AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes

Anja Gruenheid*
Microsoft
Zurich, Switzerland

Jesús Camacho-Rodríguez*
Microsoft
Mountain View, CA, USA

Carlo Curino
Microsoft
Redmond, WA, USA

Raghu Ramakrishnan
Microsoft
Redmond, WA, USA

Stanislav Pak*
LinkedIn
Sunnyvale, CA, USA

Sumedh Sakdeo*
LinkedIn
Sunnyvale, CA, USA

Lenisha Gandhi
LinkedIn
Sunnyvale, CA, USA

Sandeep K. Singhal
LinkedIn
Sunnyvale, CA, USA

Pooja Nilangekar†
University of Maryland
College Park, MD, USA

Daniel J. Abadi
University of Maryland
College Park, MD, USA

Abstract

The proliferation of small files in data lakes poses significant challenges, including degraded query performance, increased storage costs, and scalability bottlenecks in distributed storage systems. Log-structured table formats (LSTs) such as Delta Lake, Apache Iceberg, and Apache Hudi exacerbate this issue due to their append-only write patterns and metadata-intensive operations. While compaction–the process of consolidating small files into fewer, larger files–is a common solution, existing automation mechanisms often lack the flexibility and scalability to adapt to diverse workloads and system requirements while balancing the trade-offs between compaction benefits and costs. In this paper, we present AutoComp, a scalable framework for automatic data compaction tailored to the needs of modern data lakes. Drawing on deployment experience at LinkedIn, we analyze the operational impact of small file proliferation, establish key requirements for effective automatic compaction, and demonstrate how AutoComp addresses these challenges. Our evaluation, conducted using synthetic benchmarks and production environments via integration with OpenHouse–a control plane for catalog management, schema governance, and data services–shows significant improvements in file count reduction and query performance. We believe AutoComp's built-in extensibility provides a robust foundation for evolving compaction systems, facilitating future integration of refined multi-objective optimization approaches, workload-aware compaction strategies, and expanded support for broader data layout optimizations.

CCS Concepts

• **Information systems → Data management systems**.

*Authors contributed equally.
†Work done while author was at Microsoft.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGMOD-Companion '25, Berlin, Germany

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724430

Keywords

Compaction, log-structured tables, data lake, storage optimization

ACM Reference Format:
Anja Gruenheid, Jesús Camacho-Rodríguez, Carlo Curino, Raghu Ramakrishnan, Stanislav Pak, Sumedh Sakdeo, Lenisha Gandhi, Sandeep K. Singhal, Pooja Nilangekar, and Daniel J. Abadi. 2025. AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724430

1 Introduction

In recent years, enterprises have undergone a significant shift in their approach to data management, progressively gravitating towards data lake-centric architectures. Data lakes originated as cost-effective storage for large volumes of unstructured, uncleaned, or ungoverned data in scalable distributed file systems like HDFS, providing an alternative to storing this data in expensive proprietary data management or file systems. Over time, the declining cost of data lake storage has encouraged organizations to use these systems for managing core, governed, and structured data as well. This shift was enabled by the widespread adoption of scalable storage services [4, 13, 30, 42] and efficient open-source data formats [12, 15] that serve as foundational elements for persisting data across diverse workloads. Data stored in distributed storage systems is then accessible to various engines and applications, providing several advantages: (*i*) independent scaling of storage and compute, enhancing efficiency and cost savings, (*ii*) elimination of data silos, which streamlines workflows and simplifies complex data movement across systems, and (*iii*) the flexibility to choose the optimal engine for each application, thereby mitigating lock-in concerns. Various commercial platforms embrace this approach [26, 46, 63].

Engines and applications accessing these distributed storage systems require guarantees such as consistency and isolation during complex transactions involving read and write operations. However, these storage systems are primarily designed for scalability and durability, and lack the concurrency and recovery capabilities needed to meet these requirements. As a result, open table formats such as Delta Lake [18, 27], Apache Iceberg [7], and Apache

Hudi [5], also referred to as log-structured tables or LSTs in the following, have emerged to enable structured data to be stored in data lake storage solutions while remaining organized and optimized for external query engines, achieving excellent query performance.

At their core, these LSTs store data persistently in immutable files relying on open-source columnar formats [12, 15] and propose (*i*) a metadata layer that records table versions and attributes such as data schemas and statistics, and (*ii*) a protocol to coordinate interactions with a table during read and write operations. Catalogs play a critical role in this context by maintaining references to table metadata and enabling seamless access and updates across various systems [16, 37, 67]. With each write operation, new data files are added to the table, and the corresponding table metadata is updated. Over time, layers of data files (small in size in common trickle-write scenarios and untuned writers) can accumulate within the table.

The Challenge of Small Files. The accumulation of numerous small files presents a significant challenge in data lake-centric architectures, impacting all engines and LST implementations, as extensively documented in prior studies [22, 32, 40, 54]. This proliferation of small files increases overhead due to a higher number of managed objects and more frequent IO requests, which can strain the distributed storage systems underpinning data lakes and impact their performance and scalability [56, 61]. For instance, HDFS encounters scalability challenges as the NameNode, which maintains file system metadata, can manage only a limited number of objects. As file counts grow, the number of managed objects rises proportionally, placing additional pressure on the NameNode and often necessitating federation to distribute the load. Additionally, elevated RPC traffic generated by small files places further burden on HDFS, requiring additional observer NameNodes (i.e., read-only replicas) to manage the increased traffic effectively.

Small files storing a limited number of rows also reduce the efficiency of columnar formats that rely on robust encoding and compression to optimize data access and storage. Moreover, the presence of these files contributes to bloated metadata in LSTs. Each transaction appends references to the files in logs or manifests, causing metadata size to grow and increasing the time required for query processing and maintenance operations, thereby affecting overall performance and efficiency. The problem is exacerbated by potential monetary costs, as cloud service providers often charge based on IO requests and data transfer volume [3, 31, 43].

Compaction as a Solution. The most prevalent *storage healing* mechanism to address this issue is *compaction*. Compaction is the process of rewriting data files in a table to create fewer, larger files according to a target file size, which helps improve storage efficiency, query performance (both in planning and execution), and overall data organization. Each LST implements its own version of compaction mechanisms [6, 8, 28]. Industry practices for compaction vary widely [25, 32, 47, 65], from reactive strategies that trigger compaction after a data write operation to standalone solutions that periodically optimize the storage layout. While some engines integrate proactive mechanisms to maintain an optimal data layout, these optimizations are typically performed in isolation, addressing only the needs of the specific engine without considering other engines that might access the same data.

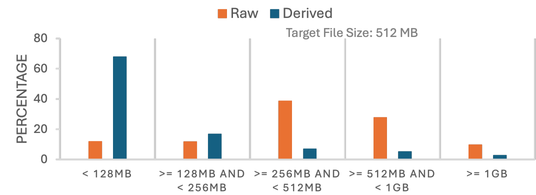


Figure 1: File size distribution for ingested data (raw ingestion vs. user-derived data).

Contributions. In this work, we address the challenge of small file proliferation in LSTs by drawing on our practical experience to develop an automated data compaction solution. Our contributions are as follows:

- **Analysis of Small File Proliferation in Industry Scenario.** We highlight the operational challenges posed by data fragmentation across numerous small files in a real-world scenario at LinkedIn. We identify the most common causes of this small file proliferation and illustrate its impact on storage efficiency and query performance (§2).
- **Definition of Requirements and Introduction of AutoComp.** Guided by our findings, we establish a set of essential functional and non-functional requirements to effectively address small file proliferation in LSTs in practice. We then introduce AutoComp, a framework designed to meet these requirements and enable automatic scalable data compaction (§3-5).
- **Comprehensive Evaluation of AutoComp.** We evaluate AutoComp in a cloud-based deployment using synthetic benchmarks to assess its effectiveness. We also report on its impact after deployment at LinkedIn, demonstrating substantial improvements of up to 44% reduction in the number of files smaller than "128MB" in a production environment (§6-7).
- **Discussion of Future Directions.** We identify areas for improvement and propose future research directions to improve AutoComp and data compaction techniques in LSTs (§8).

2 Motivating Scenario

At LinkedIn, raw event data is ingested from thousands of services into a data lake through a centrally managed pipeline powered by Apache Gobblin [36]. The organization is structured into various lines of business, each responsible for maintaining data pipelines that produce derived data, business metrics, and feature sets. Line-of-business engineers, referred to as end-users in the following to distinguish them from data infrastructure engineers, primarily develop these pipelines using compute engines such as Apache Spark, Trino, and Apache Flink.

LinkedIn has adopted Apache Iceberg as the LST for storing data generated by these pipelines, thereby standardizing data storage practices across its analytics and artificial intelligence workloads. Building on the adoption of Iceberg, LinkedIn has also developed and open-sourced **OpenHouse** [37], a control plane that provides a declarative catalog for table definitions, schema management, and metadata maintenance, along with data services to reconcile observed and desired states. For over a year, LinkedIn has been onboarding existing and new tables into OpenHouse. The distribution of file sizes exhibits a marked difference between raw data ingested

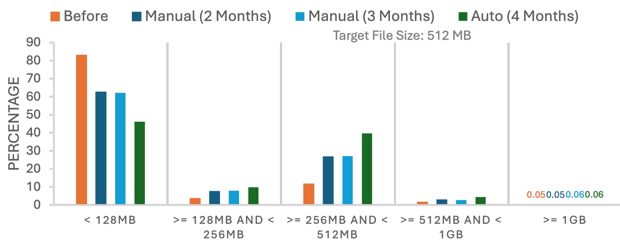


Figure 2: File size distribution for OpenHouse-managed Iceberg tables, shown before and after compaction.

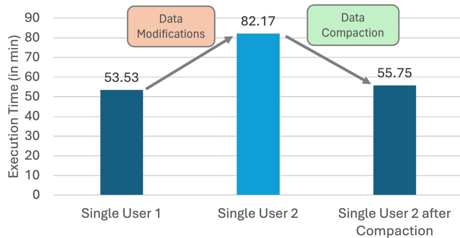


Figure 3: TPC-DS experiment (Apache Spark & Iceberg): Comparison of execution time before and after compaction.

by the central pipeline and derived data generated by end-user jobs, as illustrated in Figure 1. The central pipeline follows a well-defined pattern, writing raw event data from Kafka to HDFS every five minutes and incrementally compacting and deduplicating it into hourly partitions, resulting in files of approximately 512MB in size, i.e., our target file size. Daily partitions, composed of 24 hourly segments, are retained for long-term storage, while smaller checkpoint files are expired after three days. In contrast, end-user jobs using Spark, Trino, and Flink are neither designed nor tuned for generating optimal file sizes, resulting in a high concentration of small files. Expecting end-users to prioritize file optimization is unrealistic, as their primary focus is addressing business challenges rather than managing low-level data storage concerns.

Causes of Small File Existence. The proliferation of small files in these derived tables can be attributed to several factors related to Iceberg's versioning semantics as well as the design and configuration of end-user jobs. (*i*) *Inserts*: While bulk inserts can produce optimally sized files, aspects such as engine configuration, degree of parallelism, and memory constraints significantly influence the resulting number of files in the table. Incremental inserts, including Change Data Capture (CDC) scenarios [21], often lead to the rapid creation of numerous small files. (*ii*) *Updates and Deletes*: In Copy-on-Write (CoW) configurations, deletions can affect data distribution across files, leading to uneven file sizes. Merge-on-Read (MoR) configurations generate delta files that accumulate over time. (*iii*) *Migration*: When existing Parquet or ORC data is migrated into Iceberg, the original file structure is typically preserved while Iceberg metadata is layered on top [11], resulting in suboptimal file layouts. (*iv*) *Metadata*: Iceberg introduces additional metadata for each table to manage state, including manifests and manifest lists. This added metadata contributes to small file proliferation.

Impact of Compaction. The small file size distribution patterns observed in user-derived data prompted data infrastructure engineers to leverage OpenHouse to introduce centrally managed compaction for the first time at LinkedIn. In the initial implementation, compaction was triggered manually for selected tables that exhibited recurring issues such as query failures, quota breaches, and namespace growth in HDFS. This approach proved effective; as shown in Figure 2, manual compaction realigned file size distribution toward the target, reducing the storage system's load and improving overall efficiency.

In addition to its impact on the storage layer, we also note that file proliferation impacts query performance. However, directly measuring the impact of compaction on query performance is challenging in a production environment, as infrastructure engineers do not control workloads executed by different lines of business. To provide insights, we conducted a synthetic experiment using the TPC-DS benchmark [49] at a scale factor of 1000. The results, shown in Figure 3, capture the end-to-end runtime of the *single-user* phase, which includes all TPC-DS queries, on a 16-node Spark cluster before and after a *data maintenance* phase. During the *data maintenance* phase, about 3% of the data is modified via delete and insert operations, resulting in new files being added to the table. This significantly degrades performance in the subsequent *single-user* phase, increasing execution time by a factor of 1.53×. However, manually triggering compaction restored performance to levels comparable to the initial execution of the workload. This experiment highlights that effective data maintenance is not only necessary for the storage layer but also significantly impacts query execution robustness and efficiency. As a drawback, note that compaction running concurrently with a user's workload may cause write-write conflicts, resulting in longer execution time due to potential retries and wasted resources. However, we experimentally show in §6 that the benefits usually outweigh the cost.

Limitations of Manual Intervention. Although compaction has proven effective, manually selecting tables for compaction is clearly not scalable to meet LinkedIn's operational demands. Specifically, to address the small file problem, data infrastructure engineers had been dedicating increasing amounts of time to developing cost-effective strategies for reorganizing onboarded data, while ensuring scalability and managing the maintenance and onboarding of additional tables. This reactive approach is unsustainable, as it allows user workflows to fail before compaction can be applied. At the same time, enabling periodic compaction across the entire fleet of 21K onboarded tables in OpenHouse (projected to grow to 100K by next year) was also determined to be prohibitively expensive in terms of both capital expenditure (capex) and operational expenditure (opex). Our analysis quantified the magnitude of this challenge: compacting approximately 3K raw event tables in the managed ingestion pipeline uses a daily average compute capacity of 150TBhrs, and a daily peak compute capacity of 600TBhrs. As a result, we started to develop AutoComp as a *resource-conscious* way of enabling compaction in LinkedIn, initially executing it over a limited selection of tables. Its effectiveness even within a short timeframe is shown in Figure 2, allowing OpenHouse to shift the file size distribution towards the target file size at an accelerated pace since its rollout, as discussed further in §7. In contrast, running compaction on a fixed set of tables at a predefined frequency did not yield a significant impact on file size distribution, especially once the system reached a quasi-normal state, leading to fewer

opportunities for further optimization. As a result, subsequent compaction runs often processed files that were already well-sized and balanced, yielding minimal improvements in file size distribution. This diminished return highlights the inefficiency of static compaction schedules that do not adapt based on the dynamic changes in data patterns and table usage as well as the need for automatic table selection and compaction capabilities.

3 AutoComp Overview

Building on the conclusions drawn from the previously discussed scenario, our objective is to design and implement a framework that enables automatic data compaction in production environments, carefully balancing its benefits and associated costs. Our design is guided by a set of functional (**FR**) and non-functional (**NFR**) requirements, which we outline in the following sections. We then introduce AutoComp, our proposed solution.

3.1 Functional Requirements

Our functional requirements define the necessary capabilities that a framework must have to effectively address the identified challenges for auto-compaction.

FR1: *Fine-grained work units.* AutoComp should automatically select compaction candidates based on dynamic data analysis. It should also identify fine-grained work units to execute compaction at the optimal level of granularity, maximizing potential benefits. By providing the option to break down compaction workloads into smaller, sub-table work units that can be processed independently, the framework can effectively distribute the compaction tasks across segments from different large tables. This approach enhances parallelism and resource use, allowing the system to prioritize the most impactful segments across the large number of tables. Smaller work units are also easier to schedule and need fewer resources, which is particularly advantageous in resource-constrained environments. It ensures incremental progress, enhances fault tolerance by reducing the need for full table restarts after failures or conflicts, and minimizes disruptions to ongoing operations. However, we must remain aware of the start-up cost of instantiating more compaction tasks.

FR2: *Support for multiple compaction strategies.* The framework should support various compaction strategies that can encode the benefits, costs, or a combination of both, depending on the optimization objective. For instance, to reduce the load on the storage layer, a benefit-based trigger could greedily prioritize tables with a higher number of small files. Additionally, in resource-constrained situations, this trigger could be enhanced with cost-awareness to prioritize operations that yield higher benefits at a lower cost. Switching between triggers ensures adaptability to diverse scenarios and maintains balance between performance and resource use.

FR3: *Periodic and post-write execution triggers.* The framework should support execution triggered both periodically and immediately after large write operations. Periodic execution ensures regular data layout optimization, preventing excessive fragmentation over time and offering predictable cost management. Post-write execution enables immediate reorganization, improving performance and curbing file proliferation after significant data ingestion.

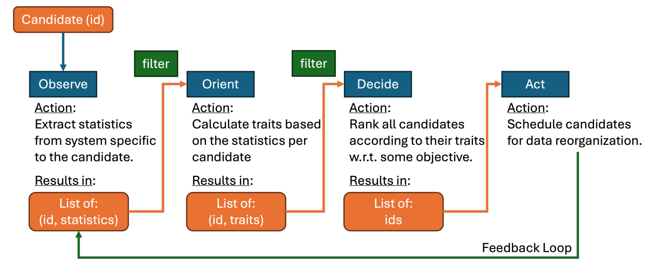


Figure 4: End-to-end workflow for AutoComp.

3.2 Non-Functional Requirements

Next, we introduce non-functional requirements that help us design AutoComp as those requirements that broaden its applicability beyond the scope of our specific use case.

NFR1: *Extensibility.* The framework should be designed with future extensibility in mind, enabling it to integrate additional compaction strategies and adapt to new workloads as needed. This is important due to diversity of data lake workloads, and the ability to mix and match components (e.g., compaction strategies, scheduling policies) ensures the system can evolve without major re-engineering.

NFR2: *Explainability.* The framework should produce consistent compaction decisions under identical input conditions (e.g., file size distribution, workload characteristics). Deterministic decision-making simplifies debugging, testing, benchmarking, and documenting the optimizer's behavior in large-scale production environments, making the system more transparent and manageable.

NFR3: *Cross-platform compatibility.* The framework should be designed to work seamlessly across different LST and catalog implementations. This approach extends its utility beyond LinkedIn's use case to other data lake-centric platforms such as Fabric [46]. Such flexibility enables the framework to adapt to a wider range of deployment environments, broadening its impact and applicability.

3.3 AutoComp: A Framework for Compaction

Considering the desiderata previously outlined, we next describe the workflow for a universal, automated compaction framework for LSTs, referred as to AutoComp in the following. We employ a decision-making model known as 'Observe, Orient, Decide, Act' (OODA) to map the compaction workflow within the framework. which is a model that has been similarly employed in Netflix's auto-optimize functionality [65], tailored to their specific use case. Our work leverages the same foundational concept but generalizes and modularizes the components of the compaction decision workflow, allowing AutoComp users to customize it according to their specific requirements. As illustrated in Figure 4, each of the four phases in the OODA model is associated with an input, an output, and an action that transforms the input into the output. We first generate compaction candidates that are used in the **observe** phase to extract relevant statistics needed for downstream decision-making. These statistics may include file-level metrics, as well as table or partition-level statistics specific to the candidate. A fine-grained approach to candidate generation directly supports **FR1**, ensuring that compaction tasks can be executed at sub-table levels for efficient resource management. The output of the observe

phase feeds into the **orient** phase, where the extracted statistics are used to generate *traits*. Traits are characteristics that describe either the current state of the candidate or its future potential. Examples of traits include file entropy or the estimated computational cost of rewriting data files for compaction. The use of traits allows AutoComp to support **FR2** by representing different decision strategies, which facilitate the ranking of candidates according to various objectives in the **decide** phase. In this phase, candidates are ranked based on a predefined ranking function, resulting in an ordered list for compaction, which is then processed in the **act** phase.

Optional filtering mechanisms are optional between the observe and orient phases, as well as between the orient and decide phases, to refine the candidate pool. Example filters might check the table size to skip tables that are too small or verify whether a compaction candidate has undergone recent frequent writes to avoid potential conflicts during compaction. AutoComp also supports an optional feedback loop from the act phase back to the observe phase. This feedback loop can include updated information such as the new number of partitioned files or layout changes, enabling continuous refinement of the compaction process.

AutoComp's architecture supports various modes of operation, including standalone execution on a schedule and proactive use triggered by specific events, aligning with **FR3**. This flexibility enables the framework to adapt to different operational needs without significant reconfiguration. Furthermore, the modular design of these phases supports **NFR1**, allowing new compaction strategies or decision criteria to be integrated seamlessly as long as the data exchanged between phases maintains a consistent structure. In addition, by choosing deterministic algorithms for each phase, AutoComp can address **NFR2**, making decision-making transparent and predictable. Finally, AutoComp can interface with different catalogs or LSTs through connectors that feed data into the system according to a consistent data model. This approach fulfills **NFR3**, enhancing the framework's reusability and extending its applicability to different data lake-centric platforms. Guided by AutoComp's workflow, we detail crucial implementation details necessary for building our compaction framework in §4. We then discuss execution strategies for triggering compaction in §5.

4 Implementation Details of AutoComp

This section covers identifying eligible entities for compaction, referred to as **candidate generation** (§4.1), our approach to **trait generation**, which outlines how system statistics are utilized effectively (§4.2), methods for objective-oriented **ranking** of candidates (§4.3), and the **scheduling** of selected candidates (§4.4).

4.1 Generation and Filtering of Candidates

In the following, we term a *candidate* a collection of files to be compacted. While this could represent an entire table, the scope of candidates can be adjusted to fit partitions or snapshots either manually or automatically. For example for larger tables, scoping candidates at the partition level enables parallel processing of multiple compaction tasks. Adjusting the scope to the snapshot level is particularly beneficial when (reasonably) fresh data needs more frequent access, ensuring performance objectives are met for a subset of the data. Candidates can be generated for a single scope or a combination of scopes within the workflow. Triggering the workflow for a single scope simplifies the downstream scheduling phase by eliminating the need to manage overlapping scopes. However, it is less flexible than considering the entire candidate space, as different table layouts may benefit from different scoping strategies.

Once candidates are generated, filtering mechanisms are applied throughout the workflow to refine the exhaustively generated candidate pool based on statistics and current table usage. The challenge to address is understanding how the tables containing these candidates are being utilized and applying filters accordingly. For example, we need to consider the impact of table deletions, table overwrites, or the creation of a table as an 'intermediate table' to avoid redundant or conflicting efforts. These filtering steps are specific to the platform where the framework is deployed and depend on the engines executing the workloads. For example in OpenHouse, we ensure that tables are not compacted if they have been created recently, i.e., within a preset time window. This approach enables us to avoid spending the computation budget on tables that are not going to affect the long-term health of the system.

Similarly, the specifics of extracting statistics during the observe phase are also dependent on platform characteristics. To modularize this step, we propose a standardized layout for statistics that accommodates both generic and custom metrics. Examples of generic statistics include the number of files in a candidate as well as their corresponding file sizes. Custom statistics, on the other hand, could include candidate access patterns and usage metrics–information that may not be available in all systems.

4.2 Trait Generation

The second phase, orient, uses statistics collected during the observe phase to calculate so-called *traits* that act as decision helpers for prioritizing and ranking candidates in the next step. Traits in AutoComp are defined independently of one another and can be partially combined during ranking. In our work, we primarily focus on two categories of traits: those describing the *benefit* of compaction, such as **file count reduction** and **file entropy** [65], and those representing its *cost*, such as **compute cost**. This combination enables a cost-benefit analysis to determine the most effective candidates for compaction.

File Count Reduction. For a given compaction candidate c, we estimate file count reduction after compaction, denoted as ΔF_c, as:

$$\Delta F_c = \sum_{i=1}^{FileCount_c} \mathbf{1}\left(FileSize_{i,c} < TargetFileSize_c\right)$$

The target file size is a configurable parameter that can be chosen based on factors such as the system setup. For instance, in HDFS deployments, it is often set to match the HDFS block size. The selection can also be influenced by workload characteristics. Further discussion on tuning such parameters is provided in §6.3.

Compute Cost. Compaction itself incurs costs that need to be considered, especially in a production environment where the benefit/cost ratio is crucial. For instance, if two candidates yield different file count reductions (e.g., 200 files versus 100 files) but share similar compute costs, the table with the greater reduction should be prioritized. However, if the compute cost for the first candidate is

significantly higher–perhaps due to larger file sizes–the benefit/cost ratio may favor the second candidate. In resource-constrained scenarios, compaction tasks must be managed within available capacity. Candidates with a compute cost that exceeds the allocated budget can be either automatically discarded or flagged for further review if the potential benefit justifies the higher cost.

To estimate the compute resources required to compact a candidate c, denoted as $GBHr_c$, we use:

$$GBHr_c = ExecutorMemoryGB \times \left(\frac{DataSize_c}{RewriteBytesPerHour} \right)$$

where *ExecutorMemoryGB* is the memory allocated to executors for processing the compaction task, $DataSize_c$ is the sum of the candidate files in bytes, and *RewriteBytesPerHour* indicates the system's throughput in terms of bytes that can be processed per hour. While this model focuses on executor memory, additional factors–such as compute units, disk, and network I/O–are left for future work.

4.3 Candidate Ranking and Selection

The core objective of the decide phase is to rank compaction candidates and prioritize them for execution. We consider two main scenarios for ranking, **unconstrained** resource availability and **resource-constrained** compaction systems.

Unconstrained Resource Scenario. When AutoComp operates without resource constraints, ranking is simplified to a decision function that selects candidates for (immediate) compaction when specific traits exceed predefined thresholds. For instance, an engine focused on maintaining optimal query performance might set a target to trigger compaction when the estimated file count reduction, ΔF_c, reaches at least 10%. In this scenario, when a table update occurs, candidates and their traits are generated during the observe and orient phases. If ΔF_c for any candidate indicates a potential file count reduction of 10% or more, the candidate is passed to the act phase for prompt execution. While this approach minimizes file count proactively and enhances user performance, it may also lead to inefficient resource use, particularly for temporary or non-critical tables (though custom filtering rules can be enabled in AutoComp to mitigate this). Additionally, frequent compactions can drive up resource costs, making this approach unsuitable for certain production environments.

Resource-Constrained Scenario. When AutoComp operates in environments where resources must be carefully managed, we propose ranking candidates based on a combination of traits to balance trade-offs, such as maximizing file count reduction while minimizing compute cost, and to align compaction tasks with available capacity. We formalize the candidate ranking process as a Multi-Objective Optimization Problem (MOOP) and scalarize it into a single-objective function using a weighted sum to simplify prioritization. To facilitate consistent comparisons, each trait is first normalized using min-max normalization:

$$T'_{i,c} = \frac{T_{i,c} - \min(T_i)}{\max(T_i) - min(T_i)}$$

where $T_{i,c}$ represents the actual value of trait i for candidate c, and $T'_{i,c}$ is its normalized value. This normalization scales trait values to a range of $[0, 1]$. Next, we define weights w_i for the objectives, ensuring that $\sum(w_i) = 1$. These weights indicate the relative importance of each trait within the MOOP function and can be adjusted dynamically to reflect current priorities. As an example, consider a MOOP function that maximizes the file count reduction while minimizing the associated compute cost as pointed out above. Here the scalarized score for a candidate c is expressed as:

$$S_c = w_1 \times T'_{1,c} - w_2 \times T'_{2,c}$$

where $T'_{1,c}$ represent normalized file count reduction and $T'_{2,c}$ normalized compute cost. Candidates are then ranked in descending order based on S_c, with higher scores indicating better overall performance relative to the specified objectives. To determine the available compute budget based on the cluster's characteristics, AutoComp can calculate it using available resources such as *ExecutorMemoryGB* and the predicted time to compact the chosen candidates. Alternatively, the compaction budget may vary depending on the production environment. For example, some production systems may instead use a fixed budget determined by capex and organizational limits to ensure to ensure compaction does not exceed preset constraints. After finalizing the available budget, AutoComp selects the top-k candidate compaction tasks, where k is the maximum number of candidates that fit within the budget. Note that the selection function may again differ depending on the production system; however, a reasonable greedy heuristic is to fit as many high-priority compaction tasks as possible within the budget.

By integrating these multi-objective considerations into the ranking phase, AutoComp ensures that compaction decisions are optimized for both performance and resource efficiency, dynamically adapting to operational constraints and shifting priorities.

4.4 Compaction Scheduling

The final step in the auto-compaction process is scheduling the selected compaction candidates as part of the act phase. Depending on the cluster configuration, compaction can be scheduled on the same cluster or offloaded to a dedicated compaction cluster to minimize the impact on user performance caused by high write operation volumes and resource utilization. In practice, AutoComp allows users to customize the scheduler to suit specific cluster needs. For example, when compaction runs on the same cluster as user transactions, compaction tasks can be scheduled sequentially to mitigate resource contention or deferred to off-peak hours if usage patterns are predictable. Furthermore, the choice of LST also influences scheduling decisions. For instance, in our experiments with Apache Iceberg v1.2.0 and OpenHouse, we observed that, counterintuitively, compaction operations executed concurrently could result in conflicts when targeting distinct partitions within a table, leading to failed compaction attempts. Thus, any scheduling algorithm must consider the specific characteristics of the chosen LST, not only in terms of conflict resolution mechanisms but also task failure, recovery, and checkpointing during compaction [10], which can impact scheduling decisions.

5 Automatic Data Compaction

With candidates for compaction identified, we next need to determine when to trigger compaction. Instead of relying on manual intervention, we envision automatic scheduling and execution of compaction operations based on factors such as cluster health, compute resources availability, and other relevant metrics. Automatic

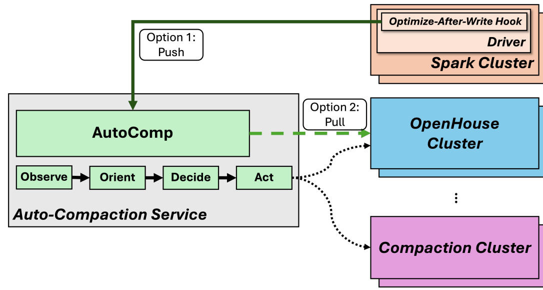


Figure 5: Cluster integration of AutoComp.

compaction can be implemented in two different ways: (*i*) *Optimize-After-Write*, where a candidate's potential for compaction is evaluated each time its files are modified, and (*ii*) *Periodic Compaction*, which runs the compaction workflow at regular intervals, such as once per day, to assess and schedule compaction as part of periodic evaluation of the data lake's state.

Optimize-After-Write. Several existing architectures [1, 32, 41] leverage hooks integrated within the engine to enable automatic compaction in response to write modifications, 'pushing' the compaction decision onto the engine. The same traits described earlier can be used as triggers; if a trait value surpasses a defined threshold, a compaction operation can either be triggered immediately or the optimize-after-write hook can notify the auto-compaction service that changes have occurred and related candidates' traits need recalculation. The immediate triggering approach ensures the table remains in an optimal state but requires an unlimited compaction budget. Its alternative, which decouples the hook from scheduling, provides more flexibility in terms of resource usage, allowing for controlled trait generation and efficient compaction task execution.

Periodic Compaction. Instead of modifying engine drivers directly, we can choose to implement auto-compaction as a standalone service [6, 64, 65], potentially integrated into a catalog or control plane like OpenHouse. This service runs independently, periodically evaluating whether compaction criteria are met, 'pulling' information about the state of the cluster and scheduling compaction accordingly. It is especially advantageous in scenarios with predictable compaction cycles, such as scheduling compaction when cluster utilization is low during off-peak hours or ensuring that compaction does not interfere with other active workloads.

These strategies integrate seamlessly with AutoComp's workflow, as depicted in Figure 5. Here, AutoComp functions as a standalone component that supports both push and pull operations, allowing for the (re-)calculation of a candidate's traits either triggered by a hook or retrieved periodically from the OpenHouse cluster.

6 Evaluation of Compaction Framework

This section presents the evaluation of AutoComp using synthetic workloads, focusing on its effectiveness in mitigating the impact of small file proliferation in LSTs.

Cluster Infrastructure and Configuration. We conducted the experiments on clusters running Apache Spark v3.1.1 and Apache Iceberg v1.2.0 libraries, mirroring the setup used in LinkedIn's production environment. We specified standard configurations for both the driver and executor nodes and enabled Adaptive Query Execution (AQE) [17]. The query-processing cluster consisted of one driver node and 15 executor nodes, while the compaction cluster used one driver node and three executor nodes. Both clusters were provisioned using Azure VMSS, with each node being an Azure Standard E8s v3 instance (Intel® Xeon® CPU E5-2673 v4 @ 2.30GHz, 8 virtual cores, 64GB RAM). OpenHouse v0.5.131 was deployed on a separate Azure Kubernetes Service (AKS) cluster using its default Terraform configuration [53]. The AutoComp extension was configured to run periodically and triggered Spark compaction jobs based on its decision logic. The data for the experiments was stored in Azure Data Lake Storage Gen2 (ADLS) [42]. In addition to Iceberg metadata tables [9], we leveraged Logs Analytics [45] to monitor telemetry data across different services.

Design of Experimental Workloads. We used the *CAB-gen* tool [24, 68] to generate metadata for multiple databases and query streams, modeled after real-world usage patterns in cloud data warehouse environments [72]. The database schemas are based on the TPC-H schema, while the query streams mimic usage patterns such as constant demand with sinusoidal variations (e.g., dashboards), short bursts (e.g., interactive queries), large bursts (e.g., daily maintenance jobs), and predictable workloads triggered at specific times (e.g., hourly jobs). The *CAB-gen* tool required several parameters: raw *data size*, *number of databases*, *CPU time* (representing total computational workload), and *execution time* (duration of the experiment). For our test scenario, we set the parameters to 500GB of data, 20 databases, 1 total CPU hours, and 5 hours of experiment time. After generating the database definitions with *CAB-gen*, we used the *dbgen* tool from the TPC-H benchmark [66] to generate synthetic data. The LINEITEM table was partitioned by SHIPDATE with monthly granularity, producing a workload with mixed data update patterns across partitioned (LINEITEM) and non-partitioned (ORDERS) tables[1]. For query execution, we extended the *LST-Bench* benchmarking tool [38] to run the streams produced by *CAB-gen*. These extensions are now available in OSS LST-Bench [39].

Candidate Selection and Scheduling. Our synthetic experiments focus on three different candidate selection strategies: (*i*) no compaction, (*ii*) *table*-scope compaction, and (*iii*) a *hybrid* compaction strategy that chooses partition-scope compaction if the table is partitioned and otherwise defaults to table-scope. The table-scope compaction mimics the current OpenHouse implementation while the hybrid strategy explores whether partition-based compaction can help to balance the resource utilization load. Candidates are compacted in parallel on the table level but sequentially on the partition level as we have noticed compaction operations getting dropped due to conflicts even for distinct partitions otherwise, see §4.4 for details. Compaction execution is triggered every hour of the experiment, i.e., a successful experiment should contain four compaction executions in a 5 hour timeframe.

Metrics. We capture both client- and server-side statistics for a comprehensive understanding of the impact of compaction on workload execution. On the client side, we focus primarily on workload query execution times and the number of errors observed during execution. On the server side, we gather several compaction-related metrics, including current file counts for tables, rewritten bytes,

[1]The original CAB-gen only generated updates on the ORDERS table; we extended this to include updates on both ORDERS and LINEITEM tables.

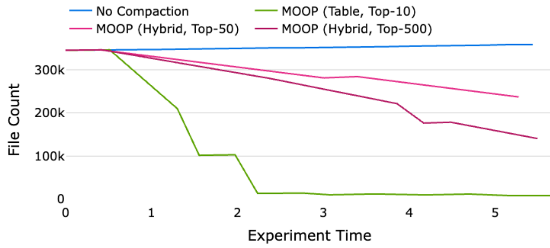


Figure 6: Compaction strategy impact on file count over time.

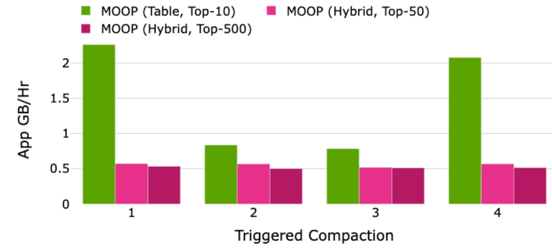


Figure 7: Mean $GBHr_{App}$ for various compaction strategies.

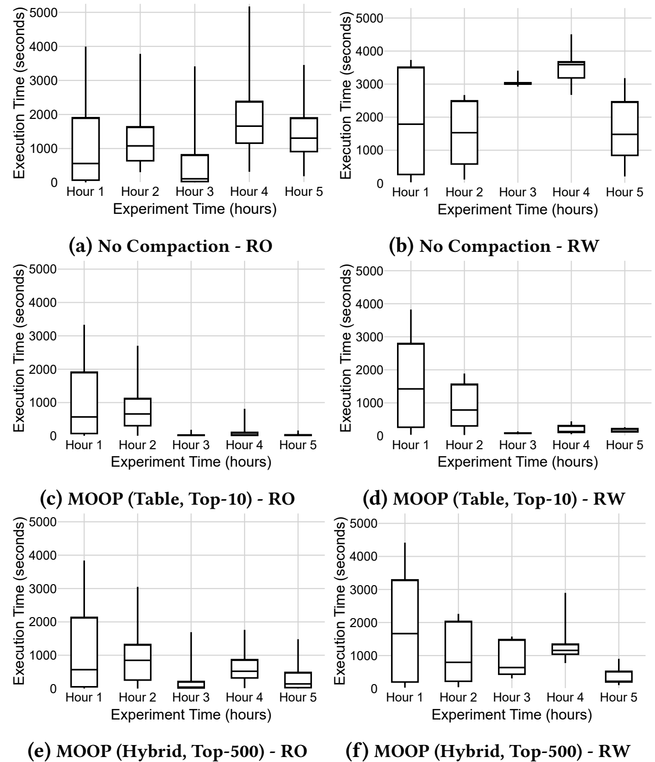
and added files. In addition, we compute a custom metric, $GBHr_{App}$, which reflects the compute resources needed by an application *App*; here, an application is defined at the job level, meaning each triggered compaction operation is treated as a distinct instance.

6.1 File Count

Our first goal is to evaluate AutoComp's effectiveness in handling HDFS' *small files* problem in the storage layer. We run the CAB workload on our query-processing cluster, executing streams for the 20 databases concurrently while our compaction strategies operated on the separate compaction cluster, triggered at 1-hour intervals. Figure 6 shows the file count over time for the baseline with no compaction and for AutoComp using the MOOP strategy that balances the benefit of reducing the estimated number of small files against the cost of rewriting a table or partition. We set k– the number of work units compacted in each AutoComp run– to 10 for table-scope compaction and 50 resp. 500 for the hybrid compaction strategy, the target file size to 512 MB, and the weights for MOOP to 0.7 (file count reduction) and 0.3 (computation cost), mimicking our OpenHouse deployment. Note that in practice, we may choose to vary the value of k depending on constraints such as available compaction resources or for a gradual rollout in a production environment. The values chosen for visualization here exemplify trends that we can see across a range of k values.

Storage Layer Changes. In our baseline (no compaction), we observe a high initial file count, as the data load operation generates many small files–a common scenario in practice due to factors like cluster misconfiguration (§2). During the experiment, the file count increases steadily, with an average increase of approximately 2,640 files per hour, although the exact number fluctuates with the write queries executed during each interval. On average, the experiment runs for about five hours, with a noticeable spike in data write operations around hour four due to workload patterns that increase load on the query-processing cluster during that time window. With compaction enabled, we observe a significant reduction in file count across all compaction strategies, with an initial sharp decline in file count followed by a more gradual flattening of the curve. For the *hybrid* strategies, the reduction curve is less steep, as fewer entities are compacted in each round, leading to more gradual, controlled reduction in file count.

Compaction Cost. While each compaction strategy reduces file count, it is also important to consider the cost associated with compaction. Figure 7 shows the average $GBHr_{App}$ for compaction across strategies during the experiment. Compaction at the *table*



(a) No Compaction - RO (b) No Compaction - RW

(c) MOOP (Table, Top-10) - RO (d) MOOP (Table, Top-10) - RW

(e) MOOP (Hybrid, Top-500) - RO (f) MOOP (Hybrid, Top-500) - RW

Figure 8: Impact of compaction on query latency.

level can be advantageous and more effective when a table layout is highly fragmented, but a finer-grained approach, such as the *hybrid* method with partition-level compaction, provides more control, allowing file count reductions at a slower pace and thus balancing resource usage for compaction over time as documented with a more stable value for $GBHr_{App}$ across compaction operations.

6.2 Query Performance

The proliferation of small files can significantly impact query performance. To evaluate AutoComp's effect on query performance, we measured query execution times over the course of our experiments. Figure 8 shows execution times for both read-only (left column) and read-write (right column) queries under no compaction and various compaction strategies. Each candlestick bar represents the min, 25th percentile, median, 75th percentile, and max execution times per hour. Focusing first on read-only queries, we observe that

Table 1: Client and cluster-side conflicts per execution hour.

Hour	# Write Queries	Client-side Conflict			Cluster-side Conflict	
		NoComp	Table-10	Hybrid-500	Table-10	Hybrid-500
2	12	1	**11**	4	**23**	**0**
3	5	**0**	2	**0**	**17**	**0**
4	15	1	5	6	4	**0**
5	8	4	**0**	2	**0**	**0**

performance across all strategies is similar in the first hour, but from the second hour onward, compaction consistently improves query performance, with faster reductions in execution times under the more aggressive compaction strategy (*table*, top-10). Additionally, execution time variability decreases, as shorter query runtimes help reduce resource contention in the query-processing cluster. For the less aggressive strategy (*hybrid*, top-500), we also see significant performance improvements, but not at the level of the more aggressive strategy which is expected and correlates with our previous findings relating to the number of files successfully compacted and thus decreasing the system's file count. However, we note that the experiment's end-to-end runtime for both compaction strategies meets the pre-set 5-hour limit, while the *no compaction* baseline incurs an additional 25 minutes of overhead due to queuing and longer query execution times.

In addition to the execution time of the workload queries, we also examined their retry behavior due to write-write conflicts as shown in Table 1. Here, we show queries that have been retried due to client-side errors, i.e., versioning conflicts that cause a client-side operation to terminate, and cluster-side conflicts that occur during compaction operations. We observe that conflicts are present even without compaction due to concurrent writes to the same tables and commonly correlate with spikes in workload patterns as shown by the number of write queries issued within a specified hour. An exception is our experiment for table-scope compaction. We observe that conflicts occur early during the experiment due to a large number of compaction operations and subsequent conflicts about stale metadata. By the fifth hour, the tables with write activity–namely, lineitem and orders–have been largely compacted, leading to a reduction in write-write conflicts. Interestingly, we observe no cluster-side conflicts for our hybrid approach, suggesting that the probability of disrupted compaction operations decreases with the size of the candidate to be compacted which is expected.

6.3 Auto-Tuning Compaction Triggers

As noted earlier, a key challenge in auto-compaction is determining parameter values that best fit a given workload. Thus, we experiment with an auto-tuning framework in conjunction with AutoComp, using a simplified optimize-after-write hook setup, i.e., unlimited compaction resources. We use two compaction traits–small file count and file entropy [65]–and tune the thresholds that determine when compaction is triggered. As before, we deploy LST-Bench with three of its in-built workloads: TPC-DS WP1, a long-running workload with frequent data modifications; TPC-DS WP3, where one compute cluster handles all writes while another handles all reads; and TPC-H. Both TPC-DS and TPC-H datasets use a scale factor of 100, with experiments running on a 16-node Spark cluster (plus a 7-node sidecar cluster for writes in TPC-DS WP3), using Delta Lake v2.4.0 as the LST. This demonstrates AutoComp's flexible design,

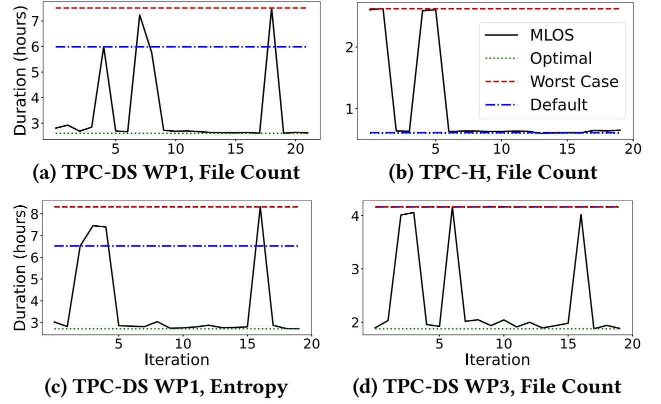


Figure 9: Comparison of compaction decisions and results.

enabling support for various LST implementations. To optimize parameters, we leverage the FLAML optimizer [73] implemented within MLOS [34, 48], an open-source optimization framework, to iteratively refine threshold values. Figure 9 presents the results, with the y-axis showing total end-to-end experiment duration and the x-axis representing iterations, each with a threshold selected by MLOS, leading to the following observations. (*i*) For TPC-H (Figure 9b), the default setting (no auto-compaction) performs best, as compaction rewrites entire non-partitioned tables, making it costly, and its long data modification phase already dominates execution time. In contrast, TPC-DS WP1 (Figure 9a) benefits from compaction when tables become too fragmented, reducing query time by up to 2× when applied appropriately. Finally, TPC-DS WP3 (Figure 9d) sees consistent benefits from compaction, as its decoupled read and write clusters minimize resource contention with other queries. (*ii*) We observe similar query performance when using small file count- and entropy-based triggers, as shown in Figure 9a and Figure 9c. This suggests both decision functions can yield comparable results depending on the chosen thresholds. Note that here, we experiment with single-trait decision functions only; more complex approaches, such as multi-objective ranking functions that consider computation cost, may lead to different outcomes.

Overall, these experiments demonstrate that auto-tuning a compaction framework like AutoComp is a promising direction for future exploration, which we discuss further in §8.

7 AutoComp Impact in Practice

As discussed in §2, the initial solution to LinkedIn's small file problem was manual intervention via compaction tasks. However, this approach quickly proved infeasible at scale. To address the challenge of compacting over 35*K* tables in LinkedIn's OpenHouse deployment, we implemented an instantiation of AutoComp with the following characteristics. First, we scoped compaction at the table level, consistent with the existing manual compaction strategy. Second, we combined the file count reduction estimator and compute cost calculator introduced in §4.2 within the MOOP ranking function described in §4.3. We tuned the MOOP weights to reflect our specific objectives, adjusting the file count reduction weight (w_1) based on a database's quota utilization–measured by its total

number of files or namespace objects. Each database represents a logical group of tables associated with a specific tenant:

$$w_1 = 0.5 \times \left(1 + \left(\frac{UsedQuota}{TotalQuota}\right)\right)$$

Here, *TotalQuota* is the HDFS namespace quota (in number of filesystem objects) allocated to a database, and *UsedQuota* is the currently utilized portion. Finally, we implemented a periodic scheduling strategy that triggers once daily, selecting a set of k compaction candidates. Fixing the number of candidates was critical during initial rollout to ensure predictable behavior, a key requirement when introducing a new (automated) mechanism into production. Over time, we transitioned to dynamically selecting k based on available compaction resources.

Given our experience with both manual and automatic compaction, we share several observations from production deployments: (*i*) whether manual compaction alone is sufficient in production, (*ii*) how compaction has shifted file distributions in OpenHouse, and (*iii*) how auto-compaction impacts user workload execution and HDFS metadata operations.

Diminishing Returns of Manual Compaction. Our initial mitigation approach was an ad-hoc manual compaction strategy that repeatedly compacted a fixed set of $k \approx 100$ tables at a high frequency (e.g., daily). These tables were chosen because of their susceptibility to high fragmentation, and early results showed marked improvements: reduced small file counts, improved query performance, and lower storage overhead. However, these benefits tapered off over time. Once small files were merged, further compaction yielded limited gains. Figure 2 shows that the file size distribution remained largely unchanged between the second and third month of manual compaction. In general, we observe that not all tables benefit equally from compaction in LinkedIn's OpenHouse deployment. While some tables significantly benefit from compaction at a low cost, others incur a high cost with minimal benefits. In practice, identifying high-impact candidates is non-trivial, as users interact with the system on a daily basis by modifying their data, creating new tables, and adjusting workflows. As a result, manually defining compaction targets is suboptimal, and motivated our shift towards an automated solution.

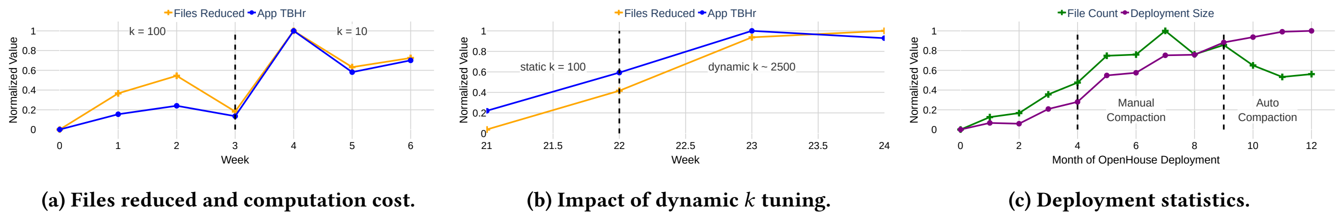
Deploying Compaction Mechanisms. Deploying automatic compaction has significantly alleviated the small file problem in OpenHouse. Specifically, prior to deployment, users encountered frequent issues, including: (*i*) query failures caused by HDFS read timeouts due to excessive RPC traffic, (*ii*) frequent breaches of user HDFS namespace quotas, and (*iii*) rapid growth in object count, requiring frequent HDFS federations to distribute the load. As shown in our motivating example, Figure 2, prior to compaction tasks being executed regularly, 83% of the system's files were smaller than 128MB. When we introduced manual compaction, we saw a significant shift in overall file distribution with the percentage of small files dropping from 83% to 62%. We further reduced this number by gradually rolling out AutoComp as part of the OpenHouse compaction decision mechanism. We began by deploying AutoComp with a highly conservative choice of k, i.e., $k \approx 10$, to closely examine the impact of compaction in our production environment without disrupting users. Interestingly, we observed that switching from manual top-100 compaction to automatic top-10 compaction strategy effectively increased overall file count reduction, even though we compacted 10× fewer tables. More specifically, we observed an average reduction of 6.59 million files via manual compaction versus 7.44 million using AutoComp with a top-10 selection–an improvement of 12%. Figure 10a further shows the relationship between file count reduction and compaction cost (measured in App TBHr) over a 6-week period. The transition from manual (k=100) to auto-compaction (k=10) was done in week 3, resulting in both higher effectiveness and higher computation cost. Figure 10b illustrates the transition in week 22 of the auto-compaction deployment from fixed to dynamic k selection, constrained by the maximum allocated compaction budget. With a budget of 226 TBHr, we successfully compacted around $k \approx 2500$ tables per iteration of auto-compaction. Overall, we observe that despite the growing deployment, auto-compaction mechanisms have significantly decreased the file count in HDFS over time, as shown in Figure 10c.

Model Accuracy and Estimation Errors. We evaluated the accuracy of our estimators by comparing predicted and actual values for file count reduction and compute cost. Unfortunately, we occasionally observe a discrepancy between these values. For example, we estimated a compute cost of 108 TBHr for one compaction task, but actually consumed 129 TBHr (a 19% underestimation), while the file count reduction was overestimated by 28%. These mismatches suggest that while the current model is generally effective for ranking, it requires further refinement to improve accuracy–particularly in accounting for partition boundaries, as table-level estimates may overestimate the number of small files that can be merged, since compaction does not cross partitions.

Impact of Compaction on Workloads and HDFS. We also analyzed the performance of a scan-heavy workload that runs daily, in conjunction with AutoComp. To assess the impact of periodic auto-compaction on this workload, we plot the number of files scanned during workload execution and correlated it with query execution time and query cost, as shown in Figure 11a. The chart captures normalized observations across 1291 unique tables chosen by AutoComp for compaction over the most recent 30-day window.

We observe a strong correlation between compaction runs that reduce file counts and subsequent decreases in files scanned during query execution. Furthermore, the reduction in files scanned closely corresponds to a decrease in query execution time and query cost, measured in App TBHr. However, when tables were not selected by AutoComp for compaction in a given cycle, small files accumulated again–resulting in a recurring sawtooth pattern.

Overall, we observe that despite the increasing size of our deployment over time, the introduction of manual compaction in month 4 and auto-compaction in month 9 resulted in a significant reduction in filesystem open() calls on HDFS as shown in Figure 11b. The sharp decline observed in month 4 coincides with the introduction of manual compaction on a subset of heavily fragmented tables, each comprising an average of 42M small files with an average size of 64 MB. The queries against these tables had previously exhibited frequent HDFS read timeouts due to excessive RPC traffic to the NameNode. Such timeouts often led to simultaneous client retries, exacerbating the load and triggering a thundering herd problem.



(a) Files reduced and computation cost.

(b) Impact of dynamic k tuning.

(c) Deployment statistics.

Figure 10: AutoComp behavior and impact on file count.

(a) Key workload metrics.

(b) HDFS filesystem operations

Figure 11: Impact of AutoComp on workload metrics, including file scanning, query execution, and HDFS file opens.

8 Discussion and Future Directions

The lightweight design of LST implementations such as Apache Iceberg, combined with the deployment of control planes like OpenHouse in data lake-centric architectures, provides an effective setup for defining and implementing data reorganization strategies. Through our exploration of the auto-compaction problem both in practice and using our experimental setup, we have identified several exciting directions for future research and innovation in this space.

Navigating Multi-Objective Trade-offs. In this work, we approached the file compaction problem as a multi-objective optimization task with two primary goals: maximizing file count reduction and minimizing compute cost during compaction. Our current method computes a single solution by applying weighted objectives, where the weights reflect our chosen priorities. This single solution represents a specific trade-off, providing what we consider to be the *best* answer given the current objective weights and constraints. While this approach has shown good results in practice, it inherently risks overemphasizing one metric at the expense of the other by collapsing multiple objectives into a single weighted score. In some cases, the chosen solution may not align well with changing system conditions or varying operational requirements, as the optimal balance between file count reduction and compute cost can shift based on workload patterns, resource availability, or specific database needs. To address these challenges, we propose exploring the use of the Pareto frontier in future work to offer broader perspective on the trade-offs involved. Instead of converging on a single *best* answer, leveraging the frontier would allow us to generate a set of Pareto-optimal solutions, each representing a unique balance between file count reduction and compute cost. Solutions on the Pareto frontier are non-dominated, meaning that improving one objective would necessarily worsen the other. To compute weights dynamically, we propose leveraging regression analysis techniques in machine learning, enabling us to move beyond the reliance on fixed weights for different objectives.

Conflict Resolution. Our experience revealed that understanding LST conflict resolution mechanisms and predicting potential conflicts is challenging. For example, the experiments with OpenHouse revealed unexpected compaction conflicts involving disjoint partitions, suggesting potential gaps in the *conflict filtering* implementation. Recent work, including approaches leveraging formal verification methods, addresses knowledge gaps in conflict resolution for LSTs [69–71]. AutoComp already separates scheduling as an independent step to support diverse LST conflict resolution strategies, and engine-specific behaviors during maintenance tasks like compaction may still require additional extensions.

Automatic Data Layout Optimization. While compaction focuses primarily on managing small files and fragmentation, it can be extended to address broader data layout optimization strategies. For instance, data clustering techniques–such as Z-ordering or V-ordering [35, 44, 55]–can improve compression ratios, encoding efficiency, and query performance by co-locating related data. These techniques are complementary to compaction and can be integrated into AutoComp's decision-making process. Achieving this integration would require extensions to both the candidate generation phase and the computation of traits. For example, some layout optimizations operate at the file level, while others apply at coarser granularities such as partitions or tables. These differing scopes would need to be considered during candidate generation. Similarly, new traits would need to account for both the benefits of these optimizations–such as improved compression or filtering efficiency–and their costs, including computational overheads like data sampling or multiple data passes.

Workload Awareness. Incorporating workload-awareness into trait computation can further refine AutoComp's decision-making process by aligning layout optimizations with query patterns and access frequency, potentially leading to improvements in query performance. Moreover, partitioning and clustering strategies, chosen with query patterns in mind, can also influence the efficiency of writes and compaction by reducing unnecessary data conflict

errors [70]. Therefore, the choice of data layout optimization strategy should account not only for query workloads but also for their broader implications on compaction and conflict resolution.

Tuning Write and Compaction Mechanisms and Policies. Engines and LSTs expose a wide range of configuration parameters that significantly influence data layout on write. For instance, Spark's adaptive query execution framework may inadvertently choose an excessively small shuffle partition size for final writes or a suboptimal distribution mode for table setup, resulting in an excessive number of small files [51]. In large organizations like LinkedIn, engineers may not have direct control over engine configurations across all workloads. Control planes like OpenHouse thus offer a valuable opportunity to analyze and surface such issues, with actionable insights for stakeholders. This increased visibility enables timely recommendations to manually mitigate these challenges. Compaction tasks in LSTs, as well as AutoComp itself, offer configurable parameters that influence auto-compaction behavior, as discussed in §6.3. Our experiments suggested that "one size does not fit all": compaction triggers and data layout strategies should ideally be tailored to individual workloads rather than standardized for all engines. However, workload-specific auto-tuning is computationally expensive and in our experiments, each iteration required multiple hours and consumed significant cluster resources. Optimizing experimentation time and reducing computational costs will be critical to making auto-tuning feasible for practical use.

9 Related Work

The study of automatic compaction has become crucial with the adaptation of DBMS-like structures to LSTs within general-purpose distributed storage systems. For example, foundational work by [60] introduced *delta files* to mitigate write amplification caused by updates, a concept now central to LST implementations.

Database Defragmentation. [59] analyzed how object size and data fragmentation affect system performance in the context of a DBMS, highlighting two key findings: First, that optimal data layouts are workload-dependent, and second, that fragmentation over time leads to significant performance degradation. DBMS have historically addressed fragmentation using both online and offline approaches. Online methods involve human oversight for reorganization. For instance, [50] proposed an online approach for index defragmentation in modern databases using a what-if API to estimate performance benefits. Their algorithm performs range-level index defragmentation, analogous to fine-grained compaction scopes in AutoComp. The system recommends optimal strategies, but a DBA ultimately schedules and triggers defragmentation. In contrast, offline approaches, such as the one adopted by AutoComp, rely on automated algorithms for reorganization. For example, [33] described a multi-level index structure that stores mutable data on magnetic disks and immutable archival data on write-once-read-many optical disks. A size-based algorithm triggers a vacuum process to migrate data, optimizing read and write latency while reducing storage costs. Modern in-memory databases [19, 29, 62] apply similar principles by maintaining separate write and read-optimized regions, with data migration triggered by thresholds for size and time. Similarly, [58] extended these concepts with the bLSM tree, combining B-Tree and LSM tree functionalities. Their spring-and-gear scheduler balances compactions across levels, ensuring predictable throughput and consistent latency for uniform workloads. Recent work has also systematically explored the LSM compaction design space, analyzing trade-offs across different strategies and workloads [57]. LSTs such as Hudi and the recent Apache Paimon [14] incorporate these principles–write and read-optimized regions, along with automatic compaction–directly into their core implementations.

Automatic Compaction in Data Lakes. Some engines running on data lakes, while not exposing their table format as LSTs interoperable across engines, employ similar techniques and still produce numerous small files at the storage layer. Apache Hive [23] introduced ACID-compliant tables built on HDFS, employing compaction triggered by thresholds for delta file counts and fragmentation ratios. Its design separates cleaning and merging phases to minimize query disruptions, similar to the techniques used in LSTs. Nova [52] implemented comparable compaction and cleaning tasks for Apache Pig workflows, but without automation, relying on manual triggers. Similarly, [2] proposed a stand-alone compaction server for HBase, isolating compaction tasks from user workloads to improve system efficiency. Our AutoComp deployment, tested in both synthetic experiments and production at LinkedIn, adopts a similar approach. However, AutoComp offers enhanced flexibility, supporting execution in different clusters and multiple operational modes, adapting to diverse workload and system requirements.

Automatic Layout Tuning in LSTs. Auto-tuning mechanisms have become a critical challenge in LSTs, with recent efforts [1, 6, 20, 32, 41, 64, 65] highlighting the need for flexible solutions to address data layout challenges. AutoComp advances these efforts as the first comprehensive proposal for automatic compaction that offers flexibility to adapt to various algorithms, models, and operational scenarios, ensuring compatibility with diverse workloads and infrastructure setups.

10 Conclusion

In this paper, we introduced AutoComp, an automated compaction framework to address small file proliferation, a common problem in data lake infrastructure settings. AutoComp is designed according to both functional and non-functional requirements resulting from our deployment at LinkedIn, and its usefulness is shown experimentally with both synthetic and real-world deployments. We observe that key features such as multi-objective optimization functions and workload-aware compaction strategies and schedules improve the compaction results in our deployments significantly, not only impacting the storage layer positively but also query performance. We further list a variety of future research opportunities in the broader area of data layout optimization which we think will enhance data lakes effectively moving forward.

Acknowledgments

We would like to thank the entire OpenHouse team at LinkedIn for their collective efforts and contributions. We also extend our heartfelt thanks to Mridul Muralidharan for his thoughtful review of early drafts of this paper and for offering valuable insights.

References

[1] Josep Aguilar-Saborit, Raghu Ramakrishnan, Kevin Bocksrocker, Alan Halverson, Konstantin Kosinsky, Ryan O'Connor, Nadejda Poliakova, Moe Shafiei, Taewoo Kim, Phil Kon-Kim, Haris Mahmud-Ansari, Blazej Matuszyk, Matt Miles, Sumin Mohanan, Cristian Petculescu, Ishan Rahesh-Madan, Emma Rose-Wirshing, and Elias Yousefi. 2024. Extending Polaris to Support Transactions. In *ACM SIGMOD*.

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-Value Datastores. In *PVLDB*.

[3] Amazon. 2025. AWS Pricing Calculator. https://calculator.aws.

[4] Amazon. 2025. S3 - Cloud Object Storage. https://aws.amazon.com/s3/.

[5] Apache Hudi. 2025. https://hudi.apache.org/.

[6] Apache Hudi. 2025. Compaction. https://hudi.apache.org/docs/0.13.1/compaction/.

[7] Apache Iceberg. 2025. https://iceberg.apache.org/.

[8] Apache Iceberg. 2025. Compaction. https://iceberg.apache.org/docs/1.4.3/maintenance/#compact-data-files.

[9] Apache Iceberg. 2025. Metadata Tables. https://iceberg.apache.org/docs/1.4.3/spark-queries/#all-metadata-tables.

[10] Apache Iceberg. 2025. Rewrite Data Files Parameters. https://iceberg.apache.org/javadoc/1.4.3/org/apache/iceberg/actions/RewriteDataFiles.html.

[11] Apache Iceberg. 2025. Table Migration. https://iceberg.apache.org/docs/1.4.3/spark-procedures/#table-migration.

[12] Apache ORC. 2025. https://orc.apache.org/.

[13] Apache Ozone. 2025. https://ozone.apache.org/.

[14] Apache Paimon (incubating). 2025. https://paimon.apache.org/.

[15] Apache Parquet. 2025. https://parquet.apache.org/.

[16] Apache Polaris (incubating). 2025. https://polaris.apache.org/.

[17] Apache Spark. 2025. Adaptive Query Execution. https://archive.apache.org/dist/spark/docs/3.1.1/sql-performance-tuning.html#adaptive-query-execution.

[18] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. In *PVLDB*.

[19] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *ACM SIGMOD*.

[20] Jeff Barr. 2024. Amazon S3 Tables: Storage optimized for analytics workloads. https://aws.amazon.com/blogs/aws/new-amazon-s3-tables-storage-optimized-for-analytics-workloads/

[21] Ryan Blue. 2023. *Hello, World of CDC!* https://tabular.io/blog/hello-world-of-cdc/

[22] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2024. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. In *ACM SIGMOD*.

[23] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *ACM SIGMOD*.

[24] Cloud Analytics Benchmark (CAB) Tool. 2025. https://github.com/alexandervanrenen/cab.

[25] Databricks. 2025. Announcing General Availability of Predictive Optimization. https://www.databricks.com/blog/announcing-general-availability-predictive-optimization.

[26] Databricks. 2025. Databricks Platform. https://www.databricks.com/.

[27] Delta Lake. 2025. https://delta.io/.

[28] Delta Lake. 2025. Compaction. https://docs.delta.io/2.4.0/optimizations-oss.html#compaction-bin-packing.

[29] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *ACM SIGMOD*.

[30] Google Cloud. 2025. Cloud Storage. https://cloud.google.com/storage.

[31] Google Cloud. 2025. Google Cloud Pricing Calculator. https://cloud.google.com/products/calculator.

[32] Avijit Goswami and Rajarshi Sarkar. 2023. Apache Iceberg optimization: Solving the small files problem in Amazon EMR. https://aws.amazon.com/blogs/big-data/apache-iceberg-optimization-solving-the-small-files-problem-in-amazon-emr/

[33] Curtis P. Kolovson and Michael Stonebraker. 1989. Indexing Techniques for Historical Databases. In *IEEE ICDE*.

[34] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. In *PVLDB*.

[35] Alexey Kudinkin and Tao Meng. 2021. *Hudi Z-Order and Hilbert Space Filling Curves*. https://hudi.apache.org/blog/2021/12/29/hudi-zorder-and-hilbert-space-filling-curves/

[36] Zihan Li, Sudarshan Vasudevan, Lei Sun, and Shirshanka Das. 2021. *FastIngest: Low-latency Gobblin with Apache Iceberg and ORC format*. https://www.linkedin.com/blog/engineering/open-source/fastingest-low-latency-gobblin

[37] LinkedIn. 2025. OpenHouse. https://github.com/linkedin/openhouse/.

[38] LST-Bench. 2025. https://github.com/microsoft/lst-bench.

[39] LST-Bench. 2025. Cloud Analytics Benchmark (CAB) Integration. https://github.com/microsoft/lst-bench/issues/335.

[40] Alex Merced. 2022. Compaction in Apache Iceberg: Fine-Tuning Your Iceberg Table's Data Files. https://www.dremio.com/blog/compaction-in-apache-iceberg-fine-tuning-your-iceberg-tables-data-files/

[41] Microsoft. 2025. Auto compaction for Delta Lake on Azure Databricks. https://learn.microsoft.com/en-us/azure/databricks/delta/tune-file-size#--auto-compaction-for-delta-lake-on-azure-databricks.

[42] Microsoft. 2025. Azure Data Lake Storage. https://azure.microsoft.com/products/storage/data-lake-storage.

[43] Microsoft. 2025. Azure Pricing Calculator. https://azure.microsoft.com/pricing/calculator/.

[44] Microsoft. 2025. Delta Lake table optimization and V-Order. https://learn.microsoft.com/fabric/data-engineering/delta-optimization-and-v-order.

[45] Microsoft. 2025. Log Analytics in Azure Monitor. https://learn.microsoft.com/azure/azure-monitor/logs/log-analytics-overview.

[46] Microsoft. 2025. Microsoft Fabric. https://www.microsoft.com/microsoft-fabric.

[47] Microsoft. 2025. The need for optimize write on Apache Spark. https://learn.microsoft.com/azure/synapse-analytics/spark/optimize-write-for-apache-spark.

[48] MLOS. 2025. https://github.com/microsoft/MLOS.

[49] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *PVLDB*.

[50] Vivek R. Narasayya and Manoj Syamala. 2010. Workload driven index defragmentation. In *IEEE ICDE*.

[51] Anton Okolnychyi, Chao Sun, Kazuyuki Tanimura, Russell Spitzer, Ryan Blue, Szehon Ho, Yufei Gu, Vishwanath Lakkundi, and D. B. Tsai. 2024. Petabyte-Scale Row-Level Operations in Data Lakehouses. In *PVLDB*.

[52] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B. N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. 2011. Nova: continuous Pig/Hadoop workflows. In *ACM SIGMOD*.

[53] OpenHouse. 2025. Azure deployment using Terraform scripts. https://github.com/linkedin/openhouse/tree/main/infra/recipes/terraform/azure.

[54] Matthew Powers. 2023. *Delta Lake - Small File Compaction with OPTIMIZE*. https://delta.io/blog/2023-01-25-delta-lake-small-file-compaction-optimize/

[55] Matthew Powers. 2023. *Delta Lake - Z Order*. https://delta.io/blog/2023-06-03-delta-lake-z-order/

[56] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *ACM SIGMOD*.

[57] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. In *PVLDB*.

[58] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *ACM SIGMOD*.

[59] Russell Sears and Catharine van Ingen. 2007. Fragmentation in Large Object Repositories. In *CIDR*.

[60] Dennis G. Severance and Guy M. Lohman. 1976. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.* 1, 3 (1976), 256–267.

[61] Konstantin V. Shvachko, Chen Liang, and Simbarashe Dzinamarira. 2021. *The exabyte club: LinkedIn's journey of scaling the Hadoop Distributed File System*. https://www.linkedin.com/blog/engineering/open-source/the-exabyte-club-linkedin-s-journey-of-scaling-the-hadoop-distr

[62] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *ACM SIGMOD*.

[63] Snowflake. 2023. The Snowflake Platform. https://www.snowflake.com/.

[64] Sébastien Stormacq. 2023. AWS Glue Data Catalog now supports automatic compaction of Apache Iceberg tables. https://aws.amazon.com/blogs/aws/aws-glue-data-catalog-now-supports-automatic-compaction-of-apache-iceberg-tables/

[65] Anupom Syam. 2023. Optimizing data warehouse storage. https://netflixtechblog.com/optimizing-data-warehouse-storage-7b94a48fdcbe

[66] TPC. 2021. TPC-DS Specification Version 3.2.0. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf.

[67] Unity Catalog. 2025. https://www.unitycatalog.io/.

[68] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. In *PVLDB*.

[69] Jack Vanlightly. 2024. *Understanding Apache Hudi's Consistency Model*. https://jack-vanlightly.com/analyses/2024/4/24/understanding-apache-hudi-

consistency-model-part-1
[70] Jack Vanlightly. 2024. *Understanding Apache Iceberg's Consistency Model.* https://jack-vanlightly.com/analyses/2024/7/30/understanding-apache-icebergs-consistency-model-part1
[71] Jack Vanlightly. 2024. *Understanding Delta Lake's consistency model.* https://jack-vanlightly.com/analyses/2024/4/29/understanding-delta-lakes-consistency-model
[72] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *USENIX NSDI*.
[73] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: A Fast and Lightweight AutoML Library. In *MLSys*.