PeerJ Computer Science

Dynamic parallelization in distributed join optimization

Fatih Türkmen and Belgin Ergenç Bostanoğlu

Computer Engineering, Izmir Institute of Technology, Izmir, Turkey

ABSTRACT

Selection of appropriate level of parallelism is critical in distributed join optimization for efficient execution of data-intensive query workloads. Different join strategies use different communication patterns, which requires the level of parallelism must be determined for each strategy. Although most cost-based join optimizers have advanced in modeling local computation and network communication costs, they still struggle to precisely determine parallelism levels tailored to different join strategies. In addition, the impact of data replication levels on join performance is often overlooked, despite its critical role in improving data availability during execution. We propose a cost model for optimizing join queries that supports multiple distributed join methods in data-intensive processing environments. In our approach, the optimal level of parallelism for each join strategy is determined individually based on key factors such as data size, replication level, and query complexity. Then, the strategy with the lowest cost under its own optimal parallelism configuration is selected for execution. DPJoin (Dynamic Parallelization Based Join Query Optimization), our proposed method, uses a saturation-based approach to estimate strategy-specific parallelism-based cost, leveraging adaptive runtime statistics to optimize the physical plan. Experimental findings indicate that DPJoin delivers the best performance in analytical queries, achieving an average reduction of approximately 9% in execution time compared to the closest baseline strategy. In addition, DPJoin achieves faster execution times of up to 60% on low and medium scale data under a full data replication setting.

Submitted 2 October 2025
Accepted 29 December 2025
Published 12 February 2026

Corresponding author
Fatih Türkmen,
fatihturkmen@iyte.edu.tr

Academic editor
Sándor Szénási

Additional Information and Declarations can be found on page 34

DOI **10.7717/peerj-cs.3614**

Subjects Computer Architecture, Databases, Digital Libraries, Distributed and Parallel Computing, Optimization Theory and Computation

Keywords Distributed query optimization, Join selection, Join optimization in distributed query optimization, Parallelism level, Replication level

INTRODUCTION

Join operations combine multiple datasets by matching records on a common key. This process is often computationally intensive. In distributed environments, data frequently needs to be transferred between nodes. As a result, efficient join optimization becomes critical for performance. Recent work on distributed relational systems highlights this need (*Du, Cai & Ding, 2024*). Optimization is especially important at large scales, where partitioning, communication overhead, and resource contention strongly influence execution.

Cost-based query optimization is a well-established approach in join processing. Most mainstream systems adopt this approach. PostgreSQL uses cost-based planning even in its distributed extensions, such as Citus (*Cubukcu et al., 2021*). Spark SQL follows the same

idea. Previous work extended Spark's optimizer by integrating runtime statistics and adaptive mechanisms. These additions were incorporated into a cost-based join framework known as RelJoin (*Liang et al., 2024*). Cost-based methods in distributed settings typically estimate computation and communication costs to select join strategies. These factors are important, but existing models often overlook two additional dimensions: the degree of parallelism and the replication factor of data. These dimensions were identified as critical in early MapReduce-based systems (*Sarma et al., 2013*). They affect throughput, shuffle overhead, and coordination complexity. Therefore, they deserve greater attention in distributed join optimization, as also noted by *Polychroniou, Zhang & Ross (2018)*.

Although distributed join queries rely on the same logical operations as local joins, their physical execution methods are more complex. This complexity arises from inter-node data movement and coordination overhead. Unlike local joins, which primarily involve selecting between sort-merge and hash join strategies, distributed joins include a data exchange phase. In this phase, data needs to be moved across nodes in the cluster before any local join operation can take place. This exchange can be performed using different strategies, most notably broadcasting smaller tables or shuffling data based on join keys. Each of these strategies exhibits different performance and cost behaviors. As a result, the selection process is more critical than in the execution of the local join. An inappropriate partitioning choice in distributed joins can lead to inefficient shuffle operations. Such operations increase network overhead. They also lead to suboptimal degrees of parallelism, which reduces overall query performance. Empirical studies on distributed join optimization confirm this effect (*Liang et al., 2018*). Another key difference is the cost of network communication, which constitutes a substantial portion of the total query cost in distributed systems. Depending on the amount of data to be transferred, this cost can become a significant factor. As noted in research on approximate join optimization (*Quoc et al., 2018*), the challenge in network cost estimation makes accurate cost estimation more challenging. In addition to these challenges, the degree of parallelism used during execution plays a crucial role in determining coordination overhead. In contrast to local environments, distributed systems must manage task execution across multiple nodes. Consequently, excessive parallelism can significantly increase coordination cost. This makes it a non-negligible factor in overall system performance. Similar observations have been reported in distributed SQL engines such as Presto, where incorrect parallelism settings or poor resource allocation significantly degrade query performance. These findings underscore the importance of adaptive resource-aware tuning (*Gourishetti, 2024*).

In this study, we address a fundamental limitation in distributed join query optimization by introducing DPJoin (Dynamic Parallelization Based Join Query Optimization). This strategy explicitly incorporates parallelism and data replication into the cost estimation process. Although approaches like RelJoin (relative-cost-based selection of distributed join methods for query plan optimization) have shown the effectiveness of using runtime adaptive statistics for join selection in distributed systems (*Liang et al., 2024*), they do not explicitly model the interplay between replication and parallelism in determining execution costs. Building on this gap, our approach introduces a parameterized cost model that balances computation, communication, and coordination

overhead. This balance allows the optimizer to dynamically determine the most suitable execution configuration. The design is inspired in part by elasticity-aware systems like ElasticJoin (*Zhang, Zhang & Meng, 2025*), which adjust join strategies based on cluster load and runtime conditions. DPJoin evaluates multiple levels of parallelism under different replication conditions. This evaluation helps identify the most efficient strategy for the physical and workload characteristics of the distributed environment. As highlighted in data-adaptive systems research (*Giampà et al., 2021*), this model bridges a critical shortcoming in conventional cost-based optimizers. These optimizers often treat parallelism and replication as static assumptions rather than dynamic optimization parameters.

Building on this, we integrated the proposed DPJoin strategy into the Apache Spark framework. The implementation leverages Spark's Catalyst Optimizer and dynamically collects runtime statistics such as data size and query plan complexity. To evaluate the effectiveness of DPJoin, we designed a series of experiments focusing on analytical queries that involve complex join operations. Our experimental setup systematically varies the volume of input data, the replication ratio between worker nodes, and the degree of parallelism to simulate various execution scenarios. These variations allow us to benchmark DPJoin against standard physical join strategies such as Shuffle Hash Join and Shuffle Sort Merge Join, as well as the adaptive RelJoin optimizer (*Liang et al., 2024*). This study makes the following contributions:

- We introduce DPJoin (Dynamic Parallelization Based Join Query Optimization), a novel distributed join optimization strategy that incorporates degree of parallelism and replication factors in addition to data size into a cost-based decision process. Unlike traditional join strategies, DPJoin explicitly models the interaction between replication ratio, parallelism level, and input data size, which influences execution cost in distributed environments.
- We implement DPJoin in Apache Spark by extending the Catalyst optimizer. It includes a saturation-aware component to select optimal parallelism and a cost-based strategy selector that chooses the best join method using runtime statistics.
- We conduct extensive experiments, evaluating performance across varying replication levels, data sizes, and query structures. Our results show that DPJoin consistently outperforms baseline strategies such as RelJoin, reducing the average query execution time and achieving an improvement of up to 60% over shuffle-based join strategies in the best-case scenario.

RELATED WORK

Cost-based query optimization forms the backbone of modern relational database systems. It enables the selection of efficient execution strategies based on estimated resource costs. This paradigm was first introduced by *Selinger et al. (1979)* in their pioneering work at IBM. They developed a systematic cost-based dynamic programming framework within the experimental System R project. Their optimizer, now known as the Selinger-style

optimizer, marked a major shift in query planning. It pioneered the cost-based approach and replaced traditional rule-based strategies with a statistically grounded framework. The core architecture of this approach integrates metadata such as table cardinalities, index selectivity, and operator-specific cost functions. It combines these elements into a unified search procedure for join ordering and access path selection. A key innovation was the memoization of optimal subplans during query plan enumeration. This reduced redundant computation and enabled efficient exploration of multi-way join combinations. This foundational design has strongly influenced the evolution of query optimizers. It remains central to systems such as IBM DB2, Oracle, SQL Server, PostgreSQL, and Apache Spark SQL. It also laid the foundation for extensible optimization architectures such as the Volcano Optimizer Framework (*Graefe & McKenna, 1993*) and the Cascades Framework (*Graefe, 1995*). These systems generalized the dynamic programming paradigm for broader and more flexible plan exploration.

As cost-based optimization matured in centralized relational systems, research increasingly shifted toward the challenges of distributed data processing. This shift was driven by the limitations of traditional techniques in large-scale environments. The growing scale of data and the need for high-throughput query processing led to the development of distributed and parallel database systems. These developments required the adaptation of cost-based techniques to new execution settings. One of the earliest and most pressing challenges in distributed systems was minimizing communication cost. This factor quickly became the dominant influence on query performance. To mitigate this, strategies such as semi-joins (*Bernstein & Goodman, 1981*) and join indices were proposed to reduce the amount of data exchanged across the network. These techniques helped reduce communication overhead. With the emergence of shared-nothing architectures and parallel database systems in the late 1980s and early 1990s, join optimization techniques continued to evolve. They expanded to include hash-based partitioning and distributed sort-merge joins. Systems like Gamma (*DeWitt et al., 1986*) demonstrated how intra-query parallelism could be exploited to improve scalability. Later frameworks such as Hadoop and Apache Spark extended these principles to cluster-scale data analytics by distributing computation across multiple nodes and stages.

In recent years, distributed query optimization has shifted toward adaptive and data-aware approaches that use runtime feedback and handle resource heterogeneity. This shift reflects the limitations of static, compile-time assumptions. Techniques such as Adaptive Query Execution (AQE) in Apache Spark (*Zaharia et al., 2016*) and reinforcement learning–based join planners (*Wu & He, 2023*) move beyond static models. These techniques instead respond dynamically to execution-time uncertainties. Recent studies also emphasize that traditional cost models must evolve. These works show that dynamic parallelism, data replication, and coordination overhead are critical factors for robust performance in modern distributed systems.

Despite these advancements, many cost-based optimizers still rely on assumptions developed for centralized environments such as fixed parallelism levels, uniform data distribution and homogeneous resources. These assumptions often fail to hold in distributed contexts. Execution characteristics in such settings are strongly influenced by

dynamic parallelism configurations, data skew, replication strategies, and cluster heterogeneity. Conventional models that focus only on join ordering and access path selection become insufficient for robust performance at scale. Distributed processing frameworks like Apache Spark attempt to address these limitations through runtime re-optimization and more granular control over physical execution plans. However, empirical evaluations (*Ren et al., 2018*) show that these mechanisms still fall short in capturing the full range of resource variability and cost trade-offs inherent in distributed query execution.

Starting in the early 2000s, as data volumes grew and hardware clusters became increasingly affordable, research in cost-based join optimization gradually expanded its focus from centralized databases to distributed and parallel environments. This shift reflected the need to handle data at much larger scales. Foundational work in distributed query processing, including *Schneider & DeWitt*'s *(1989)* study of parallel joins in shared-nothing systems and *Graef & McKenna*'s *(1993)* Volcano optimizer framework, demonstrated how partitioning and pipelining could be leveraged to achieve intra-query parallelism. As commercial and academic interest in scalable data analytics grew, research increasingly emphasized the role of parallel execution plans and their scheduling, as shown by *Garofalakis & Ioannidis (1996)*. It also examined challenges related to resource contention and distributed join techniques. These advances laid the groundwork for more sophisticated systems, exemplified by the introduction of MapReduce (*Dean & Ghemawat, 2008*) and later Apache Spark (*Zaharia et al., 2012*). In these systems, the resilient distributed dataset (RDD) abstraction highlighted new cost-based optimization challenges.

The 2010s saw the emergence of adaptive and feedback-driven approaches, marking a significant shift from static cost models to runtime-aware optimization frameworks. Traditional optimizers relied heavily on compile-time statistics, but these statistics often failed to represent actual execution conditions due to data skew, outdated metadata, or workload variability. As observed in IBM DB2 (*Stillger et al., 2001*) and Microsoft SQL Server (*Markl et al., 2004*), commercial systems began integrating adaptive mechanisms capable of modifying query plans during execution. These mechanisms included query re-optimization, dynamic join reordering, and cardinality feedback loops that updated selectivity estimates on the fly.

On the academic front, many studies explored how runtime statistics can be used to adaptively refine cost models and execution strategies. Prior research has shown that updating cost parameters during execution can significantly improve accuracy under varying workloads. *Pavlopoulou, Carey & Tsotras (2023)* extended AsterixDB with runtime re-optimization to demonstrate how execution feedback can improve join ordering compared to static plans. These findings highlight runtime adaptivity as a key design principle for modern distributed query optimizers.

Among the most notable applications of these ideas is Adaptive Query Execution (AQE) in Apache Spark (*Zaharia et al., 2016*). It supports dynamic modifications based on runtime feedback. Studies evaluating AQE (*Ren et al., 2018*) indicate that Catalyst's optimization rules and cost-based reasoning often provide only marginal improvements for distributed joins. Similarly, *Zhao & Chen (2022)* proposed a runtime optimization

technique for Spark SQL that uses Bloom filters to refine the planning of joins. The optimization strategy showed that lightweight runtime statistics can significantly reduce data transfer overhead. Although AQE represents a major step toward runtime-aware query optimization, it remains largely rule-based and reactive. It also lacks a unified cost model that proactively accounts for parallelism, replication, and cluster heterogeneity. Consequently, AQE often fails to fully exploit execution flexibility or reason about system-wide trade-offs. Recent studies emphasize Spark-specific tuning strategies, including adaptive partitioning, shuffle compression, skew mitigation, and dynamic resource allocation. To address these limitations, adaptive and resource-aware techniques have been proposed. For example, RelJoin (*Liang et al., 2024*) applies adaptive cost models in distributed contexts by leveraging runtime signals such as network throughput and computation overhead to guide join planning dynamically. However, even in such frameworks, variables such as degree of parallelism and data replication are often treated as static configuration parameters. When these variables are static, the optimizer cannot fully capture the interaction between join parallelism, data movement, and coordination overhead.

Other approaches attempted to incorporate elasticity and workload awareness into join optimization. Elastic Join (*Zhang, Zhang & Meng, 2025*), for example, adjusts join strategies based on the load on the real-time cluster and integrates these dynamics into its cost model. However, it treats parallelism reactively and lacks an explicit model of how replication increases shuffle cost. Similarly, the cross-query optimization technique proposed by *Wu & He (2023)* improves resource efficiency by coordinating shared data usage.

Runtime optimization methods, such as those by *Zhao & Chen (2021)* and *Phan et al. (2021)*, use data summaries and Bloom filters to plan joins without relying on precomputed statistics. These methods are efficient and adaptive. However, they rely only on limited local information. Similarly, hardware-aware techniques such as *Neumann*'s *(2011)* LLVM-based query compiler improve cache usage and instruction-level parallelism. However, because they are designed mainly for single-node systems, they are not suitable for handling distributed concerns such as shuffle overhead or task spillover.

Recent studies show that parallelism and replication are important for distributed join performance. They also point out that these factors must be included in plan selection to achieve scalable and efficient execution. By studying how different levels of parallelism and replication affect computation, communication, and shuffle costs, optimizers can choose better strategies. These strategies work well on each node and also adapt to the overall system. Previous research often treats parallelism and replication as fixed system settings, not as variables that can be tuned during optimization. However, treating these factors as part of the plan search space helps the optimizer use system resources more effectively and reduce shuffle and contention costs. This view also shows that the optimizer must reason dynamically about different combinations of replication levels and parallelism.

In addition, approaches like Elastic Join (*Zhang, Zhang & Meng, 2025*) adapt plans when resources change, which gives them some elasticity. However, their adaptation is mostly reactive and lacks a predictive model that explains how replication and parallelism

together affect shuffle behavior. Combining data transfer estimates with replication effects can give a clearer view of when broadcasting is possible and how much data needs to be exchanged.

In summary, even though cost-based optimization has advanced in distributed systems, most existing approaches still do not model parallelism and replication clearly. These factors, together with communication and coordination overhead, strongly influence query performance at large scale. Addressing this gap requires new optimization frameworks that treat parallelism and replication as key decision parameters rather than fixed assumptions. This approach allows more adaptive and resource-efficient planning in distributed environments.

TECHNICAL BACKGROUND

This chapter begins with the Distributed Query Processing, which explains step by step how a query is executed in a distributed environment. It follows the process from query submission through logical and physical planning, task execution, and result collection. The next subsection, Physical Implementations in Distributed Join Optimization, focuses on concrete execution strategies for distributed joins (*Jayashree, 2013*). It describes the two main data exchange methods, broadcast and shuffle, along with common local join algorithms including hash join, sort-merge join, and nested loop join. The chapter then presents detailed execution flows for Broadcast Hash Join and Shuffle Sort Join using diagrams to illustrate data movement and processing phases. These examples demonstrate how physical implementations work in practice and how execution choices influence performance in distributed systems. All cost models presented in this section are taken from the RelJoin study (*Liang et al., 2024*), which serves as the baseline reference for the optimization of distributed joins.

Distributed query processing

Figure 1 illustrates the distributed query processing workflow in a modern data processing engine. The process begins with the submission of a high-level SQL query by the user. Let us assume that we have students table with columns id, name and age and the following query is executed:

```
"Select name from students where age > 23"
```

The query is first parsed by the coordinator node, resulting in a raw logical plan. At this stage, the plan only reflects the syntactic structure of the query; no semantic resolution has yet been performed. In particular, the system does not yet know whether the table `students` exists or whether the referenced columns `name` and `age` are valid. The column types and table metadata are also not yet available. A sample raw plan may look like:

```
UnresolvedProject[name]
      UnresolvedFilter[age > 23]
          UnresolvedRelation[students]
```

Once the raw plan is generated, the analyzer component of the query engine is responsible for performing semantic resolution. During this phase, the system verifies the

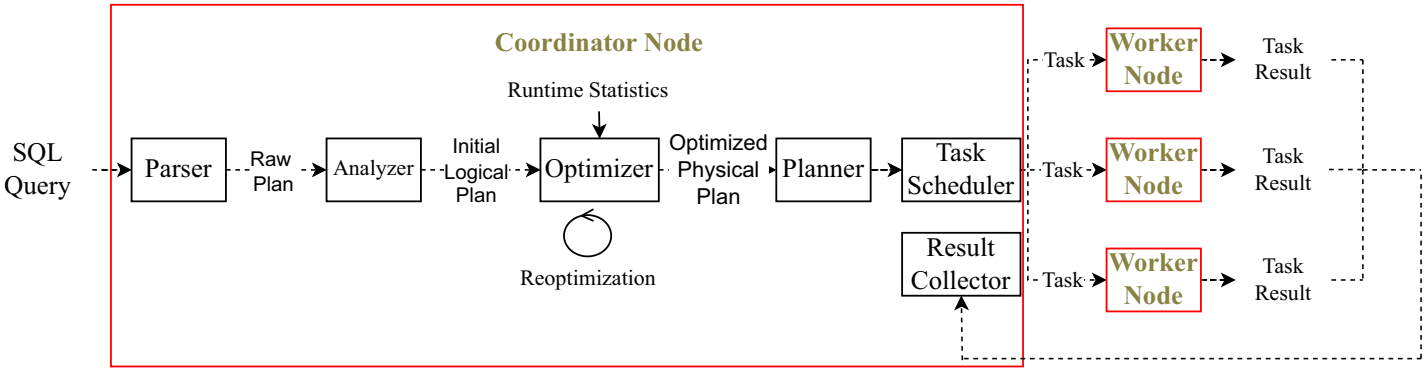

Figure 1 **Distributed query processing.** Full-size DOI: 10.7717/peerj-cs.3614/fig-1

existence of the referenced table and columns by consulting the catalog. It also resolves attribute names to their fully qualified forms, determines data types, and ensures that all expressions are valid. The previously unresolved operators are replaced with their resolved counterparts.

For the running example, the analyzer will resolve the plan into the following initial logical representation:

```
Project[students.name]
    Filter[students.age > 23]
        Scan[students]
```

At this stage, all table and column references are fully qualified and valid. The plan is now semantically meaningful but still abstract in the sense that no concrete execution strategy has been chosen yet. It serves as the input for the logical optimizer, which will apply transformation rules to improve efficiency.

Once the logical plan is constructed, it is passed to the optimizer, which applies rule-based and cost-based transformations to improve execution efficiency. These may include predicate pushdown, projection pruning, and join reordering.

In distributed systems, the optimizer can use runtime statistics like output sizes or data skew to improve its decisions. Adaptive query engines may even change the plan during execution to match the actual data.

For our example, the optimizer might push down the filter and choose an efficient scan operator, producing a physical plan such as:

```
ProjectExec[name]
    FileScanExec students
      [PushedFilters: [IsNotNull(age), GreaterThan(age,23)]]
      [DataFilters: [age > 23]]
```

This plan serves as the basis for generating parallel execution tasks, which are dispatched across the cluster. At this stage, the system decides whether to use broadcast or shuffle for data exchange. It also selects concrete execution operators. For example, it

chooses the join algorithm and scan method based on data size and format. These decisions complete the physical plan before execution begins.

Once execution starts, the plan is split into multiple tasks. Each task is assigned to a worker node. The coordinator node communicates with each worker to send task instructions and monitor their progress. For every task, there is a control flow between the coordinator and the corresponding worker.

As tasks run, they process their input partitions and produce partial results. These results may be shuffled, written to disk, or held in memory, depending on the execution plan. After all tasks complete, their outputs are collected by the coordinator. If further aggregation or sorting is required, final-stage tasks are executed. The final result is either written to storage or returned to the user.

Physical implementations in distributed join optimization

In distributed join optimization, a physical implementation specifies *how* a logical join executes within a distributed environment. It describes the mechanisms of data movement and the procedures for combining rows on each worker node. Each plan pairs a data exchange pattern such as *broadcast* (replicating a small input) or *shuffle* (repartitioning both inputs by the join key) with a local join algorithm such as *hash*, *sort-merge*, or *nested loop*. The optimizer selects among these options based on input sizes, data distribution, and available resources.

Broadcast replicates a small dataset to all workers, enabling local joins without shuffling the large side and thus reducing network cost when the small input fits in memory. In Fig. 2, *L1*, *L2*, and *L3* are the large-input partitions that remain on their original workers, while *S1*, *S2*, and *S3* (*e.g.*, keys 1–3, 4–6, 7–9) are replicated to every worker for local processing.

Shuffle repartitions both inputs by the join key so that matching records colocate on the same partition. In Fig. 3, the exchange uses hash mode 3: each row is routed to partition $p = h(\text{key}) \bmod 3$. Thus keys with the same remainder (*e.g.*, $3, 6, 9, \ldots$; $1, 4, 7, \ldots$; $2, 5, 8, \ldots$) are sent to the same worker, enabling local processing after the exchange. This scales to large inputs but incurs two-sided network and coordination costs and may degrade under key skew.

Hash, sort, and nested loop joins are the common local algorithms: hash join builds a hash table and probes (fast for equi-joins with sufficient memory), sort-merge join sorts both sides then merges (robust at very large scale or with pre-sorted data), and nested loop compares all pairs under a predicate (useful for small inputs or complex/non-equi predicates).

Cartesian product returns all row pairs when no join condition exists; it is typically impractical for large inputs due to quadratic output.

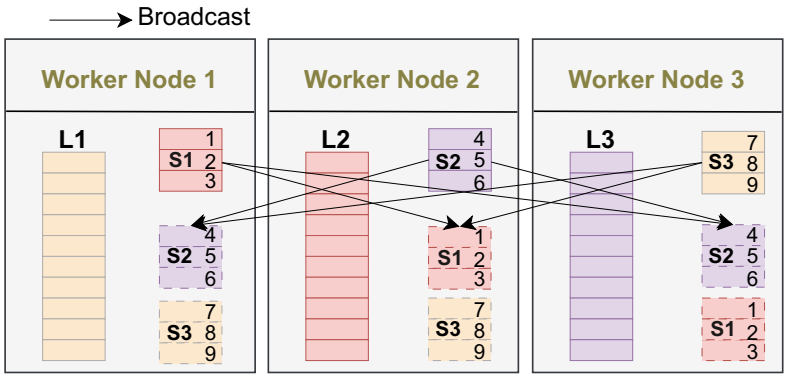

Figure 2 **Broadcast: small input replicated to all workers.** Full-size DOI: 10.7717/peerj-cs.3614/fig-2

Broadcast Hash Join (BHJ). Broadcast the small input; each worker builds a hash table on it and probes with its local partition of the large input. The cost model for BHJ is given in Eq. (1).

$$C_{\text{BHJ}} = (P \cdot S + L + R) + w \cdot S \cdot (P - 1). \tag{1}$$

where P is the number of partitions, S is the size of the small table, L and R are the sizes of the left and right tables, representing local scan/probe and result materialization costs, and w weights communication relative to computation.

Shuffle Hash Join (SHJ). Hash-partition both inputs on the join key and shuffle so matching keys meet on the same partition; each worker builds and probes a per-partition hash table. The cost model for SHJ is given in Eq. (2).

$$C_{\text{SHJ}} = (L + R + S) + w \cdot \tfrac{(L+R)(P-1)}{P}. \tag{2}$$

where P is the number of partitions, L and R are the sizes of the left and right tables, representing local scan/probe costs, S is the smaller table size, and w weights communication relative to computation. It is suitable for large equi joins when neither side is broadcastable; limited by two-sided shuffle and key skew.

Sort-Merge Join (SMJ). Shuffle both inputs by the join key, sort each partition locally, and merge the sorted streams to match equal keys. The cost model for SMJ is given in Eq. (3).

$$C_{\text{SMJ}} = \left(L \cdot \log_2 \frac{LR}{P} + R \cdot \log_2 \frac{RR}{P} + L + R\right) + w \cdot \frac{(L+R)(P-1)}{P}. \tag{3}$$

where P is the number of partitions, L and R are the sizes of the left and right tables, representing local scan/probe costs, LR and RR are row counts, and w weights communication relative to computation.

Broadcast Nested Loop Join (BNLJ). Broadcast a small input; each worker scans its local partition of the other input and evaluates the (possibly non-equi) predicate against all broadcast rows. The cost model for BNLJ is given in Eq. (4).

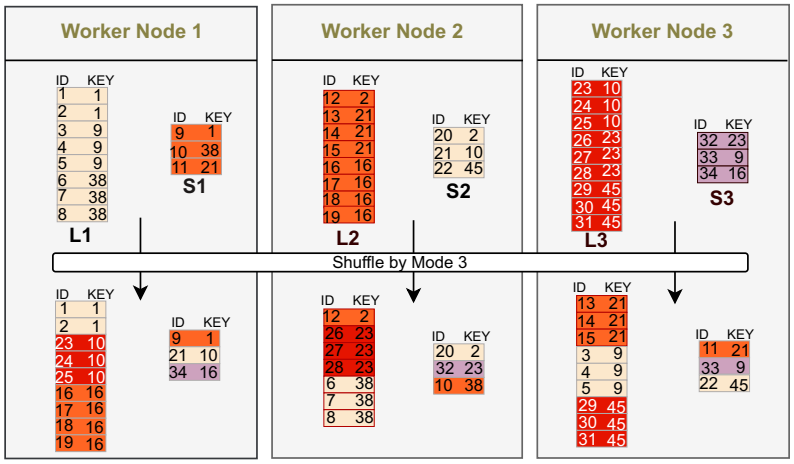

Figure 3 **Shuffle (mode 3): $p = h(key)\ mod\ 3$; equal keys colocate.**
Full-size DOI: 10.7717/peerj-cs.3614/fig-3

$$C_{\mathrm{BNLJ}} = (B + R_{\max} \cdot S) + w \cdot S \cdot (P - 1). \tag{4}$$

where P is the number of partitions, B is the size of the bigger input, S is the size of the smaller table, $R_{\max}$ is the row count of the bigger table, and w weights communication relative to computation. It is suitable for small build sides and non-equi predicates; worst-case compute and memory growth with $S \cdot R_{\max}$.

Cartesian Product Join (CPJ). Without a join condition, every row of one input pairs with every row of the other; engines may broadcast the small side or shuffle both. The cost model for CPJ is given in Eq. (5).

$$C_{\mathrm{CPJ}} = (B + \tfrac{R_{\max} \cdot S}{P}) + w \cdot \tfrac{(L+R)(P-1)}{P}. \tag{5}$$

where P is the number of partitions, B is the size of the bigger table, S is the size of the smaller table, $R_{\max}$ is the row count of the bigger table, L and R denote size of the left and right tables, representing local scan/probe costs, and w weights communication relative to computation. It is reasonable only for small inputs or explicit cross joins; the output grows with the product of input cardinalities.

RESEARCH METHODOLOGY

This section outlines the experimental methodology used to evaluate DPJoin, a cost-based distributed join optimization model. DPJoin extends the traditional cost model by explicitly incorporating coordination cost, making computation, communication, and coordination central elements in distributed query optimization. The objective is to improve query performance by accounting for how replication and parallelism influence execution time through their effects on data locality and resource utilization.

Unlike traditional optimizers that overlook the impact of replication and parallelism, DPJoin analyzes data locality based on replication levels and selects an appropriate degree of parallelism during planning. This coordination-aware approach enables more balanced

resource utilization and efficient execution, which is particularly important in shared-nothing architectures where task distribution and inter-node data movement strongly affect performance.

The methodology is presented in three main parts. First, we describe the Environmental Setup, including the hardware configuration, software stack, and benchmark workload. This ensures reproducibility and provides the necessary context for interpreting results. Unless otherwise stated, all Spark configuration parameters were kept at their default values; the only modified parameter was `spark.sql.shuffle.partitions`, which was dynamically adapted by DPJoin. Second, we conduct a Preliminary Analysis to examine how replication factor and parallelism level affect query performance using the TPC-DS benchmark (*Transaction Processing Performance Council, 2021*) at scale factor 1. Metrics such as local *vs* remote read ratios, partition counts, and execution time trends are collected to guide the design of DPJoin's cost model. Each experiment was repeated three times consecutively, and the average execution time was reported to mitigate runtime variability. Third, we introduce the Baseline Optimizer, RelJoin (*Liang et al., 2024*), which serves as a comparison point. While RelJoin estimates join costs based on cardinalities and data placement, DPJoin addresses its limitations by explicitly incorporating replication- and coordination-aware factors into plan selection.

Finally, we present the DPJoin Optimization Workflow, detailing the mathematical formulation of the cost model and how it dynamically selects both the physical join strategy and the level of parallelism. The workflow also incorporates adaptive thresholds to prevent over-saturation or underutilization. Together, these methodological components enable a fair and comprehensive evaluation of DPJoin against RelJoin and Spark's Adaptive Query Execution.

Environmental setup

The experiments were conducted on a dedicated cluster consisting of one master node and four worker nodes. Each node was equipped with 16 CPU cores and 32 GB of main memory, providing sufficient resources to simulate realistic distributed query execution scenarios.

The software environment used Apache Spark 3.3.2 (*Apache Software Foundation, 2023*), compiled from source to integrate DPJoin into the Catalyst optimizer. The cluster ran on Hadoop 3.3.2 (*Apache Software Foundation, 2022a*) with Hadoop Distributed File System (HDFS) as the underlying shared-nothing storage layer.

We varied the replication factor across four levels (1–4) in HDFS, which are the only meaningful settings because the NameNode (master) does not allow a DataNode (worker node) to store more than one replica of the same block and placing duplicates on the same node violates HDFS block placement policies; thus, the effective maximum replication factor cannot exceed the number of worker nodes (*Apache Software Foundation, 2022b*).

The experimental study executes all TPC-DS queries end-to-end without excluding any operators; their full operator pipelines run as specified. However, our performance analysis focuses on join-related behavior, and we do not isolate or report the individual costs of

non-join operators. This clarification ensures that the scope of our evaluation is consistent with the characteristics of TPC-DS and the objectives of our join-optimization study.

Preliminary analysis

Before implementing the full cost-based optimization model of DPJoin, we conducted a preliminary analysis to explore how replication level and degree of parallelism, affect query performance in a distributed environment. The aim of this analysis was twofold: first, to better understand how data locality and task coordination influence execution time, and second, to clarify how to integrate replication level and parallelism into cost model.

Effect of replication factor on query performance: Replication in distributed systems is traditionally used to improve availability and fault tolerance. However, it also has a substantial effect on query performance by influencing data locality, especially in distributed execution engines like Apache Spark. In Spark, the runtime metrics that quantify read locality are summarized in Table 1.

To investigate this, we conducted a series of experiments across 99 scan-and join-intensive queries on a 5-node Spark cluster (1 master, 4 workers), using scale factor 1 data (approximately 0.4 GB). We systematically varied the replication factor from 1 to 4. As illustrated in Fig. 4, increasing the replication factor led to a clear rise in LocalRead (MB) and a corresponding decrease in RemoteRead (MB). This behavior reflects the scheduler's increased ability to place tasks on nodes where data is already replicated, thereby enhancing I/O locality and reducing costly inter-node data transfers.

To statistically validate the strength of this relationship, we conducted a Pearson correlation analysis between the replication factor and four key execution metrics: Local Processing, Remote Processing, LocalRead, and RemoteRead. The results, summarized in Fig. 5, reveal strong correlations: Local Processing ($\rho = 0.81$) and LocalRead ($\rho = 0.77$) both increase with replication, while Remote Processing ($\rho = -0.81$) and RemoteRead ($\rho = -0.75$) show negative correlations. These results reinforce the interpretation that increased replication enables better scheduling decisions, improving data locality and reducing network bottlenecks.

Collectively, these findings demonstrate that replication not only enhances availability but also serves as a powerful lever for query performance optimization. While excessive replication may introduce write and coordination overhead, its ability to improve read-locality makes it a valuable parameter to incorporate into cost-based join optimization models. Our empirical study thus highlights a critical trade-off that is often overlooked in current distributed query planners, particularly those like RelJoin (*Liang et al., 2024*) or ElasticJoin (*Zhang, Zhang & Meng, 2025*), which do not quantify replication-level effects in their cost estimations.

From these observations, we derive our first two preliminary findings:

Preliminary Finding 1 (PF1): Increasing the replication factor improves data locality and reduces the volume of inter-node transfers, suggesting that replication should be explicitly modeled to reduce network cost.

Table 1 Runtime metrics used to quantify read locality and processing behavior in Spark.

Metric	Definition
LocalRead	Total data read from local storage (MB)
RemoteRead	Total data read from remote nodes over the network (MB)
LocalProcessing	Portion of tasks executed on nodes holding the required data locally
RemoteProcessing	Portion of tasks executed on nodes without local data access

Preliminary Finding 2 (PF2): The statistically strong correlation between replication and execution metrics supports the integration of replication-awareness into analytical cost models. These findings provide a quantitative basis for developing cost estimators that are sensitive to the effects of data replication, especially in dynamic environments.

Effect of parallelism on average execution time: Parallelism is a key determinant of performance in distributed query engines. By increasing the number of concurrent tasks, systems like Apache Spark aim to reduce end-to-end query latency by exploiting intra-query parallel execution. However, the relationship between degree of parallelism (DOP) and execution time is not strictly linear, due to overheads associated with task scheduling, coordination, and data shuffling.

To empirically analyze this relationship, we executed 99 TPC-DS benchmark queries on a five-node cluster using scale factor 1 data (approximately 0.4 GB in total). For each configuration, we measured the average execution time across all queries. As shown in Fig. 6, increasing the DOP initially results in substantial reductions in query latency, demonstrating effective utilization of available computational resources. However, beyond a certain point, further increases in parallelism yield diminishing returns—and in some cases, slight regressions in performance. This observation aligns with prior findings that excessive parallelism should be curtailed in favor of locality and fairness, as demonstrated by the delay scheduling algorithm (*Zaharia et al., 2010*).

Excessive task parallelism can lead to overheads from increased scheduling effort, resource contention, and inefficient utilization of I/O and memory bandwidth. Therefore, while parallelism is a powerful lever for improving performance, it must be carefully tuned to match the system's physical constraints (*e.g.*, CPU cores, memory, disk throughput) and workload characteristics. This finding further motivates the need for dynamic and cost-based parallelism control, which we later address through the DPJoin optimizer's runtime elasticity component.

We summarize the remaining two insights as follows:

Preliminary Finding 3 (PF3): Increasing the degree of parallelism does not always yield performance gains due to growing coordination and scheduling overheads. This observation is consistent with earlier analytical models of Spark partitioning, as demonstrated in *Gounaris et al. (2017)*, where the authors formalized the existence of saturation points by analyzing how increasing the number of shuffle partitions initially improves parallelism but eventually leads to diminishing returns due to coordination and scheduling overhead.

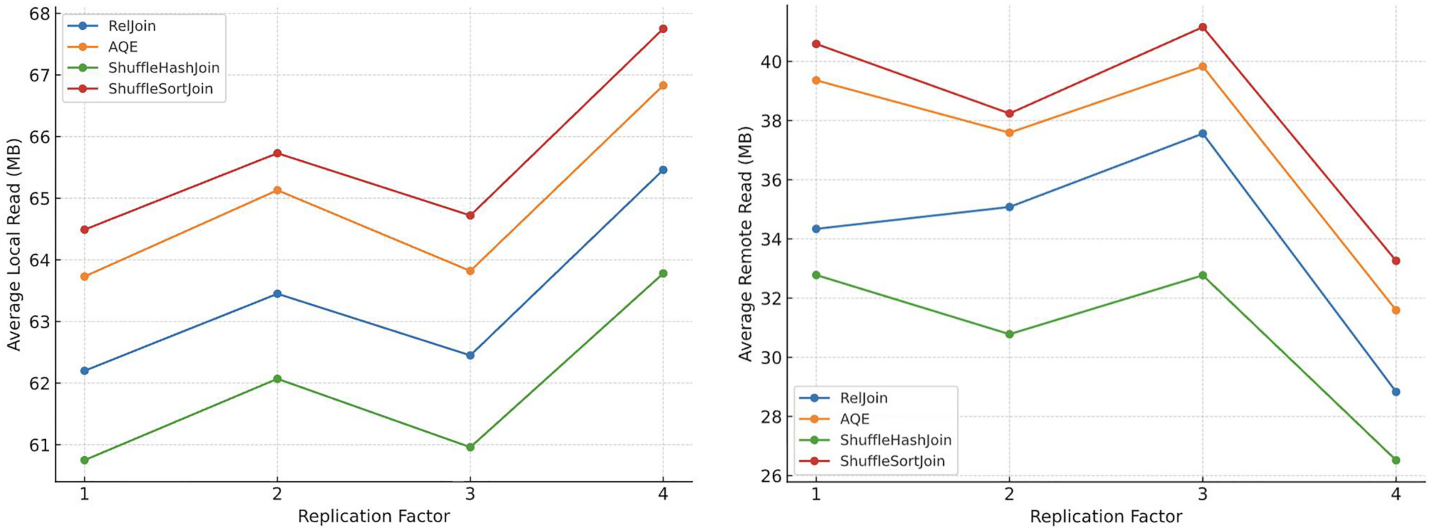

Figure 4 **Effect of replication factor on local and remote read volumes.** Full-size DOI: 10.7717/peerj-cs.3614/fig-4

Preliminary Finding 4 (PF4): A significant increase in task fragmentation and coordination messages was observed at higher parallelism levels, highlighting the need to incorporate coordination cost as an explicit factor in cost modeling. We see that this forms the foundation for designing parallelism-aware optimizers that dynamically adjust execution plans based on system capacity and workload intensity.

DPJoin algorithm

DPJoin is introduced as a planning-time optimizer that balances *computation*, *communication*, and *coordination* costs when selecting distributed join strategies. Unlike static approaches that fix the degree of parallelism or treat replication externally, DPJoin explicitly explores a grid of candidate parallelism levels for each feasible join method and detects the saturation point where additional parallelism no longer provides meaningful benefit. At this saturation point, the algorithm records the total cost of the method. Finally, it compares all methods at their own optimized saturation levels and selects the join strategy m^* together with its parallelism p^* that minimize the overall execution cost. This design ensures that DPJoin accounts for the trade-offs introduced by parallelism and replication, avoids over-parallelization, and produces robust plan choices tailored to distributed execution environments.

Algorithm inputs: cost models. For each join method $m \in \mathscr{M}$ and parallelism level p, the total cost is the sum of three components: computation, communication, and coordination as represented in Eq. (6).

$$C_{\text{total}}(p) = C_{\text{comp}}(p) + C_{\text{net}}(p) + C_{\text{coord}}(p). \tag{6}$$

These three components are instantiated differently depending on the join strategy, as described below.

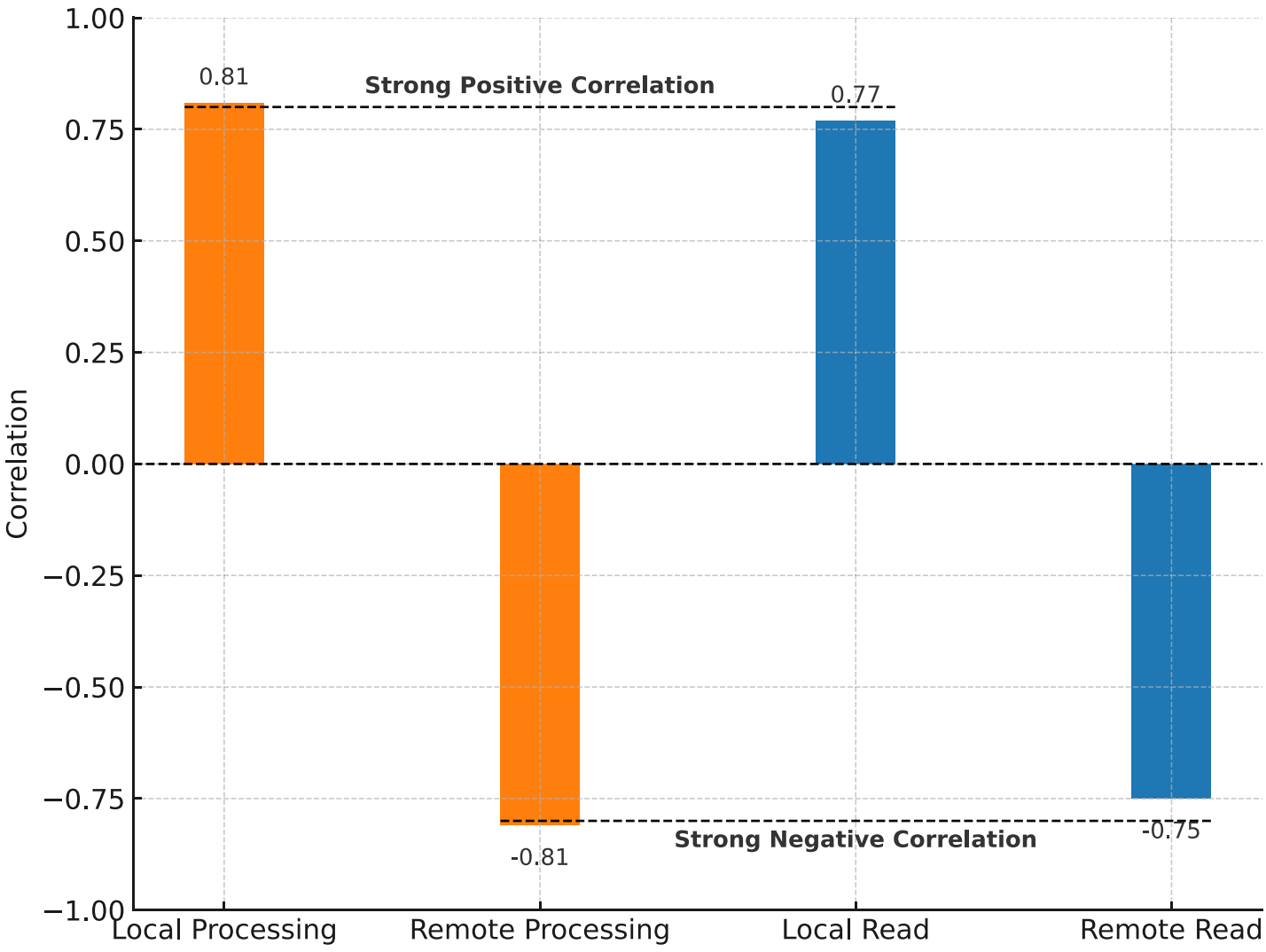

Figure 5 **Correlation between replication factor and execution metrics.** Full-size DOI: 10.7717/peerj-cs.3614/fig-5

Broadcast Hash Join (BHJ)

$$C_{\text{comp}}(p) = \frac{S}{p}, \qquad C_{\text{net}}(p) = S \cdot (N-1) \cdot (1 - \tfrac{r}{N}), \qquad C_{\text{coord}}(p) = O \cdot c. \tag{7}$$

The computation term(C_{comp}) divides the build+probe work by p to capture parallel speedup. The network term(C_{net}) reflects broadcasting the small side S to the other $N-1$ nodes, with replication reducing the fraction of remote transfers. The coordination term (C_{coord}) grows linearly with the number of operators O, since each operator incurs an average operator cost c.

Shuffle Hash Join (SHJ)

$$C_{\text{comp}}(p) = \frac{S+L}{p}, \qquad C_{\text{net}}(p) = \frac{S+L}{N} \cdot (N-1) \cdot (1 - \tfrac{r}{N}), \qquad C_{\text{coord}}(p) = O \cdot c. \tag{8}$$

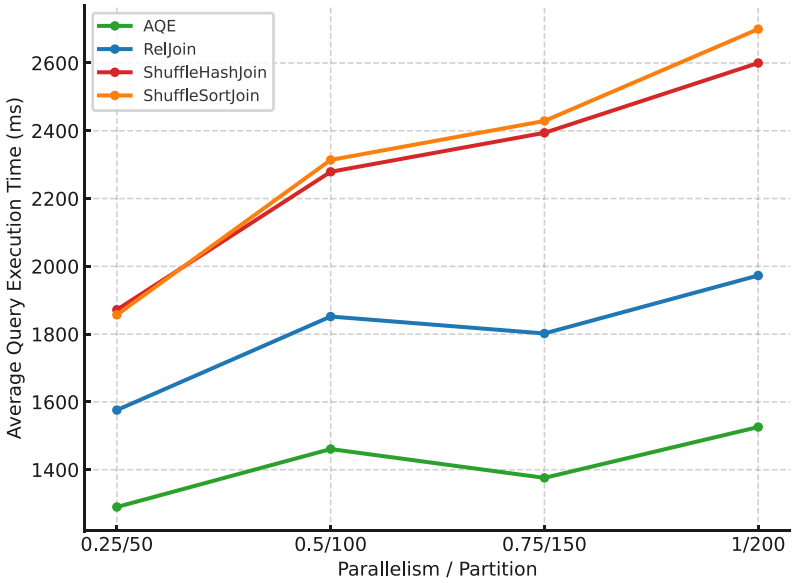

Figure 6 **Effect of parallelism on query performance.** Full-size DOI: 10.7717/peerj-cs.3614/fig-6

The computation cost scales with the total input size because both sides are partitioned and probed. The network term models the expected $(N-1)/N$ fraction of each partition that must be shuffled to remote nodes, with replication reducing this amount. The coordination term(C_{coord}) grows proportionally with O, reflecting scheduling and driver–executor overheads.

Sort–Merge Join (SMJ)

$$C_{\text{comp}}(p) = L \cdot \log_2(\tfrac{LR}{p}) + R \cdot \log_2(\tfrac{RR}{p}), \qquad C_{\text{net}}(p) = (S+L) \cdot (1 - \tfrac{r}{N}), \qquad C_{\text{coord}}(p) = O \cdot c. \quad (9)$$

The computation term accounts for parallel sorting, where each partition processes $\frac{LR}{p}$ and $\frac{RR}{p}$ rows respectively, making sorting logarithmic in partition size. The network term captures the shuffle of both sides to colocate join keys, with replication reducing transfers. The coordination term again scales with the operator count.

Broadcast Nested Loop Join (BNLJ)

$$C_{\text{comp}}(p) = B + R_B \cdot S, \qquad C_{\text{net}}(p) = S \cdot (N-1) \cdot (1 - \tfrac{r}{N}), \qquad C_{\text{coord}}(p) = O \cdot c. \quad (10)$$

The computation cost $C_{\text{comp}}(p)$ reflects the sum of the larger table size B and the product of its row count R_B with the smaller table size S. The communication cost $C_{\text{net}}(p)$ models broadcasting the small side S to the other $N-1$ nodes, adjusted by the replication factor r over four nodes. The coordination cost $C_{\text{coord}}(p)$ is proportional to the operator count O, with c denoting the per-operator overhead (*e.g.*, 1.45 KB).

Cartesian Product Join (CPJ)

$$C_{\text{comp}}(p) = N \cdot M, \qquad C_{\text{net}}(p) = (L+R) \cdot (N-1) \cdot (1 - \tfrac{r}{N}), \qquad C_{\text{coord}}(p) = O \cdot c. \quad (11)$$

The computation cost grows multiplicatively with the effective input size M across N nodes. The network term(C_{net}) represents colocating both sides across nodes, with replication reducing the remote fraction. The coordination term(C_{coord}) reflects per-operator overhead, scaling linearly with O and c.

Algorithm output: selected strategy and parallelism. At the end of the optimization procedure, DPJoin produces two outputs: the best join method m^* and its associated parallelism p^* as illustrated in Eq. (12).

$$m^* = \arg\min_{m \in \mathscr{M}} C_m^*, \qquad p^* = p_{m^*}^* \tag{12}$$

Here C_m^* is the total cost of method m measured at its saturation point p_m^*. The algorithm first scans all candidate parallelism levels $p \in P$ for each method m, identifies the saturation point where further increases yield negligible improvements, and records the corresponding total cost C_m^*. Among all methods, the one with the smallest C_m^* is chosen as m^*, and the parallelism level fixed at p^*.

Algorithm description. The algorithm takes as input a set of feasible join methods and a monotone grid of parallelism levels. Its task is to determine, for each method, the parallelism level at which adding more tasks no longer yields meaningful improvements, and then to select the globally best method at saturation points.

The first step is to compute the adaptive convergence threshold (ε). We set $\varepsilon = 1/O$, where O is the total number of operators in the logical plan. This ensures that complex queries require stronger evidence of improvement before allowing additional parallelism (see line 3 in Algorithm 1).

$$\varepsilon = \frac{1}{O}. \tag{13}$$

Here, O denotes the total number of operators in the plan; it is computed by recursively summing all operators (scan, filter, project, join, *etc.*) contained in the subplan that includes the join.

The algorithm then begins its outer loop, iterating once for each candidate join strategy $m \in \mathscr{M}$ (line 4). For every strategy, it scans the parallelism grid $P = \{p_1, \ldots, p_K\}$ in increasing order (line 7). At each step, the three cost components are computation, communication, and coordination, and aggregated into the total cost, together with a scale-normalized mean that is easier to compare across different scales (lines 9–10).

$$C_{\text{total}}(p) = C_{\text{comp}}(p) + C_{\text{net}}(p) + C_{\text{coord}}(p), \qquad C_{\text{mean}}(p) = \frac{C_{\text{total}}(p)}{3}. \tag{14}$$

Here, the three cost components are computed using the cost formulas defined for each join strategy: $C_{\text{comp}}(p)$ models per-partition computation based on input sizes and operator complexity; $C_{\text{net}}(p)$ captures shuffle or broadcast communication as a function of data volume, replication level, and node count; and $C_{\text{coord}}(p)$ reflects the coordination overhead, computed from the total operator count and a fixed per-operator message cost. These strategy-specific cost definitions are provided in the corresponding cost model descriptions.

Algorithm 1 **DPJoin: saturation-based selection.**

1: **Input:** Methods $\mathscr{M}$, parallelism grid $P = \{p_1 < \ldots < p_K\}$, operator count $O \geq 1$
2: **Output:** Best method m^* and parallelism p^*
3: $\varepsilon \leftarrow 1/O$ ▷ adaptive convergence threshold
4: **for** $m \in \mathscr{M}$ **do**
5: $p_m^* \leftarrow$ undefined; $C_m^* \leftarrow +\infty$
6: $bestP \leftarrow p_1$; $bestMean \leftarrow +\infty$; $lastMean \leftarrow +\infty$
7: **for** $p \in P$ **do** ▷ scan parallelism in increasing order
8: $(C_{\text{comp}}, C_{\text{net}}, C_{\text{coord}}) \leftarrow calculate\_cost(m, p)$
9: $C_{\text{total}} \leftarrow C_{\text{comp}} + C_{\text{net}} + C_{\text{coord}}$
10: $C_{\text{mean}} \leftarrow C_{\text{total}}/3$
11: **if** $|C_{\text{mean}} - lastMean| < \varepsilon$ **then** ▷ saturation guard
12: $(p_m^*, C_m^*) \leftarrow (p, C_{\text{total}})$; **break**
13: **end if**
14: **if** $C_{\text{mean}} < bestMean$ **then** ▷ robustness/fallback guard
15: $(bestP, bestMean) \leftarrow (p, C_{\text{mean}})$; $C_m^* \leftarrow C_{\text{total}}$
16: **end if**
17: $lastMean \leftarrow C_{\text{mean}}$
18: **end for**
19: **if** p_m^* is undefined **then**
20: $(p_m^*, C_m^*) \leftarrow (bestP, C_m^*)$
21: **end if**
22: **end for**
23: **return** $m^* \leftarrow \arg\min_{m \in \mathscr{M}} C_m^*$, $p^* \leftarrow p_{m^*}^*$

The key stopping rule is the *saturation guard*. Whenever the improvement between consecutive parallelism levels becomes negligible when the mean cost difference falls below the adaptive threshold, the algorithm halts the scan for that method and records its current parallelism and cost (lines 11–12).

$$|C_{\text{mean}}(p) - C_{\text{mean}}(p^-)| < \varepsilon \quad \Rightarrow \quad (p_m^*, C_m^*) \text{ fixed at } p. \tag{15}$$

Here p^- denotes the previous grid point. This condition prevents unnecessary over-parallelization once the efficiency gains have saturated.

At the same time, a *robustness guard* tracks the best mean value encountered so far (lines 13–14). If the saturation guard never triggers, for instance when the cost curve decreases very smoothly, the algorithm falls back to the parallelism level that minimized the mean cost (line 16).

$$p_m^* = \arg\min_{p \in P} C_{\text{mean}}(p), \qquad C_m^* \text{ taken at } p_m^*. \tag{16}$$

Once all strategies have been processed independently, the algorithm proceeds to global selection. It compares the total costs recorded at each method's saturation point and returns the method–parallelism pair with the overall lowest cost (line 18):

$$m^* = \arg\min_{m \in \mathscr{M}} C_m^*, \qquad p^* = p_{m^*}^*. \tag{17}$$

In effect, lines 3–18 describe a two-stage process that governs the optimization logic of DPJoin. First, each candidate method is optimized locally. In this stage, the algorithm systematically scans the available parallelism grid, starting from the lowest level of parallelism and gradually increasing the degree until a saturation point is reached or a

predefined fallback condition triggers termination. The idea is to explore the efficiency frontier of each method individually, so that DPJoin can observe how costs evolve with higher degrees of parallelism. This local search is not arbitrary: it explicitly monitors the interaction between computation cost, communication overhead, and coordination complexity. Rather than relying on static assumptions or default partition counts, the algorithm evaluates how performance behaves in practice as the parallelism level changes.

Second, once local optimization has been completed for all candidate strategies, DPJoin performs a global comparison across the outcomes. Here, the optimizer identifies the best method among the locally optimized candidates, ensuring that the final plan reflects not only the intrinsic behavior of each strategy but also the relative advantages across the entire set. This global decision step is critical: it prevents the optimizer from being biased towards a method that looks efficient under narrow conditions but is suboptimal when compared more broadly. For instance, a shuffle-based join might appear strong when considered in isolation, but when its high coordination cost is contrasted with the communication efficiency of a broadcast join, the balance may shift. By incorporating this second decision layer, DPJoin integrates both depth (detailed within-strategy exploration) and breadth (comparative across-strategy evaluation) into its optimization logic.

Together, these two stages, local exploration followed by global selection, give DPJoin the ability to handle the multidimensional trade-offs of distributed query processing. Unlike traditional optimizers that rely heavily on static cardinality estimates or fixed cost coefficients, DPJoin adapts its choices to the actual structure and dynamics of the workload. It balances computation, communication, and coordination factors in a principled way. At the same time, it avoids the common pitfall of over-parallelization, which can waste resources, create scheduling bottlenecks, and increase latency. By explicitly controlling parallelism levels, DPJoin ensures that additional tasks are created only when they improve throughput, not when they merely inflate scheduling overhead. This makes the approach resilient against inefficiencies that often arise in large clusters. Issues such as task granularity, shuffle imbalances, or stragglers can quickly offset any theoretical gains from fine-grained partitioning, but DPJoin is able to mitigate these risks.

Moreover, the design acknowledges heterogeneity: in real distributed systems, resources are often uneven, workloads fluctuate, and runtime conditions deviate from initial estimates. By accounting for such variability, DPJoin produces plan choices that are not only theoretically optimal but also practically robust. This robustness is particularly important for modern engines such as Spark or Presto, where adaptive query execution is essential to sustain performance under diverse workloads. The two-stage process also provides a natural foundation for future extensions. For example, additional cost factors such as memory pressure, I/O contention, or skew-handling mechanisms can be incorporated into the local optimization phase without changing the overall structure.

PERFORMANCE EVALUATION

This section empirically evaluates DPJoin after its integration into Apache Spark. Our goal is to quantify the benefits and trade-offs of replication-and parallelism-aware planning compared to baseline strategies (AQE, RelJoin, ShuffleHashJoin, SortMergeJoin). We use

the full TPC-DS workload (99 analytical queries) on a fixed 5 node cluster and vary both data scale (SF = 1, 10, 50) and replication factor (RF = 1, 2, 3, 4). For each configuration, queries are executed three times; we analyze average runtimes, winner counts (for each query, the strategy with the lowest average runtime is counted as the winner), and the degree of parallelism chosen by each planner (task counts). We also examine sensitivity to query plan structure *via* a composite complexity score.

The section is organized as follows. Integration of DPJoin into Apache Spark summarizes how DPJoin is integrated into Spark Catalyst and how it enables replication- and coordination-aware decisions. Experimental Design details the experimental setup, workloads, metrics, and evaluation axes. Experimental Results reports the results across replication levels and data scales, analyzes the impact of query complexity, and contrasts DPJoin with baseline strategies using both winner counts and average execution times.

Integration of DPJoin into apache spark

To operationalize the DPJoin strategy and its underlying cost model, we integrated it into the physical planning stage of Apache Spark's Catalyst optimizer. Specifically, we developed a new custom Strategy class named DPJoin, which extends Spark's native join planning logic by introducing adaptive, cost-aware decision making at runtime.

The DPJoin strategy is invoked during the physical planning phase and intercepts logical Join nodes. When such a node is detected, the system first checks for equi-join keys and evaluates whether existing strategies (*e.g.*, BroadcastHashJoin, ShuffleHashJoin, SortMergeJoin, Broadcast Nested Loop Join, CartesianProduct) are applicable. For each feasible strategy, the integrated CostModel is queried to compute execution costs using the four proposed findings (PF1, PF2, PF3, PF4), which stem from our preliminary analysis. These findings capture different aspects of query complexity namely input size, data movement, compute, coordination, and saturation overhead and serve as guiding signals for the optimizer.

Based on the computed costs, the planner identifies the strategy with the minimum estimated execution time. The selected strategy is then instantiated *via* Spark's native SparkPlan constructs (*e.g.*, BroadcastHashJoinExec, SortMergeJoinExec, *etc.*), and the optimal number of shuffle partitions is assigned dynamically by modifying the *spark.sql.shuffle.partitions* configuration entry within the same session (*Ninan, 2023*).

To enable fine-grained elasticity control, the implementation further includes a saturation estimation mechanism based on the number of join operators in the logical plan. A query complexity factor is computed based on the number of join operators (JoinCount) in the plan. This value is used to adaptively compute the convergence threshold $\varepsilon = \frac{1}{O}$, enabling more fine-grained cost comparisons for complex queries. This value controls the granularity of performance estimation and acts as a sensitivity factor during the cost comparisons. More complex queries (*i.e.*, those with more joins) trigger finer-grained comparisons, helping avoid suboptimal decisions in heavily parallel plans.

Additionally, the planner logs detailed runtime metadata in JSON format within Spark's SparkConf object. These logs contain per-query metrics, such as data sizes, row counts,

selected strategies, estimated costs, and partition counts. This feature facilitates traceable experimentation and debugging across multiple runs and queries.

Overall, the integration of DPJoin into Catalyst is modular and non-invasive. It maintains full compatibility with Spark's optimizer API while offering advanced, dynamic decision-making capabilities that account for both system-level constraints and query-specific complexity.

Experimental design

To systematically evaluate the effectiveness and robustness of the proposed DPJoin strategy, we designed three complementary experiments that target different aspects of distributed join optimization. All experiments were conducted on a 5-node Spark cluster under varying data scales and replication levels. For each configuration, every query was executed three times consecutively to observe how effectively each strategy leveraged data locality, and the average execution time was used for comparison. To ensure comparable block distributions across scales, the HDFS block size was set to 16 MB for Scale 1, 64 MB for Scale 10, and 128 MB for Scale 50. The experiments are:

1. **Robustness of join strategies under varying replication levels.**
   This experiment analyzes how five different join strategies; Shuffle Hash Join (SHJ), Shuffle Sort Join (SSJ), Spark's Adaptive Query Execution (AQE), RelJoin, and DPJoin respond to decreasing replication factors. For scale factor 1, 10 and 50 datasets, we varied the replication level from 1 to 4 and executed all benchmark queries 3 consecutive times per setting. For each query, the strategy with the lowest average execution time was marked as the winner. We then counted the number of queries where each strategy outperformed the others, allowing us to evaluate the tolerance of each strategy to replication loss. This analysis helps quantify how robust each approach is when data locality deteriorates due to limited data replication.
2. **Resource efficiency: task counts and shuffle traffic**
   To assess resource efficiency independently of query complexity, we evaluate two orthogonal proxies: (i) *task-level parallelism* (the total number of tasks produced by a plan) and (ii) *shuffle traffic* (sum of shuffle read and write, in MB). We fix the replication factor to $RF = 4$ and compute these metrics per strategy over the 99 TPC-DS queries (averaging multiple runs when applicable). For task-level parallelism, we report per-query series to reveal fragmentation and coordination costs; for shuffle, we report the average volume across queries to capture network I/O pressure. Together, these measurements provide an apples-to-apples view of how each strategy uses resources—distinguishing cost-aware planning from shuffle-heavy execution.
3. **Join strategy execution time analysis across data scales**
   In this section, we analyze how the execution-time performance of five join strategies—Shuffle Hash Join, Shuffle Sort Join, Adaptive Query Execution (AQE), RelJoin, and DPJoin—changes as data replication decreases. The evaluation is based on TPC-DS benchmark results executed over datasets of scale factor 10 and 50, with

replication levels ranging from 4 to 1. Each query was run three times per setting to obtain stable average runtimes.

We analyzed the results using two key metrics: (1) Winner count: For each query, the strategy with the lowest average execution time across repetitions was marked as the winner. We then counted how many times each strategy outperformed the others. (2) Average execution time: We computed the mean runtime for each strategy across all queries and replication settings.

These two metrics jointly capture both the consistency and overall efficiency of each join strategy under changing replication levels. Winner Count reflects how often a strategy is the fastest across diverse queries, while Average Execution Time summarizes its general performance. Together, they provide a clear view of how replication reductions influence the relative behavior of the join algorithms.

Experimental results

Robustness of join strategies under varying replication levels. The first analysis investigates how different join strategies perform under decreasing replication levels. The replication factor (RF) was varied from 4 to 1, while the experiments were repeated across three data scales ($SF = 1, 10, 50$, where SF denotes the *scale factor*). The aim is to evaluate the robustness of each strategy in the face of limited data locality. In each configuration ($RF = 4, 3, 2, 1$), all TPC-DS benchmark queries were executed three times consecutively. The average execution time was computed per query, and the strategy with the lowest average was considered the winner for that query. This evaluation was conducted for five strategies: *ShuffleHashJoin*, *ShuffleSortJoin*, *Adaptive Query Execution (AQE)*, *RelJoin*, and the proposed *DPJoin*. The reported metric is the number of queries in which a given strategy outperformed all others. Additional comparisons across different scale factors allow us to observe how strategies react not only to reduced replication but also to growing dataset sizes. These observations provide early insights into which strategies are more resilient to both limited data locality and increased workload demands.

Small scale (Scale 1). In Fig. 7, DPJoin delivers the lowest overall execution times across all replication settings, with AQE following closely. RelJoin shows stable behavior and remains more consistent than the shuffle-based strategies. ShuffleHashJoin and ShuffleSortJoin exhibit higher variability when replication decreases, reflecting their sensitivity to data locality and shuffle overhead. By contrast, DPJoin, AQE, and RelJoin maintain relatively stable trends, benefiting from their lower shuffle intensity and more locality-aware planning.

Medium scale (Scale 10). In Fig. 8, DPJoin achieves the best overall performance, maintaining the lowest execution times across replication settings, though it still incurs some overhead as replication increases.

AQE follows closely, showing stable behavior, while RelJoin provides moderate but reliable performance. ShuffleHashJoin and ShuffleSortJoin degrade sharply, particularly at

higher replication levels. This is because additional replicas introduce extra coordination and shuffle traffic, which amplify network and memory overhead.

Large scale (Scale 50). At the largest scale in Fig. 9, AQE achieves the best results, consistently delivering the lowest execution times across all replication factors. DPJoin follows closely, maintaining lower execution times than RelJoin and shuffle-based methods, but it does not surpass AQE at this scale. RelJoin shows moderate performance, remaining more stable than shuffle-heavy strategies. ShuffleHashJoin and ShuffleSortJoin show sharp changes in execution time as replication levels decrease, indicating that they are more sensitive to limited data locality and coordination overhead.

The results across all three data scales reveal how replication levels influence different join strategies, although the magnitude and direction of the effect vary by scale. Shuffle-based strategies, ShuffleHashJoin and ShuffleSortJoin, tend to exhibit higher sensitivity under reduced replication, especially at larger scales, where lower data locality can amplify shuffle and coordination overhead. At smaller scales, the trend is less pronounced, and execution times may fluctuate rather than consistently increase as replication decreases. RelJoin, DPJoin, and AQE show greater resilience across replication settings, benefiting from either replication-aware costing or adaptive planning mechanisms that help mitigate data movement overhead. RelJoin offers moderate stability but lacks adaptability beyond a certain threshold, while DPJoin and AQE maintain robust performance patterns across scales. While DPJoin achieves strong results at smaller scales ($SF = 1, 10$), AQE delivers the best performance at the largest scale ($SF = 50$), reflecting its ability to incorporate runtime adaptivity and maintain efficiency as data size grows.

Resource efficiency: task counts and shuffle traffic.

Parallelism view. Figure 10 shows that **DPJoin** uses the *fewest tasks* across nearly all queries, *i.e.*, it chooses the lowest effective degree of parallelism.

At this small scale (SF = 1, RF = 4), limiting task count prevents using more tasks than necessary and reduces scheduler/coordination overhead, shuffle overhead, and memory pressure, which are the costs that dominate when data volumes are modest. AQE and RelJoin employ moderate parallelism, while ShuffleHashJoin and ShuffleSortJoin create many more tasks, which can inflate coordination and shuffle costs without proportional speedups. Overall, DPJoin's replication-aware costing steers it toward a conservative yet sufficient parallelism level, yielding a more resource-efficient execution plan.

Shuffle overhead by strategy.

Figure 11 shows a stark separation between shuffle-based and cost-aware approaches. Shuffle Sort and Shuffle Hash incur the largest data movement (∼1,548 and ∼1,544 MB on average), which is about *6–8* × higher than the others. Among the cost-aware methods, AQE yields the lowest shuffle volume (∼193 MB), followed by RelJoin (∼242 MB) and DPJoin (∼253 MB). Lower shuffle read+write correlates with fewer coordination stalls and more stable runtimes, whereas the shuffle-heavy strategies pay substantial network and scheduling overhead without commensurate gains.

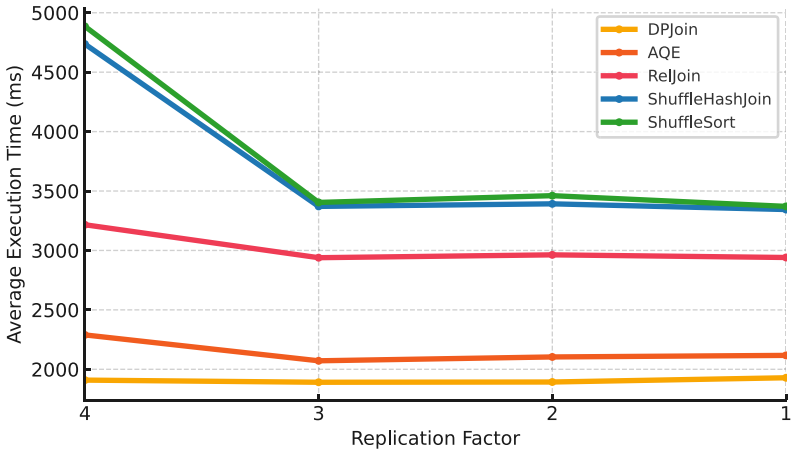

Figure 7 **Average execution time over 99 queries *vs* replication (Scale 1 & Fixed Parallelism 1).**
Full-size DOI: 10.7717/peerj-cs.3614/fig-7

Join strategy execution time analysis across data scales.

The Fig. 12 above shows execution time per query under Scale 10 with replication factor 4. Join strategies were evaluated under varying data scales and replication factors using two metrics: the number of fastest executions (*i.e.*, wins) and the average execution time. We first analyze the per-query distribution (Fig. 12), and then summarize the outcomes in Table 2.

The results clearly indicate that DPJoin outperforms all competing strategies across all replications and both scales. In Scale 1, DPJoin wins 80 out of 99 queries at replication factor 4. Similarly, in Scale 10, it maintains dominance with 81 wins under the same replication level. While AQEJoin shows moderate performance, it falls significantly behind DPJoin. Traditional strategies such as RelJoin, ShuffleHashJoin, and ShuffleSort consistently exhibit low win rates, often failing to adapt to the increasing replication and data scale.

Table 3 above shows that DPJoin wins more frequently and achieves the lowest average execution times.

Figure 13 above illustrates execution time per query for each join strategy at Scale 50 with replication factor 4. Compared to the earlier analysis at Scale 10, a general upward shift in execution times is clearly visible across all strategies, as expected due to the larger data volume. This increase confirms the sensitivity of join performance to input size and highlights the importance of scalable join strategies. As summarized in Table 4, AQE achieves the highest number of fastest executions across all replication levels, consistently outperforming other strategies at this scale. While DPJoin remains competitive and secures a substantial number of wins, its relative advantage diminishes as data size grows, indicating that AQE adapts more effectively under large-scale conditions.

At Scale 50, AQE achieves the highest number of fastest executions across all replication levels, clearly outperforming DPJoin. RelJoin shows moderate success, while ShuffleHashJoin and ShuffleSortJoin remain marginal with only a few wins.

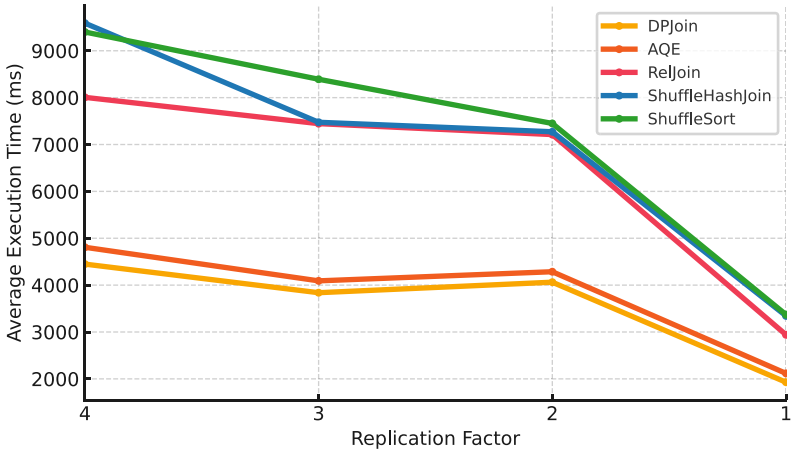

Figure 8 **Average execution time over 99 queries *vs* replication (Scale 10 & Fixed Parallelism 1).**
Full-size DOI: 10.7717/peerj-cs.3614/fig-8

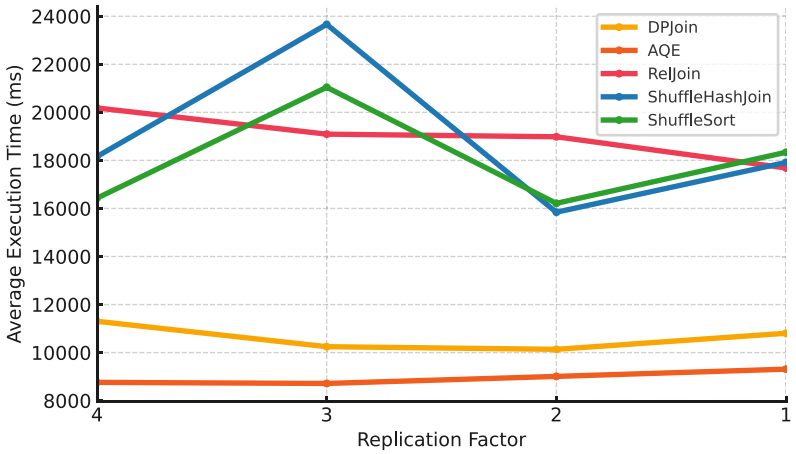

Figure 9 **Average execution time over 99 queries *vs* replication (Scale 50 & Fixed Parallelism 1).**
Full-size DOI: 10.7717/peerj-cs.3614/fig-9

When we examine the average execution times, a clear trend emerges, as shown in Table 5. AQE consistently achieves lower runtimes than DPJoin across all replication factors. For instance, at replication factor 3, AQE records an average of 8,715.0 ms, compared to DPJoin's 10,246.5 ms. This indicates that at large scale, AQEJoin not only wins more queries but also executes them more efficiently on average, whereas DPJoin's relative advantage diminishes as data size increases.

Table 6 reports the standard deviation of execution times across repeated runs for each strategy. DPJoin shows consistently lower variability at both scales, indicating more stable and predictable performance. In contrast, ShuffleHashJoin and ShuffleSortJoin exhibit substantially higher variance, especially at Scale 10, suggesting reduced stability and greater sensitivity to increasing data size.

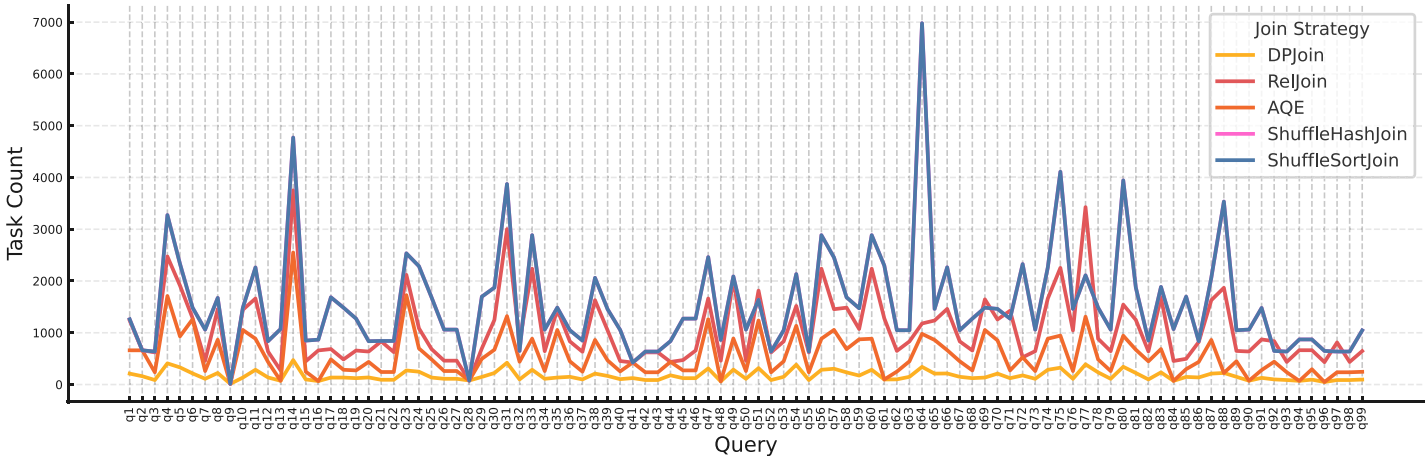

Figure 10 **Task count per query by join strategy (*SF* = *1*, *RF* = *4*).** Full-size DOI: 10.7717/peerj-cs.3614/fig-10

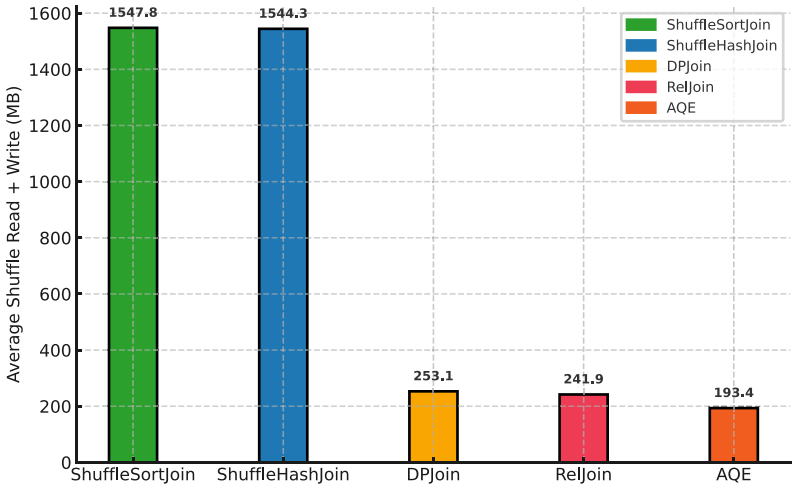

Figure 11 **Average shuffle read+write (MB) by strategy (Scale 10, Replication 4).** Full-size DOI: 10.7717/peerj-cs.3614/fig-11

Table 2 **Number of fastest executions by strategy in Scale 1 and Scale 10.**

Strategy	Scale 1				Scale 10			
	Rep.1	Rep.2	Rep.3	Rep.4	Rep.1	Rep.2	Rep.3	Rep.4
DPJoin	61	65	67	80	74	76	76	81
AQE	26	23	21	19	16	14	20	11
RelJoin	2	2	2	0	9	5	0	0
ShuffleHashJoin	4	5	4	0	0	2	2	0
ShuffleSortJoin	6	4	5	0	0	1	1	7

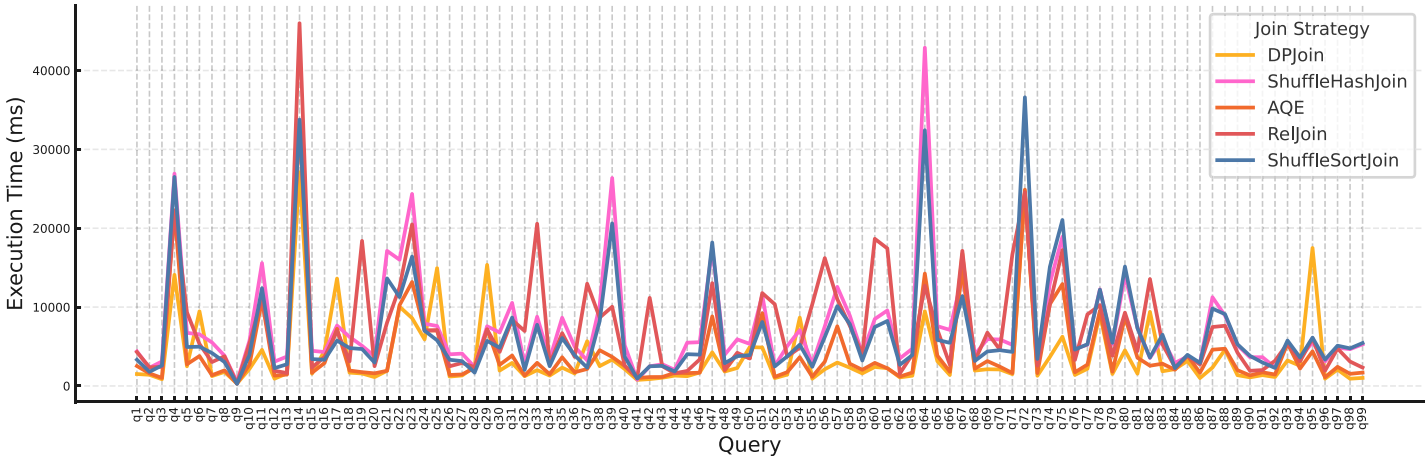

Figure 12 **Execution time per query for each join strategy (Scale 10, Replication 4).** Full-size DOI: 10.7717/peerj-cs.3614/fig-12

Table 3 **Average execution time (ms) by strategy in Scale 1 and Scale 10.**

Strategy	Scale 1				Scale 10			
	Rep.1	Rep.2	Rep.3	Rep.4	Rep.1	Rep.2	Rep.3	Rep.4
DPJoin	1,928.7	1,893.0	1,890.9	1,909.0	4,203.9	4,062.6	3,838.4	4,448.4
AQE	2,117.0	2,103.9	2,071.5	2,289.3	4,336.3	4,285.9	4,091.0	4,807.5
RelJoin	2,940.7	2,962.9	2,939.0	3,216.2	7,104.3	7,211.7	7,445.0	8,006.1
ShuffleHashJoin	3,344.2	3,392.2	3,370.6	4,736.3	7,313.4	7,272.1	7,474.6	9,588.9
ShuffleSortJoin	3,370.7	3,460.9	3,404.0	4,885.1	7,386.2	7,449.3	8,392.4	9,400.1

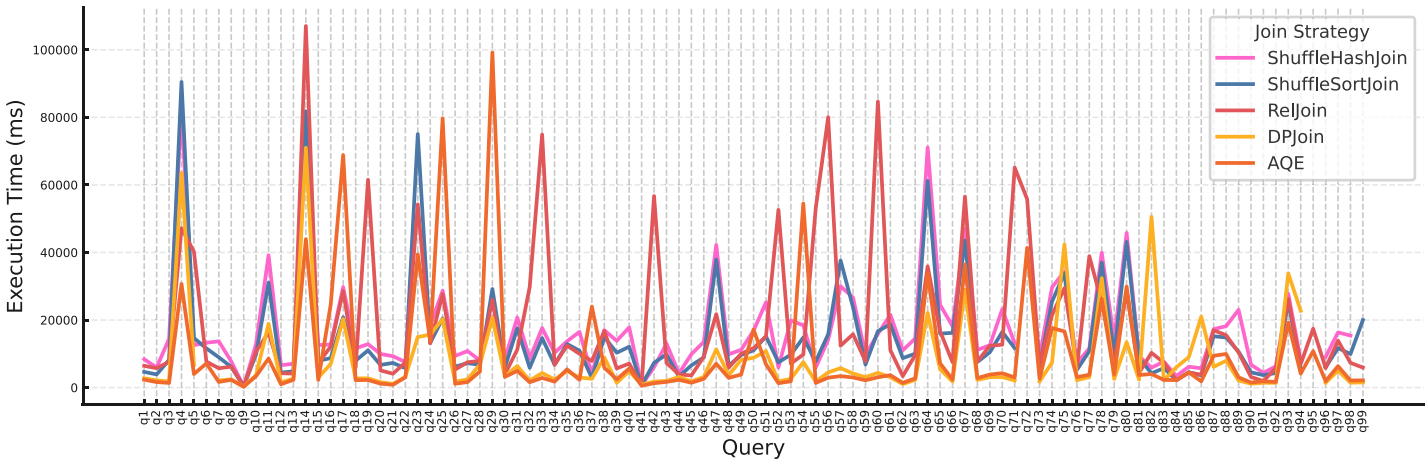

Figure 13 **Execution time per query for each join strategy at scale 50 with replication factor 4.** Full-size DOI: 10.7717/peerj-cs.3614/fig-13

As shown in Table 7, execution-time variability increases substantially at Scale 50, particularly for the ShuffleHashJoin and ShuffleSortJoin strategies. DPJoin maintains comparatively lower variability across all repetitions, supporting the stability trend previously observed in Table 6. The extremely high deviation values in Rep. 3 for both

Table 4 Number of fastest executions by strategy in Scale 50.

Strategy	Rep.1	Rep.2	Rep.3	Rep.4
AQE	50	48	41	45
DPJoin	38	40	37	31
RelJoin	10	4	17	20
ShuffleHashJoin	1	3	2	2
ShuffleSortJoin	0	4	2	1

shuffle-based strategies indicate significant performance instability under large-scale data processing.

In general, the variability patterns observed across the experiments show that different join strategies respond differently to increasing data scale. While some methods maintain relatively steady behavior, others exhibit higher dispersion as the workload grows. These results suggest that performance stability is influenced by both the underlying join mechanism and the scale of the input, highlighting the importance of considering variability—not only mean execution time—when evaluating distributed query processing approaches.

As shown in Fig. 14, DPJoin performs well at scale 1 and scale 10, where its relatively low level of parallelism does not place significant pressure on executor memory. At these smaller data sizes, the amount of intermediate data remains manageable and GC time stays low. The situation changes at scale 50. Because DPJoin continues to operate with the same low parallelism while the input volume increases substantially, more intermediate results accumulate in executor memory. This accumulation increases memory pressure and leads to a much higher garbage collection time. In other words, DPJoin becomes affected by large data sizes because it does not take system resources, particularly executor memory, into account when determining how much parallelism to use. A more resource aware design would increase parallelism when the data size grows and prevent the rise in memory usage that causes the GC overhead observed at scale 50.

DISCUSSION

The experimental results across different data scales and replication levels provide a clear picture of how modern join strategies react to varying data locality and workload sizes. Overall, DPJoin shows strong performance at small and medium scales (SF = 1 and 10). It consistently achieves the highest winner count and maintains competitive average runtimes. Its replication-aware and coordination-aware planning enables efficient use of parallelism, especially when data volumes are modest.

At larger scale (SF = 50), the trend changes: AQE delivers the best performance in terms of both average execution time and winner count. This behavior is expected because AQE incorporates adaptive runtime feedback that becomes increasingly valuable as dataset sizes grow and plan misestimations become more costly. DPJoin selects lower degrees of parallelism for certain queries, which is beneficial at small and medium scales, but results in more data being processed per executor when the dataset becomes large. This increases

Table 5 Average execution time (ms) by strategy in Scale 50.

Strategy	Rep.1	Rep.2	Rep.3	Rep.4
AQE	9,310.4	9,008.9	8,715.0	8,760.7
DPJoin	10,804.3	10,136.2	10,246.5	11,302.8
RelJoin	17,684.9	18,983.6	19,092.7	20,183.7
ShuffleHashJoin	17,916.4	15,843.3	23,662.4	18,162.7
ShuffleSortJoin	18,338.3	16,212.5	21,044.4	16,429.1

memory pressure and leads to higher garbage-collection activity, which contributes to the performance gap observed at SF = 50. This behavior indicates that DPJoin currently lacks resource awareness and does not consider available executor memory when deciding parallelism levels. This behavior indicates that DPJoin currently lacks resource awareness and does not consider available executor memory when deciding parallelism levels. DPJoin should include a mechanism that jointly evaluates the data size and the available cluster resources when determining the parallelism level. For example, if the input data is large and the cluster resources are relatively limited, DPJoin should assign more parallelism to distribute the workload and reduce memory pressure on each task. Conversely, if the data size is small relative to the available resources, DPJoin can use lower parallelism. Establishing this direct relationship between data volume and cluster capacity would enable DPJoin to make more accurate and resource-aware parallelism decisions.

A notable point is the behavior of RelJoin. In the original RelJoin study, the strategy often outperformed AQE. In our experiments, however, RelJoin performs moderately and does not surpass AQE or DPJoin. This difference can be attributed to several key factors:

Cluster size and resource differences: The RelJoin article evaluated the method on a larger cluster and at higher data scales (up to SF = 100). Larger clusters reduce contention during shuffle, provide more memory per executor, and allow RelJoin's broadcast-based decisions to be more effective. Our environment uses a smaller shared-nothing cluster with four worker nodes. Under these constrained resources, shuffle pressure and memory limits have a stronger effect, causing RelJoin's cost decisions to be less advantageous.

Differences in dataset volume and replication behavior: At larger scales, RelJoin benefits significantly from broadcasting relatively small tables. In our smaller dataset scenarios, broadcast opportunities are fewer and the relative cost differences between strategies shrink. As a result, RelJoin's advantage becomes less visible.

Configuration constraints: The RelJoin article evaluates multiple configurations, including varying network-weight parameters. In our setup, we use a fixed configuration for fairness across strategies, which limits RelJoin's tunability and adaptability.

These differences explain why RelJoin appears weaker in our environment, and we clarified this in the manuscript.

Although TPC-DS queries include operators such as Aggregation and Sort, our evaluation mainly focuses on join-related behavior. We did not isolate the individual cost contributions of these additional operators. Their interactions may influence overall execution time; therefore, future work should examine operator-level effects explicitly.

Table 6 Standard deviation (ms) by strategy in Scale 1 and Scale 10.

Strategy	Scale 1				Scale 10			
	Rep.1	Rep.2	Rep.3	Rep.4	Rep.1	Rep.2	Rep.3	Rep.4
DPJoin	2,835.4	2,812.8	2,834.5	2,557.7	2,835.4	5,278.5	4,699.2	5,467.9
AQE	3,884.1	3,747.7	3,651.0	3,620.7	3,884.1	5,099.7	5,016.5	5,651.4
RelJoin	4,832.0	4,732.9	4,738.7	4,342.5	4,832.0	6,891.3	6,929.8	7,283.2
ShuffleHashJoin	3,529.1	3,518.3	3,592.6	4,496.0	3,529.1	6,428.4	7,052.7	23,304.4
ShuffleSortJoin	3,654.5	3,614.5	3,728.7	4,570.3	3,654.5	6,705.2	7,612.4	23,747.6

Table 7 Standard deviation (ms) by strategy in Scale 50.

Strategy	Rep.1	Rep.2	Rep.3	Rep.4
DPJoin	19,198.6	17,591.9	18,566.8	18,993.6
AQE	12,013.0	12,130.6	11,522.1	11,558.3
RelJoin	22,006.9	23,353.3	24,303.2	22,768.5
ShuffleHashJoin	17,441.2	16,562.6	82,490.4	16,053.5
ShuffleSortJoin	17,985.1	16,325.3	82,046.5	18,813.9

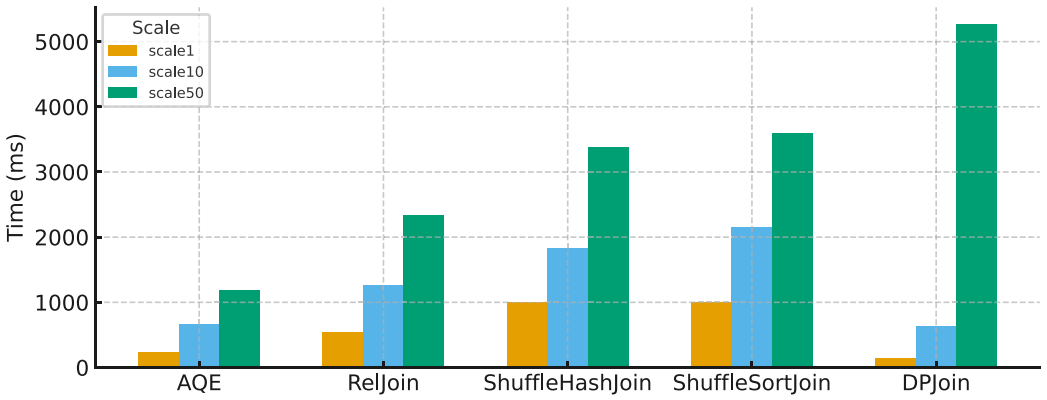

Figure 14 Average garbage collection on 99 queries time for all join strategies across scale factors (Replication 4). Full-size DOI: 10.7717/peerj-cs.3614/fig-14

Additionally, our analysis relies primarily on execution time. Metrics such as memory usage, shuffle volume, network traffic, and system behavior under concurrent workloads would provide a broader view of optimizer efficiency. Evaluating strategies in heterogeneous clusters, where node capabilities differ, would also reveal how robust each optimizer is in real-world settings.

Another important consideration is whether DPJoin introduces additional overhead in cases where the optimizer decides to use lower degrees of parallelism. DPJoin reduces unnecessary parallelism and avoids excessive shuffle operations, which helps lower network and coordination costs. This behavior does not introduce notable overhead in our experimental environment because the parallelism decisions remain aligned with the data volumes and cluster capacity at small and medium scales. However, the effect becomes

more pronounced at larger data sizes, where lower parallelism increases per-task memory usage and amplifies garbage-collection time. This suggests a clear avenue for improvement: DPJoin can be extended with resource-aware optimization that incorporates available executor memory and other cluster-level resource constraints into its cost model. Such an enhancement would allow DPJoin to adjust its parallelism decisions more smoothly at large scales and avoid memory pressure. We plan to address this limitation in future work.

In summary, DPJoin demonstrates strong and stable performance at small and medium scales, while AQE remains dominant at larger scales. RelJoin performs moderately under our constrained environment but may show stronger behavior in larger clusters. Future work will explore adaptive extensions to DPJoin, resource-aware cost modeling, larger datasets, increased hardware resources, more operators, and a richer set of performance metrics to build more comprehensive and practical distributed join optimization strategies.

LIMITATIONS

This study has several limitations related to computational resources, cluster size, and dataset scale. Each experiment was repeated three times, which was sufficient to observe consistent trends but not enough for strong statistical confidence. Increasing the number of repetitions would significantly extend the total execution time, since the full TPC-DS workload must be executed across multiple optimizers and replication configurations.

The largest dataset used in our evaluation was 16 GB. This is smaller than typical large-scale TPC-DS deployments and mainly reflects the compute and time budget available on our cluster. Some effects that emerge only at larger scales, such as severe memory pressure, deep pipeline behavior, or skew-induced imbalance, may therefore not be fully visible in our current results.

All experiments were executed on a shared-nothing cluster with four worker nodes and one master node. When the replication factor approaches the number of worker nodes, many blocks become locally available, reducing the sensitivity of some strategies to replication changes. A larger cluster would allow a more detailed evaluation of data locality, network saturation, and coordination behavior under varying levels of parallelism.

Although TPC-DS queries contain operators such as aggregations and sorts, our analysis primarily focused on join behavior. We did not isolate the individual cost contributions of other operators, nor did we evaluate how these operators interact with DPJoin's parallelization decisions. Future work will address these limitations by increasing iteration counts, scaling experiments to larger datasets and clusters, incorporating additional benchmarks such as TPC-H and the Join Order Benchmark, and studying skewed or heterogeneous environments to better understand the scalability and robustness limits of the proposed approach.

CONCLUSION

This study introduced DPJoin, a distributed join strategy that dynamically adjusts parallelism levels based on the replication factor and query characteristics. Through extensive experimentation across multiple data scales and replication settings, we

demonstrated that both parallelism and replication play a significant role in improving query performance in distributed environments.

DPJoin consistently outperformed baseline strategies such as RelJoin, ShuffleHashJoin, and ShuffleSortJoin in terms of the number of fastest executions. In most configurations, it also surpassed AQE, particularly at smaller and medium data scales. However, on the largest data scale (scale 50), AQEJoin achieved better average execution times, indicating that adaptive mechanisms can provide sharper performance on specific query subsets. This highlights the importance of balancing general-purpose robustness with targeted optimization.

The experiments confirmed that increasing the replication factor improves overall performance, especially for complex join queries. Higher replication allows for more flexible data placement and execution plans, which in turn reduces shuffle overhead and coordination cost. DPJoin effectively leveraged this replication awareness to scale with query complexity and data volume.

However, the evaluation also revealed certain limitations. Although DPJoin proved robust across a wide range of queries, its performance on large data scale or computationally intensive queries can still be optimized. The results also indicate that current cost models may benefit from integration with adaptive planning techniques, especially under resource constraints or high-concurrency workloads.

This study focused primarily on the join operators. Future work should extend the evaluation framework to include additional heavy operators such as Aggregate and Sort, which commonly appear in analytical workloads and may interact nontrivially with join strategies. Furthermore, incorporating metrics beyond execution time, such as memory usage, shuffle volume, and network traffic, would provide a more comprehensive understanding of the effectiveness of the strategy in distributed systems.

In summary, the proposed DPJoin strategy demonstrates that dynamic parallelization informed by replication can lead to notable improvements in distributed query performance. The findings lay the groundwork for future extensions involving hybrid execution plans, adaptive cost models, and broader operator integration in modern distributed query engines.

DATA AND CODE AVAILABILITY

All datasets and scripts used in the experiments are fully accessible. The experimental data is based on the publicly available TPC-DS benchmark, which can be generated following the official instructions at: http://www.tpc.org/tpcds/.

The exact datasets used in this study (SF1, SF10, SF50), generated using the official TPC-DS toolkit, have been published in Zenodo and are available at: https://doi.org/10.5281/zenodo.17866010.

The full implementation of DPJoin, including all source code, configuration scripts, and experiment drivers, is publicly available in our open-source repository: https://github.com/fatihtrkmen/dpjoin. This repository contains detailed instructions for reproducing all experiments described in this article.

FUTURE WORK

Future work will explore larger datasets that go beyond the scales used in this study. Increasing data volume will help evaluate the behavior of DPJoin under conditions that better reflect real-world analytical workloads. We also plan to scale the input size to substantially larger datasets (*e.g.*, 100 GB and above), which will allow us to study scenarios where memory pressure, deep pipelines, and network saturation become more pronounced. We also plan to use larger cluster configurations to study how replication, parallelism, and data locality interact when more worker nodes are available.

Another direction is to increase the number of experimental iterations. More repetitions will provide stronger statistical confidence and allow us to measure execution variance more accurately. Larger compute resources will make this possible.

Future studies will also examine additional operators. Analytical workloads commonly include heavy operators such as Aggregation and Sort. Evaluating these operators separately and together with joins will give a more complete understanding of system behavior.

In addition, we plan to extend the experimental evaluation to other established benchmarks such as TPC-H and the IMDb Join Order Benchmark (JOB). TPC-H includes decision-support queries with larger intermediate results and more complex operator pipelines, which will allow us to assess how DPJoin behaves under heavier analytical workloads. The IMDb JOB benchmark, on the other hand, is specifically designed to stress-test join ordering and includes highly selective predicates and complex join graphs. Incorporating these benchmarks will provide a broader and more realistic view of DPJoin's robustness, enable deeper comparisons with existing optimizers, and evaluate whether our parallelization decisions remain effective across different schema designs, cardinality patterns, and join shapes.

Finally, we will consider broader performance metrics. Execution time is important but not sufficient on its own. Future evaluations will include memory usage, shuffle volume, network traffic, and coordination overhead. These metrics will offer a more detailed view of how each strategy behaves under different distributed environments.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Fatih Türkmen conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.

- Belgin Ergenç Bostanoğlu conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The source code is available at GitHub and Zenodo: https://github.com/fatihtrkmen/dpjoin

TÜRKMEN, F. (2025). fatihtrkmen/dpjoin: dpjoin (dpjoin). Zenodo. https://doi.org/10.5281/zenodo.18099913.

The data is available at Zenodo:

TÜRKMEN, F. (2025). TPC-DS Benchmark Dataset Generated for DPJoin Optimization Experiments [Data set]. Zenodo. https://doi.org/10.5281/zenodo.17866010.

REFERENCES

Apache Software Foundation. 2022a. Apache Hadoop 3.3.2 release. *Available at https://hadoop.apache.org/release/3.3.2.html* (accessed 18 August 2025).

Apache Software Foundation. 2022b. HDFS architecture guide. *Available at https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html* (accessed 15 January 2025).

Apache Software Foundation. 2023. Apache Spark 3.3.2 release. *Available at https://spark.apache.org/news/spark-3-3-2-released.html* (accessed 18 August 2025).

Bernstein PA, Goodman N. 1981. Power of natural semijoins. *SIAM Journal on Computing* **10(4)**:751–771 DOI 10.1137/0210059.

Cubukcu U, Erdogan O, Pathak S, Sannakkayala S, Slot M. 2021. Citus: distributed PostgreSQL for data-intensive applications. In: *Proceedings of the 2021 International Conference on Management of Data*. New York: ACM.

Dean J, Ghemawat S. 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM* **51**:107–113 DOI 10.1145/1327452.1327492.

DeWitt DJ, Gerber RH, Graefe G, Heytens ML, Kumar KB, Muralikrishna M. 1986. Gamma—a high performance dataflow database machine. In: *Proceedings of the 12th International Conference on Very Large Data Bases*.

Du Y, Cai Z, Ding Z. 2024. Query optimization in distributed database based on improved artificial bee colony algorithm. *Applied Sciences* **14(2)**:846 DOI 10.3390/app14020846.

Garofalakis MN, Ioannidis YE. 1996. Multi-dimensional resource scheduling for parallel queries. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. New York: ACM.

Giampà S, Belcastro L, Marozzo F, Talia D, Trunfio P. 2021. A data-aware scheduling strategy for executing large-scale distributed workflows. *IEEE Access* **9**:47354–47364 DOI 10.1109/access.2021.3067815.

Gounaris A, Kougka G, Tous R, Montes CT, Torres J. 2017. Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems* **28(7)**:1891–1904 DOI 10.1109/tpds.2017.2647939.

Gourishetti S. 2024. Performance optimization in distributed SQL environments: a comprehensive analysis of presto query engine. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* **10(6)**:241–253 DOI 10.32628/cseit24106173.

Graefe G. 1995. The cascades framework for query optimization. *IEEE Data Engineering Bulletin* **18**:19–29.

Graefe G, McKenna WJ. 1993. The volcano optimizer generator: extensibility and efficient search. In: *Proceedings of the Ninth International Conference on Data Engineering*. Piscataway: IEEE.

Jayashree J. 2013. Distributed database management system and query processing. *International Journal of Computer Science and Technology* **4(3)**:182–185.

Liang F, Lau FC, Cui H, Li Y, Lin B, Li C, Hu X. 2024. RelJoin: relative-cost-based selection of distributed join methods for query plan optimization. *Information Sciences* **658**:120022 DOI 10.1016/j.ins.2023.120022.

Liang F, Lau FCM, Cui H, Wang C-L. 2018. Confluence: speeding up iterative distributed operations by key-dependency-aware partitioning. *IEEE Transactions on Parallel and Distributed Systems* **29(2)**:351–364 DOI 10.1109/tpds.2017.2756054.

Markl V, Raman V, Simmen D, Lohman G, Pirahesh H, Cilimdzic M. 2004. Robust query processing through progressive optimization. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York: ACM.

Neumann T. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* **4(9)**:539–550 DOI 10.14778/2002938.2002940.

Ninan A. 2023. Performance tuning and optimization of apache spark applications. *International Journal of Computer Trends and Technology* **71(5)**:10–14 DOI 10.14445/22312803/ijctt-v71i5p103.

Pavlopoulou C, Carey MJ, Tsotras VJ. 2023. Revisiting runtime dynamic optimization for join queries in big data management systems. *ACM SIGMOD Record* **52(1)**:104–113 DOI 10.1145/3604437.3604460.

Phan A-C, Phan T-C, Trieu T-N, Tran T-T-Q. 2021. A theoretical and experimental comparison of large-scale join algorithms in spark. *SN Computer Science* **2(5)**:5:1–5:16 DOI 10.1007/s42979-021-00738-x.

Polychroniou O, Zhang W, Ross KA. 2018. Distributed joins and data placement for minimal network traffic. *ACM Transactions on Database Systems* **43(3)**:14:1–14:45 DOI 10.1145/3241039.

Quoc DL, Akkus IE, Bhatotia P, Blanas S, Chen R, Fetzer C, Strufe T. 2018. ApproxJoin: approximate distributed joins. In: *Proceedings of the ACM Symposium on Cloud Computing*. New York: ACM.

Ren Z, Yun N, Shi W, Li Y, Wan J, Yu L, Fan X. 2018. Characterizing the effectiveness of query optimizer in spark. In: *2018 IEEE World Congress on Services (SERVICES)*. Piscataway: IEEE.

Sarma AD, Afrati FN, Salihoglu S, Ullman JD. 2013. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment* **6(4)**:277–288 DOI 10.14778/2535570.2488334.

Schneider DA, DeWitt DJ. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *ACM SIGMOD Record* **18(2)**:110–121 DOI 10.1145/66926.66937.

Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG. 1979. Access path selection in a relational database management system. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. New York: ACM.

Stillger M, Lohman GM, Markl V, Kandil M. 2001. Leo—DB2's learning optimizer. In: *Proceedings of the 27th International Conference on Very Large Data Bases*.

Transaction Processing Performance Council. 2021. TPC-DS benchmark. *Available at http://www.tpc.org/tpcds/* (accessed 30 June 2025).

Wu X, He Y. 2023. Optimization of the join between large tables in the spark distributed framework. *Applied Sciences* **13(10)**:6257 DOI 10.3390/app13106257.

Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: *Proceedings of the 5th European Conference on Computer Systems*. Piscataway: IEEE.

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. New York: ACM.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I. 2016. Apache spark: a unified engine for big data processing. *Communications of the ACM* **59(11)**:56–65 DOI 10.1145/2934664.

Zhang X, Zhang H, Meng X. 2025. Intra-query runtime elasticity for cloud-native data analysis. *Proceedings of the ACM on Management of Data* **3(3)**:1–28 DOI 10.1145/3725315.

Zhao Y, Chen R. 2021. Spark SQL query optimization based on runtime statistics collection. In: *2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. Piscataway: IEEE.

Zhao Y, Chen R. 2022. Research on runtime query optimization technology of spark SQL. *International Journal of Computer Theory and Engineering* **14(1)**:15–19 DOI 10.7763/ijcte.2022.v14.1305.