Workload-Aware Incremental Reclustering in Cloud Data Warehouses

YIPENG LIU, Tsinghua University, China
RENFEI ZHOU, Carnegie Mellon University, USA
JIAQI YAN, Snowflake Inc., USA
HUANCHEN ZHANG*, Tsinghua University, China

Modern cloud data warehouses store data in micro-partitions and rely on metadata (e.g., zonemaps) for efficient data pruning during query processing. Maintaining data clustering in a large-scale table is crucial for effective data pruning. Existing automatic clustering approaches lack the flexibility required in dynamic cloud environments with continuous data ingestion and evolving workloads. This paper advocates a clean separation between *reclustering policy* and *clustering-key selection*. We introduce the concept of boundary micro-partitions that sit on the boundary of query ranges. We then present WAIR, a workload-aware algorithm to identify and recluster only boundary micro-partitions most critical for pruning efficiency. WAIR achieves near-optimal (with respect to fully sorted table layouts) query performance but incurs significantly lower reclustering cost with a theoretical upper bound. We further implement the algorithm into a prototype reclustering service and evaluate on standard benchmarks (TPC-H, DSB) and a real-world workload. Results show that WAIR improves query performance and reduces the overall cost compared to existing solutions.

CCS Concepts: • **Information systems → Data warehouses**; **Cloud based storage**; **Database query processing**; **Autonomous database administration**.

Additional Key Words and Phrases: cloud data warehouses; analytical query processing; micro-partitions; zonemap pruning; incremental table reclustering; workload-aware optimization; cost-based optimization

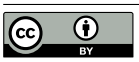
ACM Reference Format:
Yipeng Liu, Renfei Zhou, Jiaqi Yan, and Huanchen Zhang. 2026. Workload-Aware Incremental Reclustering in Cloud Data Warehouses. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 250 (June 2026), 27 pages. https://doi.org/10.1145/3802127

1 INTRODUCTION

The rapid growth of data in modern enterprises has created significant challenges for managing and analyzing large datasets. To handle this scale, organizations are progressively migrating their database systems from traditional on-premises infrastructure to cloud-based environments [35]. Cloud data warehouses, such as Snowflake [31, 48], Redshift [26], and BigQuery [41], have emerged as a popular solution. They adopt an architecture that disaggregates compute from storage to provide outstanding elasticity and scalability. A key feature of these cloud data warehouses is that they partition data into small, fixed-size units (we call these units "micro-partitions", following Snowflake's terminology) and store them in cloud object storage (e.g., AWS S3) using proprietary file formats or open-source ones such as Apache Parquet [10]. The system typically maintains the

*Huanchen Zhang is also affiliated with the Shanghai Qi Zhi Institute. Corresponding author.

Authors' Contact Information: Yipeng Liu, Tsinghua University, Beijing, China, yipeng.liu@tuna.tsinghua.edu.cn; Renfei Zhou, Carnegie Mellon University, Pittsburgh, USA, renfeiz@andrew.cmu.edu; Jiaqi Yan, Snowflake Inc., San Carlos, USA, jiaqi@snowflake.com; Huanchen Zhang, Tsinghua University, Beijing, China, huanchen@tsinghua.edu.cn.



ACM 2836-6573/2026/6-ART250
https://doi.org/10.1145/3802127

metadata (e.g., min-max zonemaps [42]) for each micro-partition in a separate service to enable efficient scan-set pruning during query optimization and execution. Studies have shown that effective pruning is critical for improving the performance of analytical queries [31, 54].

While micro-partitions are automatically created in data ingestion order, this natural ordering often does not align with the optimal physical layout for analytical workloads. Maintaining optimal clustering is essential to ensure that only relevant micro-partitions are accessed during queries. For example, consider a table storing sales information where data arrives chronologically (in batches) in *order_timestamp* order. If the micro-partitions are clustered according to the data's natural ingestion order, queries that contain filter conditions on *customer_id* must access a large portion of these micro-partitions, each containing only a few records that match the filter conditions.

For large tables with continuous data ingestion, manually maintaining the clustered state places a huge burden on database users to monitor and analyze DML patterns and workload distributions. Existing approaches, such as Qd-tree [49] and MDDL in Redshift [33], assume a stationary workload and learn an optimal layout for the entire table from historical queries. They are, however, suboptimal under continuous data ingestion and evolving workload patterns. Iceberg [8], Delta Lake [15, 16], and Databricks [12, 13] allow dynamically switching clustering keys to accommodate workload shifts, but the updated physical layout is only applied to newly ingested data. Snowflake [22] and Dremio [19] propose the data-driven incremental reclustering, where they develop static metrics to evaluate the clustering quality of each micro-partition and only recluster the ill-clustered ones. Although these approaches reduce the overall reclustering cost, they are unaware of workloads, resulting in unnecessary reclustering of rarely accessed data.

In this paper, we introduce Workload-Aware Incremental Reclustering (**WAIR**), a reclustering algorithm (and service) that can efficiently and automatically maintain a high-quality clustered state for large, micro-partitioned tables in a cloud data warehouse under continuous data ingestion and evolving query patterns. The key insight is that a small set of boundary micro-partitions (i.e., those that sit on the boundary of query ranges) largely determines pruning effectiveness. Built upon the theoretical analysis of WAIR's amortized cost, WAIR adopts an incremental workload-aware approach that prioritizes boundary micro-partitions with higher expected payoff. We develop a cost model for selecting a proper set of micro-partitions to recluster after each query to maximize the overall cost reduction (i.e., query execution savings - reclustering cost) within a time window. By decoupling the reclustering policy from the clustering-key selection, WAIR provides the flexibility for each micro-partition to choose its own clustering key to maximize the pruning gains. We then implement an automatic reclustering service based on the WAIR algorithm. The service is completely off the query's critical path and is a drop-in component that leverages query-execution statistics to schedule and execute reclustering for a cloud data warehouse. We experimentally compare WAIR against a variety of baseline approaches, including research proposals and publicly documented commercial methods, using standard benchmarks (TPC-H [30], DSB [32]) and a real-world workload. Our results show that WAIR significantly improves query performance and lowers the overall operational cost simultaneously compared to the baselines. Sensitivity analysis using our workload generator shows that WAIR remains robust under a wide range of workload scenarios.

We make four primary contributions in this paper. First, we propose a policy-key decoupled taxonomy and framework for reclustering. Second, we introduce the concept of boundary micro-partitions that dominate scan-set pruning efficiency, and prove a theoretical upper bound for the cost of reclustering all these boundary micro-partitions. Third, we propose WAIR based on boundary micro-partitions that can automatically determine the reclustering timing and the micro-partition set to maximize overall cost reduction. Finally, we implement a workload-aware automatic reclustering service based on WAIR and demonstrate its advantages in cost, query performance, and robustness against diverse workloads over existing solutions.

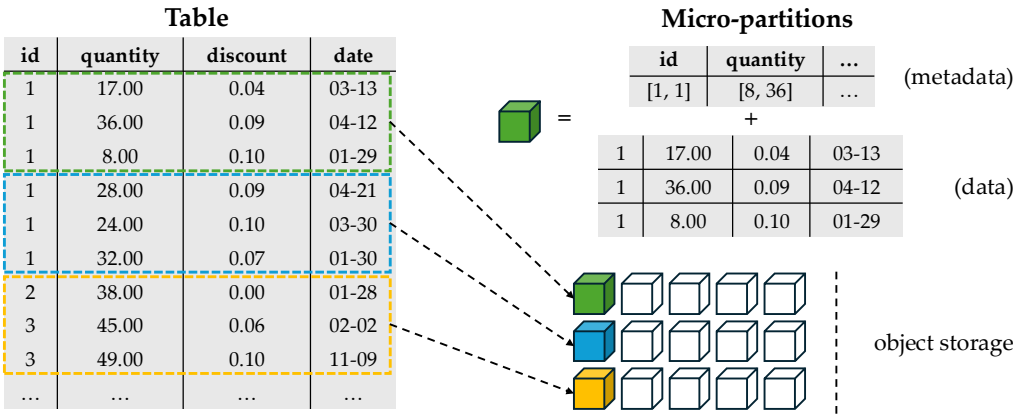


Fig. 1. Micro-partitions in Cloud Data Warehouses. A table is structured as multiple micro-partitions stored in the object storage layer. Each micro-partition comprises both metadata (e.g., zonemaps) and data blocks. During query execution, the metadata is used to prune irrelevant micro-partitions.

2 PRELIMINARIES

A defining feature of cloud data warehouses is the disaggregation of compute and storage for independent scaling [27, 31, 50]. For queries on large tables, avoiding full table scans is critical for achieving optimal analytical performance. A widely adopted approach is partitioning [34, 39, 49]. In cloud data warehouses such as Snowflake, tables are divided into **micro-partitions**[1] [31], a concept similar to "row groups" in columnar data formats like Parquet [10] and ORC [9]. As illustrated in Figure 1, incoming data is automatically partitioned into *immutable* files (i.e., micro-partitions) according to its natural ingestion order. These micro-partitions reside in cloud object storage (e.g., AWS S3, Azure Blob Storage) and serve as fundamental units for both data retrieval and pruning during query processing. Each micro-partition is maintained at a fixed size, typically a few tens of megabytes in practice (e.g., 32MB) [36].

To facilitate scan-set pruning, the system maintains metadata such as zonemaps [42] to store the min/max values for the columns in each micro-partition. During query execution, predicates (e.g., `WHERE` *col* `BETWEEN` *a* `AND` *b*) are evaluated against the zonemap of each micro-partition. If the partition's min-max range falls outside the query predicate, the entire partition can be safely skipped, thus drastically reducing the amount of data read from cloud storage.

The cloud's ability to provision nearly unlimited resources on demand has made *cost* a first-class citizen in system optimization [52]. In most cloud services, compute costs are directly tied to the CPU times with negligible setup fees [1]. For the reclustering task, the storage cost remains stable because the overall data volume does not vary much in different clustering states. Costs generated by sending S3 requests are minor compared to the compute and storage costs [4, 46]. Data transfer within the same region (e.g., from S3 buckets to compute instances) is free [4]. Therefore, the overall operational cost for query processing and reclustering is dominated by the consumed CPU time.

2.1 Benefits and Challenges of Clustering

By default, micro-partitions naturally cluster data based on its arrival order. However, many queries include predicates that do not align with this order, making an alternative data clustering beneficial [28, 33]. A table is considered clustered if its micro-partitions are organized according to a specific

[1]Throughout this paper, the term "partition" typically refers to micro-partitions, unless otherwise specified.

ID	Age	Gender
1	19	Male
2	20	Female
3	21	Male
4	21	Male
5	18	Other
6	20	Female
…	…	…

Micro-Partitions

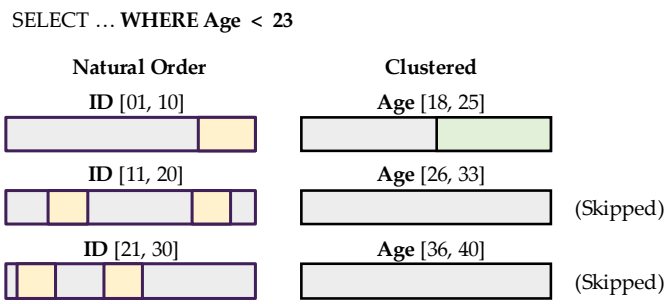


Fig. 2. Benefits of Clustering. The query predicated on Age must scan all micro-partitions in a naturally ordered table, while clustering by Age allows pruning all but the first micro-partition.

order that aligns with common query patterns. In this clustered state, rows that are logically close (e.g., having a small value distance in the specified column) are also stored physically close together, ideally within the same partition.

A key benefit of clustering is to improve the effectiveness of partition pruning. As shown in Figure 2, after sorting the table by the Age column, the query filtering on Age only needs to scan one partition rather than three if the data were ordered differently by ID. Joins, LIMITs, and other SQL operations that benefit from partition pruning [54] naturally perform better when the table is clustered. Maintaining effective clustering is crucial to achieving optimal query performance, especially for large tables [54].

Maintaining effective yet efficient clustering, however, presents significant challenges. Large tables in cloud data warehouses rarely remain static; they grow continuously through high-volume ingestions with diverse DML paths, including continuous streams and bulk loading [47]. Selecting an effective clustering key with the most performance benefit (i.e., a single column or a combination of columns using methods such as Z-ordering [43]), poses additional challenges. As query workload patterns evolve, a clustering key that was once optimal may become sub-optimal or even ill-suited, leading to wasted resources and negative performance impacts.

2.2 Related Work

Table 1. Flexibility Dimensions of Related Work. Representative systems classified by their policy and clustering key selection. Each cell denotes a distinct clustering maintenance strategy.

Policy \ Key	Fixed	Dynamic (*Manual*)	Dynamic (*Workload*)
Full Table	legacies		Qd-tree [49], Redshift [2]
New Data		Iceberg [7], Delta Lake [14]	Databricks [11]
Incremental (*Data-Driven*)		Snowflake [21], Dremio [18]	
Incremental (*Workload-Aware*)			**WAIR** (ours)

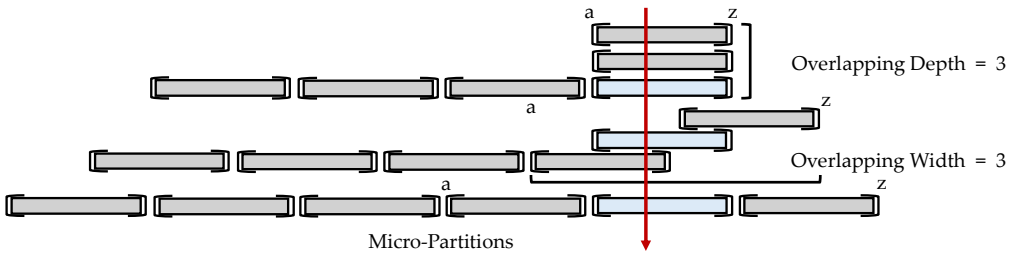


Fig. 3. Overlapping Depth and Width. Number of micro-partitions overlapping a given micro-partition (width) and a given point (depth). By reclustering, both the overlapping width and depth decrease from three to one.

The term "partitioning" is overloaded in database literature. A significant body of work focuses on *horizontal partitioning* (or sharding) for distributed databases (e.g., Azure Synapse SQL [17, 25]) or load balancing in streaming systems (e.g., Dalton [51]). The primary goal of these techniques is to minimize cross-node traffic and distribute compute load. In contrast, modern cloud data warehouses adopt micro-partitioning within the cloud storage layer for *data skipping*. This relies on lightweight metadata (e.g., zone maps) [53, 54] and space-filling curves (e.g., Z-ordering [43]) to prune irrelevant micro-partitions during execution. In this paper, we target this micro-partition level optimization to facilitate effective pruning, which is orthogonal to and can coexist with prior works.

A variety of research prototypes and commercial systems[2] have been proposed to maintain table clustering. A complete clustering solution usually comprises two algorithmic components: **Reclustering Policy** (when/what to reorganize) and **Clustering Key Selection** (sort order). We classify representative systems along these two dimensions in Table 1 and discuss their limitations.

For the *reclustering policy*, we consider four levels: 1) **Full Table:** Reorganizing the entire table during maintenance windows; 2) **New Data:** Clustering only newly ingested data; 3) **Incremental, Data-Driven:** Reclustering targeted subsets based on data distribution metrics (e.g., overlap) to capture most of the pruning benefit; 4) **Incremental, Workload-Aware:** The most adaptive class integrates workload statistics, data distribution, and a cloud cost model to prioritize the highest-impact partitions and schedule incremental reclustering at the optimal timing. For *clustering key selection*, we distinguish between: 1) **Fixed** keys defined at creation; 2) **Dynamic, Manual** keys updated by users for further optimization; 3) **Dynamic, Workload** keys automatically refined by the system, potentially using composite keys and hybrid layouts.

Qd-tree tailors the block assignment strategy for a given query workload to reduce the number of blocks accessed when running that workload [49]. **Redshift** uses a multi-dimensional data layout sort key to reorganize tables at system idles [3, 33]. While effective for static data, full table repartitioning is computationally prohibitive and less robust for large tables in cloud data warehouses with continuous ingestion and evolving workloads.

Iceberg supports in-place table evolution, where newly ingested data follow the updated partition layout or sort order [8]. **Delta Lake** organizes files into stable "*ZCubes*," each produced by a single `OPTIMIZE` job. During every optimization cycle, it clusters newly ingested files together with any undersized, partial *ZCubes* left behind by `DELETE` operations [15, 16]. **Databricks** extends Delta Lake by analyzing query workloads to identify the most effective clustering columns, then applies those keys to subsequent ingests [12, 13]. These approaches have two main drawbacks: 1) Existing historical data remains untouched, and 2) independent "sorted runs" across batches degrade global clustering quality over time.

[2]Because most industrial implementations are proprietary features of commercial cloud data warehouses, our analysis relies on publicly accessible documentation, patents, and published literature gathered on our best-effort basis.

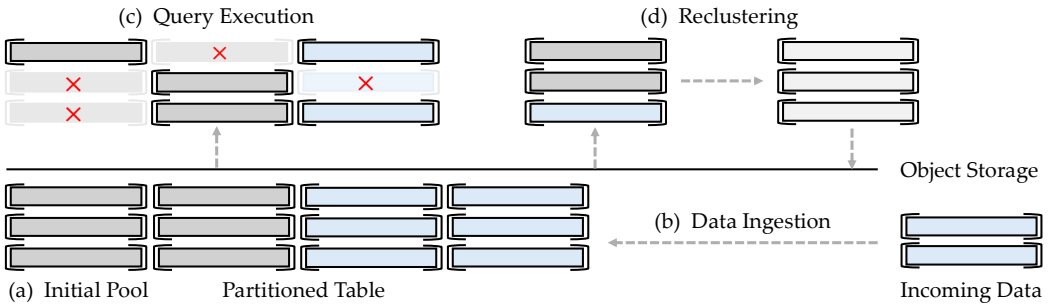


Fig. 4. System Model with Reclustering. (a) An existing data pool containing previously stored data. (b) Incoming data is partitioned and ingested in its natural order. (c) Query workloads use metadata to fetch only relevant partitions for processing. (d) Partitions are fetched, sorted, and then persisted back into storage.

Snowflake evaluates a table's clustering quality by the overlapping metrics [22]. Using partition metadata, it counts the number of micro-partitions in the table whose key ranges overlap with a given micro-partition and a given point (see Figure 3). These metrics averaged across the table reflect its overall clustering quality. **Dremio** uses the same metric, where only a small subset of files with the highest overlapping depth within their key ranges are reclustered at optimization cycles, until the table satisfies a user-defined overlapping depth threshold [19]. While more efficient than full repartitioning, these data-driven techniques: 1) involve potential expensive scans to compute metrics, 2) rely on manual parameter tuning; 3) ignore query patterns, resulting in wasteful reclustering of rarely accessed historical data.

The two strategies outlined above typically require a predefined clustering key, which leads to additional drawbacks: 1) Key selection requires domain expertise. 2) Key updates fragment the layout (old data retains old keys); 3) Even composite keys (e.g., Z-ordering) may fail to serve diverse access patterns effectively.

3 Boundary Micro-Partitions

A key observation behind incremental reclustering is that most of the query performance benefits come from reclustering only a strategic subset of partitions. In this section, we introduce the concept of boundary partitions and explain why they are critical for pruning efficiency. We then propose a simple greedy reclustering algorithm based on boundary partitions with a theoretical analysis of its amortized cost.

3.1 The Reclustering Workflow

Figure 4 shows the typical workflow in a cloud data warehouse. An initial pool of micro-partitions exists in cloud object storage and is clustered based on certain clustering keys. A **clustering key** is the column(s) used to sort a micro-partition. For the theoretical analysis in this section, we assume a clustering key contains a single column. New data is periodically ingested into database tables as micro-partitions. These incoming micro-partitions are clustered in their **natural ingestion order**, which is typically different from the clustering-key order.

Each **query** in our problem context defines its scan set by specifying the target table along with the pushed-down range predicates. Meanwhile, **reclustering** is triggered in the background, following a three-step process: 1) retrieve relevant micro-partitions from object storage, 2) sort the records by the clustering key and assemble new micro-partitions, and 3) write newly created micro-partitions back to object storage. Although reclustering is not on the critical path to affect

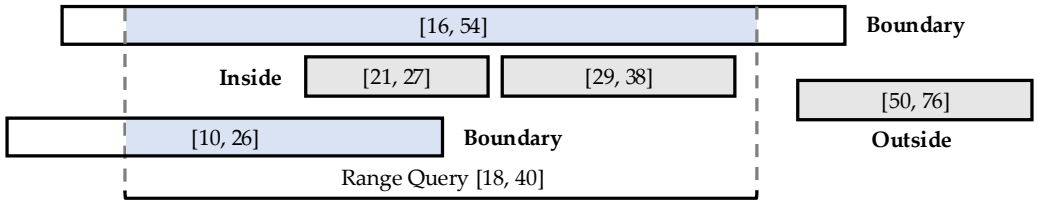


Fig. 5. Partitions on the Boundary. Boundary partitions overlap with query range edges and primarily affect pruning efficiency.

Algorithm 1 A Greedy Algorithm on Boundary Partitions

Input: Partitions $\{P_i\}$ with ranges $\{[a_i,\ b_i]\}$; Predicate $[l,\ r]$

$boundaries \leftarrow \{P_i \mid l \in [a_i,\ b_i]\}$
Recluster($boundaries$)

$boundaries \leftarrow \{P_i \mid r \in [a_i,\ b_i]\}$
Recluster[3]($boundaries$)

query latency, it consumes additional compute resources. Therefore, there is a trade-off between the immediate resource expenses on reclustering operations and the long-term resource savings on future queries.

3.2 Micro-Partitions on the Query Boundaries

A basic assumption of workload-aware algorithms is that future queries follow a similar statistical pattern as the historical ones. As depicted in Figure 5, we define **boundary micro-partitions** as those that *partially* overlap with at least one already-queried range. Each range query generates two sets of boundary partitions that overlap either the left or right border of its range predicate. Micro-partitions that fall completely inside or outside any query ranges do not require reclustering because all the records in them are either included or excluded from query results. Boundary micro-partitions, on the other hand, are critical to pruning efficiency because scanning these micro-partitions leads to retrieving unnecessary rows and therefore wastes I/O bandwidth during query execution. Algorithm 1 presents a basic greedy algorithm that reclusters all boundary micro-partitions after each query. We then provide a theoretical analysis to show that this algorithm incurs at most *logarithmic* additional cost.

Assumptions. Without loss of generality, we assume each value in the clustering-key column is a real number, and each micro-partition is a multiset of m such values. We represent each micro-partition as $P[a_i, b_i]$ using its clustering-key range. Consider performing a sequence of *ingestion* and *reclustering* operations on a table with a collection of initial micro-partitions, where

- `Ingestion`: Add a new micro-partition of size m into the table. The cost of this operation is excluded from the analysis.
- `Recluster`(x): Given a value x, recluster all the micro-partitions whose range covers x (i.e., $x \in [a_i, b_i]$). Skip micro-partitions with a degenerate range $P[x, x]$. Assume that the cost of reclustering a micro-partition is 1.

[3]Duplicate partitions should be omitted from reclustering; however, their presence does not affect the theoretical analysis of the algorithm.

Let n denote the total number of micro-partitions ingested. Let q denote the total number of range queries performed in the entire process. The number of reclustering operations is $2q$ because each range query $[l,\ r]$ is transformed into two reclustering operations at l and r. We assume that only keys $1, 2, \ldots, 2q$ might be operated with reclustering and all clustering keys lie in the range $(0, 2q + 1)$; otherwise, we could rescale them using a piecewise linear function to make this true while preserving their relative orders.

Potential Function. We use the classic potential method [29] to examine the *amortized cost* of the operations. A potential function, typically denoted as $\Phi(S)$, maps each state of the data structure to a non-negative number. "Potential" here refers to a conceptual "energy" or "prepaid work" stored within a data structure. For a micro-partition $P[a_i, b_i]$, we define its (reclustering) potential as:

$$\phi(a_i, b_i) \coloneqq \begin{cases} 0 & \text{if } a_i = b_i \text{ or } \lceil a_i \rceil > \lfloor b_i \rfloor, \\ 4 + 4\log(\lfloor b_i \rfloor - \lceil a_i \rceil + 1) & \text{otherwise.} \end{cases}$$

Note that $0 \le \phi(a_i, b_i) \le O(\log q)$. A micro-partition with a wider key range has a larger $\phi(a_i, b_i)$ because it is more likely to be involved in a future reclustering operation. The potential of all micro-partitions of a table (i.e., the data structure) is defined as $\Phi = \sum \phi(a_i, b_i)$. Each ingestion operation increases the potential by $O(\log q)$.

Micro-Partition Matching. Suppose `Recluster`(x) takes k micro-partitions $P[a_i, b_i], i = 0, 1, \ldots, k-1$ as input and outputs k micro-partitions $P[c_i, d_i]$. In this case, the potential of each input micro-partition $P[a_i, b_i]$ is at least 1. We also note that the key ranges of the output micro-partitions are pairwise disjoint except at their endpoints. Therefore, any output clustering key can exist in at most two non-degenerate micro-partitions.

Lemma 1. *We say an output $P[c_j, d_j]$ is **matched** to an input $P[a_i, b_i]$ if $[c_j, d_j] \subseteq [a_i, b_i]$. Then there exists a matching of size $k - 3$ between the input and output micro-partitions (i.e., there are at most 3 unmatched output micro-partitions).*

Proof. Construct a bipartite graph G whose left vertices represent the k input micro-partitions and right vertices represent the k output micro-partitions. Draw an edge from input $P[a_i, b_i]$ to output $P[c_j, d_j]$ if and only if $[c_j, d_j] \subseteq [a_i, b_i]$. We apply Hall's theorem [38] to show that G has a matching of size $k-3$. Let S be a subset of the input micro-partitions, and let $N(S)$ be the neighbors of S in the output micro-partitions. We only need to show that $|N(S)| \ge |S| - 3$ for all S.

Let $A = \min_{i \in S} a_i$ and $B = \max_{i \in S} b_i$. Since each input micro-partition covers x, there is an input $P[a_i, b_i] \supseteq [A, x]$ and an input $P[a_j, b_j] \supseteq [x, B]$. Moreover, the input micro-partitions in S provide a total of $m \cdot |S|$ keys in the range $[A, B]$, which implies that there are at least $|S|$ output micro-partitions whose ranges intersect with $[A, B]$. Among these outputs, at most one can have its left endpoint smaller than A, and at most one can have its right endpoint greater than B; the remaining $|S| - 2$ output micro-partitions must be subintervals of $[A, B]$. Furthermore, at most one output micro-partition $P[c_i, d_i]$ can strictly contain x (i.e., $c_i < x < d_i$), and the remaining $|S| - 3$ outputs must be subintervals of either $[A, x]$ or $[x, B]$, which implies that they are the neighbors of S. Therefore, $|N(S)| \ge |S| - 3$ for all S, and the lemma follows. □

Lemma 2. *For all but $O(\log q)$ matched input-output pairs, $\phi(c_j, d_j) \le \phi(a_i, b_i) - 4$.*

Proof. We first prove this claim for output micro-partitions $P[c_j, d_j]$ with $c_j \ge x + 12$. Assume there are $r \le k$ such outputs $P[c_1, d_1], \ldots, P[c_r, d_r]$ satisfying $d_i \le c_{i+1}$ for $1 \le i < r$. Let $d_0 = x + 12$ for convenience, and let $P[a_i, b_i]$ be the input micro-partition matched with $P[c_i, d_i]$. Each of these r outputs falls into either case:

- $(d_i - x) \geq 3/2 \cdot (d_{i-1} - x)$. Since the maximum $\lfloor d_i \rfloor$ is at most $2q$, there can only be $O(\log q)$ such output micro-partitions.
- $(d_i - x) < 3/2 \cdot (d_{i-1} - x)$. This simplifies to $3(d_i - d_{i-1}) < d_i - x$, which further implies

$$d_i - c_i \leq d_i - d_{i-1} \leq \frac{d_i - x}{3} \leq \frac{d_i - x}{2} - 2 < \frac{b_i - a_i - 1}{2} - 1,$$

  where the second-to-last inequality holds because $d_i - x \geq 12$; the last inequality holds because $P[a_i, b_i]$ covers both x and $P[c_i, d_i]$. Therefore,

$$\phi(c_i, d_i) \leq 4 + 4\log(d_i - c_i + 1) \leq 4 + 4\log(\frac{b_i - a_i - 1}{2}) \leq \phi(a_i, b_i) - 4.$$

That is, for all but $O(\log q)$ output micro-partitions $P[c_i, d_i]$ with $c_i \geq x + 12$, we have $\phi(c_i, d_i) \leq \phi(a_i, b_i) - 4$. By symmetry, the same holds for output micro-partitions $P[c_i, d_i]$ with $d_i \leq x - 12$. For the remaining partitions, either:

- $P[c_i, d_i]$ is non-degenerate and covers an integer in $[x - 12, x + 12]$. There are only $O(1)$ such micro-partitions, since at most two non-degenerate outputs cover any given integer.
- $P[c_i, d_i]$ is degenerate or does not cover any integer. In either case, its potential is 0 and is at least 4 less than its matched input partition (which is at least 4).

Putting all cases together completes the proof. □

Lemma 3. *A reclustering operation of k micro-partitions decreases the potential Φ by at least $4k - O(\log q)$.*

Proof. We analyze the potential change $\Delta\Phi$ of a reclustering operation as follows:

- At most 3 unmatched output micro-partitions are created, which increase the potential by at most $O(\log q)$.
- At most 3 unmatched input micro-partitions are removed, which do not increase the potential.
- For each matched pair of an input $P[a_i, b_i]$ and an output $P[c_j, d_j]$, we have $\phi(c_j, d_j) \leq \phi(a_i, b_i)$. For all but $O(\log q)$ matched pairs, we have $\phi(c_j, d_j) \leq \phi(a_i, b_i) - 4$.

Taking a summation over all these changes yields the lemma. □

If a range query accesses k boundary micro-partitions, it incurs k cost to fetch and another k cost to recluster them. Combined with a potential change of at most $O(\log q) - 4k$, the amortized cost of fetching and reclustering boundary micro-partitions is at most $O(\log q)$ per query. Non-boundary micro-partitions accessed by a query are fully utilized and their cost is bounded by the query's optimal output size. Putting all pieces together, we have:

Theorem 4. *For a sequence of batched ingestions and range queries, the algorithm achieves a total cost of less than*

$$O((n + q)\log q) + \sum \lceil |\text{output}_i|/m \rceil,$$

where output_i *is the number of keys returned by the i-th range query.*

3.3 Greedy Algorithm Extensions

To bridge the gap between theory and practice, we propose three modifications to the greedy algorithm in Algorithm 1. These adjustments address system constraints (e.g., memory limits) while still being amenable to theoretical analysis:

(1) **Memory Limit:** Let $k_{\max} = \Omega(\log n)$ be the memory capacity for reclustering micro-partitions. If a target set $|S| > k_{\max}$, we split S into disjoint subsets of size $k_{\max}$ and recluster them separately to avoid expensive external-memory sorting.

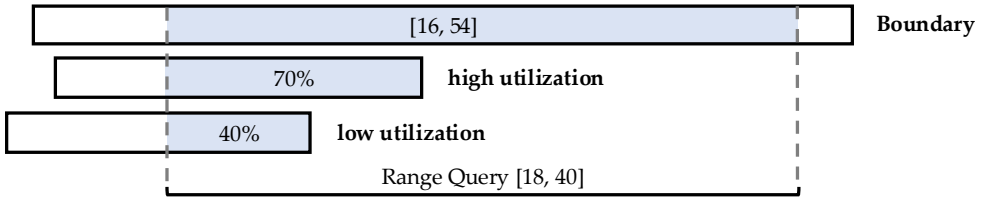


Fig. 6. Micro-Partition Utilization. Utilization measures percentage of a micro-partition's bytes actually needed by the query.

(2) **Warm Start:** Let $C_{\text{start}} = \Omega(\log n)$ be a chosen parameter. We enforce a budget of $C_{\text{start}} \cdot t$ over the baseline cost during the first t steps to prevent initial cost spikes. A subset of proper size is allowed if a reclustering operation violates this constraint.
(3) **Clustering Key Reselection:** Every $t_{\text{resel}} = \Theta(n)$ operations, we re-evaluate the clustering key for subsequent reclustering operations to use. This enables adaptation to workload shifts while preventing rapid oscillations.

We analyze the cost of the extensions using the same potential function analysis as in Section 3.2. By Lemma 3, a reclustering operation of $k_{\max}$ micro-partitions incurs non-positive amortized cost. Thus, each range query and its subsequent reclustering operations incur an amortized cost of at most $O(\log n)$ under the memory limit. The warm-start phase affects only the initial $O(n)$ steps and does not impact the amortized cost thereafter. Periodic clustering-key reselection adds $O(n \log n)$ potential every $\Theta(n)$ operations, incurring at most $O(\log n)$ to the amortized cost per operation. Putting all pieces together, we have the following theorem.

Theorem 5. *For a sequence of batched ingestions and range queries, the adjusted greedy algorithm achieves a total cost of at most*

$$O((n+q)\log n) + \sum \lceil |\text{output}_i|/m \rceil + (\text{warm starting cost}),$$

where output_i *is the number of keys returned by the i-th range query, and the warm starting phase lasts for at most* $O(n)$ *operations.*

4 WORKLOAD-AWARE RECLUSTERING

In this section, we build upon the concepts of boundary micro-partitions and propose a practical automatic reclustering framework. We first introduce a key metric "utilization" to quantify the expected savings of reclustering each boundary micro-partition. We then present a cost model that trades off between the reclustering overhead and the expected query execution savings to guide the proper timing and micro-partition selection for reclustering. Finally, we propose a hybrid layout where different micro-partitions can adopt different clustering keys to maximize the reclustering benefit under a diverse workload.

4.1 Micro-Partition Utilization

As illustrated in Figure 6, we define the **utilization** of a micro-partition with respect to a query as the ratio between the number of bytes read by the query and the total number of bytes of the micro-partition[4]. A low utilization indicates poor clustering: the system reads the full micro-partition, yet the query uses only a small fraction of the retrieved rows. Reclustering poorly utilized micro-partitions can yield more pruning benefits. For example, reclustering 100 micro-partitions at

[4]The utilization calculation in our prototype system considers practical aspects such as projection pushdown, row groups in storage format, and data encoding and compression (see Section 5.2).

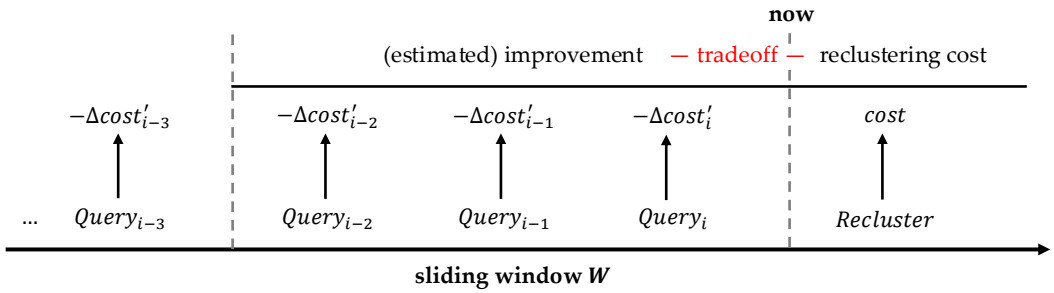


Fig. 7. Tradeoff between Reclustering Cost and Estimated Improvement. Reclustering incurs a fixed overhead. Its potential benefit is estimated by summing the projected improvements over a preceding sliding time window. If the estimated benefit outweighs its overhead, a reclustering operation is triggered.

utilization 10% could generate 10 fully utilized micro-partitions for the same query, thus reducing the scan set by $\approx 10\times$ for similar future queries.

4.2 Cost-based Trade-off

In cloud data warehouses, reclustering imposes a one-time cost but delivers future performance benefits. An effective reclustering policy, therefore, initiates the operation only when its expected gains outweigh incurred costs. To quantify this trade-off, we develop a cost model that translates each micro-partition's utilization into predicted query-time savings. By comparing predicted savings with reclustering overhead, the cost model guides our reclustering policy to the optimal timing and target micro-partition set.

Let $cost_q(Q)$ denote the cost of executing a single query Q. Let $cost_r(\mathcal{P})$ represent the cost of reclustering a set of micro-partitions $\mathcal{P} = \{P_i\}$. Given a preceding sliding time window W, as shown in Figure 7, we estimate the cost reduction $\Delta\hat{cost}_q(\mathcal{P} \mid Q)$ if the micro-partition set $\mathcal{P}$ were reclustered. Then, the `Recluster`($\mathcal{P}$) operation is triggered only when the predicted aggregate savings from all queries in W outweigh the estimated reclustering expense, and the current reclustering "debt" is under a predefined limit:

$$\Delta\hat{cost} = \hat{cost}_r(\mathcal{P}) - \sum_{Q \in W} \Delta\hat{cost}_q(\mathcal{P} \mid Q) < 0 \tag{1}$$

$$\Delta cost = \sum_{\mathcal{P}_i \in W} cost_r(\mathcal{P}_i) - \sum_{Q \in W} \Delta cost_q(Q) < cost\_limit \tag{2}$$

The purpose of the cost limit is to bound the wasted effort under a dramatic workload shift. We next express the above estimated costs in CPU times. For a query Q, the estimated execution time reduction contributed by a particular micro-partition P is:

$$\Delta\hat{t}_q(P \mid Q) = \frac{(1-u) \cdot size_{read}(P)}{size_{read}(Q)} \cdot t_{read}(Q) \tag{3}$$

where u denotes the utilization of P with respect to Q, and $(1-u) \cdot size_{read}(P)$ represents the scan size that could be saved after reclustering P. Then $(1-u) \cdot size_{read}(P)/size_{read}(Q)$ represents the estimated fraction of query read time saving.

The time estimation $t_r(\mathcal{P})$ for reclustering a micro-partition set $\mathcal{P}$ depends on whether $\mathcal{P}$ fits in memory. If they fit, we perform an in-memory quicksort on the records according to their clustering keys. Otherwise, we fall back to an external merge sort. In both cases, $t_r(\mathcal{P})$ is expressed as a function of the aggregate size of $\mathcal{P}$.

Algorithm 2 Workload-Aware Reclustering Policy

Input: All micro-partitions $\mathcal{P}$; Time window W

```
for P ∈ 𝒫 do
    Δcôst_q(P) ← Σ_{Q∈W} Δcôst_q(P | Q)
[P_i] = sort_desc({Δcôst_q(P), P ∈ 𝒫})
for cut ∈ {1, 2, ..., |𝒫|} do
    [P_i]_cut = {P_1, P_2, ..., P_cut}
    Δcôst = côst_r([P_i]_cut) − Σ_{P∈[P_i]_cut} Δcôst_q(P)
cut* ← arg min_cut Δcôst                          ▷ the smaller the better
if Δcôst < 0 and
   current Δcost + côst_r([P_i]_cut*) < cost_limit then
    Recluster([P_i]_cut*)
```

Additionally, we dynamically adjust the sliding window size based on its prediction accuracy to adapt to the evolution of the query workload and the data distribution. After completing a window W of queries, we compare the actual cost reduction $\Delta cost_q = \sum_{Q \in W} \Delta cost_q(Q)$ against the predicted cost reduction $\Delta \hat{cost}_q = \sum_{Q \in W} \Delta \hat{cost}_q(Q)$. If $\Delta cost_q > \Delta \hat{cost}_q$, it indicates that the workload is relatively stable, and we therefore double the window size to capture a longer query history. Otherwise, we halve the window size to focus more narrowly on the most recent workload shifts and distribution changes.

Algorithm 2 summarizes our workload-aware reclustering policy. The system triggers potential `Recluster` operations right after completing each query. To determine the set of micro-partitions for reclustering, we loop through all the micro-partitions accessed in W. For each micro-partition P, we first estimate the cost reduction $\Delta \hat{cost}_q(P)$ for all queries in W that could benefit from reclustering P. We then sort the micro-partitions according to the cost reduction in descending order and determine a *cut* in the sorted sequence where reclustering all the micro-partitions before the *cut* yields the largest estimated cost reduction. Finally, we perform these reclustering operations subject to a cost limit, as shown in Figure 7. The size of W is adjusted after each sliding window as described above. Reclustering decisions primarily reuse query-execution statistics already logged by the system, incurring minimal overhead.

4.3 Cost Model Extensions

We introduce two cost model extensions to allow adjusting reclustering policies to specific budget constraints or performance goals.

(1) Reclustering Aggressiveness. Our standard cost model triggers reclustering when estimated query savings exceeds immediate reclustering costs. To accommodate users' different risk preferences, we extend the model with an aggressiveness factor $\alpha > 0$ and a dynamic budget constraint:

$$\sum_{Q \in W} \Delta \hat{cost}_q(\mathcal{P} \mid Q) > \alpha \cdot \hat{cost}_r(\mathcal{P}) \tag{4}$$

$$\hat{cost}_r(\mathcal{P}) \leq \sum_{j>i} \Delta cost_q(\mathcal{P}_i \mid Q_j) - \sum cost_r(\mathcal{P}_i) + c \cdot |Q| \tag{5}$$

As shown in Equation (4), reclustering is triggered when the estimated savings are at least α times the reclustering cost. The smaller the α, the more proactive the policy. Equation (5) enforces a reclustering budget based on realized savings. Query savings are measured by comparing zonemaps

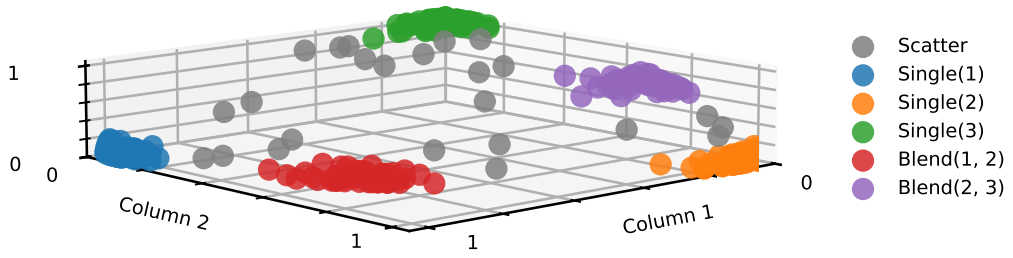


Fig. 8. Hybrid Layout for Clusters. After clustering, each partition group is assigned a physical layout that best suits its savings signature, producing a hybrid table layout.

before and after reclustering to track additional micro-partitions skipped at future queries. A reclustering operation is permitted only if its cost does not exceed the accumulated "credit" (i.e., net savings from past reclustering decisions plus a per-query "allowance" c).

(2) Interplay with Performance. While our primary objective is to minimize cost, users could prioritize query latency over cost aggressively. Integrating performance into a cost-based model is challenging because the monetary value of query speed varies by application. We discuss how our cost model can be extended to accommodate performance-critical scenarios. Assuming that all queries are equally important, we introduce a parameter β to denote the price a user is willing to pay for every additional pruned micro-partition. Then, for each query $Q \in W$, we augment its estimated savings with a performance bonus:

$$\Delta\hat{cost}_q(\mathcal{P} \mid Q) \leftarrow \Delta\hat{cost}_q(\mathcal{P} \mid Q) + \beta \cdot |\Delta\hat{pruned}(\mathcal{P} \mid Q)| \tag{6}$$

where $|\Delta\hat{pruned}(\mathcal{P} \mid Q)|$ denotes the estimated increase in skipped micro-partitions for Q after reclustering $\mathcal{P}$. By tuning β, users can express their specific exchange rates between cost budget and query performance.

4.4 Hybrid Layout

Unlike prior data-driven methods that must commit to a pre-selected clustering key, our approach decouples the reclustering policy from the clustering key selection by providing the flexibility for each micro-partition to choose its most beneficial clustering key.

To understand how each micro-partition might benefit from reclustering, we decompose its aggregate estimated savings into per-column components. We assume that each predicated column contributes equally in a query. Specifically, for each selected micro-partition P, we distribute its aggregated savings across all columns that appear in a query's predicate within the sliding window:

$$s_P = \left(s_P^{(1)}, s_P^{(2)}, \ldots, s_P^{(d)}\right),$$

where d is the number of attributes, $s_P^{(i)}$ is the share of P's projected benefit attributed to column i. Each savings vector is then plotted in a high-dimensional space to create the micro-partition's "**savings signature**", as illustrated in Figure 8.

Building on the theoretical analysis in the previous section, a straightforward approach is to derive a single global clustering key corresponding to the base column with the highest aggregated savings. While this strategy inherits the theoretical cost guarantees, practical workloads often exhibit heterogeneous access patterns where no single key is optimal for all partitions. To address this, we leverage the decoupling of our framework to propose a **hybrid layout** strategy. This

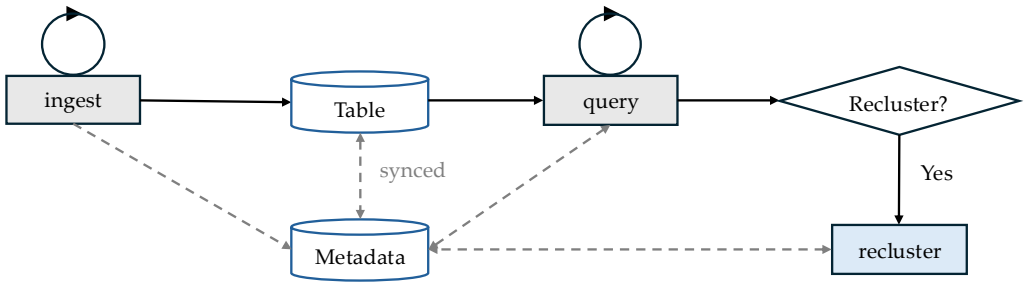


Fig. 9. Service Framework Workflow. A metadata service continuously collects partition metadata, workload execution statistics, and reclustering snapshots. After a query completes, a reclustering operation may be triggered, which uses the metadata to determine its scope and is then executed on a dynamically provisioned compute node.

approach allows different micro-partitions to adopt different clustering keys, trading the strict theoretical simplicity for greater flexibility and empirical performance gains.

We first normalize each vector to unit length and then cluster the normalized points with DBSCAN [37] using cosine distance. To interpret the resulting clusters, we define two families of anchor points: 1) the unit basis vectors e_i (single-column dominance), and 2) uniform multi-column anchors v_S with equal weights (multi-column blend). After clustering, each cluster is labeled with the anchor that is closest to its centroid.

- **Single-column dominance.** If a micro-partition's projected savings are contributed almost entirely by a single column, reclustering using the column as the clustering key is sufficient. Micro-partitions within the same cluster are grouped and sorted together by their shared dominant column.
- **Multi-column blend.** When savings concentrate on a small set of columns, we recluster using a Hilbert-curve ordering [40] over those columns. The columns are arranged in descending order of their overall contribution.
- **High-dimensional scatter.** Micro-partitions with diffuse savings across many columns do not benefit from a lexicographic key. Instead, we organize these micro-partitions with Qd-tree [49] to adapt to arbitrary query patterns.

The candidate micro-partitions are therefore divided into mutually exclusive groups, each assigned the physical layout that promises the greatest benefit. This strategy produces a hybrid table organization: different regions of the data are clustered with different keys, yet every reclustering decision is still guided by the same workload-aware cost model described earlier. Micro-partitions thus shift between groups as the workload evolves, allowing the table layout to adapt over time.

5 RECLUSTERING AS A SERVICE

Having presented the WAIR algorithm, we now introduce the Automatic Reclustering Service (ARS), a scalable reclustering framework that autonomously manages background operations to preserve table clustering quality over time. ARS leverages execution statistics with minimal compute overhead to decide the scope and timing of reclustering. ARS moves all reclustering work off the query-execution critical path, ensuring zero disruption to production workloads in practice. In this section, we describe how to embed the WAIR algorithm into ARS with minimal engineering effort and full compatibility on top of cloud data warehouses. We present the practical feasibility of ARS and the implementation details to be evaluated, and provide an overview of pruning opportunities across representative query patterns.

5.1 Service Framework

As shown in Figure 9, integrating the reclustering framework into an existing cloud data warehouse additionally requires: 1) an (often already in place) metadata service, and 2) a dedicated reclustering job executor.

Metadata Service. Three main categories of metadata are collected: 1) *Partition-level table statistics*, including min-max values, sizes, and compression ratios for each column chunk collected from partition metadata; 2) *Partition utilization statistics*, capturing detailed partition-level information for each query, such as partition pruning outcomes, data access volume, and measurements of filtered rows along with their sizes; 3) *Query-level statistics*, including total I/O size, I/O time, network transfer speed, and CPU time.

Metadata is continuously tracked and preserved during data ingestion and query execution asynchronously for further utilization. In modern distributed database systems, the metadata service is typically a standalone component. It should integrate seamlessly with a cloud data warehouse's existing metadata infrastructure as a drop-in module.

Reclustering Executor. After specifying the partition set to be reclustered and corresponding layouts, the reclustering executor retrieves these partitions from storage layer, loads their data into local memory, and reorganizes them into new partitions. These clustered partitions are then persisted back to the object storage.

Reclustering operations are designed to be non-blocking in distributed systems. When a reclustering operation completes, ARS publishes a new table snapshot that references the current set of partitions. Because reclustering reorganizes data without changing any row values, queries that began before completion continue to read the prior snapshot along with any newly ingested partitions. New queries started after completion read the reclustered partitions referenced by the latest, updated snapshot. Once no queries reference an older snapshot, ARS safely removes obsolete partitions during garbage collection. The table snapshot chain incurs minimal storage overhead and allows new queries to benefit immediately once reclustering completes. Reclustering tasks run on dynamically provisioned compute nodes, entirely outside the query execution critical path. In cloud data warehouses with a cache layer, reclustering naturally benefits from scan sharing as it requires scanning complete micro-partition data. Moreover, systems can opt for the k-th quantiles method [29] over standard, well-optimized sorting to recluster partitions in $O(n \log k)$ time.

5.2 Implementation Details

We implement the service using DuckDB [44, 45] as the execution engine and deploy in a cloud environment backed by AWS S3 and Redis [20] (metadata). Data is stored in Parquet [10]. During ingestion, we use Arrow [6] to write data into fixed-size partitions and extract partition-level statistics from file headers into Redis.

We instrument DuckDB to extract predicates from the ongoing query and evaluate against the active table snapshot in Redis. We modify the execution engine to collect relevant statistics when executing queries directly on S3. Data transfer size of each partition is recorded by patching the `HTTPFS` extension. Data chunk before and after filtering are captured at the `TableScan` executor.

We consider three major optimizations in DuckDB: 1) projection pushdown, 2) column chunks, and 3) encoding and compression. For unpruned partitions, we record both the total column chunk size retrieved and the actual size read by a query in each column after projection pushdown. A partition's utilization is then adjusted according to each column chunk's compression ratio. These cost metrics are evaluated asynchronously to determine a reclustering decision. After a reclustering job finishes, the metadata of these new partitions is recorded, and a new table snapshot is published to keep production workloads uninterrupted.

5.3 Pruning Opportunities

Zonemap-based pruning extends beyond simple filters to support complex operations. We summarize the pruning opportunities for representative query types, as documented by Snowflake [54]. For **LIMIT** clauses, a well-clustered table facilitates early termination once enough *fully-matching* partitions are found. For **JOIN** operations, build-side values are summarized into zonemaps for the probe side to prune partitions, which are essentially range queries that benefit from our reclustering approach. Snowflake builds a *pruning tree* to handle complex expressions across **diverse predicates**. In our context, each node in this tree identifies a subset of boundary partitions associated with the base columns. Our approach improves the efficiency of this pruning tree, and the resulting evaluation process guides our reclustering decisions.

6 EVALUATION

We evaluate our designs on *TPC-H* [30], *DSB* [32], and real-world workload *Mirrors*. Our principal findings are:

(1) **Incremental Reclustering** is more efficient and practical than full table repartitioning, achieving effective partition pruning at lower reclustering cost.
(2) **Workload-Aware Incremental Reclustering** consistently outperforms baselines, delivering significant reductions in I/O volumes and overall costs.
(3) **WAIR** remains robust across diverse data distributions, continuous ingestion, and evolving workload patterns.

6.1 Workload Generator

Standard benchmarks (e.g., TPC-H) largely rely on static datasets with fixed query templates and limited dynamic features. We therefore develop a workload generator to extend standard benchmarks with a set of configurable parameters to capture the interplay between continuous data ingestion and evolving workload patterns. The generator divides a workload into consecutive **periods**. Each period is configured to represent a specific workload pattern and consists of a sequence of workload **batches** (e.g., 12 batches per period). Each batch includes a data ingestion phase, a query execution phase, and an optional reclustering phase[5].

Data Ingestion. For static benchmarks such as TPC-H, we slice the dataset into chronological batches (e.g., grouped monthly by order date), preserving the original data ordering. For DSB and Mirrors (our real-world dataset), because their datasets have built-in timestamps, we use "refresh runs" in DSB and "replay logs" in Mirrors to generate the ingestion batches. An initial, configurable number of ingestion batches forms the initial state of the database.

Query Mix. For each batch, the workload generator constructs a **query mix** derived from the benchmark's query templates. A query mix can cover the complete set of query templates or target specific subsets (e.g., join-heavy queries). We introduce fine-grained control over queries' **selectivity** and data access patterns. For example, the query predicates can follow a specific **distribution** (e.g., uniform, skewed towards recent). Within each period, we use the **shifting rate** to control the percentage of queries regenerated between consecutive batches. Between periods, the main **predicated column** may switch to represent a major workload shift (e.g., from ship date to commit date in TPC-H).

6.2 Workloads

We describe how the above workload generator applies to TPC-H, DSB, and Mirrors.

[5]Reclustering is invoked after each batch but may decide to take no action.

6.2.1 TPC-H. We generate a TPC-H dataset with a scale factor of 720. We divide the dataset into 72 monthly batches grouped by (`o_orderdate`)[6]. Reclustering targets the largest fact table `lineitem`. Each batch comprises all 8 query templates that filter on `l_shipdate`. The default predicate selectivity is set to a two-month interval. Each batch runs: 1) a `global` workload of the 8 queries with uniformly sampled predicates across the entire date range, and 2) a `local` workload of the 8 queries following a Zipf ($\alpha = 2$) distribution skewed toward recent months.

The workload includes a total of 6 periods (**P1** to **P6**), each consisting of 12 consecutive batches. The first two periods **P1**, **P2** form the initial data pool. The workload maintains a shifting rate of 25% in **P2** and **P3** (**P2** for warm up). Reclustering begins from **P3** and continues thereafter. Workload shifting rate rises to 75% from **P4**. To mimic a major workload shift, the primary predicate column changes to `l_commitdate` in **P5**, and to a 2:1 mix of `l_shipdate` and `l_commitdate` in **P6**.

6.2.2 DSB. Compared to TPC-H, DSB has more complex schema with multiple fact tables. DSB's query templates target (i) complex joins (averaging a degree of 10.8), (ii) more filters on diverse table columns, and (iii) LIMIT queries [32].

We follow settings similar to the TPC-H setup above. The workload consists of 6 periods, each containing 12 batches. An initial dataset (SF=240) and these 72 "refresh runs" ingestions bring the final scale factor to around 720. The first two periods (**P1**, **P2**) expand the initial data pool. Reclustering begins from **P3** and targets the three largest fact tables: `store_sales`, `catalog_sales`, and `web_sales`. Each batch includes all 30 DSB queries that filter on these tables. Predicate values are sampled from DSB's default Gaussian ($\sigma = 2$) distribution, with dates centered on a reference month that advances by one month per batch. We set the workload shifting rate to 100% and fix query selectivity at a two-month interval.

6.2.3 Mirrors. Our real dataset, called **Mirrors**, comprises access logs collected from a major open-source software mirror [23]. Spanning the last 120 days, it contains 2.4TB of raw web server logs, which compress to a total of 212GB. Each record includes source address, user agent, access path, response status code, response size, and multiple access related information fields[7] [24].

Massive suspicious IP addresses are identified and blocked by the web firewall. The dataset is paired with an internal workload derived from user tickets requesting IP unblocking. To address these requests, we investigate the access logs across a range of associated IP addresses to determine if the flagged addresses are indeed safe. The workload involves queries for distinct IP counts, user agent counts, access path frequencies, and the distinct count of user agents associated with distinct IP addresses. These analyses are performed at the `/16` subnet level for IPv4 addresses and `/32` for IPv6. Over the 120-day period, 1,210 unique IP addresses with their corresponding ranges are queried for potential unblocking, with an overall workload selectivity of 0.1%. Reclustering begins on day 30, using the initial 30 days of data as a natural pool.

6.3 Experimental Setup

We compare **WAIR**[8], our workload-aware approach against four alternate approaches described in Section 2:

[6]The maximum gap between `l_shipdate` and `o_orderdate` is extended from TPC-H default of 121 days to 1,000 days accordingly to create a larger mismatch between ingestion order and query predicates.

[7]An obfuscated example: 117.176.220.183 - - [29/Sep/2024:10:00:31 +0800] "GET /pypi/web/packages/f6/ab/c7d5e79d2984001911d864af8ec74492da5dba558737b10774ce27587239/duckdb-1.1.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl HTTP/1.1" 200 20097722 "application/octet-stream" "-" "poetry/1.8.3 CPython/3.10.15 Linux/5.15.153.1-microsoft-standard-WSL2" - https

[8]with the standard cost model and a hybrid layout as default.

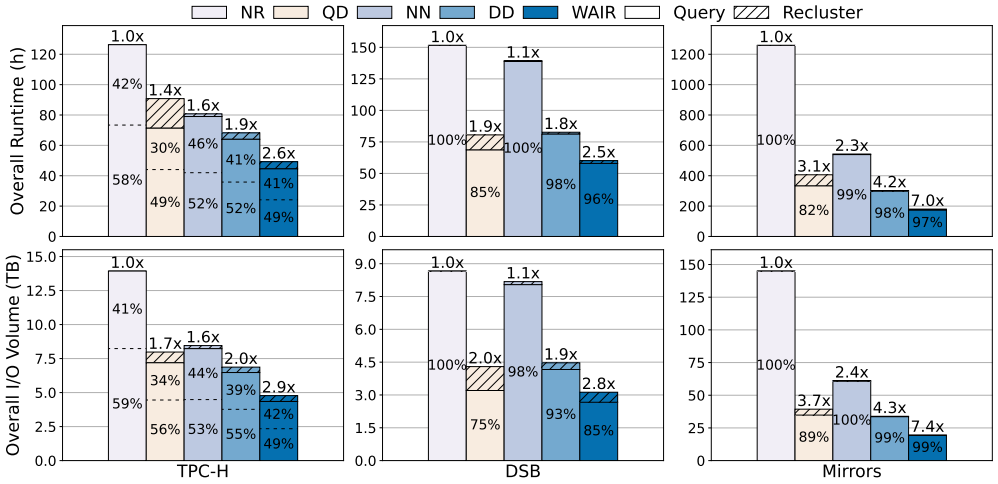


Fig. 10. End-to-End Results. Each bar shows total runtime or I/O volume per method on TPC-H, DSB and Mirrors; numbers above bars indicate their overall speedup relative to NR. Bars are split into query (solid) and reclustering (hatched); for TPC-H, query is further divided into local- (lower) and global- (upper) components.

(1) **No Recluster (NR)**: A simple baseline that keeps the data untouched in its original ingest order without applying any reclustering.
(2) **Qd-tree (QD)**: Build a Qd-tree from historical access patterns and then fully repartition the table at the start of TPC-H and DSB periods, and the end of months in the Mirrors workload.
(3) **New Data New Cluster (NN)**: A Delta Lake-style approach that sorts newly ingested data in each batch into stable *ZCubes*, leaving existing data unchanged.
(4) **Data-Driven Incremental (DD)**: We choose Dremio to represent the data-driven incremental approach. Dremio uses Snowflake's overlapping metric to rank partitions and, in each batch, reclusters up to a fixed number of partitions whose overlapping depth exceeds a predefined overlapping threshold.

All data are stored as 32MB Parquet files[9] with a single row group per file. To ensure a fair comparison, we evaluate all approaches under the same cloud configuration and random seed. Once a method selects partitions and the sort order for reclustering, the identical reclustering code is applied. We omit control-plane overheads such as overlapping metrics costs and Qd-tree construction; for full-table repartitioning, we assume local memory is sufficient to hold the entire table. We strengthen the baselines that require a predefined sort order (NN, DD) by providing an oracle key that is most suitable for each period: for TPC-H, `l_shipdate` in *P*3, *P*4, `l_commitdate` in *P*5, a Hilbert-curve combination of the two in *P*6; for DSB, `(ss|cs|ws)_sold_date_sk`. We tune DD via grid search and select the optimal threshold at an overall reclustering cost comparable to WAIR, ensuring a fair, budget-matched comparison.

All experiments are conducted on the AWS cloud platform [5], using EC2 instances as compute nodes and S3 Express One Zone as the object storage. For TPC-H and DSB, we use `m8gd.8xlarge` instances, because smaller instance types provide unstable bandwidth. For Mirrors, we use larger `m8gd.16xlarge` instances to accommodate the increased data volume. All data transfers between EC2 and S3 are routed through S3 gateway endpoints.

[9]We repeated with various partition sizes and observed consistent results.

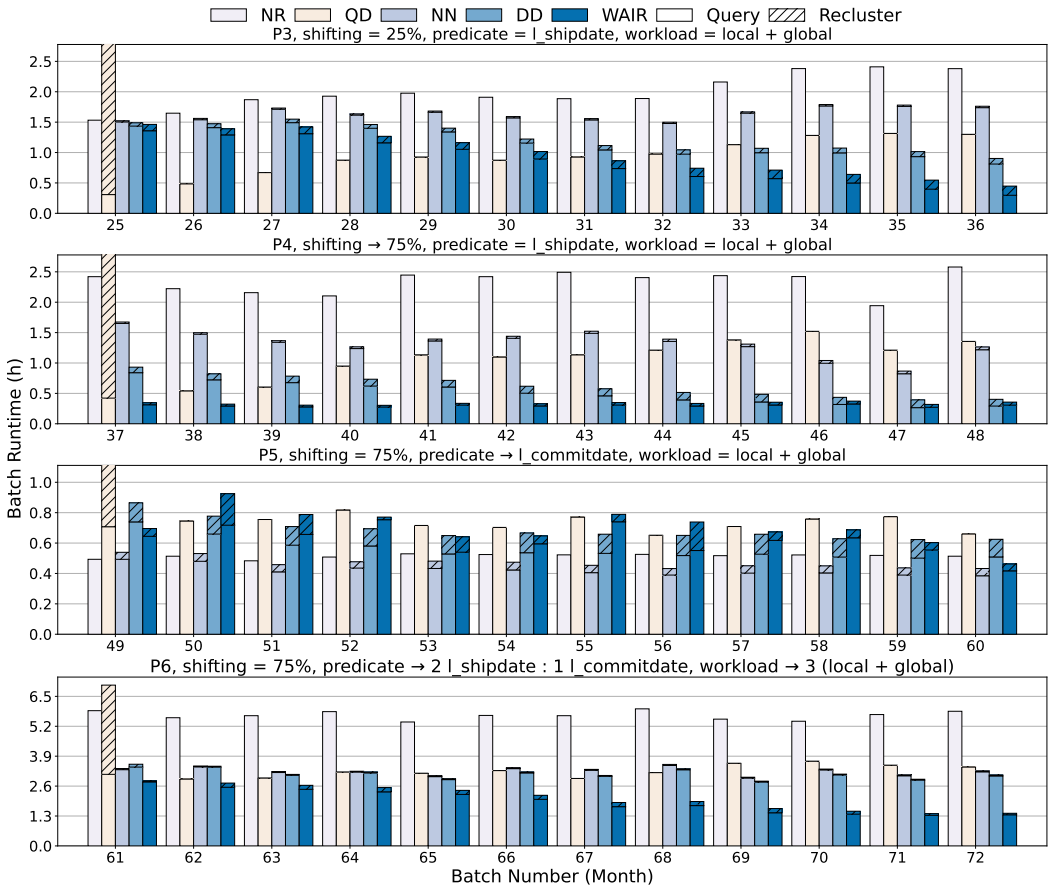


Fig. 11. TPC-H Runtime Breakdown. Each row corresponds to a period. Within each row, solid bars show the monthly query cost and hatched overlays indicate the previous month's reclustering cost. Row titles summarize the period's primary settings.

We use CPU time as a *unified metric* to assess the end-to-end cost trade-offs (discussed in Section 2). We then break the total runtime into query execution time and reclustering time for detailed analysis. We also report the average pruning rates as a stable metric to evaluate across different cloud settings.

6.4 End-to-End Results

Figure 10 summarizes the end-to-end results. Across all workloads, WAIR consistently outperforms the baselines. Compared to NR, WAIR reduces the cumulative runtime by 61.1% on **TPC-H**, 60.3% on **DSB,** and 86.2% on **Mirrors**. The I/O volume closely matches the runtime results. Although QD spends a significant portion of runtime on full-table repartitioning at the beginning of each period, its query execution is still slower than WAIR because QD is unable to adapt to the minor workload shifts within a period. DD incurs almost identical reclustering overhead as WAIR, but the query-time saving is much smaller due to its lack of workload awareness. We next analyze the end-to-end results of each benchmark in detail with further breakdowns and sensitivity analysis over key parameters.

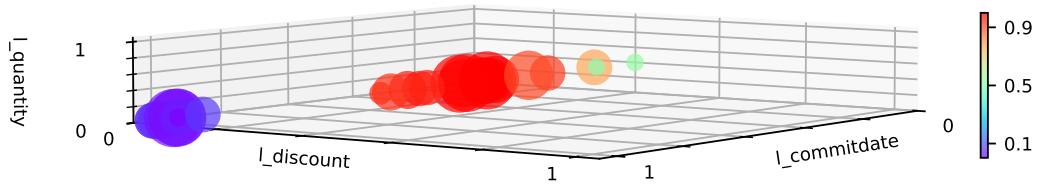


Fig. 12. Centroids of Reclustered Partitions. Each point denotes a centroid identified across all evaluation periods in TPC-H. Point size reflects the number of partitions sharing that centroid. Color encodes the `l_shipdate` coordinate value.

6.5 TPC-H

6.5.1 ***Per-Period Breakdown.*** Figure 11 breaks down TPC-H's runtime on a per-period, per-batch basis, allowing us to trace how well each algorithm reacts to the four evaluation periods.

Period 3 (mild drift; warm-up). Modest workload variation warms up the methods. Incremental methods (NN, DD, WAIR) improve steadily with low per-batch cost. Full repartitioning (QD) pays a large one-time rewrite, starts strong, then degrades as the workload evolves. WAIR's cost model detects a highly disordered pool and invests more reclustering, yielding large immediate gains.

Period 4 (higher variability; same predicate). WAIR's cost model and sliding window converge, scaling back maintenance and focusing on critical partitions. WAIR sustains the best query cost with lower reclustering overhead. QD incurs similar overhead yet its performance drops sharply, making it hard to recoup the costly repartition. DD and NN are not workload-aware; they keep a fixed maintenance budget and deliver smaller gains than WAIR. Even when DD eventually approaches WAIR's query performance, it does so with higher reclustering cost both per batch and in aggregate.

Period 5 (predicate shift). The primary predicate switches to `l_commitdate`. Because ingestion now aligns more with the new predicate (30-90 day instead of prior 1000-day mismatch), NR and NN benefit "for free." Since QD's training did not consider the new column, its heavy rewrite yields little improvement. Both DD and WAIR recluster more in response to the shift, causing a temporary dip. DD discards its earlier metrics and reclusters broadly, while WAIR targets boundary partitions. Guided by its cost model and sliding window, WAIR adapts quickly in a sharp workload shift.

Period 6 (mixed predicates). With a 2:1 mix of `l_shipdate` and `l_commitdate`, all methods face challenges. QD continues to degrade. DD and NN adopt a Hilbert-composed key but cannot adequately separate competing access patterns and serve the mixed workload, limiting gains despite reclustering cost. WAIR maintains its performance advantage by switching to a hybrid layout that quickly places centroids balancing both columns (see Figure 12).

6.5.2 ***Workload Characteristics.*** We then analyze the strengths and weaknesses of each method by comparing their different query costs under the recency-skewed `local` workload and the uniform `global` workload in TPC-H. As shown in Figure 10, DD's gains on `local` are smaller than its gains on `global`. This is because DD treats the dataset uniformly without workload awareness. It fails to prioritize recent partitions that dominate local queries. NN achieves comparable `local` performance to DD but performs worse on `global`. Each batch is sorted in isolation, producing many disjoint "sorted runs" that fragment the table and gradually degrade clustering quality. Because NN never reclusters existing data, the initial pool remains unordered, resulting in poor performance on the global workload. QD excels on the global workload because that workload is uniformly distributed and repeats over time, making a costly one-time full repartition beneficial. However, a Qd-tree trained on historical statistics quickly becomes stale on `local`, which is biased toward recent

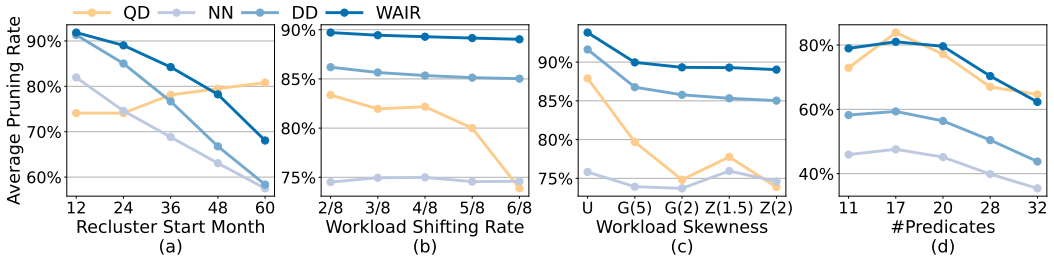


Fig. 13. Sensitivity Study. Each subplot varies one key parameter while holding the others at their default values. U indicates a Uniform distribution. G indicates a Gaussian distribution with σ, and Z indicates a Zipfian distribution with α.

data. In contrast, WAIR adapts gracefully to both workload profiles by continuously adjusting its reclustering strategy to match observed query patterns, delivering balanced and robust performance in a mixed workload.

6.5.3 ***Sensitivity Analysis.*** We then use our workload generator to perform controlled sensitivity analysis on the following key parameters. The default workload follows the same modified **P4** settings for each of the 6 periods: predicate = `l_shipdate`, shifting rate = 75%, workload = `local`. We vary one parameter at a time while holding the others at their default values.

Reclustering Start Time. As shown in Figure 13 (a), all methods but QD perform better when reclustering starts earlier because a delayed start time leads to a larger unclustered data pool that degrades query performance. In contrast, because QD reclusters the entire table, it benefits slightly from a delayed start time due to more historical query data. WAIR remains competitive even when 70% of the data has already been ingested without clustering.

Workload Shifting Rate. As shown in Figure 13 (b), QD's pruning rate drops dramatically as the shifting rate increases because its static partitioning quickly becomes stale. NN and DD are less affected because they are data-driven. WAIR, despite its workload awareness, remains robust because its cost model tracks the shifts and adapts the reclustering accordingly.

Workload Skewness. A skewed workload favors recently ingested data, making it harder to maintain effective clustering. As shown in Figure 13 (c), QD's pruning rate drops sharply even under mild skewness because its static partitioning cannot adapt. WAIR exhibits slightly smaller pruning rate regression than DD under highly skewed workloads because WAIR continuously adjusts its clustering strategy according to the workload skewness.

6.6 DSB

Although DSB is far more complex than TPC-H, WAIR consistently outperforms the baselines as the workload evolves, as shown in Figure 14. These results validate WAIR's pruning effectiveness on the complex workloads. Although more non-leaf operators (e.g., join, aggregate) in a query plan add compute rather than I/O complexity directly, they are also likely to involve more predicates and columns from different base tables (and thus more pruning opportunities). WAIR's savings signatures and hybrid layouts can help identify the most effective base column(s) for reclustering.

Figure 14 (right) also reports the distribution of WAIR's pruning rate per query template grouped by (1) number of base tables involved, and (2) number of predicated attributes per table. WAIR consistently achieves a high pruning rate as the query complexity increases (i.e., more joins and diverse predicates). The outliers (e.g., Q23) typically do not have effective range filters applied on the base tables, leaving limited pruning opportunities.

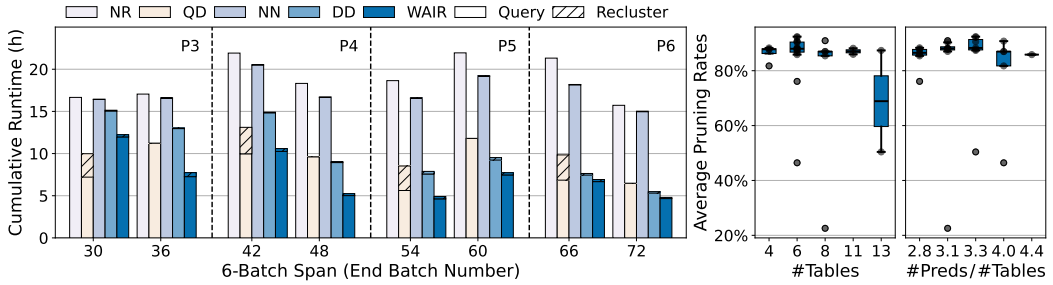


Fig. 14. DSB Breakdown. Left: Cumulative 6-batch runtime. Right: Box plots of query pruning rates in WAIR, grouped by (1) the number of distinct base tables and (2) the number of distinct predicate attributes per table.

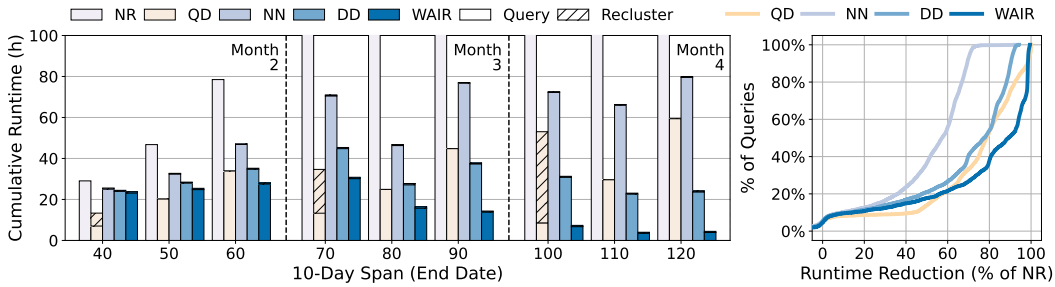


Fig. 15. Mirrors Breakdown. Left: Cumulative 10-day runtime. Right: ECDF of per-query runtime reduction percentages relative to NR.

Leveraging our workload generator, we conduct a sensitivity analysis by varying the average number of distinct predicated columns per query. As shown in Figure 13 (d), the pruning rates for all methods follow similar trends as more predicated columns are included in each query. WAIR outperforms NN and DD by a large margin because WAIR is able to identify the most effective base columns and apply hybrid layouts to improve pruning efficiency. QD also performs well in this analysis because there are sufficient query statistics to train a complex learned layout.

6.7 Mirrors

While synthetic benchmarks provide controlled environments, the Mirrors scenario represents the chaotic reality of production cloud data warehouses. Derived from 2.4 TB of web access logs, this workload is characterized by *extreme data skew* and *highly selective access patterns* (e.g., risk checks on specific IPs or narrow time windows). Mirrors yields an average selectivity of only 0.1%. This offers a rigorous test for reclustering decisions in "needle-in-a-haystack" real-world scenarios.

As shown in Figure 15, WAIR's performance advantage over the baselines increases as the workload execution proceeds because WAIR continuously reclusters the most critical boundary micro-partitions adaptive to the workload. The data-driven approach (i.e., DD), on the other hand, makes suboptimal reclustering decisions in this skewed, highly selective real workload due to the lack of workload awareness. QD struggles in this scenario because a static layout optimized for yesterday's traffic often fails to effectively prune today's IP queries. Figure 15 (right) shows the ECDF of each method's speedup relative to the NR baseline. WAIR makes more than 60% of the queries execute at least 80% faster than the NR baseline with negligible reclustering overhead.

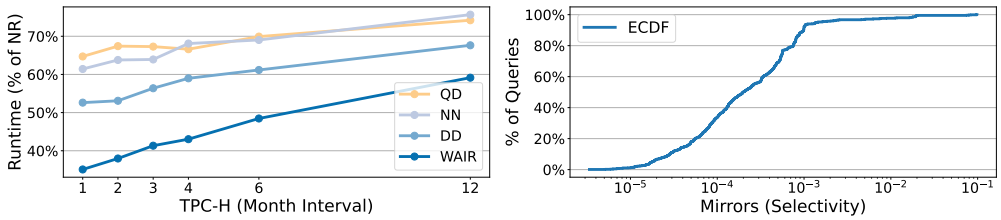


Fig. 16. Selectivity Study. We vary query selectivity by adjusting the TPC-H predicate window and report performance relative to NR. The Mirrors workload exhibits low selectivity.

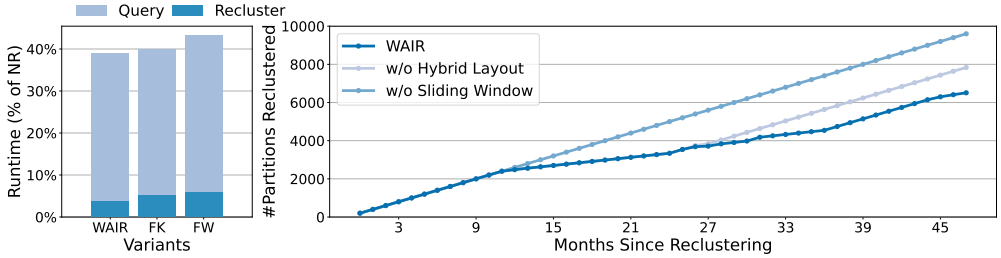


Fig. 17. Ablation Study on WAIR. Divided runtime and reclustering intensity for WAIR and its ablated (FK: fixed-key, FW: fixed-window) variants on the TPC-H workload.

6.8 Selectivity Analysis

We provide a sensitivity analysis on the predicate selectivity using the TPC-H workloads. As shown in Figure 16 (left), WAIR consistently outperforms all the baselines, and the performance advantages grow as the selectivity decreases. This is because WAIR's cost model adapts to selectivity to balance query and reclustering costs, while the baselines are unaware of selectivity. WAIR's dominance in the Mirrors workload is partly because of the low-selectivity nature of the queries. Figure 16 (right) presents the ECDF of the selectivity of Mirrors' queries.

6.9 Ablation Study and Optimality Gap

We test WAIR's resilience to continuous ingestion and workload drift by comparing it with two ablated variants. Figure 17 shows that WAIR consistently outperforms these variants, and WAIR's reclustering cost fluctuates over time as it adapts to workload shifts. When hybrid layouts are disabled, the fixed-key policy cannot adapt to shifting predicates, resulting in degraded query performance even with higher reclustering cost. Replacing the adaptive sliding window with a static policy reclusters the same, fixed number of partitions in each batch. This variant reclusters far more data than necessary, incurring excessive reclustering costs with marginal query improvements.

We evaluate WAIR's "aggressiveness" extension (described in Section 4.3) against its standard configuration using the TPC-H workload. As shown in Figure 18, the aggressive variant ($\alpha = 0.5$, relaxed budget $c = 10$) yields greater cost reductions during early unclustered and workload-stable batches. However, as workload shifts intensify at **P5** and **P6**, it incurs two significant cost spikes. The overhead from heavy reclustering is not fully amortized by subsequent savings. Although the aggressive variant improves overall query performance by 9.5% with comparable total costs, the aggressive settings introduce non-trivial tuning complexity and excessive volatility (20% more cost fluctuations in Figure 18), which is undesirable for production environments.

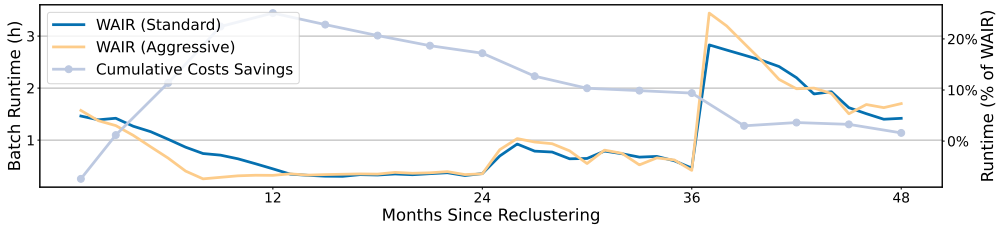


Fig. 18. Aggressiveness Extension on WAIR. A more aggressive setting ($\alpha = 0.5$, $c = 10$) in Section 4.3 is evaluated against the standard WAIR. The figure reports per-batch runtime and the cumulative cost reduction relative to the standard WAIR.

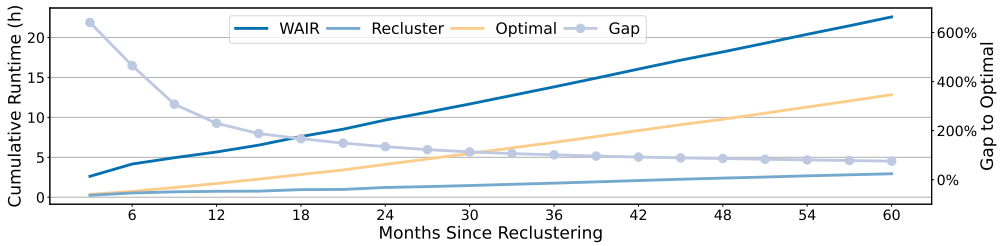


Fig. 19. Optimality Gap. Relative gap between WAIR's cumulative cost and the oracle's query-only cost. Evaluated under the same TPC-H setup with a fixed predicate column (`l_shipdate`).

We further evaluate WAIR against a theoretical oracle in which the table is fully sorted before query execution in Figure 19. The relative gap narrows from roughly 640% in the early batches to cumulatively 75% by the end, at only 12.9% overall reclustering cost. These findings indicate that WAIR achieves near-optimal query performance at a fraction of the maintenance overhead, and WAIR's sliding window and hybrid layout mechanisms are essential for maintaining high performance under dynamic workloads.

7 CONCLUSION

This paper advocates a clean separation between *reclustering policy* and *clustering-key selection*, classifying prior work to highlight their inflexibility. We formalize boundary micro-partitions and prove that reclustering them yields near-optimal pruning with bounded logarithmic amortized overhead. Building on this, we present the WAIR algorithm and implement it into a prototype automatic reclustering service. WAIR uses a sliding window of recent queries and a cloud-cost model to select high-payoff boundary partitions and reorganize them into hybrid layouts with per-group keys that maximize expected savings. Across TPC-H, DSB, and a large real-world workload, WAIR cuts total cost by up to 61%, 60%, and 86%, respectively, outperforming research prototypes and documented commercial baselines. Breakdowns and sensitivity analyses show these gains are robust under dynamic conditions. In short, WAIR turns continuous, cost-effective reclustering into a practical reality for modern cloud data warehouses.

Acknowledgments

This paper was supported by the National Natural Science Foundation of China (Grant No. 62532001), Xiongan AI Institute, and Shanghai Qi Zhi Institute. We would also like to thank Jiaoyi Zhang, Miao Wang, and Yihao Liu for their helpful discussions and guidance.

References

[1] 2025. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing.
[2] 2025. Amazon Redshift. https://aws.amazon.com/redshift/.
[3] 2025. Amazon Redshift Vacuuming Tables. https://docs.aws.amazon.com/redshift/latest/dg/t_Reclaiming_storage_space202.html.
[4] 2025. Amazon S3 pricing. https://aws.amazon.com/s3/pricing/.
[5] 2025. Amazon Web Services. https://aws.amazon.com.
[6] 2025. Apache Arrow. https://arrow.apache.org.
[7] 2025. Apache Iceberg. https://iceberg.apache.org.
[8] 2025. Apache Iceberg In-Place Table Evolution. https://iceberg.apache.org/docs/latest/evolution.
[9] 2025. Apache ORC. https://orc.apache.org.
[10] 2025. Apache Parquet. https://parquet.apache.org.
[11] 2025. Databricks. https://www.databricks.com.
[12] 2025. Databricks Automatic Liquid Clustering. https://www.databricks.com/blog/announcing-automatic-liquid-clustering.
[13] 2025. Databricks Liquid Clustering. https://www.databricks.com/blog/announcing-general-availability-liquid-clustering.
[14] 2025. Delta Lake. https://delta.io.
[15] 2025. Delta Lake Liquid Clustering. https://github.com/delta-io/delta/issues/1874.
[16] 2025. Delta Lake Liquid Clustering Design Doc. https://docs.google.com/document/d/1FWR3odjOw4v4-hjFy_hVaNdxHVs4WuK1asfB6M6XEMw.
[17] 2025. Distribution Advisor in Azure Synapse SQL. https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/distribution-advisor.
[18] 2025. Dremio. https://www.dremio.com.
[19] 2025. Dremio Apache Iceberg Clustering. https://www.dremio.com/blog/dremios-apache-iceberg-clustering-technical-blog/.
[20] 2025. Redis. https://redis.io.
[21] 2025. Snowflake. https://www.snowflake.com.
[22] 2025. Snowflake Clustering Keys and Clustered Tables. https://docs.snowflake.com/en/user-guide/tables-clustering-keys.
[23] 2025. Tsinghua Open Source Mirror. https://mirrors.tuna.tsinghua.edu.cn.
[24] 2025. Tsinghua Open Source Mirror Logs. https://mirrors.tuna.tsinghua.edu.cn/logs/neomirrors.
[25] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216. doi:10.14778/3415478.3415545
[26] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. doi:10.1145/3514221.3526045
[27] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant Mcalister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (2024), 4038–4050. doi:10.14778/3685800.3685825
[28] Suratna Budalakoti, Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Ramarajan Krishnamachari, and Alan Wood. 2024. Automated Clustering Recommendation With Database Zone Maps. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 68–79. doi:10.1145/3626246.3653397
[29] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2022. *Introduction to Algorithms, fourth edition.* MIT Press.
[30] The Transaction Processing Council. 2024. TPC-H Benchmark (Revision 4.0.0). (2024).
[31] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. doi:10.1145/2882903.2903741

[32] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[33] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 55–67. doi:10.1145/3626246.3653379

[34] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 418–431. doi:10.1145/3448016.3457270

[35] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-Native Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 36, 12 (2024), 7772–7791. doi:10.1109/TKDE.2024.3397508

[36] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. doi:10.14778/3611479.3611486

[37] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad (Eds.). AAAI Press, 226–231. http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[38] P. Hall. 1987. *On Representatives of Subsets.* Birkhäuser Boston, Boston, MA, 58–62. doi:10.1007/978-0-8176-4842-8_4

[39] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL server column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1159–1168. doi:10.1145/2463676.2463708

[40] Jonathan K. Lawder and Peter J. H. King. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD Rec.* 30, 1 (2001), 19–24. doi:10.1145/373626.373678

[41] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. doi:10.14778/3415478.3415568

[42] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[43] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

[44] Mark Raasveldt. 2022. DuckDB - A Modern Modular and Extensible Database System. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zaït (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/Keynotes/Abstract_MarkRaasveldt.pdf

[45] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[46] Pierangelo Rosati, Frank Fowley, Claus Pahl, Davide Taibi, and Theo Lynn. 2018. Right Scaling for Right Pricing: A Case Study on Total Cost of Ownership Measurement for Cloud Migration. In *Cloud Computing and Services Science - 8th International Conference, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1073)*, Víctor Méndez Muñoz, Donald Ferguson, Markus Helfert, and Claus Pahl (Eds.). Springer, 190–214. doi:10.1007/978-3-030-29193-8_10

[47] Chiara Rucco, Antonella Longo, and Motaz Saad. 2024. Optimizing Data Ingestion for Big Data: A Cloud-Based Design Pattern Approach. In *IEEE International Conference on Big Data, BigData 2024, Washington, DC, USA, December 15-18, 2024*, Wei Ding, Chang-Tien Lu, Fusheng Wang, Liping Di, Kesheng Wu, Jun Huan, Raghu Nambiar, Jundong Li, Filip Ilievski, Ricardo Baeza-Yates, and Xiaohua Hu (Eds.). IEEE, 3556–3561. doi:10.1109/BIGDATA62323.2024.10825970

[48] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[49] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland,*

OR, USA], June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 193–208. doi:10.1145/3318464.3389770

[50] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[51] Eleni Zapridou, Ioannis Mytilinis, and Anastasia Ailamaki. 2022. Dalton: Learned Partitioning for Distributed Data Streams. *Proc. VLDB Endow.* 16, 3 (2022), 491–504.

[52] Huanchen Zhang, Yihao Liu, and Jiaqi Yan. 2024. Cost-Intelligent Data Analytics in the Cloud. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p78-zhang.pdf

[53] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Janaki Lahorani, Dmitry Potapov, and Murali Krishna. 2017. Dimensions Based Data Clustering and Zone Maps. *Proc. VLDB Endow.* 10, 12 (2017), 1622–1633. doi:10.14778/3137765.3137769

[54] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working Smarter, Not Harder. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 757–770. doi:10.1145/3722212.3724447

Received October 2025; revised January 2026; accepted February 2026