Experiments & Analyses Paper

Apache Ignite + Calcite Composable Database System: Experimental Evaluation and Analysis

Mark Dodds and Khuzaima Daudjee
Cheriton School of Computer Science
University of Waterloo
{mtdodds,khuzaima.daudjee}@uwaterloo.ca

ABSTRACT

Recent interest in composable data management systems is leading to a growing use of Apache Ignite composed with Apache Calcite for query processing. Apache Ignite is an open-source distributed 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 paper conducts an experimental analysis of the performance of Apache Ignite composed with the Apache Calcite system for online analytical processing (OLAP) with varying workload and data distribution settings. From empirical studies and technical analysis, multiple areas for improvement are identified, and associated enhancements are implemented into the system. Through experimental evaluation using the de facto TPC-H and Star Schema benchmarks, it is demonstrated that each strategy yields performance improvements across multiple queries, and with all strategies enabled, performance improves generally for all queries in the workload.

1 INTRODUCTION

The "one size does not fit all" tenet has sparked recent interest in moving away from monolithic database systems to *composable* database systems in which modular components that map to core functionality can be composed together to deliver a complete and customized system. This paradigm shift touts advantages that include faster and focused innovation, co-evolution of components, efficiency, resource disaggregation and better user experience, among others [26, 27]. In this paper, we conduct an experimental study and analysis of Apache Ignite + Apache Calcite, forming a composable system with an Ignite user base advertised to include several major commercial vendors [14]. Currently, no published work exists detailing the use of Apache Ignite + Apache Calcite as a composable database system for complex queries on distributed data.

Apache Ignite is a distributed in-memory key-value data system whose operations are interacted with through its open-source API. The data can be partitioned and distributed across multiple nodes or sites to achieve horizontal scaling and distribute query processing. Apache 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 in-memory store, Ignite implements a database engine using Apache Calcite (Ignite+Calcite) [13]. Apache Calcite is a high-performance dynamic data management framework that provides many functionalities of a typical database management system [11]. It contains an out-of-the-box SQL parser, validator, and optimizer but defers data storage, metadata storage, and data processing functions 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 both open-source projects supported by a community of developers both within and outside of the Apache Software Foundation. This can allow development teams to focus on system improvements and feature enhancements without being limited by commercial support obligations. However, being open-source, these projects come with a level of assumed risk and responsibility for the user not shared by commercial projects. 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.

Consequently, we conducted this research to understand and evaluate the processing capabilities of OLAP workloads using Ignite+Calcite. Multiple issues arose quickly when Ignite+Calcite was tested with TPC-H [28], a popular de facto benchmark for OLAP workloads. Of the 22 TPC-H queries, eight failed to execute using a standard deployment. Query 15 requires SQL Views which are not supported in Ignite+Calcite and Query 20 raised an exception in the planning process. Queries 17, 19, and 21 produced partially optimized execution plans containing multiple nested-loop joins that exceeded a four-hour runtime limit. Queries 2, 5, and 9 failed to generate execution plans entirely. When queries were executed, work was often isolated to a single CPU core, leaving other cores idling. These experimental studies yielded the following problems to address in Apache Ignite+Calcite as the core of this research: (i) the query planning process is unstable and can fail to generate execution plans; (ii) execution plans are often not fully optimized, resulting in long execution times, particularly for join processing; (iii) execution plans do not fully utilize modern CPUs with multiple cores. To address these issues, this paper presents an experimental evaluation and analysis that gave rise to three key contributions:

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

To the best of our knowledge, this paper is the first work that provides insight, through experimentation and analysis, on the use of the Ignite+Calcite composable system for OLAP query processing. Furthermore, we discuss our experience with modularity, interoperability, developer support, and documentation in using this composable system.

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 1: Sample Schema and Join Query

2 RELATED WORK

There is growing interest in composable data management systems [26, 27]. Instead of a single-purpose monolithic system, composable systems, like Ignite, are built from a collection of reusable components. These reusable components promote standardization and can considerably speed up development time. Research topics vary from standardizing relational algebra [36], integrating external query optimizers [21] to developing entire query engines using existing projects [24]. Being a relatively new area, there are currently no published performance evaluations of systems built using this composable paradigm.

Despite claims by industry giants like Microsoft, Netflix, IBM, and Yahoo of using Apache Ignite [14], there is no detailed information about their usage. General implementations of Apache Ignite exist [1, 25] with one for a simple e-commerce platform [33] but none of these include performance evaluations. Stan et. al. [35] 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.

Query optimization as a general research area has been a heavily researched topic as it is a core feature of any database management system. Jarke and Koch [18] provide a foundational relational calculus framework for general query evaluation. Bernstein et al. [5] presented techniques for optimizing relational queries in the early distributed database system SDD-1 [30]. Almost two decades later, Chaudhuri [6] focussed on optimization of SQL queries in relational database systems using relational algebra.

Since join operations are typically among the most resource-intensive processes in a database system, a large body of research is dedicated to them [16, 22, 39]. Hash and sort-merge join algorithms under varying memory and CPU constraints have been the focus of many studies [2, 3, 19, 32]. Schneider and DeWitt [31] evaluated parallelized versions of the join algorithms from [8].

3 QUERY PROCESSING AND OPTIMIZATION

As Apache Ignite's core functionality is a distributed data storage system, it composes most of the querying logic from an external library. Apache Calcite was built for this exact purpose, providing out-of-the-box query parsing and optimization capabilities that is composable with a system like Apache Ignite. Knowing how these two composable systems work is essential for understanding how Ignite+Calcite processes queries. The majority of this knowledge was derived from analyzing the source code of Apache Ignite and Apache Calcite, and how they interfaced with each other, driven by investigative evaluation and testing.

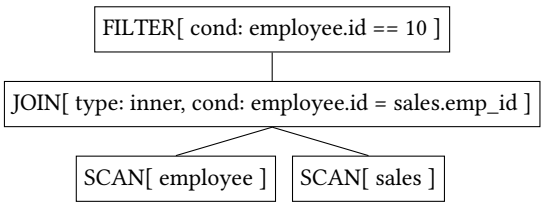


Figure 2: Unoptimized Query Tree of Query A from Figure 1

3.1 Apache Calcite

Relational algebra [7] operators are the main building blocks for Calcite. They can be either *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 sort order 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 2 shows the resulting query tree when Query A in Figure 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 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 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 of the query tree. Calcite provides many rules [12] 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 [17]. 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 O 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 1).

$$C(QT) = \sum_{O \in QT} C(O) \tag{1}$$

3.2 Apache Ignite

Ignite supports a standard architecture for distributed query processing described in [20]. 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 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 shown in Equation 2 (Section 4.2 discusses the standardization of the relevant operator cost units).

$$C(O) = O_{\text{CPU}} + O_{\text{Memory}} + O_{\text{IO}} + O_{\text{Network}} \tag{2}$$

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 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, e.g., pushing `filter` operators down the query tree (Figure 3). The objective of this first stage is to apply transformations that are guaranteed to improve the execution performance of the query.

In the second stage that is the main optimization phase, 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,

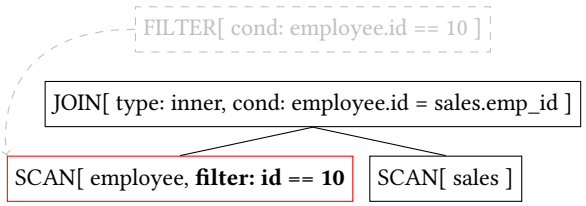


Figure 3: Query Tree After Applying Filter Pushdown Rules to Figure 2

compare their costs, and return the query tree with the lowest overall cost using the cost model. An essential part of the second stage is optimizing the physical traits of each operator in the query tree to create as efficient a physical execution plan as 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 has a distribution trait (property) value. Since operators are inputs (sources) for other operators, the destination or target distribution trait of the main operator must be compatible with the distribution traits of all its sources (source distribution traits) for data to flow between them. A source distribution is compatible (satisfies) a target distribution if the source distribution executes at a superset of the target distribution sites (see Table 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 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 attempts 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

Table 1: Distribution Satisfaction Matrix

Source \ Target	Single	Broadcast	Hash
Single	Yes	No	No
Broadcast	Yes	Yes	Yes
Hash	No	Yes*	Yes*

*Only if the hash function produces a superset of the target sites

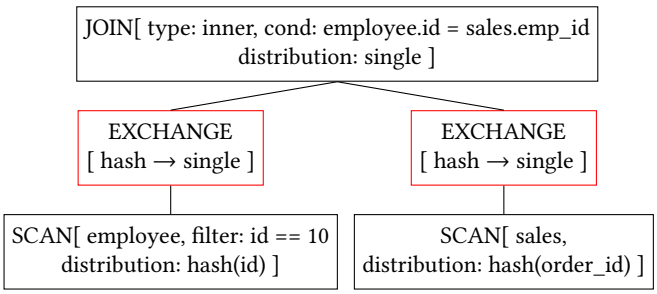


Figure 4: Enforcing Distribution Trait Satisfaction on Figure 3

plans, knowing that if trait satisfaction is maintained, then so is correctness.

This exploration is facilitated by the *deriveDistribution* method, which any operator can implement. This method generates a list of distribution mappings, each containing a possible 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). The only operators that implement this method are the physical join operators: `merge-join` and `nested-loop-join`. Table 2 shows the set of distribution mappings currently generated in Ignite+Calcite. 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 partitions 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 [34] where the left-hand equi-join condition is a partition key for the left relation of the join operation. Each row of the right relation uses the hashing function of the left relation to determine the partition of the left relation that contains possible matches. Once the partition is identified, the row of the right relation is sent to that partition's location to be processed.

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*. Each fragment contains a *root* 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`

Table 2: Possible Join Operator Distribution Mappings

	Required Source Distributions	
Possible Join Distribution	Left Source	Right Source
Single Broadcast Hash	Single Broadcast Hash	Single Broadcast Hash

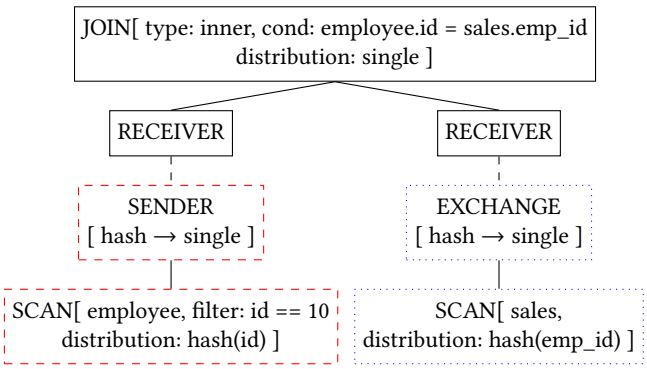


Figure 5: Query A (Figure 1) Single-Site Join Execution Fragments

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 the processing sites to which a fragment will be sent. 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 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 5 shows a possible set of fragments for executing Query A from Figure 1 as a single-site join. Each colour group, shown by red-dashed, blue-dotted, and black-solid outlined boxes, represents a fragment to be executed. The red group will be 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.

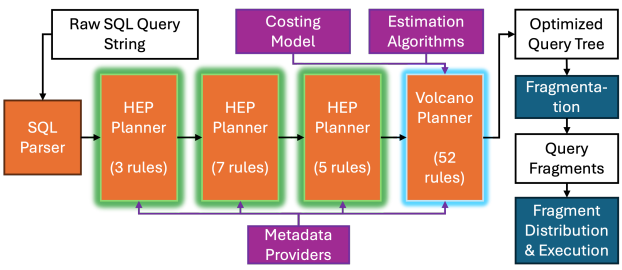


Figure 6: SQL Query Execution Flow

Algorithm 1 Execution Plan Fragmentation

```
procedure ExecPlanFrag(rootNode):
    global fragments ← []
    fragments.add(Fragment(root = rootNode))
    create_fragments(node ← rootNode)
    return fragments
end procedure
procedure create_fragments(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
        create_fragments(child)
    end for
end procedure
```

Figure 6 shows the architectural workflow 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 and black boxes represent information states. The boxes with green outlines constitute the first optimization stage, and the box with the blue outline constitutes the second optimization stage.

4 IMPROVING IGNITE+CALCITE'S QUERY PLANNER

The first goal of this research was to analyze Ignite+Calcite's query planner and consequently to improve its 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

Our experiments revealed multiple problems in the planning code that required attention. The most significant problem was with the join result size estimation algorithm. It had an edge case where, if the estimated cardinality of either join input was very small, the estimated join result cardinality would always be 1. 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. Thus, there would be a chain of joins, each with at least one source having an estimated cardinality of 1. Then, when the planner converted the generic join operation to a specific join method, it chose the most efficient way to perform an $N \times 1$ join, which is a nested-loop join. This produced query plans with a chain of `nested-loop-join` operators predicted to be $N \times 1$ joins. However, when executed, these joins were $N \times M$ 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+Calcite, any non-equi-join operation is limited to using a `nested-loop-join` (although `merge-join` can handle inequality conditions, this is not implemented). Equation 3 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 [37].

$$|(A \bowtie B)| = \frac{|A| \times |B|}{\max(d_A, d_B)} \tag{3}$$

Equation 3 is correct when at least one of the join columns is uniformly distributed [29]. While it cannot be definitively stated that this restriction holds for all joins in Ignite+Calcite, it is believed to be a fair assumption. Furthermore, empirical testing showed estimations from Equation 3 were as good or better compared to the original join estimation algorithm and did not suffer from the issue above.

Another problem was discovered with the HepPlanner in the first planning stage. It was missing an essential rule called `FILTER_CORRELATE` [12], 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.

Finally, 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.

4.2 Cost Model

As shown in Equation 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 shows the cost of a `sort` operator for a relation A with cardinality $|A|$ and width (i.e., column count) $deg(A)$. As Ignite+Calcite is an in-memory system, it is assumed that relation A fits entirely in memory.

$$\begin{aligned} O_{\text{CPU}} &= |A| * RPTC + |A| * \log(|A|) * RCC \\ O_{\text{Memory}} &= |A| * deg(A) * AFS \end{aligned} \tag{4}$$

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

Algorithm 2 Distribution Factor Calculation

```
procedure DistFactorCalc(rootNode)
    if hasExchange(rootNode) then
        return 1
    end if
    return dataPartitionSites(rootNode)
end procedure
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
    return false
end procedure
```

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 methods. Equation 5 shows this change applied to the cost of a `sort` operator.

$$\begin{aligned} O_{\text{CPU}} &= |A| * RPTC + |A| \log(|A|) * RCC \\ O_{\text{Memory}} &= |A| \end{aligned} \tag{5}$$

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 6 shows a distribution factor df applied to the updated `sort` operator cost equation (Equation 5). In this equation, df is determined by using Algorithm 2 on the child of the `sort`.

$$\begin{aligned} O_{\text{CPU}} &= df^{-1}|A| * RPTC + df^{-1}|A| \log(df^{-1}|A|) * RCC \\ O_{\text{Memory}} &= df^{-1}|A| \end{aligned} \tag{6}$$

4.3 Planner Exploration Efficiency

The final issue unearthed by our experiments 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 possible plans as the Cartesian product of logical and physical possibilities, leading to an impossible number of alternatives to explore. Although such single-phase optimization could theoretically discover an optimal plan, it can be expensive and practically infeasible to do so [16]. Our solution was to replace this process with the popular two-phase plan generation and optimization process [15]. This approach generates a logically optimized plan first, followed by a physically optimized plan, which is a practical, systems-oriented, approach that breaks away from the expensive single phase optimization process. 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 (JoinCommuteRule [12]) and permuting the order of a nested join (JoinPushThroughJoinRule [12]), 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 generate a query plan. This was the root cause of multiple planning failures in the original Ignite+Calcite 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 [28] 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.

5 IMPROVING QUERY EXECUTION

Once the query planner, based on our analysis, was stabilized, the next task was to improve the performance of query execution. This task deals with optimizing the execution of `join` operators and increasing the level of parallel computation within execution plans.

5.1 Join Operation Optimizations

This section focusses on optimizations applied to `join` operators.

5.1.1 Fully Distributed Joins. When analyzing the distribution mappings in Table 2, it was apparent that 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:

$$A \bowtie B \equiv A \bowtie B_1 \cup A \bowtie B_2 \cup \dots \cup A \bowtie B_k \ .$$

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 broadcasted, 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 [20]. 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 [38] 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 7 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 the operator processes ($|A| + |B| * df_r^{-1}$). 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. As for Equation 4, it is expected that both relations A and B fit entirely in memory.

$$\begin{aligned} O_{\text{CPU}} &= (|A| + |B| * df_r^{-1}) \ * (RCC + RPTC + HAC) \\ O_{\text{Memory}} &= |B| * df_r^{-1} \end{aligned} \tag{7}$$

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 to only 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. This design follows the principle of the Simple-Hash join algorithm [8]. Section 5.1.3 details the derivation of the CPU cost component. When implementing the `hash-join` operator, the same distribution mappings as the existing join algorithms were used, including the changes in Section 5.1.1.

5.1.3 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, respectively. The CPU cost of a `hash-join` operator H for this join using Equation 7 is:

$$\begin{aligned} H_{CPU} &= df_A * A(RCC + RPTC + HAC) \\ &\quad + B(RCC + RPTC + HAC) \\ &\quad \text{where } A = df_A^{-1}|A| \text{ and } B = df_B^{-1}|B| \ . \end{aligned} \tag{8}$$

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:

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

Combining this with the cost of two `sort` operators from Equation 6 (for the left and right input relations A and B, respectively) yields the total CPU cost of a `merge-join` operator MJ:

$$\begin{aligned} MJ_{CPU} &= A(RCC + RPTC + \log(A) * RCC + RPTC) \\ &\quad + B(RCC + RPTC + \log(B) * RCC + RPTC) \ . \end{aligned} \tag{9}$$

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. As the relations grow, the extra cost of sorting will outweigh the constants in H_{CPU} and `hash-join` will be chosen. However, if one relation's sort costs are removed, the determining factor is the cost of sorting the other relation. If both sorting costs are removed, then the MJ_{CPU} will always be less than H_{CPU}.

If $df_A \neq 1$, then $df_A > 1$. In this case, existing code dictated that relation B is data shipped to build the hash table so $df_B = 1$. By swapping the input relation order of H, a new `hash-join` operator H^* is created with CPU cost:

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

Join operations are commutative excluding attribute ordering [34], 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 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.

5.2 Join Condition Simplification

When running tests initially, Query 19 from TPC-H [28] was disabled because it did not complete execution after 4 hours. After investigation, 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 $6M \times 200K = 1.2 \times 10^{12}$ 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 other conditions 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. This extraction can be repeated until all common conditions are removed, yielding a 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.

5.3 Multi Threaded Execution Plans

Fragments are executed in parallel at different sites to take advantage of the data partitioning functionality in Ignite. This parallelization was improved by creating 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.

5.3.1 Variant Fragment Creation. To create a variant fragment (VF), Algorithm 3 is run with a non-root fragment and the target number of VFs n. Since each VF will run in its own thread, n

Algorithm 3 Variant Fragment Creation

```
procedure VFC(node ← rootNode, type ← SPLITTER)
    if node is source then
        return node.copyWithType(type)
    else if node is reduction operator then
        Raise Exception
    else if node is join then
        newLeft ← VFC(node.left, DUPLICATOR)
        newRight ← VFC(node.right, type)
        return node.copyWithChildren([newLeft, newRight])
    else
        newChildren ← []
        for child in node.children do
            newChildren.add( VFC(child, type) )
        end for
        return node.copyWithChildren(newChildren)
    end if
end procedure
```

represents the number of threads desired. n VFs are created, each assigned a unique ID $v_{id} \in \{0 \ldots n-1\}$, and a copy of the original fragment query tree is created. 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 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 makes 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 into other fragments that merge partial reduction results into the query tree. These changes would add significant complexity to 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 5, 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.

5.3.2 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+Calcite operates entirely in memory, these reads are inexpensive and far outweighed by the benefits of parallel computation.

Again, using Figure 5 as an example, assume 2 VFs were created for the blue fragment, with each VF knowing its v_{id} and the total number n of VFs = 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.

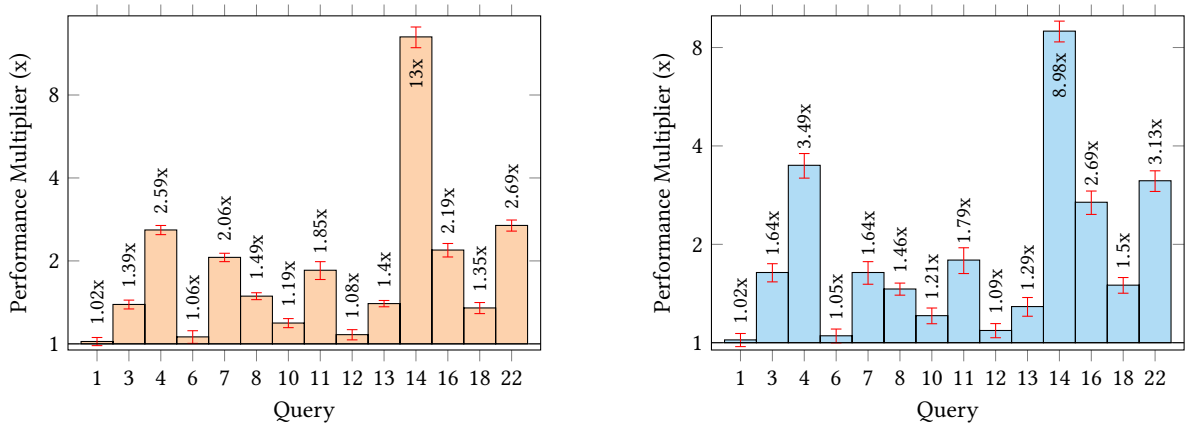
6 EXPERIMENTAL EVALUATION

To evaluate the performance of the system before, and after, the aforementioned changes, the open-source system Benchbase [9] was used with the TPC-H [28] benchmark and the Star Schema Benchmark (SSB) [23]. The goal was to provide a performance evaluation and analysis of a composable system using benchmark workloads. For TPC-H, minor configuration changes were made to Benchbase to ensure compatibility with Ignite+Calcite. SSB was added to Benchbase as a new benchmark[1]. The TPC-H [28] 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+Calcite, and Query 20 contained an unresolved bug in the planning code that caused the query planner to fail.

6.1 Methodology

Each system variant was tested using 4 and 8 machines (sites). Each machine had two Intel E5-2620v2 CPUs (2.1GHz, 12 physical cores, 24 logical cores) with 32GB of RAM, connected by 10 GB ethernet. Ignite+Calcite was configured in a partitioned cache mode with zero backups and statistics enabled. Ignite+Calcite was started on each machine as a Java 11 process (OpenJDK Runtime Environment, build 11.0.11+9-Ubuntu-0ubuntu2.20.04) [10]. The standard Apache Ignite version 2.16 built from source without any improvements was the *baseline* system that we will call IC. Our improvements from Section 4, Section 5.1, and Section 5.2 (Query Planner Changes and Join Optimizations) were implemented to enhance the baseline system, compiled and executed on each machine. We call this improved system IC+. IC+ augmented with multithreading (Section 5.3) is called IC+M. The improvements were tested in these groups because the changes in Section 4, Section 5.1, and Section 5.2 are dependent on one another, but are independant of the multithreading changes from Section 5.3. For

[1]Code is available from 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



(a) IC (4 sites) vs. IC+ (4 sites)

(b) IC (8 sites) vs. IC+ (8 sites)

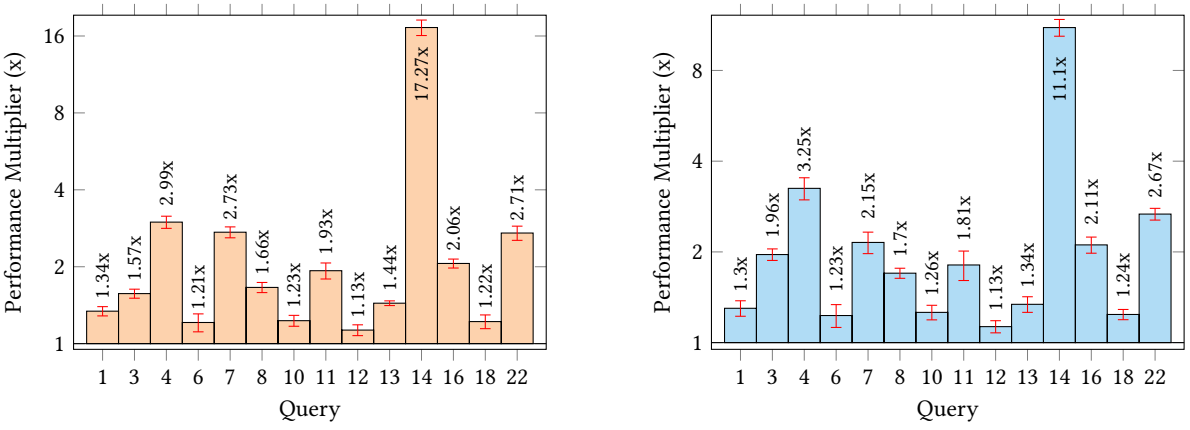
Figure 7: Join Optimizations & Query Planner Performance Improvements over Baseline

each benchmark, every combination of scale factor and system configuration was tested thrice. The average performance gain across all scale factors was used as the performance gain for that configuration. Red error bars in the graphs represent 95% confidence intervals around each mean (data point). Results are presented for two configurations of the number of processing sites, 4 and 8, per graph. Each 4- or 8-site result is compared with its respective (4- or 8-site) baseline. While not explicitly shown, all 8-site configurations consistently outperformed their 4-site counterparts in all tests.

6.2 TPC-H Per Query Response Time

Individual query response time was measured on the three test systems: IC, IC+, and IC+M. A *test* consisted of a warm-up execution followed by three measured executions for each query. The mean response time of the three independent query executions was used as the query's execution time for that test. 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.

6.2.1 Join Optimizations & Query Planner Improvements. Figure 7 shows the performance of IC+ compared to IC for 4 and 8 sites. With the changes incorporated into IC+, there were performance improvements for every query. The biggest gains from these changes were in Queries 4, 14, and 22. Queries 4 and 22 show large 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. Query 14 improved due to a change in the sort order (i.e., sorting order) of an index scan on a base relation. The new sort order changed the aggregation function from hash-map-based to sort-based on an already sorted input, removing an intermediary sort operation entirely. There were also large gains in Queries 7, 8, 11, 13, and 16. In IC, 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 7, 8, 16, and 18 also benefit from the new `hash-join` operator in place of what were initially merge joins on intermediate results. This removed multiple intermediate sort operations from the query tree, decreasing the total operation time and increasing the parallelism since the sort operation cannot be distributed. Queries 3, 10, and 14 all have performance gains owing to the *broadcast* distribution mapping. These queries all perform a join operation with the `LINEITEM` table, the largest



(a) IC (4 sites) vs. IC+M (4 sites)

(b) IC (8 sites) vs. IC+M (8 sites)

Figure 8: Overall Performance Improvement over Baseline

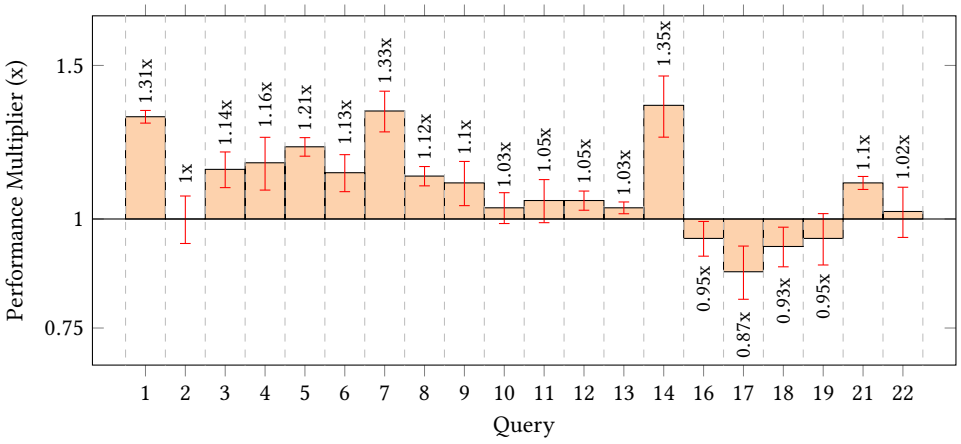


Figure 9: Multithreading Incremental Performance Difference: IC+ (4 sites) vs. IC+M (4 sites)

in the benchmark. With this new mapping, IC+ avoids data shipping this relation entirely, instead broadcasting tuples to the sites containing partitions of this relation as needed.

In Query 12, the input ordering of a join was swapped (i.e., $A \bowtie B$ became $B \bowtie A$), but this did not result in any meaningful performance changes. Queries 1 and 6 produce the same execution plan as the baseline system and show no significant changes. Comparisons for Queries 2, 5, 9, 17, 19 and 21 are not available because they did not complete execution in the IC baseline system. However, all six of these queries completed execution in under one minute on average in IC+, a significant improvement from the timeouts and planning failures seen in IC. For Queries 2, 5, and 9, the main factor was the improved query planning process which could successfully generate execution plans for these complex queries. For Queries 17, 19, and 21, the main factors were the additional optimization rules (Section 4.1 and Section 5.2) and the join optimizations (Section 5.1), yielding far more efficient execution plans.

6.2.2 Overall Performance. Figure 8 shows the performance of IC+M compared to IC. Performance improved for every query and configuration. Queries 2, 5, 9, 17, 19, and 21 are not shown here because the baseline system failed to plan or execute them.

6.2.3 Multithreading. Figure 9 and Figure 10 show the performance changes between IC+ and IC+M, representing the impact of adding multithreading. 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 15% to 35% 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, and 21 all have negligible changes in performance. Queries 9, 10, and 19 are queries in which most of the work is executed in the root fragment, which does not support multithreading. For Query 19, this becomes a bottleneck and causes a slight performance decrease. Queries 12 and 13 have very restrictive filters that limit the amount 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.

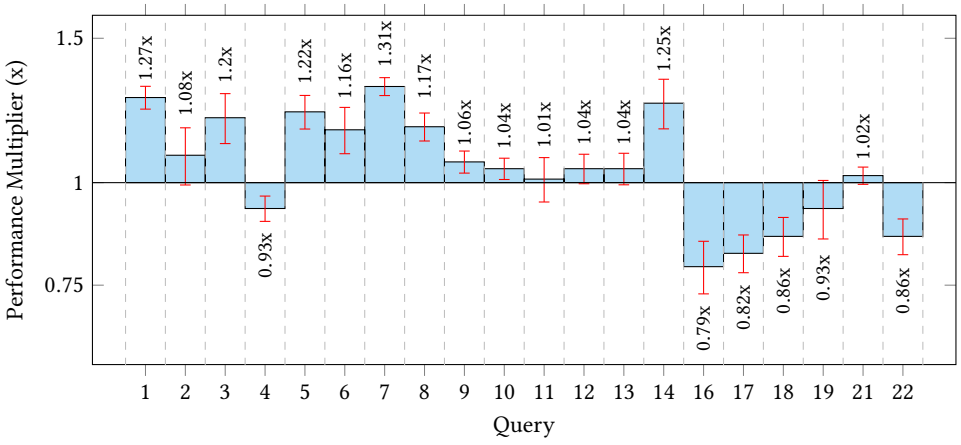


Figure 10: Multithreading Incremental Performance Difference: IC+ (8 sites) vs. IC+M (8 sites)

Table 3: Average Query Latency (seconds) for 4 and 8 Sites

	4 Sites			8 Sites		
Clients	IC	IC+	IC+M	IC	IC+	IC+M
2	9.91 *s*	7.08 *s*	6.87 *s*	7.15 *s*	5.19 *s*	4.97 *s*
4	13.82 *s*	9.25 *s*	9.95 *s*	9.86 *s*	6.21 *s*	6.24 *s*
8	20.24 *s*	13.38 *s*	16.38 *s*	14.98 *s*	9.13 *s*	9.46 *s*

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. There are no operations that benefit from the base relation scans being multi-threaded. Thus, dynamic partitioning is performed only to rejoin everything into a single thread directly after. This overhead increases the wait time between data shipping requests, causing 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 minimal gain from the multi-threaded plans. When run on 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.

6.3 TPC-H Average Query Latency

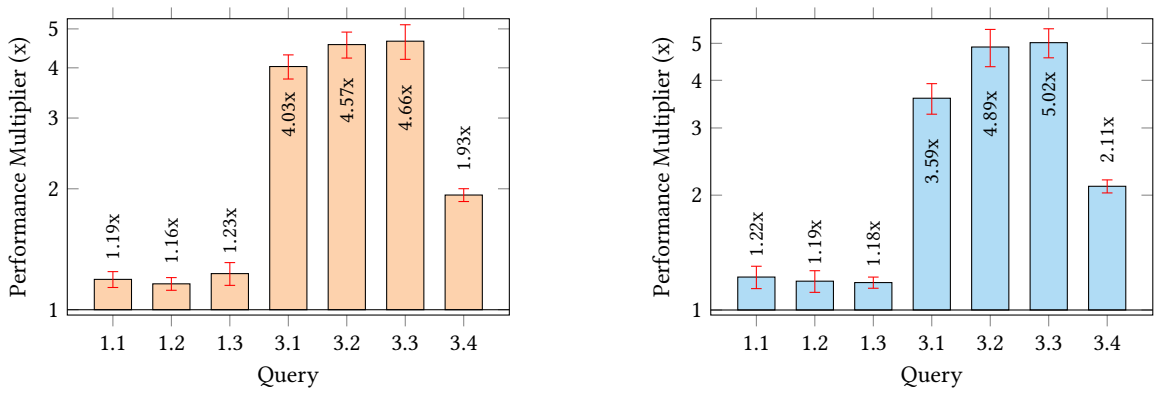
Average query latency (AQL) was measured on the three test systems: IC, IC+, and IC+M. Each system had one warm-up execution of each query before measurement. A test consisted of one or more *terminals* (clients) submitting randomized queries sequentially until the specified time elapses. Tests were 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, IC+) was executed five times, and the average latency 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 this test suite to ensure a fair comparison. Table 3 shows the AQL testing results. Both IC+ and IC+M showed statistically significant decreases in AQL relative to IC. These decreases ranged from 20% – 40% depending on the experiment setting. The largest was a 39% decrease in AQL using IC+ with eight clients and eight sites. The lowest was a 19% decrease using IC+M with eight clients and four sites.

Independently for each system, performance increased (meaning AQL decreased) with an increased number of sites, and decreased (meaning AQL increased) as the number of clients increased, as expected. The performance increase came from more processing sites yielding improved load distribution and parallel processing capabilities. The performance decrease came from an increase in the processing load due to more concurrent client requests. When comparing systems, IC+ consistently outperforms IC for each same-site configuration with an increasing number of clients. In contrast, IC+M outperforms IC+ with two clients but has a decrease in performance as the number of clients increases to four and eight. This is because more clients results in more parallel requests to the point where the number (2×) of concurrent processing threads surpasses the CPU core count resulting in CPU contention.

6.4 Star Schema Benchmark

Our second evaluation used the Star Schema Benchmark [23], a variation of TPC-H modified to more accurately represent a classical data warehousing structure. Nine indexes were created: one on the primary key of each relation, and four on columns of the `LINEORDER` table (`LO_ORDERDATE`, `LO_PARTKEY`, `LO_SUPPKEY`, `LO_CUSTKEY`) used as join conditions. Figure 11 shows the per query response time performance multiplier of IC+M relative to IC for each query. Each value is the average of scale factors 0.5, 1, 2, and 3, for four and eight sites. Both query sets contained a parent query whose parameters were varied to create specific queries (e.g., Q1.1 and Q1.3) that executed on disjoint data.

The largest performance improvements were seen in query set three (QS3). Response times improved by 2×–5× depending on the query variation and was due to multiple factors. The first factor was a difference in the ordering of join operations. The IC+M system performed the joins with more restrictive conditions at the base of the query tree, reducing the number of rows intermediary operations had to process. The second factor was



(a) IC+M (4 sites) vs IC (4 sites)

(b) IC (8 sites) vs IC+M (8 sites)

Figure 11: Star Schema Benchmark Per Query Performance: IC vs IC+M

multiple `hash-join` operations in place of `merge-sort` operations, removing multiple `sort` operations from the query tree. The third factor, the broadcast-distributed join mapping, allowed the largest relation `LINEORDER` to remain in place and smaller relations to be shipped to it.

Query set one (QS1) had moderate performance improvements between 1.1× to 1.25× depending on the variation. This improvement came from the broadcast-distributed join mapping, which allowed the smaller `DATE` relation to be data-shipped instead of the `LINEORDER` relation, the largest in the test bench.

Query sets two (QS2) and four (QS4) were excluded from the SSB test bench. When executing the main optimization phase, QS2 and QS4 both produce search spaces that are too large for Apache Calcite to process, resulting in a process timeout and a failure to generate a fully optimized query plan. For QS4, this happens on both the baseline and modified systems due to the number of nested join operations (QS4 is a 5-way join). For QS2 this happens on the modified system due to the additional join algorithm (hash-join) and join distribution mappings (fully distributed joins). In both cases, this is a limitation of the Calcite library itself and thus not a focus of this research.

7 EXPERIENCES AND TAKEAWAYS

Throughout the course of this research, much insight was gained into Ignite and Calcite as components of a composable system. Our takeaways can be organized around four key areas that we consider essential for components of any successful composable system: *modularity, interoperability, developer support*, and *documentation*.

Modularity, the degree to which a piece of software is divided into reusable components, is the core feature of every composable system. Apache Calcite was designed with modularity as the key consideration, and this is very evident. All of its features (e.g., the VolcanoPlanner, the HepPlanner, and the SQL Parser) can be used independently of one another and are highly configurable. Apache Ignite also has a high degree of modularity, from its large feature offering to its design. The internal modules that control different features (e.g., data storage, message streaming, and distributed computation) can all be selectively enabled and have programming interfaces in multiple modern programming languages. Thus, both Ignite and Calcite have a high degree of modularity, which make them easy to integrate into new and existing systems.

Interoperability, the ability of a system to operate and communicate with other systems, is another core aspect of a composable system. Ignite shines here, supporting multiple industry-standard communication protocols (e.g., REST API's, JDBC) and maintaining client libraries in multiple programming languages (Java, C++, Python, Node.JS, PHP). Apache Calcite, however, is more limited in the systems it can interface with as it is developed solely as a Java package. Wrappers can always be used to port Calcite's functionality to different languages, but a lack of out-of-the-box solutions limits the breadth of systems with which it can be integrated. This is particularly relevant for industry standard databases (e.g., MySQL, PostgreSQL, Oracle Database, and MongoDB) that could benefit from Calcite but are not written in Java.

Developer support and documentation are both essential pieces for any component in a composable system. Without these, it would be nearly impossible to combine unfamiliar components into a coherent system. Apache Ignite has excellent user documentation which allows easy integration into other systems. However, its developer documentation (e.g., README files and code comments) is quite sparse. As open-source software that relies on public contributors, this can pose a challenge for making improvements by those not familiar with the codebase. As a more consumer facing product, Ignite does not have a large developer support community. The methods of support available publicly are limited to GitHub issues, a developer e-mail forum, and Atlassian's Jira that is closed-source. Calcite also falls short on these categories, having the same level of developer support and little documentation. The documentation that does exist is limited to technical documentation (JavaDocs) with minimal explanations/examples and no significant user guides. As software designed to be used by other developers, this limitation can present a barrier for adoption and is a key shortcoming of Apache Calcite.

It should be noted that there are significant benefits to be gained from building a system using composable components. By using a composable system, the knowledge it imbues from the skills, technical expertise and effort that developers have invested into the design of each component is effectively leveraged. For open-source projects, this can encompass the time of many developers working on each component, exceeding that of what many operations could provide if the system were to be built from scratch. Another potential benefit is a shorter turnaround time for new projects. A minimum viable product could be returned more quickly by a small number of developers as they need to only compose components together. This provides a viable opportunity for creating production-level systems without large budgets. A third potential benefit of composing systems is that if much of the system testing is completed by developers of individual components, then only the outputs that come from combining the components together need to be validated. Thus, this approach leverages prior testing and validation of individual components of the target composable system.

8 CONCLUSION

This paper experimentally studied composable data management through the composition of Apache Ignite and Calcite system components to process OLAP workloads. It provided an analysis of these components used to execute SQL queries. Leveraging this analysis, three distinct strategies to improve performance were implemented: improving the stability and efficiency of the query planner, adding new operations for executing queries, and multi-threaded query execution support. Each strategy was experimentally evaluated. With all strategies combined, significant query response time improvements were obtained ranging from 1.2× to 17× faster than the baseline Ignite+Calcite system. This work highlights the potential of composable systems to deliver functionality for processing data system workloads and the importance of experimental benchmarking and analysis when developing them.

ACKNOWLEDGMENTS

This work was supported by a Salesforce Research Award, Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation and Ontario Research Fund.

REFERENCES

[1] Murali Ande and Narendra Paruchuri. [n. d.]. 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. 2013. Multi-core, main-memory joins: sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96. https://doi.org/10.14778/2732219.2732227

[3] Cağrı Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1754–1766. https://doi.org/10.1109/TKDE.2014.2313874

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. 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)*. Association for Computing Machinery, 221–230. https://doi.org/10.1145/3183713.3190662

[5] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. 1981. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems* 6, 4 (1981), 602–625. https://doi.org/10.1145/319628.319650

[6] Surajit Chaudhuri. 1998. 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)*. Association for Computing Machinery, 34–43. https://doi.org/10.1145/275487.275492

[7] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. Association for Computing Machinery, 1–8. https://doi.org/10.1145/602259.602261

[9] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288. https://doi.org/10.14778/2732240.2732246

[10] Mark Dodds. 2025. Publication Resources. Retrieved Feb 17, 2025 from https://github.com/marktdodds/ignite/blob/ignite-2.16/publication_resources.md

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

[12] The Apache Software Foundation. 2024. Apache Calcite Java Documentation. Retrieved Dec 19, 2024 from https://calcite.apache.org/javadocAggregate/

[13] The Apache Software Foundation. 2024. Calcite-based SQL Engine. Retrieved Dec 19, 2024 from https://ignite.apache.org/use-cases/provenusecases.html

[14] The Apache Software Foundation. 2024. Proven Business Use Cases. Retrieved Dec 19, 2024 from https://ignite.apache.org/use-cases/provenusecases.html

[15] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[16] Goetz Graefe. 1993. Query evaluation techniques for large databases. *Comput. Surveys* 25, 2 (1993), 73–169. https://doi.org/10.1145/152610.152611

[17] Goetz Graefe and William J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. https://doi.org/10.1109/ICDE.1993.344061

[18] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *ACM Computing Survey* 16, 2 (1984), 111–152. https://doi.org/10.1145/356924.356928

[19] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[20] Donald Kossmann. 2000. The state of the art in distributed query processing. *Comput. Surveys* 32, 4 (2000), 422–469. https://doi.org/10.1145/371578.371598

[21] Arunprasad P. Marathe, Shu Lin, Weidong Yu, Kareem El Gebaly, Per-Åke Larson, and Calvin Sun. 2022. Integrating the Orca Optimizer into MySQL. In *Proceedings of the 25th International Conference on Extending Database Technology (EDBT), Edinburgh, UK, 29th March-1st April, 2022* (2022). OpenProceedings.org. https://doi.org/10.48786/EDBT.2022.45

[22] Priti Mishra and Margaret H. Eich. 1992. Join processing in relational databases. *Comput. Surveys* 24, 1 (1992), 63–113. https://doi.org/10.1145/128762.128764

[23] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[24] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zaït (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper1_MoshaPasumansky.pdf

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

[26] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[27] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4249–4252. https://doi.org/10.14778/3685800.3685847

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

[29] Arnon S. Rosenthal. 1981. Note on the expected size of a join. *SIGMOD Record* 11, 4 (1981), 19–25. https://doi.org/10.1145/984488.984491

[30] J. B. Rothnie, P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. 1980. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems* 5, 1 (1980), 1–17. https://doi.org/10.1145/320128.320129

[31] Donovan A. Schneider and David J. DeWitt. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *ACM SIGMOD Record* 18, 2 (1989), 110–121. https://doi.org/10.1145/66926.66937

[32] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, 1961–1976. https://doi.org/10.1145/2882903.2882917

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

[34] Abraham Silberschatz, Henry Korth, and S. Sudarshan. 2019. *Database System Concepts* (7th ed.). McGraw-Hill Education. 1376 pages.

[35] Cristiana-Stefania Stan, Adrian-Eduard Pandelica, Vlad-Andrei Zamfir, Roxana-Gabriela Stan, and Catalin Negru. 2019. Apache Spark and Apache Ignite Performance Analysis. In *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. 726–733. https://doi.org/10.1109/CSCS.2019.00129

[36] Substrait. 2024. Substrait: Cross-Language Serialization for Relational Algebra. [Online]. Available: https://substrait.io/. Accessed: 2024-10-05.

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

[38] Jingren Zhou. 2009. Hash Join. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer US, 1288–1289. https://doi.org/10.1007/978-0-387-39940-9_869

[39] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems* (4th ed.). Springer. https://doi.org/10.1007/978-3-030-26253-2