Improving OLAP Workload Performance in Apache Ignite

by

Mark Dodds

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Mark Dodds 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Apache Ignite is an open-source system that is used to process database workloads. It utilizes Apache Calcite as the underlying query engine to parse SQL queries into relational algebra operators that are passed to the Ignite execution engine. This thesis investigates the performance of online analytical processing (OLAP) workloads with varying memory and data distribution settings using Apache Ignite with Apache Calcite. From empirical studies, practical strategies to decrease response times for OLAP queries are designed and implemented in Apache Ignite. Through studies using the TPC-H benchmark, it is demonstrated that each strategy yields performance improvements across multiple queries, and when all strategies are enabled, average performance improves generally for all queries in the workload.

Acknowledgements

I would like to thank my supervisor, Professor Khuzaima Daudjee, for his support and guidance over the past two years. I would also like to extend my appreciation to my committee members for agreeing to read and review my thesis.

I must thank my partner, my parents, my brother, and my sister (in-law) for their support and encouragement throughout this entire process.

Finally, I would like to thank Nik, for starting me on this journey over a decade ago and giving a much-needed nudge over the finish line.

Dedication

This is dedicated to the one I love. Without her boundless support and unwavering faith in me, I would not have succeeded.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

1 Introduction 1

2 Related Work 3

3 Query Processing and Optimization 6

3.1 Apache Calcite 6

3.2 Apache Ignite 8

3.2.1 Optimizing the Query Tree 9

3.2.2 Distribution Trait 10

3.2.3 Fragmentation 12

4 Improving Ignite's Query Planner **16**
4.1 Query Planner . . . 16
4.2 Cost Model . . . 17
4.3 Planner Exploration Efficiency . . . 19

5 Improving Query Execution **21**
5.1 Join Operation Optimizations . . . 21
5.1.1 Fully Distributed Joins . . . 21
5.1.2 Hash Join . . . 22
5.2 Multi Threaded Execution Plans . . . 22
5.2.1 Variant Fragment Creation . . . 23
5.2.2 Variant Fragment Execution . . . 24
5.3 Join Condition Simplification . . . 25

6 Other Improvements **26**
6.1 Apache Ignite Improvements . . . 26
6.2 Apache Calcite Improvements . . . 27

7 Performance Evaluation **28**
7.1 TPC-H Per Query Response Time . . . 29
7.1.1 Query Planner Changes . . . 29
7.1.2 Join Optimizations . . . 31
7.1.3 Multithreading . . . 33
7.1.4 Overall Performance . . . 35
7.2 TPC-H Average Query Latency . . . 35

8 Conclusions and Future Work **38**
8.1 Conclusions . . . 38
8.2 Future Work . . . 39

References 40

APPENDICES 44

A Modified Cost Model Calculations 45

A.1 CPU Cost Comparison of Hash Join and Merge Join 45

B TPC-H Results 48

B.1 TPC-H Per Query Results 48

B.2 TPC-H Average Query Latency Results 51

List of Figures

3.1 Sample Schema and Join Query 7
3.2 Unoptimized Query Tree of Query A in Figure 3.1 7
3.3 Query tree after applying filter pushdown rules to Figure 3.2 9
3.4 Enforcing distribution trait satisfaction on Figure 3.3 11
3.5 A hash-distributed join of **sales** and **employee** with data sites A and B . 12
3.6 Fragments for executing Query A from Figure 3.1 using a single-site join . 14
3.7 Execution Flow of an SQL Query 14

7.1 Incremental Performance from Query Planner Stabilization 30
7.2 Incremental Performance from Join Optimizations 32
7.3 Incremental Performance from Multithreading 33
7.4 Overall Performance Increase 34
7.5 Average Query Latency for 4 Sites 35
7.6 Average Query Latency for 8 Sites 36

List of Tables

3.1 Target Distribution Satisfaction Matrix 10

3.2 Join Operator Distribution Mappings 11

B.1 Per Query Response Time: Baseline 49

B.2 Per Query Response Time: Query Planner Changes 49

B.3 Per Query Response Time: Query Planner Changes and Join Optimizations 50

B.4 Per Query Response Time: Query Planner Changes, Join Optimizations, and Multithreading 50

B.5 Average Query Latency: Baseline (seconds) 51

B.6 Average Query Latency: Query Planner Changes and Join Optimizations (seconds) 51

B.7 Average Query Latency: Query Planner Changes, Join Optimizations, and Multithreading (seconds) 51

Chapter 1

Introduction

At its core, Apache Ignite is a distributed storage system that stores information using in-memory key-value caches, which are interacted with through its open-source API. These caches can be partitioned and distributed across multiple data sites to achieve massive horizontal scaling and dramatically improve processing capabilities. Ignite also provides an ANSI-99-compliant SQL interface to support more traditional applications using the standard JDBC API. To allow the JDBC API to interact with Ignite's key-value caches, Ignite implements a database engine using Apache Calcite (Ignite+Calcite)[1]. Apache Calcite is a high-performance dynamic data management framework that provides many functionalities of a typical database management system [13]. It contains an out-of-the-box SQL parser, validator, and optimizer but defers the functions of data storage, metadata storage, and data processing to the implementer. The result is a highly extensible framework that handles query parsing and optimization while remaining completely agnostic to storage and processing implementations.

Apache Ignite and Apache Calcite are open-source projects supported by a community of developers both within and outside of the Apache Software Foundation. Being open-source, these projects come with a level of assumed risk and responsibility of the user not shared by commercial projects. This can allow development teams to focus on system improvements and feature enhancements without being limited by commercial support obligations. However, one possible trade-off is the lack of testing in complex environments during the development process. Instead, development teams must rely on users deploying the software in real systems to test and report any issues they uncover. While the user

[1] https://ignite.apache.org/docs/latest/SQL/sql-calcite

base does include several major commercial vendors[2], most employ Ignite as a data storage system or to process simple queries. Currently, no published work exists detailing the use of Ignite as an engine for complex queries on distributed data.

Consequently, this research began with an investigation into the processing capabilities of OLAP workloads using Ignite+Calcite. Multiple issues arose quickly when Ignite+Calcite was tested with the TPC-H [22] test bench, an industry standard for OLAP workloads. Some queries that selected data from multiple tables (i.e., Q5 and Q7) did not complete execution most of the time, primarily due to inefficient join operations. When queries were executed work was often isolated to a single CPU core, leaving other cores idling. One query (Q2) simply failed to generate an execution plan. This yielded the following problems to address as the core of this research:

1. The query planning process is unstable and can fail to generate execution plans.
2. Execution plans are often not fully optimized, resulting in long execution times, particularly for join operations.
3. Execution plans do not fully utilize modern CPUs with multiple cores.

To address each of these issues, this thesis presents three distinct contributions:

1. The implementation of Apache Calcite within Ignite was investigated, and multiple areas where Calcite was incorrectly configured were discovered and resolved.
2. New strategies for executing join operations were implemented, including a new join algorithm (hash-join).
3. Support for multi-threaded execution plans was added to increase CPU utilization when processing queries.

This thesis is structured as follows. Chapter 2 reviews prior work related to this topic. Chapter 3 details the current querying process in Ignite+Calcite. Chapter 4 describes the improvements made to the query planner implementation in Ignite+Calcite. Chapter 5 explains the efforts to improve query execution. Chapter 6 details other minor improvements made to Ignite+Calcite. Chapter 7 presents the performance evaluation results. Chapter 8.1 states conclusions from this research. Chapter 8.2 proposes ideas for future work.

[2] https://ignite.apache.org/use-cases/provenusecases.html

Chapter 2

Related Work

Query optimization has been a heavily researched topic as it is a core feature of any database management system. Jarke and Koch [16] provide a concrete base by presenting a framework for general query evaluation using relational calculus. They focused on efficient methods for processing and optimizing unanticipated queries in centralized database systems, which is still relevant in modern business environments. They also touch on some aspects of query optimization for distributed databases like network structure and data distribution strategies, but it is not the main focus. Around the same time, Bernstein et al. [5] discussed methods for optimizing relational queries in the then-newly created distributed database system SDD-1 [24]. Bernstein et al. describe a query processing strategy that includes cost and benefit estimations to evaluate possible execution strategies and an algorithm for optimizing relational operations.

Almost two decades later, Chaudhuri [8] recapped much of the new work in query optimization, specifically for SQL queries in relational database systems using relational algebra. This work covers essential topics in query optimization including defining search spaces, cost estimation techniques, and enumeration algorithms. Meanwhile, Kossman [18] presented a "textbook architecture for distributed query processing" and "a series of techniques that are particularly useful for distributed database systems." Various topics are covered, including join algorithms, communication reduction techniques, caching techniques, and database architectures for distributed processing. Graefe also provides many contributions in [14]. This work comprises almost every aspect of query evaluation for large databases ranging from system architecture, sorting, and disk access, to binary matching algorithms, parallel query execution, and nonstandard query processing. Finally, Özsu and Valduriez [32] provide a rich repository of information on distributed database systems that has been continually refreshed over the past three decades. The section on

distributed query processing covers data localization techniques, distributed join ordering, costing models, and various query optimization techniques.

Since join operations are typically among the most resource-intensive processes in a database system, a large body of research is also dedicated to them. In [19], Mishra and Eich provide a comprehensive overview of join operations. The basics of join processing like implementations of specific join algorithms and join processing in distributed environments are covered. More advanced topics are also covered such as specialized data structures for join executions, hardware support for specific join algorithms, and selectivity factor calculations. In [10] DeWitt et. al. research changes necessary to support large in-memory databases. Much of this work includes evaluating four join algorithms (three hash-based, one merge-sort-based) for use in fully in-memory systems.

Following this, much research has focused on analyzing and perfecting the performance of these techniques under specific settings and as hardware capabilities have grown. Hash and sort-merge join algorithms under varying memory and CPU constraints have been the focus of many studies [17, 26, 2, 3]. In addition, research into parallel join processing algorithms has increased significantly as modern CPUs have increased their parallel processing capabilities. Schneider and DeWitt [25] present performance results for parallelized versions of the four join algorithms originally evaluated by DeWitt et. al. in [10]. Patel and DeWitt extend this in [20] by exploring spatial join algorithms and evaluating the effectiveness of parallelizing them. Blanas et. al. [7] perform an experimental analysis on join algorithms under the MapReduce paradigm that is widely used in Big Data processing applications. They include implementations of join strategies in MapReduce and an experimental analysis using the Apache Hadoop framework.

Despite claims stating use by industry giants like Microsoft, Netflix, IBM, and Yahoo[1], there is little information about these specific use cases. In [27], they describe implementing Apache Ignite for a simple e-commerce platform but do not detail any performance evaluations. [21] and [1] again show implementations of Apache Ignite in their respective systems but do not detail performance.

Stan et. al. [29] present a performance comparison of Apache Ignite and Apache Spark through the lens of Big Data processing. However, Apache Ignite is used only as a distributed processing engine for two generic data processing algorithms, not as a database management system. In [6] Bhuiyan et. al. provide a large repository of technical information about Ignite. They cover many practical use cases and implementation details from simple data storage to distributed computing and event streaming. However, their coverage of Ignite as a DBMS is not in-depth, focusing mainly on database setup, system

[1] https://ignite.apache.org/use-cases/provenusecases.html

architectures, and performance tuning. Furthermore, there is no information on Apache Calcite as it was published before the release of Ignite+Calcite functionality, and no performance evaluations of Apache Ignite. There was no material covering OLAP workload performance using Apache Ignite with Apache Calcite.

Chapter 3

Query Processing and Optimization

3.1 Apache Calcite

Relational algebra [9] operators are the main building blocks for Calcite. They can either be *logical* operators that are agnostic to the execution environment or *physical* operators that have *traits*. Traits are physical properties associated with an operator that describe some aspect of the execution but do not change the logical expression itself. They enforce specific properties for an execution plan (such as the collation of a *sort* operator) and can be varied during the planning process to explore alternative execution plans. Calcite implements a wide range of logical operators, but physical operators and traits are left to the implementer.

The main entry point is the Calcite SQL parser. It consumes an SQL string and returns a tree-like structure representing the query (a query tree) that contains the operators, their properties, and the required data flow between them. Figure 3.2 shows the resulting query tree when Query A in Figure 3.1 is parsed by Calcite. Each operator is responsible for performing its operation and passing the result up the tree to its parent, with leaf nodes being the starting point for the data flow. Since Calcite does not inherently have any context about the expression it evaluates, it relies on the implementer to provide this context via *provider hooks*. These provider hooks consume a *provider function* that produces metadata required for a Calcite procedure, such as schema definitions, table statistics, or estimation algorithms. When a procedure requires context-specific information about a table (e.g., table cardinality), it calls the requisite provider function to retrieve the information, defaulting to no-operation (NO-OP) implementations if one is not provided. The NO-OP implementations can prevent catastrophic errors but can also result in inefficient

Table Schema:

```
employee (int id, primary key(id));
sales (int order_id,
       int emp_id,
       primary key(order_id));
```

Query A:

```
SELECT * FROM employee
INNER JOIN sales ON employee.id = sales.emp_id
WHERE employee.id = 10;
```

Figure 3.1: Sample Schema and Join Query

query plans being generated. Therefore, care must be taken to ensure providers have valid implementations.

When an SQL expression is converted into a query tree, it is semantically equivalent to the original expression but rarely efficient and thus must be optimized. Optimizing a query tree means swapping or re-ordering operators to reduce the execution time. Calcite uses *rules* to define how the swaps and re-orderings occur. A *rule* is an operation that consumes a single operator and produces a set of operators that are semantically equivalent to the original but potentially more efficient. Each rule also has a predicate indicating which operator it can operate on. Rules can either be *logical*, which modifies the relational algebra of the query tree, or *physical*, which determines the implementation and execution

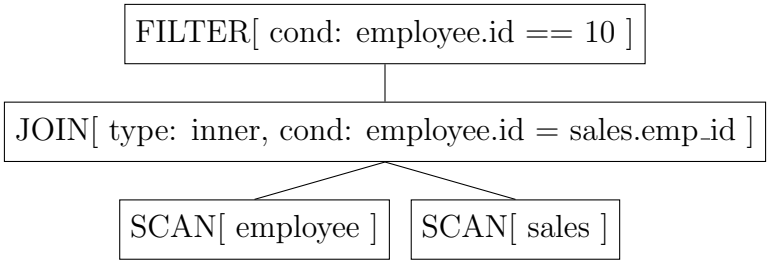


Figure 3.2: Unoptimized Query Tree of Query A in Figure 3.1

of the query tree. Calcite provides many rules[1] that the implementer can enable depending on the type of query trees their application can execute.

To execute rules, Calcite provides two *planner engines*. A planner engine consumes a set of rules and applies them repeatedly until it reaches a specified objective (or is forcibly stopped). Once the objective is reached, it returns a semantically equivalent query tree optimized according to the provided rule set. The first is an exhaustive planner [4] called the *HepPlanner*, which performs logical optimizations via query rewrites. It consumes a list of optimization rules and continuously applies them until it produces an expression that is no longer altered by any rules [4].

The second planner engine is a cost-based planner [4] called the *VolcanoPlanner* that uses a dynamic programming algorithm similar to [15]. It repeatedly applies the input rules to reduce the overall cost of the query tree until it reaches a configurable termination point [4]. Each operator is responsible for determining its cost $C(O)$ by using query metadata and a cost model that the implementer injects through provider functions. Calcite does not apply any implementation constraints on the cost model; it simply defaults to NO-OP implementations if an implementation for an operator does not exist. The cost C of a query tree QT is the sum of the individual costs of each operator O in the tree (Equation 3.1).

$$C(QT) = \sum_{O \in QT} C(O) \tag{3.1}$$

where QT is a valid query tree

3.2 Apache Ignite

Before Ignite can optimize any queries, it must provide Calcite with the required metadata, estimation algorithms, and cost model. Ignite already tracks metadata related to the data it is storing (schemas, cardinality, etc.), so it simply re-formats and serves this information to Calcite as needed. Estimation algorithms (e.g., join result size) are implemented by overriding Calcite's default implementations with custom algorithms. Some of these algorithms also use the metadata Ignite collects (e.g., cardinality of distinct column values). Ignite's cost model contains four parameters, each representing some of the overall costs of executing an operation: CPU, Memory, IO, and Network. The CPU parameter approximates the number of operations performed on tuples (e.g., reading the fields). The

[1] https://calcite.apache.org/javadocAggregate/org/apache/calcite/plan/RelRule.html

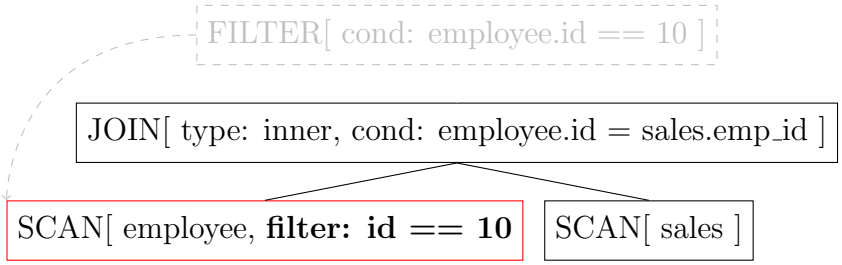


Figure 3.3: Query tree after applying filter pushdown rules to Figure 3.2

memory and network parameters approximate the number of bytes used to store and send data, respectively. The IO parameter is always 0 since Ignite is an in-memory system. Each operator is responsible for implementing a *getSelfCost* method, which returns a cost object consisting of these four parameters. The equal-weighted sum of the four parameters represents the cost of a specific operator (Equation 3.2).

$$C(O) = O_{\text{CPU}} + O_{\text{Memory}} + O_{\text{IO}} + O_{\text{Network}} \tag{3.2}$$

3.2.1 Optimizing the Query Tree

Once Ignite has a valid query tree from the Calcite SQL parser, it employs Calcite planners to optimize the query tree using a two-stage approach. The first stage employs multiple HepPlanners, each using a different set of logical rules: one with three rules, another with seven rules, and the third with five rules. These rules are standard optimizations Calcite provides, for example, pushing `filter` operators down the query tree (see Figure 3.3). The objective of this first stage is to apply transformations that are guaranteed to improve the execution performance of the query.

The second stage is the main optimization phase. In this stage, Ignite uses a VolcanoPlanner with a set of 52 rules that are a mix of logical rules provided by Calcite, logical rules written for Ignite, and physical rules specific to the Ignite execution engine. The VolcanoPlanner uses these rules to generate alternate query trees, compare their costs, and return the query tree with the lowest overall cost according to the cost model. An essential part of the second stage is optimizing the physical traits of each operator in the query tree to create the most efficient physical execution plan possible.

3.2.2 Distribution Trait

The distribution trait has the most impact on the cost of a query tree, as it determines at which processing sites an operator will be executed. There are three values for this trait that Ignite uses when optimizing query plans:

1. *Single*: the operator will be executed at a single site
2. *Broadcast*: the operator will be executed at all processing sites
3. *Hash*: the operator will be executed at a subset of sites determined by a hash function

Each operator always has a distribution trait value. Since operators are inputs (sources) for other operators, the distribution of the main operator (the target) must be compatible with the distribution of all its sources for data to flow between them. This compatibility is known as *satisfaction* (see Table 3.1). If the source distribution does not satisfy the target distribution, an *exchange* operator is inserted between the target and source operators. The exchange will take data from its source and send it to one or more targets over the network, acting as an intermediary between two operators with incompatible distributions.

Using Figure 3.4 as an example, there are two relations `employee` and `sales`, which are partitioned on their primary key using a hash function, yielding a *hash* distribution. The VolcanoPlanner is attempting to create an execution plan where the `join` is processed at one site. Thus, it assigns a *single* distribution type to the `join` operator. Since a hash distribution source does not satisfy a *single* distribution target, an `exchange` is added between the operators. This `exchange` executes on the same sites as the `scan`, sending its result tuples to the `join` for processing. Calcite uses this trait satisfaction as a proxy for correctness, allowing the implementer to define what constitutes a valid execution plan. The VolcanoPlanner is then free to explore different execution plans, knowing that if trait satisfaction is maintained, so is correctness.

Table 3.1: Target Distribution Satisfaction Matrix

Source	Single	Broadcast	Hash
Single	Yes	No	No
Broadcast	Yes	Yes	Yes
Hash	No	Yes*	Yes*

* Only if the hash function produces a super-set of the destination sites

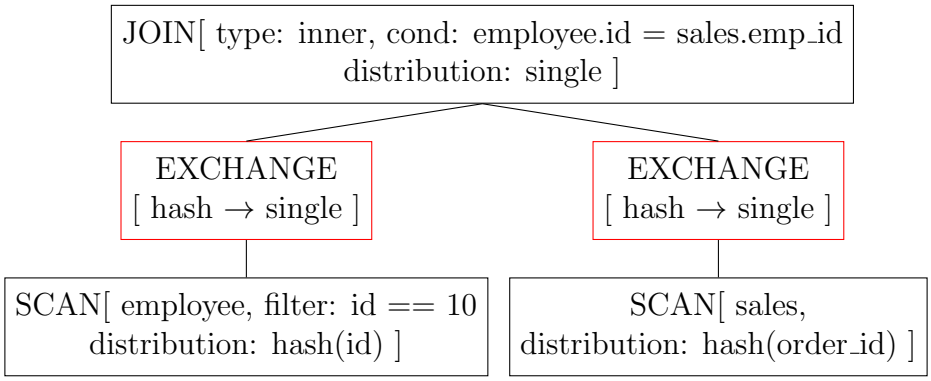


Figure 3.4: Enforcing distribution trait satisfaction on Figure 3.3

This exploration is facilitated by the *deriveDistribution* method, which every operator can implement. This method generates a list of distribution mappings, each containing a target distribution for the current operator and a corresponding set of required source distributions, one for each of the operator's sources. Most operators do not implement this method as they take on the distribution of their source(s) (e.g., the `sort` operator takes on the *single* distribution from the `join` operator in Figure 3.4). The only operators that implement this method are the physical join operators: `merge-join` and `nested-loop-join`. Table 3.2 shows the distribution mappings they generate. The first mapping, *single*, is where all data is shipped to a single processing site. This is the most frequently generated plan because it has no restrictions on the join properties (e.g., the join condition). The second mapping, *broadcast*, is a fully replicated join, where all sources are sent to all sites to perform the complete join. This approach does not restrict join properties but incurs a high cost for transmitting the same data to multiple sites. Thus, it is rarely used and is typically reserved for scenarios where the join results are inputs to another expensive operation that can be distributed among all processing sites.

The third mapping, *hash*, is for a distributed equi-join [28] where the left source must

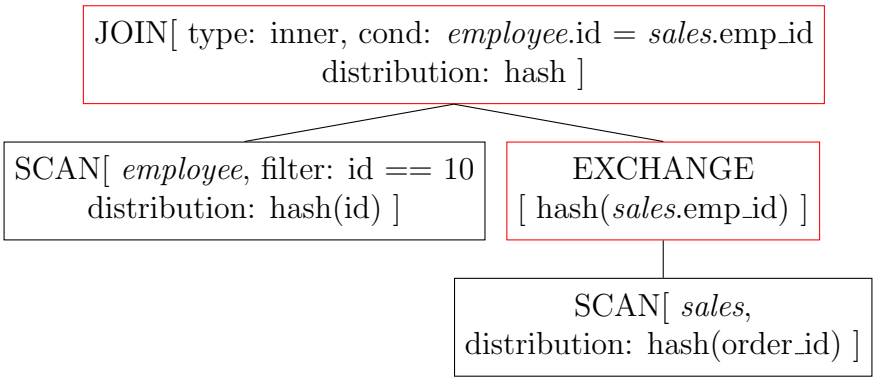
Table 3.2: Join Operator Distribution Mappings

	Join Operator	Left Source	Right Source
1	Single	Single	Single
2	Broadcast	Broadcast	Broadcast
3	Hash[L]	Hash[L]	Hash[R → L]

be partitioned on a key-set k using a hash function $h(k)$ and the key-set for one side of the equi-join condition j_α is a super-set of k (i.e., $j_\alpha \subseteq k$). The right source uses the same hash function $h(k)$ on the key-set for the opposite side of the equi-join condition j_β to direct data to the appropriate sites, allowing each site to process its subset of the data in parallel. Using Figure 3.5 as an example, the *employee* relation is partitioned on its `id` column (i.e., *h(id)* determines a tuples storage site) and the *sales* relation is partitioned on its `order_id` column (i.e., *h(order_id)* determines a tuples storage site). This means $k = \{\text{employee.id}\}$, $j_\alpha = \{\text{employee.id}\}$ and $j_\beta = \{\text{sales.emp_id}\}$. When the *sales* relation is scanned and exchanged, each tuple $s \in sales$ has $h(s_{emp_id})$ calculated. Each tuple s is then sent to site $h(s_{emp_id})$ as that is the only site where s could find a join match. Using examples from Figure 3.5, the first two sales ($order_id = 100$ and $order_id = 101$) are sent to A and the third ($order_id = 102$) is sent to B.

3.2.3 Fragmentation

Once a query tree has been fully optimized, Ignite must convert it into an execution plan consisting of one or more *fragments*. A fragment represents a distinct segment of the query tree slated for execution at a specific set of processing sites. Each fragment contains a *root*



employee

id	*h(id)*
1	A
2	B
3	B

sales

order_id	emp_id	*h(order_id)*	*h(emp_id)*
100	1	B	A
101	1	A	A
102	2	A	B

Figure 3.5: A hash-distributed join of **sales** and **employee** with data sites A and B

Algorithm 1 Execution Plan Fragmentation

```
Require: rootNode ← a root operator in a query tree
  procedure FRAGMENT(node):
      current ← node
      if node is exchange then
          fragments.add(Fragment(root = node))
          sender ← new sender
          sender.children ← node.children
          node.children ← [new receiver]
          current ← sender
      end if
      for child ∈ current.children do
          FRAGMENT(child)
      end for
  end procedure
  global fragments ← []
  fragments.add(Fragment(root = rootNode))
  FRAGMENT(rootNode)
  return fragments
```

operator, serving as the starting point for the subsection of the query tree it is assigned to execute. While traversing the query tree depth-first from the root, if an `exchange` operator is encountered, the fragment undergoes *splitting*. This entails replacing the `exchange` with two operators: a `receiver` and a `sender`. A `sender` sends results from its child operator over the network to a corresponding `receiver` in another fragment. The `receiver` becomes the leaf in the current fragment's subtree, while the `sender` becomes the new parent of the children of the original `exchange` and the root of a new fragment. This process iterates until no `exchange` operators remain in the overall tree. The outcome is a collection of fragments, each containing a subsection of the original query tree, that can be entirely executed at one processing site.

Data enters through the leaf operators and leaves through the root operator. The fragment containing the root of the original query tree is responsible for collecting and sending the final results to the end user and is called the *root fragment*. After the fragments have been generated, they are sent to their respective processing sites for execution. The distribution traits from the operators in each fragment determine which processing sites a fragment will be sent to. Each fragment is executed in a dedicated thread, using the `sender` at its root to send the result to the next fragment (or to the end user in the case

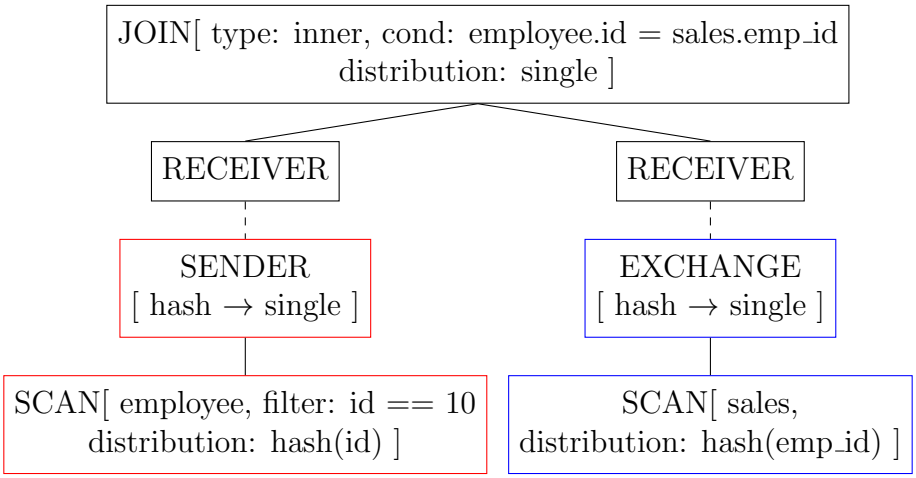


Figure 3.6: Fragments for executing Query A from Figure 3.1 using a single-site join

of the root fragment). This thread-isolated execution allows independent fragments to run concurrently using a thread pool without any interaction between them.

Algorithm 1 shows this process in detail. Figure 3.6 shows a possible set of fragments for executing Query A from Figure 3.1 as a single-site join. Each colour group (shown by red, blue, and black boxes) represents a fragment to be executed. The red group will be

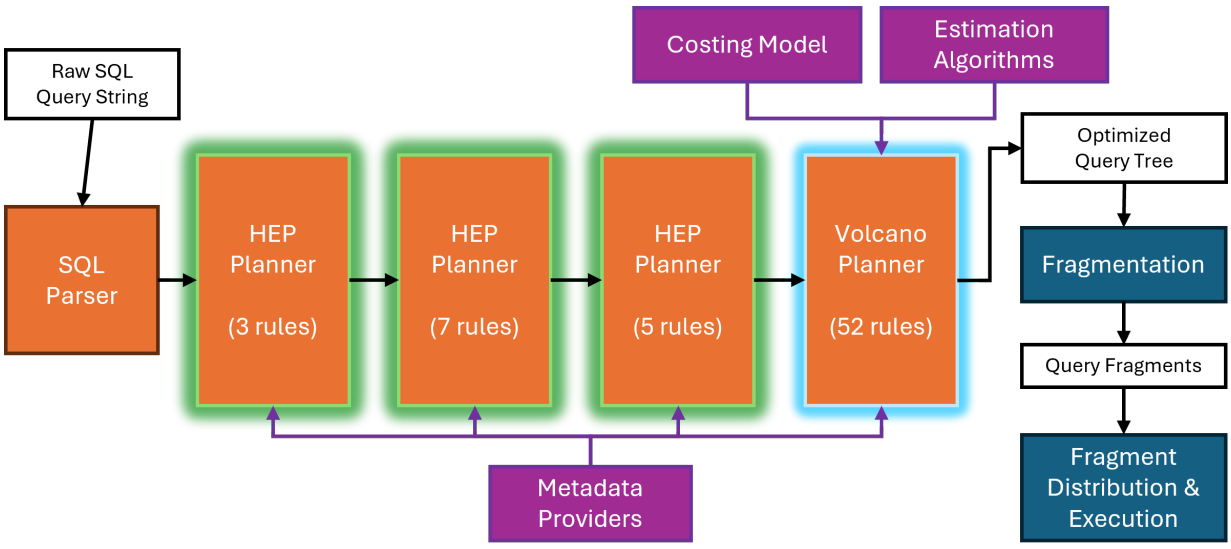


Figure 3.7: Execution Flow of an SQL Query

executed at all data sites housing a segment of the `employee` relation. The blue group will be executed at all data sites housing a segment of the `sales` relation. The black group will be executed at the site that received the original request and will send results back to the requester as they materialize.

Figure 3.7 shows the architectural flow when executing an SQL query. Blue boxes represent processes in Ignite, orange boxes represent processes in Calcite that have been customized for Ignite, purple boxes represent inputs to Calcite processes (via provider functions), and black boxes represent the information states. The boxes with green outlines constitute the first optimization stage, and the box with the blue outline constitutes the second optimization stage.

Chapter 4

Improving Ignite's Query Planner

The first goal of this research was to improve Ignite's query planning process. This meant stabilizing the query planner to ensure it consistently generated fully optimized execution plans for all queries.

4.1 Query Planner

When beginning this research, multiple problems were encountered in the planning code that required attention. In the first planning stage, the HepPlanner was missing an essential rule called `FILTER_CORRELATE`[1] which pushes a `filter` down past a logical correlation in the query tree. Without this rule, `filter` operations that could be executed close to the leaves of the query tree would instead be executed near the root. This meant many operators in the middle of the tree performing unnecessary work on tuples that should have been filtered out much earlier.

Next, the `exchange` operator had a programming error, which can result in incorrect costs being generated. A penalty was supposed to be applied when an `exchange` sends data to more than one site. However, the constant used in the functional check shared its name with a constant from another class, leading to the incorrect constant being used and no penalty being applied. As a result, these `exchange` operators would have an identical cost to an `exchange` operator that sent data to a single site.

[1] https://calcite.apache.org/javadocAggregate/org/apache/calcite/rel/rules/FilterCorrelateRule.html

Finally, the join result size estimation algorithm had an edge case where, if the estimated cardinality of either join input was very small, the estimated join result cardinality would be 1 every time. When this join was nested with another join (e.g., a join with another join as an input), each join in the chain would have an estimated result cardinality of 1. As a result, there would be a chain of joins, each with at least one source having an estimated cardinality of one. Then, when the planner converted the generic join operation to a specific join method, it chose the most efficient way to perform an $Nx1$ join, which is a nested-loop join. This produced query plans with a chain of `nested-loop-join` operators predicted to be $Nx1$ joins. However, when executed, these joins were NxM with $M >> 1$. The algorithm was unintuitive, and its origin was unclear, so an algorithm with theoretical backing was desired. The requirement for this algorithm was to accurately estimate the result size of an equi-join operation. The equi-join restriction arises because, in Ignite, any non-equi-join operation is limited to using a `nested-loop-join` (although `merge-join` can, in theory, handle inequality conditions, this is not implemented). In [30] Equation 4.1 is proposed as an estimate of the result size for an equi-join $A \bowtie B$ where d_A and d_B are the number of distinct values in A and B, respectively.

$$|(A \bowtie B)| = \frac{|A| \times |B|}{\max(d_A, d_B)} \tag{4.1}$$

In [23], Rosenthal proved that Equation 4.1 is correct when at least one of the join columns is uniformly distributed. While it cannot be definitively stated this restriction holds for all joins in Ignite, it is believed to be a fair assumption. Furthermore, empirical testing showed estimations from Equation 4.1 were as good or better compared to the original join estimation algorithm and did not suffer from the issue above.

4.2 Cost Model

As shown in Equation 3.2, the cost of an operator is the sum of four components: CPU, Memory, IO, and Network, with each component having an equal impact on the overall cost. Costing algorithms vary between different operators based on their requirements, but they all use the same constants to have comparable costs. For example, Equation 4.2 shows the cost of a `sort` operator for a relation A with cardinality $|A|$ and width (i.e., column count) $deg(A)$.

$$\begin{aligned} O_{\text{CPU}} &= |A| * RPTC \ + \ |A| * \log(|A|) * RCC \\ O_{\text{Memory}} &= |A| * deg(A) * AFS \end{aligned} \tag{4.2}$$

Since a `sort` operator does not perform IO or network operations, those components are zero in its cost model. The memory cost is the product of the relation cardinality, the relation width ($deg(A)$), and a constant approximating the average field size (AFS) in bytes. The CPU cost is the sum of two estimations: the cost of passing A through the operator and the cost of sorting A. The cost of passing A through the operator is the cardinality $|A|$ multiplied by a constant approximating the CPU work required to pass a single tuple through the operator (RPTC). The cost of sorting is an $n \log(n)$ sort operation on $n = |A|$ tuples multiplied by a constant estimating the cost of comparing two rows (RCC).

The issue with this cost model is the difference in units between the CPU and memory/network components. Since the CPU approximates the number of operations performed on tuples, its only variable is the relation cardinality. However, since the memory and network components are estimates of the bytes used, they use the relation cardinality multiplied by the relation width. This resulted in a much higher effective weighting for those components. Thus, when VolcanoPlanner performed its cost-based optimization, it implicitly prioritized reducing memory and network consumption on a system with high memory availability and low network latency. The solution was to remove the column count component and make input cardinality the sole variable factor in the costing meth-

Algorithm 2 Distribution Factor Calculation

```
Require: rootNode ← an operator in a query tree
  procedure hasExchange(node):
      if node is Exchange then
          return true
      end if
      for child ∈ node.children do
          if hasExchange(child) then
              return true
          end if
      end for
  end procedure
  if hasExchange(rootNode) then
      return 1
  end if
  return dataPartitionSites(rootNode)
```

ods. Equation 4.3 shows this change applied to the cost of a `sort` operator (Equation 4.2).

$$\begin{aligned} O_{\text{CPU}} &= |A| * RPTC \ + \ |A| \log(|A|) * RCC \\ O_{\text{Memory}} &= |A| \end{aligned} \tag{4.3}$$

Additionally, a *distribution factor* was added to the operators capable of distributed computation. This factor rewarded operators for performing a distributed execution (i.e., the map phase of a map-reduce operation) by reducing its cost relative to a non-distributed execution. Algorithm 2 shows the calculation of a distribution factor for a certain operator. If an operator has a path to a leaf operator in the query tree, which did not include an `exchange`, it can be executed in parallel on partitions of the leaf operators' relation. Since all leaves in the query tree are base relation operations (either an index scan or table scan), the distribution factor would be the number of partition sites for that base relation (where a replicated base relation has one partition). If the operator had an `exchange` operator between itself and all the leaves, it would be operating on a whole relation, yielding a distribution factor of 1. Equation 4.4 shows a distribution factor *df* applied to the updated `sort` operator cost equation (Equation 4.3). In this equation, *df* is determined by using Algorithm 2 on the child of the `sort`.

$$\begin{aligned} O_{\text{CPU}} &= df^{-1}|A| * RPTC \ + \ df^{-1}|A| \log(df^{-1}|A|) * RCC \\ O_{\text{Memory}} &= df^{-1}|A| \end{aligned} \tag{4.4}$$

4.3 Planner Exploration Efficiency

The final issue was that the VolcanoPlanner took too long to generate and evaluate alternatives. This was due to the extensive rule set, which included almost all possible physical and logical rules. All the corresponding physical optimizations had to be regenerated for every logical alternative (e.g., pushing a filter one level down). Consequently, Calcite could generate as many as the Cartesian product of logical and physical possibilities, leading to an impossible number of alternatives to explore. This led to long query planning times, inefficient query plans, and occasionally a failure to generate a query plan. The solution was to split the second stage (the VolcanoPlanner stage) into two sequential phases: a logical optimization phase and a physical optimization phase. Each phase ran a subset of the original rule set using a Volcano Planner. The logical optimization phase contained 20 logical rules, and the physical optimization phase contained 36 rules, with about two-thirds being physical rules and the rest being logical rules.

Two of the logical rules included in the physical optimization phase were responsible for permuting the inputs of a join[2] and permuting the order of a nested join[3] (i.e., $((A \bowtie B) \bowtie C) \bowtie D)$). When these rules were included with the physical optimization phase, on queries with multiple joins or nested joins, the query planner would exceed either the computation time limit or the system resource limit and fail to create a query plan. This was the root cause of multiple planning failures in the original Ignite planner. To resolve this, a second physical optimization phase was created with these two rules disabled. This phase was used conditionally on queries that contained more than three nested joins or more than four join operations. These conditions were determined by trial and error to target only the queries in TPC-H [22] that failed to generate execution plans with the two rules enabled. By splitting the second planning stage into two phases and conditionally applying these two rules in the physical optimization phase, execution plans were successfully generated for all queries in the testing suite.

[2] https://calcite.apache.org/javadocAggregate/org/apache/calcite/rel/rules/JoinCommuteRule.html

[3] https://calcite.apache.org/javadocAggregate/org/apache/calcite/rel/rules/JoinPushThroughJoinRule.html

Chapter 5

Improving Query Execution

Once the query planner was stabilized, the next step was to improve the performance of query execution. This involved optimizing the execution of `join` operators and increasing the level of parallel computation within execution plans.

5.1 Join Operation Optimizations

5.1.1 Fully Distributed Joins

When analyzing the distribution mappings in Table 3.2, it was immediately clear a distribution mapping was missing. Assume the goal is to perform the join $A \bowtie B$ where both $A = A_1 \dots A_n$ and $B = B_1 \dots B_k$ are partitioned at sites $\alpha = \alpha_1 \dots \alpha_n$ and $\beta = \beta_1 \dots \beta_k$ respectively. The join can be expressed as follows:

$$A \bowtie B \equiv A \bowtie B_1 \cup A \bowtie B_2 \cup \cdots \cup A \bowtie B_k$$

Then, k partial joins can be performed in parallel by sending all of A to every site in β. Each partial join operates on a partition of B and decreases the input relation size by a factor of k. The disadvantage is since A is being broadcast, the same data is scanned and sent to all sites in β, which incurs extra CPU and network overhead. However, on a modern network with low latency, the gains from parallel computation often outweigh this extra data shipping cost [18]. Implementing this strategy also allowed join operations to broadcast their results to other operations that would benefit from parallel computation, like other joins. This was particularly effective for queries involving multiple large tables,

as it eliminated the need to perform data shipping for these large relations and allowed for the parallelization of their operations simultaneously.

5.1.2 Hash Join

The hash-join algorithm [31] was implemented as a `hash-join` operator with both input relations fully in memory. In this implementation, the right relation is used during the build phase and the left during the probe phase. Equation 5.1 shows the cost for the `hash-join` operator when performing $A \bowtie B$. The distribution factor df_r is calculated using Algorithm 2 on the right relation (relation B). The CPU component is the cost of passing through ($RPTC$), comparing (RCC), and hashing (HAC) all the tuples that pass through the operator ($|A| + |B| * df_r^{-1}$). The memory component is the cost of storing $|B|$ tuples in the hash table. Since the `hash-join` operation does not perform IO or network operations, those values are zero.

$$\begin{aligned} O_{\text{CPU}} &= (|A| + |B| * df_r^{-1}) \ * (RCC + RPTC + HAC) \\ O_{\text{Memory}} &= |B| * df_r^{-1} \end{aligned} \tag{5.1}$$

Applying the distribution factor to only the right join relation (B) was an intentional choice. Since the `hash-join` builds the hash table on the right join relation, performance can suffer if it must wait for data to be shipped to perform the build. Recalling the specifics of Algorithm 2, the distribution factor is ≥ 1 only if the operator operates on a local data partition (i.e., it does not have to be shipped). Thus, by applying the distribution factor only to the right join relation, the planner is rewarded only for plans where the right relation is partitioned, not the left. This behaviour is complemented by the memory component being a function of only the right relation, which rewards the planner for making the right relation as small as possible. Combining these means the planner prioritizes a hash table built on a small, local partition, with data shipping happening for the probe phase on the larger relation. The CPU cost component was designed carefully to maintain this preference while still being comparable with the existing `merge-join` operator (see Appendix A.1). When implementing the `hash-join` operator, the same distribution mappings as the existing join algorithms were used, including the changes in Chapter 5.1.1.

5.2 Multi Threaded Execution Plans

Fragments were already capable of executing in parallel at different sites to take advantage of Ignite's data partitioning functionality. This parallelization was improved by creat-

ing runtime sub-partitions at each site and having multiple threads process distinct sub-partitions instead of a single thread processing the whole partition. A fragment would be duplicated into multiple *variant fragments*, each responsible for creating and processing sub-partitions at a target site.

5.2.1 Variant Fragment Creation

To create a variant fragment (VF), Algorithm 3 is run with a non-root fragment and the target number of variants n. Since each variant will run in its own thread, n represents the number of threads desired. n variants are created, each assigned a unique ID $v_{id} \in \{0 \dots n-1\}$, and a copy of the original fragment query tree is made. When the original query tree is copied, three operators are modified: the two base relation scans (`table scan` and `index scan`) and the `receiver`. These three operators constitute a *source*. They are the leaf nodes of every query tree in every fragment, so any data in the fragment must start

Algorithm 3 Variant Fragment Creation

```
Require: non-root fragment
Require: n ≥ 1
  procedure COPY(node, type)
      if node is source then
          return node.copy([])
      else if node is reduction operator then
          Raise Exception
      else if node is join then
          newLeft ← COPY(node.left, DUPLICATOR)
          newRight ← COPY(node.right, type)
          return node.copy([newLeft, newRight])
      else
          newChildren ← []
          for child in node.children do
              newChildren.add( COPY(child, type) )
          end for
          return node.copy(newChildren)
      end if
  end procedure
  return COPY(fragment.rootNode, SPLITTER)
```

at one of these operators. When a source is encountered, it is converted into a *splitter* or a *duplicator*. A splitter splits its data between its variant fragments, whereas a duplicator duplicates its data to all. Most sources are splitters, which is how dynamic sub-partitioning is achieved. The exception is when two sources are the left and right sources to a `join` operator (i.e., there are no other sources between them and the `join` operator). If both `join` sources are partitioned, some partitions may not be properly combined, yielding incomplete results. Thus, one of the sources must be a duplicator to maintain correctness. Algorithm 3 always make the direct source of the left input a duplicator as the right source is more often a base relation scan that benefits from the dynamic sub-partitioning.

Sources are the only operators that are modified. All others are identical to the original fragment query tree. Additional restrictions are placed on root fragments and reduction operators (e.g., the reduce phase of a map-reduce operation). As root fragments are responsible for providing the final result to the user, they must have access to all the data to serve it in the correct sequence. Reduction operations would require non-trivial changes to support multi-threading, such as injecting collection operators (possibly into other fragments) that merge partial reduction results into the query tree. These changes would add significant complexity to the variant creation and execution processes to maintain correctness. A fragment is skipped if it is a root fragment or contains a reduction operator.

For example, in Figure 3.6, there are three fragments: black, blue, and red. Since the black fragment is the root fragment, no variants are created. When Algorithm 3 is run on the blue fragment, it starts at the `sender` operation and traverses down the tree to the `scan` operation, copying each operation as it goes. Since there are no special cases (e.g., a `join` operator in the tree), the `scan` operation becomes a *splitter* by default. There are no more operations in the blue fragment, so the variant creation is successful, and the variants are returned. The process is identical for the red fragment.

5.2.2 Variant Fragment Execution

When a VF is executed, it knows its variant id (v_{id}) and the total number of variant fragments (n). If a source in the VF is a splitter, it keeps an internal counter, c of the number of tuple reads it has done. If $c \% n = v_{id}$, it will send the tuple to the next operator. Otherwise, it will skip it. This is how the runtime partitions are created without impacting the base-level storage of the relations. This method does incur a slight penalty, as the entire partition is read in all threads. However, because Ignite operates entirely in memory, these reads are inexpensive and far outweighed by the benefits of parallel computation.

Again, using Figure 3.6 as an example, assume 2 VFs were created for the blue fragment,

with each VF knowing its v_{id} and the total number of VFs $n = 2$. Each VF would execute immediately and concurrently when it reached the processing site. In both VFs, the `scan` operation would read in all tuples from the *sales* relation but would only pass every 2nd tuple to the `sender` operation. The VF with $v_{id} = 0$ would pass the 2nd, 4th, 6th, etc. tuples and the VF with $v_{id} = 1$ would pass the 1st, 3rd, 5th, etc. tuples. When the corresponding receiver in the black fragment receives the tuples from the VFs, they would be consolidated and sent up the query tree to the `join` operation.

5.3 Join Condition Simplification

When running tests initially, Query 19 from TPC-H [22] was disabled because it did not complete execution after 4 hours. After investigating, it was discovered that the query does a nested-loop join on the `LINEITEM` and `PART` tables. With a scale factor of 1 (i.e., ∼1GB of raw data), this would be a join with $6M * 200K = 1.2 * 10^{12}$ tuples to process. The join predicate for Query 19 was an *OR* of multiple *AND* conditions where each *AND* shared a common condition but had multiple others bundled with it:

$$P \equiv ((c_1 \wedge c_2 \wedge c_3) \vee (c_1 \wedge c_4 \wedge c_5) \vee (c_1 \wedge c_6 \wedge c_7))$$

In this format, P could only be executed using a nested-loop join. However, the common condition c_1 could be pulled outside the *OR*:

$$P \equiv c_1 \wedge ((c_2 \wedge c_3) \vee (c_4 \wedge c_5) \vee (c_6 \wedge c_7))$$

If c_1 is a literal condition (e.g., $A.id = 123$), it can be converted to a filter on the corresponding input. This reduces the amount of data the join operation has to process, greatly improving the performance of a nested-loop join. If c_1 is a join condition (e.g., $A.id = B.id$) it can be split into two operations: a join operation with the predicate $P_1 \equiv c_1$ and a filter operation with the predicate $P_2 \equiv ((c_2 \wedge c_3) \vee (c_4 \wedge c_5) \vee (c_6 \wedge c_7))$. With a simplified join condition, the query planner can choose a more efficient algorithm to execute the join operation, which massively improves the performance. This extraction can be repeated until all common conditions are removed, yielding a far superior execution plan.

This logic was added as a new rule to the new logical optimization phase in the VolcanoPlanner stage, which significantly improved the performance of Query 19.

Chapter 6

Other Improvements

Throughout this research, multiple minor issues were discovered and fixed. These issues did not improve the system's performance but were necessary to ensure tests did not periodically fail.

6.1 Apache Ignite Improvements

Two issues were discovered in Ignite's implementation of join executions. The first issue was with the `CorrelatedNestedLoopJoin`, an operator that executes a variation of a nested-loop join. When performing left-join, an internal state variable was not set correctly. An infinite loop was possible without this state variable set correctly, occasionally causing a stack overflow. By setting the state variable correctly at the beginning of the operator execution, this potential infinite loop was eliminated. The second issue was a potential deadlock in the implementation of the `merge-join` operator. When performing a right-join, if the last sorted element of the left input relation had 2+ matches with the right relation, and the last sorted element of the right relation had no matches, the merge-join loop would break and wait for more tuples to process. However, no more tuples were available, so the `merge-join` operator would never finish, and the query would hang indefinitely. The exit conditions of `merge-join` operator were modified to handle this, ensuring the operator would always complete.

6.2 Apache Calcite Improvements

Early in this research, a problem was discovered with Apache Calcite choosing sub-optimal execution plans for queries with joins [1]. The root cause of this problem was how Calcite explored alternative physical execution plans. When an operator failed to satisfy a trait, the optimizer attempted to apply a conversion to produce a new operator that satisfied said traits. If this conversion was successfully applied, only the converted operator continued in the optimization process, and the original operator was discarded. This was not an issue for most operators; however, multiple possible conversions were available for the join operator. Despite this, only the first conversion was applied, missing out on other potentially more efficient ones. The solution was to re-add the original operator to the optimization queue so that other conversions could occur. This allowed the optimizer to choose from all possible variations of the operator.

[1] https://issues.apache.org/jira/browse/CALCITE-4542

Chapter 7

Performance Evaluation

To evaluate the performance of the baseline system and the aforementioned changes, the open-source system Benchbase [11] (formerly OLTP-Bench) was used with the TPC-H [22] benchmark. Minor configuration changes were made to the software to ensure compatibility with Ignite[1]. The TPC-H [22] schema was followed and 16 indexes were created across all tables[2]. Two TPC-H queries were disabled during this testing. Query 15 required SQL VIEWS, which are not supported in Ignite, and Query 20 contained an unresolved bug in the planning code that caused the query planner to fail.

Two tests were run for each configuration with four and eight machines. Each machine had two Intel E5-2620v2 CPUs (2.1GHz, 12 physical cores, 24 logical cores) with 32GB of RAM, connected by a 10GB ethernet line. Ignite was configured in a partitioned cache mode with zero backups and statistics enabled:

```
<property name="cacheMode" value="PARTITIONED"/>
<property name="backups" value="0"/>
<property name="statisticsEnabled" value="true"/>
```

Ignite was started on each machine as a Java 11 process (OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu-0ubuntu2.20.04)) with the following arguments:

```
-Xmx20G -Xms4G -Dfile.encoding=UTF-8 -server
```

[1]Modified software is available at https://github.com/marktdodds/benchbase/tree/ignite-tests

[2]Full DDL can be found at https://github.com/marktdodds/benchbase/blob/ignite-tests/src/main/resources/benchmarks/tpch/ddl-ignite.sql

```
-XX:+UseG1GC -XX:+ScavengeBeforeFullGC
-XX:+DisableExplicitGC -XX:+AlwaysPreTouch
-DIGNITE_UPDATE_NOTIFIER=false
-DIGNITE_SYSTEM_WORKER_BLOCKED_TIMEOUT=600000
-DIGNITE_JVM_PAUSE_DETECTOR_DISABLED=true
-Djava.net.preferIPv4Stack=true
```

The standard Apache Ignite version 2.16 built from source[3] was the baseline system. The changes in Chapter 4, Chapter 5, and Chapter 6 were implemented on top of the version 2.16 source code, compiled on each machine, and executed.

7.1 TPC-H Per Query Response Time

Individual query response time was measured on three different systems. The first system contained only changes from Chapter 4 (Query Planner Changes). The second system combined changes from Chapter 4 and Chapter 5.1 (Query Planner Changes and Join Optimizations). The third system used all changes from Chapter 4, Chapter 5.1, and Chapter 5.2 (Query Planner Changes, Join Optimizations, and Multithreading). A *test* consisted of a warm-up execution followed by three measured executions for each query. The average of the three executions was used as the overall execution time. Every system ran tests using TPC-H scale factors 0.5, 1, 2, and 3, corresponding to ∼0.5GB, ∼1GB, ∼2GB, and ∼3GB of raw data (excluding indices and other system views built). The average across scale factors was used as the execution time for a particular site configuration. Two configurations consisting of four and eight homogeneous sites were used. The result was eight tests for each system configuration, 24 in total. A full table of testing results can be found in Appendix B.1.

7.1.1 Query Planner Changes

Figure 7.1 shows the performance changes between the first system and the baseline system. Queries 7, 8, 10, 11, 12, 13, and 16 all experienced moderate to severe performance decreases. Query 8 is not shown in Figure 7.1 because it did not finish after four hours of execution. However, all of these decreases were superficial and share a root cause.

[3] https://github.com/marktdodds/ignite/tree/ignite-2.16

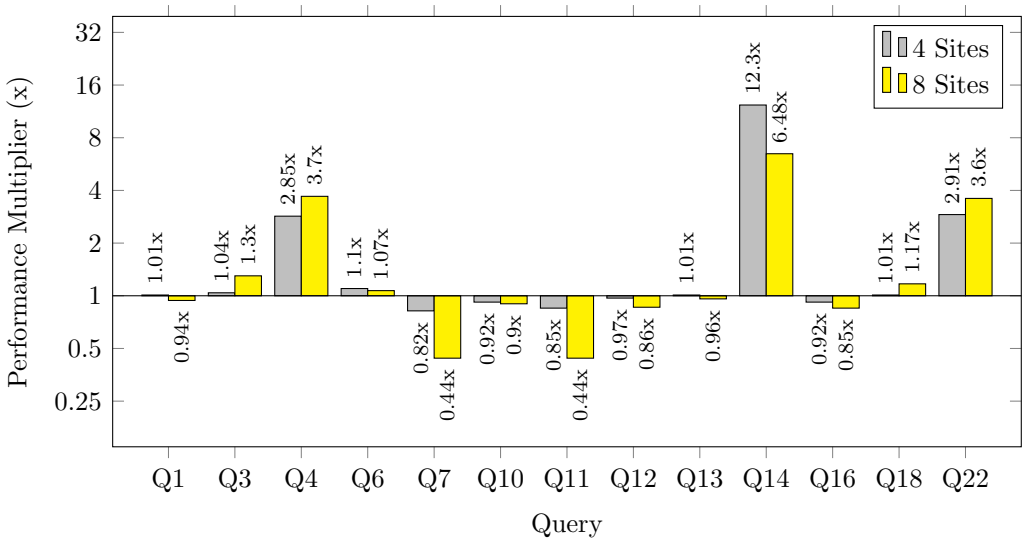


Figure 7.1: Incremental Performance from Query Planner Stabilization

The baseline system, by performing logical and physical optimizations together, generated execution plans that were logically inefficient but contained hash-distributed joins. With Chapter 4.3 enabled (i.e., logical and physical optimizations split into two phases), a logically optimized plan was generated, but the join ordering and join conditions had changed. The new join conditions did not satisfy the conditions for a hash distribution (Chapter 3.2.2), preventing hash-distributed joins from being used. This resulted in inefficient single-site joins since broadcast-distributed joins were not enabled until Chapter 5.1. This slowdown is superficial because once the broadcast-distribution join was added, all these queries surpassed the baseline performance. It also shows that missing configurations can not only abate improvements but can be seriously detrimental. Queries 5 and 9 were not completed after four hours, but the baseline system failed to generate an execution plan entirely, so this is considered an improvement.

Queries 4 and 22 show massive improvements from adding the missing rule to the first optimization phase. This allowed `filter` operations to be pushed down multiple levels into the base relation scans, significantly reducing the tuples being processed in the remainder of the query. The speedups in queries 3 and 14 are from a change in the collation (i.e., sorting order) of an index scan on a base relation. In Query 3, the new collation acts as a partial sort for a full sort operation higher in the query tree. For Query 14, the new collation changes the aggregation function from hash-map-based to sort-based on an already sorted input. These differences are from separating the second optimization stage

into two sub-phases, allowing Calcite to explore all possible execution plans. Queries 2, 17, and 21 all had sub-one-minute execution times. While no direct self-comparison to the baseline system is available, this is a massive improvement over the execution timeouts the baseline system experienced. In the baseline system, the query plans generated had multiple `nested-loop-join` operators. With the updated join estimation algorithm and the changes to the planning stages, these `nested-loop-join` operators were changed to `merge-joins` operators, which are significantly faster. Query 19 experienced a dramatic improvement due to the new Join Condition Simplification (JCS) rule from Chapter 5.3, with the longest execution time taking just over 10 seconds (compared to being cancelled after four hours). The JCS rule extracted three common conditions in the original join predicate. Two conditions were applied as filters to the input relations. The third was used as the new join predicate, enabling a `correlated-nested-loop-join` operator to be utilized, which was far more efficient. Query 18 had a minor performance improvement by changing the distribution of a join operation from *hash* distributed to *single*. Further investigation found the join operands were small enough that data shipping costs likely outweighed the benefit of distributed computation. With the modifications to the query planner, this extra cost was eliminated. Queries 1 and 6 did not display any significant changes, as the query plans and execution environments were identical to the baseline system.

7.1.2 Join Optimizations

Figure 7.2 shows the incremental improvements from Chapter 7.1.1 after additionally enabling the Join Optimizations in Chapter 5.1 (i.e., Chapter 4 and Chapter 5.1 enabled). Once these optimizations were enabled, there were performance improvements in almost every query. The biggest gains from this change are in Queries 2, 7, 8, 11, 13, and 16. In the original execution plans, all of these queries process at least one join by data shipping the larger relation via a hash distribution mapping. With the broadcast distribution mapping, the smaller relation is now data shipped, and the large relation is kept in place. Queries 2, 7, 8, 16, and 18 also benefit from the new `hash-join` operator in place of what initially were merge joins on intermediate results. This removes multiple intermediate sorting operations from the query tree, decreasing the total operation time and increasing the parallelism since the sort operation cannot be distributed. Queries 3, 10, 14, 17, and 21 all have moderate performance gains owing to the *broadcast* distribution mapping. These queries all perform a join operation with the `LINEITEM` table, the largest in the testing bench. With this new mapping, Ignite avoids data shipping this relation entirely, instead broadcasting tuples to the sites containing partitions of this relation as needed.

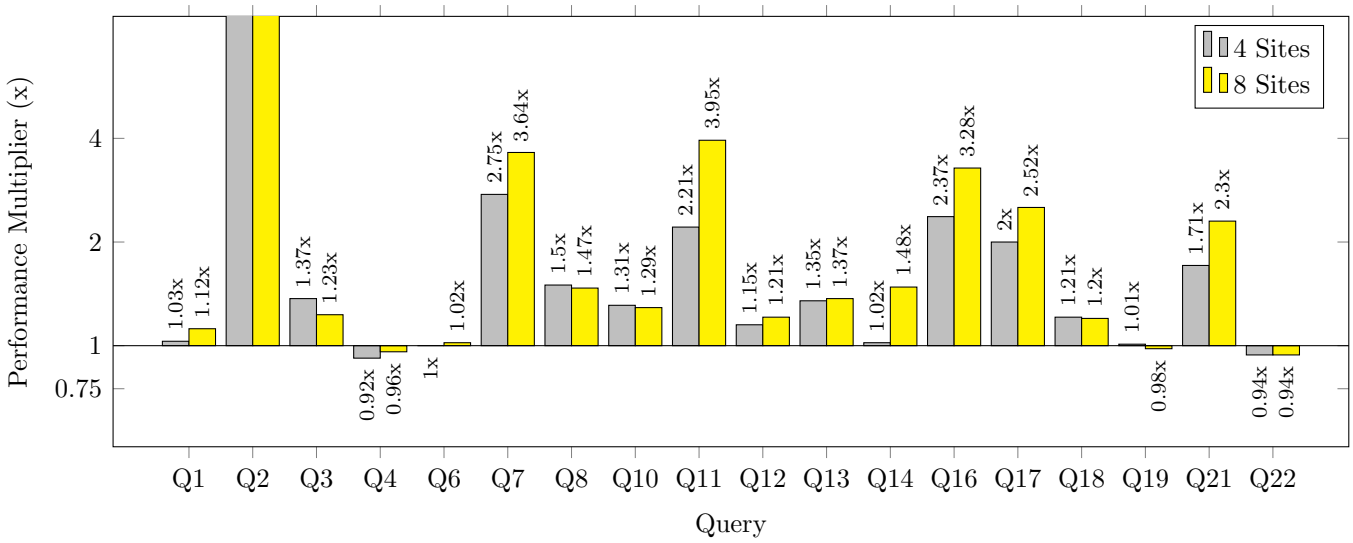


Figure 7.2: Incremental Performance from Join Optimizations

The performance gain in Query 12 is from the input ordering on a distributed join. With the query planner improvements, the planner chose (arbitrarily) to put the data shipping request tasks in the right input and the local relation scan in the left, whereas in the baseline system, the inverse happened. In the `merge-join` operator, the right source is requested first, then the left. In the baseline system, the base relation scan happened first, and then the data shipping requests were performed. In contrast, this system made the data shipping requests first, and then when it performed the base relations scan, other sites were already shipping their data. This extra parallelism is where the performance gain comes from. Queries 1, 6, and 19 produce the same execution plan as the baseline system and show no significant changes. Comparisons for Queries 5 and 9 are not available because they did not complete execution in Chapter 7.1.1 or the baseline system, but both had sub-one-minute execution times. Their execution times are shown in Appendix B.1.

Query 22 experienced minor slowdowns compared to the prior section because of a distributed join on the `CUSTOMER` relation, one of the smaller relations in the test bench. The broadcasting of the relation was the dominating cost in the execution, not the join operation itself. Query 4 also experienced minor performance decreases compared to the results from the prior section. However, when run in isolation through an SQL terminal, the decreases disappeared entirely. When investigating, it was discovered that the first measured run of Q4 was noticeably longer than the rest, and Q4 executes directly after Q3, which has the longest and most CPU-intensive planning process of all the queries.

Furthermore, when the query execution order was swapped so that Q4 was executed before Q3, the slowdowns disappeared. Thus, it is believed that the Calcite planner negatively impacted the execution of Q4, possibly due to large garbage collection activities or a memory leak. However, even with these relative decreases, the performance of Q4 and Q22 still far surpasses the baseline system.

7.1.3 Multithreading

Figure 7.3 shows the incremental improvements from enabling Multithreading, relative to the performance in Chapter 7.1.2. When testing different multi-threaded configurations, a dual-threaded configuration (i.e., each fragment is split into two variants) had the best performance so those results are presented here. Queries 1, 3, 5-8, and 14 all experienced significant performance improvements ranging from 20% to 50% depending on the configuration. These queries all have multiple distributed computation components (e.g., distributed joins, distributed aggregations, partially distributed sorts) that benefit from the extra parallelism. They also operate on very large subsets of the benchmark data, which is reduced substantially by the dynamic sub-partitioning. Conversely, Queries 2, 9-13, 19, and 21 all have negligible changes in performance. Queries 9, 10, and 19 are simple queries in which most of the work is executed in the root fragment, which does not support multithreading. Queries 12 and 13 have very restrictive filters that limit the amount

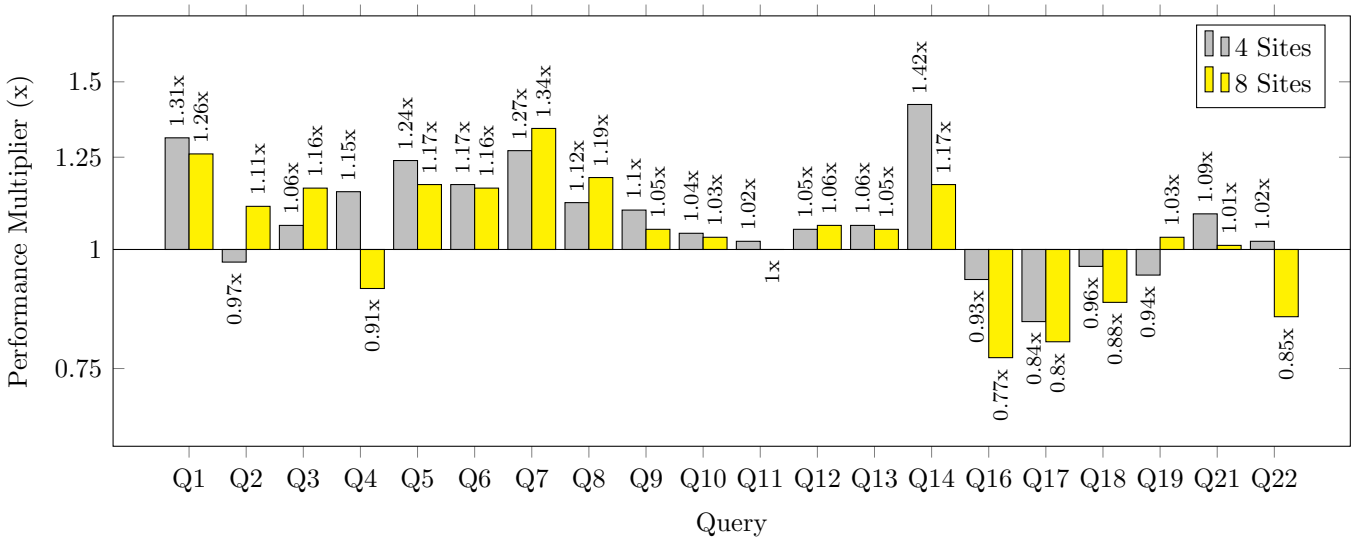


Figure 7.3: Incremental Performance from Multithreading

of data being processed, thus limiting the effectiveness of the dynamic sub-partitioning. Queries 2, 11, 17, and 21 are complex distributed join plans with multiple nested joins. These queries generate deep query trees where time is spent on data shipping and waiting for source materialization. These dominate the execution times and do not benefit from multithreading.

Queries 16, 18, and 22 all experience slowdowns because they contain a reduction operator in the fragment performing most of the computation and are kept single-threaded. The base relation scans still become multi-threaded, but there are no operations that benefit from this. Thus, dynamic partitioning is performed only to rejoin everything into a single thread directly after. This extra overhead increases the wait time between data shipping requests and causes slowdowns. Query 4 is a relatively simple aggregation query with two base relation scans. Only the base relation scans were suitable for multi-threaded execution, and they constitute a fraction of the overall execution time, resulting in a minimal gain from the multi-threaded plans. When run with eight processing sites, this caused a performance decrease resulting from idle time during data shipping. Query 17 suffered from a similar issue: there was little distributed computation, so most of the work was performed in the root fragment, which does not support multithreading. Again, the base relation scans were the only multi-threaded components and constituted a fraction of the overall execution time. The extra overhead of splitting and collecting the scan results caused the performance decrease.

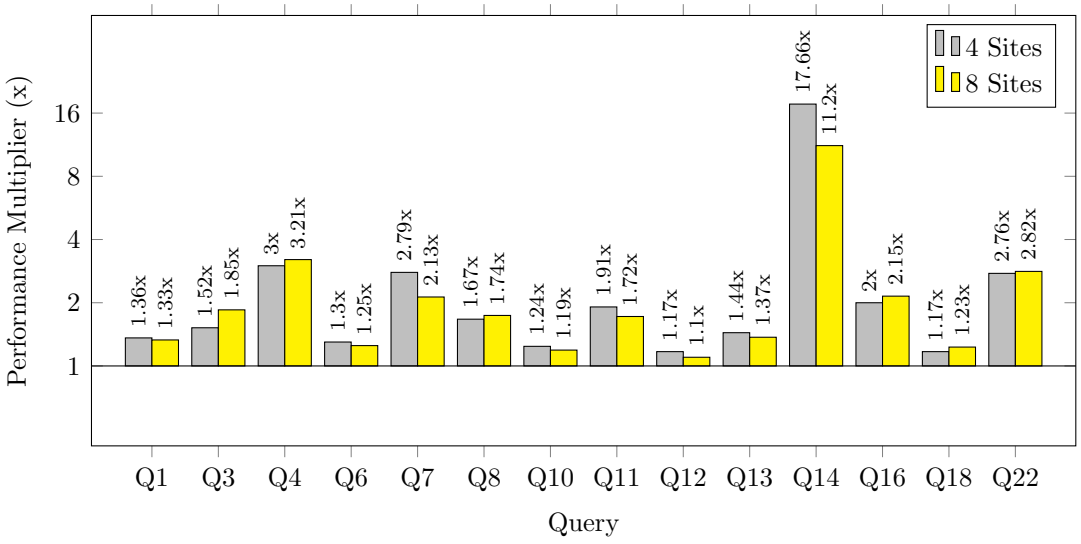


Figure 7.4: Overall Performance Increase

7.1.4 Overall Performance

Figure 7.4 shows the system performance with all changes enabled compared to the baseline system. Performance improved in every query and every configuration. Queries 2, 5, 9, 17, 19, and 21 are not shown here because the baseline system failed to plan or execute them. Their execution times are available in Appendix B.1.

7.2 TPC-H Average Query Latency

Average query latency (AQL) was tested against the baseline system using two system configurations: one with multi-threaded execution plans disabled (*Modified*) and one with them enabled (*Multi-Threaded*). Each system had one warm-up execution of each query before beginning the measurements. A test consisted of one or more *terminals* (clients) submitting randomized queries sequentially until the specified time elapses. Tests were run with two, four, and eight clients submitting work in parallel for 300 seconds on four and eight sites. AQL was measured as the arithmetic mean latency of all completed requests. Each configuration (e.g., two clients, four sites, *Modified*) was executed five times, and the average throughput across all five runs was used as the final value. Queries 2, 5, 9, 17, 19, and 21 do not complete execution on the baseline system, so they were disabled for throughput testing to ensure a fair comparison. A complete results table is shown in

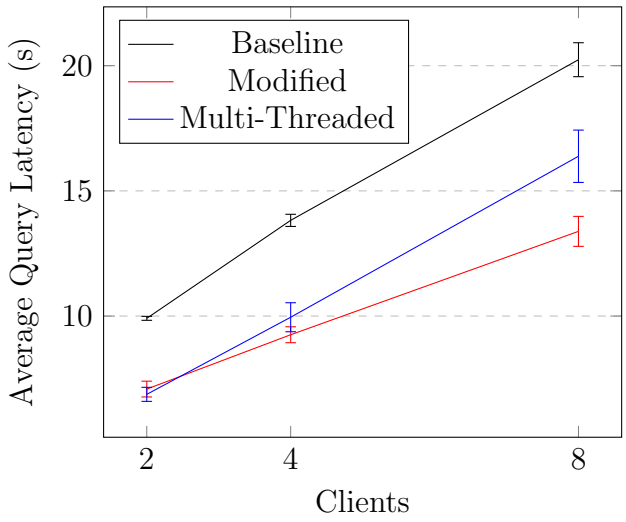


Figure 7.5: Average Query Latency for 4 Sites

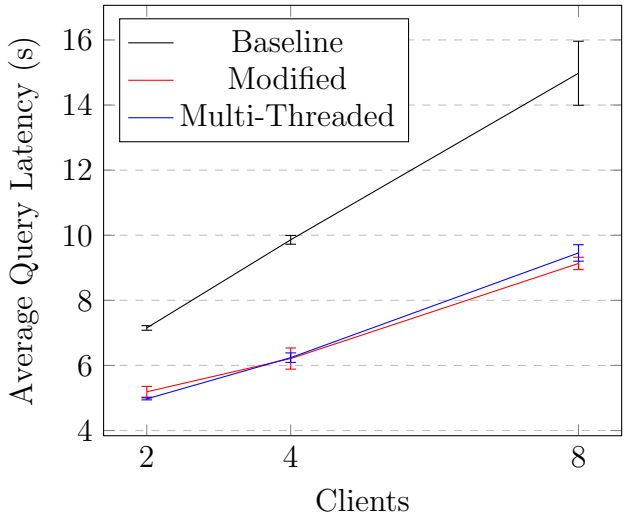


Figure 7.6: Average Query Latency for 8 Sites

Appendix B.2. Figure 7.5 and Figure 7.6 show the AQL testing results for four and eight sites respectively. Both system configurations showed statistically significant decreases in AQL relative to the baseline system. These decreases varied from 20% - 40% depending on the system and site configuration. The largest was a 39% decrease in AQL using the *Modified* system with eight clients and eight sites. The lowest was a 19% decrease using the *Multi-Threaded* system with eight clients and four sites. When directly comparing the eight-site tests to the four-site tests, the eight-site tests were consistently faster. This is not surprising as more sites mean more parallel processing of data and smaller data partitions for distributed operations.

With multi-threaded execution plans disabled, the AQL consistently decreased with the number of clients. Furthermore, the *Modified* graph begins to diverge from the *Baseline* graph, suggesting that the performance gap grows with the number of clients. This was expected as most queries have more distributed computation than the baseline system, allowing the load to be distributed more evenly amongst the processing sites. With multi-threaded execution plans enabled, similar performance was recorded for the 8-site configuration. On four sites, AQL with two clients was similar but did not decrease as much with four and eight clients relative to the *Modified* configuration. It is believed this was due to too many variant fragments competing for CPU time. Each query must have at least two fragments (as they all must retrieve data and collect the results at minimum) and thus two threads. Then, when dual-threaded executions are enabled, all queries will have at least four threads to execute. With four queries executing in parallel (from four

clients), there are at least 16 threads, and it takes only three fragments per query to match the logic processor count of 24 available on the testing hardware. With eight queries (from eight clients), the logic processor count is immediately surpassed with at least 32 threads executing in parallel. In the TPC-H [22] test bench, all but two queries (Q1 and Q6) have at least three fragments, with the highest (Q9) having nine fragments. The result is that the thread count quickly surpasses the number of CPU cores, resulting in CPU contention. The issue was worsened by a lack of IO operations (as Ignite is fully in-memory) that the thread scheduler would normally use to make threads sleep and ease contention.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis explored the use of Apache Ignite and Apache Calcite to process OLAP workloads of varying sizes. It provided an overview of the Apache Calcite framework used to parse and optimize SQL queries. It investigated three distinct strategies for improving the performance of OLAP workloads: improving the stability and efficiency of the query planner, adding new operations for executing queries, and implementing multi-threaded query executions. Each strategy achieved performance improvements independently of the other, and when all strategies were combined, every test had performance improvements. These improvements ranged from 1.2x to 17x faster than the baseline Ignite+Calcite system. This work also highlights Ignite+Calcite's ability to process OLAP workloads. While unsuitable for extensive datasets due to its in-memory design, Ignite+Calcite is well-matched for efficiently processing moderately sized datasets. With scale factors up to three (i.e., 3GB of raw data), Ignite+Calcite could execute almost all queries in under one minute, with most taking less than 30 seconds. Furthermore, the ease of setup and lack of required tuning make it appealing to those without extensive database knowledge.

Finally, this work highlights the potential pitfalls of query optimization systems and the importance of standard benchmark tests when developing them. Initially, Ignite+Calcite could not execute five of the 22 queries in the TPC-H [22] benchmark. Whether this be caused by poor execution plans or a lack thereof, if a standardized test bench like TPC-H [22] had been used throughout development, many of the issues discovered throughout this research would likely have been resolved.

8.2 Future Work

This research highlighted one area with great potential for future work: join operation executions. When creating an execution plan for Query 19 in the TPC-H benchmark, Ignite creates a `join` with an OR condition comprised of AND's (i.e., OR(AND(...), AND(...))). This condition is not suitable for a `merge-join` or `hash-join`, so a `nested-loop-join` is used, which is highly inefficient. It should be possible to split simple OR conditions into parallel joins, each processing an AND condition and taking the union as the final result. By splitting the condition, more efficient join algorithms could be used, significantly improving the execution time. In addition, joins with inequality conditions were always executed using a `nested-loop-join` instead of a `merge-join`. This simple improvement could produce substantial performance improvements depending on the executed workloads. Finally, since all but two queries include joins [12], more advanced join algorithms could provide significant performance improvements.

Finally, some aspects of the contributions of this thesis could be improved. The updated cost model presented here still has some inconsistencies with the execution engine. For example, when creating execution plans, the cost model implicitly assumes sources will be requested simultaneously. However, most operators request sources in right-to-left order. If a query has many local operations on the right-most source followed by a data shipping request, it would benefit from making the data shipping request first and then processing the local operations. Removing these inconsistencies would increase the overlap of non-competing operations, reducing the overall execution time of queries. The implementation of creating variant fragments (Algorithm 3) is also rudimentary in the rules and restrictions it uses. This process could be improved to support root fragments in the multithreading process and more intelligently choose which fragments to use and which to omit, depending on their operations.

References

[1] Murali Ande and Narendra Paruchuri. In-Memory Computing Patterns for High Volume, Real-Time Applications. Presented at *In-Memory Computing Summit*, Oct. 2018. [Online]. Available: https://www.imcsummit.org/2018/us/session/memory-computing-patterns-high-volume-real-time-applications. Accessed: 2024-06-29.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[3] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1754–1766, 2015.

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 221–230. Association for Computing Machinery, 2018.

[5] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[6] Shamim Bhuiyan, Michael Zheludkov, and Timur Isachenko. *High Performance in-Memory Computing with Apache Ignite*. Lulu Press, Inc., 2017.

[7] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MaPreduce. In

Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pages 975–986. Association for Computing Machinery, 2010.

[8] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43. Association for Computing Machinery, 1998.

[9] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, page 1–8. Association for Computing Machinery, 1984.

[11] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: an extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[12] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment*, 13(8):1206–1220, 2020.

[13] Apache Software Foundation. Apache Calcite. [Online]. Available: https://calcite.apache.org. Accessed: 2024-07-31.

[14] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[15] Goetz Graefe and William J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993.

[16] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Survey*, 16(2):111–152, 1984.

[17] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[18] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[19] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[20] Jignesh M. Patel and David J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *Proceedings of the 8th ACM International Symposium on Advances in Geographic Information Systems*, GIS '00, page 54–61. Association for Computing Machinery, 2000.

[21] Kinjal Patel. Real-time Data Access with Apache Ignite SQL. Presented at *Ignite Summit Europe*, Nov. 2022. [Online]. Available: https://www.youtube.com/watch?v=3w0H3zLH594, 2022. Accessed: 2024-06-29.

[22] Meikel Poess and Raghu Nambiar. TPC Benchmark H Standard Specification. Available: http://dx.doi.org/10.13140/RG.2.1.1883.9288, 2010. Accessed: 2024-03-12.

[23] Arnon S. Rosenthal. Note on the expected size of a join. *SIGMOD Record*, 11(4):19–25, 1981.

[24] J. B. Rothnie, P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):1–17, 1980.

[25] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *ACM SIGMOD Record*, 18(2):110–121, 1989.

[26] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1961–1976. Association for Computing Machinery, 2016.

[27] Roman Shtykh and Toru Yabuki. 'Recent purchases' with Apache Ignite. [Online]. Available: https://techblog.yahoo.co.jp/oss/yahoo_shopping_purchases_ignite, 2019. Accessed: 2024-06-29.

[28] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 7th edition, 2019.

[29] Cristiana-Stefania Stan, Adrian-Eduard Pandelica, Vlad-Andrei Zamfir, Roxana-Gabriela Stan, and Catalin Negru. Apache Spark and Apache Ignite Performance Analysis. In *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*, pages 726–733, 2019.

[30] Arun Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '94, page 287–300. Springer-Verlag, 1994.

[31] Jingren Zhou. Hash join. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1288–1289. Springer US, 2009.

[32] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 4th edition, 2020.

APPENDICES

Appendix A

Modified Cost Model Calculations

A.1 CPU Cost Comparison of Hash Join and Merge Join

Assume $A \bowtie B$ is executing where A and B have distribution factors df_A and df_B resp. The CPU cost of a `hash-join` operator H for this join using Equation 5.1 is:

$$\begin{aligned} H_{CPU} &= (|A| + |B| * df_B^{-1}) * (RCC + RPTC + HAC) \\ &= \frac{A}{df_A^{-1}}(RCC + RPTC + HAC) \\ &\quad + B(RCC + RPTC + HAC) \\ &\quad \text{where } A = df_A^{-1}|A| \text{ and } B = df_B^{-1}|B| \\ &= df_A * A(RCC + RPTC + HAC) \\ &\quad + B(RCC + RPTC + HAC) \end{aligned} \tag{A.1}$$

The CPU cost of a `merge-join` operator is the cost of sorting both inputs and performing the merge. The CPU cost of performing the merge is:

$$CPU = (|A| * df_A^{-1} + |B| * df_B^{-1}) * (RCC + RPTC + HAC)$$

Combining this with the cost of two `sort` operators from Equation 4.4 (for the left and right input relations) yields the total CPU cost of a `merge-join` operator MJ:

$$\begin{aligned} MJ_{CPU} &= (df_A^{-1} * |A| + df_B^{-1} * |B|) * (RCC + RPTC) \\ &\quad + (df_A^{-1}|A| * RPTC + df_A^{-1}|A| \log(df_A^{-1}|A|) * RCC) \\ &\quad + (df_B^{-1}|B| * RPTC + df_B^{-1}|B| \log(df_B^{-1}|B|) * RCC) \\ &= (A + B) * (RCC + RPTC) \\ &\quad + A(RPTC + \log(A) * RCC) \\ &\quad + B(RPTC + \log(B) * RCC) \\ &\quad \text{where } A = df_A^{-1}|A| \text{ and } B = df_B^{-1}|B| \\ &= A(RCC + RPTC + \log(A) * RCC + RPTC) \\ &\quad + B(RCC + RPTC + \log(B) * RCC + RPTC) \end{aligned} \tag{A.2}$$

There are two cases to consider: $df_A = 1$ and $df_A > 1$ (since a distribution factor can never be < 1). First, assume the distribution factor df_A has a value of 1. The red and blue sections of MJ_{CPU} are the cost of sorting the two input relations. As the relations grow, the extra cost of sorting will outweigh the constants in H_{CPU} and the `hash-join` will be chosen. However, if either of the sort costs are removed (e.g., by reading from a sorted index), the determining factor is the cost of sorting the other relation. A `merge-join` will be chosen if the cost is small enough. Otherwise, the `hash-join` will be selected. If both sorting costs are removed the MJ_{CPU} will always be less than H_{CPU} and a `merge-join` will be chosen.

If $df_A \neq 1$, then $df_A > 1$. In this case, it is assumed that relation B is data shipped to build the hash table (this was the case for all tests in the testing bench) so $df_B = 1$. By swapping the input relation order of the `hash-join` H a new `hash-join` operator H^* is created with a CPU cost:

$$\begin{aligned} H^*_{CPU} &= df_B * B(RCC + RPTC + HAC) \\ &\quad + A(RCC + RPTC + HAC) \\ &= B(RCC + RPTC + HAC) \\ &\quad + A(RCC + RPTC + HAC) \\ &< H_{CPU} \end{aligned}$$

In [28] it is shown that join operations are commutative excluding attribute ordering, so swapping the input relation order is a valid query rewrite and does not affect correctness. The planner now has a new `hash-join` operator H^* with a smaller CPU cost and a relative

value of $df_A = 1$ (as relation A and relation B have been swapped) that can be compared to a `merge-join` using the first case described above. This is how the restriction of not building a hash table on shipped data is enforced.

Appendix B

TPC-H Results

B.1 TPC-H Per Query Results

The following tables show the average execution time in milliseconds for each query in the TPC-H test bench across different configurations. Table B.1 is the baseline system with no modifications. Table B.2 is the system with Chapter 4 enabled. Table B.3 is the system with Chapter 4 and Chapter 5.1 enabled. Table B.4 is the system with all modifications enabled.

Table B.1: Per Query Response Time: Baseline

	4 Sites				8 Sites			
	0.5 SF	1 SF	2 SF	3 SF	0.5 SF	1 SF	2 SF	3 SF
Q1	4407.58	10296.62	18609.12	29968.01	2696.76	5200.41	9404.68	14927.74
Q2	-	-	-	-	-	-	-	-
Q3	6389.54	13999.65	22804.8	39565.31	5073.48	11298.98	20797.96	29697.13
Q4	11125.66	22076.05	45172.09	61327.84	10660.37	23987.99	47143.08	78872.49
Q5	-	-	-	-	-	-	-	-
Q6	578.23	1004.31	2113.16	2869.08	349.09	593.94	1118.39	1665.28
Q7	6995.89	14353.24	26047.5	39499.71	3616.34	6079.89	13159.2	20606.48
Q8	13374.91	27456.62	59606.94	86974.32	6752.75	14706.56	33966.9	51464.43
Q9	-	-	-	-	-	-	-	-
Q10	4619.28	9575.91	19153.08	28579.26	2878.92	5265.6	10982.73	15616.48
Q11	468.83	958.29	1875.81	3287.47	301.58	544.26	1330.2	1881.94
Q12	520.84	1103.37	2176.41	3200.11	309.03	636.17	1182.67	1658.98
Q13	1475.24	2842.32	5727.01	8396.5	874.96	1435.09	3652.47	4951.33
Q14	2075.09	4534.82	8005.51	12570.03	1178.09	2173.56	4676.96	7122.28
Q16	693.71	1422.59	2915.35	3849.37	659.46	1307.29	2799.53	3196.19
Q17	-	-	-	-	-	-	-	-
Q18	8482.71	21201.76	33320.48	50978.57	10135.24	20329.48	40391.96	64297.88
Q19	-	-	-	-	-	-	-	-
Q21	-	-	-	-	-	-	-	-
Q22	1233.27	2402.33	4248.63	7127.09	1618	2960.76	4769.24	8381.87

Table B.2: Per Query Response Time: Query Planner Changes

	4 Sites				8 Sites			
	0.5 SF	1 SF	2 SF	3 SF	0.5 SF	1 SF	2 SF	3 SF
Q1	5295.66	9183.19	18184.62	27780.08	3028.78	4874.32	10792.17	16073.03
Q2	44361.32	160306.18	664516.39	1315715	32907.24	144482.04	664516.39	1289619.52
Q3	5941.07	12194.51	25999.25	36932.4	3975.64	7901.65	16245.38	24160.19
Q4	3513.87	7273.72	16842.21	24414.59	2472.98	6979.79	14095	21346.77
Q5	-	-	-	-	-	-	-	-
Q6	479.59	1032.06	1940.01	2507.49	275.66	580.67	1129.85	1662.58
Q7	7452.62	15154.83	35647.15	59035.33	7234.63	14002.82	30793.51	52275.96
Q8	13374.91	27456.62	59606.94	86974.32	6752.75	14706.56	33966.9	51464.43
Q9	-	-	-	-	-	-	-	-
Q10	5370.25	10967.46	19992.67	29326.7	3291.49	5990.62	11367.32	17925.37
Q11	568.35	1205.42	2354.68	3401.02	678.71	1488.35	2590.52	4379.22
Q12	614.29	1129.83	2051.49	3214.4	385.81	775.83	1233.23	1892.38
Q13	1503.95	3077.59	5093.63	8237.43	1080.14	1670.83	3195.44	4838.26
Q14	171.81	358.43	668.34	1006.31	203.44	392.53	652.85	956.62
Q16	702.85	1504.42	2940.54	5106.14	757.01	1595.55	2814.35	4358.49
Q17	10144.37	17193.53	37176.03	54524.79	9418.36	15290.01	37176.03	62141.08
Q18	10179.02	16746.85	35345.46	-	9954.01	16925.52	31268.67	-
Q19	1681.39	3815.81	7053.99	11882.39	1105.55	2429.21	4231.61	5124.52
Q21	23151.68	44751.88	103879.83	151151.91	22098.45	41409.47	103879.83	134555.25
Q22	481	730.22	1633.88	2246.22	376.47	859.32	1550.62	2350.04

Table B.3: Per Query Response Time: Query Planner Changes and Join Optimizations

	4 Sites				8 Sites			
	0.5 SF	1 SF	2 SF	3 SF	0.5 SF	1 SF	2 SF	3 SF
Q1	4767.81	8988.66	18087.25	28173.89	2710.24	4468.93	9565.08	13987.44
Q2	1264.82	2918.49	6195.4	11360.69	1407.57	2189.4	4562.97	8114.66
Q3	4534.33	8949.23	17617.98	27352.38	3529.13	6419.67	13133.35	18410.71
Q4	4214.27	8673.4	16594.38	24573.54	2817.19	6519.54	15735.25	21021.39
Q5	7219.67	14671.71	29075.64	43817.38	5519.94	11692.7	23632.53	34129.91
Q6	495.9	990.68	1851.7	2615.82	296.38	575.56	1028.85	1603.49
Q7	3380.91	6277.05	11726.42	17754.26	1853.89	4472.29	8648.78	13243.68
Q8	8586.94	18770.41	37337.16	62653.82	4219.34	10487.06	23478.8	36079.77
Q9	13589.22	27965.25	75288.94	145867.56	6240.12	15737.03	41267.04	81225.6
Q10	4216.72	8294.13	14807.1	22629.63	2561.5	4740.08	8617.2	13693.85
Q11	298.72	507.74	1026.32	1494.29	184	342.6	753.7	1007.8
Q12	523.33	1004.61	1880.55	2690.57	307.6	560.27	1086.84	1806.08
Q13	1101.75	2136.06	4116.91	6074.66	769.62	1208.84	2511.43	3384.17
Q14	214.68	309.51	580.36	1022.44	144.12	244.5	472.63	632.63
Q16	372.5	633.83	1139.32	1926.81	242.79	528.54	807.18	1253.04
Q17	4388.59	9581.36	18564.15	28720.29	3893.77	6777.9	14858.35	21467.11
Q18	7771.76	13968.64	26935.49	38740.56	6813.65	13353.95	28912.98	38605.43
Q19	2213.66	3319.95	6753.95	10780.57	1273.07	1950.97	4530.33	5843.08
Q21	13879.72	28050.35	55951.74	88415.7	9448.08	20949.48	39579.98	59604.71
Q22	499.72	794.58	1789.01	2285.94	420.81	872.51	1786.14	2353.19

Table B.4: Per Query Response Time: Query Planner Changes, Join Optimizations, and Multithreading

	4 Sites				8 Sites			
	0.5 SF	1 SF	2 SF	3 SF	0.5 SF	1 SF	2 SF	3 SF
Q1	3651.2	6993.61	14043.89	20597.73	2162.42	3490.37	7368.65	11499.7
Q2	1411.59	2515.42	6853.16	12428.23	1088.61	1962.89	4705.19	7644.92
Q3	4417.88	8416.63	16920.77	24299.08	2756.79	5561.97	12478.07	15887.85
Q4	3227.02	7690.23	15263.07	22442.89	3110.89	7492.44	16083.82	24018.74
Q5	6155.72	11391.02	23477.83	35179.06	4131.81	10250.93	20862.1	32201.09
Q6	412.73	976.38	1518.11	2094.15	234.81	536.87	1016.33	1258.73
Q7	2377.84	5053.66	9457.71	15100.87	1446.21	3321.78	6183.21	10020.74
Q8	7439.33	17035.69	34738.98	55487.17	4074.49	8134.12	19142.46	30016.43
Q9	10171.81	26459.72	77306.22	143046.87	5948.6	14985.48	39410.25	78099.66
Q10	4015.26	7834.69	14527.83	22204.01	2457.46	4412.54	9618.91	12533.39
Q11	289.9	539.21	961.57	1440.99	221.54	327.43	738.62	920.29
Q12	507.41	988.16	1684.25	2580.7	309.68	578.14	1021.43	1472.41
Q13	1054.21	2020.38	3839.73	5762.42	683.31	1142.68	2442.4	3459.89
Q14	142.08	255.59	448.37	615.05	121.64	226.07	380.68	539.53
Q16	346.04	715.33	1415.74	1990.57	329.76	655.11	1123.73	1506.81
Q17	6707.38	11742.15	19726.98	30597.36	4520.24	10041.38	18050.07	26143.39
Q18	7068.23	15255.43	30935.06	41290.44	7508.54	15463.66	31763.43	44963.64
Q19	2506.9	3987.87	6844.12	10240.28	1215.53	2429.38	3627.72	5796.02
Q21	12813.92	25794.94	52760.3	77212.71	8770.82	19868.66	41258.79	61932.86
Q22	511.55	867.75	1456.69	2421.14	598.77	997.63	1881.93	2727.58

B.2 TPC-H Average Query Latency Results

The tables below show the TPC-H Throughput results in requests-per-second for the different system configurations tested.

Table B.5: Average Query Latency: Baseline (seconds)

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVG	Confidence Interval
2 Term, 4 Sites	9.8479	9.8443	9.9503	9.8536	10.0498	9.9092	[9.8301, 9.9882]
2 Term, 8 Sites	7.0316	7.25	7.1338	7.1535	7.1917	7.1521	[7.0814, 7.2228]
4 Term, 4 Sites	13.4417	13.6157	13.9967	13.9897	14.0687	13.8225	[13.5798, 14.0652]
4 Term, 8 Sites	10.0372	9.7978	9.7057	9.745	10.0097	9.8591	[9.7242, 9.994]
8 Term, 4 Sites	20.3426	19.5551	19.6801	20.1219	21.5002	20.24	[19.5617, 20.9182]
8 Term, 8 Sites	16.7932	15.3287	14.2333	14.4279	14.1041	14.9774	[13.9936, 15.9612]

Table B.6: Average Query Latency: Query Planner Changes and Join Optimizations (seconds)

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVG	Confidence Interval
2 Term, 4 Sites	7.3514	6.6764	7.5503	6.9815	6.8576	7.0834	[6.7683, 7.3985]
2 Term, 8 Sites	5.5071	5.1273	5.1321	5.0063	5.1815	5.1909	[5.0259, 5.3558]
4 Term, 4 Sites	9.7127	9.4825	9.284	8.8528	8.9345	9.2533	[8.9352, 9.5713]
4 Term, 8 Sites	6.7581	5.9247	6.2328	6.3246	5.815	6.211	[5.8856, 6.5365]
8 Term, 4 Sites	12.9654	13.5115	14.4624	13.3047	12.6739	13.3836	[12.7851, 13.9821]
8 Term, 8 Sites	9.3172	8.9807	9.2026	8.8369	9.3295	9.1334	[8.9433, 9.3235]

Table B.7: Average Query Latency: Query Planner Changes, Join Optimizations, and Multithreading (seconds)

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVG	Confidence Interval
2 Term, 4 Sites	6.9016	7.3604	6.8874	6.4857	6.7267	6.8723	[6.5917, 7.153]
2 Term, 8 Sites	4.973	4.934	4.9486	5.0235	4.9867	4.9732	[4.9426, 5.0037]
4 Term, 4 Sites	10.9182	10.0879	9.6127	10.0155	9.1297	9.9528	[9.3735, 10.5321]
4 Term, 8 Sites	6.4011	6.1092	6.0237	6.3818	6.2794	6.239	[6.0927, 6.3854]
8 Term, 4 Sites	15.3417	17.6323	16.9161	14.8984	17.128	16.3833	[15.338, 17.4285]
8 Term, 8 Sites	9.2574	9.4161	9.1842	9.5015	9.9219	9.4562	[9.2029, 9.7095]