레이크하우스상에서 실체화 뷰의 지원*†‡

김경민°, 이상원
서울대학교
{kyungminkim, swlee69}@snu.ac.kr

Supporting Materialized Views on Data Lakehouses

Kyungmin Kim, Sang-Won Lee
Seoul National University

Abstract

Materialized view (MV) has long been a core optimization technique for boosting analytical queries in traditional database management systems (DBMS). The rise of cloud-native data architectures and lakehouse systems built on open table formats (OTFs), however, calls for a need to fundamentally reiterate MV design. In this paper, we investigate the opportunities of enabling MVs in cloud-native lakehouse environments and present a MV architecture that supports incremental view maintenance (IVM) and transparent query rewriting. The key design principle is to exploit the immutable snapshot model of lakehouse systems to capture fine-grained data deltas for efficient IVM. We implement our approach in Apache Spark SQL and Apache Iceberg, and evaluate it on the TPC-H benchmark. Our results show significant reductions in end-to-end query latency compared to baseline execution without MVs.

1 Introduction

MV is a traditional optimization for caching precomputed query results, reducing latency and improving performance by avoiding redundant computation on expensive joins and aggregations [11, 12]. However, in traditional on-premise DBMS environments, its adoption has been limited. Maintaining MVs consistently and rewriting queries to leverage them is complex, and selecting an optimal set of views under storage constraints remains a hard problem [8, 9]. As a result, practical deployments have often been sparse and application-specific [11].

Cloud-native lakehouse systems shift many of these assumptions. Unlike monolithic DBMSs, lakehouses separate compute from storage and manage data in versioned, columnar formats such as Apache Iceberg [1], Delta Lake [10], or Apache Hudi [2]. While disaggregation introduces additional network I/O and latency when scanning remote storage, it also opens new opportunities for MVs.

We focus on Apache Iceberg deployed on MinIO [3], a lightweight, high-performance storage layer compatible with S3. Iceberg's append-only design supports snapshot-based maintenance and fine-grained delta tracking for not only inserts but also updates and deletes via merge-on-read (MoR). This capability allows incremental maintenance of MVs.

In this paper, we present a system for MVs tailored to the lakehouse setting. Our design targets workloads dominated by joins and aggregations and supports IVM through snapshot differencing. Using the TPC-H benchmark, we demonstrate significant reductions in query latency compared to execution over raw tables.

Contribution: IVM atop lakehouse systems. While several vendors (e.g., Databricks, StarRocks) advertise MV support over OTFs [4, 5], their internal implementations remain proprietary. Our prototype supports IVM for built-in aggregate functions (e.g., `SUM, COUNT, AVG`) and multi-table joins. By supporting MVs following the specification discussed in Iceberg community [6], we reveal the internal implementation details.

2 Background and Motivation

2.1 Background

MVs are a long-standing optimization in DBMSs, explicitly storing precomputed results of queries to accelerate response times and reduce resource consumption [11]. Unlike virtual views, they consume additional storage but substantially speed up queries over large joins and aggregations, and can serve as reusable building blocks for related queries.

MVs face three persistent challenges despite these advantages [11]:

- **Freshness:** MVs must be kept consistent with base tables to avoid stale results.

*) This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00454666,Developing a Vector DB for Long-Term Memory Storage of Hyperscale AI Models)
†) This work was supported by an Industry- Academia collaborative project with Dnotitia (Project No. 2025-SNU-001)
‡) This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(RS-2025-25402724)

- **Selection:** Determining which MVs to materialize under storage constraints is NP-hard [8, 9].

- **Query rewrite:** Query optimizers must substitute MVs for base tables without sacrificing correctness.

These issues are common across both on-premise and cloud-native environments, but their implications differ significantly in modern systems.

2.2 Motivation

The rise of lakehouses in cloud-native environment introduces new opportunities for MV design. They provide transactional guarantees, schema evolution, and abundant cost-efficient storage that can be exploited for supporting MVs and their maintenance. Now, MV in lakehouse can effectively state traditional problems of MV we outlined earlier.

Addressing freshness with snapshot isolation. OTFs track versioned snapshots, allowing MVs to be incrementally refreshed by scanning only files added or deleted between snapshots, rather than recomputing the entire view [1, 10]. This mechanism naturally supports IVM and additionally enables time-travel queries for reproducible analytics.

Relaxing MV selection with the cloud-native cost model. Cloud-native environments offer cheap, scalable object storage and relatively expensive compute. This cost asymmetry encourages aggressive materialization. Instead of optimizing for a minimal MV set, systems can persist a broad range of candidate MVs without prohibitive overhead. Thus, the selection problem is greatly simplified.

3 Design and Implementation Details

3.1 Overall Implementation Design

We design a snapshot-based MV framework on Spark SQL and Iceberg. Iceberg's metadata hierarchy and immutable, append-only snapshots align naturally with delta-based MV refresh, while Spark provides the execution engine. We actively adopt standard specification from ongoing efforts toward native MV support of Iceberg community [6].

MV is implemented as a fully materialized Iceberg table. This allows us to reuse existing primitives such as snapshot management, scan pruning, and time-travel queries. By embedding MV metadata in Iceberg's existing view structures, we extend the system without breaking compatibility. This design ensures that MV management benefits from Iceberg's ongoing evolution while minimizing additional complexity.

3.2 Implementations

3.2.1 Incremental View Maintenance Algorithm

IVM in traditional MV systems (e.g., PostgreSQL extension pg_ivm [7]) rely on triggers and additional tables to store deltas. By contrast, our approach leverages Iceberg's snapshot model. Each snapshot introduces immutable data files for inserts and delete files for removals, eliminating the need for extra efforts to capture deltas. However, our current prototype implementation only works with the insert-only behavior, that deleting or updating the residing base tables need a full refresh.

It is worth highlighting that our design follows the Iceberg community's recommendation of defining a standard specification for MVs. The key idea is to introduce additional metadata fields and align them with existing ones. The system determines validity by comparing the view's `current-version-id` with the table's `refresh-version-id`. A mismatch marks the MV as *invalid*, requiring a full recomputation. If the IDs match, the system further checks the source-table snapshot IDs from the last refresh against those of the current state. When all entries align, the MV is considered *fresh*; otherwise, it is labeled *stale* and subject to either incremental or full refresh.

3.2.2 Transparent Query Rewrite for MVs

Query rewrite ensures MVs are used without user intervention. We extend Spark SQL's Catalyst optimizer with additional rules to identify candidate MVs. Currently, we support substitution when the query's projections and filters exactly match those of MVs.

3.2.3 Supported Operations

Our prototype supports the full MV lifecycle. `CREATE` registers the MV definition and materializes the initial results. `REFRESH` can be triggered automatically or manually and applies append-only IVM by default, with full rebuilds reserved for exceptional cases. Queries over MVs are treated identically to regular Iceberg tables, and `DROP` removes both its definition and physical storage.

4 Experiments

We ran a TPC-H benchmark of scale factor 10 with our implementation and compared the performance of 1) baseline, for full table scan, 2) full refresh, that utilizes MV but refreshing the MV

Table 1: Benchmark Experiment Environment

Hardware Environment	
Model Name	MacBook Pro
Chip	Apple M3 Pro
Total Cores	11 (5 Performance + 6 Efficiency)
Memory	18 GB
Storage	APPLE SSD AP0512Z, 500 GB
Software Environment	
Operating System	macOS 26.0 Tahoe
Query Engine	Spark SQL 4.0

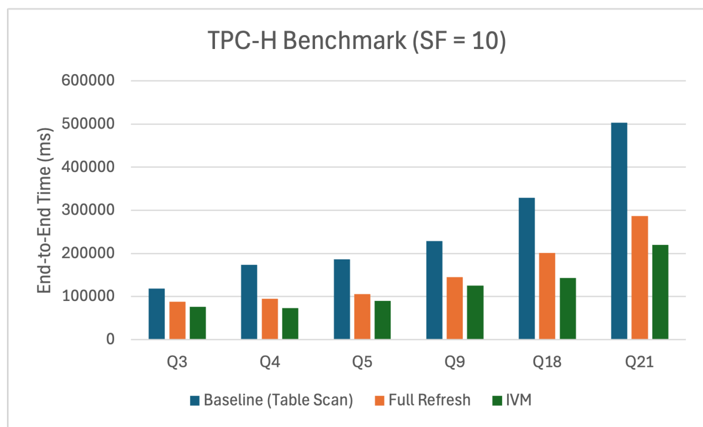

Figure 1: TPC-H benchmark result

fully, and 3) MV with IVM. We use our custom workload scenario for benchmarking, consisting of `SELECT` and `INSERT` queries with a 3:1 read-to-write ratio. The experiment environment is Table 1.

The experimental result Figure 1 shows clear performance differences among the three approaches. The baseline table-scan method consistently exhibited the slowest execution times. Compared with it, full refresh improved performance by approximately 45–55% on average. IVM achieved a further 15–25% improvement over full refresh for most queries, and for large, complex queries such as Q18 and Q21 the performance gain exceeded 30%.

5 Conclusion and Future Works

We argue that the shift from on-premises to cloud-native systems necessitates rethinking the role of MVs, exploiting opportunities enabled by the new architecture. We implemented a novel MV paradigm that aligns with the Iceberg community's ongoing discussion on defining a standard MV specification for lakehouse systems. Our evaluation demonstrates that this approach is feasible, delivering significant performance gains over full table scans and even outperforming full MV recomputation.

However, several directions remain for future work still: (1) supporting delete- and update-aware IVM, (2) further exploiting statistics and metadata available in lakehouse systems and columnar formats such as Parquet, and (3) optimizing IVM through storage-layer computation pushdown and pruning techniques leveraging inter-table relationships (e.g., foreign key constraints [13]). Ongoing research continues to explore these opportunities in cloud-native architectures, and we hope our approach represents a small but meaningful step forward in this direction.

References

[1] https://iceberg.apache.org/.

[2] https://hudi.apache.org/.

[3] https://www.min.io/.

[4] https://www.databricks.com/blog/announcing-general-availability-materialized-views-and-streaming-tables-databricks-sql.

[5] https://docs.starrocks.io/docs/using_starrocks/async_mv/use_cases/data_lake_query_acceleration_with_materialized_views.

[6] https://github.com/apache/iceberg/issues/10043.

[7] https://github.com/sraoss/pg_ivm/.

[8] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[9] R. Ahmed, R. Bello, A. Witkowski, and P. Kumar. Automated generation of materialized views in oracle. *Proceedings of the VLDB Endowment*, 13(12):3046–3058, 2020.

[10] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, and M. Zaharia. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13:3411–3424, 08 2020.

[11] R. Chirkova, J. Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[12] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *arXiv preprint arXiv:1509.07454*, 2015.

[13] C. Svingos, A. Hernich, H. Gildhoff, Y. Papakonstantinou, and Y. Ioannidis. Foreign keys open the door for faster incremental view maintenance. *Proc. ACM Manag. Data*, 1(1), May 2023.