



Beyond Shuffling: PAC-Tree for Efficient Distributed Joins via Hierarchical Data Placement

Pengju Liu^{1,2} · Cuiping Li^{1,2} · Hong Chen^{1,2}

Received: 5 June 2025 / Revised: 5 September 2025 / Accepted: 28 October 2025 / Published online: 7 January 2026
© The Author(s) 2025

Abstract

Efficiently execution of join operations over large-scale distributed data is critical in modern analytical engines, yet costly shuffle operations involving extensive network transfer and remote I/O often hinder performance. Existing data layout techniques struggle with complex multi-table joins due to inflexible distribution key assignments, suboptimal cluster key choices that require significant manual tuning, and weak co-optimization with downstream join strategy selection. This paper introduces PAC-Tree (**Partition-Aligned Co-Join Tree**), a novel query-aware data layout designed to comprehensively address these limitations in an end-to-end manner. PAC-Tree features a two-level hierarchical structure: the upper level replicates fact tables with diverse distribution keys to improve co-location across join patterns, while the lower level applies advanced multidimensional partitioning with predicate-aware splits for fine-grained intra-shard data skipping. Leveraging PAC-Tree's statistics, we further propose a block-grouping algorithm to construct co-partitions under memory constraints and a PRIM-based reordering policy for multi-table joins, together significantly reducing shuffle overhead. Experiments on a real Spark cluster demonstrate that PAC-Tree reduces query latency by up to 45.7% and improves throughput by up to 2× compared with state-of-the-art layouts, especially for join-heavy workloads.

Keywords Data layout · Distributed joins · Hierarchical partitioning tree · Data skipping · Data shuffling reduction

1 Introduction

In modern analytical systems [1–3], efficiently leveraging limited resources to process large-scale data is a critical indicator of system design effectiveness. Data partitioning plays a pivotal role in enhancing system performance by supporting both parallel processing and hierarchical data management [4]. By constructing predetermined mapping

functions based on query access patterns, partitioning can group highly correlated data into the same logical partitions [5], thereby minimizing inter-partition data reads and movements. Each logical partition is instantiated as one or more uniformly-sized data blocks, persistently stored on local disks or remote storage.

Within analytical workloads, join operators often constitute the primary performance bottleneck in query execution plans [6]. The core challenge arises because join processing frequently requires scanning and redistributing data across nodes so that tuples with identical join keys are co-located. Such redistribution is typically implemented via shuffle operations, which entail large-scale data transfers over the network, along with serialization, deserialization, and remote I/O overhead. These costs are especially pronounced for high-cardinality join keys, where intermediate results are widely scattered across the cluster and cannot be effectively co-located. Consequently, cross-node data exchange and shuffle operations can dramatically amplify system I/O and network load. To mitigate these costs, data partitioning [6] can be applied before formulating the join strategy. By jointly partitioning multiple tables based on their join keys,

✉ Pengju Liu
liupengju@ruc.edu.cn
Cuiping Li
licuiping@ruc.edu.cn
Hong Chen
chong@ruc.edu.cn

¹ Engineering Research Center of Database and Business Intelligence, Ministry of Education, Haidian Huangzhuang, Haidian District Beijing 100872, People's Republic of China

² School of Information, Renmin University of China, No. 59 Zhongguancun Street, Haidian District Beijing 100872, People's Republic of China

relevant tuples (i.e., the outputs of TableScan with Filters) can be co-located on the same machine node, thereby creating more opportunities for local join execution. The process can be divided into three phases: (1) **selecting optimal distribution keys for each table to control data sharding** and minimize shuffle costs during joins and aggregations; (2) **choosing appropriate cluster keys within each shard (or machine) node to optimize internal data layout**, enhance data skipping, and reduce scan overhead; and (3) **enabling the cost-based optimizer (CBO) to determine the most efficient join strategy** (including join type and order) by leveraging knowledge of how joining data is distributed across machines.

The profound impact of data layout on multi-table query execution is illustrated in Fig. 1. Consider a query joining between the TPC-H [7] *Orders* and *Lineitem* tables with a filter on *Orders.orderdate*. Upon query submission, compute nodes request the referenced data blocks from the storage layer, materialize the required partitions locally, execute the hash join, and return the results. There are two optional layouts, with each partitioning *Orders* into four blocks and *Lineitem* into two blocks. Layout 1 distributes data by primary keys (*orderkey* and *partkey*) across two nodes and sorts locally within each node. Layout 2, in contrast, uses the join key (*orderkey*) as the distribution key for both tables and the filter column (*orderdate*) as the cluster key for sorting the *Orders* data. This Layout 2 design reduces the number of partition blocks loaded into compute nodes during join processing. Specifically, compared to Layout 1, which requires all-to-all pairwise joins, it reduces the number of hash joins from four

to two and the scanned blocks from twelve to four, leading to substantial performance gains.

The current research landscape implements join-aware data partitioning but suffers from several fundamental limitations as follows:

(1) *Inflexibility in Data Distribution and Clustering.* First, to identify optimal distribution keys, various data layout techniques [8–13] have been proposed, which extract join graphs from workload traces to pinpoint high-frequency join keys, aiming to maximize local join opportunities. Nevertheless, in complex join scenarios, a single, fixed sequence of distribution keys is insufficient to accommodate all join patterns. For example, consider three fact tables (M_{1-3}) to be joined sequentially using keys c_1 and c_2 (i.e., $M_1 \bowtie_{c_1} M_2 \bowtie_{c_2} M_3$). Irrespective of the distribution key chosen for table M_2 , it is impossible to optimally collocate data for both joins at the same time.

Second, traditional techniques arrange intra-machine data using cluster keys with uniform value ranges (e.g., linear order, Z-order [14], and Hilbert-order [15]). Selecting and ordering these cluster keys often requires extensive manual tuning and only partially facilitates block skipping. More recent approaches [5, 6, 16–19] that leverage query predicate information have attracted attention. These techniques improve block skipping opportunities by exploring dependencies among filtered data and sorting accordingly, and have been successfully validated in production systems such as Databricks and Redshift.

(2) *Lack of Integration with Global Query Optimization.* Aside from MTO [17], most existing query-aware layout methods have not been specifically optimized for join queries. Although MTO exploits join-induced filtering

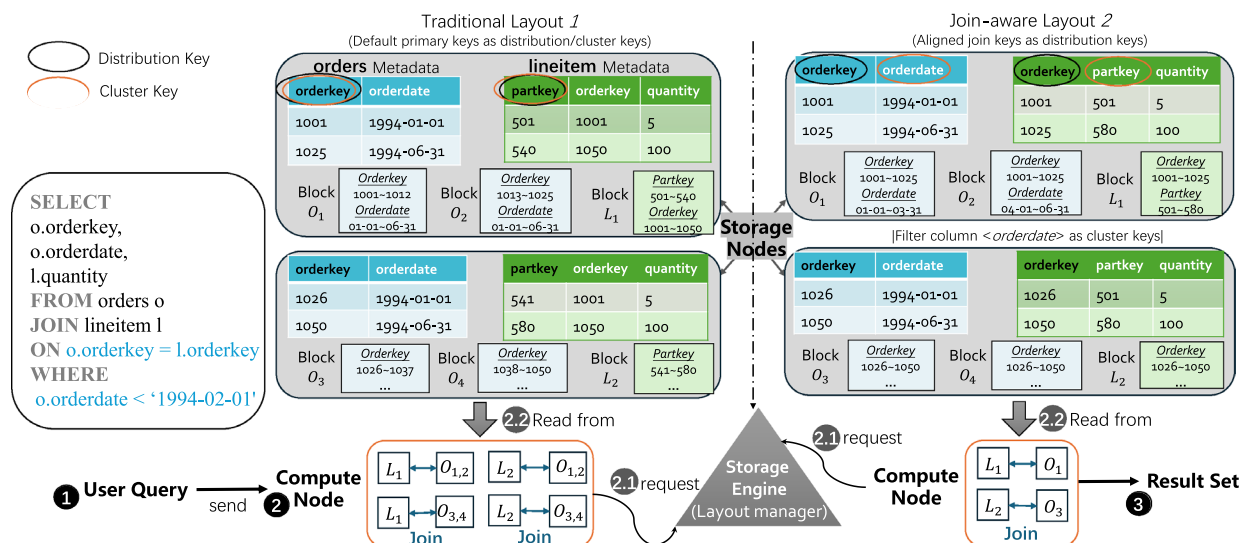


Fig. 1 A motivating example of join-aware layouts for query optimization. Layout 1 partitions by default primary keys with local sorting; Layout 2 co-locates *Orders* and *Lineitem* on the join key

(*orderkey*) and clusters *Orders* by *orderdate*, reducing remote reads and join pairs (4 $\rightarrow$ 2) as well as scanned blocks (12 $\rightarrow$ 4)

conditions to concurrently reduce the number of blocks that need to be processed for each join, it merely treats join filters as equivalent to other filters. This diminishes their relative importance and overlooks the substantial data-shuffling costs inherently associated with join operators, which are fundamentally different from the block-skipping benefits of standard filter operators. Furthermore, MTO does not fully exploit such layout-aware prior information to guide the subsequent selection of optimal join strategies.

To address these limitations, this paper proposes the **partition-aligned co-Join Tree** (PAC-Tree¹), which integrates optimizations from both data statistics and query execution plans to improve end-to-end query processing. Our core idea is to create a layout structure that not only optimizes data access but also provides rich metadata to guide downstream query optimization stages. Specifically, PAC-Tree features a two-level hierarchical structure. The upper level partitions tables based on distribution keys and allows different distribution keys for replicas of the same fact table to leverage data replication. This approach alleviates resource contention among diverse join patterns. The lower level focuses on intra-shard organization by assigning appropriate cluster keys for multidimensional partitioning within each data shard. This mechanism inherits the partitioning logic of MTO while incorporating bounding splits at leaf nodes to further enhance partitioning efficiency. The detailed statistical and structural information captured by PAC-Tree enables precise identification of the relevant data blocks for each join side. Building upon this, we propose: (1) an efficient block-grouping algorithm that builds joint partitions for local joins under memory constraints, minimizing data shuffling; and (2) a join reordering strategy informed by PAC-Tree's view of data distribution, using the PRIM [20] algorithm to optimize the overall execution plan.

The main contributions of this paper are as follows:

- We propose PAC-Tree, a novel two-level data layout structure specifically optimized for multi-table join queries, which significantly reduces data access I/O in join-heavy analytical workloads.
- We introduce a distribution key selection strategy that leverages data replication to create multiple PAC-Tree replicas for tables involved in frequent or conflicting join patterns, thereby substantially reducing costly shuffle operations.
- We develop a PAC-Tree-aware block grouping algorithm and join reordering method that maximize local join opportunities while minimizing shuffle joins.
- We conduct a systematic end-to-end evaluation of PAC-Tree on a real Spark cluster. Compared to existing data

layout strategies, experimental results demonstrate a 42.2% reduction in query latency and a $2.1\times$ improvement in throughput, with sensitivity analyses confirming its robustness.

2 Related Work

Physical Database Design. Similar to partitioning, indexes and materialized views are also widely employed in physical database design. Indexes can be categorized into one-dimensional indexes (e.g., B-tree [21]) and multi-dimensional indexes (e.g., Kd-tree [22], R-tree [23], and its variant, RW-tree [24]). However, these indexing techniques generally are less effective on high-dimensional data and do not explicitly optimize join scenarios. On the other hand, materialized views [25, 26] employ a space-for-time strategy by pre-calculating and storing query results. This decouples query execution from accessing raw data, allowing queries to be directed to the most appropriate pre-computed view. Nonetheless, their optimizations primarily focus on queries featuring multiple group-by keys and complex aggregation computations. In contrast, the PAC-Tree approach is specifically designed to optimize scan and join operators.

Data Partitioning for Single-table Queries. (i) Data-driven approaches rely solely on data distribution and include methods such as hash and range partitions, SMA [27], Zone Maps [13, 28], and Kd-tree [22] strategies. These methods are typically workload-agnostic, making them suitable for scenarios with infrequent data updates where stable, albeit not always optimal for specific queries, performance is desired. (ii) Query-driven approaches consider both data distribution and workload characteristics. For instance, Sun et al. [29] employed the Apriori [30] algorithm to mine representative filters from historical workloads, using them as candidate splits. Yang et al. [16] developed a query data routing tree (QdTree) using a greedy strategy, while PAW [18] generates irregularly shaped partitions and integrates two plugins (precise descriptor & storage tuner) to optimize QdTree. Similar to SOP [29], MDDL [19] augments each table with an additional column (mddl_col) that stores an n -length one-hot vector for each tuple, indicating which of n predicates it satisfies. The table is then sorted by mddl_col, which forms the basis for partitioning data into file pages, while maintaining the min-max values of mddl_col for each page. At query time, the bitmask encoding of the query predicates is bitwise-ANDed with the min and max values for each file page, enabling rapid page skipping.

However, such layouts are optimized only for workloads that target individual tables. In practice, workloads frequently span multiple tables with diverse join patterns, such as those encountered in star and snowflake schemas [31].

¹ Our implementation is available at <https://gitfront.io/r/bytedance1598/QtexXnJu8Ntg/PAC-tree/>.

These challenges constitute the primary optimization targets of PAC-Tree.

Data Partitioning for Multi-table Queries. CoHadoop [32] proposed a co-partitioning policy in HDFS [33] that significantly accelerates join operations between two large tables by reducing data shuffling. Despite this, its benefits are limited when joins involve more than two tables. PREF [9] addressed data locality by co-partitioning multiple tables based on predefined join relationships and table schemas. GPT [10] further reduces data redundancy in PREF. First, it selects vertices and edges from the join graph, considering storage overhead and query benefits of shuffle-free execution. Then, it applies multi-column partitioning to hash partition key values for each edge. BaW [11] presents an assumption-free framework that transforms the partition key recommendation problem as an exact graph matching problem. Building on these ideas, Ding et al. [17] proposed the MTO layout, which extends QdTree to join scenarios by incorporating additional join-induced predicates [34] into the candidate split set to prune more blocks before joining.

While techniques such as BaW can identify co-partitioning keys and MTO can build layouts accordingly, merely combining these approaches only reduces the number of joined blocks and the resulting co-partitions. In contrast, PAC-Tree further leverages knowledge of how co-partitions are distributed across compute nodes to optimize the multi-table join order.

3 PAC-Tree Overview

In this section, we present a high-level overview of the PAC-TREE algorithm, highlighting its workflow and strengths for optimizing complex distributed joins. The

Table 1 Symbols and Notation

Symbol	Definition
$M(c_1 \sim c_m)$	a relation with m columns $c_1, \dots, c_m$.
$Q(q_1 \sim q_n)$	a set of n queries.
$K(k_1 \sim k_n)$	a set of selected join keys.
T	a hierarchical tree for routing logical partitions.
$\mathcal{R}(R_1 \sim R_n)$	a set of logical partitions $\{R_1, \dots, R_n\}$.
$\mathcal{R}_{acc}(q, T)$	a set of partitions accessed by query q on tree T .
P	a co-partition that consists of multiple joined blocks.
C_{-}	the cost estimate for various types of operations ().
N_{-}	this symbol represents the number of objects ()
b_{min}	the minimal block size in default storage settings.
r_{-}	permitted replication factor for different tables ().
κ	hyperparameter 1: used to limit the maximum depth of top-part tree.
θ	hyperparameter 2: a threshold that used to filter out low-cardinality keys and mitigate data skew.

notation summarized in Table 1 serves as a quick reference for the mathematical and technical terms used throughout this paper.

3.1 Problem Formulation

Query-aware data partitioning refers to constructing a data layout for each table based on collected query patterns. This layout comprises multiple logical partitions, enabling more efficient data retrieval during query execution. To facilitate the construction and maintenance of non-overlapping logical partitions, an index tree structure is often employed. Since join operations are typically the most expensive operators in distributed query processing, hierarchical join trees are proposed (See Definition 1) to optimize multi-table joins.

Definition 1 (Hierarchical Join Tree) A join tree T is a coarse-grained index structure where each non-leaf node stores metadata including the domains of its child nodes and their corresponding storage addresses. Each leaf node corresponds to a logical partition R , and the number of leaf nodes equals the number of partitions. The join tree is organized in two hierarchical layers: the top layer T^t , built over join attributes, and the bottom layer T^b , constructed over all high-selectivity attributes. Formally, the structure is denoted as $T = T^t \bowtie T^b$.

Based on this background, we formally define the central problem addressed in this paper.

Problem 1 (Workload-aware Logical Partitioning Optimization) Given a set of n tables $\{M_1, \dots, M_n\}$ and a representative historical workload Q_H , we assume that $Q_F \approx Q_H$ holds over a sufficiently long interval, where Q_F denotes the future high-frequency workload. The goal is to construct a join tree T_i for each table M_i , and use it to partition the data into a layout $\mathcal{R}_i$, such that the total I/O cost of executing Q_H over the resulting layout set $\mathbb{R} = \{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n\}$ is minimized. Formally:

$$T_i = \underbrace{T_i^t(Q_H)}_{\text{Top part}} \bowtie \underbrace{T_i^b(Q_H)}_{\text{Bottom part}} \quad (1)$$

$$\mathbb{R}' = \arg \min_{\mathbb{R}} \sum_{i=1, \mathcal{R}_i \in \mathbb{R}}^n \mathcal{C}(\mathcal{R}_i(T_i), Q_H)$$

Here, $T_i(Q)$ denotes the join tree extended according to the captured workload patterns. $\mathcal{R}_i(T_i)$ represents the logical partitioning of table M_i induced by its join tree T_i . The cost function $\mathcal{C}(\cdot, Q_H)$ estimates the total I/O overhead incurred when executing the workload Q_H on the resulting partition layout.

Finding a globally optimal data layout that minimizes this cost is computationally intractable due to the combinatorial nature of the partitioning space. Even optimally partitioning a single table for a given workload is NP-hard [35]. The challenge is further amplified for multiple tables, as the search space must account for all possible joint partitioning configurations. We formalize this in Theorem 1 as follows:

Theorem 1 *The workload-aware multi-table partitioning problem (defined in Problem 1) is NP-hard.*

Proof The proof is by restriction. Consider a special instance of Problem 1 where the number of tables $n = 1$. In this case, the objective function: $\min_{\mathcal{R}} \sum_{i=1}^n \mathcal{C}(\mathcal{R}_i(T_i), Q_H)$ reduces to the optimization problem for a single table: $\min_{\mathcal{R}_1} \mathcal{C}(\mathcal{R}_1(T_1), Q_H)$. This is precisely the formulation of the single-table partitioning problem, which is known to be NP-hard [35]. Since our general problem contains an NP-hard problem as a specific instance, the general problem itself must be NP-hard. A polynomial-time algorithm for the multi-table problem would imply a polynomial-time solution for the single-table case by setting $n = 1$, which is a contradiction.

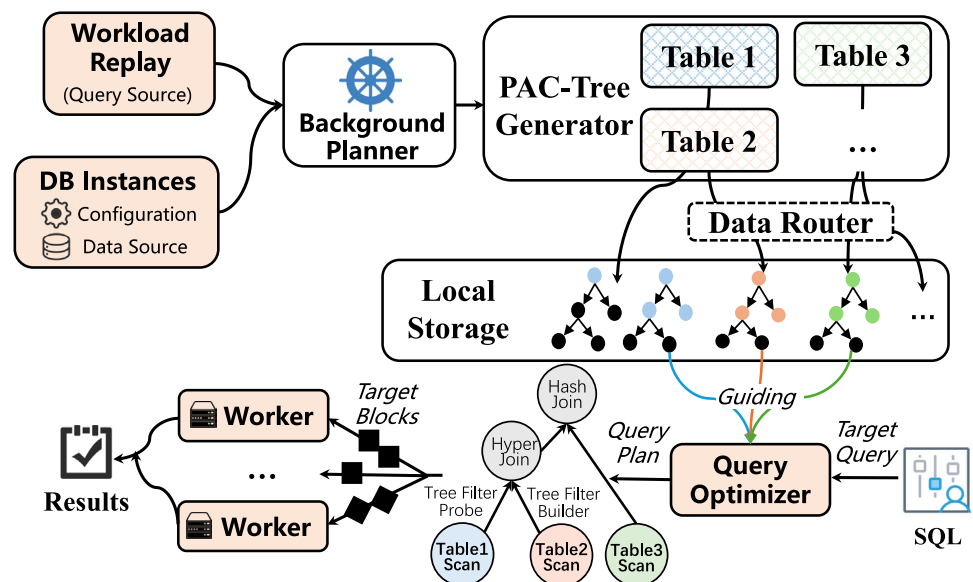
The NP-hardness of the problem implies that finding exact solutions is intractable for non-trivial scenarios. Consequently, practical systems must rely on heuristic or approximate methods. This motivates our query-aware approach, PAC-TREE, which is designed to achieve significant performance gains with high-quality, albeit potentially sub-optimal, layouts in reasonable time.

3.2 PAC-Tree Workflow

Figure 2 illustrates the complete life cycle of the PAC-TREE algorithm:

- Initially, a dedicated real-time component, background planner, periodically monitors and ingests key features from three sources: query workloads, fundamental database configuration, and sampled table data. First, query logs are analyzed to extract information on accessed columns, filter predicates, join keys, and their occurrence frequencies. Second, the database configuration provides parameters such as compute workers' memory capacity, the number of available workers, storage node capacity, typical partition file sizes (e.g., 256MB per file in HDFS), and network latency. Third, sampled table data supplies details of the .part files generated by the storage engine, including dataset size, sampling ratio, table schema, column types, and data distributions.
- Subsequently, in the offline construction phase, PAC-TREE leverages historical telemetry data to determine the optimal data layout for each table. The hierarchical PAC-Tree structure functions as a data router for instantiating data blocks. Each table can utilize one or several PAC-trees, constructed based on different column keys; the number of PAC-trees is determined by the replication factor. By constructing multiple replicas of the data layout, PAC-Tree maximizes co-partitioning opportunities and facilitates a greater number of local joins.
- Finally, in the online inference phase, the query optimizer utilizes PAC-trees (as indices) to rapidly generate improved query plans for incoming queries. This involves quickly searching for relevant blocks to apply filter operations and determining the optimal join order.

Fig. 2 Overview of the PAC-TREE Workflow



for tables. Moreover, when joining large tables, to fully exploit limited worker memory while minimizing expensive data shuffling, we propose a novel join operator, namely the co-partition-based hyper join (see Sect. 5), as an alternative to the default sort-merge join. The query executor then pipelines the physical execution operators to produce the final query results.

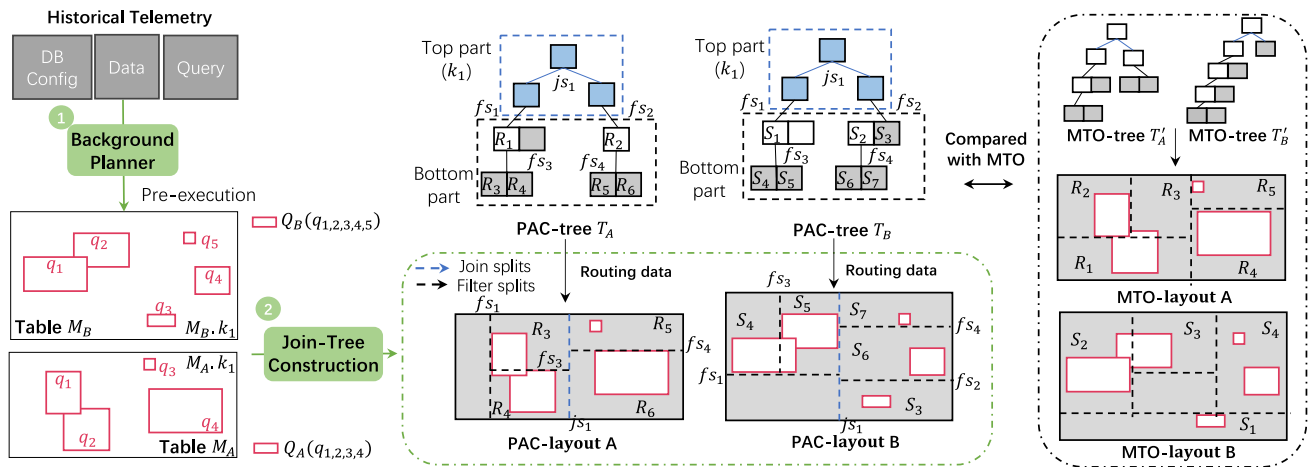
To provide a more concrete understanding of how these offline construction and online inference phases operate and deliver benefits, Figure 3 presents a motivating example that depicts the generation of PAC-layout for a two-table join query ($Q_{A,B}$) with filters on tables M_A and M_B and contrasts its advantages with a traditional MTO-layout.

3.3 Offline Layout Optimization

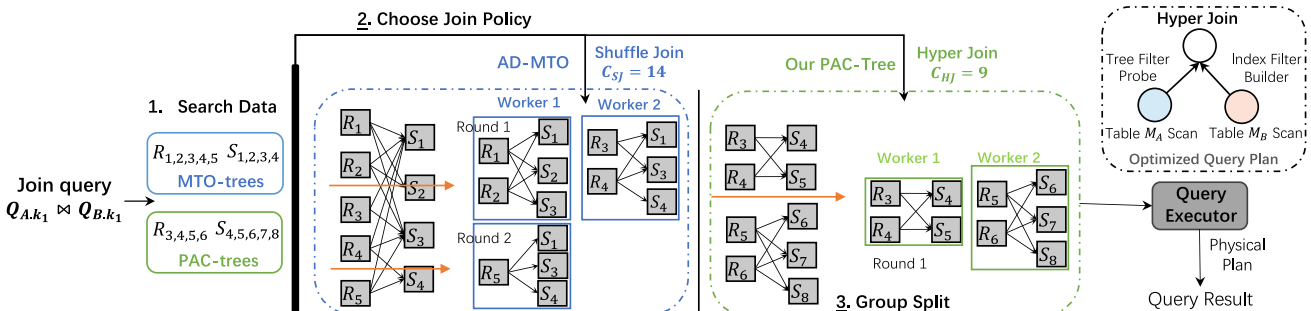
Our system employs the SQLParse library [36] to parse queries into logical filters (in DNF and CNF forms), and estimates the accessed column domains (i.e., TupleDomains) by evaluating them on sampled tables. In Fig. 3(a), we have visualized the scanning ranges of $Q_{A,B}$ within a two-dimensional table space defined by $M_A(k_1, k_2)$, $M_B(k_1, k_3)$,

where each red rectangle represents a dependent filter, and the column key k_1 serves as the primary key for M_A and the foreign key for M_B .

For each table M_i , we instantiate a dedicated hierarchical join tree T_i . Taking the PAC-tree T_A as an example, we first identify median conditions ($\sum_{js_i} \in JS$) of join-related columns (k_1), and predicate/median-related conditions ($\sum_{fs_i} \in FS$) from other filters, as the candidate split set. With the entire table M_A as the root node, we build the upper part of the tree using JS and the lower part using FS . Both T_A and T_B are constructed with a total depth of 4, with 2 levels for the upper tree and 2 levels for the lower tree. By exploiting the hierarchical parent-child dependencies inherent in the tree, the original data is effectively routed into instantiated file blocks, thus forming the PAC-layout for each table. Within each tablespace, blue lines (js_i) indicate the splitting operations in the upper portion, while black lines (fs_i) represent those in the lower part. The split sequence aligns exactly with the tree construction process, and the leaf nodes are labeled in accordance with the partition files.



(a) Offline layout optimization phase



(b) Online query inference phase

Fig. 3 An Example Demonstrating the PAC-TREE Layout versus the MTO Layout

For comparison, the figure also presents a query-driven layout scheme based on the traditional MTO method. For query $Q_{A,B}$, both layouts initially appear to access a similar number of potentially relevant blocks. Under the PAC-layout, $Q_{A,B}$ scans 9 blocks, with $R_{acc}(Q_{A,B}, T_A) = (R_3, \dots, R_6)$ and $R_{acc}(Q_{A,B}, T_B) = (S_3, \dots, S_7)$ to retrieve necessary data. Under the MTO-layout, the query also accesses 9 blocks, where $R_{acc}(Q_{A,B}, T_A) = (R_1, \dots, R_5)$ and $R_{acc}(Q_{A,B}, T_B) = (S_1, \dots, S_4)$. The essential distinction lies in how the internal organization of blocks influences downstream query processing and pruning efficiency.

3.4 Online Query Inference

When executing the target join query $Q_A \bowtie_{k_1} Q_B$, the query plan undergoes optimization along two key dimensions to improve efficiency. First, its table scan costs are reduced by accurately searching the required data within the PAC-layout (compared to a query-oblivious layout like random partitioning). Second, the join execution policy is refined to reduce data shuffling. In distributed environments, where compute nodes are subject to memory capacity constraints, join tasks may require multiple rounds of data block loading. Fig.3(b) shows that the block-level join relationships differ significantly across layouts. In the case of the MTO-layouts, block connectivity is sparse and unordered, whereas PAC-layouts exhibit distinct partition boundaries, as the upper layers of both PAC-trees are consistently split based on the join key distribution. As a result, blocks are co-distributed in an ordered fashion, which simplifies inter-block join relationships.

Assuming that each compute node can hold two blocks at a time, we propose a group-based loading strategy. In the MTO case, a shuffle join is required: blocks like S_1 and S_3 must be broadcast to all workers, incurring 14 blocks (I/O operations). Alternatively, PAC-layouts leverage the hierarchical structure to enable a hyper join. Two co-partitions (groups), $\{R_{8,10} \bowtie R_{3,4}\}$ and $\{R_9 \bowtie R_{3,4}\}$, are constructed, with the corresponding M_B blocks $\{S_4, S_5\}$ and $\{S_6, S_7, S_8\}$ placed on different workers. Each worker then performs a local hash join, thus reducing the total I/O to 9 blocks.

In light of these observations, the default query plan is further optimized through filter push-down and join order refinement, prior to submitting the pipelined operators to the execution engine for final query results.

4 PAC-Tree Construction for Distributed Joins

This section introduces how to create a hierarchical PAC-tree for use in distributed join scenarios.

4.1 The Motivation for PAC-tree

Our intuitions come from representative benchmarks such as TPC-H and JOB, where we observe that on average 84.6% of fact tables contain more than two candidate join keys. This reflects inherently complex access relationships among fact tables, which can be elucidated by a hypergraph structure. Figure 4 shows such a hypergraph derived from TPC-H queries (Q2-Q8), where hypernodes represent tables and hyperedges capture join operations. Each hypernode further encapsulates subnodes corresponding to its join keys. In the case of Q8, where $lineitem \bowtie ?$ occurs simultaneously, multiple join trees must be built for the $lineitem$ table (i.e., $T_{s_1} \bowtie T_{l_1}$ and $T_{p_1} \bowtie T_{l_2}$) to create additional co-partitions. However, two key challenges remain. First, the default "distributed by" clause in SQL, even when configured with composite distribution over multiple keys, often fails to guarantee co-partitioning across all relevant join-key pairs, thus limiting co-partitioning opportunities. Second, this clause relies primarily on data distribution to partition the dataset, but it can frequently overlooking filtering operations that could significantly reduce data volume before joins.

To address the first challenge, it is necessary to relax the constraint that each hierarchical tree for a table must be built upon a single, predetermined (e.g., primary) join key. Consider queries that involve a join between two tables $M_A(c_A, r_A)$ and $B(c_B, r_B)$ on an unknown join key k , where c_A and c_B denote the number of distinct referenced join keys and r_A and r_B denote the number of replicas. The probability that the replica set of a table (e.g., M_A) contains a specific join key k is $p_A(k) = \frac{r_A}{c_A}$, $p_B(k) = \frac{r_B}{c_B}$, since selecting a fixed key in a non-repetitive combination has probability $\frac{r}{m}$. Consequently, the probability that the join can be localized on a replica pair is $p_{local}^{(M_A, M_B)}(k) = \frac{r_A}{c_A} \cdot \frac{r_B}{c_B}$. Compared with the baseline configuration where each table has a single replica ($r = 1$), for which $p_{local, baseline} = \frac{1}{c_A} \cdot \frac{1}{c_B}$, the probability gain factor is $gain = \frac{p_{local, replicas}}{p_{local, baseline}} = r_A \cdot r_B$. This flexibility enables better alignment of data blocks across different join operations involving that table. In practice, distributed systems [1, 3, 37] typically maintain multiple data replicas controlled by a replication factor that usually ranges from 2 to 4 to enhance availability and alleviate hotspots. PAC-Tree exploits this property by partitioning different replicas

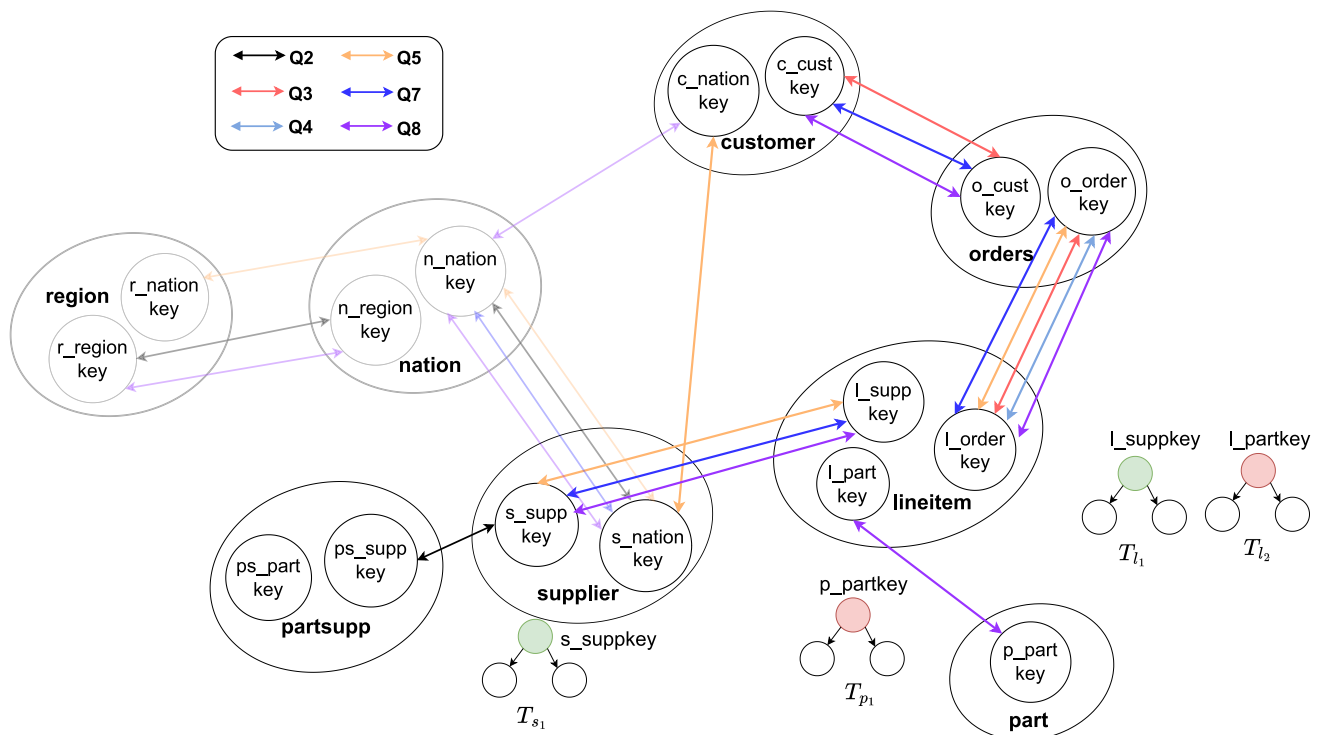


Fig. 4 For TPC-H queries Q2 to Q8, join operations are depicted by connecting lines, with same-colored lines indicating joins belonging to the same query. These operations typically involve multiple join keys

of the same table on different join keys without incurring additional storage overhead.

For the second challenge, recent work has demonstrated the efficacy of filter-driven partitions in enhancing data skipping. In particular, the AdaptDB strategy [6] organizes a partitioning tree in which the upper layers are partitioned by join keys, while the lower layers focus on filter predicates as cluster keys. This design strikes a balanced trade-off between co-partitioning and data skipping, thereby improving both join performance and overall query efficiency. PAC-Tree builds upon this insight.

Overall, Algorithm 1 outlines the PAC-Tree construction to tackle the above challenges, comprising three partitioning stages. Stage 1 (Sect. 4.2) selects the top n representative join keys for each table (where n is the maximum allowed number of replicas) to prepare for per-replica PAC-Trees. Stage 2 (Sect. 4.3) builds the tree's top part based on the distribution of these join keys. Stage 3 (Sect. 4.4) constructs the bottom part based on access patterns and distributions of other columns.

and include both fact and dimension tables. Importantly, dimension tables are illustrated as translucent ellipses due to their relatively low join cost

```

1 Tables to be partitioned  $\mathbf{M}$ , PAC-Trees  $\mathbf{T}_{best} \leftarrow \emptyset$ 
2 // Stage 1: Join Key Selection
3 Obtain optimal join keys  $K_i$  for each table  $M_i$ 
4 foreach  $K_i \in \mathbf{K}\{K_1, \dots, K_n\}$  do
5   foreach  $k_j \in K_i$  do
6     // Stage 2: Top-Part Construction
7     Create the top part of the PAC-Tree  $T_j$ 
8     // Stage 3: Bottom-Part Construction
9     Extract all candidate split conditions  $S_s$ 
10    Greedily pick an item from  $S_s$  to extend  $T_j$ 
11    Add  $T_j$  to  $\mathbf{T}_{best}$ 
12 return  $\mathbf{T}_{best}$ 

```

Algorithm 1 Overall Workflow of PAC-TREE

4.2 Stage 1: Join Key Selection

PAC-Tree adopts a greedy strategy to select optimal join keys as distribution keys. This selection is guided by a lightweight evaluation function, that estimates the benefit of a candidate key based on the size of the join result when using that key. For each table (represented as a hypernode), all candidate join keys (represented as subnodes) are ranked by this estimated benefit. The top-ranked keys are then chosen subject to a predefined replication factor.

```

1 Function getCandidateKeys( $M_i$ ):
2   return  $pk(M_i) \cup fk(M_i)$ 
3 Function getDegreeFeature( $c_i, c_j$ ):
4   join frequency ( $f \leftarrow 0$ ), scan selectivity ( $s \leftarrow 0$ )
5   if  $\max(\text{mark}(c_i), \text{mark}(c_j)) \leq \frac{1}{3}$  then
6     return (0, 0)
7   foreach  $q_k \in Q$  do
8     if  $c_i, c_j \in q_k \wedge c_i \bowtie c_j$  then
9        $f \leftarrow f + 1$ 
10       $s \leftarrow s + q_k^{sel}(c_j)$ 
11   return ( $f, s/|Q|$ )
12  $F_{i,j} \leftarrow \forall c_i \in M_i \forall c_j \in M_j \text{ getDegreeFeature}(c_i, c_j)$ 
13 foreach  $M_i \in \mathbf{M}$  do
14   Initialize weight map  $w[\cdot] \leftarrow 0$ 
15   foreach  $c_i^0 \in \text{getCandidateKeys}(M_i)$  do
16     foreach  $c_j^1 \in \text{adj}[i]$  do
17        $w(c_i^0) \leftarrow w(c_i^0) + F_{i,j}[0] \cdot F_{i,j}[1] \cdot N_{tbl}(M_j)$ 
18    $K_i \leftarrow \text{TopK}(c_i, \text{key} = w(c_i), \text{limit} = N_{rep})$ 
19 return ( $K_1, \dots, K_n$ )

```

Algorithm 2 Join-Key Selection for Each Table

Algorithm 2 details the procedure. The significance of each candidate key is quantified by the degree of its corresponding subnode in the hypergraph (lines 3–11), which combines two metrics: its participation frequency in join operations and its average scan selectivity. Both metrics are typically available from query optimizer statistics extracted from historical logs. To filter out uninformative candidates, we discard columns whose mark value falls below a threshold θ , setting their degrees to zero. The mark value for a column c is defined as the ratio between its number of distinct values (NDV) in a sampled partition block and the total number of entries in that block: $\text{mark}(c) = \frac{\text{NDV}(c)}{\text{count}(R_{\text{sample}})}$. The threshold θ addresses the issue of low-cardinality columns that cause data skew: only a few compute nodes handle a disproportionate fraction of the workload while others remain largely idle. In practice, we found that effective values of θ fall within the range of (0.1, 0.5), which balances two competing factors: (i) very low thresholds ($\theta < 0.1$) tend to admit low-cardinality columns, leading to severe skew; (ii) very high thresholds ($\theta > 0.5$) may discard useful keys and reduce join opportunities. Empirically, our implementation adopts $\frac{1}{3}$ as a representative compromise.

Following this, each subnode is assigned a weight, defined as the product of its potential join result size for that key and its degree (lines 13–17). Subnodes within a hypernode are then ordered in descending weight, and the highest-ranked ones that satisfy the replication factor constraint are selected as the final join keys (line 18).

4.3 Stage 2: Top-Part Tree Construction

In the second stage, we construct the upper hierarchy of the PAC-tree based on chosen join keys, which is responsible for data redistribution across shards. The root node is first split along the chosen key into child nodes with uniform ranges. To fully exploit parallelism among workers, the number of top-level leaf nodes is chosen as an integer multiple of the available worker count (N_{worker}), ensuring balanced utilization during join processing. Specifically, we draw a random sample from the base table, sort it on the selected join key, and split it into N_x temporary partitions. Each partition functions as a leaf node, with boundaries defined by the column-wise minimum and maximum values. The height of the top-part tree, denoted by h , is the smallest integer satisfying $2^h \geq N_x$.

A critical step is the determination of N_x . A natural heuristic is setting $N_x = \kappa N_{\text{worker}}$, where κ is a small tunable integer factor. The parameter κ controls two competing aspects: (i) sharding granularity, which affects the uniformity of data distribution across machines, and (ii) the flexibility of the bottom-part tree, where a smaller κ can avoid imposing restrictions on the performance potential of the cluster keys. We evaluate $\kappa \in \{1, 2, 3\}$ using historical workloads by comparing two measures of data-skipping efficacy: the join-filter cost C_{join} and the scan-filter cost C_{scan} . The configuration that minimizes the total cost is then selected. Formally, we solve

$$\begin{aligned}
 & \underset{\kappa}{\text{maximize}} \quad -(C_{\text{join}} + C_{\text{scan}}) \\
 & \text{subject to} \quad \begin{cases} \kappa \in \mathbb{Z}^+, \\ 1 \leq \kappa \leq 3, \\ N_{\text{tbl}} (\kappa N_{\text{worker}})^{-1} \geq b_{\min}, \end{cases} \quad (2)
 \end{aligned}$$

The chosen κ must ensure that partitions satisfy the minimum block size constraint, where $b_{\min}$ denotes the smallest block size permitted by the storage system. Let k_i denote the finalized join key and T_0 the constructed index tree. We approximate the join-filter cost C_{join} of a query q as the memory overhead of evenly distributing intermediate results across workers (excluding CPU cost). Formally,

$$C_{\text{join}} = \sum_{q \in Q} \sum_{\substack{k_j \in \text{adj}[i] \wedge \\ k_i, k_j \in q}} \times \left\lceil \frac{2 \sum_{R \in \mathcal{R}_{\text{acc}}(q, T_0)} \text{Size}(R)}{N_{\text{worker}}} \right\rceil \times N_{\text{worker}}, \quad (3)$$

where $\mathcal{R}_{\text{acc}}(q, T_0)$ is the set of leaf partitions of T_0 accessed by q . Similarly, we estimate the scan-filter cost C_{scan} as

$$C_{\text{scan}} = \sum_{q \in Q} \left(|\mathcal{R}_{\text{acc}}(q, T_0)| - 1 \right) \times \frac{\sum_{R \in \mathcal{R}_{\text{acc}}(q, T_0)} \text{Size}(R)}{|\mathcal{R}_{\text{acc}}(q, T_0)|}, \quad (4)$$

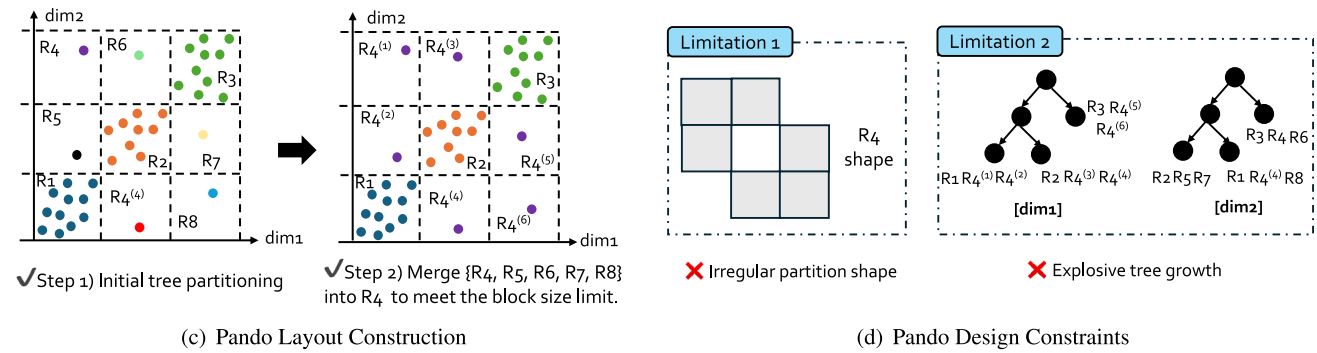


Fig. 5 We do not adopt the Pando [38] design for constructing the bottom-part of PAC-Tree due to its high management overhead, which would increase the complexity of the hierarchical PAC-Tree and hinder system integration and maintenance

Table 2 Comparison of MTO and Pando logical tree structures based on the TPC-H benchmark

	#Table	#Trees	#Avg Leaf Nodes	% of logical partitions(merged to satisfy $[b_{min}, 2b_{min}]$)
MTO	8	8	83	0
Pando	8	32	97906	96%

which captures the extra I/O caused when query predicates require further subdivision of tree nodes.

4.4 Stage 3: Bottom-Part Tree Construction

To determine suitable cluster keys, various mature query-driven techniques have been proposed to maximize data skipping by exploiting frequently filtered columns. PAC-tree adopts these techniques to construct its bottom layer.

Partitioning Tree selection. MTO [17] and Pando [38] are among the most effective approaches. MTO extends QD-Tree by propagating join-related predicates across connected tables [34], which is especially useful when join predicates are implicit but can be inferred from schema relationships. By contrast, Pando employs a multi-tree-per-table design to mitigate data skew (Fig. 5), achieving higher data skipping than MTO's one-tree-per-table design. However, Pando faces two fundamental drawbacks: (1) It produces highly irregular partition shapes, complicating metadata management in storage engines that typically rely on lightweight file headers. Moreover, when new data accumulates in R_4 (initially a single block), redistributing the data evenly across new blocks is difficult. (2) Each dimension requires maintaining a separate tree, so wide tables with hundreds of dimensions may require constructing and maintaining hundreds of index trees. As shown in Table 2, Pando generates more than three orders of magnitude more leaf nodes than MTO, with 96% subsequently merged into irregular partitions, resulting in significant management overhead. From a system design perspective, we argue that manageability

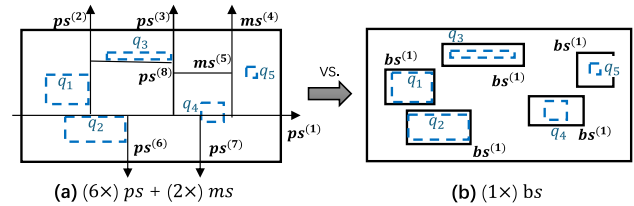


Fig. 6 Bounding split

must take precedence over query-level optimization, and therefore adopt MTO as the basis for PAC-Tree's bottom-layer construction.

Bottom-Part Tree. Following the split policy defined by the MTO-tree, we extend the top-part tree by generating sufficient candidate splits for the leaf nodes. Specifically, we extract: (i) all filter predicates $S^P = \{ps_1, \dots, ps_n\}$ from the WHERE clause, (ii) data-induced join conditions $S^J = \{js_1, \dots, js_n\}$ via equi-join information passing, and (iii) a set of median values $S^M = \{ms_1, \dots, ms_n\}$ from all relevant columns. These are combined into form the overall candidate split set $S_s = S^P \cup S^J \cup S^M$. At each split, we choose the condition from S_s that maximizes the split benefit for the current query data skipping, denoted as ξ , and use it to partition the current node, recursively generating new subnodes until no further splits are possible.

Split Policy Optimization. MTO employs a single split condition (js , ps or ms) at each step. However, repeated single-step splits on nodes that will become final leaf nodes can be merged. In PAC-Tree, this merged multi-step operation is called a bounding split (bs). Formally, we define a single-step tree split as c_{sps} , multiple single-step tree splits as c_{mps} , and the merged multi-step bounding tree split as c_{bs} .

Figure 6 illustrates the difference between these two types of splits applied to a rectangular data area maintained by an internal node. The regions $q_1 \dots q_8$ represent query ranges. In Fig. 6(a), the leaf nodes are constructed using eight single-step splits, i.e., $c_{mps} = (6ps, 2ms)$. In contrast, Fig. 6(b) employs a single bounding split. Here, the superscript indicates the order of execution. The bounding

split first determines the ideal boundary for each query and then proceeds as follows: (1) If the query region is smaller than the minimum block size b_{min} , its partition boundaries are symmetrically expanded along all dimensions until the size constraint is satisfied. (2) When expanded partition boundaries overlap or the existing boundaries exceed $2b_{min}$, they are merged and subsequently divided evenly into two new partitions.

Compared with c_{mps} , c_{bs} favors parallel tree extension and avoids creating excessively large partitions. Notably, while c_{sps} may occur at any internal node, c_{bs} only produces leaf nodes.

Theorem 2 (Guidance for Favoring Multi-Step Splits) *Generally, there is a progressive relationship between c_{sps} and c_{mps} , i.e., $\xi(c_{mps}) \gg \xi(c_{sps})$, because the cumulative benefit of a multi-step split sequence far exceeds that of any individual single-step split within it. This is due to the fact that finer-grained partitions can accommodate a broader range of query filter patterns. Consequently, if $\xi(c_{sps}) > \xi(c_{bs})$, then it follows that $\xi(c_{mps}) \gg \xi(c_{bs})$.*

Theorem 3 (Preference for Bounding Splits in Small Node Contexts) *The smaller the leaf node size (n_{size}) considered for a split, the more advantageous c_{bs} becomes relative to c_{sps} . A special case is when $n_{size} = 2b_{min}$, c_{bs} still finds the optimal boundary for each query, but c_{sps} will no longer split because the size limit has been reached, so $\xi(c_{bs}) \geq \xi(c_{sps})$ must hold. As n_{size} grows larger, query distributions become denser, and fewer queries meet the boundary-expansion condition of c_{bs} ; thus, the advantage of c_{bs} gradually decreases. We therefore idealize that c_{bs} outperforms c_{sps} with probability $\frac{X_{ct}}{n_{size}}$ (i.e., when $n_{size} \leq X_{ct}$), where X_{ct} is a critical size. When n_{size} fluctuates around or below X_{ct} , then $\xi(c_{bs}) \geq \xi(c_{sps})$.*

We show under which conditions the bounding split yields greater benefits, thereby guiding the optimal split scheme for any node. When splitting a node, one should fairly compare $\xi(c_{mps})$ with $\xi(c_{bs})$. However, since c_{mps} is derived from multiple expansions of c_{sps} , computing $\xi(c_{mps})$ directly is time-consuming. Therefore, we rely on an estimate: (1) Based on Theorem 2, if $\xi(c_{sps}) > \xi(c_{bs})$, then it follows that $\xi(c_{mps}) \gg \xi(c_{bs})$, and thus we need only select the optimal c_{sps} at the current split. (2) On the other hand, if $\xi(c_{sps}) < \xi(c_{bs})$, Theorem 3 suggests that, for smaller nodes ($n_{size} \leq n_{size}^{(bs)}$), c_{bs} should be chosen. We found that configuring $n_{size}^{(bs)}$ as a small multiple of b_{min} , typically $6b_{min}$ across a comprehensive set of benchmarks, consistently yields effective performance. Within this range, query distributions tend to be smaller and more scattered, which facilitates splitting into leaf nodes.

5 Query Plan Optimization with Hyper Join

Once PAC-Trees are defined, they guide the instantiation of data into a specific physical layout. Attention then shifts to the challenge of minimizing join costs within this established layout. This section tackles this problem systematically, beginning by defining the optimal hyper join problem for pairs of relations and subsequently extending the framework to multi-table joins.

5.1 The Optimal Hyper Join Problem

Traditional shuffle joins typically incur extensive data movement overhead when joining distributed tables, particularly when join-participating data is scattered across the cluster. In such cases, blocks from participating tables are typically reshuffled across machines. In contrast, a hyper join aims to significantly reduce this data transfer overhead compared to a full shuffle join. Its efficiency is maximized when the data layout ensures that blocks from one table need to access only a small subset of blocks from the other table on the same machine.

Consider two tables, M_1 and M_2 . Let $S_{M_1} = \text{lookup}(T_1, q) = \{R_1, \dots, R_n\}$ denote the set of n blocks retrieved from the partition tree (T_1) of M_1 for query q 's join operation. The central objective is to determine an optimal split of S_{M_1} into co-partitions, denoted $\mathcal{P}_{co}$, such that the total cost of accessing blocks is minimized. For each co-partition $P_k \in \mathcal{P}_{co}$ and each block $R \in P_k$, it is necessary to ensure that all data blocks overlapping from M_2 with R are also included in P_k . Furthermore, practical systems impose memory constraints. Assume each worker node can simultaneously hold at most B_{max} blocks in memory. At least $c = \lceil \frac{n}{B_{max}} \rceil$ co-partitions are required, corresponding to c hash tables. This limited memory characteristic allows us to formally define the optimal hyper join problem.

Definition 2 (Optimal Hyper Join) Let $J_{M_2}^{M_1}(x)$ denote a lookup function that returns the set of blocks from relation M_2 that join with block x from relation M_1 . The optimal hyper join between tables M_1 and M_2 is defined as a set of co-partitions $\mathcal{P}_{co}^*$ that minimizes the total number of blocks read from M_2 .

$$\mathcal{P}_{co}^* = \underset{\mathcal{P}_{co}}{\operatorname{argmin}} \sum_{P_k \in \mathcal{P}_{co}} \left| \bigcup_{R_i \in P_k} J_{M_2}^{M_1}(R_i) \right| \quad (5)$$

$$s.t. \begin{cases} |\mathcal{P}_{co}| \geq c, \\ \forall P_k \in \mathcal{P}_{co}, |P_k| \leq B_{max} \end{cases}$$

5.2 Queue-Driven Grouping (QDG) for Hyper Join

Prior work, such as AdaptDB [6], has explored a bottom-up co-partitioning policy for the hyper join problem. The approach strictly follows the default block placement order by sequentially selecting the next unused block and greedily grouping it with other blocks to form a co-partition. However, reliance on a fixed initial order can lead to suboptimal groupings. To address this, we introduce a queue-driven grouping algorithm, termed *QDG*. *QDG* repeatedly merges pairs of blocks or block groups that maximize the locality of their required join blocks from M_2 . The algorithm consists of two stages:

```

1 for  $i \leftarrow 1$  to  $|S_{M_1}|$  do
2    $I_{max} \leftarrow -1, \chi_{partner} \leftarrow [];$ 
3   for  $j \leftarrow i$  to  $|S_{M_1}|$  do
4      $I_{cur} \leftarrow \text{count}(J_{M_2}^{M_1}(R_i) \cap J_{M_2}^{M_1}(R_j));$ 
5      $I_{record}[i][j] \leftarrow I_{cur}, I_{record}[j][i] \leftarrow I_{cur};$ 
6     if  $I_{cur} > I_{max}$  then
7        $I_{max} \leftarrow I_{cur}, \chi_{partner} \leftarrow (j);$ 
8   if  $I_{max} = 0$  then
9     Append  $i$  to  $D_{ids}$ ;
10  else
11    Push  $[(i), \chi_{partner}, I_{max}]$  into  $PQ$ ;
12 return  $PQ, I_{record}, D_{ids};$ 

```

Algorithm 3 QDG-Stage 1: Generate Priority Queue

(1) **Stage 1: Priority Queue Generation.** This stage constructs a priority queue (PQ) that guides the subsequent grouping (Algorithm 3). Each element χ dynamically maintains three attributes:

- A block group $S_g \subset S_{M_1}$, where initially each block R_i belongs to a individual group ($S_g = \{R_i\}$).
- An identifier of its best merge partner, $\chi_{partner}$, which points to the element in PQ (either a block or a block group) that yields the maximum intersection value with S_g .
- The maximum intersection value between itself and $\chi_{partner}$: $I_{max}(S_g(\chi), S_g(\chi_{partner})) = |\left(\bigcup_{R \in \chi} J_{M_2}^{M_1}(R)\right) \cap \left(\bigcup_{R' \in \chi_{partner}} J_{M_2}^{M_1}(R')\right)|$.

Here, I_{max} measures the total size of the overlapping set of required M_2 joining blocks between the given two elements in PQ . PQ is ordered by I_{max} (descending), and by group size $|S_g|$ for tie-breaking. During its construction, two auxiliary structures are created: (i) I_{record} , storing all pre-computed intersections between any two blocks from S_{M_1} (lines

1–7), and (ii) D_{ids} , recording block identifiers with zero intersection values (lines 8–9).

(2) **Stage 2: Table Co-Partitioning.** This stage performs co-partitioning of M_1 and M_2 (Algorithm 4). The process is guided by three rules. In particular, *Rule₁* and *Rule₃* focus on improving merging efficiency at each iteration, while *Rule₂* governs the dynamic maintenance of intersection values (I_{max}) for relevant elements within the priority queue.

- *Rule₁*: Blocks in S_{M_1} that do not intersect with any other blocks are independent with respect to the final co-partitioning cost ($\mathcal{P}_{co}$). Consequently, they can be directly assigned to co-partitions without affecting the overall objective.
- *Rule₂*: When a new merged group S_g^* is formed, the associated maximum intersection values (I_{max}) of potentially affected elements in PQ must be recalculated to identify updated merging partners. To reduce overhead, these updates can be safely skipped for elements that no longer have sufficient remaining capacity to merge with S_g^* .
- *Rule₃*: To accelerate convergence, when multiple elements in PQ exhibit the same maximum intersection value, priority is given to the element with the largest group size. This heuristic ensures faster reduction of the queue while still preserving the quality of merges.

The algorithm initializes by partitioning D_{ids} into multiple groups according to *Rule₁*, under B_{max} constraints (line 1). After this initialization, the remaining block groups are progressively merged through a multi-round process managed by the priority queue PQ (lines 3–26). In each round, the top element (with the highest I_{max}) is merged with its identified partner, $\chi_{partner}$, forming a new and larger group S_g^* . If S_g^* either reaches the capacity limit B_{max} or if PQ contains only one element, then S_g^* is finalized as a complete co-partition and appended to the result set $\mathcal{P}_{co}$ (lines 11–13). Following *Rule₂*, the $\chi_{partner}$ and associated I_{max} values for elements related to $\bar{\chi}$ are dynamically updated. This recomputation involves each potential element pair (lines 17–26), which can leverage the pre-computed intersection values in I_{record} to reduce computational overhead (lines 17–26, particularly 20–21).

Overall, the *QDG* algorithm ensures that at each step, the pair of elements offering the greatest merging benefit is chosen for co-partitioning. This local optimality underpins the algorithm's performance guarantees and contributes to its practical efficiency.

```

1  $\mathcal{P}_{co} \leftarrow \{S_g^* : S_g^* \in D_{ids}.splitBy(B_{max})\}$ 
2  $n \leftarrow n - |\mathcal{P}_{co}|$ 
3 while  $n > 0 \wedge \bar{\chi} \leftarrow PQ.pop(0)$  do
4   // Step-1: Merge candidate with the highest benefit.
    $\bar{\chi} \leftarrow (\bar{\chi}[0] \cup \bar{\chi}[1], \bar{\chi}[1], -\infty)$ 
5   // Step-2: Delete any conflicting elements from  $PQ$ 
   foreach  $\chi \in PQ$  do
6     if  $\chi[0] \cap \bar{\chi}[0] \neq \emptyset$  then
7       Remove  $\chi$  from  $PQ$ 
8     else
9       if  $\chi[1] \cap \bar{\chi}[0] \neq \emptyset$  then
10        Update  $\chi$ :  $\chi[1] \leftarrow [], \chi[2] \leftarrow -\infty$ 
11   if  $|\bar{\chi}[0]| = B_{max}$  or  $|PQ| = 1$  then
12     New group:  $S_g^* \leftarrow \bar{\chi}, S_g^* \leftarrow PQ[0]$ 
13     Add  $S_g^*[0]$  to  $\mathcal{P}_{co}$ , Update  $n \leftarrow n - |S_g^*[0]|$ 
14   else
15     // Step-3: Recalculate intersection values.
16     Push  $\bar{\chi}$  back into  $PQ$ 
17     foreach  $\chi_1 \in PQ$  with  $\chi_1[2] = -\infty$  do
18       Set  $I_{max} \leftarrow -\infty$  and  $\chi_{partner} \leftarrow []$ 
19       foreach  $\chi_2 \in PQ$  do
20         if  $|\chi_1[0]| = 1$  and  $|\chi_2[0]| = 1$  then
21            $I_{cur} \leftarrow I_{record}[\chi_1[0]][\chi_2[0]]$ 
22         else
23            $I_{cur} \leftarrow \left| \bigcup_{R_i \in \chi_1[0]} J_{M_2}^{M_1}(R_i) \cap \bigcup_{R_j \in \chi_2[0]} J_{M_2}^{M_1}(R_j) \right|$ 
24         if  $I_{cur} > I_{max}$  then
25            $I_{max} \leftarrow I_{cur}, \chi_{partner} \leftarrow \chi_2[0]$ 
26          $\chi_1[1] \leftarrow \chi_{partner}, \chi_1[2] \leftarrow I_{max}$ 
27 return  $\mathcal{P}_{co}$ 

```

Algorithm 4 QDG-Stage 2: Co-partition Blocks over Tables M_1 and M_2

Time Complexity Analysis. Efficient execution is essential since co-partitioning is only one component of the overall query plan. Compared with a baseline bottom-up strategy with $O(n^2)$ complexity, the proposed QDG incurs $O(n^2)$ in Stage 1 and $O(\mathbb{K}n^2)$ in Stage 2, where n denotes the number of initial block groups and $\mathbb{K}$ is the average number of queue elements whose intersection values (I_{max}) must be recomputed after each merge. We formally bound $\mathbb{K}$ within $[1, \rho \frac{n+1}{2} + 1]$, where $\rho \in [0, 1]$ is the disorder coefficient capturing the misalignment between partitions on the two join sides. Intuitively, $\rho = 1$ corresponds to a fully randomized shuffle join, in which all elements must be updated, then thus the overall complexity increases to $O(\frac{n^2(n+1)}{2}) \approx O(n^3)$. In contrast, $\rho = 0$ indicates near-perfect alignment of join keys, reducing the complexity to $O(n^2)$. Since QDG operates on the PAC-Tree layout, both sides are pre-partitioned according to identical or proportionally scaled join-key ranges, keeping ρ very small (i.e., $\rho \rightarrow 0$), which makes $\mathbb{K}$ a constant approaching its lower-bound. Furthermore, with

bounded compute-node memory ($n \leq \frac{B_{max}}{2b_{min}}$), the value of $\mathbb{K}$ is strictly constrained. Consequently, the additional runtime overhead remains minimal in practice.

5.3 Hyper Joins Over Multiple Relations

This section extends the hyper join methodology to multi-table join operations. A central challenge is that hyper joins cannot be directly applied to adjacent join operators, which often necessitates shuffling tables when joining more than two relations, thereby disrupting partition alignment. For instance, consider the join sequence $((M_A \bowtie_{c_{ab}} M_B) \bowtie_{c_{bc}} M_C) \bowtie_{c_{cd}} M_D$, with join keys c_{ab}, c_{bc}, c_{cd} . A hyper join between M_A and M_B on c_{ab} produces an intermediate result (*tempAB*). To continue, *tempAB* and M_C must be repartitioned on c_{bc} , typically requiring a shuffle join. Similarly, joining *tempABC* with M_D on c_{cd} requires another shuffle join. In this plan, one hyper join and two shuffle joins are needed. Alternatively, rearranging the join order as $(M_A \bowtie M_B) \bowtie (M_C \bowtie M_D)$ allows hyper joins to be applied independently to (M_A, M_B) and (M_C, M_D) , followed by a single shuffle join between *tempAB* and *tempCD*. This reordering significantly reduces the total I/O cost, compared with the previous plan. Thus, the core optimization problem is to determine an join order that maximizes the use of hyper joins while minimizing the number of required shuffle join operations.

Definition 3 (Join Graph) For a multi-table query, the join relationships are modeled as an undirected weighted graph $G_j = (V, E, w_V, w_E)$, termed the join graph. The vertices V correspond to the joining tables, each vertex $v \in V$ assigned a weight $w_V(v)$ representing the size of its joined blocks. Edges E represent potential join operations between table pairs. Each edge $e(u, v) \in E$ connects tables u and v and is associated with two potential execution branches: (i) a hyper-join edge (e_h) and (ii) a shuffle-join edge (e_s), with the branch choice determined by the edge's position in the join order.

Given a join graph G_j for a query q , let v_{M_1} and v_{M_2} denote the vertices corresponding to tables M_1 and M_2 . The vertex weights are defined as: $w(v_{M_1}) = \sum_{R_i \in S_{M_1}} \text{size}(R_i)$ and $w(v_{M_2}) = \sum_{R_i \in S_{M_2}} \text{size}(R_i)$. To compute edge weights, we adopt the empirical cost model proposed in [6] (see Equation 6), which quantifies the relative overhead of shuffle and hyper joins. Following their analysis, we set $\rho_{SJ} = 3$ and $\rho_{HJ} = 1.5$. Specifically, $\rho_{SJ} = 3$ indicates that shuffling a relation on average incurs about three times the baseline cost of accessing the same data, while $\rho_{HJ} = 1.5$ reflects the average number of additional disk reads during a hyper join. Note that in theory, for a fully co-partitioned case, ρ_{HJ} would be close to 1.

$$w(e) = \begin{cases} w(v_{M_1}) + w(v_{M_2}), & \text{if } \min\{w(v_{M_1}), w(v_{M_2})\} \leq B_{\max} \text{ and } e \in \{e_h, e_s\}, \\ \rho_{SJ} (w(v_{M_1}) + w(v_{M_2})), & \text{if } e = e_s, \\ \rho_{HJ} \min\{w(v_{M_1}), w(v_{M_2})\} + \max\{w(v_{M_1}), w(v_{M_2})\}, & \text{if } e = e_h. \end{cases} \quad (6)$$

The optimization goal is to identify a directed path that joins all tables with minimum total cost. Because consecutive hyper joins are disallowed on adjacent edges, pre-assigning a branch (i.e., assigning a join type) to the starting edge implicitly determines the join types of subsequent edges in G_j during path exploration. Conceptually, this problem is analogous to constructing a minimum spanning tree (MST). Inspired from PRIM [20]'s algorithm, we generate candidate paths for each chosen starting edge and then select the one with the lowest cost.

```

1 // Step-1: Identify candidate starting hyper edges.
2  $\bar{E} \leftarrow \{e_h(u, v) \mid (u, v) = \arg \max_{u, v \in V} (w(e_s(u, v)) - w(e_h(u, v)))\}$ 
3 foreach  $u \in V$  with  $|adj(u)| = 1$  do
4    $\bar{E} \leftarrow \bar{E} \cup \{e_h(u, adj(u)[0])\}$ 
5 // Step-2: Generate candidate paths.
6  $ans \leftarrow \emptyset$ 
7 foreach  $e_h(u, v) \in \bar{E}$  do
8   // label 1: hyper-join edge; label 0: shuffle-join
9    $S_{path} \leftarrow \{(u, v, 1)\}$ ,  $V_{used} \leftarrow \{(u, 1), (v, 1)\}$ 
10  Initialize  $V_{rest} \leftarrow V - \{u, v\}$ 
11  while  $V_{rest} \neq \emptyset$  do
12    Select  $e_{\min} = \arg \min_{e: e \text{ connects } V_{used}, V_{rest}} w(e)$ 
13    Update  $S_{path}, V_{used}, V_{rest}$  by including  $e_{\min}$ 
14     $ans \leftarrow ans \cup \{S_{path}\}$ 
15 // Step-3: Select the optimal join order.
16 return  $Join_{order} \leftarrow \arg \min_{path \in ans} \sum_{e \in S_{path}} w(e)$ 

```

Algorithm 5 Hyper-Join-Driven Join Reordering Policy

As shown in Algorithm 5, the procedure is as follows. First, candidate starting edges $\bar{E}$ are selected to be initially treated as hyper joins. These include (line 2) the edge offering the largest reduction compared with its shuffle alternative, and (lines 3–4) all edges adjacent to degree-one vertices. The former avoids expensive shuffle edges, while the latter broadens the search space of hyper edges. For each starting edge, we then generate an MST-like path using a greedy strategy (lines 7–13).

1. The starting edge $(u \bowtie v)$ is added to the candidate path S_{path} (line 8). The set V_{used} is dynamically updated and tracks available vertices for extending S_{path} via new edges. Label are used to mark edges: 1 for hyper join and 0 for shuffle join.
2. The algorithm iteratively selects the edge (cur, x) of connecting a vertex $cur \in V_{used}$ to a vertex $x \in V_{rest}$

that has the minimum weight (either shuffle or hyper cost) among all such candidate edges (lines 10–12). This edge is added to S_{path} and x is moved to V_{used} , ensuring the evolving path remains acyclic. The join type of (cur, x) is determined by the label of the preceding edge, enforcing the constraint that hyper edges cannot be adjacent.

This process repeats until all vertices are included in S_{path} . The final join order is the path with the minimum total weight among all candidates (line 15).

As an illustration, Figure 7(a) shows the join graph G_j for TPC-H Q5, with vertex weights based on pre-constructed partitioning. With $B_{max} = 4$, each edge is annotated with its potential join cost computed via Equation 6. The initial edge set is $\bar{E} = [(region \bowtie nation), (lineitem \bowtie orders)]$. Candidate join paths generated from these starting edges are shown in Figure 7(b) (c), using the Algorithm 5. The resulting candidate paths are $(lineitem \bowtie orders \bowtie customer \bowtie supplier \bowtie_{cycle} supplier \bowtie nation \bowtie region)$ and $(region \bowtie nation \bowtie supplier \bowtie lineitem \bowtie orders \bowtie customer \bowtie_{cycle} customer)$, with costs 1343 and 2761, respectively. Thus, the first path is selected as the optimal join sequence. Importantly, due to cycles in the original join graph, some equi-joins are treated as self-joins to ensure a valid ordering, with their join predicates potentially applied as filters on intermediate results. The cost of these self-joins is approximated by the number of scanned blocks in the intermediate result, typically determined by the maximum block count among participating tables.

6 Experiment

In this section, we present an experimental evaluation of PAC-Tree, addressing the following research questions (RQs):

- **RQ1:** How does PAC-Tree compare with baseline methods in terms of the amount of read I/O performed?
- **RQ2:** What is the computational overhead of finding optimized layouts and organizing data accordingly?
- **RQ3:** How do key hyperparameters (e.g., dataset characteristics, replication factor, block size) influence PAC-Tree's performance?

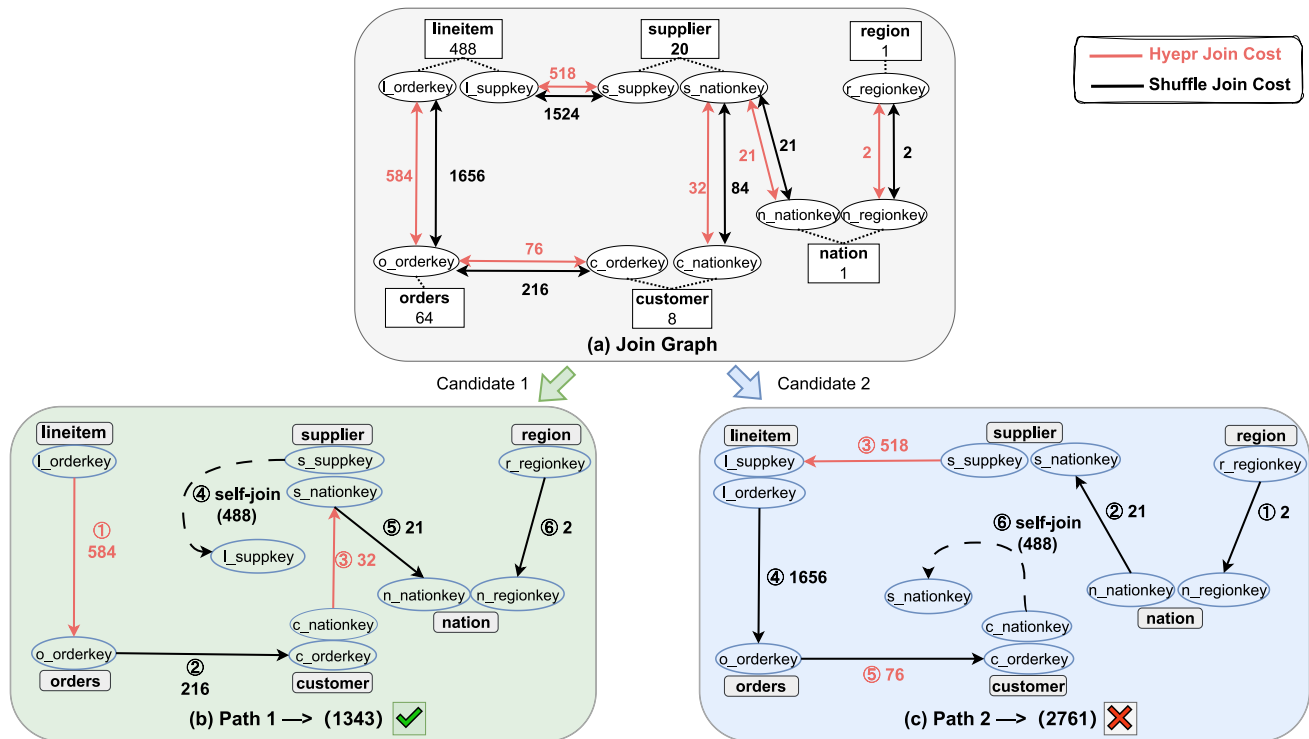


Fig. 7 (a) Join graph for TPC-H Q5, with node and edge weights estimated from pre-constructed join trees. (b-c) Two candidate join paths generated by Algorithm 5; the path with minimum cost is selected as the optimal join order

Table 3 Basic Information of Benchmarks (TPC-H, TPC-DS, JOB) and statistics for percentage of multi-queries and average joins/filters

Database	Scale	#Tables(#Fact #Dimension)	#Query Templates	#Multi-queries (%)	#Avg Joins (per multi-query)	#Avg Filters (per query)
TPC-H	max 100 GB	8 (1 7)	22	95.5% (21/22)	3.7	3.5
TPC-DS	max 98.2 GB	24 (6 18)	99	97% (96/99)	4.8	4.5
JOB	6.9 GB	16 (3 13)	33	100% (33/33)	5.4	4.7

- **RQ4:** How do individual components of PAC-Tree contribute to the overall query efficiency?
- **RQ5:** How does PAC-Tree perform in realistic system settings, in terms of query response time and resource utilization?

6.1 Experimental Setup

Evaluation Platform. All experiments are conducted on a 3-node Apache Spark (v3.2) cluster. Each machine node has a 16-core 2.4GHz Intel Xeon E5-2640 CPU, 32GB of RAM and a 252GB HDD, and runs Ubuntu 18.04 LTS. We use HDFS (from Apache Hadoop 3.3.1) as the distributed file system, and store all partitioned datasets in Parquet format. The Scala runtime ensures seamless integration with Spark SQL.

The PAC-Tree layout generator is implemented in PySpark, with a shallow integration into Spark's internal

catalog. Each leaf node is assigned a unique ID, allowing the system to identify the actual storage directory path of partitioned blocks and thereby supporting efficient data filtering during query execution.

Benchmarks and Datasets. We evaluate three standard OLAP workloads consisting of complex SQL queries (see Table 3). TPC-H and TPC-DS are executed at scale factors of 1, 10, 20, 50, and 100. Representative join test cases are constructed from selected tables with foreign keys. We also include the Join Order Benchmark (JOB) on the 6.9GB IMDB dataset. To adhere to the block size limitations while maintaining a stable overhead for partition tree construction across datasets of different scales, we vary sampling rates (defaulting to 1% for 100GB dataset).

Baseline Methods. We compare PAC-Tree with two alternatives. First, a baseline layout (PAW [18]) constructs its predicate-based binary tree but does not distinguish join-related columns from other attributes, nor consider primary/

foreign key relationships. Second, a hybrid of AdaptDB [6], BaW [11] and MTO [17] builds an instance-optimized layout (referred to as AD-MTO). The layout organizes data according to a standardized hierarchical tree backbone [6], where upper-layer nodes are split exclusively on selected join keys (following the principles of [11]), while lower-layer nodes adopt an extended QdTree-like approach that leverages complete query filters, including join-induced predicates [17]. Since these techniques have been developed by Amazon teams and integrated into the Redshift optimizer [39], their synthesis into a hybrid approach is a logical step for comparison.

Evaluation Metrics. We evaluate the layout generators primarily based on four aspects. (1) *Data Scan Ratio* is defined as the fraction of scanned blocks relative to dataset size (lower is better). (2) *Query Response Time* (also referred to as query latency) measures the entire query execution lifecycle from submission to result materialization, and serves as a key metric for system responsiveness to individual user requests. (3) *System Throughput* quantifies the system's aggregate data processing capabilities by capturing the rate of query processing per unit time (higher is better). (4) *Model Runtime* indicates the execution time to generate the partition tree (both top-part and bottom-part phases); lower values reflect higher model efficiency.

6.2 Results (RQ1 and RQ2)

Query Performance. To answer RQ1, we compare PAC-Tree with PAW and AD-MTO (the SOTA baseline). Fig. 8 illustrates that PAC-Tree achieves strong cross-database generalization. Since directly measuring query latency is influenced by disk read rate, worker memory and buffer caching, we use scanned data volume here as a robust metric. Across various workloads with table schemas from diverse application datasets, PAC-Tree achieves up to a 67% reduction in scanned data compared to PAW and up to a 45% reduction compared to AD-MTO.

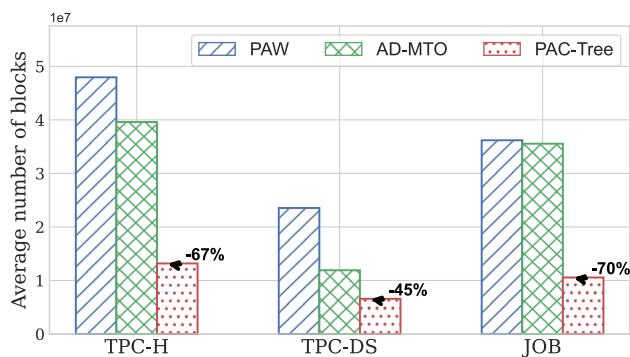


Fig. 8 Average scanned blocks per query across the evaluated benchmarks. We listed the percentage reduction over the AD-MTO layout

Fig. 9(a)–9(c) show detailed results for TPC-H, TPC-DS, and JOB. PAC-Tree is not always optimal for each individual query. In particular, suboptimal outcomes are observed for TPC-H Q1, TPC-DS Q4, Q6, Q9, Q11, as well as JOB Q5 and Q10. This can be attributed to two factors. First, some tables feature a larger number of join keys, and chosen co-partitioning keys cannot fully capture all join conditions, causing more shuffling. Second, the MST-based join ordering used by PAC-Tree may misjudge the cost of shuffle edges, as these costs are only accurately known during execution. This misestimation can lead to overvaluing certain hyperedges, which in turn restricts join orders and raises overall join costs.

Nevertheless, PAC-Tree is still able to optimize the majority of queries and delivers substantial cumulative improvements. As shown in Fig. 10, reduction in data scan ratio steadily increases with more queries, following a smooth upward trend compared to other advanced methods. Gains are most pronounced and consistent in TPC-H and JOB (fewer tables, more stable join keys), while TPC-DS shows slightly smaller improvements due to its more complex schema.

Where Does the Advantage Come From? In these benchmarks, fact tables typically contain 2–3 join keys plus selection filters (e.g., LIKE, IN, and BETWEEN AND) on various columns. PAC-Tree can easily exploit these features by (i) using the lower part of the tree for early pruning of irrelevant joined blocks, and (ii) using the upper part of the tree for join-key alignment under fixed storage constraints, often proving more impactful for overall query performance. Taking TPC-H Q3 as an example, PAC-Tree might not strictly minimize joined blocks between *lineitem* and *orders* during the scanning phase. Instead, in the subsequent hash join phase involving *lineitem* $\bowtie$ *orders* $\bowtie$ *customer*, it reduces intermediate data read by approximately 63% compared to alternatives. This requires only two hyper joins and two shuffle joins. By contrast, AD-MTO also aiming for similar efficiency on the default join tree, but it lacks specialized hyper-join optimizations for operations like *lineitem.l_orderkey* $\bowtie$ *orders.o_orderkey* and *orders.o_custkey* $\bowtie$ *customer.c_custkey*, incurring five shuffle operations.

Model Efficiency Analysis. Instance-optimized data layouts are valuable only if they can be efficiently identified and organized. To answer RQ2, we analyze offline model runtime and data routing time for PAC-Tree and comparative approaches. As shown in Fig. 11, "data routing time" measures the time required to locally partition the full dataset via join trees (i.e., assigning tuples to blocks). Both times generally grow with the number of candidate join trees: AD-MTO has the shortest average model runtime (33s), while PAC-Tree requires the longest (81s). However,

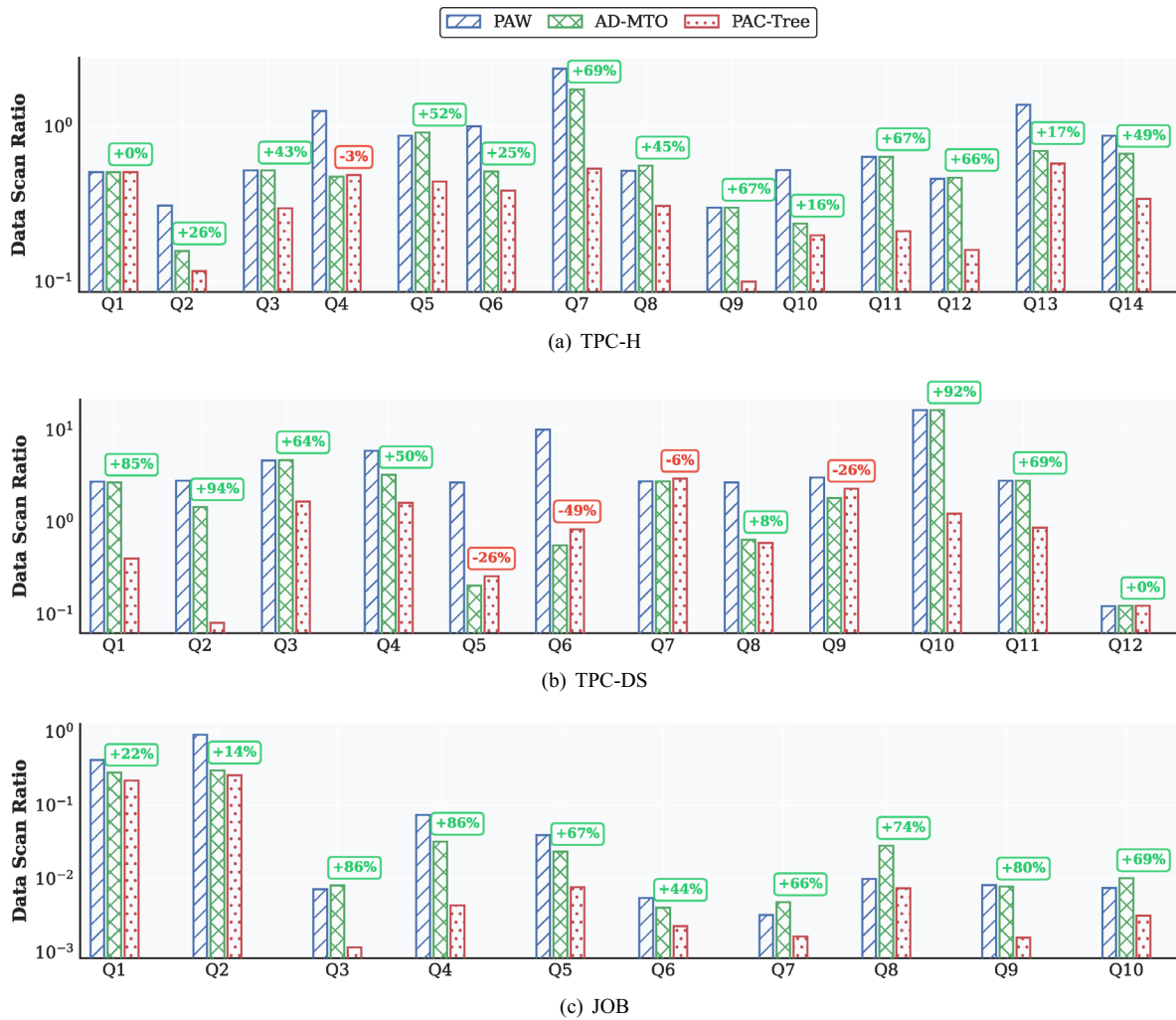


Fig. 9 Comparison of data scanning metrics observed during query execution on different data layouts. Results are shown for a selection of representative queries from each benchmark

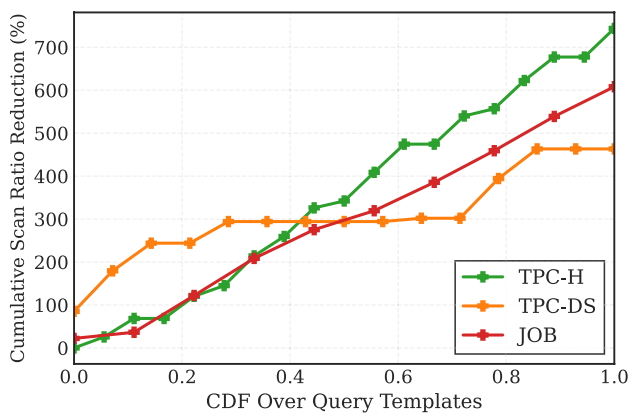


Fig. 10 Cumulative reduction in data scan ratio achieved by PAC-Tree relative to AD-MTO across query templates

this 48 s gap is negligible relative to the overhead for processing all benchmark datasets (model runtime plus data routing, approximately 3.8h). After local partitioning, each

partition is uploaded to HDFS. Since the total data volume to route remains constant, all schemes with the same minimum block size take roughly equivalent upload time; thus, we exclude this metric from our analysis.

6.3 Sensitivity Analysis (RQ3)

To further evaluate the PAC-Tree's reliability (RQ3), we observe fluctuations in its key metrics by altering the contextual environment parameters of the partitioning process.

Optimal Number of Join Tree Replicas. PAC-Tree constructs multiple replicas for each table to accommodate diverse join keys, thereby improving block alignment. The upper bound on data replicas directly influences the likelihood of hyper join formation. As shown in Fig.12(top), increasing the replication factor (r_{max}) generally improves performance up to an optimal point (around $r_{max} = 3$ in our workloads). Thus, selecting an appropriate r_{max} is

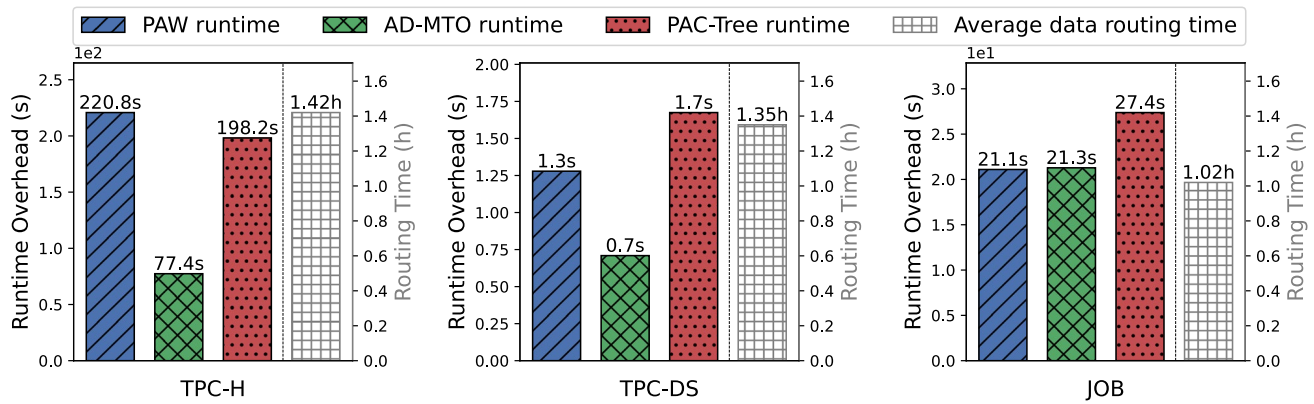
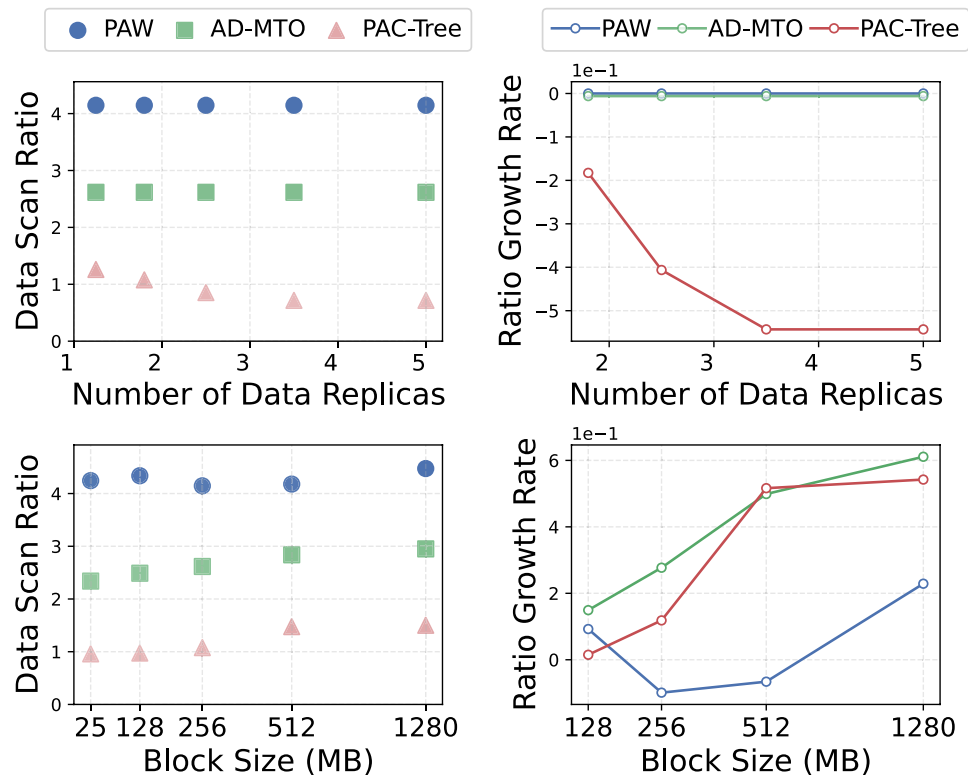


Fig. 11 Offline time overhead for layout generation and subsequent data routing

Fig. 12 Variation in PAC-TREE's data scan ratio as a function of replicate factors and block sizes



critical for maximizing PAC-Tree's layout efficiency across scenarios. For simplicity, r_{max} can be set to the system's default replication factor recommendation.

Partition Size Impact. The logical partition size is typically a multiple of the block size. Figure 12(left) shows that PAC-Tree maintains its performance advantage over the baseline across various minimum block sizes (25MB, 128MB, 256MB, and 1280MB). Figure 12(right) reveals that both hierarchical trees (MTO and PAC-Tree) are sensitive to this parameter. Larger partitions reduce the flexibility in distinguishing data groups by query access patterns, thereby gradually diminishing partitioning effectiveness.

From an algorithmic perspective, the total data read from partitions is determined by both the block count and redundant reads, with the latter largely influenced by the initial block distribution across machines.

Impact of Number of Joined Tables. Figure 13 evaluates the effect of varying the number of joined tables (1, 2, 3, 4) using TPC-DS queries with a replication factor of 2 and minimum block size of 64MB, we can observe that as join complexity increases, the performance gap of all layouts widens for both single-table and multi-table workloads. While PAC-Tree demonstrates comparable performance to AD-MTO with fewer joined tables, its advantages grow

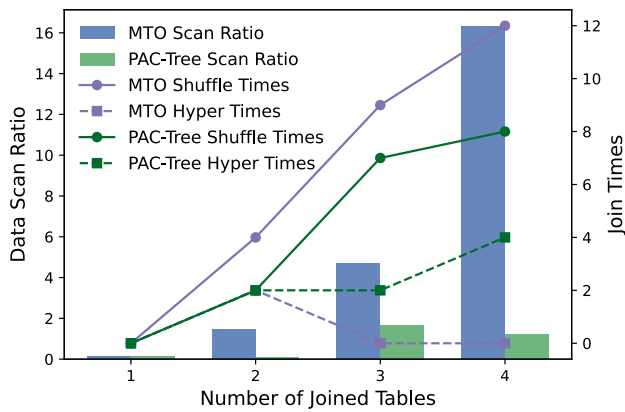


Fig. 13 Evaluation of performance metrics when varying the number of tables participating in join operations

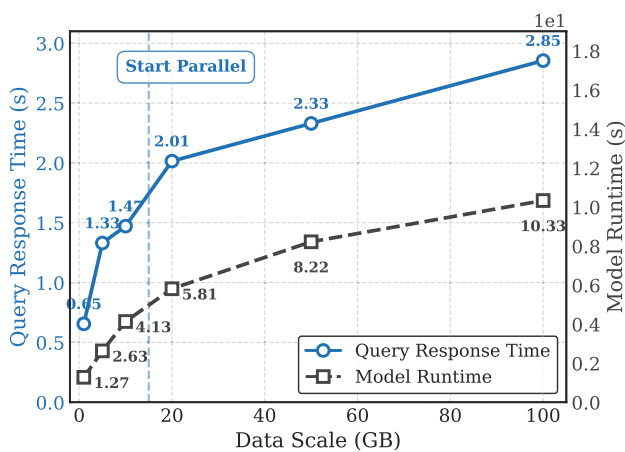


Fig. 14 Sensitivity analysis of PAC-TREE performance with respect to variations in table data scale

significantly as joins become more complex, highlighting broader applicability. The figure also shows hyper/shuffle join times during multi-join phases: reduced data shuffling and increased hyper join usage drive PAC-Tree's stable performance. Hyper joins are typically executed first, producing smaller intermediate result sets before shuffling to minimize read amplification.

Varying Data Scale. To analyze the effect of dataset size, we vary the TPC-H scale factor from 1 to 100 using all 22 query templates. As shown in Fig. 14, query response time grows with data size, while offline model runtime remains relatively stable. Large datasets increase processing requirements proportionally, resulting in higher query latency. However, after enabling parallel threads, the processing load is distributed across threads, leading to a more stable and controllable growth in query latency. In contrast, model runtime stays stable because the sampling ratio is pre-adjusted to keep that sampled data within a fixed threshold, thereby bounding overhead regardless of dataset size.

Table 4 Ablation study on PAC-Tree components and strategies

Configuration	TPC-H	JOB	TPC-DS
Join Key Selection Strategy			
- Random ($r = 1$)	0.536	0.060	1.296
- Weight-based ($r = 1$)	0.536	0.052	1.260
- Weight-based ($r = 2$)	0.340	0.049	1.070
Bounding Node Split			
- Disable	0.158	0.035	0.254
- Enable ($n_{size} \leq 6b_{min}$)	0.149	0.029	0.254
Join Reorder Strategy			
- Default Order	0.560	0.068	2.810
- QDG+PRIM-based Reorder	0.340	0.049	1.070

Values indicate the data scan ratio (lower is better), and the best results are highlighted in bold

6.4 Ablation Studies (RQ4)

This section presents extensive ablation studies to evaluate the effectiveness of individual components of our proposed approach, addressing RQ4.

Ablations on Join Key Selection. To overcome the limitations of fixed join keys, we introduce a weight-based key selection strategy. Next, we validate its effectiveness against alternative approaches, with results reported in Table 4. In the random setting, each table is assigned one arbitrary join key. By contrast, our strategy selects up to two recommended join keys (i.e., $r_{max} \leq 2$). We observe that smaller r_{max} values and randomly chosen keys reduce effective co-partitioning opportunities, increase shuffle overhead, and ultimately degrade query performance. Consequently, under our selection strategy, employing the largest feasible replication factor offers the greatest potential for performance gains.

Ablations on Bounding Node Split. To evaluate the bounding split (bs) optimization for small nodes, we compare scan costs for single-table operators with and without bs enabled. In TPC-DS, the presence of complex table and query patterns often leads to effective multi-step split conditions even without bs . In contrast, bs proves its effectiveness in TPC-H and JOB (yielding 5.6% and 17% improvements). This indicates that bs identifies optimal split paths within a constrained search space, effectively replacing a series of sub-optimal single-step splits. We also found that the benefits are concentrated in a few tables, delivering substantial data skipping gains, particularly for nodes with dispersed query accesses.

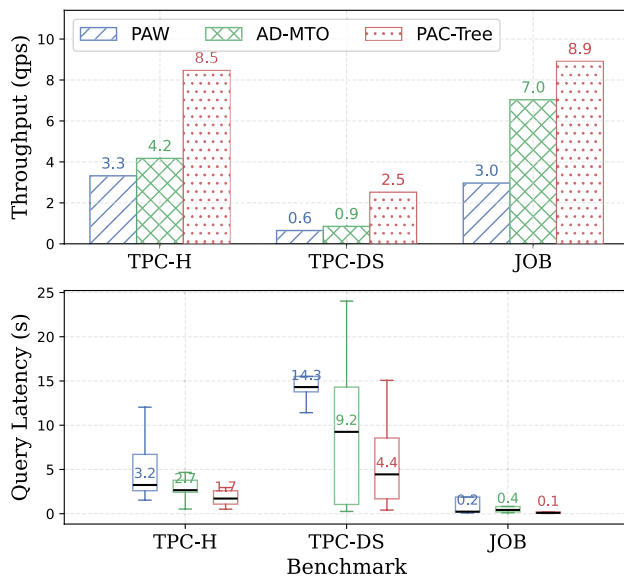
Ablations on Join Order Optimization. After partitioning and establishing the data layout, determining an optimal join order is essential for multi-table queries (discussed in Section 5). We compare the default query optimizer order with our QDG-driven hyper join-based reordering policy. Results show that PAC-Tree's join order consistently outperforms the default. On a per-query basis, the improvements

Table 5 Per-query average read bytes (GB) for each benchmark

Method	TPC-H (100GB)			TPC-DS (100GB)			JOB (6.9GB)		
	Mean	p50	p95	Mean	p50	p95	Mean	p50	p95
PW	33	30	46	42	46	47	0.91	0.02	6.0
AM	27	24	46	33	46	47	0.36	0.05	1.72
PT	15	14	27	32	41	47	0.26	0.01	1.43

The best results for each statistical metric are highlighted in bold

Abbreviations: PW = PAW, AM = AD-MTO, PT = PAC-Tree

**Fig. 15** End-to-end system performance evaluation on Apache Spark, conducted with three representative benchmarks

are most pronounced for queries involving complex join graphs, which is consistent with our expectations.

6.5 End-to-End System Performance Evaluation (RQ5)

To address RQ5, we conducted an empirical evaluation of read bytes, query latency, and system throughput on a production-grade Spark cluster. In our setup, TPC-H and TPC-DS datasets were scaled to 100 GB, with 100 queries generated from built-in templates. For JOB (6.9 GB), we sampled 100 queries from JOB-standard and JOB-light. All queries were consumed using a fixed-size client thread pool of eight workers, which regulates concurrency and caps the

number of in-flight queries, thereby preventing slow queries from monopolizing resources. Table 5 summarizes the average read bytes, showing that PAC-Tree consistently achieves the highest data-skipping efficiency, strongly correlated with the scan ratio results reported earlier.

Figure 15 further shows that read bytes are correlated with other performance metrics. Scanning fewer partitions reduce data loading, scanning, and predicate evaluation overheads, thereby alleviating query queue congestion especially when large queries block smaller ones ultimately lowering latency and improving throughput. The boxplot results confirm that PAC-Tree accelerates most queries while narrowing the distribution of latencies. To provide finer-grained insights, we partition our queries into four latency "bins" (Table 6). This categorization is crucial, as the vast majority of queries in our settings are short. The strength of PAC-Tree lies in its ability to effectively bound query latency within 3 seconds under TPC-H and TPC-DS, reducing queuing delays and preventing long-running queries, which in turn improves throughput by factors of $2\times$ and $2.7\times$ over AD-MTO. By contrast, in JOB workloads where most queries already complete within 3 seconds further latency reductions (sometimes below 1 second) yield only modest throughput gains ($1.27\times$).

Overall, PAC-Tree achieves the lowest latency across all benchmarks, with reductions of 65.9% and 45.7% relative to PAW and AD-MTO, respectively. These gains are accompanied by substantial decreases in read bytes (49% vs. PAW and 25% vs. AD-MTO) and significant improvements in system throughput ($3.2\times$ and $2\times$). More importantly, the benefits hold across diverse workloads, ranging from TPC-H to complex analytical TPC-DS and filter-heavy JOB, demonstrating that PAC-Tree generalizes well to different query characteristics.

Table 6 End-to-end query latency distribution (absolute error in seconds)

Query Exec-time	#Queries (TPC-H)			#Queries (TPC-DS)			#Queries (JOB)		
	PW	AM	PT	PW	AM	PT	PW	AM	PT
<1s	0	13	21	8	24	15	69	80	80
1s-3s	36	38	79	0	7	22	11	0	0
3s-10s	61	49	0	8	26	49	0	20	20
10s+	3	0	0	84	43	14	20	0	0

Bold values mark the method achieving the smallest number of slow queries (10s+)

Abbreviations: PW = PAW, AM = AD-MTO, PT = PAC-Tree

7 Conclusion and Future Work

This paper addresses the critical challenge of optimizing distributed joins, which are often constrained by inefficient data layouts. We introduce PAC-Tree, a novel hierarchical data layout that mitigates the limitations of existing partitioning schemes via a query-aware approach. PAC-Tree integrates data replication across diverse distribution keys at the upper level and employs predicate-aware multidimensional partitioning at the lower level. Together with efficient block grouping and join-order optimization, our design enhances data co-location and reduces I/O overhead. Evaluated on a Spark cluster with the TPC-H, TPC-DS, and JOB benchmarks, PAC-Tree achieved remarkable improvements, including up to a 45.7% reduction in query latency and a $2\times$ increase in throughput compared to state-of-the-art layouts.

Future work will further optimize PAC-Tree along two key directions. (1) We plan to investigate adaptive mechanisms that enable PAC-Tree to dynamically respond to workload shifts. This involves aggregating and filtering collected features to ensure the robustness of static partitions, as well as detecting recurring filter patterns to better inform the design of distribution and clustering keys. (2) We aim to reduce the reliance on manually tuned hyper-parameters within the algorithm, thereby minimizing intervention during deployment and enhancing the generality of PAC-Tree.

Acknowledgements The authors would like to acknowledge the support from the Bytehouse team at ByteDance.

Author Contributions The idea and implementation were jointly developed by all authors with equal contributions.

Funding This work was supported by the National Key Research and Development Program of China (Grant No. 2023YFB4503600) and the National Natural Science Foundation of China (Grant Nos. U23A20299, 62072460, 62172424, 62276270, and 62322214).

Data Availability Yes, all my experiments are performed on publicly available datasets.

Declarations

Competing Interests The authors declare that they have no conflicts of interest.

Ethical Approval and Consent to Participate Not applicable.

Consent for Publication Not applicable.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed

material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

References

1. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: Cluster computing with working sets. In 2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)
2. Gupta A, Agarwal D, Tan D, Kulesza J, Pathak R, Stefani S, Srinivasan V (2015) Amazon redshift and the case for simpler data warehouses. In Proceedings of the 2015 ACM SIGMOD international conference on management of data, 1917–1923
3. Doris: A High-Performance Real-Time Analytical Database Based on MPP Architecture. <https://doris.apache.org/docs/3.0/gettingStarted/what-is-new>
4. Papadomanolakis S, Ailamaki A (2004) Autopart: Automating schema design for large scientific databases using data partitioning. In Proceedings. 16th International Conference on Scientific and Statistical Database Management, 383–392. IEEE
5. Sun L, Franklin MJ, Krishnan S, Xin RS (2014) Fine-grained partitioning for aggressive data skipping. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 1115–1126
6. Yi L, Shanbhag A, Jindal A, Madden S (2017) Adaptdb: Adaptive partitioning for distributed joins. Proc VLDB Endow 10:589–600
7. Transaction Processing Performance Council (2021) TPC-H: benchmark specification. <http://www.tpc.org/tpch/>
8. Eadon G, Chong EI, Shankar S, Raghavan A, Srinivasan J, Das S (2008) Supporting table partitioning by reference in oracle. In SIGMOD Conference
9. Zamanian E, Binnig C, Salama A (2015) Locality-aware partitioning in parallel database systems. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data
10. Nam Y-M, Kim M-S, Han D (2018) A graph-based database partitioning method for parallel olap query processing. In 2018 IEEE 34th International Conference on Data Engineering (ICDE), 1025–1036. IEEE
11. Parchas P, Naamad Y, Van Bouwel P, Faloutsos C, Petropoulos M (2020) Fast and effective distribution-key recommendation for Amazon Redshift. Proc VLDB Endow 13(12):2411–2423
12. Zhou X, Li G, Feng J, Liu L, Guo W (2023) Grep: a graph learning based database partitioning system. Proc ACM Manag Data 1(1):1–24
13. Budalakoti S, Ziauddin M, Witkowski A, Kim YJ, Krishnamachari R, Wood A (2024) Automated clustering recommendation with database zone maps. In Companion of the 2024 International Conference on Management of Data, 68–79
14. Morton GM (1966) A computer oriented geodetic data base and a new technique in file sequencing
15. Bongki M, Jagadish Hosagrahar V, Christos F, Saltz Joel H (2001) Analysis of the clustering properties of the hilbert space-filling curve. IEEE Trans Knowl Data Eng 13(1):124–141
16. Yang Z, Chandramouli B, Wang C, Gehrke J, Li Y, Minhas UF, Larson P, Kossmann D, Acharya R (2020) Qd-tree: Learning data layouts for big data analytics. In Proceedings of the 2020 ACM

- SIGMOD International Conference on Management of Data, 193–208
17. Ding J, Minhas UF, Chandramouli B, Wang C, Li Y, Li Y, Kossmann D, Gehrke J, Kraska T (2021) Instance-optimized data layouts for cloud analytics workloads. *Proceedings of the 2021 International Conference on Management of Data*
18. Li Z, Yiu ML, Chan TN (2022) Paw: Data partitioning meets workload variance. *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 123–135
19. Ding J, Abrams M, Bandyopadhyay S, Di Palma L, Ji Y, Pagano D, Paliwal G, Parchas P, Pfeil P, Polychroniou O, et al (2024) Automated multidimensional data layouts in amazon redshift. In *Companion of the 2024 International Conference on Management of Data*, 55–67
20. Prim RC (1957) Shortest connection networks and some generalizations. *Bell Syst Tech J* 36(6):1389–1401
21. Bayer R, McCreight E (1970) Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 107–141
22. Bentley JL (1975) Multidimensional binary search trees used for associative searching. *Commun ACM* 18:509–517
23. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84*
24. Dong H, Chai C, Luo Y, Liu J, Feng J, Zhan C (2022) Rw-tree: A learned workload-aware framework for r-tree construction. *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2073–2085
25. Yuan H, Li G, Feng L, Sun J, Han Y (2020) Automatic view generation with deep learning and reinforcement learning. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 1501–1512
26. Han Y, Li G, Yuan H, Sun J (2021) An autonomous materialized view management system with deep reinforcement learning. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2159–2164
27. Moerkotte G (1998) Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*
28. Graefe G (2009) Fast loads and fast queries. In *DaWaK*
29. Liwen S, Franklin Michael J, Jiannan W, Eugene W (2016) Skipping-oriented partitioning for columnar layouts. *Proc VLDB Endow* 10(4):421–432
30. Agrawal R, Imielinski T, Swami AN (1993) Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference*
31. Transaction Processing Performance Council (2021) TPC-DS: decision support benchmark. <http://www.tpc.org/tpcds/>
32. Eltabakh MY, Tian Y, Özcan F, Gemulla R, Krettek A, McPherson J (2011) Cohadoop: flexible data placement and its exploitation in hadoop. *Proc VLDB Endow* 4:575–585
33. Apache Software Foundation. Hadoop distributed file system design (2025). <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>, Accessed: 2025-04-24
34. Kandula S, Orr L, Chaudhuri S (2019) Pushing data-induced predicates through joins in big-data clusters. *Proc VLDB Endow* 13(3):252–265
35. Jindal A, Palatinus E, Pavlov V, Dittrich J (2013) A comparison of knives for bread slicing. *Proc VLDB Endow* 6(6):361–372
36. sqlparse AA (2025) A non-validating sql parser for python. <https://github.com/andialbrecht/sqlparse>. Version 0.4.4
37. ClickHouse: Fast Open-Source OLAP DBMS. <https://clickhouse.com/docs/architecture/cluster-deployment>
38. Sudhir S, Tao W, Laptev N, Habis C, Cafarella M, Madden S (2023) Pando: enhanced data skipping with logical data partitioning. *Proc VLDB Endow* 16(9):2316–2329
39. Amazon Web Services. Enabling, disabling, and monitoring automatic table optimization. https://docs.aws.amazon.com/redshift/latest/dg/c_ato-enabling-disabling-monitoring.html, 2025. Accessed: 2025-05-18