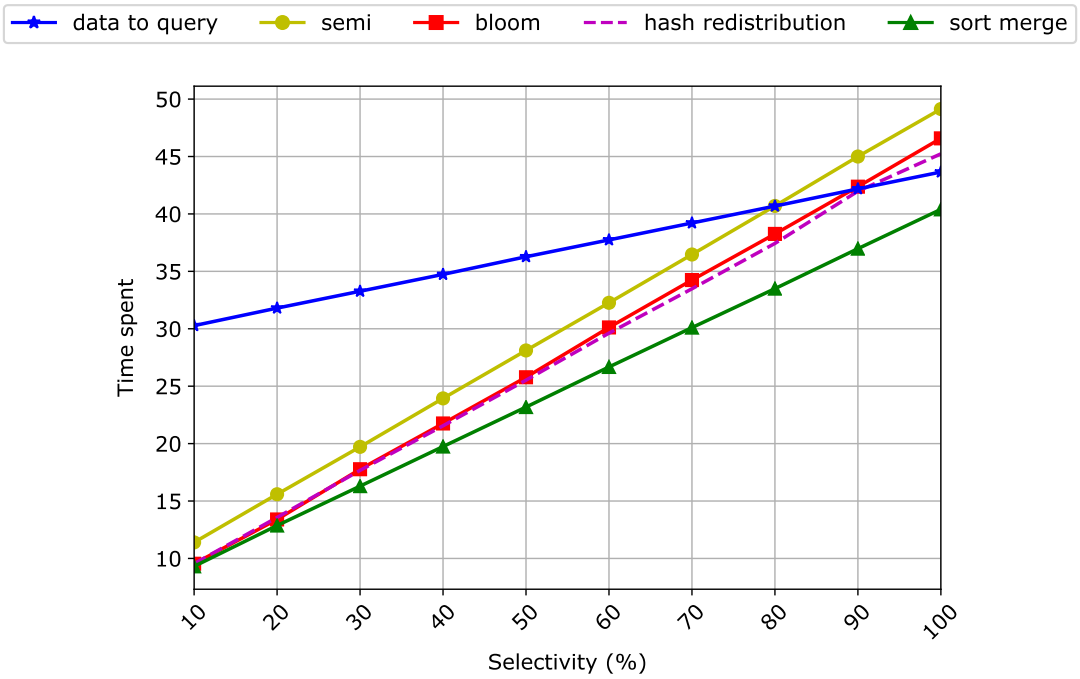


Figure 6.9: Varying selectivity with 16 nodes and 2^{19} rows

The selectivity result graph, which can be seen in figure 6.9, shows the performance of the strategies through different levels of selectivity in the joins. As expected data-to-query is not affected much. This because it does most of the heavy lifting before any join is executed.

The strategies performing the join before shipping entire rows, namely semi and bloom, perform well and use very little time when there are fewer rows in the end result. This is in line with our predictions. When moving towards full selectivity the performance of these strategies decreases and at 100% lose their advantage, and due to the overhead and relative complexity of their methods, perform worse than data-to-query. Hash redistribution follows a similar trend to bloom and semi, for a similar reason. The redistribution step of this strategy remains constant throughout, but the result being shipped back to the master becomes significantly larger as selectivity increases. Sort-merge also degrades as the selectivity goes up, but is still able to be faster than data-to-query in this test.

Investigating sort-merge advantage

The great performance of sort-merge throughout this test and others was a bit of a mystery at first because sort-merge collects just as much data as data-to-query

before actually joining any rows. However, upon further investigation, we discovered that the adverse effects displayed by the strategies in this test were not due to network shipping times, but rather time spent inserting rows into interim tables.

The sort-merge algorithm only inserts matching rows, unlike data-to-query, and therefore is able to significantly outperform it.

As for the other strategies, the reason was not as apparent and needed some further investigation. The query-to-data strategies do not do nearly as much insertion as data-to-query and do their work in parallel on multiple nodes.

One possible explanation was the fact that sort-merge does not use indices upon inserting rows, because it never uses a nested loop join like the other strategies. To test this, we added an unnecessary index to its final interim table. All the other strategies use indices because they all rely on a nested loop join at some point in their execution. The results of this can be seen in figure 6.10. As the graph shows there is a very small negative effect of generating the extraneous index in sort-merge. It does not, however, come close to explaining the performance gap between sort-merge and the query-to-data strategies.

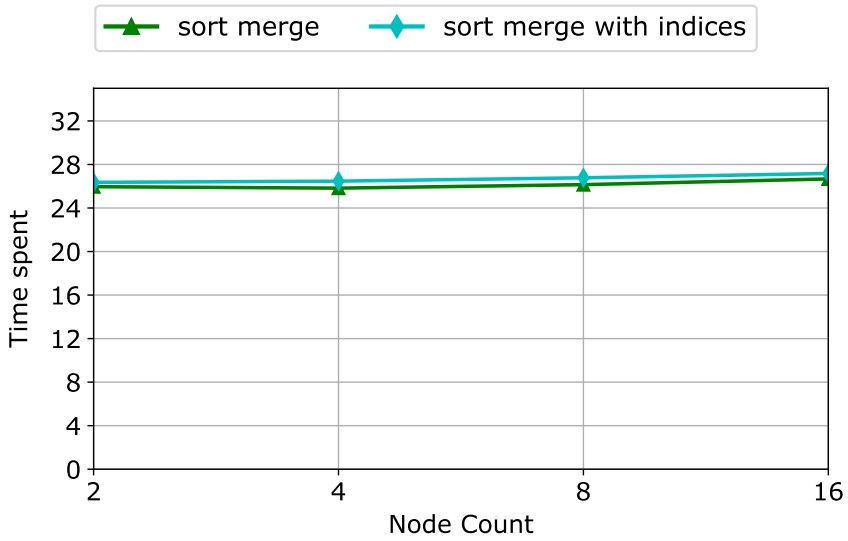


Figure 6.10: Sort-merge with and without indices. Varying selectivity with 16 nodes and 2^{19} rows

What it comes down to is the fact that insertion is a performance bottleneck in these tests. And when strategies make heavy use of inserts in their sequential stages they are punished for it. Sort-merge inserts the theoretical minimum amount, so is, therefore, the least affected by this issue. This is a limitation of the design/architecture of the system as insertion statements need to be parsed, and at the plugin architecture/abstraction level that is the only way to insert rows. It is also likely that the hardware used for these test magnify this effect, as all the nodes have

only one core, thus limiting the parallelism of our insertion logic. This is explored in section 6.7. The performance hit of row insertion can also be looked at as a sort of simulated network delay when looking at all strategies, except for sort-merge. Though, to get a clearer picture of the networking efficiency of all the strategies we have to introduce a significant network delay. This is explored in section 6.6

6.5.5 Semi vs Bloom

As seen in the previous results, such as figure 6.4, bloom-join generally performs better than semi-join. This is due to bloom using fewer inserts than semi. Specifically when it ships the filter instead of the join column. However, in the plot 6.8 showcasing the results of the normal and uniform value distributions, we see the opposite, semi having better throughput than bloom. Figure 6.11 shows the intersection of the two strategies, where semi surpasses bloom in performance.

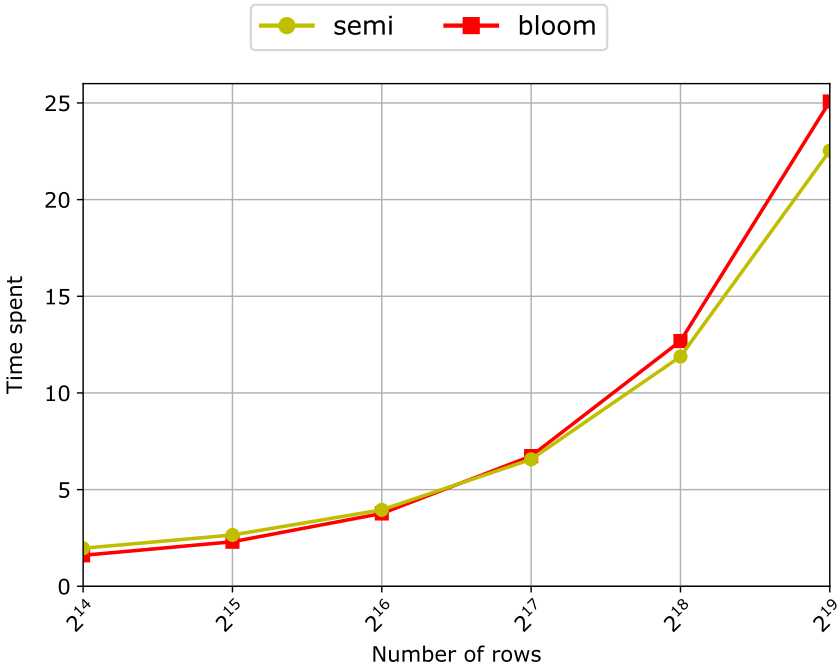


Figure 6.11: Semi- and Bloom-join with 16 nodes, using the normal distribution join column

An important aspect of the normal and uniform distributions is that when the size of the dataset increases the value range stays the same. This means that for a given node size the overhead of shipping the distinct values of the join column in semi-join, and generating and shipping the filter of bloom stays constant.

So to explain this we needed to look at why the characteristics of these datasets

negatively affected bloom and not semi. As the difference in the two strategies is the use of a bloom filter, this was the obvious area to investigate.

False positive rates for bloom filters

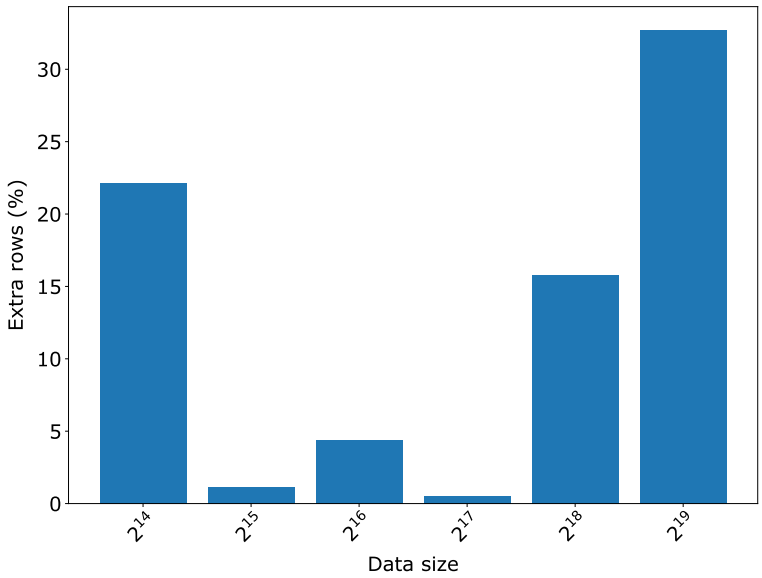


Figure 6.12: Extra rows included due to bloom false positive with normal distribution join column for the different data sizes tested

In figure 6.12 we can see a small sample of the number of rows wrongly included in the output of the Bloom filter. Here, the false positive rate of the filter produces, with the normal distribution dataset, up to 30% extra rows. This can be a problem when shipping the rows over a network. However, in this case, and the result in figure 6.11, the rhs normal column is used to create the filter and the filter is applied on the lhs table.

The lhs column used is a sequence of integers overlapping the ”rhs normal” column from the center, meaning it overlaps in the value range 300 to 500. Given this, the maximum number of true positives that can come from the filter is 200 rows. This gives the high sample of 30% false positives only 60 extra rows. And this remains constant through the increase in dataset size, as explained earlier. Therefore we can conclude that neither the network traffic of shipping the extra rows nor the insertion mechanism is to blame for the decrease in performance in relation to semi-join. The reason for this degradation is the extra processing of the nested loop performing the final join between the filtered rows and the original rhs table. Like the original, in this case comparatively large, rhs table is not guaranteed to be in-

dexed, even a small number of extra rows can have a large impact on performance. This becomes a bigger problem as the dataset size increases as well, as can be seen in the plot. This highlights an issue with using Bloom filters in that with certain configurations and data distributions the false positive rate of the filter can degrade performance significantly, either through extra processing or network volume.

We will review the possibilities for better tuning a bloom filter for use in joins in (the discussion) section 6.8.

6.6 Slow network simulation

To investigate network effects on the strategies and attempt to answer some of the hypotheses regarding networks, we in this section simulated decreased bandwidth and increased latency. This was also, in part, an effort to mitigate the large factor of insertion statements present in previous results.

6.6.1 Bandwidth

To test the effects of slower networks on the strategies we decreased the bandwidth between the nodes by throttling the network interface on each of the 16 virtual machines. We were, however, not able to produce an environment, with the VMs, in which the bandwidth of the network sufficiently starved the insertion logic used by the strategies, without rate limiting to an unusable rate where other mechanisms started to break down. This, as discussed earlier, might be a problem with the small number of cores given to the VMs, limiting the parallelism of the insertion logic. This is investigated in section 6.7. Thus, the results we produced in this test all showed the same trends as the previous virtual machine tests.

6.6.2 Latency

For this test, we added artificial latency to the network interface of each of the 16 virtual machines. We did this to see how the different strategies respond to networks with higher latency. Specifically, we added a variable delay of $10ms \pm 5ms$ with a 25% correlation between the delay of the previous and current packet delay. The tc-netem[7] command we used is shown here:

```
1 tc qdisc add dev ens3 root netem delay 10ms 5ms 25%
```

[7] http://man7.org/linux/man-pages/man8/tc-netem.8.html

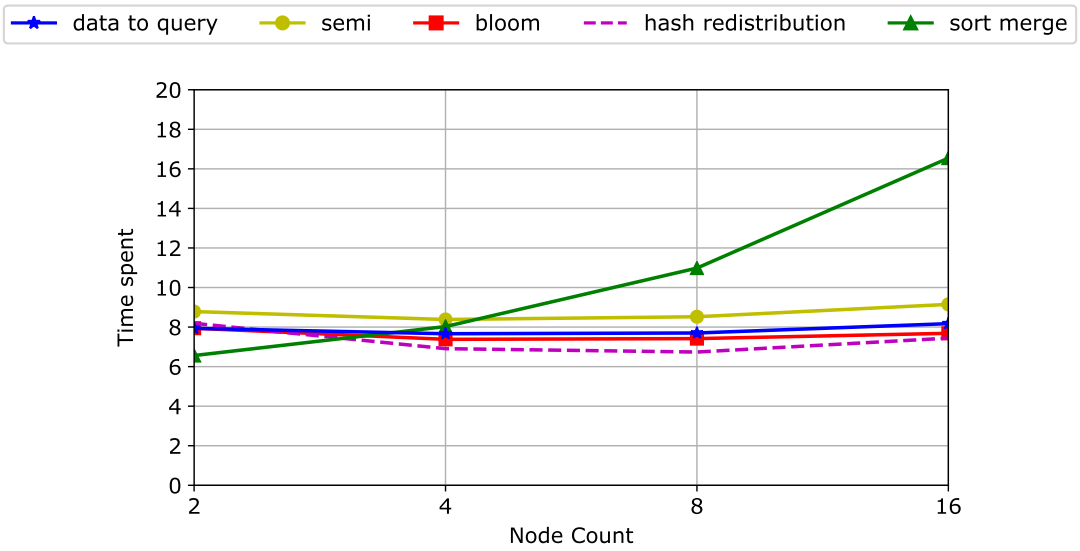


Figure 6.13: Scaling number of nodes with the 50% selectivity column, with 2^{18} rows

The results of the latency test can be seen in figure 6.13. Here, data-to-query performs relatively better compared to earlier tests. This is because data-to-query sends out a small number of large requests, making it less sensitive to increased round-trip times. While other strategies have sequential stages and queries that they have to wait for in order to continue with execution, making latency a bigger pain point. This has impacted hash redistribution, semi, and bloom, weakening their performance and placing them right next to data-to-query. Sort-merge has been especially punished by the added latency, but for a different reason. As the Connector/C++ library does not support asynchronous execution of queries, sort-merge's order-by queries are sent out sequentially. Note that this does not mean it needs to fetch the entire result between each query, but it has to wait for a query to respond before sending out the next one. This makes sort-merge very sensitive to increased round-trip times, and explains why it displays poor performance in this test, but not in others.

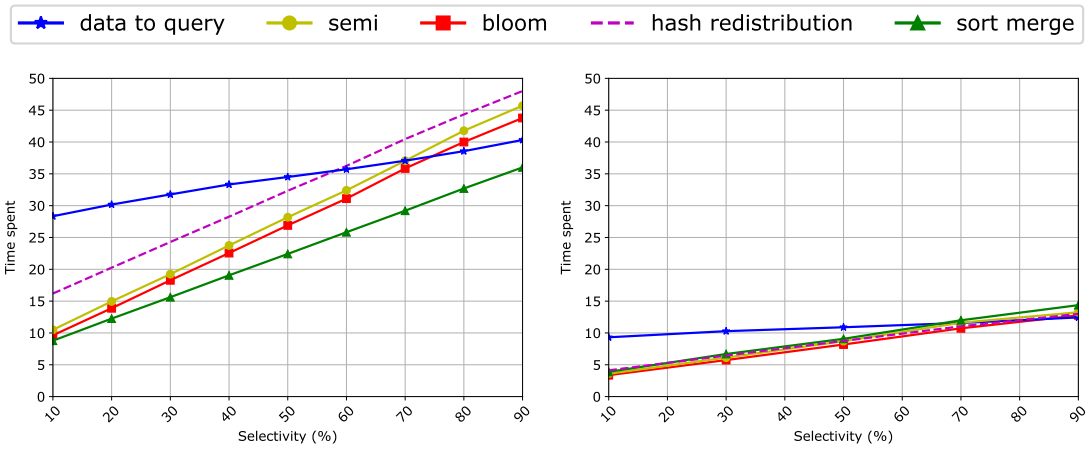
6.7 Physical machines

In section 6.5.4, we suspected that the insertion logic was slowed down by only running on one core. The reasoning behind this comes from the fact that the insertion logic is written to be concurrent and utilize multiple threads to handle the result sets coming from the other nodes. To get a more balanced picture of performance, and to try to mitigate the insertion factor, we decided to give the test suite a run on a couple of physical machines.

We ran tests on two machines, with each machine serving two MySQL instances, for a total of four MySQL instances. Both the machines have four-core processors, meaning one more core per MySQL instance compared to the virtual machine setup. In addition, the processor cores of the physical machine are notably faster, as can be seen in table 6.3.

Machine type	CPU Model (Intel)	Memory (GB)	Freq (GHz)
Physical	Core i7-7700[8]	32	3.60
Virtual	Xeon Platinum 8167M[9][10]	15	2.00

Table 6.3: Comparison of the specifications of the physical and virtual machines



(a) Virtual machines

(b) Physical machines

Figure 6.14: Comparison of VMs vs physical machines, varying selectivity with 4 nodes and 2^{19} rows

In figure 6.14, VM results and physical machine results of the same test case are displayed. Between them we see that the physical machines have been able to complete the queries in a much shorter timespan, indicating that the queries running on the VMs are CPU bound.

[10] https://ark.intel.com/content/www/us/en/ark/products/97128/intel-core-i7-7700-processor-8m-cache-up-to-4-20-ghz.html

[10] https://cloud.oracle.com/compute/virtual-machine/features (Accessed: 2019-05-24)

[10] Each virtual machine only have one OCPU (see section 6.3).

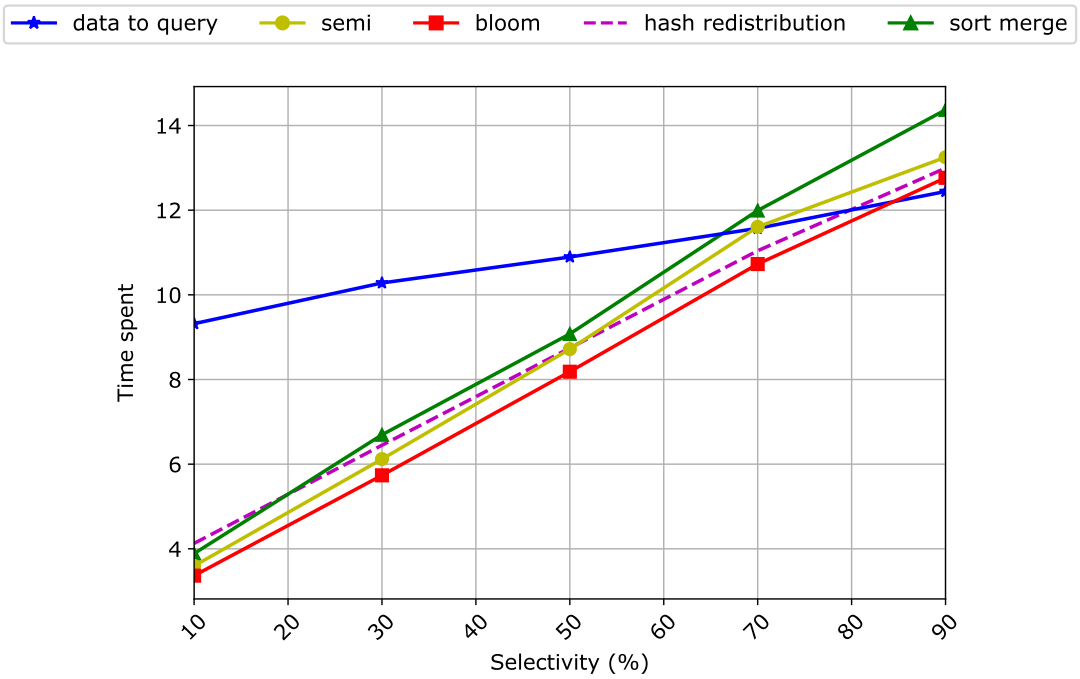


Figure 6.15: Varying selectivity with 4 nodes and 2^{19} rows

When zooming in on the physical machine results, see figure 6.15, it becomes apparent that adding more cores changes the landscape between the strategies a bit as well. Insertion is no longer as dominating of a factor. This is clear to see as bloom, semi, and hash-redistribution, are able to outperform sort-merge, even though all of them have more insert statements.

As explained in design, bloom-join spends processing time in order to lower the total amount of data sent. And we can see that when using machines with more available processing power bloom-join is rewarded for this tactic.

The parallelism of semi, bloom, and hash redistribution gets less hampered by the insertion overhead of previous tests, and they are therefore able to beat the less parallel sort-merge strategy.

Adding more power to the nodes increases the favorability of the more parallel and processing heavy strategies.

6.8 Discussion

Through the evaluation of the strategies, we have highlighted strengths and weaknesses in the theory, design, and implementation of all the strategies. We have shown the merits of parallelism through the results of hash redistribution, showing that doing some work upfront, in order to parallelize execution, can be very

beneficial when there are multiple nodes that can contribute.

We have shown indications as to how network communication can be expensive and impact performance in a distributed system, but in most of our tests, this was not a dominating factor. This means that with a sufficiently speedy network, e.g. a data center setting, network cost is not likely to be a big concern when choosing a join strategy and that the weight should really be put on maximizing utilization of processing power in the system. This is in line with earlier research on the subject, see section 2.4.

We looked at how the strategies deal with different datasets, and how different properties affect the strategies. Semi and bloom-join are able to capitalise on duplicates in the dataset, i.e. the uniform and normal distributions. All the query-to-data strategies are inefficient at executing joins that have 100% selectivity, but perform better than data-to-query when selectivity is lower. Hash redistribution is mostly unaffected by the different dataset properties in our tests, showing it to be a very general strategy that fits many cases.

We also investigated the differences between semi- and bloom-join, and showed that bloom is able to outperform semi through its usage of the filter to represent the join column, but also that in some cases produces a significant number of false positives, making it less reliable and predictable.

Using the VMs, we uncovered some of the limitations of the plugin architecture we used. The biggest one being the insertion-bottleneck, which ended up becoming a dominant factor in many of the VM tests. We were, however, able to produce tests with more powerful machines that dampened the insertion-factor, and gave a more nuanced perspective of the relative performance of the strategies.

We compared a more specialized algorithm, sort-merge, to the lightweight data movement strategies. We found that our sort-merge implementation was demonstrably more suited to the distribution problem space, but was still outperformed when using more powerful nodes. This tells us that the data movement strategies, using nested-loop at the local level and the plugin philosophy in general, all show promise.

Chapter 7

Conclusion

Join-queries in distributed systems can be executed using a number of strategies that vary in communication requirements, compute-heaviness, and complexity. Evaluating different ways to facilitate distributed join query execution has been the goal of this project.

We have looked at previous research into the field of distributed joins, and investigated existing products and how they handle them. From that process, we selected five strategies to investigate: data-to-query, semi-join- bloom-join, hash redistribution, and sort-merge. And have presented the design, implementation, and evaluation of these five join-strategies in a distributed MySQL plugin based system.

Through the design phase, we have presented the architecture of the plugin and the design of the different strategies. The design details a plugin system enabling a distribution layer for the strategies to operate over multiple MySQL instances. It was designed to allow swapping between multiple strategies. Each strategy was modeled and detailed in sequence diagrams, through several iterations and discussions. The time spent on this allowed for a better understanding of how each strategy works and thoughts on how to best implement them. The resulting design served as a great basis for the implementation.

Even given a good design there are almost always some compromises that have to be made during implementation. Our development process has been an iterative one, with a focus on code reusability and modularity between the different strategies. During it, we have encountered limitations in the plugin architecture and devised solutions and workarounds to make our strategies work. We have gotten to know MySQL well and explored its plugin API extensively. When preparing for tests we optimized the system and fixed outstanding problems to ensure that the evaluation would work smoothly. This has resulted in a plugin system capable of executing all our strategies in a distributed environment.

The results of the evaluation have shown how the strategies perform in relation to each other and exposed some of their strengths and weaknesses.

The data-to-query strategy, being the simplest strategy, had the overall poorest performance. However, its simplicity made it very consistent and in some of the corner test cases, e.g. joins with 100% selectivity it prevails. The consistency displayed by data-to-query shows the value that can come from using a simpler and more general strategy.

Hash redistribution was consistently able to evenly distribute the workload and performed well when adding more nodes to the system. Showing that doing some work upfront in order to maximize parallelity is worth it in the long run, especially when adding more nodes.

Semi- and bloom-join did generally well, especially in when running on more powerful machines. Showing the cleverness of these strategies, and their viability when processing power is more abundant. They are also better suited to certain configurations. E.g. they do great with a normal distribution of values.

When comparing semi and bloom against each other, we discovered a few key differences. Semi sends more data and therefore hits the architectural limits sooner than bloom. Bloom can sometimes produce false positives, making it less predictable.

Sort-merge, being a more bottom-up strategy using its own join algorithm, had the best performance in the virtual machine tests. Highlighting the difference between a specialized strategy versus the lighter weight data-movement strategies. The fact that a simple, but more specialized strategy, did so well challenges the lightweight plugin philosophy. It indicates that a more powerful plugin interface might be needed to make the lightweight plugin a viable solution. Although, when adding more power sort-merge was beaten by all the query-to-data strategies. Displaying that the theory, design, and implementation of these are sound.

In total, we have shown that no one strategy fits all and that the choice of strategy depends on what topology, data, hardware, etc, are available. Any system should be explicit in the choice of an appropriate strategy given a certain problem. Distributed database systems built from top-to-bottom have the advantage of controlling the problem space to a higher degree, thus allowing it to make decisions ahead of time to facilitate a certain execution strategy. While a plugin-based system, like ours, needs to be able to adapt more to the environment in which it runs. For MySQL, this means using the strategies that are best able to take advantage of the parser-access, the indexed nested loop join, and get around the limitations concerning mutating data in the fastest way. Namely, hash redistribution, bloom-join, and sort-merge. Although, in general, a system should support multiple strategies either as a user choice, a runtime decision, or as a hybrid solution.

Chapter 8

Future work

We will in this section present proposals for future work to improve and expand upon the plugin system, the evaluation process and the project as a whole. We will present potential solutions to standing problems, optimizations, new features and interesting avenues for further development. Mainly focusing on the aspects of this project, and beyond, which are the most interesting to us, and that we wish to see completed.

8.1 Further testing

When testing a system like ours there is always the possibility of adding more nodes, bigger datasets, more powerful machines, etc. Our resources were limited, however. We completed the test cases we set out to do and expanded on the test cases that produced interesting or ambiguous results to properly highlight the differences between the strategies. Although, there are some aspects we would like to look further in to. We would like to closer investigate bloom filters and find the most efficient configuration of it in different join scenarios. And we would like to test with several more powerful machines, up to 16 perhaps, to get a more complete picture of the scalability and network effects when insertion is less of a factor. Other possibilities for testing include running an industry standard benchmark, or using production data, to get an idea of how the strategies perform with data representing real-life use cases.

8.2 Speeding up inserts

As discussed in the evaluation chapter 6 our strategies were often performance bound by the insertion of data when running on the slow virtual machines. Having to parse and process our insertions as SQL statements certainly did not help in this

regard. Optimizing inserts in our system is, therefore, a potentially fruitful avenue for further development of this project.

One potential immediate solution is to use a plugin for the InnoDB storage engine called ”innodb-memcached”. This plugin hosts a Memcached server and allows a client to mutate InnoDB tables by sending commands through the Memcached protocol [1]. By itself, Memcached is a high-performance in-memory key-value store intended to lighten database load by serving as a cache [2]. In the innodb-memcached plugin the in-memory storage of memcached is replaced by the InnoDB buffer pool. This way Memcached commands can be used to manipulate the consistent and reliable storage of InnoDB through a simpler and highly performant interface. For our plugin, this could allow inserts to bypass all the layers of the MySQL server, e.g. parsing, before being stored in interim tables [3].

However, this solution creates some dependencies. The plugin becomes dependent on the InnoDB storage engine, breaking the abstraction of the plugin. And it becomes dependent on Memcached, which does not support all platforms, the big one missing being Windows. A more general and plugin-esque (not breaking abstraction and not dependent on the storage engine) solution would require a more powerful plugin API.

8.3 Closer integration with MySQL

Developing a system using the plugin-architecture for MySQL, without breaking the abstractions and conveniences of a plugin, places you at the mercy of the plugin API available. If you need anything beyond the offered functionality, you are left to either modify the MySQL server and break compatibility and the abstraction, or use cumbersome workarounds.

In our plugin, we made use of the interface for traversing and reading the parse-tree generated for a query. While this part worked nicely, we did find the overall API to be lacking in functionality when it came to performing other tasks. For example, the fact that there is not an interface in regards to querying and mutating tables. To get around this we ended up creating new application-level sessions to the MySQL server for simple queries and, as previously discussed, insert statements, which was sub-optimal.

A remedy to this could be to expand the plugin API by giving it more functionality. We would like to see something like a *Object relational mapping* (ORM) style interface for constructing parse-trees. By directly building parse-tree structures, or close relatives, a plugin would have more fine-grained control over querying at a lower cost. Allowing plugins to affect query plans and lower level optimizations of query execution makes sense because a plugin is very different from a user application,

[1] https://dev.mysql.com/doc/refman/8.0/en/innodb-memcached.html
[2] https://www.memcached.org/about
[3] https://dev.mysql.com/doc/refman/8.0/en/innodb-memcached-intro.html

as it is loaded as a compiled component of the MySQL server itself. As discussed earlier, the innodb-memcached plugin provides an alternative interface to interact with InnoDB. A similar interface in the plugin API providing CRUD operations on the SQL level could also be very useful.

Changes like these could allow for more powerful plugins in the future, and make a distribution plugin a viable solution to add a distribution layer to MySQL.

8.4 Join algorithms

Natively MySQL supports only one join algorithm, a sequential nested-loop join. So in our plugin, this was used for every strategy, except for sort-merge. In theory, though, nested loop joins can be parallelized to an extent, by using pipelining on the inner relation [8]. However, this means you need to have the entire outer relation before starting the loop.

Other joins algorithms can also be parallelized in different ways, and some to a greater extent. We investigated sort-merge in our project, and seeing how well it worked it makes sense to investigate the other major one we covered in the theory chapter 2, namely the hash join algorithm.

As seen in the state-of-the-art chapter 3 it is a common method in distributed database systems. It lends itself well to parallelism in that nodes can hash rows as it receives them.

Implementing a distribution aware pipelined nested-loop and a distributed hash-join goes beyond scope and philosophy of this project, but could be interesting as a further comparison of the lightweight plugin architecture to a heavier one.

8.5 Hybrid join strategies

Looking at hybrids of the strategies we have evaluated through this project might yield a better strategy than either of them by themselves. As stated in the state-of-the-art chapter 3 it is common for systems to combine different techniques to create a general and efficient solution for real-world use. One example being the hash redistribution + sort-merge join hybrid of CockroachDB. An investigation into the different possible hybrids might prove useful as an extension of this project.

8.6 More Join types

In our research, we have been looking at one specific type of join, inner equi-joins. Other common join types are outer- and full joins, as well as joins with join-clauses using other operators than equality, e.g. ”greater-than” [4]. It could be interesting to

[4] https://dev.mysql.com/doc/refman/8.0/en/join.html

devise and evaluate strategies for these as they might have different performance shortcuts and pitfalls compared to inner equi-joins. One example is that for greater- and less-than operators hash redistribution and hash joins become useless. This is because, as mentioned in the design phase, once you hash a sequence of numbers the transitive relationships of the $<$ and $>$ operators cannot be guaranteed. Another case is full outer joins. When doing full outer joins the result can potentially contain twice the number of rows as the input tables. It is, therefore, reasonable to assume that the strategies performing data-retrieval before any joins will prevail here, e.g. data-to-query and sort-merge. Other cases might be more nuanced and an investigation into them might uncover interesting results.

8.7 Consistency

An important aspect of RDMBSs is consistency. This is however an ambiguous term when talking about distributed databases, as it might refer to either consistency in *ACID* [29] (Atomicity, Consistency, Isolation, Durability) or in *CAP* theorem [30] (Consistency, Availability, Partition tolerance). ACID properties are required of a database system if it is to perform transactions. The CAP theorem talks about the fundamental properties of a distributed system, and how only two of the three properties can be guaranteed.

As part of ACID - consistency is a guarantee that data that can be read from the database always is in a valid state according to the rules put in place both by the user and the database system. E.g. upholding primary and foreign key relations or ensuring a unique value is actually unique.

When distributing a database consistency in CAP comes into play. It states that all the replicas of a row have the exact same value at all times. This becomes a challenge when introducing replication into the system. Replication is a common feature used for availability and load-balancing, by introducing replicas of records/values which can be used in parallel. This opens up for another avenue of inconsistencies as changes to any replica need to be propagated to all other replicas.

In distributed systems, all of these become more difficult to maintain, because there are multiple fully autonomous nodes operating in parallel. Of course, this only becomes an issue when mutating data or tables, which has not been a part of this project. If one is to make a full-fledged system one needs to address these issues at some point.

To solve it, there needs to be some type of locking or consistent sequence of operations between the nodes, keeping operations from interfering with each other.

Ensuring that multiple operations are performed atomically, consistent, isolated and durable across several nodes is a problem discussed in a lot of research, under the name of distributed concurrency control. There exist several different mechanisms and strategies, some using locks like 2 phase locking (2PL) and CALVIN. Some have a planning phase in which an execution order is determined using timestamps

or logic (CALVIN and Timestamp ordering). While some, like ”Optimistic concurrency control”, execute the operations first, and then validate if it was done in a serializable manner after-the-fact [31]. What they all have in common is that they produce a sequence of operations upholding the ACID properties, or they abort the transaction and try again later. Implementing a mechanism like one of these is required to maintain strict consistency in a distributed database system.

Bibliography

[1] A. Pavlo and M. Aslett, "What's really new with newsql?", *SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, Sep. 2016, ISSN: 0163-5808. DOI: 10.1145/3003665.3003674. [Online]. Available: http://doi.acm.org/10.1145/3003665.3003674.

[2] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*, 3rd ed. Springer-Verlag New York, 2011, ISBN: 978-1441988331.

[3] D. Kossmann, "The state of the art in distributed query processing", *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, Dec. 2000, ISSN: 0360-0300. DOI: 10.1145/371578.371598. [Online]. Available: http://doi.acm.org/10.1145/371578.371598.

[4] K.-U. Sattler, "Distributed join", in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 904–908, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_705. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_705.

[5] G. Graefe, "Parallel hash join, parallel merge join, parallel nested loops join", in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 2029–2030, ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1086. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_1086.

[6] D. A. Schneider and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines", in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 469–480, ISBN: 1-55860-149-X. [Online]. Available: http://dl.acm.org/citation.cfm?id=645916.672141.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems", in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, Boston, Massachusetts: ACM, 1984, pp. 1–8, ISBN: 0-89791-128-8. DOI: 10.1145/602259.602261. [Online]. Available: http://doi.acm.org/10.1145/602259.602261.

[8] H. Lu and M. J. Carey, "Some experimental results on distributed join algorithms in a local network", in *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, ser. VLDB '85, Stockholm, Sweden: VLDB Endowment, 1985, pp. 292–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286760.1286787.

[9] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment", *SIGMOD Rec.*, vol. 18, no. 2, pp. 110–121, Jun. 1989, ISSN: 0163-5808. DOI: 10.1145/66926.66937. [Online]. Available: %5Curl%7Bhttp://doi.acm.org/10.1145/66926.66937%7D.

[10] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation", *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1484–1495, Aug. 2014, ISSN: 2150-8097. DOI: 10.14778/2733004.2733020. [Online]. Available: http://dx.doi.org/10.14778/2733004.2733020.

[11] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler, "Distributed join algorithms on thousands of cores", *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 517–528, Jan. 2017, ISSN: 2150-8097. DOI: 10.14778/3055540.3055545. [Online]. Available: https://doi.org/10.14778/3055540.3055545.

[12] X. Wang and M. Cherniack, "Avoiding sorting and grouping in processing queries", in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03, Berlin, Germany: VLDB Endowment, 2003, pp. 826–837, ISBN: 0-12-722442-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315522.

[13] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie Jr., "Query processing in a system for distributed databases (sdd-1)", *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 602–625, Dec. 1981, ISSN: 0362-5915. DOI: 10.1145/319628.319650. [Online]. Available: http://doi.acm.org/10.1145/319628.319650.

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970, ISSN: 0001-0782. DOI: 10.1145/362686.362692. [Online]. Available: http://doi.acm.org/10.1145/362686.362692.

[15] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries", in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 149–159, ISBN: 0-934613-18-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671480.

[16] *Clustrixdb - how clustrixdb optimizes joins*, http://docs.clustrix.com/display/CLXDOC/Frequently+Asked+Questions#FrequentlyAskedQuestions-HowdoesClustrixDBoptimizejoins, Accessed: 2019-01-22.

[17] *Clustrixdb - scaling joins*, http://docs.clustrix.com/display/CLXDOC/Evaluation+Model#EvaluationModel-JoinswithMassivelyParallelProcessing(ClustrixDB), Accessed: 2019-01-22.

[18] *Mysql federated engine*, https://dev.mysql.com/doc/refman/8.0/en/federated-usagenotes.html, Accessed: 2019-01-21.

[19] *Mysql cluster 7.2 ga released, delivers 1 billion queries per minute*, https://www.mysql.com/why-mysql/white-papers/mysql-cluster-7.2-ga.html, Accessed: 2019-01-29.

[20] *Cockroachdb - join expressions*, Accessed: 2019-01-23. [Online]. Available: https://www.cockroachlabs.com/docs/stable/joins.html.

[21] *Spanner: Becoming a sql system*, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/46103.pdf, Accessed: 2019-01-23.

[22] *Memsql documentation - distributed sql*, https://docs.memsql.com/concepts/v6.7/distributed-sql/, Accessed: 2019-01-23.

[23] *Ignite sql - distributed joins*, https://apacheignite-sql.readme.io/docs/distributed-joins, Accessed: 2019-01-23.

[24] N. Provos and D. Mazières, "A future-adaptable password scheme", in *USENIX Annual Technical Conference, FREENIX Track*, 1999. [Online]. Available: http://www.usenix.org/events/usenix99/provos.html.

[25] C. Percival, "Stronger key derivation via sequential memory-hard functions", in *BSDCan - The BSD Conference*, May 2009. [Online]. Available: https://www.bsdcan.org/2009/schedule/attachments/87_scrypt.pdf.

[26] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, ISBN: 0-201-89685-0.

[27] G. Fowler, L. C. Noll, and K.-P. Vo, *The fnv non-cryptographic hash algorithm*, https://tools.ietf.org/pdf/draft-eastlake-fnv-16.pdf, Accessed: 2019-04-03.

[28] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560.

[29] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983, ISSN: 0360-0300. DOI: 10.1145/289.291. [Online]. Available: http://doi.acm.org/10.1145/289.291.

[30] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: 10.1145/564585.564601. [Online]. Available: http://doi.acm.org/10.1145/564585.564601.

[31] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control", *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, Jan. 2017, ISSN: 2150-8097. DOI: 10.14778/3055540.3055548. [Online]. Available: https://doi.org/10.14778/3055540.3055548.

Appendix A

Lundgren plugin source code

`lundgren/lundgren.cc`

```
/*  Copyright (c) 2015, 2017, Oracle and/or its affiliates. All rights
    reserved.

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License, version 2.0,
    as published by the Free Software Foundation.

    This program is also distributed with certain software (including
    but not limited to OpenSSL) that is licensed under separate terms,
    as designated in a particular file or component or in included license
    documentation.  The authors of MySQL hereby grant you an additional
    permission to link the program and your derivative works with the
    separately licensed software that they have included with MySQL.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License, version 2.0, for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA
    */

#include <ctype.h>
#include <mysql/components/services/log_builtins.h>
#include <mysql/plugin.h>
```

```
26 #include <mysql/plugin_audit.h>
27 #include <mysql/psi/mysql_memory.h>
28 #include <mysql/service_mysql_alloc.h>
29 #include <string.h>
30
31 #include "my_inttypes.h"
32 #include "my_psi_config.h"
33 #include "my_thread.h"  // my_thread_handle needed by mysql_memory.h
34
35 #include <iostream>
36
37 #include "plugin/lundgren/distributed_query_manager.h"
38 #include "plugin/lundgren/parse_tree_walker.h"
39 #include "plugin/lundgren/distributed_query.h"
40 #include "plugin/lundgren/query_acceptance.h"
41 #include "plugin/lundgren/join_strategies/data_to_query.h"
42 #include "plugin/lundgren/join_strategies/semi_join.h"
43 #include "plugin/lundgren/join_strategies/bloom_join/bloom_join.h"
44 #include "plugin/lundgren/join_strategies/sort_merge/sort_merge.h"
45 #include "plugin/lundgren/join_strategies/hash_redistribution.h"
46 #include "plugin/lundgren/helpers.h"
47
48 /* instrument the memory allocation */
49 #ifdef HAVE_PSI_INTERFACE
50 static PSI_memory_key key_memory_lundgren;
51
52 static PSI_memory_info all_rewrite_memory[] = {
53     {&key_memory_lundgren, "lundgren", 0, 0, PSI_DOCUMENT_ME}};
54
55 static int plugin_init(MYSQL_PLUGIN) {
56   const char *category = "sql";
57   int count;
58
59   count = static_cast<int>(array_elements(all_rewrite_memory));
60   mysql_memory_register(category, all_rewrite_memory, count);
61   return 0; /* success */
62 }
63 #else
64 #define plugin_init NULL
65 #define key_memory_lundgren PSI_NOT_INSTRUMENTED
66 #endif /* HAVE_PSI_INTERFACE */
67
68 static int lundgren_start(MYSQL_THD thd, mysql_event_class_t event_class,
```

```
69                          const void *event) {
70  if (event_class == MYSQL_AUDIT_PARSE_CLASS) {
71    const struct mysql_event_parse *event_parse =
72        static_cast<const struct mysql_event_parse *>(event);
73    if (event_parse->event_subclass == MYSQL_AUDIT_PARSE_POSTPARSE) {
74
75
76      if (!accept_query(thd, event_parse->query.str)) {
77        return 0;
78      }
79
80      bool is_join = detect_join(event_parse->query.str);
81
82      L_Parser_info *parser_info = get_tables_from_parse_tree(thd);
83
84      Distributed_query* distributed_query;
85
86      if (is_join) {
87
88        L_parsed_comment_args parsed_args =
          ↪  parse_query_comments(event_parse->query.str);
89
90        if (parser_info != NULL && parser_info->tables.size() > 2) {
91          parser_info->tables.pop_back(); //hack
92        }
93
94        switch(parsed_args.join_strategy) {
95        case SEMI:
96          distributed_query = make_semi_join_distributed_query(parser_info,
            ↪  parsed_args);
97          break;
98        case BLOOM:
99          distributed_query = make_bloom_join_distributed_query(parser_info,
            ↪  parsed_args);
100         break;
101       case SORT_MERGE:
102         distributed_query =
            ↪  execute_sort_merge_distributed_query(parser_info);
103         break;
104       case HASH_REDIST:
105         distributed_query =
            ↪  make_hash_redist_join_distributed_query(parser_info, parsed_args,
            ↪  event_parse->query.str);
```

```
106             break;
107           case DATA_TO_QUERY:
108           default:
109             distributed_query = make_data_to_query_distributed_query(parser_info,
                ↪  true);
110             break;
111           }
112
113       } else {
114         distributed_query = make_data_to_query_distributed_query(parser_info,
            ↪  false);
115       }
116
117
118       if (distributed_query == NULL) {
119           return 0;
120       }
121
122       execute_distributed_query(distributed_query);
123
124       size_t query_length = distributed_query->rewritten_query.length();
125       char *rewritten_query = static_cast<char
          ↪  *>(my_malloc(key_memory_lundgren, query_length+1, MYF(0)));
126       memset(rewritten_query, 0, query_length+1);
127
128       strncpy(rewritten_query, distributed_query->rewritten_query.c_str(),
          ↪  query_length);
129
130       MYSQL_LEX_STRING new_query = {rewritten_query, query_length};
131
132       mysql_parser_parse(thd, new_query, false, NULL, NULL);
133
134       delete distributed_query;
135
136       *((int *)event_parse->flags) |=
137           (int)MYSQL_AUDIT_PARSE_REWRITE_PLUGIN_QUERY_REWRITTEN;
138     }
139   }
140
141   return 0;
142 }
143
144 /* Audit plugin descriptor */
```

```
145 static struct st_mysql_audit lundgren_descriptor = {
146     MYSQL_AUDIT_INTERFACE_VERSION, /* interface version */
147     NULL,                          /* release_thd()     */
148     lundgren_start,                /* event_notify()    */
149     {
150         0,
151         0,
152         (unsigned long)MYSQL_AUDIT_PARSE_ALL,
153     } /* class mask        */
154 };
155 
156 /* Plugin descriptor */
157 mysql_declare_plugin(audit_log){
158     MYSQL_AUDIT_PLUGIN,   /* plugin type                    */
159     &lundgren_descriptor, /* type specific descriptor       */
160     "lundgren",           /* plugin name                    */
161     "Kristian Andersen Hole & Haavard Ola Eggen", /* author */
162     "Distributed query plugin", /* description                  */
163     PLUGIN_LICENSE_GPL,         /* license                      */
164     plugin_init,                /* plugin initializer           */
165     NULL,                       /* plugin check uninstall       */
166     NULL,                       /* plugin deinitializer         */
167     0x0002,                     /* version                      */
168     NULL,                       /* status variables             */
169     NULL,                       /* system variables             */
170     NULL,                       /* reserverd                    */
171     0                           /* flags                        */
172 } mysql_declare_plugin_end;
```

`lundgren/parse_tree_walker.h`

```
1  #include <mysql/service_parser.h>
2  #include <string.h>
3  #include <algorithm>
4  #include <iostream>
5  #include "plugin/lundgren/constants.h"
6  #include "plugin/lundgren/distributed_query.h"
7  #include "plugin/lundgren/partitions/node.h"
8  #include "plugin/lundgren/partitions/partition.h"
9  #include "sql/item.h"
10 #include "sql/table.h"
11 
```

```
12 #ifndef LUNDGREN_DQR
13 #define LUNDGREN_DQR
14
15 struct L_Item {
16   std::string sql;
17   Item::Type type;
18   std::string alias;
19 };
20
21 int catch_item(MYSQL_ITEM item, unsigned char *arg) {
22   std::vector<L_Item> *fields = (std::vector<L_Item> *)arg;
23
24   if (item != NULL) {
25     String s;
26     item->print(&s, QT_ORDINARY);
27
28     std::string item_sql = std::string(s.ptr());
29     // hack
30     item_sql.erase(std::remove(item_sql.begin(), item_sql.end(), '`'),
31                    item_sql.end());
32     item_sql.erase(std::remove(item_sql.begin(), item_sql.end(), '('),
33                    item_sql.end());
34     item_sql.erase(std::remove(item_sql.begin(), item_sql.end(), ')'),
35                    item_sql.end());
36
37     L_Item fi = {item_sql, item->type()};
38
39     // There is an alias provided
40     if (!item->item_name.is_autogenerated()) {
41
42       char *alias_buffer = new char[item->item_name.length()];
43       item->item_name.strcpy(alias_buffer);
44       fi.alias = std::string(alias_buffer);
45       delete alias_buffer;
46     }
47
48     fields->push_back(fi);
49   }
50
51   return 0;
52 }
53
54 int catch_table(TABLE_LIST *tl, unsigned char *arg) {
```

```
55
56   std::vector<L_Table> *tables = (std::vector<L_Table> *)arg;
57
58   if (tl != NULL) {
59     L_Table t = {std::string(tl->table_name)};
60     tables->push_back(t);
61     return 0;
62   }
63   return 1;
64 }
65
66 static void place_projection_in_table(L_Item field_item,
67                                       std::vector<L_Table> *tables,
68                                       bool where_transitive_projection) {
69
70   std::string projection = field_item.sql;
71
72   std::string field =
73       projection.substr(projection.find(".") + 1, projection.length());
74   for (auto &table : *tables) {
75     if (table.name == projection.substr(0, projection.find("."))) {
76       if (where_transitive_projection) {
77
78         // only add as where transitive if not in regular projection set
79         if (std::find(table.projections.begin(), table.projections.end(),
           ↪  field) == table.projections.end()) {
80           table.where_transitive_projections.push_back(field);
81         }
82         table.join_columns.push_back(field);
83       } else {
84         table.projections.push_back(field);
85         table.aliases.push_back(field_item.alias);
86       }
87       break;
88     }
89   }
90 }
91
92 static L_Parser_info *get_tables_from_parse_tree(MYSQL_THD thd) {
93
94   /*
95    * Walk parse tree
96    */
```

```
97
98  std::vector<L_Item> fields = std::vector<L_Item>();
99  mysql_parser_visit_tree(thd, catch_item, (unsigned char *)&fields);
100
101 std::vector<L_Table> tables = std::vector<L_Table>();
102
103 mysql_parser_visit_tables(thd, catch_table, (unsigned char *)&tables);
104
105 if (tables.size() == 0) {
106   return NULL;
107 }
108
109 std::string where_clause = "";
110 bool passed_where_clause = false;
111
112 std::vector<L_Item>::iterator f = fields.begin();
113
114 switch (f->type) {
115   case Item::FIELD_ITEM:
116
117     while (f != fields.end()) {
118       if (f->sql.find("=") != std::string::npos) {
119         where_clause += f->sql;
120         passed_where_clause = true;
121
122         f++;
123         continue;
124       }
125       if (f->type != Item::FIELD_ITEM) {
126         f++;
127         continue;
128       }
129
130       place_projection_in_table(*f, &tables, passed_where_clause);
131       f++;
132     }
133     break;
134   default:
135     break;
136 }
137
138 L_Parser_info *parser_info = new L_Parser_info();
139 parser_info->tables = tables;
```

```
140
141   if (where_clause.length() > 0) {
142
143     std::string clean_where_clause = tables[0].name + "." +
      ↪  tables[0].join_columns[0] + " = " + tables[1].name + "." +
      ↪  tables[1].join_columns[0];
144
145     parser_info->where_clause = clean_where_clause;
146   }
147
148   return parser_info;
149 }
150
151 #endif  // LUNDGREN_DQR
```

lundgren/distributed_query.h

```
1  #include <string.h>
2  #include "plugin/lundgren/partitions/partition.h"
3  #include "plugin/lundgren/partitions/node.h"
4
5  #ifndef LUNDGREN_DISTRIBUTED_QUERY
6  #define LUNDGREN_DISTRIBUTED_QUERY
7
8  struct L_Table {
9    std::string name;
10   std::string interim_name;
11   std::vector<std::string> projections;
12   std::vector<std::string> where_transitive_projections;
13   std::vector<std::string> join_columns;
14   std::vector<std::string> aliases;
15 };
16
17
18 struct L_Parser_info {
19   std::vector<L_Table> tables;
20   std::string where_clause;
21 };
22
23 //------------------------
24
25 struct Interim_target {
```

```
26      std::string interim_table_name;
27      Node node;
28      std::string index_name;
29      //bool is_temp;
30  };
31
32  struct Partition_query {
33    std::string sql_statement;
34    Node node;
35    Interim_target interim_target;
36  };
37
38  struct Stage {
39      std::vector<Partition_query> partition_queries;
40  };
41
42  struct Distributed_query {
43      std::string rewritten_query;
44      std::vector<Stage> stages;
45  };
46
47  #endif  // LUNDGREN_DISTRIBUTED_QUERY
```

`lundgren/distributed_query_manager.h`

```
1  #include <mysqlx/xdevapi.h>
2  #include <string.h>
3  #include <thread>
4  #include "plugin/lundgren/internal_query/internal_query_session.h"
5  #include "plugin/lundgren/distributed_query.h"
6  #include "plugin/lundgren/constants.h"
7
8  #ifndef LUNDGREN_DQM
9  #define LUNDGREN_DQM
10
11 std::string get_column_length(unsigned long length);
12 std::string generate_table_schema(mysqlx::SqlResult *res);
13 int connect_node(std::string node, Partition_query *pq);
14 std::string generate_connection_string(Node node);
15 void execute_distributed_query(Distributed_query* distributed_query);
16
17 std::string get_column_length(unsigned long length) {
```

```
18   return std::to_string(length/4);
19 }
20
21 std::string generate_table_schema(mysqlx::SqlResult *res) {
22   std::string return_string = "(";
23   for (uint i = 0; i < res->getColumnCount(); i++) {
24     return_string += res->getColumn(i).getColumnLabel();
25     return_string += " ";
26     switch (res->getColumn(i).getType()) {
27       case mysqlx::Type::BIGINT :
28         return_string += (res->getColumn(i).isNumberSigned()) ? "BIGINT" :
           ↪  "BIGINT UNSIGNED"; break;
29       case mysqlx::Type::INT :
30         return_string += (res->getColumn(i).isNumberSigned()) ? "INT" : "INT
           ↪  UNSIGNED"; break;
31       case mysqlx::Type::DECIMAL :
32         return_string += (res->getColumn(i).isNumberSigned()) ? "DECIMAL" :
           ↪  "DECIMAL UNSIGNED"; break;
33       case mysqlx::Type::DOUBLE :
34         return_string += (res->getColumn(i).isNumberSigned()) ? "DOUBLE" :
           ↪  "DOUBLE UNSIGNED"; break;
35       case mysqlx::Type::STRING :
36         return_string += "VARCHAR(" +
           ↪  get_column_length(res->getColumn(i).getLength()) + ")"; break;
37       default: break;
38     }
39     return_string += ",";
40   }
41   return_string.pop_back();
42   return return_string + ")";
43 }
44
45 int connect_node(std::string node, Partition_query *pq) {
46   mysqlx::Session s(node);
47   mysqlx::SqlResult res = s.sql(pq->sql_statement).execute();
48
49   if (res.hasData()) {
50     std::string table_schema = generate_table_schema(&res);
51
52     if (pq->interim_target.index_name.length() > 0) {
53       table_schema.pop_back();
54       table_schema += ", INDEX (" + pq->interim_target.index_name + "))";
55     }
```

```
56
57      mysqlx::Session
        ↪  interim_session(generate_connection_string(pq->interim_target.node));
58
59      std::string create_table_query = "CREATE TABLE IF NOT EXISTS " +
        ↪  pq->interim_target.interim_table_name + " " + table_schema + " " +
        ↪  INTERIM_TABLE_ENGINE ";";
60      interim_session.sql(create_table_query).execute();
61
62      mysqlx::Schema schema = interim_session.getSchema(pq->node.database);
63      mysqlx::Table table =
        ↪  schema.getTable(pq->interim_target.interim_table_name);
64
65      mysqlx::Row row;
66      while((row = res.fetchOne())) {
67        auto insert = table.insert();
68        insert.values(row);
69        int n = BATCH_SIZE - 1;
70        while(n-- && (row = res.fetchOne())){
71          insert.values(row);
72        }
73        insert.execute();
74      }
75
76      interim_session.close();
77    }
78    s.close();
79    return 0;
80  }
81
82  std::string generate_connection_string(Node node) {
83    return (std::string("mysqlx://")
84      + node.user + "@"
85      + node.host
86      + ":"
87      + std::to_string(node.port)
88      + "/" + node.database);
89  }
90
91  void execute_distributed_query(Distributed_query* distributed_query) {
92
93    for (auto &stage : distributed_query->stages) {
94
```

```
95      std::vector<Partition_query> partition_queries = stage.partition_queries;
96      const int num_thd = partition_queries.size();
97
98      std::thread *nodes_connection = new std::thread[num_thd];
99      for (int i = 0; i < num_thd; i++) {
100       std::string node = generate_connection_string(partition_queries[i].node);
101       nodes_connection[i] = std::thread(connect_node, node,
          ↪  &(partition_queries[i]));
102     }
103
104     for (int i = 0; i < num_thd; i++) {
105       nodes_connection[i].join();
106     }
107     delete [] nodes_connection;
108   }
109 }
110
111 #endif  // LUNDGREN_DQM
```

lundgren/constants.h

```
1  #ifndef LUNDGREN_CONSTANTS
2  #define LUNDGREN_CONSTANTS
3
4  #define INTERIM_TABLE_ENGINE "ENGINE = MEMORY"
5
6  #define PLUGIN_FLAG "distributed"
7
8  #define BATCH_SIZE 100000
9  #define BLOOM_SLAVE_BATCH_SIZE 100000
10
11 // SEMI (& bloom)
12 #define IGNORE_TABLE_PARTITIONS_FLAG "ignore_table_partitions"
13
14 // BLOOM JOIN
15
16 #define BLOOM_SLAVE_FLAG "bloom_slave"
17 #define BLOOM_FILTER_FLAG "bloom_filter"
18 #define BLOOM_FILTER_PARAMETER_COUNT_FLAG "filter_parameter_count"
19 #define BLOOM_FILTERED_INTERIM_NAME_FLAG "filtered_interim_name"
20 #define BLOOM_FILTER_REMOTE_TABLE_FLAG "remote_table_name"
21 #define BLOOM_FILTER_REMOTE_JOIN_COLUMN_FLAG "remote_join_column"
```

```
22 #define BLOOM_FILTER_MASTER_ID_FLAG "master_id"
23
24 // HASH REDISTRIBUTION
25 #define HASH_REDIST_SLAVE_FLAG "hash_redist_slave"
26
27 // SORT MERGE
28 #define SORT_MERGE_BATCH_SIZE 100000
29
30
31 #endif  // LUNDGREN_CONSTANTS
```

lundgren/join_strategies/data_to_query.h

```
1 #include <string.h>
2 #include "plugin/lundgren/distributed_query.h"
3 #include "plugin/lundgren/partitions/partition.h"
4 #include "plugin/lundgren/helpers.h"
5 #include "plugin/lundgren/join_strategies/common.h"
6
7 #ifndef LUNDGREN_DATA_TO_QUERY
8 #define LUNDGREN_DATA_TO_QUERY
9
10
11 // Pure Distributed_query strategy, can be fed into DQM
12
13 static Distributed_query *make_data_to_query_distributed_query(L_Parser_info
  ↪  *parser_info, bool is_join) {
14
15   std::vector<L_Table> tables = parser_info->tables;
16   std::string where_clause = parser_info->where_clause;
17
18
19   std::vector<Partition_query> partition_queries;
20
21   for (auto &table : tables) {
22
23     std::vector<Partition> *partitions =
24         get_partitions_by_table_name(table.name);
25
26     if (partitions == NULL) {
27       return NULL;
28     }
```

```
29
30    std::string partition_query_string = "SELECT ";
31
32    partition_query_string +=
       ↪  generate_projections_string_for_partition_query(&table);
33
34    std::string from_table = " FROM " + std::string(table.name);
35    table.interim_name = generate_interim_name();
36
37    partition_query_string += from_table;
38
39    if (!is_join && where_clause.length() > 0)
40      partition_query_string += " WHERE " + where_clause;
41
42    for (std::vector<Partition>::iterator p = partitions->begin();
43         p != partitions->end(); ++p) {
44
45      Interim_target interim_target;
46
47       if (is_join) {
48        interim_target = {table.interim_name, SelfNode::getNode(),
          ↪  table.join_columns[0]};
49
50       } else {
51        interim_target = {table.interim_name, SelfNode::getNode()};
52       }
53
54
55      Partition_query pq = {partition_query_string, p->node, interim_target};
56      partition_queries.push_back(pq);
57    }
58
59    delete partitions;
60  }
61
62  /*
63   * Generate final rewritten query
64   */
65
66  std::string final_query_string;
67
68  if (is_join) {
69
```

```
70      final_query_string = generate_final_join_query_string(tables,
        ↪  where_clause);
71
72    } else {
73
74      final_query_string = "SELECT ";
75
76      L_Table first_table = tables[0];
77      std::vector<std::string>::iterator p = first_table.projections.begin();
78
79      while (p != first_table.projections.end()) {
80        final_query_string += first_table.interim_name + "." + *p;
81        ++p;
82        if (p != first_table.projections.end()) final_query_string += ", ";
83      }
84      final_query_string += " FROM " + first_table.interim_name;
85    }
86
87    // Construct distributed query object
88    Distributed_query *dq = new Distributed_query();
89
90    Stage stage = {partition_queries};
91    dq->stages.push_back(stage);
92    dq->rewritten_query = final_query_string;
93
94    return dq;
95  }
96
97  #endif  // LUNDGREN_DATA_TO_QUERY
```

lundgren/join_strategies/semi_join.h

```
1  #include <string.h>
2  #include "plugin/lundgren/distributed_query.h"
3  #include "plugin/lundgren/helpers.h"
4  #include "plugin/lundgren/partitions/partition.h"
5  #include "plugin/lundgren/join_strategies/common.h"
6  #include "plugin/lundgren/constants.h"
7
8  #ifndef LUNDGREN_SEMI_JOIN
9  #define LUNDGREN_SEMI_JOIN
10
```

```
11 // Semi join
12
13 static std::string semi_join_generate_final_join_query_string(L_Table
   ↪  *stationary_table, L_Table *remote_table, std::string join_on);
14
15 // n = 1
16 static Distributed_query
   ↪  *make_one_sided_semi_join_distributed_query(L_Parser_info *parser_info
   ↪  MY_ATTRIBUTE((unused)), L_Table* stationary_table, L_Table* remote_table,
   ↪  std::vector<Partition>* remote_partitions) {
17
18   //std::vector<L_Table> *tables = &(parser_info->tables);
19   std::string where_clause = parser_info->where_clause;
20
21
22   std::vector<Stage> stages;
23
24   stationary_table->interim_name = generate_interim_name();
25   remote_table->interim_name = generate_interim_name();
26
27   // STAGE 1
28   Stage stage1;
29
30   std::string stationary_join_column = stationary_table->join_columns[0];
31   std::string remote_join_column = remote_table->join_columns[0];
32
33   std::string join_column_projection_query_string = "SELECT DISTINCT " +
     ↪  stationary_join_column + " FROM " + stationary_table->name;
34
35   // std::vector<Node> target_nodes;
36
37   for (auto &p : *remote_partitions) {
38     // target_nodes.push_back(p.node);
39     Interim_target interim_target = {stationary_table->interim_name, p.node,
       ↪  stationary_join_column}; // index
40     Partition_query pq = {join_column_projection_query_string,
       ↪  SelfNode::getNode(), interim_target};
41
42     stage1.partition_queries.push_back(pq);
43   }
44
45   stages.push_back(stage1);
46
```

```
47   // STAGE 2
48   Stage stage2;
49
50   std::string semi_join_query_string = "SELECT ";
51
52   semi_join_query_string +=
     ↪  generate_projections_string_for_partition_query(remote_table);
53
54   semi_join_query_string += " FROM " + remote_table->name +
55                             " JOIN " + stationary_table->interim_name +
56                             " ON " + stationary_table->interim_name + "." +
                               ↪  stationary_join_column +
57                             " = " + remote_table->name + "." +
                               ↪  remote_join_column;
58
59   Interim_target stage2_interim_target = {remote_table->interim_name,
     ↪  SelfNode::getNode(), remote_join_column}; // index
60
61   for (auto &p : *remote_partitions) {
62     Partition_query pq = {semi_join_query_string, p.node,
       ↪  stage2_interim_target};
63     stage2.partition_queries.push_back(pq);
64   }
65
66   stages.push_back(stage2);
67
68   // Construct distributed query object
69   Distributed_query *dq = new
     ↪  Distributed_query{semi_join_generate_final_join_query_string(stationary_table,
     ↪  remote_table, where_clause), stages};
70
71   delete remote_partitions;
72
73   return dq;
74
75   // TODO: remember delete remote_partitions
76 }
77
78 // n > 1
79 static Distributed_query
   ↪  *make_recursive_semi_join_distributed_query(L_Parser_info *parser_info
   ↪  MY_ATTRIBUTE((unused)), L_Table* remote_table, std::vector<Partition>
   ↪  *remote_partitions) {
```

```
80
81    std::vector<Stage> stages;
82
83    Stage stage1;
84
85    std::string join_union_interim_table_name = generate_interim_name();
86
87    //Distributed Partition queries
88
89    std::string recursive_distributed_join_query_string = "/*" PLUGIN_FLAG
      ↪  "<join_strategy=semi," IGNORE_TABLE_PARTITIONS_FLAG "=";
90    recursive_distributed_join_query_string += remote_table->name + ">*/";
91
92    recursive_distributed_join_query_string +=
      ↪  generate_join_query_string(parser_info->tables,
      ↪  parser_info->where_clause, false);
93
94    Interim_target interim_target = {join_union_interim_table_name ,
      ↪  {SelfNode::getNode()}};
95
96    for (auto &p : *remote_partitions) {
97      Partition_query pq = {recursive_distributed_join_query_string, p.node,
        ↪  interim_target};
98      stage1.partition_queries.push_back(pq);
99    }
100
101   stages.push_back(stage1);
102
103   // Final query
104
105   std::string final_query_string = "SELECT * FROM " +
      ↪  join_union_interim_table_name;
106
107   // Construct distributed query object
108   Distributed_query *dq = new Distributed_query{final_query_string, stages};
109
110   delete remote_partitions;
111
112   return dq;
113 }
114
115 static bool has_ignore_partitions_arg_for_table(L_Table table,
    ↪  L_parsed_comment_args parsed_args) {
```

```
116   return parsed_args.comment_args_lookup_table[IGNORE_TABLE_PARTITIONS_FLAG] ==
      ↪  table.name;
117 }
118
119
120 static Distributed_query *make_semi_join_distributed_query(L_Parser_info
    ↪  *parser_info, L_parsed_comment_args parsed_args) {
121
122   // ----------------------------------------------------
123   std::vector<L_Table> *tables = &(parser_info->tables);
124   std::string where_clause = parser_info->where_clause;
125   // ----------------------------------------------------
126
127   std::vector<Stage> stages;
128
129   L_Table* stationary_table = NULL;
130   L_Table* remote_table = NULL;
131   std::vector<Partition>* remote_partitions = NULL;
132   bool has_stationary_table = false;
133   unsigned int biggest_partition_count = 0;
134
135   for (auto &table : *tables) {
136
137     std::vector<Partition> *partitions =
138         get_partitions_by_table_name(table.name);
139
140     if (partitions == NULL) {
141       return NULL;
142     }
143
144     // Choose table with one partition, or ignore flag
145     if (!has_stationary_table && (partitions->size() == 1 ||
        ↪  has_ignore_partitions_arg_for_table(table, parsed_args))) {
146       has_stationary_table = true;
147       stationary_table = &table;
148       delete partitions;
149     }
150     else {
151       if (partitions->size() > biggest_partition_count ||
          ↪  biggest_partition_count == 0) {
152         // Choose the most partitioned table, and its partitions
153         remote_partitions = partitions;
154         remote_table = &table;
```

```
155         biggest_partition_count = partitions->size();
156       }
157     }
158   }
159
160   /* static Distributed_query
      ↪  *make_one_sided_semi_join_distributed_query(L_Parser_info *parser_info,
      ↪  L_Table* stationary_table, L_Table* remote_table, std::vector<Partition>*
      ↪  remote_partitions) { */
161   if (has_stationary_table) {
162     // n=1
163     return make_one_sided_semi_join_distributed_query(parser_info,
        ↪  stationary_table, remote_table, remote_partitions);
164
165   }
166   else {
167     // n=2
168     return make_recursive_semi_join_distributed_query(parser_info,
        ↪  remote_table, remote_partitions);
169   }
170 }
171
172 static std::string semi_join_generate_final_join_query_string(L_Table
    ↪  *stationary_table, L_Table *remote_table, std::string join_on) {
173
174   L_Table stat_table = *stationary_table;
175   L_Table rem_table = *remote_table;
176
177   std::string final_query_string = "SELECT ";
178
179   std::vector<std::string>::iterator p = stat_table.projections.begin();
180   std::vector<std::string>::iterator a = stat_table.aliases.begin();
181
182   while (p != stat_table.projections.end()) {
183     final_query_string += stat_table.name + "." + *p;
184     final_query_string += a->length() > 0 ? " as " + *a: "";
185     ++p;
186     ++a;
187     if (p != stat_table.projections.end()) final_query_string += ", ";
188   }
189
190   final_query_string += ", ";
191
```

```
192   join_on.replace(join_on.find(rem_table.name), rem_table.name.length(),
      ↪  rem_table.interim_name);
193
194   p = rem_table.projections.begin();
195   a = rem_table.aliases.begin();
196
197   while (p != rem_table.projections.end()) {
198     final_query_string += rem_table.interim_name + "." + *p;
199     final_query_string += a->length() > 0 ? " as " + *a: "";
200     ++p;
201     ++a;
202     if (p != rem_table.projections.end()) final_query_string += ", ";
203   }
204
205   final_query_string += " FROM " + stat_table.name + " JOIN " +
206                         rem_table.interim_name + " ON " + join_on;
207
208   return final_query_string;
209 }
210
211 #endif  // LUNDGREN_SEMI_JOIN
```

lundgren/join_strategies/bloom_join/bloom_join.h

```
1  #include <string.h>
2  #include <tuple>
3  #include "plugin/lundgren/distributed_query.h"
4  #include "plugin/lundgren/helpers.h"
5  #include "plugin/lundgren/partitions/partition.h"
6  #include "plugin/lundgren/join_strategies/common.h"
7  #include "plugin/lundgren/join_strategies/semi_join.h"
8  #include "plugin/lundgren/constants.h"
9  #include "plugin/lundgren/join_strategies/bloom_join/bloom_join_executor.h"
10 #include "plugin/lundgren/join_strategies/bloom_join/bloom_slave.h"
11
12 #ifndef LUNDGREN_BLOOM_JOIN
13 #define LUNDGREN_BLOOM_JOIN
14
15
```

```
16 static Distributed_query
   ↪ *make_one_sided_bloom_join_distributed_query(L_Parser_info *parser_info
   ↪ MY_ATTRIBUTE((unused)), L_Table* stationary_table, L_Table* remote_table,
   ↪ std::vector<Partition>* remote_partitions);
17 static Distributed_query
   ↪ *make_recursive_bloom_join_distributed_query(L_Parser_info *parser_info
   ↪ MY_ATTRIBUTE((unused)), L_Table* remote_table, std::vector<Partition>
   ↪ *remote_partitions);
18 static bool is_bloom_slave(L_parsed_comment_args parsed_args);
19 static Distributed_query *make_bloom_join_distributed_query(L_Parser_info
   ↪ *parser_info, L_parsed_comment_args parsed_args);
20
21
22 // Bloom join
23
24 // n = 1
25 static Distributed_query
   ↪ *make_one_sided_bloom_join_distributed_query(L_Parser_info *parser_info
   ↪ MY_ATTRIBUTE((unused)), L_Table* stationary_table, L_Table* remote_table,
   ↪ std::vector<Partition>* remote_partitions) {
26
27   //std::vector<L_Table> *tables = &(parser_info->tables);
28   std::string where_clause = parser_info->where_clause;
29
30   std::vector<Stage> stages;
31
32
33   // std::string filtered_remote_interim_name = generate_interim_name();
34   remote_table->interim_name = generate_interim_name();
35
36   std::string stationary_join_column = stationary_table->join_columns[0];
37   std::string remote_join_column = remote_table->join_columns[0];
38
39
40   std::string join_column_projection_query = "SELECT DISTINCT " +
     ↪ stationary_join_column + " FROM " + stationary_table->name;
41
42   std::string bloom_filter_base64;
43   uint64 bf_inserted_count;
44
45   std::tie(bloom_filter_base64, bf_inserted_count) =
     ↪ generate_bloom_filter_from_query(join_column_projection_query);
46
```

```
47
48  // STAGE 2
49  Stage stage2;
50
51  std::string bloom_join_query_string = "/*" PLUGIN_FLAG
    ↪  "<join_strategy=bloom,";
52
53  bloom_join_query_string += BLOOM_SLAVE_FLAG "=true,";
54  bloom_join_query_string += BLOOM_FILTERED_INTERIM_NAME_FLAG "=" +
    ↪  remote_table->interim_name + ",";
55  bloom_join_query_string += BLOOM_FILTER_REMOTE_TABLE_FLAG "=" +
    ↪  remote_table->name + ",";
56  bloom_join_query_string += BLOOM_FILTER_REMOTE_JOIN_COLUMN_FLAG "=" +
    ↪  remote_join_column + ",";
57  bloom_join_query_string += BLOOM_FILTER_MASTER_ID_FLAG "=" +
    ↪  std::to_string(SelfNode::getNode().id) + ",";
58  bloom_join_query_string += BLOOM_FILTER_PARAMETER_COUNT_FLAG "=" +
    ↪  std::to_string(bf_inserted_count) + ",";
59  bloom_join_query_string += BLOOM_FILTER_FLAG "=" + bloom_filter_base64 +
    ↪  ">*/";
60
61  bloom_join_query_string += "SELECT ";
62
63  std::vector<std::string>::iterator p = remote_table->projections.begin();
64
65  while (p != remote_table->projections.end()) {
66    bloom_join_query_string += remote_table->name + "." + *p;
67    ++p;
68    if (p != remote_table->projections.end()) bloom_join_query_string += ", ";
69  }
70  if (remote_table->where_transitive_projections.size() > 0)
71    bloom_join_query_string += ", ";
72  p = remote_table->where_transitive_projections.begin();
73  while (p != remote_table->where_transitive_projections.end()) {
74    bloom_join_query_string += remote_table->name + "." + *p;
75    ++p;
76    if (p != remote_table->where_transitive_projections.end())
77      bloom_join_query_string += ", ";
78  }
79
80  bloom_join_query_string += " FROM " + remote_table->name;
81
```

```
82   //Interim_target stage2_interim_target = {remote_table->interim_name,
     ↪  SelfNode::getNode(), remote_join_column}; // index
83
84   for (auto &p : *remote_partitions) {
85     Partition_query pq = {bloom_join_query_string, p.node}; //,
       ↪  stage2_interim_target};
86     stage2.partition_queries.push_back(pq);
87   }
88
89   stages.push_back(stage2);
90
91   // Construct distributed query object
92   Distributed_query *dq = new
     ↪  Distributed_query{semi_join_generate_final_join_query_string(stationary_table,
     ↪  remote_table, where_clause), stages};
93
94   delete remote_partitions;
95
96   return dq;
97
98   // TODO: remember delete remote_partitions
99 }
100
101 // n > 1
102 static Distributed_query
    ↪  *make_recursive_bloom_join_distributed_query(L_Parser_info *parser_info
    ↪  MY_ATTRIBUTE((unused)), L_Table* remote_table, std::vector<Partition>
    ↪  *remote_partitions) {
103
104   std::vector<Stage> stages;
105
106   Stage stage1;
107
108   std::string join_union_interim_table_name = generate_interim_name();
109
110   //Distributed Partition queries
111   std::string recursive_distributed_join_query_string = "/*" PLUGIN_FLAG
      ↪  "<join_strategy=bloom," IGNORE_TABLE_PARTITIONS_FLAG "=";
112   recursive_distributed_join_query_string += remote_table->name + ">*/";
113
114   recursive_distributed_join_query_string +=
      ↪  generate_join_query_string(parser_info->tables,
      ↪  parser_info->where_clause, false);
```

```
115
116   Interim_target interim_target = {join_union_interim_table_name ,
      ↪  SelfNode::getNode()};
117
118   for (auto &p : *remote_partitions) {
119     Partition_query pq = {recursive_distributed_join_query_string, p.node,
        ↪  interim_target};
120     stage1.partition_queries.push_back(pq);
121   }
122
123   stages.push_back(stage1);
124
125   // Final query
126
127   std::string final_query_string = "SELECT * FROM " +
      ↪  join_union_interim_table_name;
128
129   // Construct distributed query object
130
131   Distributed_query *dq = new Distributed_query{final_query_string, stages};
132
133   delete remote_partitions;
134
135   return dq;
136 }
137
138 static bool is_bloom_slave(L_parsed_comment_args parsed_args) {
139   return parsed_args.comment_args_lookup_table[BLOOM_SLAVE_FLAG] == "true";
140 }
141
142
143 static Distributed_query *make_bloom_join_distributed_query(L_Parser_info
    ↪  *parser_info, L_parsed_comment_args parsed_args) {
144
145   if (!is_bloom_slave(parsed_args)) {
146
147     // ----------------------------------------------------
148     std::vector<L_Table> *tables = &(parser_info->tables);
149     std::string where_clause = parser_info->where_clause;
150     // ----------------------------------------------------
151
152     std::vector<Stage> stages;
153
```

```
154     L_Table* stationary_table = NULL;
155     L_Table* remote_table = NULL;
156     std::vector<Partition>* remote_partitions = NULL;
157     bool has_stationary_table = false;
158     unsigned int biggest_partition_count = 0;
159
160     for (auto &table : *tables) {
161
162       std::vector<Partition> *partitions =
163           get_partitions_by_table_name(table.name);
164
165       if (partitions == NULL) {
166         return NULL;
167       }
168
169       // Choose table with one partition, or ignore flag
170       if (!has_stationary_table && (partitions->size() == 1 ||
          ↪  has_ignore_partitions_arg_for_table(table, parsed_args))) {
171         has_stationary_table = true;
172         stationary_table = &table;
173         delete partitions;
174       }
175       else {
176         if (partitions->size() > biggest_partition_count ||
            ↪  biggest_partition_count == 0) {
177           // Choose the most partitioned table, and its partitions
178           remote_partitions = partitions;
179           remote_table = &table;
180           biggest_partition_count = partitions->size();
181         }
182       }
183     }
184
185     if (has_stationary_table) {
186       // n=1
187       return make_one_sided_bloom_join_distributed_query(parser_info,
          ↪  stationary_table, remote_table, remote_partitions);
188
189     }
190     else {
191       // n=2
192       return make_recursive_bloom_join_distributed_query(parser_info,
          ↪  remote_table, remote_partitions);
```

```
193     }
194   }
195   else {
196     return bloom_slave_execute_strategy(parser_info, parsed_args);
197   }
198 }
199
200 #endif  // LUNDGREN_BLOOM_JOIN
```

lundgren/join_strategies/bloom_join/bloom_join_executor.h

```
1  #include <tuple>
2  #include <mysqlx/xdevapi.h>
3  #include "plugin/lundgren/partitions/node.h"
4  #include "plugin/lundgren/distributed_query_manager.h"
5  #include "plugin/lundgren/join_strategies/bloom_join/bloom_filter.h"
6  #include "plugin/lundgren/join_strategies/bloom_join/bloom_filter_parameters.h"
7  #include "plugin/lundgren/join_strategies/bloom_join/filter_coding.h"
8
9  #ifndef LUNDGREN_BLOOM_EXECUTOR
10 #define LUNDGREN_BLOOM_EXECUTOR
11
12 std::tuple<std::string, uint64> generate_bloom_filter_from_query(std::string
   ↪  query);
13
14 std::tuple<std::string, uint64> generate_bloom_filter_from_query(std::string
   ↪  query) {
15
16
17     std::string con_string = generate_connection_string(SelfNode::getNode());
18
19     mysqlx::Session s(con_string);
20     mysqlx::SqlResult res = s.sql(query).execute();
21
22     const mysqlx::Columns *columns = &res.getColumns();
23     uint64 row_count = res.count();
24
25     //Instantiate Bloom Filter
26     bloom_filter filter(get_bloom_parameters(row_count));
27
28     // Insert into Bloom Filter
29     {
```

```
        mysqlx::Row row;
        while ((row = res.fetchOne())) {

            switch ((*columns)[0].getType()) {
            case mysqlx::Type::INT :
                filter.insert(int(row[0]));
                break;
            case mysqlx::Type::DECIMAL :
                filter.insert(double(row[0]));
                break;
            case mysqlx::Type::DOUBLE :
                filter.insert(double(row[0]));
                break;
            case mysqlx::Type::STRING :
                filter.insert(std::string(row[0]));
                break;
            default:
            break;
            }
        }
    }
    s.close();

    std::vector<unsigned char> bit_table_ = filter.bit_table_;

    std::string ress = encode_bit_table(bit_table_);

    return {ress, row_count};
}

#endif  // LUNDGREN_BLOOM_EXECUTOR
```

lundgren/join_strategies/bloom_join/bloom_slave.h

```
#include <string.h>
#include "plugin/lundgren/distributed_query.h"
#include "plugin/lundgren/distributed_query_manager.h"
#include "plugin/lundgren/constants.h"
#include "plugin/lundgren/helpers.h"
#include "plugin/lundgren/join_strategies/bloom_join/bloom_filter.h"
#include "plugin/lundgren/join_strategies/bloom_join/bloom_filter_parameters.h"
#include "plugin/lundgren/join_strategies/bloom_join/filter_coding.h"
```

```
9
10 #ifndef LUNDGREN_BLOOM_SLAVE
11 #define LUNDGREN_BLOOM_SLAVE
12
13
14 std::string generate_filtered_insert_statement(mysqlx::SqlResult *res,
   ↪  bloom_filter filter, std::string filter_column);
15
16
17 bloom_filter parse_bloom_filter(L_parsed_comment_args parsed_args) {
18
19     std::string bloom_filter_base64 =
       ↪  parsed_args.comment_args_lookup_table[BLOOM_FILTER_FLAG];
20
21     std::vector<unsigned char> bit_table =
       ↪  decode_bit_table(bloom_filter_base64);
22
23     uint64 bf_inserted_count =
       ↪  std::stoi(parsed_args.comment_args_lookup_table[BLOOM_FILTER_PARAMETER_COUNT_FLAG]);
24
25     bloom_filter bf(get_bloom_parameters(bf_inserted_count));
26
27     bf.bit_table_ = bit_table;
28
29     return bf;
30 }
31
32
33 Distributed_query *bloom_slave_execute_strategy(L_Parser_info *parser_info
   ↪  MY_ATTRIBUTE((unused)), L_parsed_comment_args parsed_args) {
34
35     bloom_filter filter = parse_bloom_filter(parsed_args);
36
37     std::string filtered_interim_name =
       ↪  parsed_args.comment_args_lookup_table[BLOOM_FILTERED_INTERIM_NAME_FLAG];
38     std::string join_column =
       ↪  parsed_args.comment_args_lookup_table[BLOOM_FILTER_REMOTE_JOIN_COLUMN_FLAG];
39
40     std::string remote_table_name =
       ↪  parsed_args.comment_args_lookup_table[BLOOM_FILTER_REMOTE_TABLE_FLAG];
41
42     std::string master_node_id =
       ↪  parsed_args.comment_args_lookup_table[BLOOM_FILTER_MASTER_ID_FLAG];
```

```
43
44      std::string query_for_filtering = "SELECT ";
45
46      L_Table remote_table = parser_info->tables[0];
47      std::vector<std::string>::iterator p = remote_table.projections.begin();
48
49      while (p != remote_table.projections.end()) {
50        query_for_filtering += remote_table_name + "." + *p;
51        ++p;
52        if (p != remote_table.projections.end()) query_for_filtering += ", ";
53      }
54
55      query_for_filtering += " FROM " + remote_table_name;
56
57      Node master_node = getNodeById(master_node_id);
58
59      std::string con_string = generate_connection_string(master_node);
60
61      mysqlx::Session s(con_string);
62      mysqlx::SqlResult res = s.sql(query_for_filtering).execute();
63
64      std::string create_table_statement = "CREATE TABLE IF NOT EXISTS " +
        ↪  filtered_interim_name + " ";
65
66      std::string schema = generate_table_schema(&res);
67
68      schema.pop_back();
69      schema += ", INDEX (" + join_column + "))"; // index
70
71      create_table_statement += schema + " " + INTERIM_TABLE_ENGINE ";";
72
73      s.sql(create_table_statement).execute();
74
75      mysqlx::Schema sch = s.getSchema(SelfNode::getNode().database);
76      mysqlx::Table tbl = sch.getTable(filtered_interim_name);
77
78      mysqlx::Row row;
79      const mysqlx::Columns *columns = &res.getColumns();
80      uint num_columns = res.getColumnCount();
81
82      uint filter_column_index = 0;
83
84      for (uint i = 0; i < num_columns; i++) {
```

```
85        if (std::string((*columns)[i].getColumnLabel()) == join_column) {
86            filter_column_index = i;
87            break;
88        }
89      }
90
91      int n = BLOOM_SLAVE_BATCH_SIZE;
92      auto insert = tbl.insert();
93
94      while ((row = res.fetchOne())) {
95        switch ((*columns)[filter_column_index].getType()) {
96          case mysqlx::Type::INT:
97            if (!filter.contains(int(row[filter_column_index]))) continue;
98            break;
99          case mysqlx::Type::DECIMAL:
100           if (!filter.contains(double(row[filter_column_index]))) continue;
101           break;
102         case mysqlx::Type::DOUBLE:
103           if (!filter.contains(double(row[filter_column_index]))) continue;
104           break;
105         case mysqlx::Type::STRING:
106           if (!filter.contains(std::string(row[filter_column_index])))
            ↪  continue;
107           break;
108         default:
109         break;
110       }
111
112       insert.values(row);
113
114       if (n == 0) {
115         insert.execute();
116         insert = tbl.insert();
117         n = BLOOM_SLAVE_BATCH_SIZE;
118       } else {
119         n--;
120       }
121     }
122     if (n != BLOOM_SLAVE_BATCH_SIZE) {
123       // final batch
124       insert.execute();
125     }
126
```

```
127     s.close();
128
129     // rewrite to NO-OP
130     Distributed_query *dq = new Distributed_query{"DO 0;"};
131
132     return dq;
133 }
134
135
136 #endif  // LUNDGREN_BLOOM_SLAVE
```

lundgren/join_strategies/bloom_join/filter_coding.h

```
1  #include "include/base64.h"
2
3  #ifndef LUNDGREN_FILTER_CODING
4  #define LUNDGREN_FILTER_CODING
5
6  std::string encode_bit_table(std::vector<unsigned char> bit_table);
7  std::vector<unsigned char> decode_bit_table(std::string base64);
8
9  std::string encode_bit_table(std::vector<unsigned char> bit_table) {
10     unsigned char* bit_array = bit_table.data();
11
12     uint64 size_of_bit_array = sizeof(unsigned char) * bit_table.size();
13     uint64 needed_length = base64_needed_encoded_length(size_of_bit_array);
14
15     char *base64_dst = new char[needed_length];
16
17     base64_encode(bit_array, size_of_bit_array, base64_dst);
18
19     std::string res(base64_dst, needed_length);
20
21     delete base64_dst;
22     return res;
23 }
24
25 std::vector<unsigned char> decode_bit_table(std::string base64) {
26
27     uint64 needed_length = base64_needed_decoded_length(base64.length());
28
29     unsigned char* bit_array = new unsigned char[needed_length];
```

```

    base64_decode(base64.c_str(), base64.length(), bit_array, NULL,
     ↪  MY_BASE64_DECODE_ALLOW_MULTIPLE_CHUNKS);

    std::vector<unsigned char> bit_table(bit_array, bit_array + needed_length);

    delete bit_array;
    return bit_table;
}


#endif  // LUNDGREN_FILTER_CODING
```

lundgren/join_strategies/bloom_join/bloom_filter_parameters.h

```
#include "plugin/lundgren/join_strategies/bloom_join/bloom_filter.h"

#ifndef LUNDGREN_BLOOM_FILTER_PARAMETERS
#define LUNDGREN_BLOOM_FILTER_PARAMETERS

bloom_parameters get_bloom_parameters(uint64 expected_count);

bloom_parameters get_bloom_parameters(uint64 expected_count) {

    bloom_parameters parameters;
    parameters.projected_element_count = expected_count;
    parameters.false_positive_probability = 0.0001; // 1 in 10000
    parameters.random_seed = 0xA5A5A5A5;

    if (!parameters) {
        std::cout << "Error - Invalid set of bloom filter parameters!" <<
         ↪  std::endl;
        return bloom_parameters();
    }

    parameters.compute_optimal_parameters();

    return parameters;
}


#endif  // LUNDGREN_BLOOM_FILTER_PARAMETERS
```

lundgren/join_strategies/hash_redistribution.h

```
#include "plugin/lundgren/distributed_query.h"
#include "plugin/lundgren/helpers.h"
#include "plugin/lundgren/partitions/partition.h"
#include "plugin/lundgren/distributed_query_manager.h"
#include "plugin/lundgren/partitions/node.h"
#include "plugin/lundgren/constants.h"
#include "plugin/lundgren/partitions/node.h"
#include "plugin/lundgren/join_strategies/common.h"

#ifndef LUNDGREN_HASH_REDIST_JOIN
#define LUNDGREN_HASH_REDIST_JOIN

static Distributed_query *make_hash_redist_join_distributed_query(L_Parser_info *parser_info, L_parsed_comment_args parsed_args, const char *original_query);
static bool is_hash_redist_slave(L_parsed_comment_args parsed_args);
Distributed_query *execute_hash_redist_slave(L_Parser_info *parser_info, L_parsed_comment_args parsed_args);

static bool is_hash_redist_slave(L_parsed_comment_args parsed_args) {
  return parsed_args.comment_args_lookup_table[HASH_REDIST_SLAVE_FLAG] == "true";
}

static Distributed_query *make_hash_redist_join_distributed_query(L_Parser_info *parser_info, L_parsed_comment_args parsed_args, const char *original_query) {

    if (!is_hash_redist_slave(parsed_args)) {

        std::vector<Stage> stages;

        std::vector<L_Table> *tables = &(parser_info->tables);
        std::map<std::string, std::string> interim_table_names; // Maps table names to interim tabel names
        std::map<std::string, std::string> table_join_column; // Maps table names to its join columns
        std::map<std::string, std::vector<std::string>> nodes_and_partitions; // Maps node id to partitions the node contains
        std::map<std::string, Node> node_id_to_node_obj; // Maps node id to actual node (a bit dumdum, should be refactored)
```

```
32
33          for(auto &table : *tables) {
34              interim_table_names[table.name] = generate_interim_name();
35              table.interim_name = interim_table_names[table.name];
36              table_join_column[table.name] = table.join_columns[0];
37
38              /* Maps nodes with tables they hold and linking id to node objects
                ↪  */
39              std::vector<Partition> *partitions =
                ↪  get_partitions_by_table_name(table.name);
40              for (auto &partition : *partitions) {
41
                    ↪  nodes_and_partitions[std::to_string(partition.node.id)].push_back(table.name)
42                  node_id_to_node_obj[std::to_string(partition.node.id)] =
                    ↪  partition.node;
43              }
44              delete partitions;
45          }
46
47          Stage stage1;
48
49          for (auto node : nodes_and_partitions) {
50              std::string pq_sql_statement = "/*" PLUGIN_FLAG;
51              pq_sql_statement += "<join_strategy=hash_redist,";
52              pq_sql_statement += HASH_REDIST_SLAVE_FLAG "=true";
53              pq_sql_statement += ",tables="; // Hack - Format:
                ↪  "[table1:a6fbd:join-col|table2:bcf34:join-col]"
54              for (auto table : node.second) { // Iterates partitions
55                  pq_sql_statement += table + ':' + interim_table_names[table] +
                    ↪  ':' + table_join_column[table] +'|';
56              }
57              pq_sql_statement.pop_back(); // Delete last pipe
58              pq_sql_statement += "]>*/";
59              std::string original_query_stripped = std::string(original_query);
60              std::string comment_end = "*/";
61              pq_sql_statement +=
                ↪  original_query_stripped.substr(original_query_stripped.find(comment_end)
                ↪  + std::string(comment_end).length());
62
                ↪  stage1.partition_queries.push_back(Partition_query{pq_sql_statement,node_id_to_nc
63          }
64
65          Stage stage2;
```

```
66
67          std::string result_interim_table = generate_interim_name();
68          Interim_target it = {result_interim_table, {SelfNode::getNode()}};
69          std::string pq_sql_statement =
            ↪  generate_join_query_string(parser_info->tables,
            ↪  parser_info->where_clause, true);
70          for (auto node : nodes_and_partitions) {
71
                ↪  stage2.partition_queries.push_back(Partition_query{pq_sql_statement,
                ↪  node_id_to_node_obj[node.first], it});
72          }
73
74          std::string final_query = "SELECT * FROM " + result_interim_table; //+
            ↪  ';';
75          Distributed_query *dq = new Distributed_query{final_query,
            ↪  {stage1,stage2}};
76          return dq;
77      }
78      else
79      {
80          return execute_hash_redist_slave(parser_info, parsed_args);
81      }
82
83  }
84
85  Distributed_query *execute_hash_redist_slave(L_Parser_info *parser_info,
    ↪  L_parsed_comment_args parsed_args) {
86
87      Stage stage;
88
89      std::string local_tables = parsed_args.comment_args_lookup_table["tables"];
90      local_tables = string_remove_ends(local_tables);
91      std::vector<std::string> parsed_local_tables = split(local_tables, '|');
92      std::map<std::string, std::string> table_to_projection;
93      std::map<std::string, Node> nodes_involved;
94
95      for (auto &table : parser_info->tables) {
96          table_to_projection[table.name] =
            ↪  generate_projections_string_for_partition_query(&table);
97          std::vector<Partition> *partitions =
            ↪  get_partitions_by_table_name(table.name);
98          for (auto &partition : *partitions) {
99              nodes_involved[std::to_string(partition.node.id)] = partition.node;
```

```
100         }
101         delete partitions;
102     }
103 
104 
105     for (auto table : parsed_local_tables) {
106         std::string table_name = table.substr(0,table.find(':'));
107         std::string interim_table_name = table.substr(table.find(':')+1,
            ↪  table.rfind(':')-table.find(':')-1);
108         std::string table_join_column = table.substr(table.rfind(':')+1);
109 
110         std::string pq_sql_statement = "SELECT " +
            ↪  table_to_projection[table_name] + " FROM " + table_name +
111             " WHERE conv(substr(MD5(" + table_join_column + "), 1, 6), 16,
                ↪  10)%" + std::to_string(nodes_involved.size()) + '=';
112 
113         for (auto node : nodes_involved) {
114             Interim_target it = {interim_table_name, node.second,
                ↪  table_join_column};
115             Partition_query pq = {pq_sql_statement +
                ↪  std::to_string(node.second.id), SelfNode::getNode(), it};
116             stage.partition_queries.push_back(pq);
117         }
118     }
119 
120     Distributed_query *dq = new Distributed_query{"DO 0;", {stage}};
121     return dq;
122 }
123 
124 #endif  // LUNDGREN_HASH_REDIST_JOIN
```

lundgren/join_strategies/sort_merge/sort_merge.h

```
1 #include <mysqlx/xdevapi.h>
2 #include "plugin/lundgren/distributed_query.h"
3 #include "plugin/lundgren/join_strategies/common.h"
4 #include "plugin/lundgren/helpers.h"
5 #include "plugin/lundgren/constants.h"
6 #include "plugin/lundgren/partitions/node.h"
7 #include "plugin/lundgren/join_strategies/sort_merge/k_way_merge_joiner.h"
8 
9 #ifndef LUNDGREN_SORT_MERGE
```

```
10 #define LUNDGREN_SORT_MERGE
11
12 std::string generate_order_by_query(L_Table* table, std::string join_column);
13
14 std::string generate_joint_table_schema(mysqlx::SqlResult *lhs_res,
   ↪ mysqlx::SqlResult *rhs_res);
15 std::string generate_joint_insert_rows_statement(std::vector<mysqlx::Row>
   ↪ lhs_rows, std::vector<mysqlx::Row> rhs_rows, mysqlx::SqlResult *lhs_res,
   ↪ mysqlx::SqlResult *rhs_res);
16
17 std::string generate_projections_string_for_final_query(L_Table* table,
   ↪ std::string interim_name);
18
19
20 Distributed_query *execute_sort_merge_distributed_query(L_Parser_info
   ↪ *parser_info) {
21
22     std::string merge_joined_interim_name = generate_interim_name();
23
24     std::vector<L_Table> *tables = &(parser_info->tables);
25
26     L_Table lhs_table = tables->at(0);
27     L_Table rhs_table = tables->at(1);
28
29     std::vector<Partition>* lhs_partitions =
       ↪ get_partitions_by_table_name(lhs_table.name);
30     std::vector<Partition>* rhs_partitions =
       ↪ get_partitions_by_table_name(rhs_table.name);
31
32     std::string lhs_join_column = lhs_table.join_columns[0];
33     std::string rhs_join_column = rhs_table.join_columns[0];
34
35     std::string lhs_order_query = generate_order_by_query(&lhs_table,
       ↪ lhs_join_column);
36     std::string rhs_order_query = generate_order_by_query(&rhs_table,
       ↪ rhs_join_column);
37
38     std::vector<mysqlx::Session*> sessions;
39
40     std::vector<mysqlx::SqlResult*> lhs_streams;
41     mysqlx::SqlResult* lhs_res = new mysqlx::SqlResult[rhs_partitions->size()];
42     int z = 0;
43
```

```
44      for (auto &p : *lhs_partitions) {
45
46          std::string con_string = generate_connection_string(p.node);
47          mysqlx::Session* s = new mysqlx::Session(con_string);
48
49          lhs_res[z] = s->sql(lhs_order_query).execute();
50
51          lhs_streams.push_back(&lhs_res[z]);
52          sessions.push_back(s);
53          z++;
54      }
55
56      std::vector<mysqlx::SqlResult*> rhs_streams;
57      mysqlx::SqlResult* rhs_res = new mysqlx::SqlResult[rhs_partitions->size()];
58      z = 0;
59
60      for (auto &p : *rhs_partitions) {
61
62          std::string con_string = generate_connection_string(p.node);
63          mysqlx::Session* s = new mysqlx::Session(con_string);
64
65          rhs_res[z] = s->sql(rhs_order_query).execute();
66
67          rhs_streams.push_back(&rhs_res[z]);
68          sessions.push_back(s);
69          z++;
70      }
71
72      //-----------------------------------------------------
73
74      // FIND LHS JOIN COLUMN INDEX
75      uint lhs_join_column_index = 0;
76      const mysqlx::Columns *lhs_columns = &lhs_streams.at(0)->getColumns();
77      uint lhs_num_columns = lhs_streams.at(0)->getColumnCount();
78
79      for (uint i = 0; i < lhs_num_columns; i++) {
80          if (std::string((*lhs_columns)[i].getColumnLabel()) == lhs_join_column)
            ↪ {
81              lhs_join_column_index = i;
82              break;
83          }
84      }
85
```

```
86      // FIND RHS JOIN COLUMN INDEX
87      uint rhs_join_column_index = 0;
88      const mysqlx::Columns *rhs_columns = &rhs_streams.at(0)->getColumns();
89      uint rhs_num_columns = rhs_streams.at(0)->getColumnCount();
90
91      for (uint i = 0; i < rhs_num_columns; i++) {
92          if (std::string((*rhs_columns)[i].getColumnLabel()) == rhs_join_column)
            ↪ {
93              rhs_join_column_index = i;
94              break;
95          }
96      }
97
98      //-----------------------------------------------------
99
100     // Merging and inserting into interim
101     mysqlx::Session
        ↪ interim_session(generate_connection_string(SelfNode::getNode()));
102
103     std::string create_interim_table_sql = "CREATE TABLE IF NOT EXISTS " +
        ↪ merge_joined_interim_name + " ";
104     create_interim_table_sql += generate_joint_table_schema(lhs_streams.at(0),
        ↪ rhs_streams.at(0));
105
106     create_interim_table_sql += std::string(" ") + INTERIM_TABLE_ENGINE ";";
107
108     interim_session.sql(create_interim_table_sql).execute();
109
110     mysqlx::Schema schema =
        ↪ interim_session.getSchema(SelfNode::getNode().database);
111     mysqlx::Table table = schema.getTable(merge_joined_interim_name);
112
113     K_way_merge_joiner merge_joiner = K_way_merge_joiner(lhs_streams,
        ↪ rhs_streams, lhs_join_column_index, rhs_join_column_index);
114
115
116     std::string insert_into_interim_table_start = "INSERT INTO " +
        ↪ merge_joined_interim_name + " VALUES ";
117
118     auto insert = table.insert();
119
120     int batch_counter = SORT_MERGE_BATCH_SIZE;
121
```

```
122     bool cont = true;
123     while(cont) {
124
125         std::vector<mysqlx::Row>* lhs_matches;
126         std::vector<mysqlx::Row>* rhs_matches;
127         std::tie(lhs_matches, rhs_matches) = merge_joiner.fetchNextMatches();
128
129         if (lhs_matches->size() > 0 && rhs_matches->size() > 0) {
130
131           for (uint i = 0; i < lhs_matches->size(); ++i) {
132             for (uint z = 0; z < rhs_matches->size(); ++z) {
133
134               mysqlx::Row merged_row;
135
136               for (uint lhs_c = 0; lhs_c < lhs_num_columns; lhs_c++) {
137                 merged_row.set(lhs_c, (*lhs_matches)[i][lhs_c]);
138               }
139               for (uint rhs_c = 0; rhs_c < rhs_num_columns; rhs_c++) {
140                 merged_row.set(lhs_num_columns + rhs_c,
                    ↪  (*rhs_matches)[z][rhs_c]);
141               }
142
143               insert.values(merged_row);
144               batch_counter--;
145
146               if (batch_counter <= 0) {
147                 insert.execute();
148                 insert = table.insert();
149                 batch_counter = SORT_MERGE_BATCH_SIZE;
150               }
151             }
152           }
153
154         } else {
155           cont = false;
156         }
157     }
158
159     // insert rows that didnt fit into a batch
160     if (batch_counter != SORT_MERGE_BATCH_SIZE) {
161       insert.execute();
162     }
163
```

```
164     interim_session.close();
165
166     //--------------------------
167     // Cleanup
168     for (auto &s : sessions) {
169       s->close();
170       delete s;
171     }
172
173     delete[] lhs_res;
174     delete[] rhs_res;
175
176     //--------------------------
177
178     std::string final_query_string = "SELECT "
179         + generate_projections_string_for_final_query(&lhs_table,
            ↪ merge_joined_interim_name)
180         + ((lhs_table.projections.size() > 0 && rhs_table.projections.size() >
            ↪ 0 ) ? ", " : "")
181         + generate_projections_string_for_final_query(&rhs_table,
            ↪ merge_joined_interim_name)
182         + " FROM " + merge_joined_interim_name;
183
184     Distributed_query *dq = new Distributed_query();
185     dq->rewritten_query = final_query_string;
186     return dq;
187 }
188
189
190 std::string generate_order_by_query(L_Table* table, std::string join_column) {
191
192     std::string order_query = "SELECT ";
193     order_query += generate_projections_string_for_partition_query(table);
194     order_query += " FROM " + table->name;
195     order_query += " ORDER BY " + table->name + "." + join_column + " ASC";
196     return order_query;
197 }
198
199 std::string write_data_type_column(const mysqlx::Column& col) {
200   std::string return_string = "";
201   return_string += col.getColumnLabel();
202   return_string += " ";
203     switch (col.getType()) {
```

```
204       case mysqlx::Type::BIGINT :
205         return_string += (col.isNumberSigned()) ? "BIGINT" : "BIGINT UNSIGNED";
            ↪ break;
206       case mysqlx::Type::INT :
207         return_string += (col.isNumberSigned()) ? "INT" : "INT UNSIGNED";
            ↪ break;
208       case mysqlx::Type::DECIMAL :
209         return_string += (col.isNumberSigned()) ? "DECIMAL" : "DECIMAL
            ↪ UNSIGNED"; break;
210       case mysqlx::Type::DOUBLE :
211         return_string += (col.isNumberSigned()) ? "DOUBLE" : "DOUBLE UNSIGNED";
            ↪ break;
212       case mysqlx::Type::STRING :
213         return_string += "VARCHAR(" + get_column_length(col.getLength()) + ")";
            ↪ break;
214       default: break;
215     }
216
217     return return_string;
218 }
219
220 std::string generate_joint_table_schema(mysqlx::SqlResult *lhs_res,
    ↪ mysqlx::SqlResult *rhs_res) {
221   std::string return_string = "(";
222   uint lhs_column_count = lhs_res->getColumnCount();
223   uint rhs_column_count = rhs_res->getColumnCount();
224
225   for (uint i = 0; i < lhs_column_count; i++) {
226     const mysqlx::Column& col = lhs_res->getColumn(i);
227     return_string += write_data_type_column(col);
228     return_string += ",";
229   }
230   if (rhs_column_count == 0 && !lhs_column_count == 0) {
231     return_string.pop_back();
232   }
233
234   for (uint i = 0; i < rhs_column_count; i++) {
235     const mysqlx::Column& col = rhs_res->getColumn(i);
236     return_string += write_data_type_column(col);
237     return_string += ",";
238   }
239
240   if (!rhs_column_count == 0) {
```

```
241     return_string.pop_back();
242   }
243   return return_string + ")";
244 }
245
246 std::string generate_projections_string_for_final_query(L_Table* table,
    ↪  std::string interim_name) {
247
248   std::string proj_string = "";
249
250   std::vector<std::string>::iterator p = table->projections.begin();
251
252   while (p != table->projections.end()) {
253     proj_string += interim_name + "." + *p;
254     ++p;
255     if (p != table->projections.end()) proj_string += ", ";
256   }
257
258   return proj_string;
259 }
260
261 #endif  // LUNDGREN_SORT_MERGE
```

lundgren/join_strategies/sort_merge/k_way_merge_joiner.h

```
1 #include <mysqlx/xdevapi.h>
2 #include <tuple>
3 #include <algorithm>
4
5 #ifndef LUNDGREN_K_WAY
6 #define LUNDGREN_K_WAY
7
8
9 class K_way_node {
10 private:
11     mysqlx::SqlResult* stream;
12     mysqlx::Row current_row;
13
14 public:
15     K_way_node(mysqlx::SqlResult* s) {
16         stream = s;
17         current_row = stream->fetchOne();
```

```
18      }
19
20      mysqlx::Row& peek() {
21          return current_row;
22      }
23
24      bool is_empty() {
25          return !current_row.operator bool();
26      }
27
28      mysqlx::Row next() {
29          mysqlx::Row popped_row = current_row;
30          current_row = stream->fetchOne();
31          return popped_row;
32      }
33  };
34
35  class Bin_heap {
36
37  private:
38      int column_index;
39      std::vector<K_way_node> nodes;
40
41  private:
42      static int left(int i) {
43          return 2*i;
44      }
45
46      static int right(int i) {
47          return 2*i+1;
48      }
49
50      int at(int i) {
51          return (int) nodes[i - 1].peek()[column_index];
52      }
53
54      bool node_empty(int i) {
55          return nodes[i - 1].is_empty();
56      }
57
58      void swap(int t, int f) {
59          std::iter_swap(nodes.begin() + t-1, nodes.begin() + f-1);
60      }
```

```
61
62      void heapify() {
63          int heap_size = nodes.size();
64
65          for (int i = heap_size / 2; i > 0; --i) {
66              sift_down(i);
67          }
68      }
69
70      void sift_down(int i) {
71          int heap_size = nodes.size();
72          int l = left(i);
73          int r = right(i);
74
75          int smallest;
76
77          // Don't like the fact that empty nodes propogate all the way down..
            ↪ but it only happens for the last call to sift_down ^--^
78          if (l <= heap_size && (node_empty(i) || (!node_empty(l) && at(l) <
            ↪ at(i)))) {
79              smallest = l;
80          } else {
81              smallest = i;
82          }
83
84          if (r <= heap_size && (node_empty(smallest) || (!node_empty(r) && at(r)
            ↪ < at(smallest)))) {
85              smallest = r;
86          }
87
88          if (smallest != i) {
89              swap(i, smallest);
90              sift_down(smallest);
91          }
92      }
93
94  public:
95      Bin_heap() {}
96      Bin_heap(std::vector<mysqlx::SqlResult*> streams, int c_index) {
97          column_index = c_index;
98          for (auto &s : streams) {
99              nodes.push_back(K_way_node(s));
100         }
```

```
101         heapify();
102     }
103 
104     bool has_next() {
105         return !nodes[0].is_empty();
106     }
107 
108     mysqlx::Row pop() {
109         mysqlx::Row tmp = nodes[0].next();
110         sift_down(1);
111         return tmp;
112     }
113 
114     mysqlx::Row& peek() {
115         return nodes[0].peek();
116     }
117 };
118 
119 class K_way_merge_joiner {
120 
121 private:
122     Bin_heap lhs_heap;
123     Bin_heap rhs_heap;
124 
125     std::vector<mysqlx::Row>* lhs_buffer;
126     std::vector<mysqlx::Row>* rhs_buffer;
127 
128     int lhs_column_index;
129     int rhs_column_index;
130 
131 public:
132     K_way_merge_joiner(std::vector<mysqlx::SqlResult*> lhs,
        ↪ std::vector<mysqlx::SqlResult*> rhs, int lhs_column_index, int
        ↪ rhs_column_index) {
133 
134         lhs_heap = Bin_heap(lhs, lhs_column_index);
135         rhs_heap = Bin_heap(rhs, rhs_column_index);
136 
137         this->lhs_column_index = lhs_column_index;
138         this->rhs_column_index = rhs_column_index;
139 
140         lhs_buffer = new std::vector<mysqlx::Row>;
141         rhs_buffer = new std::vector<mysqlx::Row>;
```

```
142
143         lhs_buffer->reserve(600000);
144         rhs_buffer->reserve(600000);
145     }
146
147     ~K_way_merge_joiner() {
148         delete lhs_buffer;
149         delete rhs_buffer;
150     }
151
152     void buffer_next_value_candidates() {
153
154         while (lhs_heap.has_next() && rhs_heap.has_next() &&
            ↪  (lhs_buffer->size() == 0 || rhs_buffer->size() == 0)) {
155
156             int current_value;
157
158             // empty buffers
159             lhs_buffer->clear();
160             rhs_buffer->clear();
161
162             // Select current value
163             if (((int)lhs_heap.peek()[lhs_column_index]) >=
                ↪  ((int)rhs_heap.peek()[rhs_column_index])) {
164
165                 current_value = lhs_heap.peek()[lhs_column_index];
166
167                 // skip ahead until we find rows that are equal or higher
168                 while(rhs_heap.has_next() &&
                    ↪  ((int)rhs_heap.peek()[rhs_column_index]) < current_value) {
169                     rhs_heap.pop();
170                 }
171
172             } else {
173                 current_value = rhs_heap.peek()[rhs_column_index];
174
175                 // skip ahead until we find rows that are equal or higher
176                 while(lhs_heap.has_next() &&
                    ↪  ((int)lhs_heap.peek()[lhs_column_index]) < current_value) {
177                     lhs_heap.pop();
178                 }
179             }
180
```

```
181             // Buffer all values that are equal to the current value
182             while (lhs_heap.has_next() &&
                ↪  ((int)lhs_heap.peek()[lhs_column_index]) == current_value) {
183                 lhs_buffer->emplace_back(lhs_heap.pop());
184             }
185
186             while (rhs_heap.has_next() &&
                ↪  ((int)rhs_heap.peek()[rhs_column_index]) == current_value) {
187                 rhs_buffer->emplace_back(rhs_heap.pop());
188             }
189         }
190     }
191
192     std::tuple<std::vector<mysqlx::Row>*, std::vector<mysqlx::Row>*>
        ↪  fetchNextMatches() {
193
194         // empty buffers
195         lhs_buffer->clear();
196         rhs_buffer->clear();
197
198         buffer_next_value_candidates();
199         return std::make_tuple(lhs_buffer, rhs_buffer);
200     }
201 };
202
203 #endif  // LUNDGREN_K_WAY
```

lundgren/join_strategies/common.h

```
1 #include <string.h>
2 #include "plugin/lundgren/distributed_query.h"
3
4 #ifndef LUNDGREN_COMMON
5 #define LUNDGREN_COMMON
6
7 static std::string generate_join_query_string(std::vector<L_Table> tables,
    ↪  std::string join_on, bool interim);
8 static std::string generate_final_join_query_string(std::vector<L_Table>
    ↪  tables, std::string join_on);
9 std::string generate_projections_string_for_partition_query(L_Table* table);
10
```

```
11 static std::string generate_join_query_string(std::vector<L_Table> tables,
   ↪  std::string join_on, bool interim) {
12
13   std::string final_query_string = "SELECT ";
14
15   // iterate in reverse, because we get the tables in reverse order from mysql
16   for (auto it = tables.rbegin(); it != tables.rend();) {
17     L_Table table = *it;
18
19     // make final query join clause by replacing table names with interim
20     // names in where_clause Warning! this only replaces the first occurence!
21     join_on.replace(join_on.find(table.name), table.name.length(),
22                     (interim ? table.interim_name : table.name));
23
24     std::vector<std::string>::iterator p = table.projections.begin();
25       std::vector<std::string>::iterator a = table.aliases.begin();
26
27     while (p != table.projections.end()) {
28       final_query_string += (interim ? table.interim_name : table.name) + "." +
          ↪  *p;
29       final_query_string += a->length() > 0 ? " as " + *a: "";
30       ++p;
31       ++a;
32       if (p != table.projections.end()) final_query_string += ", ";
33     }
34
35     if (++it != tables.rend()) final_query_string += ", ";
36   }
37
38   final_query_string += " FROM " + (interim ? tables[1].interim_name :
     ↪  tables[1].name) + " JOIN " +
39                         (interim ? tables[0].interim_name : tables[0].name) + "
                           ↪  ON " + join_on;
40
41
42   return final_query_string;
43 }
44
45 static std::string generate_final_join_query_string(std::vector<L_Table>
   ↪  tables, std::string join_on) {
46   return generate_join_query_string(tables, join_on, true);
47 }
48
```

```
49 std::string generate_projections_string_for_partition_query(L_Table* table) {
50
51   std::string proj_string = "";
52
53   std::vector<std::string>::iterator p = table->projections.begin();
54
55   while (p != table->projections.end()) {
56     proj_string += table->name + "." + *p;
57     ++p;
58     if (p != table->projections.end()) proj_string += ", ";
59   }
60   if (table->where_transitive_projections.size() > 0)
61     proj_string += ", ";
62   p = table->where_transitive_projections.begin();
63   while (p != table->where_transitive_projections.end()) {
64     proj_string += table->name + "." + *p;
65     ++p;
66     if (p != table->where_transitive_projections.end())
67       proj_string += ", ";
68   }
69
70   return proj_string;
71 }
72
73 #endif  // LUNDGREN_COMMON
```

`lundgren/partitions/partition.h`

```
1 #include <string.h>
2 #include "plugin/lundgren/partitions/node.h"
3 #include "plugin/lundgren/partitions/shard_key.h"
4 #include "plugin/lundgren/internal_query/internal_query_session.h"
5 #include "plugin/lundgren/internal_query/sql_resultset.h"
6 #include "plugin/lundgren/constants.h"
7
8 #ifndef LUNDGREN_PARTITION
9 #define LUNDGREN_PARTITION
10
11 struct Partition
12 {
13     std::string table_name;
14     Node node;
```

```
15      Shard_key shard_key;
16      Partition(char *table_name_in, Node node_in, Shard_key shard_key_in) :
        ↪ node(node_in), shard_key(shard_key_in) {
17          table_name = std::string(table_name_in);
18      }
19  };
20
21  static std::vector<Partition>* get_partitions_by_table_name(std::string
    ↪ table_name MY_ATTRIBUTE((unused))) {
22      Internal_query_session *session = new Internal_query_session();
23
24      session->execute_resultless_query("USE test");
25
26      std::string partition_query =
27          "SELECT * FROM lundgren_partition p\n"
28          "INNER JOIN lundgren_node n on p.nodeId = n.id\n"
29          "INNER JOIN lundgren_shard_key s on p.shardKeyId = s.id\n"
30          "WHERE p.table_name = \"" + table_name + "\";";
31
32      Sql_resultset *result = session->execute_query(partition_query.c_str());
33
34      std::vector<Partition> *partitions = new std::vector<Partition>;
35
36      if (result->get_rows() == 0) {
37          return NULL;
38      }
39
40      do {
41
42          Node n = Node(result->getString(5), (uint)result->getLong(6),
            ↪ result->getString(7), result->getString(8), result->getLong(4));
43          Shard_key s = Shard_key(result->getString(11),
            ↪ (uint)result->getLong(12), (uint)result->getLong(13));
44
45          Partition p = Partition(result->getString(3), n, s);
46
47          partitions->push_back(p);
48      } while (result->next());
49
50      delete session;
51
52      return partitions;
53  }
```

```

#endif  // LUNDGREN_PARTITION

/*
CREATE TABLE lundgren_node (
  id INT UNSIGNED PRIMARY KEY,
  host_l VARCHAR(80) NOT NULL,
  port_l INT UNSIGNED,
  database_l VARCHAR(80) NOT NULL,
  username_l VARCHAR(50),
  password_l VARCHAR(50)
);

CREATE TABLE lundgren_shard_key (
  id INT UNSIGNED PRIMARY KEY,
  column_name VARCHAR(80) NOT NULL,
  range_start INT UNSIGNED,
  range_end INT UNSIGNED
);

CREATE TABLE lundgren_partition (
  id INT UNSIGNED PRIMARY KEY,
  nodeId INT UNSIGNED,
  shardKeyId INT UNSIGNED,
  table_name VARCHAR(80) NOT NULL,
  FOREIGN KEY (nodeId) REFERENCES lundgren_node(id),
  FOREIGN KEY (shardKeyId) REFERENCES lundgren_shard_key(id)
);

INSERT INTO lundgren_node VALUES (1, "127.0.0.1", 13000, "test", "root", NULL);
INSERT INTO lundgren_shard_key VALUES (1, "height", 0, 165);
INSERT INTO lundgren_partition VALUES (1, 1, 1, "Person");

SELECT * FROM lundgren_partition p
INNER JOIN lundgren_node n on p.nodeId = n.id
INNER JOIN lundgren_shard_key s on p.shardKeyId = s.id;
*/
```

`lundgren/partitions/node.h`

```
#include <string.h>
#include "plugin/lundgren/internal_query/internal_query_session.h"
```

```
3  #include "plugin/lundgren/internal_query/sql_resultset.h"
4
5  #ifndef LUNDGREN_NODE
6  #define LUNDGREN_NODE
7
8  struct Node {
9    std::string host;
10   uint port;
11   std::string database;
12   std::string user;
13   bool is_self = false;
14   int id;
15
16   Node(bool is_self_in) : is_self(is_self_in) {
17     host = "127.0.0.1";
18     port = 13010;
19     database = "test";
20     user = "root";
21     id = 0;
22   }
23
24   Node(char *host_in, uint port_in, char *database_in, char *user_in, int
     ↪  id_in)
25       : port(port_in), id(id_in) {
26     host = std::string(host_in);
27     database = std::string(database_in);
28     user = std::string(user_in);
29   }
30
31   Node() {}
32 };
33
34 Node getNodeById(std::string node_id) {
35
36   Internal_query_session *session = new Internal_query_session();
37
38   session->execute_resultless_query("USE test");
39
40   std::string node_query = "SELECT * FROM lundgren_node WHERE lundgren_node.id
     ↪  = " + node_id;
41
42   Sql_resultset *result = session->execute_query(node_query.c_str());
43
```

```
44   if (result->get_rows() == 0) {
45       return NULL;
46   }
47
48   Node node;
49
50   do {
51       node = Node(result->getString(1), (uint)result->getLong(2),
         ↪  result->getString(3), result->getString(4), result->getLong(0));
52   } while (result->next());
53
54
55   delete session;
56
57   return node;
58 }
59
60 class SelfNode {
61   static SelfNode *instance;
62   Node internal_node = Node(true);
63
64   SelfNode(Node n) {
65     internal_node = n;
66   }
67
68   // SET @@global.node_id = 0|1|2|3...; in startup scripts
69
70  public:
71   static Node getNode() {
72     if (!instance) {
73       Internal_query_session *session = new Internal_query_session();
74
75       session->execute_resultless_query("USE test");
76
77       // sjekk id
78       std::string id_number_query = "SELECT node_id FROM
         ↪  lundgren_self_node_id";
79       Sql_resultset *result = session->execute_query(id_number_query.c_str());
80       // if (result->get_rows() == 0) {
81       //   return Node(true);
82       // }
83       // sett portnummer
84       //uint port_number = (uint)result->getLong(0) + 10;
```

```
85        std::string node_query = "SELECT * FROM lundgren_node WHERE
          ↪  lundgren_node.id = " + std::to_string(result->getLong(0));
86        Sql_resultset *node_result = session->execute_query(node_query.c_str());
87        if (node_result->get_rows() == 0) {
88            return Node(true);
89        }
90        Node node;
91        do {
92            // node = Node(node_result->getString(1),
              ↪  (uint)node_result->getLong(2), node_result->getString(3),
              ↪  node_result->getString(4), node_result->getLong(0));
93            char* local = (char*)"localhost";
94            node = Node(local, (uint)node_result->getLong(2),
              ↪  node_result->getString(3), node_result->getString(4),
              ↪  node_result->getLong(0));
95        } while (node_result->next());
96
97        instance = new SelfNode(node);
98
99        delete session;
100     }
101     return instance->internal_node;
102   }
103 };
104
105 SelfNode *SelfNode::instance = 0;
106
107 #endif  // LUNDGREN_NODE
```

lundgren/partitions/shard_key.h

```
1  #include <string.h>
2
3  #ifndef LUNDGREN_SHARD_KEY
4  #define LUNDGREN_SHARD_KEY
5
6  struct Shard_key
7  {
8      std::string column_name;
9      uint range_start;
10     uint range_end;
11
```

```
12      Shard_key(char * column_name_in, uint range_start_in, uint range_end_in)
        ↪  :range_start(range_start_in), range_end(range_end_in) {
13          column_name = std::string(column_name_in);
14      }
15  };
16
17  #endif  // LUNDGREN_SHARD_KEY
```

lundgren/query_acceptance.h

```
1   #include <string.h>
2   #include <mysql/service_parser.h>
3   #include "plugin/lundgren/constants.h"
4
5   #ifndef LUNDGREN_QUERY_ACCEPTANCE
6   #define LUNDGREN_QUERY_ACCEPTANCE
7
8   static bool should_query_be_distributed(const char *query) {
9
10      const std::string plugin_flag(PLUGIN_FLAG);
11      const std::string query_string(query);
12      return (query_string.find(plugin_flag) != std::string::npos);
13  }
14
15  static bool accept_query(MYSQL_THD thd, const char *query) {
16
17      if (!should_query_be_distributed(query)) {
18          return false;
19      }
20
21      int type = mysql_parser_get_statement_type(thd);
22
23      return (type == STATEMENT_TYPE_SELECT);
24  }
25
26  static bool detect_join(const char *query) {
27    std::string join_keyword = "JOIN";
28    std::string join_keyword_lower = "join";
29
30    std::string query_str(query);
31
```

```
  return (query_str.find(join_keyword) != std::string::npos ||
   ↪  query_str.find(join_keyword_lower) != std::string::npos);
}
#endif  // LUNDGREN_QUERY_ACCEPTANCE
```

`lundgren/helpers.h`

```
#include <string.h>
#include <vector>
#include <sstream>
#include <iostream>
#include <boost/uuid/uuid.hpp>            // uuid class
#include <boost/uuid/uuid_generators.hpp> // generators
#include <boost/uuid/uuid_io.hpp>         // streaming operators etc.
#include <boost/algorithm/string.hpp>

#ifndef LUNDGREN_HELPERS
#define LUNDGREN_HELPERS

struct L_parsed_comment_args;
std::string generate_interim_name();
std::vector<std::string> split(std::string strToSplit, char delimeter);
L_parsed_comment_args parse_query_comments(const char *query);
std::string string_remove_ends(std::string input_string);

std::string generate_interim_name() {

  boost::uuids::random_generator generator;
  boost::uuids::uuid uuid1 = generator();
  std::string uuid_string = boost::uuids::to_string(uuid1);

  boost::erase_all(uuid_string, "-");

  return "interim_" + uuid_string; // Must prefix with a letter as numbers are
   ↪  not allowed.
}

// Split string by delimiter:
 ↪  https://thispointer.com/how-to-split-a-string-in-c/
std::vector<std::string> split(std::string strToSplit, char delimeter)
{
    std::stringstream ss(strToSplit);
```

```
34      std::string item;
35      std::vector<std::string> splittedStrings;
36      while (std::getline(ss, item, delimeter))
37      {
38          splittedStrings.push_back(item);
39      }
40      return splittedStrings;
41  }
42
43  enum JOIN_STRATEGY {DATA_TO_QUERY, SEMI, BLOOM, SORT_MERGE, HASH_REDIST};
44  const std::map<std::string, JOIN_STRATEGY> join_strategy_string_to_enum = {
45      {"data_to_query", DATA_TO_QUERY},
46      {"semi", SEMI},
47      {"bloom", BLOOM},
48      {"sort_merge", SORT_MERGE},
49      {"hash_redist", HASH_REDIST}
50    };
51
52  struct L_parsed_comment_args {
53    JOIN_STRATEGY join_strategy;
54    std::map<std::string, std::string> comment_args_lookup_table;
55  };
56
57  // Parses the parameters in the passed comment and creates a lookup table
58  L_parsed_comment_args parse_query_comments(const char *query) {
59    char delimiter;
60    std::string query_string = std::string(query);
61    int pos_start = query_string.find("<")+1;
62    int pos_end = query_string.find(">") - pos_start;
63    query_string = query_string.substr(pos_start, pos_end);
64
65    delimiter = ',';
66    std::vector<std::string> comment_parameters = split(query_string, delimiter);
67
68    delimiter = '=';
69    L_parsed_comment_args parsed_args;
70    int i;
71    for (i = 0; i < int(comment_parameters.size()); i++){
72      if (comment_parameters[i].find("join_strategy") != std::string::npos) {
73        int pos_delimiter = comment_parameters[i].find(delimiter);
74        std::string js = comment_parameters[i].substr(pos_delimiter+1);
75        parsed_args.join_strategy = join_strategy_string_to_enum.at(js);
76        break;
```

```
77      }
78    }
79    comment_parameters.erase(comment_parameters.begin() + i);
80
81    delimiter = '=';
82    std::map<std::string, std::string> comment_parameter_lookup_table;
83    for (auto const parameter : comment_parameters) {
84      int pos_delimiter = parameter.find(delimiter);
85      parsed_args.comment_args_lookup_table[parameter.substr(0,pos_delimiter)] =
        ↪  parameter.substr(pos_delimiter+1);
86
87    }
88
89    return parsed_args;
90  }
91
92  std::string string_remove_ends(std::string input_string) {
93      input_string.erase(input_string.begin());
94      input_string.pop_back();
95      return input_string;
96  }
97
98
99
100 #endif  // LUNDGREN_HELPERS
```

Appendix B

Test system source code

`test_system/run_tests_n_nodes.yml`

```
---

- hosts: lundgren_nodes{{num_nodes}}

  environment:
    PATH: "{{ ansible_env.PATH }}:/export/home/tmp/mysql-server/bld/bin"

## HUSK Å SETTE --forks 16

  tasks:
    # START
    - import_tasks: tasks/start_lundgren.yml

    - name: Copy lundgren metadata sql-script
      copy:
        src: sql/partitionings/partition_{{ num_nodes }}.sql
        dest: /export/home/tmp/partitions.sql

    - name: Import lundgren metadata
      mysql_db:
        login_unix_socket:
        ↪  /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock
        login_user: "root"
        login_password: ""
        login_port: "13000"
        state: import
        name: all
```

```
27          target: /export/home/tmp/partitions.sql
28
29      # end START
30
31  # 2^14
32      - import_tasks: tasks/load_data.yml
33        vars:
34          num_nodes: "{{num_nodes | int}}"
35          size: 16384
36
37      - import_tasks: tasks/run_tests.yml
38        vars:
39          num_nodes: "{{num_nodes | int}}"
40          size: 16384
41
42      - import_tasks: tasks/delete_data.yml
43
44  # 2^15
45      - import_tasks: tasks/load_data.yml
46        vars:
47          num_nodes: "{{num_nodes | int}}"
48          size: 32768
49
50      - import_tasks: tasks/run_tests.yml
51        vars:
52          num_nodes: "{{num_nodes | int}}"
53          size: 32768
54
55      - import_tasks: tasks/delete_data.yml
56
57  # 2^16
58      - import_tasks: tasks/load_data.yml
59        vars:
60          num_nodes: "{{num_nodes | int}}"
61          size: 65536
62
63      - import_tasks: tasks/run_tests.yml
64        vars:
65          num_nodes: "{{num_nodes | int}}"
66          size: 65536
67
68      - import_tasks: tasks/delete_data.yml
69
```

```
70 # 2^17
71     - import_tasks: tasks/load_data.yml
72       vars:
73         num_nodes: "{{num_nodes | int}}"
74         size: 131072
75
76     - import_tasks: tasks/run_tests.yml
77       vars:
78         num_nodes: "{{num_nodes | int}}"
79         size: 131072
80
81     - import_tasks: tasks/delete_data.yml
82
83 # 2^18
84     - import_tasks: tasks/load_data.yml
85       vars:
86         num_nodes: "{{num_nodes | int}}"
87         size: 262144
88
89     - import_tasks: tasks/run_tests.yml
90       vars:
91         num_nodes: "{{num_nodes | int}}"
92         size: 262144
93
94     - import_tasks: tasks/delete_data.yml
95
96 # 2^19
97     - import_tasks: tasks/load_data.yml
98       vars:
99         num_nodes: "{{num_nodes | int}}"
100        size: 524288
101
102    - import_tasks: tasks/run_tests.yml
103      vars:
104        num_nodes: "{{num_nodes | int}}"
105        size: 524288
106
107    - import_tasks: tasks/delete_data.yml
108
109
110 # 2^20
111    - import_tasks: tasks/load_data.yml
112      vars:
```

```
113         num_nodes: "{{num_nodes | int}}"
114         size: 1048576
115
116     - import_tasks: tasks/run_tests.yml
117       vars:
118         num_nodes: "{{num_nodes | int}}"
119         size: 1048576
120
121     - import_tasks: tasks/delete_data.yml
122
123
124
125 # Kill running mysql servers
126     # - import_tasks: tasks/kill_running.yml
127
128 # The end
```

test_system/load_data.yml

```
1 ---
2
3 - debug:
4     msg: "Loading {{size}} (x 2 tables) rows for {{num_nodes}} nodes..."
5
6 # Copy data of given size
7
8 - name: Copy lhs dataset
9   copy:
10    src: data/{{size}}/lhs_dataset.csv
11    dest: /export/home/tmp/data/
12
13 - name: Copy rhs dataset
14   copy:
15    src: data/{{size}}/rhs_dataset.csv
16    dest: /export/home/tmp/data/
17
18
19 # index of host:
20 #           {{play_hosts.index(inventory_hostname)}}
21 #           {{groups['lundgren_nodes{{num_nodes}}'].index(inventory_hostname)}}
22
23 # Partition data
```

```
24
25 - name: Partition lhs and rhs datasets based on node index, size and num nodes
26
27   vars:
28     index: "{{play_hosts.index(inventory_hostname) | int}}"
29     line_from: "{{ ((size | int) / (num_nodes | int)) * (index | int) + 1 |
       ↪ round}}"
30     line_to: "{{ ((size | int) / (num_nodes | int)) * (index | int) + ((size |
       ↪ int) / (num_nodes | int)) | round}}"
31
32   shell: |
33     sed -n -e {{line_from | int}},{{line_to | int}}p {{item}}.csv >
       ↪ {{item}}_partitioned.csv
34   args:
35     chdir: /export/home/tmp/data
36   with_items:
37     - lhs_dataset
38     - rhs_dataset
39
40 # Load data into MySQL
41
42 - name: Copy table-import sql-script
43   copy:
44     src: sql/load_data.sql
45     dest: /export/home/tmp/
46
47 - name: Execute import-table sql-script
48   mysql_db:
49     login_unix_socket:
       ↪ /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock
50     login_user: "root"
51     login_password: ""
52     login_port: "13000"
53     state: import
54     name: all
55     target: /export/home/tmp/load_data.sql
```

`test_system/start_lundgren.yml`

```
1 ---
2
3 # - name: Copy .my.cnf to node
```

```
4 #   copy:
5 #     src: sql/my.cnf
6 #     dest: ~/.my.cnf
7
8 #-----------------------------------------------------------------------
9
10 - name: Start MySQL with mtr
11   command: ./mtr main.lundgren_install --testcase-timeout=600 --mem
12   args:
13     chdir: /export/home/tmp/mysql-server/bld/mysql-test
14   async: 9999
15   poll: 0
16
17 - name: Wait for MySQL to start
18   command: /bin/sleep 12
19
20 - name: Alter root to use old style login, for ansible mysql module to work
21   command: mysql -u root --socket
     ↪ /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock -e
     ↪ "ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY
     ↪ ''"
22
23 - name: Copy lundgren setup sql-script
24   copy:
25     src: sql/lundgren_plugin_setup.sql
26     dest: /export/home/tmp/
27
28 - name: Execute lundgren setup script
29   mysql_db:
30     login_unix_socket:
       ↪ /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock
31     login_user: "root"
32     login_password: ""
33     login_port: "13000"
34     state: import
35     name: all
36     target: /export/home/tmp/lundgren_plugin_setup.sql
37
38 - name: Copy self node id setup sql-script
39   copy:
40     src: sql/self_node_id_setters/{{play_hosts.index(inventory_hostname)}}.sql
41     dest: /export/home/tmp/node_id_setter.sql
42
```

```
43 - name: Execute self node id setter
44   mysql_db:
45     login_unix_socket:
       ↪ /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock
46     login_user: "root"
47     login_password: ""
48     login_port: "13000"
49     state: import
50     name: all
51     target: /export/home/tmp/node_id_setter.sql
```

test_system/run_tests.yml

```
1 ---
2
3 - name: Turn on network delay
4   command: /usr/sbin/tc qdisc add dev ens3 root netem delay 10ms 5ms 25%
5   become: yes
6
7 #------------------
8
9 - name: Run tests
10  command: python3 main.py {{num_nodes}} {{size}}
11  args:
12    chdir: /export/home/tmp/test_runner/
13  run_once: true
14
15 # ------------------
16
17
18 - name: Turn off network delay
19  command: /usr/sbin/tc qdisc del dev ens3 root netem delay 10ms 5ms 25%
20  become: yes
21
22
23 - name: Copy result back
24  fetch:
25    src: /export/home/tmp/test_runner/results_{{num_nodes}}_{{size}}.csv
26    dest: result/
27  run_once: true
```

test_system/delete_data.yml

```
---

# Delete lhs and rhs data

- name: Copy table-delete-rows sql-script
  copy:
    src: sql/delete_data.sql
    dest: /export/home/tmp/

- name: Execute delete-table-rows sql-script
  mysql_db:
    login_unix_socket:
     ↪ /export/home/tmp/mysql-server/bld/mysql-test/var/tmp/mysqld.1.sock
    login_user: "root"
    login_password: ""
    login_port: "13000"
    state: import
    name: all
    target: /export/home/tmp/delete_data.sql
```

`test_system/installation/install_mysql_lundgren.yml`

```
---
- hosts: lundgren_nodes

  tasks:
    - name: ensure packages [MySQL-python, ninja-build] are installed
      yum:
        name: "{{ packages }}"
      vars:
        packages:
        - MySQL-python
        - ninja-build
      become: yes

    # - name: Install connector/c++ main
    #   yum:
     ↪ name=https://dev.mysql.com/get/Downloads/Connector-C++/mysql-connector-c++-8.0.15-1.el7.x
    #   become: yes

    # - name: Install connector/c++ jdbc
```

```
19      #   yum:
        ↪   name=https://dev.mysql.com/get/Downloads/Connector-C++/mysql-connector-c++-jdbc-8.0.15-1.
20      #   become: yes
21
22      # - name: Install connector/c++ devel
23      #   yum:
        ↪   name=https://dev.mysql.com/get/Downloads/Connector-C++/mysql-connector-c++-devel-8.0.15-1
24      #   become: yes
25
26      # - name: Install Connector C++
27      #   shell: |
28      #     yum install {{item}}
29      #   args:
30      #     chdir: /home/heggen
31      #   with_items:
32      #     - 33393928.mysql-connector-c++-8.0.16-1.el7.x86_64.rpm
33      #     - 33393924.mysql-connector-c++-jdbc-8.0.16-1.el7.x86_64.rpm
34      #     - 33393920.mysql-connector-c++-devel-8.0.16-1.el7.x86_64.rpm
35      #   become: yes
36
37
38      - name: Copy mysqlx to include directory
39        command: cp -r /usr/include/mysql-cppconn-8/mysqlx /usr/include/mysqlx
40        become: yes
41
42      - name: Download or update git repo for MySQL Lundgren
43        git:
44          accept_hostkey: yes
45          repo: https://github.com/kahole/mysql-server.git
46          dest: /export/home/tmp/mysql-server
47          depth: 1
48          version: lundgren_no_result
49
50      - name: Create build directory
51        file:
52          path: /export/home/tmp/mysql-server/bld
53          state: directory
54
55      - name: cmake MySQL
56        command: cmake .. -GNinja -DWITH_BOOST=/usr/global/share
57        args:
58          chdir: /export/home/tmp/mysql-server/bld
59
```

```
60      - name: Compile MySQL
61        command: ninja-build
62        args:
63          chdir: /export/home/tmp/mysql-server/bld
64
65
66      # Dropp å installere, waste of time:
67
68      # - name: Install MySQL
69      #   command: ninja-build install
70      #   args:
71      #     chdir: /export/home/tmp/mysql-server/bld
72      #   become: yes
73
74      # - name: Build data directories
75      #   command: chdir=/usr/local/mysql {{ item }}
76      #   with_items:
77      #   - groupadd mysql
78      #   - useradd -r -g mysql -s /bin/false mysql
79      #   - mkdir mysql-files
80      #   - chown mysql:mysql mysql-files
81      #   - chmod 750 mysql-files
82      #   - bin/mysqld --initialize --user=mysql
83      #   become: yes
```

test_system/installation/install_test_runner.yml

```
1  ---
2  - hosts: test_node
3
4    environment:
5      http_proxy: http://www-proxy.uk.oracle.com:80
6      https_proxy: http://www-proxy.uk.oracle.com:80
7      no_proxy: no.oracle.com,oraclevcn.com,169.254.169.254
8
9    tasks:
10   #   - name: ensure packages [SqlAlchemy, pymysql] are installed
11   #     yum:
12   #       name: "{{ packages }}"
13   #     vars:
14   #       packages:
15   #       - python34-sqlalchemy
```

```
16  #       - python34-PyMySQL
17  #       - python34-mysql
18  #       #- python34-mysql
19  #     become: yes
20
21    - name: Copy test_runner python script
22      copy:
23        src: ../../test_runner/main.py
24        dest: /export/home/tmp/test_runner/
```

test_system/run_all.sh

```
1  ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=2"
2
3  ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=2"
4
5  ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=4"
6
7  ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=4"
8
9  ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=8"
10
11 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=8"
12
13 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=16"
14
15 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=16"
16
17 mv
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/100.103.14.11/export/home/tmp/test_runner/
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/ekte/run_1
18
19 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=2"
20
21 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=2"
22
23 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=4"
24
25 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=4"
26
27 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=8"
28
```

```
29 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=8"
30
31 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=16"
32
33 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=16"
34
35 mv
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/100.103.14.11/export/home/tmp/test_runner/
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/ekte/run_2
36
37
38 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=2"
39
40 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=2"
41
42 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=4"
43
44 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=4"
45
46 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=8"
47
48 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=8"
49
50 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=16"
51
52 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=16"
53
54 mv
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/100.103.14.11/export/home/tmp/test_runner/
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/ekte/run_3
55
56
57 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=2"
58
59 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=2"
60
61 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=4"
62
63 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=4"
64
65 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=8"
66
67 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=8"
```

```
68
69 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=16"
70
71 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=16"
72
73 mv
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/100.103.14.11/export/home/tmp/test_runner/
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/ekte/run_4
74
75 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=2"
76
77 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=2"
78
79 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=4"
80
81 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=4"
82
83 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=8"
84
85 ansible-playbook kill_running_playbook.yml -f 16 --extra-vars "num_nodes=8"
86
87 ansible-playbook run_tests_n_nodes.yml -f 16 --extra-vars "num_nodes=16"
88
89 mv
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/100.103.14.11/export/home/tmp/test_runner/
   ↪  ~/TDT4900-masteroppgave/evaluation/ansible/result/ekte/run_5
90
91 # ansible-playbook shutdown_nodes.yml -f 16 --extra-vars "num_nodes=16"
```

`test_system/sql/load_data.sql`

```
1  USE test;
2
3  CREATE TABLE IF NOT EXISTS lhs (
4    lhs_10_10 INT,
5    lhs_20_20 INT,
6    lhs_30_30 INT,
7    lhs_40_40 INT,
8    lhs_50_50 INT,
9    lhs_60_60 INT,
10   lhs_70_70 INT,
11   lhs_80_80 INT,
```

```
  lhs_90_90 INT,
  lhs_100_100 INT,
  lhs_all_equal INT,
  lhs_normal INT,
  lhs_uniform INT
);

CREATE TABLE IF NOT EXISTS rhs (
  rhs_10_10 INT,
  rhs_20_20 INT,
  rhs_30_30 INT,
  rhs_40_40 INT,
  rhs_50_50 INT,
  rhs_60_60 INT,
  rhs_70_70 INT,
  rhs_80_80 INT,
  rhs_90_90 INT,
  rhs_100_100 INT,
  rhs_all_equal INT,
  rhs_normal INT,
  rhs_uniform INT
);

LOAD DATA INFILE '/export/home/tmp/data/lhs_dataset_partitioned.csv' INTO TABLE
  ↪  lhs FIELDS TERMINATED BY ',';
LOAD DATA INFILE '/export/home/tmp/data/rhs_dataset_partitioned.csv' INTO TABLE
  ↪  rhs FIELDS TERMINATED BY ',';
```

`test_system/sql/lundgren_plugin_setup.sql`

```
-- PLUGIN

-- INSTALL PLUGIN lundgren SONAME 'lundgren.so';

-- SELECT DB
USE test;

-- Have to set this high for the interim cleanup operations
SET GLOBAL group_concat_max_len=4294967295;

-- USER

```

```
13 -- CREATE USER IF NOT EXISTS 'lundgren_user'@'%' IDENTIFIED BY '';
14 CREATE USER IF NOT EXISTS 'lundgren_user'@'%' IDENTIFIED WITH
   ↪  mysql_native_password BY '';
15 GRANT ALL PRIVILEGES ON *.* TO 'lundgren_user'@'%';
16
17
18 -- METADATA TABLES
19
20 CREATE TABLE IF NOT EXISTS lundgren_node (
21   id INT UNSIGNED PRIMARY KEY,
22   host_l VARCHAR(80) NOT NULL,
23   port_l INT UNSIGNED,
24   database_l VARCHAR(80) NOT NULL,
25   username_l VARCHAR(50),
26   password_l VARCHAR(50)
27 );
28
29 CREATE TABLE IF NOT EXISTS lundgren_shard_key (
30   id INT UNSIGNED PRIMARY KEY,
31   column_name VARCHAR(80) NOT NULL,
32   range_start INT UNSIGNED,
33   range_end INT UNSIGNED
34 );
35
36 CREATE TABLE IF NOT EXISTS lundgren_partition (
37   id INT UNSIGNED PRIMARY KEY,
38   nodeId INT UNSIGNED,
39   shardKeyId INT UNSIGNED,
40   table_name VARCHAR(80) NOT NULL,
41   FOREIGN KEY (nodeId) REFERENCES lundgren_node(id),
42   FOREIGN KEY (shardKeyId) REFERENCES lundgren_shard_key(id)
43 );
44
45 CREATE TABLE IF NOT EXISTS lundgren_self_node_id (
46   node_id INT UNSIGNED PRIMARY KEY
47 );
```

`test_system/sql/partitions/partition_16.sql`

```
1 -- SELECT DB
2 USE test;
3
```

```
4  INSERT INTO lundgren_node VALUES (0, "100.103.14.11", 13010, "test",
   ↪  "lundgren_user", NULL);
5  INSERT INTO lundgren_node VALUES (1, "100.103.14.12", 13010, "test",
   ↪  "lundgren_user", NULL);
6  INSERT INTO lundgren_node VALUES (2, "100.103.14.13", 13010, "test",
   ↪  "lundgren_user", NULL);
7  INSERT INTO lundgren_node VALUES (3, "100.103.14.14", 13010, "test",
   ↪  "lundgren_user", NULL);
8  INSERT INTO lundgren_node VALUES (4, "100.103.14.15", 13010, "test",
   ↪  "lundgren_user", NULL);
9  INSERT INTO lundgren_node VALUES (5, "100.103.14.16", 13010, "test",
   ↪  "lundgren_user", NULL);
10 INSERT INTO lundgren_node VALUES (6, "100.103.14.17", 13010, "test",
   ↪  "lundgren_user", NULL);
11 INSERT INTO lundgren_node VALUES (7, "100.103.14.18", 13010, "test",
   ↪  "lundgren_user", NULL);
12 INSERT INTO lundgren_node VALUES (8, "100.103.14.19", 13010, "test",
   ↪  "lundgren_user", NULL);
13 INSERT INTO lundgren_node VALUES (9, "100.103.14.20", 13010, "test",
   ↪  "lundgren_user", NULL);
14 INSERT INTO lundgren_node VALUES (10, "100.103.14.21", 13010, "test",
   ↪  "lundgren_user", NULL);
15 INSERT INTO lundgren_node VALUES (11, "100.103.14.22", 13010, "test",
   ↪  "lundgren_user", NULL);
16 INSERT INTO lundgren_node VALUES (12, "100.103.14.23", 13010, "test",
   ↪  "lundgren_user", NULL);
17 INSERT INTO lundgren_node VALUES (13, "100.103.14.24", 13010, "test",
   ↪  "lundgren_user", NULL);
18 INSERT INTO lundgren_node VALUES (14, "100.103.14.25", 13010, "test",
   ↪  "lundgren_user", NULL);
19 INSERT INTO lundgren_node VALUES (15, "100.103.14.26", 13010, "test",
   ↪  "lundgren_user", NULL);
20
21 INSERT INTO lundgren_shard_key VALUES (0, "nothing", 165, 500);
22 INSERT INTO lundgren_shard_key VALUES (1, "nothing", 0, 165);
23
24 INSERT INTO lundgren_partition VALUES (1, 0, 0, "lhs");
25 INSERT INTO lundgren_partition VALUES (2, 0, 1, "rhs");
26
27 INSERT INTO lundgren_partition VALUES (3, 1, 0, "lhs");
28 INSERT INTO lundgren_partition VALUES (4, 1, 1, "rhs");
29
30 INSERT INTO lundgren_partition VALUES (5, 2, 0, "lhs");
```

```
31 INSERT INTO lundgren_partition VALUES (6, 2, 1, "rhs");
32
33 INSERT INTO lundgren_partition VALUES (7, 3, 0, "lhs");
34 INSERT INTO lundgren_partition VALUES (8, 3, 1, "rhs");
35
36 INSERT INTO lundgren_partition VALUES (9, 4, 0, "lhs");
37 INSERT INTO lundgren_partition VALUES (10, 4, 1, "rhs");
38
39 INSERT INTO lundgren_partition VALUES (11, 5, 0, "lhs");
40 INSERT INTO lundgren_partition VALUES (12, 5, 1, "rhs");
41
42 INSERT INTO lundgren_partition VALUES (13, 6, 0, "lhs");
43 INSERT INTO lundgren_partition VALUES (14, 6, 1, "rhs");
44
45 INSERT INTO lundgren_partition VALUES (15, 7, 0, "lhs");
46 INSERT INTO lundgren_partition VALUES (16, 7, 1, "rhs");
47
48 INSERT INTO lundgren_partition VALUES (17, 8, 0, "lhs");
49 INSERT INTO lundgren_partition VALUES (18, 8, 1, "rhs");
50
51 INSERT INTO lundgren_partition VALUES (19, 9, 0, "lhs");
52 INSERT INTO lundgren_partition VALUES (20, 9, 1, "rhs");
53
54 INSERT INTO lundgren_partition VALUES (21, 10, 0, "lhs");
55 INSERT INTO lundgren_partition VALUES (22, 10, 1, "rhs");
56
57 INSERT INTO lundgren_partition VALUES (23, 11, 0, "lhs");
58 INSERT INTO lundgren_partition VALUES (24, 11, 1, "rhs");
59
60 INSERT INTO lundgren_partition VALUES (25, 12, 0, "lhs");
61 INSERT INTO lundgren_partition VALUES (26, 12, 1, "rhs");
62
63 INSERT INTO lundgren_partition VALUES (27, 13, 0, "lhs");
64 INSERT INTO lundgren_partition VALUES (28, 13, 1, "rhs");
65
66 INSERT INTO lundgren_partition VALUES (29, 14, 0, "lhs");
67 INSERT INTO lundgren_partition VALUES (30, 14, 1, "rhs");
68
69 INSERT INTO lundgren_partition VALUES (31, 15, 0, "lhs");
70 INSERT INTO lundgren_partition VALUES (32, 15, 1, "rhs");
```

test_system/sql/node_id_setters/0.sql

```
USE test;
INSERT INTO lundgren_self_node_id VALUES (0);
```

`test_system/sql/delete_data.sql`

```
USE test;

delete from lhs;
delete from rhs;
```

`test_system/data_generation/main.py`

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

def generate_percentage_matches(num_rows, overlap):
    lhs_matches = np.arange(0, num_rows, dtype=int)
    overlapping_num = np.floor((1.0-overlap) * num_rows)
    rhs_matches = np.arange(overlapping_num, num_rows + overlapping_num)

    np.random.shuffle(lhs_matches)
    np.random.shuffle(rhs_matches)
    return lhs_matches.astype(int), rhs_matches.astype(int)

def generate_all_equal(num_rows):
    return np.full((num_rows), 1), np.full((num_rows), 1)


def generate_normal_distribution(num_rows, sigma, mu, sigma2, mu2):

    lhs_normal = np.arange(mu, mu + num_rows)
    rhs_normal = np.random.normal(mu, sigma, num_rows)

    np.random.shuffle(lhs_normal)
    np.random.shuffle(rhs_normal)

    return lhs_normal.astype(int), rhs_normal
```

```
30
31 def generate_uniform_distribution(num_rows):
32
33     low = 0
34     high = 1000
35
36     lhs_uniform = np.arange(high/2, num_rows + high/2)
37     rhs_uniform = np.random.uniform(low, high, num_rows)
38
39     np.random.shuffle(lhs_uniform)
40     np.random.shuffle(rhs_uniform)
41
42     return lhs_uniform.astype(int), rhs_uniform.astype(int)
43
44
45 def main(size):
46
47     np.random.seed(99)
48
49     lhs_10_10_matches, rhs_10_10_matches = generate_percentage_matches(size,
       ↪  0.1)
50     lhs_20_20_matches, rhs_20_20_matches = generate_percentage_matches(size,
       ↪  0.2)
51     lhs_30_30_matches, rhs_30_30_matches = generate_percentage_matches(size,
       ↪  0.3)
52     lhs_40_40_matches, rhs_40_40_matches = generate_percentage_matches(size,
       ↪  0.4)
53     lhs_50_50_matches, rhs_50_50_matches = generate_percentage_matches(size,
       ↪  0.5)
54     lhs_60_60_matches, rhs_60_60_matches = generate_percentage_matches(size,
       ↪  0.6)
55     lhs_70_70_matches, rhs_70_70_matches = generate_percentage_matches(size,
       ↪  0.7)
56     lhs_80_80_matches, rhs_80_80_matches = generate_percentage_matches(size,
       ↪  0.8)
57     lhs_90_90_matches, rhs_90_90_matches = generate_percentage_matches(size,
       ↪  0.9)
58     lhs_100_100_matches, rhs_100_100_matches =
       ↪  generate_percentage_matches(size, 1.0)
59
60     lhs_all_equal, rhs_all_equal = generate_all_equal(size)
61
62     sigma = 50.0
```

```
63      mu = 300.0
64      sigma2 = 50.0
65      mu2 = 550.0
66
67      lhs_normal, rhs_normal = generate_normal_distribution(size, sigma, mu,
        ↪  sigma2, mu2)
68      lhs_uniform, rhs_uniform = generate_uniform_distribution(size)
69
70      lhs_dataset = np.column_stack((lhs_10_10_matches, lhs_20_20_matches,
        ↪  lhs_30_30_matches, lhs_40_40_matches, lhs_50_50_matches,
        ↪  lhs_60_60_matches, lhs_70_70_matches, lhs_80_80_matches,
        ↪  lhs_90_90_matches, lhs_100_100_matches, lhs_all_equal, lhs_normal,
        ↪  lhs_uniform))
71      rhs_dataset = np.column_stack((rhs_10_10_matches, rhs_20_20_matches,
        ↪  rhs_30_30_matches, rhs_40_40_matches, rhs_50_50_matches,
        ↪  rhs_60_60_matches, rhs_70_70_matches, rhs_80_80_matches,
        ↪  rhs_90_90_matches, rhs_100_100_matches, rhs_all_equal, rhs_normal,
        ↪  rhs_uniform))
72
73      np.savetxt("../ansible/data/" + str(size) + "/lhs_dataset.csv",
        ↪  lhs_dataset, fmt='%i', delimiter=",")
74      np.savetxt("../ansible/data/" + str(size) + "/rhs_dataset.csv",
        ↪  rhs_dataset, fmt='%i', delimiter=",")
75
76      #------------------------------
77      # PLOTTING
78      #------------------------------
79
80      plt.rcParams["font.family"] = "Verdana"
81      plt.rcParams["font.size"] = 16
82
83      num_bins = 256
84
85      lhs_count, lhs_bins, lhs_ignored = plt.hist(lhs_normal, alpha=0.6,
        ↪  bins=num_bins, color='blue', histtype='stepfilled', density=True,
        ↪  label='lhs')
86      rhs_count, rhs_bins, rhs_ignored = plt.hist(rhs_normal, alpha=0.6,
        ↪  bins=num_bins, color='red', histtype='stepfilled', density=True,
        ↪  label='rhs')
87      # plt.hist([], alpha=0.8, color='purple', histtype='stepfilled',
        ↪  density=True, label='overlap')
88
```

```
89      plt.plot(rhs_bins, 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (rhs_bins -
        ↪  mu)**2 / (2 * sigma**2) ), linewidth=1, color='r')
90      # plt.plot(rhs_bins, 1/(sigma2 * np.sqrt(2 * np.pi)) * np.exp( - (rhs_bins
        ↪  - mu2)**2 / (2 * sigma2**2) ), linewidth=1, color='b')
91      plt.legend(loc='upper left')
92      plt.xlabel('Column value', labelpad=5)
93      plt.ylabel('Density (%)', labelpad=10)
94
95      # plt.ylim(top=0.000175)
96      # plt.ylim(bottom=0.0)
97      plt.xlim(right=600.0)
98      plt.xlim(left=(-50.0))
99
100     plt.tight_layout()
101     print(len(lhs_normal))
102
103     ax = plt.gca()
104     axins = zoomed_inset_axes(ax, 4.9, loc=5)
105     axins.hist(lhs_normal, alpha=0.6, bins=num_bins, color='blue',
        ↪  histtype='stepfilled', density=True, label='lhs')
106     axins.hist(rhs_normal, alpha=0.6, bins=num_bins, color='red',
        ↪  histtype='stepfilled', density=True, label='rhs')
107
108     x1, x2, y1, y2 = 430, 470, 0.0, 0.0002 # specify the limits
109     axins.set_xlim(x1, x2) # apply the x-limits
110     axins.set_ylim(y1, y2) # apply the y-limits
111     mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
112     plt.yticks(visible=False)
113     plt.xticks(visible=False)
114     plt.savefig("normal_distribution_histogram.pdf", format='pdf')
115
116     plt.show()
117
118     plt.hist(lhs_uniform, alpha=0.6, bins=num_bins, color='blue',
        ↪  histtype='stepfilled', density=True, label='lhs')
119     plt.hist(rhs_uniform, alpha=0.6, bins=num_bins, color='red',
        ↪  histtype='stepfilled', density=True, label='rhs')
120     #plt.hist([], alpha=0.8, color='purple', histtype='stepfilled',
        ↪  density=True, label='overlap')
121     # plt.ylim(top=0.000175)
122     # plt.ylim(bottom=0.0)
123     plt.xlim(right=1300.0)
124     plt.xlim(left=(-50.0))
```

```
    plt.legend(loc='upper right')
    plt.xlabel('Column value', labelpad=5)
    plt.ylabel('Density (%)', labelpad=10)

    plt.tight_layout()

    # ax = plt.gca()
    # axins = zoomed_inset_axes(ax, 0.7, loc=1)
    # axins.hist(lhs_uniform, alpha=0.6, bins=num_bins, color='blue',
    ↪  histtype='stepfilled', density=True, label='lhs')
    # axins.hist(rhs_uniform, alpha=0.6, bins=num_bins, color='red',
    ↪  histtype='stepfilled', density=True, label='rhs')

    # x1, x2, y1, y2 = 400, 1200, 0.0, 0.0002 # specify the limits
    # axins.set_xlim(x1, x2) # apply the x-limits
    # axins.set_ylim(y1, y2) # apply the y-limits
    # mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
    # plt.yticks(visible=False)
    # plt.xticks(visible=False)
    plt.savefig("uniform_distribution_histogram.pdf", format='pdf')

    plt.show()

if __name__ == "__main__":
    main(8192)
    main(16384)
    main(32768)
    main(65536)
    main(131072)
    main(262144)
    main(524288)
    main(1048576)
    main(2**21)
```

test_system/test_runner/main.py

```
from sqlalchemy import event
from sqlalchemy.engine import Engine
from sqlalchemy import create_engine
from sqlalchemy import exc
import time
from string import Template
```

```
7  import sys
8  import csv
9  import logging
10
11 latest_execution_time = 0.0
12
13 logging.basicConfig()
14 logging.getLogger('sqlalchemy').setLevel(logging.ERROR)
15
16 @event.listens_for(Engine, "before_cursor_execute")
17 def before_cursor_execute(conn, cursor, statement,
18                           parameters, context, executemany):
19     conn.info.setdefault('query_start_time', []).append(time.time())
20
21 @event.listens_for(Engine, "after_cursor_execute")
22 def after_cursor_execute(conn, cursor, statement,
23                           parameters, context, executemany):
24     total = time.time() - conn.info['query_start_time'].pop(-1)
25
26     global latest_execution_time
27     latest_execution_time = total
28
29 engine =
   ↪  create_engine('mysql+pymysql://root@localhost:13000/test?unix_socket=/export/home/tmp/mysql-s
   ↪  pool_reset_on_return=None)
30
31 hosts = ["100.103.14.12", "100.103.14.13", "100.103.14.14", "100.103.14.15",
   ↪  "100.103.14.16", "100.103.14.17", "100.103.14.18", "100.103.14.19",
   ↪  "100.103.14.20", "100.103.14.21", "100.103.14.22", "100.103.14.23",
   ↪  "100.103.14.24", "100.103.14.25", "100.103.14.26"]
32
33 back_num = 16 - int(sys.argv[1])
34
35 active_hosts =  hosts if back_num == 0 else hosts[:-back_num]
36
37 engines = [create_engine('mysql+pymysql://lundgren_user@' + h + ':13000/test',
   ↪  pool_reset_on_return=None) for h in active_hosts]
38
39 engines.insert(0, engine)
40
41
42 def warmup_query(query):
43     try:
```

```
44          # Run warmup query and ignore its time
45          res = engine.execute(query)
46          res.close()
47          return True, "no"
48
49      except exc.SQLAlchemyError as e:
50          return False, str(e)
51
52  def measure_query(query):
53
54      try:
55          # Run real query (how many times)
56          res = engine.execute(query)
57          res.close()
58          return latest_execution_time, 'no'
59
60      except exc.SQLAlchemyError as e:
61          return -1.0, str(e)
62
63  #--------------------------------------------------------------------------------------------
64  # QUERIES
65
66  join_strategies = ["data_to_query", "semi", "bloom", "hash_redist",
    ↪  "sort_merge"]
67
68  columns = ["normal", "uniform", "10_10", "20_20", "30_30", "40_40", "50_50",
    ↪  "60_60", "70_70", "80_80", "90_90", "100_100"]
69
70  query_template = Template("/*distributed<join_strategy=$join_strategy>*/SELECT
    ↪  ")
71  proj_template = Template("lhs.lhs_$column, rhs.rhs_$column")
72  from_template = Template(" FROM lhs JOIN rhs ON lhs.lhs_$column =
    ↪  rhs.rhs_$column;")
73
74  def generate_query(join_strategy, column):
75      query_ = query_template.substitute(join_strategy=join_strategy)
76
77      proj = ""
78      # project current column
79      proj += proj_template.substitute(column=column)
80      proj += ", "
81      # plus 3 other columns
82      proj_count = 0
```

```
83      for c in columns:
84          if (not c == column) and proj_count < 2:
85              proj += proj_template.substitute(column=c)
86              proj += ", "
87              proj_count += 1
88
89      proj = proj[:-2]
90
91      from_ = from_template.substitute(column=column)
92      return query_ + proj + from_
93
94  #-------------------------------------------------------------------------------------------
95
96  def delete_interim_tables():
97
98      # try:
99      for e in engines:
100         res = e.execute("SELECT CONCAT( 'DROP TABLE ', GROUP_CONCAT(table_name)
            ↪  , ';' ) AS statement FROM information_schema.tables  WHERE
            ↪  table_schema = 'test' AND table_name LIKE 'interim_%%';")
101
102         drop_interim_statement = ""
103         for r in res:
104             drop_interim_statement = r[0]
105
106         # drop_interim_statement = res.first()[0]
107         res.close()
108
109         if not drop_interim_statement == None:
110             e.execute(str(drop_interim_statement))
111     # except exc.SQLAlchemyError as e:
112     #     pass
113
114
115 def main():
116     num_nodes = sys.argv[1]
117     num_rows = sys.argv[2]
118
119     with open('results_' + num_nodes + '_' + num_rows + '.csv', 'w') as
        ↪  csv_file:
120
121         fieldnames = ['num_nodes', 'num_rows', 'strategy', 'column', 'time',
            ↪  'query', 'error']
```

```
122         writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
123         writer.writeheader()
124 
125         for strategy in join_strategies:
126             for column in columns:
127 
128                 entry = {'num_nodes': num_nodes, 'num_rows': num_rows,
                    ↪  'strategy': strategy, 'column': column}
129                 entry['query'] = generate_query(strategy, column)
130 
131                 ok, err = warmup_query(entry['query'])
132                 #if strategy == "sort_merge" or strategy == "hash_redist":
133                 #    time.sleep(1)
134 
135                 entry['time'], entry['error'] = measure_query(entry['query'])
136                 #if strategy == "sort_merge" or strategy == "hash_redist":
137                 #    time.sleep(1)
138                 writer.writerow(entry)
139 
140                 #print(num_nodes + " " + strategy + " " + column + " " +
                    ↪  entry['query'])
141 
142             delete_interim_tables()
143 
144 if __name__ == '__main__':
145     main()
```

test_system/test_runner/result_plots.py

```
1 import csv
2 import matplotlib.pyplot as plt
3 from matplotlib.ticker import FuncFormatter
4 import numpy as np
5 import pylab
6 
7 def power_of(x, pos):
8     'The two args are the value and tick position'
9     return x
10 
11 def log_of(x, pos):
12     'The two args are the value and tick position'
13     return "$2^{" + str(int(np.math.log(int(x), 2))) + "}$"
```

```
14
15 formatter = FuncFormatter(power_of)
16 formatter2 = FuncFormatter(log_of)
17
18
19 CSV_FILE = 'results_atum_combined.csv'
20 NUM_RUNS = 5
21 THROUGHPUT = False
22
23 def row_key(row):
24     return str(row['num_nodes']) + str(row['num_rows']) + str(row['strategy'])
       ↪  + str(row['column'])
25
26 def calculate_mean_and_throughput(rows):
27     mean_dict = {}
28     values_dict = {}
29
30     for r in rows:
31         if row_key(r) in mean_dict:
32             mean_dict[row_key(r)]['time'] += float(r['time'])
33             values_dict[row_key(r)] += [float(r['time'])]
34         else:
35             r['time'] = float(r['time'])
36             mean_dict[row_key(r)] = r
37             values_dict[row_key(r)] = [r['time']]
38
39     # dele på 5
40     for mr in mean_dict.values():
41         mr['time'] = float(mr['time']) / float(NUM_RUNS)
42         mr['mean'] = float(mr['time'])
43         mr['throughput'] = (float(mr['num_rows']) * 2.0) / float(mr['time']) /
           ↪  1000.0 # Throughput
44
45     for key in mean_dict.keys():
46         numerator = sum([(val - mean_dict[key]['mean'])**2 for val in
           ↪  values_dict[key]])
47         denominator = len(values_dict[key])
48         mean_dict[key]['variance'] = numerator / denominator
49         mean_dict[key]['standard_deviation'] =
           ↪  np.math.sqrt(mean_dict[key]['variance'])
50
51     return mean_dict.values()
52
```

```
53 def wrangle_results_from_csv():
54     with open(CSV_FILE, 'r') as csv_file:
55 
56         reader = csv.DictReader(csv_file)
57         rows = [r for r in reader]
58         return calculate_mean_and_throughput(rows)
59 
60 
61 rows = wrangle_results_from_csv()
62 
63 # Dumb way of getting all strategies, and number of nodes
64 strategies = list(set([r['strategy'] for r in rows]))
65 num_nodes = sorted(set([r['num_nodes'] for r in rows]), key=int)
66 num_rows = sorted(map(lambda x: int(x), set([r['num_rows'] for r in rows])))
67 join_columns = list(set([r['column'] for r in rows]))
68 
69 
70 join_columns = ['10_10', '20_20', '30_30', '40_40', '50_50', '60_60', '70_70',
   ↪  '80_80', '90_90', '100_100', 'normal', 'uniform']
71 
72 ################################################################################
73 ##### PLOTTING
74 ################################################################################
75 
76 line_and_color = {'data_to_query': 'b-*', 'semi': 'y-o', 'bloom': 'r-s',
   ↪  'hash_redist': 'm--', 'sort_merge': 'g-^', 'sort_merge_indices': 'c-d',
   ↪  'hash_redist_old': 'b-d'}
77 plt.rcParams["font.family"] = "Verdana"
78 plt.rcParams["font.size"] = 12
79 
80 plt.rcParams['lines.linewidth'] = 2
81 plt.rcParams['lines.markersize'] = 7
82 
83 # strategies.remove('data_to_query')
84 # strategies.remove('semi')
85 # strategies.remove('bloom')
86 # strategies.remove('hash_redist')
87 # strategies.remove('sort_merge')
88 
89 # strategies.remove('sort_merge_indices')
90 # strategies.remove('hash_redist_old')
91 
92 
```

```
93 # #
   ↪ ##############################################################################
94 # # ##### X = data size, Y = time. Plot for each #nodes and join columns
95 # #
   ↪ ##############################################################################
96
97 # join_columns = ['50_50', '60_60']
98 # fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(join_columns),
   ↪ sharey=True)
99 fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(join_columns),
   ↪ sharey=False)
100 for nodes in num_nodes:
101     for join_column in join_columns:
102         j_c = [r for r in rows if r['column'] == join_column and r['num_nodes']
            ↪ == nodes]
103         for strategy in strategies:
104             title = "Nodes: " + nodes + ' ' + "Column:" + join_column
105             strategy_rows = [[int(r['num_rows']), (r['throughput'] if
                ↪ THROUGHPUT else r['time'])] for r in j_c if r['strategy'] ==
                ↪ strategy]
106             strategy_rows = sorted(strategy_rows, key=lambda x: x[0])
107             x = [r[0] for r in strategy_rows]
108             y = [r[1] for r in strategy_rows]
109             axes[num_nodes.index(nodes),
                ↪ join_columns.index(join_column)].plot(x, y,
                ↪ line_and_color[strategy], label=strategy) #, basex=2)
110             axes[num_nodes.index(nodes),
                ↪ join_columns.index(join_column)].set_title("Nodes: " + nodes +
                ↪ ' ' + "Column:" + join_column)
111             axes[num_nodes.index(nodes),
                ↪ join_columns.index(join_column)].tick_params(labelright=True,
                ↪ labelleft=True)
112             # plt.setp(axes[num_nodes.index(nodes),
                ↪ join_columns.index(join_column)].get_xticklabels(),
                ↪ rotation=45)
113             # axes[join_columns.index(join_column)].plot(x, y, label=strategy)
114             # axes[join_columns.index(join_column)].set_title("Nodes: " + nodes
                ↪ + ' ' + "Column:" + join_column)
115             # plt.setp(axes[join_columns.index(join_column)].get_xticklabels(),
                ↪ rotation=45)
116 # plt.setp(axes, xticks=num_rows)
117 # plt.xlabel("Data size (log2)")
118 # plt.ylabel("Time (sec)")
```

```
119
120 handles, labels = axes[-1, -1].get_legend_handles_labels()
121 # fig.legend(handles, labels, loc='upper right')
122 fig.legend(handles, labels, loc='upper center', ncol=5)
123 fig.set_size_inches(78, 30)
124 plt.savefig("data_size.pdf", format="pdf")
125 plt.close()
126
127 # #
    ↪ ##############################################################################
128 # # ##### X = number of nodes, Y = time. Plot for each data size and join
    ↪ column
129 # #
    ↪ ##############################################################################
130 # num_rows = [524288, 262144]
131 # join_columns = ['50_50', 'normal']
132
133 fig, axes = plt.subplots(nrows=len(num_rows), ncols=len(join_columns),
    ↪ sharey=False)
134 for row_size in num_rows:
135     for join_column in join_columns:
136         j_c = [r for r in rows if r['column'] == join_column and r['num_rows']
            ↪ == str(row_size)]
137         for strategy in strategies:
138             title = "Data size: " + "2^" + str(int(np.math.log2(row_size))) + '
                ↪ ' + "Column:" + join_column
139             strategy_rows = [[int(r['num_nodes']), (r['throughput'] if
                ↪ THROUGHPUT else r['time'])] for r in j_c if r['strategy'] ==
                ↪ strategy]
140             strategy_rows = sorted(strategy_rows, key=lambda x: x[0])
141             x = [r[0] for r in strategy_rows]
142             y = [r[1] for r in strategy_rows]
143             axes[num_rows.index(row_size),
                ↪ join_columns.index(join_column)].semilogx(x, y,
                ↪ line_and_color[strategy], label=strategy, basex=2)
144             axes[num_rows.index(row_size),
                ↪ join_columns.index(join_column)].set_title(title)
145             # axes[num_rows.index(row_size),
                ↪ join_columns.index(join_column)].tick_params(labelright=True,
                ↪ labelleft=True)
146             axes[num_rows.index(row_size),
                ↪ join_columns.index(join_column)].tick_params(labelleft=True)
```

```
147             axes[num_rows.index(row_size),
                ↪  join_columns.index(join_column)].grid(True)
148             plt.setp(axes[num_rows.index(row_size),
                ↪  join_columns.index(join_column)].get_xticklabels(),
                ↪  rotation=45)
149             # axes[join_columns.index(join_column)].plot(x, y, label=strategy)
150             # axes[join_columns.index(join_column)].set_title(title)
151             # plt.setp(axes[join_columns.index(join_column)].get_xticklabels(),
                ↪  rotation=45)
152
153 plt.setp(axes, xticks=list(map(lambda x: int(x), num_nodes)))
154
155 handles, labels = axes[-1, -1].get_legend_handles_labels()
156 # fig.legend(handles, labels, loc='upper right')
157 fig.legend(handles, labels, loc='upper center', ncol=5)
158 fig.set_size_inches(68, 30)
159 plt.savefig("nodes.pdf", format="pdf")
160 plt.close()
161
162
163 # ################################################################################
164 # ##### X = selectivity, Y = time. Plot for each #nodes and data size
165 # ################################################################################
166 fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(num_rows),
    ↪  sharey=False, sharex=False)
167 for node in num_nodes:
168     for row_size in num_rows:
169         title = "#nodes: " + str(node) + " Data size: " + "2^" +
            ↪  str(int(np.math.log2(row_size)))
170         sel = [r for r in rows if r['num_nodes'] == node and r['num_rows'] ==
            ↪  str(row_size)]
171         for strategy in strategies:
172             sel_rows = [[int(r['column'].split('_')[0]), (r['throughput'] if
                ↪  THROUGHPUT else r['time'])] for r in sel
173                 if r['strategy'] == strategy and not (r['column'] == 'uniform'
                    ↪  or r['column'] == 'normal') ]
174             # sel_rows = sorted(sel_rows, key=lambda x: int(x[0]),
                ↪  reverse=True)
175             x = [r[0] for r in sel_rows]
176             y = [r[1] for r in sel_rows]
177             axes[num_nodes.index(node), num_rows.index(row_size)].plot(x,y,
                ↪  line_and_color[strategy], label=strategy)
```

```
178             axes[num_nodes.index(node),
                ↪  num_rows.index(row_size)].set_title(title)
179             axes[num_nodes.index(node),
                ↪  num_rows.index(row_size)].tick_params(labelright=True,
                ↪  labelleft=True, labelbottom=True)
180             plt.setp(axes[num_nodes.index(node),
                ↪  num_rows.index(row_size)].get_xticklabels(), rotation=45)
181
182 plt.setp(axes, xticks=sorted([int(r.split('_')[0]) for r in join_columns if r
    ↪  not in ['uniform', 'normal']]))
183
184 handles, labels = axes[-1, -1].get_legend_handles_labels()
185 # fig.legend(handles, labels, loc='upper right')
186 fig.legend(handles, labels, loc='upper center', ncol=5)
187 fig.set_size_inches(58, 30)
188 plt.savefig("selectivity.pdf", format="pdf")
189
190 #
191 # SEPARATE LEGEND FIGURE
192 #
193 # ordered_labels_map = ['data_to_query', 'semi', 'bloom', 'hash_redist',
    ↪  'sort_merge']
194 # ordered_labels = ['data to query', 'semi', 'bloom', 'hash redistribution',
    ↪  'sort merge']
195 ## ordered_labels_map = ['semi', 'bloom']
196 ## ordered_labels = ['semi', 'bloom']
197
198 # ordered_handles = []
199
200 # for ol in ordered_labels_map:
201 #     index = labels.index(ol)
202 #     ordered_handles.append(handles[index])
203
204 # figlegend = pylab.figure(figsize=(7.2,0.4))
205 # figlegend.legend(ordered_handles, ordered_labels, loc='upper center', ncol=5)
206 # figlegend.savefig('legend.pdf', format="pdf")
207
208 plt.close()
209
210 # ##############################################################################
211 # ##### SINGLE NODE PLOT
212 # ##############################################################################
213
```

```
214 # num_rows = [2**18]
215 # join_columns = ['50_50']
216
217 # fig, axes = plt.subplots(nrows=len(num_rows), ncols=len(join_columns),
    ↪ sharey=False)
218 # for row_size in num_rows:
219 #     for join_column in join_columns:
220 #         j_c = [r for r in rows if r['column'] == join_column and
    ↪ r['num_rows'] == str(row_size)]
221 #         for strategy in strategies:
222 #             title = "Data size: " + "2^" + str(int(np.math.log2(row_size))) +
    ↪ ' ' + "Column:" + join_column
223 #             strategy_rows = [[int(r['num_nodes']), (r['throughput'] if
    ↪ THROUGHPUT else r['time']), r['standard_deviation']] for r in j_c if
    ↪ r['strategy'] == strategy]
224 #             strategy_rows = sorted(strategy_rows, key=lambda x: x[0])
225 #             x = [r[0] for r in strategy_rows]
226 #             y = [r[1] for r in strategy_rows]
227
228
229 #             sum_dev = 0.0
230 #             for r in strategy_rows:
231 #                 sum_dev += r[2] / r[1]
232
233 #             print(strategy)
234 #             print((sum_dev / float(len(strategy_rows)))*100.0)
235
236 #             axes.semilogx(x, y, line_and_color[strategy], label=strategy,
    ↪ basex=2)
237 #             axes.tick_params(labelright=False, labelleft=True)
238 #             axes.grid(True)
239 #             axes.set_xlim(2,16)
240 #             axes.set_xlabel('Node Count', labelpad=5)
241 #             axes.set_ylabel('Time spent', labelpad=10)
242 #             # axes.set_ylabel('Throughput\n(Thousand rows/s)', labelpad=10)
243
244 #             axes.set_ylim(bottom=0, top=20)
245
246 #             #integer ticks
247 #             ya = axes.get_yaxis()
248 #             ya.set_major_locator(pylab.MaxNLocator(integer=True))
249
250 #             axes.get_xaxis().set_major_formatter(formatter)
```

```
#             # plt.setp(axes, xticks=size)
#             #plt.setp(axes.get_xticklabels()) #, rotation=45)

# fig.set_size_inches(7, 4)
# plt.setp(axes, xticks=list(map(lambda x: int(x), num_nodes)))
# plt.tight_layout()
# plt.savefig("nodes_2_18_50_latency.pdf", format="pdf")
# plt.close()


# ##############################################################################
# #### SINGLE SELECTIVITY PLOT
# ##############################################################################

num_nodes = ['4']
num_rows = [2**19]

fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(num_rows),
    sharey=False, sharex=False)
for node in num_nodes:
    for row_size in num_rows:
        sel = [r for r in rows if r['num_nodes'] == node and r['num_rows'] ==
            str(row_size)]
        for strategy in strategies:
            sel_rows = [[int(r['column'].split('_')[0]), (r['throughput'] if
                THROUGHPUT else r['time'])] for r in sel
                if r['strategy'] == strategy and not (r['column'] == 'uniform'
                    or r['column'] == 'normal') ]
            # sel_rows = sorted(sel_rows, key=lambda x: int(x[0]),
                reverse=True)
            x = [r[0] for r in sel_rows]
            y = [r[1] for r in sel_rows]
            axes.plot(x,y, line_and_color[strategy], label=strategy)
            axes.tick_params(labelright=False, labelleft=True,
                labelbottom=True)
            axes.set_xlabel('Selectivity (%)', labelpad=5)
            # axes.set_ylabel('Throughput\n(Thousand rows/s)', labelpad=10)
            axes.set_ylabel('Time spent', labelpad=10)
            plt.setp(axes.get_xticklabels(), rotation=45)

            axes.grid(True)
            axes.set_xlim(10, 90)
            axes.set_ylim(0, 50)
```

```
288             ya = axes.get_yaxis()
289             ya.set_major_locator(pylab.MaxNLocator(integer=True))
290 
291 fig.set_size_inches(7, 5)
292 plt.setp(axes, xticks=sorted([int(r.split('_')[0]) for r in join_columns if r
    ↪  not in ['uniform', 'normal', '100_100']]))
293 plt.tight_layout()
294 plt.savefig("selectivity_2_19_4_nodes_atum_y_axis.pdf", format="pdf")
295 plt.close()
296 ###------------
297 
298 
299 # #
    ↪  ##############################################################################
300 # # #### SINGLE DATASIZE AND COLUMN PLOT
301 # #
    ↪  ##############################################################################
302 
303 # join_columns = ['50_50']
304 # num_nodes = ['16']
305 # # # fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(join_columns),
    ↪  sharey=True)
306 # fig, axes = plt.subplots(nrows=len(num_nodes), ncols=len(join_columns),
    ↪  sharey=False)
307 # for nodes in num_nodes:
308 #     for join_column in join_columns:
309 #         j_c = [r for r in rows if r['column'] == join_column and
    ↪  r['num_nodes'] == nodes]
310 #         for strategy in strategies:
311 #             title = "Nodes: " + nodes + ' ' + "Column:" + join_column
312 #             strategy_rows = [[int(r['num_rows']), (r['throughput'] if
    ↪  THROUGHPUT else r['time'])] for r in j_c if r['strategy'] == strategy]
313 #             strategy_rows = sorted(strategy_rows, key=lambda x: x[0])
314 #             x = [r[0] for r in strategy_rows]
315 #             y = [r[1] for r in strategy_rows]
316 #             axes.plot(x, y, line_and_color[strategy], label=strategy) #,
    ↪  basex=2)
317 #             # axes.set_title("Nodes: " + nodes + ' ' + "Column:" +
    ↪  join_column)
318 #             axes.grid(True)
319 #             axes.set_xlim(2**14, 2**20)
320 #             axes.set_ylim(bottom=0, top=77)
321 #             axes.set_xlabel('Number of rows', labelpad=5)
```

```
#             axes.set_ylabel('Time spent', labelpad=10)
#             axes.set_xticks(x)
#             axes.tick_params(labelright=False, labelleft=True)
#             axes.get_xaxis().set_major_formatter(formatter2)
#             plt.setp(axes.get_xticklabels(), rotation=45)

# fig.set_size_inches(9, 5)
# plt.tight_layout()
# plt.savefig("data_size_50_16_nodes.pdf", format="pdf")
# plt.close()
```

`test_system/optimization_eval/main.py`

```
import matplotlib.pyplot as plt

# plt.rc('font',family='Verdana')
plt.rcParams["font.family"] = "Verdana"
plt.rcParams["font.size"] = 20

strats_x = [1, 3, 5, 7, 9]
strats = ['data_to_query', 'semi', 'bloom', 'hash_redist', 'sort_merge']

before = [0.152 , 0.154 ,   0.1 , 0.162 , 0.076]
after = [0.082 , 0.092 , 0.074 , 0.118 , 0.078]

improvements = []

for i in range(len(before)):
    improvements.append((-1) * ((after[i] - before[i]) / before[i] * 100))

plt.bar([x-0.3 for x in strats_x], before, color='#306998', width=0.5)
plt.bar([x+0.3 for x in strats_x], after, color='#ffd43b', width=0.5)
plt.ylabel('Duration', labelpad=10)
# plt.xlabel('Strategy', labelpad=10)
# plt.bar(strats, improvements, color='#ffe873', width=0.5)


plt.xticks(strats_x, strats)

print(improvements)

plt.legend(['no indices', 'w/ indices'])
```

```

# plt.tight_layout()

plt.savefig("indices_optimization_graph.pdf", format='pdf')


# after index
#| avg       | 0.082 | 0.092 | 0.074 | 0.118 | 0.078 |


# before
#| avg       | 0.152 | 0.154 |   0.1 | 0.162 | 0.076 |

plt.show()
```

test_system/optimization_eval/seq_par_insert.py

```
# import numpy as np
# import matplotlib as mpl
import matplotlib.pyplot as plt

plt.rcParams["font.family"] = "Verdana"
plt.rcParams["font.size"] = 16

sequential = []
seq_dtq = [0.3485, 0.2811, 0.2737, 0.3000, 0.2680]
seq_dtq = sum(seq_dtq)/len(seq_dtq)

parallel = []
par_dtq = [0.2450, 0.2369, 0.2112, 0.2088, 0.2321]
par_dtq = sum(par_dtq)/len(par_dtq)

x = [1,2]
# plt.bar(x, seq_dtq, color='b')
# plt.bar(x, par_dtq, color='y')
plt.bar(x, [seq_dtq, par_dtq], width=0.5)
plt.xticks(x, ('Sequential', 'Parallel'))
# plt.xticks(x, ("Data-to-query"))
plt.xlim(0.5, 2.5)
plt.ylabel("Time (s)")
plt.tight_layout()
print((1 - par_dtq/seq_dtq)*100)
```

```
print((par_dtq - seq_dtq)/seq_dtq*100)
print((seq_dtq/par_dtq)-1)
print(seq_dtq, par_dtq)
plt.show()
# plt.savefig("./asdf.pdf", format="pdf")
```

`test_system/bloom_false_positive/main.py`

```
import matplotlib.pyplot as plt
import numpy as np

# plt.rc('font',family='Verdana')
plt.rcParams["font.family"] = "Verdana"
plt.rcParams["font.size"] = 20

# data_sizes = [16384, 32768, 65536, 131072, 262144, 524288]
data_sizes = ['16384', '32768', '65536', '131072', '262144', '524288']

semi = [167, 176, 183, 196, 203, 208]
bloom = [204, 178, 191, 197, 235, 276]


false_positive_rate = []

for i in range(6):
    false_positive_rate.append((float(bloom[i]-semi[i]) / float(semi[i]) *
    ↪  100.0))

plt.bar(np.arange(6), false_positive_rate)

labs = ["$2^{" + str(int(np.math.log2(int(d)))) + "}$" for d in data_sizes]

plt.xticks(np.arange(6), labs, rotation='45')
plt.xlabel("Data size", labelpad=10)
plt.ylabel("Extra rows (%)", labelpad=10)
plt.show()

print(2**14)
# ** 16384
# bloom = 204
# semi = 167

```

```
34 # ** 32768
35 # bloom = 178
36 # semi = 176
37
38 # ** 65536
39 # bloom = 191
40 # semi = 183
41
42 # ** 131072
43 # bloom = 197
44 # semi = 196
45
46 # ** 262144
47 # bloom = 235
48 # semi = 203
49
50 # ** 524288
51 # bloom = 276
52 # semi = 208
```

Håvard Ola Eggen & Kristian Andersen Hole

An evaluation of join-strategies in a distributed MySQL plugin architecture

NTNU
Norwegian University of
Science and Technology