# AutoLiquid: Autonomic Data Layout Optimization for the Databricks Lakehouse

Supun Nakandala*, Naga Raju Bhanoori*, Yunjia Zhang, Parimarjan Negi, Ankur Sharma, Eric Liang†
Cindy Jiang, Sirui Sun, Terry Kim, Mostafa Mokhtar, Vijayan Prabhakaran, Bart Samwel
Sunitha Beeram, Siddharth Taneja, Amir Hormati, Amit Shukla, Michalis Petropoulos, Reynold Xin
vldb2026@databricks.com
Databricks, Inc., USA

## ABSTRACT

Data layout recommendation is a classic problem in database literature, yet the downstream challenge of autonomously applying these changes in live production systems is less frequently studied. In Lakehouse architectures, selecting the right clustering key is critical for data skipping performance, but remains a manual process that cannot scale to the hundreds of millions of tables managed by platforms like Databricks. We present AutoLiquid, an autonomic system that solves this by continuously selecting, verifying, and applying clustering keys for Liquid Clustering tables, requiring only a single "`CLUSTER BY AUTO`" declaration from the user.

To navigate the tension between adapting to unpredictable workloads and maintaining strong performance guarantees against regressions, AutoLiquid leverages the decoupled storage and compute architecture of the Lakehouse. It utilizes lightweight heuristics over collected scan telemetry to identify promising candidate keys, followed by a fast shadow verification process that measures actual pruning benefits on sampled data. This "verify before commit" approach ensures that only decisions with demonstrated improvement are applied. We present the algorithm and end-to-end verification flow, providing empirical results using real customer workload data. When compared to customer-selected clustering keys, AutoLiquid achieves similar or better performance on over 95% of evaluated tables. AutoLiquid is currently deployed in production at Databricks, managing clustering keys for millions of tables.

**PVLDB Reference Format:**
Supun Nakandala, Naga Raju Bhanoori, Yunjia Zhang, Parimarjan Negi, Ankur Sharma, Eric Liang, Cindy Jiang, Sirui Sun, Terry Kim, Mostafa Mokhtar, Vijayan Prabhakaran, Bart Samwel, Sunitha Beeram, Siddharth Taneja, Amir Hormati, Amit Shukla, Michalis Petropoulos, Reynold Xin. AutoLiquid: Autonomic Data Layout Optimization for the Databricks Lakehouse. PVLDB, 19(12): 4023 - 4035, 2026.
doi:10.14778/3827998.3828013


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828013

*Equal contribution.
†Corresponding author.

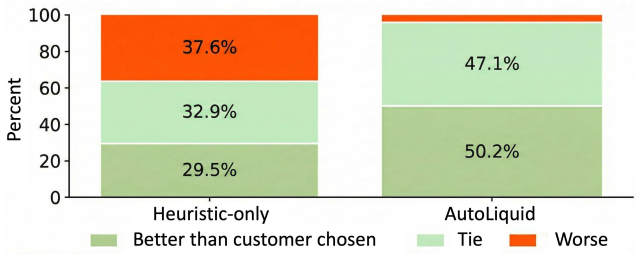



**Figure 1: Fleet-wide effectiveness: AutoLiquid matches or outperforms customer-selected keys in >95% of cases (green segments).**

## 1 INTRODUCTION

The Databricks Lakehouse Platform [4] unifies the traditionally separate paradigms of data lakes and data warehouses into a single architecture. It inherits the low-cost, open-format storage of data lakes (e.g., Parquet [16] and Delta Lake [21]) while providing the transactional guarantees, including ACID compliance, schema enforcement, and governance, that were historically exclusive to data warehouses. A key enabler of query performance in this architecture is *data skipping*: by maintaining file-level statistics, the query engine can prune files whose value ranges do not overlap with a query's predicates, avoiding unnecessary I/O, especially for large data sets.

However, the effectiveness of data skipping depends critically on the *physical layout* of the data, i.e., how values are co-located across files. Managing these layouts is difficult: as workloads evolve, the optimal clustering strategy changes, and layout decisions such as which columns to cluster on, when to re-cluster existing data, and when to disable clustering altogether must be revisited continuously. To simplify this, Databricks introduced **Liquid Clustering** [14], a dynamic data layout technique that replaces traditional partitioning (such as Hive-style partitioning) and Z-ordering. Liquid Clustering uses adaptive KD-tree structures to incrementally reorganize data (Section 2.2), enabling efficient data skipping across multiple columns without requiring full table rewrites. Importantly, the *clustering key* determines which queries benefit from pruning. A well-chosen key can reduce I/O by orders of magnitude, while a poorly chosen one provides little benefit while incurring unnecessary data ingestion costs from reorganization.

In many systems, selecting clustering keys remains a manual process. Database administrators analyze query workloads, identify

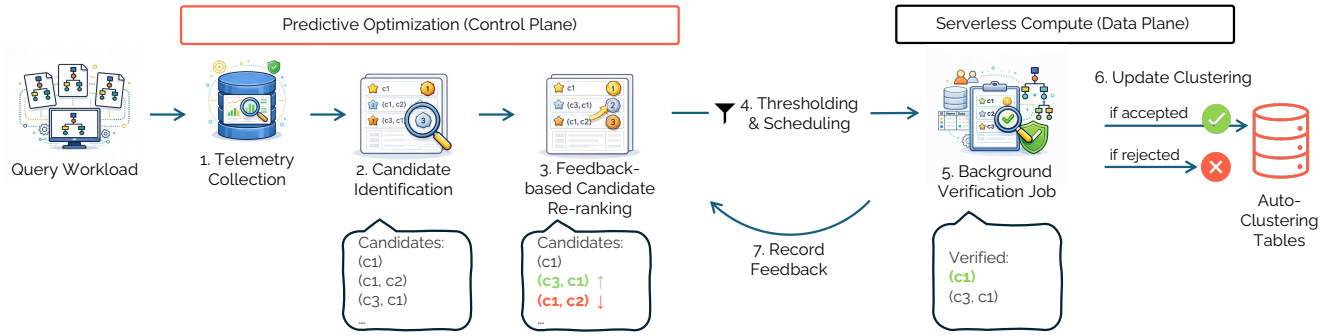



**Figure 2: Closed-loop architecture: the AutoLiquid pipeline integrates lightweight telemetry collection (Stage 1), heuristic candidate generation (Stage 2—4), and shadow verification (Stage 5) into a continuous feedback loop that adapts to workload changes. Verified candidates trigger clustering updates (Stage 6), while feedback from verification loops back to refine future candidate ranking (Stage 7), ensuring the system adapts to estimation errors over time.**

candidate columns, and issue commands to update the data layout, such as `ALTER TABLE ... CLUSTER BY (col1, col2)`. While this works for a handful of static tables, it does not scale and requires much manual effort: the Databricks platform alone manages hundreds of millions of tables, the vast majority of which receive no manual tuning. Existing database tuning advisors [2, 27, 30] can recommend layout changes, but they usually function as passive tools. They suggest actions for a human to evaluate and apply, without the feedback loops necessary to continuously adapt to evolving workloads or to guarantee that their recommendations will not cause regressions.

Transitioning from passive recommendation to fully autonomous layout management introduces challenges that go beyond candidate identification:

- **Estimation Inaccuracy.** Heuristic estimates based on selectivity statistics are inherently approximate. They struggle to account for data skew, correlated columns, and compound predicates. A heuristic may predict substantial pruning improvement for a candidate key, yet the actual benefit may be negligible once the data is physically reorganized. In an autonomous system, such mis-estimations lead to wasted compute and reduced user trust.
- **Asymmetry of Risk.** The cost-benefit landscape of layout changes is fundamentally asymmetric. For instance, a 20% aggregate improvement in scan performance is welcome but rarely noticed, whereas a performance regression on even a small fraction of the workload can be treated as a critical incident. An autonomous system must therefore be *defensive*, applying changes only when it has high confidence that they will not cause regressions.
- **Workload Evolution.** Cloud-native workloads inherently change over time. Query patterns fluctuate due to ad-hoc exploration, seasonal business cycles, and the addition of new workloads. An autonomous system must distinguish genuine, sustained shifts in workload patterns from transient noise, and must do so without incurring expensive evaluation jobs on every observed change.

In this paper, we present AutoLiquid, an autonomic system that continuously selects and applies clustering keys. In addition to the traditional `CLUSTER BY (...)` syntax, AutoLiquid introduces:

```
[CREATE|ALTER] TABLE ... CLUSTER BY AUTO
```

which delegates the entire clustering key lifecycle, including selection, verification, application, and ongoing maintenance, to the system. Once a clustering key is selected, the actual physical data reorganization is managed by Predictive Optimization (PO) (Section 2). In this paper, we focus on the clustering column selection problem as we treat the physical clustering execution as an orthogonal problem.

As a motivating example, consider a retail fact table `store_sales` that continuously ingests sales records. Initially, the table is unclustered, meaning data is laid out in its original *ingestion order*. Users begin running daily-reporting queries that filter by `ss_sold_date`. AutoLiquid observes that this filter is highly selective and automatically selects `(ss_sold_date)` as the clustering key. Over time, a second group of analysts begins filtering by `ss_item_sk` for inventory tracking, and AutoLiquid evolves the clustering to the two-column key `(ss_sold_date, ss_item_sk)` without user intervention.

AutoLiquid achieves this through a multi-stage pipeline (Figure 2). The system asynchronously collects scan telemetry from query execution logs over a rolling 30-day window, recording predicate expressions, expression selectivities, file-level pruning rates (files pruned vs. files read), and column projection ratios for each scan operation. This observation window smooths out transient workload noise and ensures that clustering decisions reflect sustained query patterns.

From this telemetry, AutoLiquid generates candidate clustering keys using a lightweight heuristic which takes the estimated selectivities into account with the column projection ratio. Multi-column candidates are formed by enumerating combinations of top single columns and scoring them with a model that accounts for diminishing returns from correlated predicates. Because these heuristic estimates are approximate, the system revises candidate rankings based on past verification results.

The candidates must then meet benefit and hysteresis thresholds before being scheduled for verification. The top-$K$ candidates that pass are batched into a single verification job, where the system samples data from the table, builds KD-tree structures to simulate

each candidate's clustered layout, imputes file-level min/max statistics from the resulting trees, and replays historical predicates against these statistics to measure the actual pruning benefit. This shadow execution produces near ground-truth measurements without actual execution of any user queries, making verification both reliable and cost-effective. Only candidates that demonstrate clear improvement over the current layout are applied. Verification results, including both successes and failures, are then recorded as correction factors that feed back into the re-ranking stage, closing the loop: candidates that verified poorly receive negative adjustments in future iterations, while those that exceeded expectations are promoted, improving estimation accuracy over time.

This design directly targets the challenges identified above. Estimation inaccuracy is mitigated by the "verify before commit" principle: no layout change is applied until shadow verification confirms its benefit on real data. The asymmetry of risk is handled by the defensive multi-stage design, adaptive thresholds that raise the bar for each successive change, and failed candidates leave the table unchanged. Workload volatility is absorbed by a 30-day observation window and feedback-based damping that prevent the system from reacting to transient noise.

AutoLiquid has been deployed in production at Databricks [13], where it manages clustering keys for millions of tables. Fleet-wide data shows that AutoLiquid selected clustering keys achieve comparable or better file pruning than customer selections in more than 95% of cases (light and dark green segments in Figure 1). It significantly outperforms the heuristic approach by identifying false positives that look promising but fail in practice. For a detailed analysis of the accuracy and rejection rates of the verifier, see Section 4.

The contributions of this paper are summarized as follows:

- We present the AutoLiquid algorithm in detail (Section 3), describing how Databricks solves the autonomous clustering key selection problem at scale. The algorithm combines lightweight heuristic candidate identification with feedback-based re-ranking and conservative thresholding to manage millions of tables while minimizing the risk of performance regressions and avoiding unnecessary churn between clustering schemes.
- We introduce a shadow verification method that accurately measures the actual pruning benefit of candidate keys on sampled data before any layout change is committed (Section 3.7). We describe the verification flow, from data sampling and KD-tree construction to predicate replay and benefit measurement.
- We provide an experimental analysis using real customer workload data from the Databricks platform (Section 4). We characterize the properties of production workloads at scale, evaluate the effectiveness of AutoLiquid's key selection against user-managed baselines, and analyze the system's behavior across diverse table sizes and workload patterns.

## 2 BACKGROUND

This section introduces the necessary background and terminology that is used throughout the paper.

### 2.1 System Architecture

The Databricks Lakehouse Platform [4] separates the *Control Plane* (which hosts platform services such as the web application, Unity Catalog [8] for metadata governance, and the Predictive Optimization (PO) infrastructure [12]) from the *Data Plane*, which handles physical data processing using customer-accessible compute. PO operates as an autonomic control loop that monitors workload telemetry and table state to manage table layouts, with decisions made in the Control Plane and physical operations executed as serverless jobs in the Data Plane.

AutoLiquid is built on top of PO. It runs in the Control Plane and extends PO with autonomous clustering-key selection, while PO continues to handle the physical execution of clustering operations. Background verification jobs that evaluate candidate keys run on serverless compute in the Data Plane.

### 2.2 Liquid Clustering

Liquid Clustering is a dynamic data layout technique designed to replace rigid, directory-based partitioning (e.g., Hive-style partitioning) and static Z-ordering. It decouples the data layout from the directory structure and uses KD-trees to co-locate data on multiple columns simultaneously, enabling efficient data skipping without the write amplification of full table rewrites. AutoLiquid operates exclusively over Liquid Clustered tables.

We highlight two behaviors relevant to AutoLiquid:

- **Incremental clustering**: new data is incrementally clustered in the background under the table's *latest* clustering scheme, while prior data remains under its previous scheme. This lets the scheme evolve over time with minimal reorganization cost, e.g., from `(date, category)` to `(date, category, region)` when queries mostly read recent data.
- **Full re-clustering**: for more drastic key changes (e.g., from `(date, category)` to `(region, uuid)`), older data must be reorganized to benefit workloads. AutoLiquid determines the re-clustering strategy based on workload characteristics (Section 3).

We note that, in Databricks, the query optimizer's selectivity estimation relies on per-table statistics that are computed independently of the clustering key. As a result, changing the clustering key does not alter how the optimizer estimates cardinalities or generates query plans. The same independence holds for runtime adaptive query execution (AQE), since scan-operator output cardinalities are determined by the rows satisfying the predicate rather than by which files were pruned. The file-sizing and compression properties of clustered data are likewise managed by Liquid Clustering itself [14], which targets a uniform file-size distribution and produces well-compacted Parquet files as part of the clustering operation. AutoLiquid's choice of clustering key does not directly affect file sizing or compression efficiency beyond what Liquid Clustering already provides. Clustering changes therefore reach the engine only through the data-skipping path, which is the focus of AutoLiquid.

## 3 AUTOMATIC KEY SELECTION ALGORITHM

In this section, we introduce the clustering key selection algorithm in AutoLiquid. The algorithm is built upon the Predictive Optimization (PO) infrastructure at Databricks (Section 2) and is structured

**Table 1: Math notation for AutoLiquid's key selection algorithm.**

| Symbol | Description |
|---|---|
| $T$ | Table managed by AutoLiquid |
| $q$ | A query (scan) over $T$ |
| $K$ | Candidate clustering key |
| $Q_{30d}$ | Scans over the trailing 30-day telemetry window for $T$ |
| $\sigma(q, K)$ | Predicate selectivity on $K$ in $q$, in $[0, 1]$ |
| $ColProj(q)$ | Column projection ratio of $q$, in $[0, 1]$ |
| $\widehat{B}(K)$ | Heuristic benefit estimate for $K$ |
| $B_{\text{verified}}(K)$ | Verified benefit from shadow verification |
| $\delta_K$ | Per-scan feedback correction factor |
| $\Delta\text{ScanCost}_{\text{read}}$ | Estimated read-compute reduction |
| $\text{ClusteringCost}_{\text{write}}$ | Re-cluster write compute |

as a continuous closed-loop pipeline (Figure 2). Table 1 summarizes the notation used throughout this section.

## 3.1 Algorithm Overview

The default optimization objective of AutoLiquid is to optimize for *cost*, rather than focusing only on the latency of individual queries. While per-query latency reduction is a desirable outcome, the primary metric driving our decisions is Total Cost of Ownership (TCO). Therefore, our target is to maximize the net benefit:

$$\text{Objective} = \max\left(\Delta\text{ScanCost}_{\text{read}} - \text{ClusteringCost}_{\text{write}}\right) \quad (1)$$

Here $\Delta\text{ScanCost}_{\text{read}}$ is the estimated reduction in read compute that the candidate layout would deliver on the 30-day historical workload, derived from heuristic benefit estimates. $\text{ClusteringCost}_{\text{write}}$ is the write compute needed to physically cluster the table to that layout.

To achieve this objective, AutoLiquid operates as a continuous closed-loop pipeline of seven stages (Figure 2). The loop runs continuously in the background for every monitored table, so a table's clustering can evolve as its workload evolves over time. The pipeline naturally splits in two: a lightweight *PO control-plane* pre-filter that proposes promising candidates from telemetry alone (Stages 1–4), and a heavier *shadow verification and commit* step that confirms a candidate on real customer data and applies it (Stages 5–7).

*PO control-plane pre-filtering (Stages 1–4).* These stages run continuously on PO services, on telemetry only, with no access to customer data. **Stage 1 (Telemetry Collection)** collects scan telemetry from real query execution logs over a 30-day rolling window $Q_{30d}$, recording column paths, predicates, selectivities, and pruning metrics for each scan. **Stage 2 (Candidate Identification)** applies a lightweight heuristic to generate a small set of single- and multi-column candidate clustering keys that are likely to improve file pruning for future scans. **Stage 3 (Feedback-based Candidate Re-ranking)** adjusts each candidate's heuristic estimate using a correction factor learned from prior verification outcomes, dampening persistent estimation biases. **Stage 4 (Thresholding and Scheduling)** drops candidates that fail to clear adaptive benefit and hysteresis thresholds, and batches the top-$K$ survivors into a single verification job to amortize cost.

*Shadow verification and commit (Stages 5–7).* The batched candidates are sent to a serverless data-plane job that has access to the customer table, and the verification outcome drives the commit and feedback steps back on the control plane. **Stage 5 (Background Verification)** runs on serverless data-plane compute and verifies each batched candidate by replaying historical predicates against shadow KD-trees built on freshly-sampled table data, producing near ground-truth pruning measurements without touching the production layout. **Stage 6 (Update Clustering)** commits the best verified candidate, when it exceeds the current layout's benefit, via an `ALTER TABLE . . . CLUSTER BY . . .` statement and delegates the physical re-layout to liquid clustering. **Stage 7 (Record Feedback)** records the estimated and verified benefits for the chosen candidate, supplying the correction factor consumed by Stage 3 in subsequent iterations and closing the loop.

## 3.2 Telemetry Collection

AutoLiquid relies on a unified telemetry stream, aggregated across all product surfaces (e.g., SQL warehouses, notebooks, and jobs) and Databricks Runtime versions, to guide clustering key selection. It collects column names, data types, predicate expressions, and corresponding selectivity metrics from query scans. This information is used to estimate the potential benefit of different clustering key candidates. In addition to scan-level telemetry, AutoLiquid also collects table write patterns to identify write-intensive tables.

AutoLiquid maintains a trailing 30-day window of query history. Telemetry is collected asynchronously after query execution, adding no overhead to production workloads. For each scan, the system extracts: (1) **predicate expressions**, including column paths, types, and selectivity estimates; (2) **pruning metrics** (files pruned vs. read); (3) **row size estimates**; and (4) the **column projection ratio** to weight benefit calculations. Dynamic filter predicates generated by semi-joins (e.g., IN list filters, and bloom filters) are also included. Currently, AutoLiquid supports analysis of accesses on struct fields, including for nested structs, as well as simple cast expressions. Future versions will add support for pruning on more complex expressions, including JSON fields and array references.

## 3.3 Candidate Identification Heuristic

AutoLiquid uses a lightweight heuristic stage to identify a small set of promising candidates before running background verification jobs. The heuristic leverages the recent scan workload $Q_{30d}$ and table-level statistics to estimate the potential file-pruning benefit of clustering on candidate columns.

***Benefit Estimation.*** For each candidate key $K$, we estimate its clustering benefit by aggregating the expected data skipping improvement across all queries in $Q_{30d}$ that contain predicates on $K$. Specifically, for each qualifying query $q \in Q_{30d}$, we compute the product of two factors: (1) the *pruning potential* $(1-\sigma(q, K))$, where $\sigma(q, K)$ is the estimated selectivity of the predicate on $K$, representing the estimated fraction of data that could be skipped if the table were clustered on $K$; and (2) the *column projection ratio* $ColProj(q)$, which accounts for the proportion of data that is scanned due to column projection. Selectivity is estimated using optimizer statistics when they are present or otherwise using heuristic values. Errors

in these selectivity estimates do not propagate into the final layout decision: the shadow verification stage (Section 3.7) measures pruning benefit directly on customer data and is independent of the optimizer's selectivity estimates. The total benefit is the sum of these per-query contributions:

$$\widehat{B}(K) = \sum_{q \in Q_{30d}} \mathbb{1}[\text{HasPredicate}(q, K)] \cdot (1 - \sigma(q, K)) \cdot ColProj(q) \tag{2}$$

Here $\mathbb{1}[\cdot]$ is the indicator function, so the sum effectively runs only over queries that contain a predicate on $K$. $\sigma(q, K) \in [0, 1]$ is the predicate's selectivity (fraction of rows matching the predicate), and $ColProj(q) \in [0, 1]$ is the column projection ratio, both as defined above.

In this heuristic, a column is a strong clustering candidate if it appears frequently in scan predicates and exhibits a high pruning rate. This heuristic stage is intentionally conservative and low-overhead. We run it in two phases: first we filter columns by predicate frequency to ensure workload coverage, then we apply Eq. 2 to the surviving candidates. The goal is not to provide precise benefit measurements (which would require materializing candidate layouts) but to efficiently filter out low-value candidates and surface a small set of promising keys for the more expensive verification step that follows. To avoid stale-metadata bias, AutoLiquid also drops scan telemetry recorded before any schema-changing event (e.g., a column rename) on the table.

## 3.4 Candidate Ranking

Given the per-column benefit estimates from the heuristic stage, AutoLiquid determines which candidates are selected for the relatively expensive verification. This decision involves comparing each candidate's estimated benefit against a *baseline benefit* that captures the pruning efficiency of the current data layout.

***Baseline Benefit.*** The baseline represents the file-skipping benefit that the table already achieves under its current configuration (liquid clustering). We distinguish two cases:

- *Table has existing clustering columns $K_{curr}$:* We compute the baseline by applying the same benefit formula (Eq. 2) to the current clustering key.
- *Table has no clustering columns:* We fall back to the *observed pruning rate* from historical scans, which reflects the natural data skipping achieved by the table's physical layout (e.g., due to ingestion-time ordering):

$$B_{\text{base}} = \sum_{q \in Q_{30d}} \frac{\text{FilesPruned}(q)}{\text{FilesPruned}(q) + \text{FilesRead}(q)} \cdot ColProj(q) \tag{3}$$

The second case is particularly important: tables with append-only ingestion patterns often exhibit natural temporal ordering, where columns correlated with ingestion time (e.g., columns such as dates and timestamps) already benefit from implicit data locality. Using observed pruning as the baseline ensures that candidates must demonstrate improvement *beyond* what the current physical layout already provides.

***Multi-Column Candidate Generation.*** A clustering key may consist of more than one column. To generate multi-column candidates, the system enumerates all non-empty subset combinations of the top single columns (sorted by weight), up to a maximum of $M = 2$ columns per candidate. For example, given single-column candidates `(a, b, c)` with the default $M$, the system evaluates: `(a)`, `(b)`, `(c)`, `(a, b)`, `(a, c)`, and `(b, c)`.

For multi-column candidates, we extend Eq. 2 by replacing the per-query selectivity $\sigma(q, K)$ with a combined selectivity computed using an *exponential backoff* over the candidate's predicates in $q$. We rank these predicates by selectivity (most selective first) and set $\sigma_{\text{combined}} = \prod_{k=1}^{|P_q|} \sigma_k^{1/k}$, where $\sigma_k$ is the selectivity of the $k$-th most selective predicate. The $k = 1$ predicate contributes its full selectivity, and each successive predicate contributes the $k$-th root of its selectivity, modeling the diminishing pruning gains from correlated columns.

## 3.5 Feedback-driven Re-ranking

The heuristic benefit estimates from Stage 2 may diverge from actual pruning performance due to data skew, correlated predicates, or other factors not captured by selectivity statistics alone, such as physical clustering quality for a specific column. In Stage 3, AutoLiquid adjusts candidate rankings using feedback from any previous verification executions.

For candidates that have been previously verified, the system applies a *correction factor* $\delta_K$ that captures the historical discrepancy between heuristic estimates and verified ground truth. The adjusted benefit estimate becomes:

$$\widehat{B}'(K) = \widehat{B}(K) + \delta_K \cdot |Q_{\text{current}}| \tag{4}$$

where $\delta_K$ is computed as the per-scan difference between verified and estimated benefits from prior verification runs. $Q_{\text{current}}$ denotes the set of scans currently observed in the 30-day telemetry window for table $T$, the same $Q_{30d}$ used in Stage 1. The per-scan correction factor $\delta_K$ is multiplied by $|Q_{\text{current}}|$ to scale the adjustment to the current scan volume.

This feedback-based re-ranking serves two purposes. First, it improves estimation accuracy over time by learning from verification outcomes, candidates that performed better than expected receive a positive adjustment, while those that underperformed receive a negative adjustment. Second, it provides *damping control* to prevent repeated verification attempts on candidates that were previously rejected. A candidate that verified poorly will have its estimated benefit reduced, making it less likely to exceed the verification threshold again.

The correction factor and feedback mechanism are described in detail in Section 3.7, which covers how verification results are recorded and applied (Stage 7).

## 3.6 Thresholding and Scheduling

In Stage 4, we implement specific thresholding formulas as heuristics to prevent unexpected column selection changes. Candidates that pass these thresholds are scheduled for verification.

***Exponential Threshold.*** To ensure that each successive clustering change requires progressively more benefit to trigger, we apply an exponential threshold:

$$B_{\text{new}} \geq B_{\text{current}} + \alpha^n - 1 \tag{5}$$

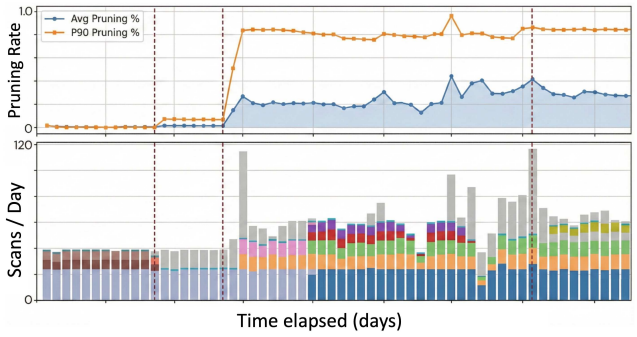



**Figure 3: Thresholds in practice: despite day-to-day volatility in the query mix (bottom), the system maintains the stable layout (top), filtering transient noise until sustained shifts justify adapting the clustering (vertical dashed lines).**

where $\alpha \approx 2$ is the threshold base and $n$ is the number of past clustering column changes. This formula starts with an improvement requirement of 1.0 for the first change, and grows exponentially thereafter (e.g., after 3 changes, the required improvement is $2^3 - 1 = 7$). Figure 3 illustrates the damping effect on a volatile mid-size production table, where the system avoids thrashing despite day-to-day query fluctuations and only adapts when sustained shifts justify it. The base $\alpha \approx 2$ is empirically chosen so that each successive change roughly doubles the required improvement. This balances responsiveness on new tables (where $n = 0$ leaves the bar at 1) against stability on tables that have already adapted several times. In our production tuning, smaller bases admitted excessive churn, while larger bases suppressed legitimate updates on tables that had already changed clustering keys a few times.

***Relative Threshold.*** In addition to the exponential threshold, we require the new benefit to exceed the current benefit by a relative margin of $\theta \approx 1.1$, with the margin made more lenient on tables that have significant headroom to the maximum achievable pruning.

***Multi-Candidate Selection.*** From the ranked set of single- and multi-column candidates that pass both thresholds, the system selects the top-$K$ candidates by adjusted benefit $\widehat{B}'(K)$ in descending order. Evaluating multiple candidates rather than a single best candidate is important for robustness: heuristic estimates can be inaccurate, and testing several promising candidates in a single verification job ensures that the best option is identified even when individual estimates are noisy.

***One-Time Verification.*** For tables that have never been verified before, the system additionally permits a single one-time verification even when none of the heuristic candidates pass the thresholds above. This guards against the heuristic missing a beneficial layout entirely on the first pass, for example because of stale or biased selectivity statistics, by giving each new table at least one opportunity to be evaluated against ground truth on freshly-sampled data.

## 3.7 Shadow Verification

Candidates that pass the thresholding stage are submitted to a verification process that computes reliable pruning benefits before any clustering key change is committed.

***Verification Process.*** Figure 4 summarizes the process. The verification job runs a trace-based simulation that measures the data-skipping efficacy of a candidate key $K$ without re-clustering the table. Because no physical re-cluster is performed, verification is fast: most jobs finish in minutes, and even large tables typically complete within tens of minutes. Verification proceeds through four steps that produce near ground-truth pruning measurements:

- **Step 1: File-level Sampling.** We draw a uniform-random sample of files from the table (capped at 200 GB per verification job), mirroring the sampling strategy used by the production Liquid Clustering algorithm so that verification stays consistent with deployment. This preserves the distribution of every column in expectation, and the resulting KD-tree is an unbiased estimator of the layout that would be produced by re-clustering the table. AutoLiquid is compatible with more advanced samplers from the literature (e.g., stratified sampling, incrementally maintained table samples) and we leave their implementation to future work.
- **Step 2: Build KD-tree Structure.** For each candidate key $K$, the sampled data is used to construct a KD-tree that simulates the target clustering layout. The tree partitions the sample according to the candidate key, and each leaf node corresponds to a shadow file representing how data would be organized under the proposed layout.
- **Step 3: Impute Shadow File Statistics.** From the simulated layout, the system derives shadow file metadata by deriving statistics such as min/max ranges for each file. These shadow files do not contain physical data. Instead, they represent the metadata that would exist if the clustering were applied. *Stats must be imputed for all relevant data skipping columns, not just the clustering columns, since data skipping works for any field correlated with the clustering.*
- **Step 4: Trace Replay.** Historical query predicates from $Q_{30d}$ are replayed against the shadow file metadata. Predicate evaluation is performed solely against the imputed statistics, allowing the system to estimate file pruning behavior without executing queries or scanning data. For each query, we compute the expected pruning rate $\mathcal{P}_{\text{cand}}(q, K)$, which serves as the verified benefit signal.

***Benefit Comparison.*** The verified candidate benefit is compared against the baseline (either the current clustering columns or the physical layout). If the candidate demonstrates sufficient improvement, the system issues an `ALTER TABLE ... CLUSTER BY ...` command to adopt the new clustering key.

***Recording Feedback (Stage 7).*** After verification completes, the system records the results to enable feedback-based re-ranking in future iterations (Stage 3). For each verified candidate, we compute a *correction factor* based on the discrepancy between the heuristic estimate and the verified ground truth:

$$\delta_K = \frac{B_{\text{verified}}(K) - \widehat{B}(K)}{|Q_{30d}|} \tag{6}$$

This per-scan correction factor captures the systematic bias in the heuristic calculation for the specific candidate and table. The correction factor is bounded to $[-1, 1]$ per scan to prevent outlier verification results from causing excessive adjustments. The recorded entry includes the baseline and candidate column sets,

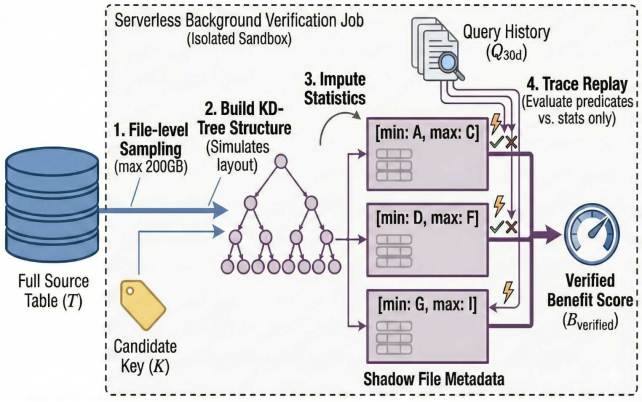



**Figure 4: Shadow verification: the system builds transient KD-trees on sampled data and replays query traces against imputed metadata only. This avoids expensive physical re-clustering while providing near ground-truth accuracy.**

the heuristic-estimated and verified benefits, and the current scan count. As described in Stage 3, this feedback is applied in subsequent evaluations to adjust candidate rankings before they reach the thresholding stage.

## 3.8 Operational Considerations

***Incremental vs. Full Re-clustering.*** Because Liquid Clustering reorganizes only newly-ingested data after a clustering change, prior data remains under the old layout until incremental compaction catches up. AutoLiquid runs a periodic *Benefit Gap* analysis that compares the calculated benefit of the latest clustering scheme against the realized benefit of the current physical layout. If the gap exceeds the write cost of a full rewrite, AutoLiquid triggers a full re-clustering job to close it. Otherwise, incremental compaction closes the gap over time.

***Write-Heavy Table Detection.*** Tables with high write intensity relative to read activity may not benefit from clustering, since maintaining the clustered layout during ingestion can exceed the read savings. AutoLiquid computes a write-intensity metric

$$\text{WriteIntensity} \simeq \frac{\text{BytesWritten/TableSize}}{\text{NumScansEligible}} \tag{7}$$

and temporarily unsets the clustering columns on tables that exceed an empirical threshold. Because telemetry is continuous, the system organically re-applies clustering keys if a table transitions from write-heavy to read-heavy.

# 4 EVALUATION

The primary goals of AutoLiquid are to meet or exceed human baselines for clustering column selection, gracefully handle workload evolution, and operate efficiently at low cost. Hence, we seek to answer the following questions:

(1) What is the clustering quality achieved by AutoLiquid?
(2) What is the benefit of the two-stage verification architecture compared to heuristic-only recommendation?

**Table 2: Comparison of clustering keys chosen by experts vs. AutoLiquid. AutoLiquid provides statistically similar performance ($\Delta < 1\%$) compared to the expert clustering.**

| Table | Expert Keys | AutoLiquid Keys |
|---|---|---|
| | *TPC-DS* | |
| s.s. | ss_sold_date_sk | ss_sold_date_sk |
| s.r. | sr_returned_date_sk | sr_returned_date_sk |
| c.s. | cs_sold_date_sk | cs_sold_date_sk |
| c.r. | cr_returned_date_sk | cr_returned_date_sk |
| w.s. | ws_sold_date_sk | ws_sold_date_sk |
| w.r. | wr_returned_date_sk | wr_returned_date_sk |
| inventory | inv_date_sk | inv_date_sk |
| customer | c_customer_sk | c_preferred_cu..., c_birt... |
| | *TPC-H* | |
| lineitem | l_shipdate | l_shipdate |
| orders | o_orderdate | o_orderdate |
| customer | c_mktsegment | c_mktsegment, c_nat... |
| part | - | p_size |

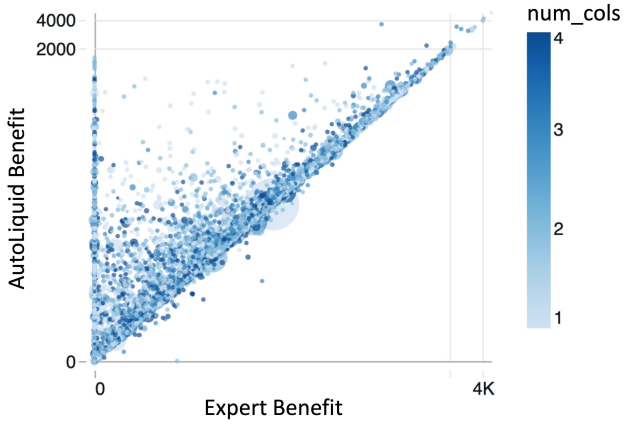



**Figure 5: Beating the human baseline: comparison of pruning benefits (scans saved) between AutoLiquid and customer choices. Most points fall above the diagonal, indicating that autonomic selection consistently yields lower TCO. The bubble size is in proportion to the table size, and the color is based on the number of customer-chosen clustering columns.**

(3) What is the efficiency of AutoLiquid and does the benefit outweigh the cost?
(4) How necessary is it to adapt clustering schemes in response to production workload changes?

## 4.1 Clustering Quality

### 4.1.1 *Microbenchmark.*

We first validate the basic decision making of AutoLiquid on standard analytical benchmarks where expert-tuned clustering keys are well known, to confirm that the autonomous selection recovers expert choices on controlled workloads.

***Setup.*** We use TPC-DS and TPC-H 1 TB tables for evaluation, run on 4-node DBR 15 clusters with three benchmark runs each. We wait for AutoLiquid to adapt the table clustering and then re-run the benchmarks post-optimization, comparing the auto-chosen keys and resulting performance against expert-tuned baselines.
***Results.*** Table 2 shows that the auto-chosen keys align closely with the expert-chosen keys, and in each scenario AutoLiquid provides similar performance to the expert clusterings.
***Takeaways.*** AutoLiquid converges on the same clustering keys an expert would choose on standard benchmarks.

#### 4.1.2 *Customer Comparison.*

Beyond microbenchmarks, we evaluate AutoLiquid against the clustering keys customers actually set in production, to measure decision quality on real workloads at scale.
***Setup.*** We use ~10,000 randomly-sampled customer-managed liquid tables from the production fleet for evaluation. We invoke the AutoLiquid background verification job on each table and compare the backtested AutoLiquid benefit against the existing customer-chosen pruning benefit. Benefit is reported as scans saved (a benefit of 1K means 1K full-table scans saved).
***Results.*** For 50% of tables, AutoLiquid picks clustering columns that improve over the customer-selected clustering; in 47% of tables, the clustering performance was the same; and in 3% of cases the customer-selected clustering was better (typically by a small margin). Inspecting a subset of sampled tables, workloads frequently filter on different keys than the customer-chosen clustering (due to workload changes over time, or users choosing keys based on only a subset of the workload); in some cases customers set too many clustering keys, where reducing the count improves overall pruning for the most frequent and expensive queries. Where AutoLiquid could not match the user-selected columns, the cases were typically legacy tables with disabled statistics collection limiting the heuristic's visibility.
***Takeaways.*** AutoLiquid matches or beats customer-set clustering keys on the large majority of production tables. The hybrid heuristic-plus-verification architecture surfaces high-quality candidates while protecting against the manual-tuning pitfalls of over-clustering and workload drift.

#### 4.1.3 *Post-Clustering Analysis.*

We next track how the resulting layouts evolve over the days following a clustering change to understand whether the benefit is realized and remains stable in production.
***Setup.*** We use the current production tables in the days immediately following an AutoLiquid clustering change. We define the *pruning rate* of a query $q$ as the fraction of files skipped by data skipping out of the files that would otherwise be read, $\text{prune}(q) = \text{FilesPruned}(q)/(\text{FilesPruned}(q) + \text{FilesRead}(q))$, and the *pruning rate change* of $q$ as the difference between the post-update and pre-update pruning rates on the same query, $\Delta\text{prune}(q) = \text{prune}_{\text{after}}(q) - \text{prune}_{\text{before}}(q)$. We measure the per-table pruning rate change and, for the same cohort, estimate the percentage of the theoretical maximum pruning rate achieved based on the observed filter selectivity.
***Results.*** For most tables, the pruning rate improvement is largely positive, providing a large net benefit (Figure 6). Some tables temporarily regress due to workload changes that arrived after the clustering decision was made; over time AutoLiquid adjusts the clustering to adapt to these shifts, resulting in a stable 10% average lift in pruning rate after auto key selection (Figure 7a). For larger tables, AutoLiquid achieves a pruning rate close to the theoretical maximum (Figure 7b), as expected since file skipping becomes more efficient when a table has many files.
***Takeaways.*** The pruning rate lift stabilizes within days of a clustering change and converges close to the theoretical maximum on larger tables; transient regressions caused by workload drift are recovered through continuous re-evaluation.

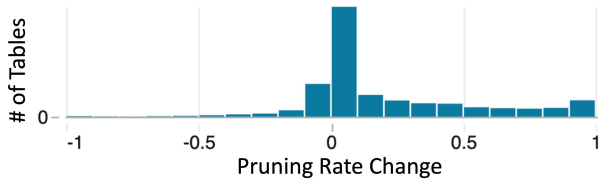
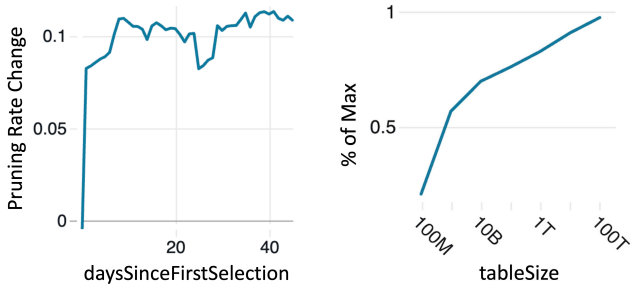



**Figure 6: Distribution of pruning rate change across production tables after clustering column selection, showing AutoLiquid provides significant net benefit.**




**(a) Average pruning rate change stabilizes in the weeks after auto clustering column selection.**

**(b) Average percent of the theoretical max pruning rate achieved by table size bucket.**

**Figure 7: Analysis of the pruning rate over time in the days following AutoLiquid selections. Data is averaged across a representative sample of tables managed by AutoLiquid.**

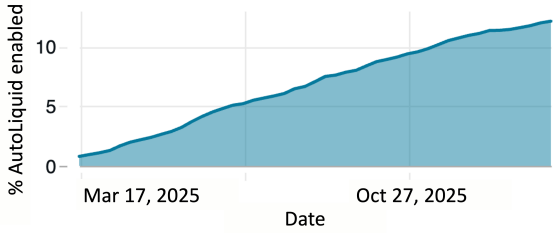



**Figure 8: Organic growth of AutoLiquid from release date, as a percentage of all liquid tables.**

#### 4.1.4 *Adoption Trends.*

Finally, as a qualitative measure of performance, we measure the

organic adoption of AutoLiquid from launch. We do not enable AutoLiquid by default, so customers have to manually choose to create tables with the `CLUSTER BY AUTO` syntax. We observe customers are enabling auto for a large fraction of new tables, resulting in a growth of AutoLiquid to more than 12% of all liquid tables in the fleet today (millions of tables). This reflects customer demand and confidence in AutoLiquid for managing their tables. We expect this number to increase further as we make AutoLiquid the default for all new tables in the future.

## 4.2 Verifier Accuracy & Utility

*4.2.1 Verifier Accuracy.*

As AutoLiquid's decision to apply or reject a candidate clustering key relies on the verifier's predicted pruning rate, the accuracy of those predictions directly determines whether the system makes safe and effective decisions. We therefore measure how closely the verifier's predicted pruning rates match the actual rates observed after clustering.

***Setup.*** We use TPC-DS tables managed by AutoLiquid for evaluation, driven by the TPC-DS workload as the query stream. For each top candidate clustering key surfaced by the verifier on these tables, we form a (predicted, observed) tuple. The prediction is the per-query pruning rate the verifier reports for the candidate prior to applying it, computed by replaying historical predicates against shadow KD-trees built on file-level samples (Section 3.7). The observation is the pruning rate measured for the same query after the candidate has been physically applied to the table. Because the verifier is expected to estimate the absolute pruning rate of each candidate rather than only preserve the relative ordering among candidates, we use the coefficient of determination $R^2$ computed pointwise against the identity line as the accuracy metric, $R^2 = 1 - \sum_i (y_i - \hat{y}_i)^2 / \sum_i (y_i - \bar{y})^2$, where $y_i$ is the observed pruning rate and $\hat{y}_i$ the verifier's prediction. A value close to 1 indicates that the verifier's estimates are nearly identical to the production ground truth, the property we need for safe deployment.

***Results.*** The strong trend line and high $R^2 = 0.998$ in Figure 9 confirm that the verifier closely tracks the post-clustering ground truth. Investigating the residual outliers identifies two sampling artifacts: (i) when the data density per KD-tree leaf is too low, the imputed boundaries become inaccurate; and (ii) file-level sampling (an engineering compromise we accepted because row-level uniform sampling on Parquet incurs prohibitive decoding cost) can occasionally miss highly active but spatially localized data clusters. We mitigate the first artifact by enforcing a minimum row coverage per leaf to ensure statistical significance. The second artifact has bounded impact in practice: because file-level sampling preserves the global data distribution, it primarily results in minor rank-swapping among similarly effective candidates rather than the selection of poor keys, and so does not lead to significant performance regressions. We discuss these artifacts for completeness; they have not proven to be practical impediments to optimization quality in our production fleet. We plan to switch to a stratified sampling algorithm in future versions of AutoLiquid to resolve these edge cases without incurring additional compute costs.

***Takeaways.*** Shadow-verification pruning rate predictions are highly correlated with observed pruning, justifying the verifier as a safe substitute for live re-clustering when deciding whether to commit a layout change.


**Figure 9: Verifier accuracy: comparison of verifier estimates vs. actual query pruning rates on TPC-DS showing high correlation ($R^2 = 0.998$).**

*4.2.2 Multi-Candidate Benefit.*

We next ablate the multi-candidate verification design, asking how much benefit comes from verifying multiple candidates per table and from the heuristic ordering used to prioritize them.

***Setup.*** We use AutoLiquid-managed customer tables across the production fleet for evaluation. We vary the number of candidates $K$ verified per table and measure the resulting expected fleet benefit (Figure 10). To isolate the contribution of the heuristic ranking, we compare AutoLiquid's heuristic-ranked candidate ordering (blue line) against an alternative in which the heuristic ordering is removed, so candidates are not prioritized by estimated benefit (gray line).

***Results.*** Expected fleet benefit increases monotonically with $K$, and verifying the top six candidates per table already captures >99% of the maximum overall benefit. Further candidates add only marginal compute cost, and $K = 6$ is the value used in production. The gap between the blue and gray lines shows that the heuristic ordering is useful to prioritize the most promising candidates first, accelerating convergence to the maximum.

***Takeaways.*** Verifying the top six heuristic-ranked candidates per table is sufficient to capture nearly all of the achievable benefit, justifying $K = 6$ as the production setting and confirming the value of pairing heuristic ranking with multi-candidate verification.

## 4.3 Cost Efficiency

***Cost Breakdown.*** The total extra cost of AutoLiquid is the sum of two factors:

- *Intelligence cost:* The cost of the telemetry processing, heuristic analysis, and the ephemeral storage and compute used for the serverless verification jobs.
- *Clustering cost:* The write cost incurred by the Liquid Clustering jobs to physically reorganize the data, including costs for clustering on new writes.

We estimate the per-table clustering cost as $\text{ClusteringCost}_{\text{write}} \approx 2 \cdot 20 \cdot |T|$, where $|T|$ is the cost of reading the table once. The factor

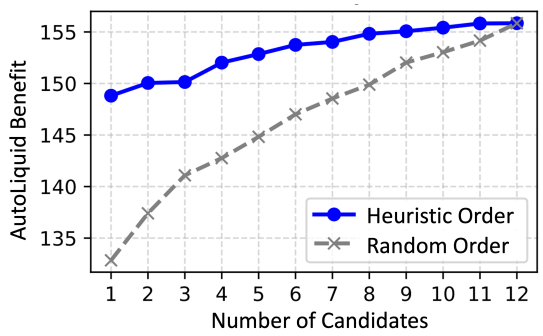



**Figure 10: Multi-candidate verification yields significantly higher benefits than single-candidate selection.**

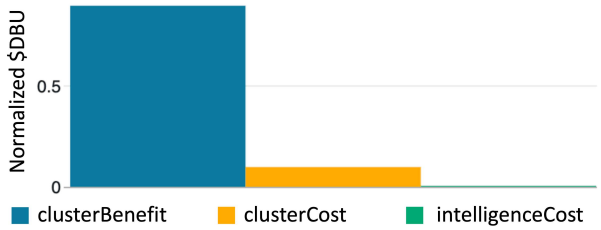



**Figure 11: Cost vs. benefit: comparison of the clustering benefit achieved by AutoLiquid with its clustering and intelligence costs. AutoLiquid is efficient, maintaining a cost:benefit ratio of ~1:10.**

of 2 is the CTAS multiplier (a re-cluster reads the data once and writes it once), and the factor of 20 is the empirically measured ratio of clustering compute to a baseline read of the same volume of data. This is a conservative upper estimate. In practice, incremental clustering only reorganizes newly-ingested data rather than the full table, so the realized cost is typically lower.

***Cost vs. Benefit.*** These costs are weighed against the *clustering benefit*, which represents the compute savings from improved data skipping in the optimized layout. Figure 11 reports the normalized clustering benefit, clustering cost, and intelligence cost across production tables with stable recurring query patterns, which lets us distinguish benefits attributable to AutoLiquid from random workload variation. The *intelligence cost* is the measured Databricks Cost Unit (DBU) consumption of AutoLiquid in production; the *clustering cost* is estimated using the upper-bound formula above; and the *clustering benefit* is estimated from the observed pruning improvement and scan workload. The majority of AutoLiquid cost comes from clustering (~1:10 cost-to-benefit ratio), with a small minority from the intelligence cost of running background verification jobs (<1:100 cost-to-benefit ratio). The aggregate cost is therefore strongly outweighed by the aggregate clustering benefit, indicating that AutoLiquid is a net positive TCO-wise in addition to improving performance and reducing administrative burden.

***Per-Phase Intelligence Cost.*** Decomposing the aggregate cost into the seven-stage pipeline (Section 3):

- **Stage 1 (Telemetry Collection):** aggregates scan and pruning metrics from the Databricks telemetry stream over the trailing 30-day window. The P90 table observes roughly 1K scans during this window.
- **Stages 2–4 (Candidate Identification, Re-ranking, Thresholding):** run as part of the Predictive Optimization control-plane service (~3 service replicas per region).
- **Stages 5–6 (Shadow Verification & Clustering Application):** the dominant intelligence cost. The combined operation runs on serverless data-plane compute: it samples up to 200 GB of table data, builds KD-trees, replays predicates, and, when verification passes, issues the clustering change. Across the production fleet, the median operation finishes in roughly ten minutes of wall-clock time and costs on the order of 0.5 DBU per operation.
- **Stage 7 (Record Feedback):** control-plane recording cost is negligible.

***Table Coverage.*** In Figure 12 we observe that AutoLiquid clusters 49% of all non-trivial (>256MiB) tables where auto is enabled. The main dropoff is between selection scheduled and selection changed, which reflects scenarios where the heuristic thinks there may be a promising clustering candidate, but the verifier rejects the candidate as not improving over the current table layout. The high fraction of table coverage, in addition to cost vs. benefit numbers, suggests that AutoLiquid is cost-effective for customers to enable for all tables. The significant fraction of rejections (~30%) also shows the importance of the verifier to avoid frequent regressions in performance.

***Takeaways.*** The aggregate clustering benefit strongly dominates the cost, yielding a favorable cost-benefit ratio. AutoLiquid clusters a significant fraction of monitored tables, while the verifier rejects a meaningful share of candidates as not improving over the current layout.

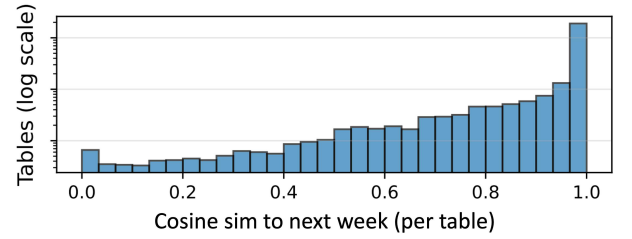
## 4.4 Adapting to Changing Workloads

***Similarity Analysis.*** In Figure 13a we examine the dynamicity of workloads by evaluating their week-over-week self-similarity. We compute a vector of the top queries per workload by literal-insensitive query signature, and measuring the cosine similarity of the vector between weeks. While many workloads are very stable (high cosine similarity), there is a long tail of workloads that change significantly week over week (low cosine similarity). This observation motivates AutoLiquid's design in accommodating query workload changes as a first class consideration.

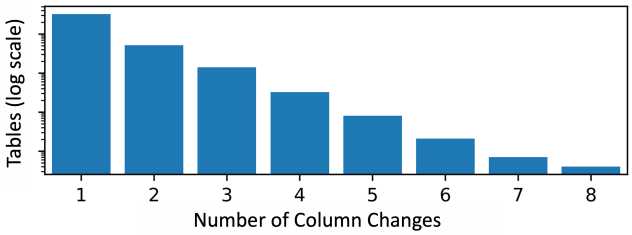
***Clustering Changes.*** In Figure 13b we examine the distribution of how many column selection changes AutoLiquid has made per table it is managing. We observe that for most tables, only a few selections are made, reflecting a relatively stable workload profile. However, there is a long tail of tables that have required as many as eight clustering changes as the workload evolves over time.

***Takeaways.*** Most workloads are stable week-over-week, but a long tail shifts substantially, motivating continuous re-evaluation rather than a one-shot decision. Correspondingly, most tables undergo only a few clustering changes while a long tail sees up to ~8 changes, confirming that adaptation is a recurring need at fleet scale.


**Figure 12: Conversion funnel of monitored tables to optimized tables. AutoLiquid clusters close to 50% of monitored tables, rejecting tables that would not benefit from optimization or have insufficient telemetry.**




**(a) Cosine similarity**




**(b) Clustering changes**

**Figure 13: Workload volatility: (a) While many workloads are stable, a "long tail" of tables experiences significant week-over-week query shifts (low cosine similarity). Such query shifts necessitate the re-clusterings shown in (b).**

## 4.5 Regression Analysis

We further investigate the cases in which the customer's existing clustering keys outperform AutoLiquid candidate (see Figure 1 for the fleet-level breakdown).

***Setup.*** We use the same cohort of customer-managed liquid tables as in Section 4.1.2 for evaluation. We focus on the regressed cases in this cohort over a 30-day window of column-selection events, and classify them by the following root causes.

- **Column-count ceiling** (~81% of regressions): the customer key spans 3–4 columns, while AutoLiquid currently caps multi-column candidates at 2 columns.
- **Complex workloads** (~10%): the workload filters on 6–16 columns and the customer's key is not ranked highly enough by the heuristic to enter the top-$K$ verified set.
- **Missing candidates** (~7%): the workload filters on $\leq 2$ columns but the customer's column does not appear in the heuristic. This arises when the customer's column delivers a high per-scan pruning benefit but appears in too few scans to clear the predicate-frequency filter in Phase 1 of candidate identification (Section 3.3).
- **Pair not enumerated** (~1%): both customer columns appear individually in the heuristic candidate set, but the multi-column pair was not generated.

The first three buckets together account for over 90% of the regressions and reflect a cost-performance tradeoff in the verification stage of AutoLiquid, where we could improve the quality of selections by increasing the search space and hence compute used by verification. We are investigating how to best close these gaps as future work.

***Takeaways.*** Strict regressions are rare and concentrated in two structural causes: the multi-column candidate ceiling and heuristic ranking on highly complex workloads. Both cases are due to conservative thresholding (candidate number ceiling and top-$K$).

## 5 DISCUSSION AND LESSONS LEARNED

This section overviews key lessons learned building and deploying AutoLiquid to fleet-scale at Databricks.

***Risk-Aware Decision Making.*** The cost-benefit landscape of an autonomic layout selection system is asymmetric, i.e., a modest aggregate performance improvement is welcome but rarely noticed, while even a small regression on a high-value workload can be treated as a critical incident. AutoLiquid addresses this asymmetry through a *verify-before-commit* principle, backed by exponential thresholds (Stage 4), shadow verification (Stage 5), and feedback damping (Stage 7). No physical layout change is applied until shadow verification confirms a measurable benefit on freshly-sampled customer data, and the bar for each successive change scales with the table's history of prior changes.

***TCO-Centric Optimization.*** Unlike approaches that minimize per-query latency, AutoLiquid optimizes against total cost of ownership, explicitly weighing the read savings of a candidate layout against the write cost of physically reorganizing the data. This prevents expensive reorganization on write-heavy or low-traffic tables where the read savings would not justify the overhead. More importantly, for an autonomic system, it shifts the conversation with customers away from per-query metrics that can be improved at the expense of overall cost. The general lesson is that for autonomic services, a good default optimization objective focuses on the customer's cost rather than per-query metrics, since the customer is more willing to universally accept optimizations that do not sacrifice cost for performance.

***Observability and User Trust.*** Initially during AutoLiquid development, we tried to hide many observability details from the user. The rationale was that hiding details can simplify the user experience and avoid exposing internal details that may change. However, this led many users to develop a distrust for the decisions made by AutoLiquid, since the decisions were often non-intuitive (e.g., not clustering an already well-performing table) and they were unable to observe the underlying rationale. We have since

corrected this and have integrated improved observability details into AutoLiquid for end users.

***POC Workloads.*** A particularly challenging class of workloads for autonomic systems are proof-of-concept (POC) workloads, which are often short-lived and run as synchronous benchmarks by end users and field engineers. These synchronous benchmarking loops mesh poorly with the asynchronous and offline nature of AutoLiquid. To resolve this gap, we are integrating AutoLiquid intelligence into the user-issuable `OPTIMIZE` command, to give the users a means to run AutoLiquid logic immediately on a table.

## 6 FUTURE WORK

We envision the evolution of AutoLiquid moving beyond only clustering column selection toward a comprehensive, self-governing data layout engine.

- **Granular Layout Optimizations:** We are working to extend the verification framework to optimize the underlying file structure as well. This includes implementing adaptive file sizing for frequently accessed recent data, determining the optimal hierarchical ordering of clustering keys, and identifying when to offload columns to secondary indexes.
- **Expanding Optimization Objectives:** While currently maximizing filter pruning, we aim to more holistically optimize workloads by incorporating objectives for co-partitioned joins and group-by aggregations. Future iterations will also include safeguards to prevent data size inflation, ensuring decisions preserve compression efficiency.
- **Decision Latency and User Control:** To address cold starts, we are exploring LLMs and similar table telemetry to infer likely clustering keys for new tables. We also aim to improve user control through optimization triggers and fine-grained mechanisms to prioritize specific queries for regression protection.
- **Cross-Table Coordination:** AutoLiquid currently optimizes each table in isolation. While per-table telemetry implicitly captures the marginal pruning value of each scan operand, jointly coordinating clustering choices across tables that frequently appear in the same query could unlock pruning and join-execution benefits beyond what isolated per-table optimization captures.

## 7 RELATED WORK

Automated physical database design has been extensively studied in both academic research and industrial systems. Prior work spans classic physical design advisors, modern cloud automation features, and adaptive or "self-driving" database systems.

***Layout and Indexing Foundations.*** Early physical design tools pioneered automated recommendation using the query optimizer's cost-estimation engine. Systems such as AutoAdmin, the Microsoft SQL Server Database Tuning Advisor [1, 2, 9, 10], and the IBM DB2 Design Advisor [30] introduced "what-if" analysis. This allowed hypothetical indexes, partitions, or materialized views [17, 32] to be evaluated without physical materialization. Subsequent research expanded these techniques to integrated physical design strategies [3, 5, 26].

The underlying primitives for these optimizations rely on physical data ordering to improve scan efficiency. Column-store systems like C-Store [29] and Vertica [22], along with modern formats like Parquet [16], leverage min-max statistics or zone maps to enable data skipping. Recent work [6] also proposes an automated workload-driven approach that analyzes query predicates to recommend effective clustering strategies and corresponding zone map constructions to improve pruning efficiency. While foundational, these classic approaches primarily functioned as recommendation engines for human administrators rather than autonomous closed-loop systems.

***Industrial Cloud Automation.*** Modern cloud data warehouses provide varying degrees of automated physical layout management. Google BigQuery [24] supports clustering and partitioning recommendations derived from query history, but requires users to manually apply changes. Snowflake provides Automatic Clustering [28], which continuously reorganizes data in the background, yet still relies on users to define the initial clustering keys.

Other industrial systems, such as Oracle and Azure SQL Database [7, 11], have pioneered automated maintenance for secondary indexing. However, these platforms avoid fully autonomous selection of *primary* data layouts due to the high operational risks and reorganization costs. Amazon Redshift's advisor [25, 27] further addresses data movement by recommending distribution or sort keys. AutoLiquid differs from these semi-automatic or advisory systems by integrating workload observation, verification, and deployment into a fully autonomous pipeline for Lakehouse primary layouts.

***Adaptive and Self-Driving Systems.*** Recent research investigates "self-driving" databases that continuously optimize based on live workload observations. This includes adaptive indexing and database cracking [18, 19], which incrementally reorganize structures based on query patterns. Modern frameworks have shifted toward learned models; for example, SWIRL [20] uses reinforcement learning for joint indexing and layout optimization, while SageDB [15] focuses on learned physical designs modeled on data distributions.

Recent work also addresses the volatility of cloud environments. ResTune [31] focuses on stability under shifting query distributions, while GPTuner [23] explores using LLMs to navigate complex knob configuration search spaces.

## 8 CONCLUSION

We present AutoLiquid, a complete system for continuous clustering key selection. By combining workload telemetry with reliable shadow verification, AutoLiquid overcomes the trade-offs between optimization benefit and regression risk. AutoLiquid efficiently manages millions of tables at Databricks, automatically adapting clustering keys in response to workload changes. We show that AutoLiquid matches or outperforms human-selected layouts in most cases, indicating that a closed-loop, verification-driven architecture can be successfully automated at scale.

## REFERENCES


[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 359–370.

[4] Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, et al. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[5] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 227–238.

[6] 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*. 68–79.

[7] Sunil Chakkappen, Shreya Kunjibettu, Daniel McGreer, Masoomeh Javidi Kishi, Hong Su, Mohamed Ziauddin, Mohamed Zait, Zhan Li, and Yuying Zhang. 2025. Automatic Indexing in Oracle. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 4924–4937. https://doi.org/10.14778/3750601.3750616

[8] Ramesh Chandra, Haogang Chen, Ray Matharu, Sarah Cai, Jeff Chen, Priyam Dutta, Bogdan Ghita, Todd Greenstein, Gopal Holla, Peng Huang, et al. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *Companion of the 2025 International Conference on Management of Data*. 310–322.

[9] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.

[10] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. San Francisco, 146–155.

[11] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[12] Databricks. 2023. *Introducing Predictive Optimization: Faster Queries, Cheaper Storage, No Sweat.* Retrieved February 2026 from https://www.databricks.com/blog/introducing-predictive-optimization-faster-queries-cheaper-storage

[13] Databricks. 2025. *Announcing Automatic Liquid Clustering.* Retrieved February 2026 from https://www.databricks.com/blog/announcing-automatic-liquid-clustering

[14] Databricks. n.d.. *Delta Lake Clustering.* Retrieved February 2026 from https://docs.databricks.com/aws/en/delta/clustering

[15] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. Sagedb: An instance-optimized data analytics system. *Proceedings of the VLDB Endowment* 15, 13 (2022).

[16] Apache Software Foundation. n.d.. *Apache Parquet Documentation.* Retrieved February 2026 from https://parquet.apache.org/docs/

[17] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[18] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[19] Stratos Idreos, Stefan Manegold, and Goetz Graefe. 2012. Adaptive indexing in modern database kernels. In *Proceedings of the 15th International Conference on Extending Database Technology*. 566–569.

[20] Jan Kossmann, Alexander Kastius, and R. Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:247921409

[21] Delta Lake. n.d.. *Delta Lake Documentation.* Retrieved February 2026 from https://delta.io/

[22] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[23] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2025. GPTuner: An LLM-Based Database Tuning System. *ACM SIGMOD Record* 54, 1 (2025), 101–110.

[24] Justin Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugam, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *SIGMOD*.

[25] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and effective distribution-key recommendation for amazon redshift. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2411–2423.

[26] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 558–569.

[27] Amazon Web Services. n.d.. *Amazon Redshift Advisor Recommendations.* Retrieved February 2026 from https://docs.aws.amazon.com/redshift/latest/dg/advisor-recommendations.html

[28] Snowflake. n.d.. *Automatic Clustering.* Retrieved February 2026 from https://docs.snowflake.com/en/user-guide/tables-auto-reclustering

[29] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[30] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. 2000. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 101–110.

[31] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2102–2114. https://doi.org/10.1145/3448016.3457291

[32] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al. 2004. Recommending materialized views and indexes with the IBM DB2 design advisor. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 180–187.