Enzyme Demo: Incremental View Maintenance for Data Engineering

Yuhong Chen
Ritwik Yadav
Min Yang
Supun Abeysinghe
Databricks, Inc.
San Francisco, California, USA
enzyme-paper@databricks.com

Manuel Ung
Jeffrey Helt
Michael Armbrust
Shrikanth Shankar
Databricks, Inc.
San Francisco, California, USA
enzyme-paper@databricks.com

ABSTRACT

This demonstration presents Enzyme, the incremental view maintenance (IVM) engine powering Spark Declarative Pipelines. While most industrial systems lack broad SQL operator coverage or require users to hand-tune refresh strategies, Enzyme provides an out-of-the-box solution for incremental ETL pipelines. Beyond classical delta-plan construction, it introduces techniques for incrementalizing non-deterministic operators such as temporal filters, a multi-versioned fingerprinter stable across system upgrades and multilingual MV definitions, and an adaptive cost model that leverages historical execution profiles to optimize refresh strategies across entire dependency DAGs. This leads to a reduced total cost of ownership (TCO) for users, who can focus on developing business logic instead.

Enzyme builds on top of Apache Spark primitives and offers an end-to-end optimization layer for the selection of optimal refresh strategies corresponding to a collection of materialized views organized into a dataflow pipeline. The design is modular and can be easily extended to a variety of data sources and query processing engines.

PVLDB Reference Format:
Yuhong Chen, Ritwik Yadav, Min Yang, Supun Abeysinghe, Manuel Ung, Jeffrey Helt, Michael Armbrust, and Shrikanth Shankar. Enzyme Demo: Incremental View Maintenance for Data Engineering. PVLDB, 19(12): 4818 - 4821, 2026.
doi:10.14778/3827998.3828130

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/enzyme-paper/enzyme-demo.

1 INTRODUCTION & RELATED WORK

Materialized views (MVs) are a fundamental building block in many database systems, traditionally used for reducing query execution costs by amortizing expensive computation. However, MVs have since evolved from query acceleration constructs to become essential components in modern data flow architectures, enabling efficient extract, transform, and load (ETL) workflows at scale.

Traditionally, data pipelines were implemented using specialized, hand-crafted logic that read from source tables, applied a sequence of transformations, and explicitly updated target tables. MVs simplify this process by allowing users to focus on *what* should be computed rather than *how* to maintain those computations. The engine assumes full responsibility for keeping the view consistent with evolving source data by automatically deriving and executing the logic required to incrementally process changes to the source tables and propagate only the necessary updates to the MV. This automated maintenance process is commonly referred to as incremental view maintenance (IVM).

Prior Work. IVM has been extensively studied in database research for several decades. Early approaches [4, 9] focused on deriving algebraic rewrite rules that compute delta queries at the operator level, enabling efficient propagation of changes from base relations to derived MVs. DBToaster [10] advances this line of work by recursively applying such rewrite rules to incrementally maintain MVs through a hierarchy of sub-materializations. However, the choice of intermediate results to materialize is highly workload-dependent, and suboptimal materialization strategies can significantly degrade performance.

DBSP [3] represents a more recent and comprehensive treatment of IVM. It models databases and their updates as streams, defining incremental counterparts for each query operator using differentiation and integration primitives. These counterparts compute output deltas from input modifications, avoiding full reprocessing. The arbitrary composition of these incremental operators makes the model extremely powerful. However, DBSP does not discuss practical concerns such as efficiently identifying input changes or optimal physical implementations for incremental operators. For example, nonlinear aggregates do not necessarily need to process the full dataset in a brute force manner.

Battle-tested production systems [1, 10, 11] provide more insight into the challenges behind efficient implementation of incremental engines. The algorithm for updating an MV operates synergistically with other optimization techniques, including the materialization of intermediate results and the efficient batching of input changes. These strategies collectively enhance the performance of IVM by reducing redundant computations, optimizing resource usage, and minimizing latency during updates.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828130

Enzyme. In Databricks, users can create MVs either within Spark Declarative Pipelines or through Databricks SQL warehouses. MVs are refreshed on a user-specified schedule or triggered automatically based on changes to their input tables. Each refresh updates the MV so that it logically reflects the result of re-running its defining query over the latest versions of all inputs. Enzyme is the incremental view maintenance engine which powers MV refresh at Databricks. It keeps track of the versions of the inputs that were used to update an MV during the last successful refresh. Changes to the inputs which have accumulated since then are computed and used to update the MV incrementally. The Enzyme cost model determines if the accumulated changes make incremental refresh more expensive than full recompute and chooses the optimal strategy for each MV.

Enzyme is implemented using primitives provided by Apache Spark SQL [2], constructing alternative query plans using MERGE INTO [5] and REPLACE WHERE [7] statements.

Demo Scenario. In this demonstration, we showcase Enzyme's capabilities through an interactive e-commerce analytics pipeline. We illustrate the breadth of SQL operators supported by Enzyme, including aggregations over joins, window functions, and temporal filters. We also demonstrate how the cost model automatically selects the optimal refresh strategy based on workload characteristics. Whereas the companion paper [11] establishes Enzyme's design and evaluates it at scale, the contribution of *this* paper is the interactive demonstration itself: attendees author and mutate live MV definitions, drive changes of varying magnitude through the pipeline, and watch Enzyme's refresh-strategy decisions unfold in real time—an experience the system paper cannot convey.

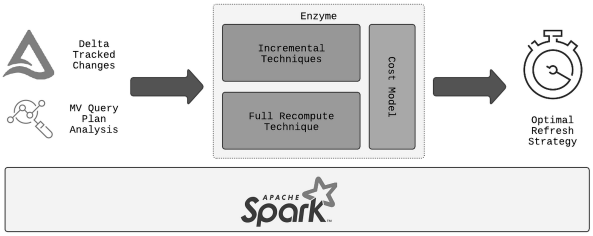

Figure 1: Enzyme IVM engine. It analyzes the changes to the inputs of an MV and the query plan of its definition to select the optimal refresh strategy.

2 DESIGN OVERVIEW

This section summarizes only the parts of Enzyme's architecture needed to explain the demonstration (Section 3). A comprehensive treatment of the production system—its incrementalization algorithms, formal model, and large-scale experimental evaluation—appears in our companion paper [11].

A simplified overview of Enzyme flow is shown in Figure 1. It utilizes primitives from Apache Spark to build and compare alternative refresh plans for updating MVs. The design ensures that there is no duplication of query processing code. Enzyme processes MV definitions through several main stages as demonstrated in Figure 2.

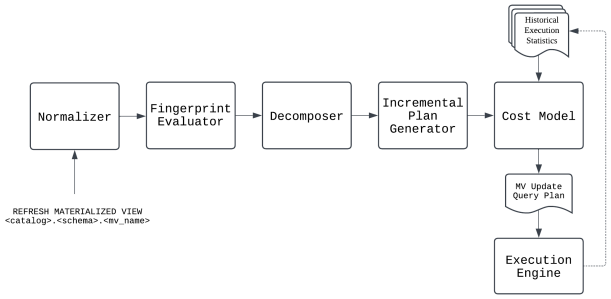

Figure 2: Components of the Enzyme IVM engine.

2.1 Normalization

The main entry point for Enzyme is a logical plan that represents the MV definition, allowing Enzyme to support all Spark frontends, including SQL and the DataFrame API. The normalizer canonicalizes the logical plan by applying simplification rules including: inlining common table expressions, merging filter predicates, collapsing unnecessary projections, and eliminating redundant operations. This is necessary before computing a unique fingerprint and because some Spark optimizations obscure information essential for incremental processing.

2.2 Query Fingerprinter

The fingerprinter creates a deterministic fingerprint of each normalized plan, allowing Enzyme to detect semantic changes in the MV's definition across refreshes. It applies canonicalization steps such as standardizing the ordering of commutative operations. The fingerprinter also tracks UDFs and their dependencies, ensuring that changes to user code trigger appropriate recomputes. A critical challenge is maintaining fingerprint stability across system upgrades; Enzyme addresses this through multi-versioning of fingerprints, preventing cascading recomputes during deployments.

2.3 Decomposition & Technique Enablers

Databricks MVs are represented internally as a reconciliation view on top of a table. Enzyme leverages this through *technique enabler* transformations to make incremental refreshes more efficient.

An example of an enabler is decomposing aggregate functions into incrementally computable components. For example, an MV containing AVG(x) is internally augmented with columns storing SUM(x) and COUNT(*), enabling efficient incremental updates. Another enabler is explicitly propagating aggregate and window keys to the output, which can be exploited by the incremental plan generator. The reconciliation view hides the extra columns added by technique enablers.

The decomposer also handles composite row identifiers for tracking data lineage. Row identifiers [6] are used for efficiently filtering rows in an MV that need updating. For joins, each output row gets a composite identifier constructed from the contributing row identifiers from each joined table, enabling precise identification of affected rows during incremental refresh.

2.4 Incremental Plan Generator

The incremental plan generator transforms a normalized and enabled query plan into one that computes the MV's incremental changes. It employs a recursive visitor pattern that traverses the query plan bottom-up, producing at each node a representation of how to compute changes to that node's output given changes to its inputs.

At the core is the concept of a *changeset computation*, consisting of three logical components: the pre-state (data before changes), post-state (data after changes), and delta (the change representation with insertion/deletion markers). As the visitor traverses each operator, it transforms the changesets from child nodes into a changeset for the current node, enabling arbitrary operator composition.

2.5 Cost Model & Execution

Incrementally refreshing every MV might not always be computationally cheaper than fully recomputing it. The computational complexity depends on factors including enablement of features such as deletion vectors [8], properties of the input changes, and hardware specification. For example, recomputes can be cheaper in the event of a large upstream backfill, or with changes that affect most of the aggregation groups in an MV.

The Enzyme cost model takes these factors into account and selects the optimal refresh strategy. Refreshes for the same MV run at predictable intervals, making it easier to incorporate feedback from previous runs into decision making. The logical query plan is then submitted to Spark's optimizer and execution engine, with provenance updates committed atomically alongside data changes.

3 DEMONSTRATION SCENARIOS

We demonstrate Enzyme through a realistic e-commerce analytics pipeline. Users can trigger data changes to observe how Enzyme selects and executes optimal refresh strategies. The associated code and video are available at https://github.com/enzyme-paper/enzyme-demo and https://youtu.be/hEYWMzrJbFo, respectively.

The demo is fully hands-on. Attendees can edit MV definitions live (adding columns, changing predicates, or swapping operators) and watch Enzyme re-fingerprint and re-plan; inject changes of varying size (single-row inserts, order returns that trigger window recomputation, large backfills) to push the workload across the cost model's incremental-vs-recompute boundary; and inspect the resulting refresh plans, per-MV cost estimates, and execution times in the event log. This lets the audience reproduce each effect from the walkthrough and try out their own queries.

We invite readers to try executing these MVs themselves in Databricks Free Edition.

3.1 Scenario: E-commerce Analytics Dashboard

We base our demonstration around a realistic scenario: an online retailer's analytics dashboard that tracks sales, customer behavior, and product performance. For the purposes of the demo, we assume the inputs are already ingested into the Data Warehouse. The pipeline processes data from three source tables that simulate a production e-commerce system:

- **customers**: 10M customer records with regional and membership tier information.
- **sales**: 500M transaction records with order details.
- **products**: 1M product catalog entries with category and pricing information.

The MVs in the pipeline power different components of the retailer's analytics dashboard, from customer lifetime value tracking to daily sales reporting.

3.2 Demonstration Walkthrough

3.2.1 Aggregate over Join Queries: Regional Customer View. This MV illustrates *aggregation over a* `LEFT JOIN`: on new sales, Enzyme updates only the affected customer aggregates instead of re-evaluating the join (Query 1).

```
CREATE MATERIALIZED VIEW mv_regional_customers AS
  SELECT c.region, c.membership_tier, c.customer_id,
         COUNT(*) AS order_count,
         SUM(s.quantity * s.unit_price) AS
              total_spend
  FROM customers c
  LEFT JOIN sales s ON c.customer_id = s.customer_id
  GROUP BY c.region, c.membership_tier, c.
       customer_id;
```

Query 1: Regional customers with aggregation over LEFT JOIN

3.2.2 Window Functions: Customer Lifetime Value. This MV illustrates *incremental window functions*: when orders are marked returned, Enzyme recomputes only the affected customer partitions rather than the entire window (Query 2).

```
CREATE MATERIALIZED VIEW mv_customer_ltv AS
  SELECT customer_id, sale_date,
         SUM(quantity * unit_price) OVER w AS
              cumulative_spend,
         COUNT(*) OVER w AS order_number
  FROM sales
  WHERE status != 'Returned'
  WINDOW w AS (PARTITION BY customer_id ORDER BY
       sale_date);
```

Query 2: Customer LTV with window functions

3.2.3 Composable Aggregation: Category Performance. This MV illustrates *composable aggregation*—nested aggregates where Enzyme computes deltas for the inner aggregation and propagates them to the outer one, while the decomposer maintains hidden `SUM`/`COUNT` columns to keep `AVG` incremental (Query 3).

```
CREATE MATERIALIZED VIEW mv_category_perf AS
  SELECT category,
         SUM(daily_revenue) AS total_revenue,
         AVG(daily_revenue) AS avg_daily_revenue,
         MAX(daily_orders)  AS peak_orders
  FROM (
    SELECT p.category, s.sale_date,
           SUM(s.quantity * s.unit_price) AS
                daily_revenue,
           COUNT(*) AS daily_orders
    FROM sales s JOIN products p USING (product_id)
    GROUP BY p.category, s.sale_date
```

```
  )
  GROUP BY category;
```

Query 3: Category performance with composable aggregation

3.2.4 Temporal Filters: Daily Sales Report. This MV illustrates *temporal filter handling*: the sliding 90-day predicate ages rows out without explicit updates, so Enzyme accounts for aged-out rows when maintaining the aggregates, not just inserted and deleted ones (Query 4).

```
CREATE MATERIALIZED VIEW mv_daily_sales AS
  SELECT s.sale_date, s.channel, p.category,
         COUNT(*) AS order_count,
         SUM(s.quantity * s.unit_price) AS revenue
  FROM sales s JOIN products p USING (product_id)
  WHERE s.status IN ('Completed', 'Shipped', '
      Delivered')
    AND s.sale_date >= DATE_SUB(CURRENT_DATE(), 90)
  GROUP BY s.sale_date, s.channel, p.category;
```

Query 4: Daily sales with temporal filter and aggregation

3.3 Numerical Comparison

Figure 3 shows execution times when each MV is initialized using full recompute, while Figure 4 shows the improvement achieved through incremental refresh with small change sets. For the demo workload, the processing time for each MV was reduced by ~50%.

The performance gains are most pronounced for MVs with expensive operations like JOINs and window functions, where incremental refresh avoids reprocessing hundreds of millions of unchanged rows. For `mv_regional_customers`, incremental refresh processes only the new sales records rather than re-joining all 500M sales with 10M customers.

Figure 3: Full recompute execution times for each MV in the demo pipeline. These serve as the baseline against which incremental refresh is compared.

Figure 4: Incremental refresh execution times for the same MVs after changes to source tables.

3.4 Cost Model Decisions

While incremental refresh provides significant improvements over recomputes for small upstream changes, recomputes can be cheaper if most of the MV contents need to be replaced. For example, when a large backfill is run on the upstream. Figure 5 shows the cost model automatically choosing the optimal refresh strategy without user intervention.

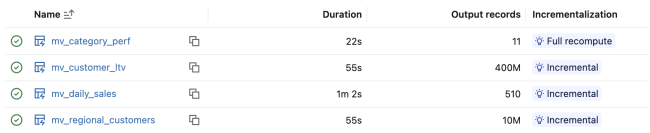

Figure 5: The cost model intelligently selects full recomputes when the change volume makes it more efficient

4 CONCLUSION

This demo presents Enzyme, Databricks' incremental view maintenance engine, through an interactive e-commerce analytics pipeline. The demo illustrates the rich technique coverage of Enzyme and the significant performance improvements in incremental updates. We also show how the cost model intelligently falls back to full recompute when the change volume makes it more efficient. Enzyme's handling of non-deterministic operators, multi-versioned fingerprinting, and adaptive pipeline-aware costing go beyond the classical IVM literature and, to our knowledge, any prior IVM system.

REFERENCES

[1] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, et al. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[3] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: Automatic incremental view maintenance for rich query languages. *arXiv preprint arXiv:2203.16684* (2022).

[4] Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance.. In *VLDB*, Vol. 91. 577–589.

[5] Databricks. 2024. Merge Into. Retrieved Jan 24, 2025 from https://docs.databricks.com/en/sql/language-manual/delta-merge-into.html

[6] Databricks. 2024. Use row tracking for Delta tables. Retrieved Nov 1, 2025 from https://docs.databricks.com/aws/en/delta/row-tracking

[7] Databricks. 2025. Selectively overwrite data with Delta Lake. Retrieved Nov 1, 2025 from https://docs.databricks.com/aws/en/delta/selective-overwrite

[8] Databricks. 2025. What are deletion vectors? Retrieved Nov 1, 2025 from https://docs.databricks.com/aws/en/delta/deletion-vectors

[9] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[10] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal* 23 (2014), 253–278.

[11] Ritwik Yadav, Supun Abeysinghe, Min Yang, Jeffrey Helt, Manuel Ung, Yuhong Chen, Melody Hu, William Wei, Yiming Yang, Tom van Bussel, Sourav Chatterji, Indrajit Roy, Paul Lappas, Yannis Papakonstantinou, Tahir Fayyaz, Bilal Aslam, Ross Bunker, Michael Armbrust, and Shrikanth Shankar. 2026. Enzyme: Incremental View Maintenance for Data Engineering. In *SIGMOD-Companion '26*. To appear. Preprint: https://drive.google.com/file/d/1HRUqRttSuoQJdGW-bUfcTa7rIaBm4LNw/view?usp=sharing.