Streaming View: An Efficient Data Processing Engine for Modern Real-time Data Warehouse of Alibaba Cloud

Fangyuan Zhang* Chunlei Xu Jiyu Qiao Hua Fan Wenchao Zhou
Mengqi Wu* Yunong Bao Yingli Zhou Caihua Yin Feifei Li
Alibaba Cloud Computing Hangzhou, China
{zhangfangyuan.zfy,mengqi.wmq,mengda.xcl,baoyunong.byn,qiaojiyu.qjy}
{zhouyingli.zyi,guanming.fh,caihua.ych,zwc231487,lifeifei}
@alibaba-inc.com

ABSTRACT

Real-time data warehouses are essential for modern applications. Extract-Transform-Load (ETL) as a fundamental component of offline data warehouses also provides crucial support within real-time data warehouses. Among various traditional ETL approaches, Lambda and Kappa have emerged as classic real-time data processing solutions due to their freshness and query performance, which best meet business demands. However, both of them often require the integration of external stream processing engines, introducing challenges related to complexity, efficiency, and consistency. ZeroETL has emerged as an approach to address these issues. Nevertheless, existing ZeroETL-based solutions primarily emphasize the implementation of extraction and loading, resulting in limitations in handling transformation. Incremental View Maintenance (IVM) offers an alternative that can enhance ZeroETL. However, existing IVM implementations often focus on query acceleration rather than supporting high-throughput, complex real-time workloads.

To address these challenges, we propose Streaming View, an efficient real-time data processing engine integrated within AnalyticDB of Alibaba Cloud. Unlike existing solutions, Streaming View supports high-throughput, complex data processing for real-time streaming ETL workloads. Furthermore, it can be leveraged to optimize ZeroETL-based approaches by enhancing transformation capabilities. We design tailored algorithms and optimizations for diverse syntaxes and high-throughput scenarios, ensuring the system meets complex application needs. By integrating incremental computation into the data warehouse, Streaming View reduces complexity, ensures data consistency, and boosts performance, offering a robust solution for real-world applications. Experiments show Streaming View improves processing performance by up to 7x and 20x over traditional ETL and IVM methods, respectively, and addresses complex scenarios unsolved by existing solutions.

PVLDB Reference Format:
Fangyuan Zhang, Mengqi Wu, Chunlei Xu, Yunong Bao, Jiyu Qiao, Yingli Zhou, Hua Fan, Caihua Yin, Wenchao Zhou, and Feifei Li. Streaming View: An Efficient Data Processing Engine for Modern Real-time Data Warehouse of Alibaba Cloud. PVLDB, 18(12): 5153 - 5165, 2025.
doi:10.14778/3750601.3750634

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750634

1 INTRODUCTION

With the rapid evolution of internet technologies and the shift of modern enterprises toward data-driven decision-making, the demand for real-time data analytics [49] within data warehouses is more critical than ever. Meanwhile, surging real-time data volumes and rising business complexity—driven by dynamic pricing [37] and personalized recommendations [18] in e-commerce [33], real-time anti-fraud and anomaly detection in finance [50], and continuous sensor data ingestion for fault detection and predictive maintenance in industrial IoT [45]—pose greater challenges to real-time data warehouses in throughput, latency, and performance.

Challenges in real-time data warehousing. Traditionally, data warehouses have been designed as batch-processing systems optimized for large-scale historical analysis [25, 46, 49]. Consequently, in the Extract-Transform-Load (ETL) processes of offline data warehouses, most data processing tasks are accomplished using batch processing techniques. In contrast, real-time data warehouses adopt a variety of data processing paradigms, including stream processing, batch/micro-batch processing, pure real-time computation, and incremental processing. Each of these paradigms offers unique advantages and trade-offs across dimensions such as data freshness, performance, efficiency, and implementation complexity. Among these approaches, the widely used Lambda [47] and Kappa [34] architectures perform real-time data processing based on the paradigm illustrated in Figure 1. They leverage standalone stream processing engines such as Apache Flink [7, 14] and Spark Streaming [9, 48]. They provide substantial improvements over traditional batch/micro-batch and pure real-time computation approaches in terms of data freshness and query performance, while also offering a more comprehensive and robust solution compared to current incremental processing techniques. However, as real-time data volumes and business complexities increase, standalone stream processing engines are facing growing challenges. Firstly, their integration with existing data warehouses significantly increases system complexity and operational overhead [41], as it requires additional efforts in system integration, debugging, and monitoring. Furthermore, the inclusion of additional system components leads to increased data movement and processing overhead, which can degrade overall performance and delay query responses, particularly in high-throughput scenarios. Additionally, ensuring strong consistency across streaming engines and data warehouses

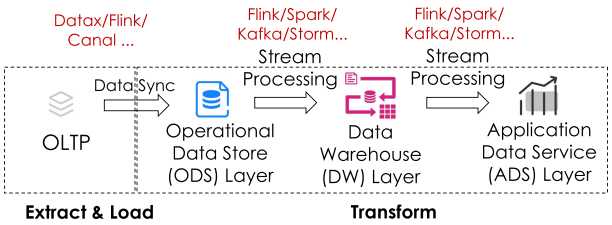

Figure 1: A common real-time data warehouse architecture with integrated streaming ETL and processing engines.

becomes a significant challenge, especially during schema changes, data corrections, or system failures, as these operations often require coordinated updates across multiple systems. Finally, this approach struggles to effectively leverage the sophisticated query optimization techniques prevalent in mature database systems, leading to suboptimal data processing performance.

The emergence of ZeroETL and its limitations. To address these challenges, ZeroETL [5] has emerged as an approach aimed at eliminating traditional ETL pipelines. ZeroETL-based solutions are implemented by establishing a direct underlying connection between data in OLTP and OLAP systems, thereby removing the necessity for additional data synchronization tools and pipelines. Existing ZeroETL-based solutions primarily emphasize the implementation of extraction and loading, while the transformation process remains relatively simplistic. However, based on observations in applications, we believe that comprehensive transformation is indispensable for real-time data warehouses. Firstly, there is an incompatibility between data models, which stems from the divergent design paradigms of OLAP and OLTP systems. OLTP systems typically employ Third Normal Form (3NF) [15], whereas OLAP systems often utilize denormalized schemas (e.g., star schema, snowflake schema) [19, 32]. This mismatch necessitates additional transformations to bridge the gap, which existing ZeroETL-based solutions struggle to handle efficiently. Furthermore, query optimization in OLAP workloads heavily relies on pre-aggregation and pre-computed views to accelerate query performance, particularly for large-scale datasets and complex application scenarios. The simplified handling of Transformation (T) steps in which existing ZeroETL-based solutions can lead to degraded query performance. As observed in practical applications, we find that this direct cross-system integration approach often falls short of meeting user requirements.

In response to these challenges and limitations, we propose Streaming View, a high-performance, real-time data processing engine integrated into AnalyticDB [6] on Alibaba Cloud. Streaming View enables defining views with SQL, supporting both result materialization and real-time streaming data processing. Unlike the Batch SQL approach in offline warehouses, our approach uses views to store SQL, allowing continuous transformation of real-time data in a streaming fashion as specified. Compared to separately integrated streaming engines, this built-in streaming capability reduces system complexity, minimizes inter-system data transfer, enhances performance, reduces latency, and simplifies data consistency management. Furthermore, Streaming View is deeply integrated with the core warehouse optimizations, including SIMD instruction sets, cost-based optimizers, and columnar storage.

In fact, apart from the Streaming View, incremental view maintenance (IVM) approaches may also serve as a potential solution, which has been widely discussed in the database community [2, 13, 28]. Systems such as Oracle [12], SQL Server [51], Greenplum [35], Doris [8], Redshift [21], and Snowflake [17] have also integrated support for it. However, in these systems, the design and implementation of IVM tend to focus on lightweight precomputation and query acceleration, particularly in *read-heavy* environments, adopting a read strong consistency model. They often lack optimizations for write throughput and distributed environments, and their limited syntax for incremental updates hinders complex SQL queries, especially in deeply nested cases. Therefore, in the context of contemporary complex data processing and high-throughput application scenarios, utilizing the existing IVM functionalities of these products fails to efficiently support data processing needs.

Another solution is to directly integrate existing stream processing engines within the data warehouse. However, most mainstream stream processing engines and databases are built with different programming languages, creating integration challenges. Additionally, standalone stream processing engines are typically memory-centric, conflicting with the disk-based operations of databases. Thus, it is essential to design a stream processing engine specifically tailored for data warehouse integration, allowing full use of the warehouse architecture and optimizations. In experiments and Alibaba business practice, Streaming View has shown clear advantages over standalone stream processing engines in simplifying real-time analytical architectures, as well as in performance and maintainability.

Unlike traditional incremental views that primarily focus on the implementation of incremental computations, the proposed solutions for streaming views aim to establish a high-throughput, low-latency, and continuously online stream processing engine integrated within the data warehouse. This approach emphasizes support for eventually consistent write models, complex SQL syntax, and adaptive optimization and operational support to sustain the stable execution of stream tasks. The implementation of Streaming View is based on our designed unified distributed stream processing framework. Within Streaming View, we have strategically designed specialized components such as pre-prune, static tuning, and adaptive optimization phases, as well as operational segments. Streaming View eliminates the need for external stream processing engines, reducing system complexity while ensuring data consistency. Additionally, Streaming View is deeply integrated with core warehouse optimizations, facilitating easier achievement of high performance and elasticity. By incorporating native stream data processing into the data warehouse, Streaming View simplifies real-time data architecture, enhances the efficient execution of T operations within the warehouse, and addresses diverse and complex user requirements. Our contributions are as follows:

- We introduce a distributed stream processing framework that comprises customized components to ensure scalability and efficiency, supporting both batch and incremental processing.
- We design algorithms and optimizations tailored for various syntax and high-throughput data processing, ensuring the system meets the needs of high-throughput and complex applications.
- We designed comprehensive monitoring and operational support for the proposed method to ensure stable and efficient large-scale real-time data processing in production-grade systems.
- Extensive experiments show our method boosts processing performance by up to 7x (resp. 20x) compared to traditional ETL (resp. IVM) methods. Moreover, it supports complex scenarios beyond the capabilities of existing approaches. Real-world tests further validate the effectiveness of our solution.

Table 1: Frequently used notations.

Notation	Description
Q	A user defined query.
Q'	A query after rewrite.
$SQ_1, \ldots, SQ_n$	Subqueries or CTEs
$R, R_1, \ldots, R_n$	Base table(s).
V	A view generated by a query Q.
$\Delta V, \Delta R$	The incremental changes of view V and base table R.
Δ_{old}	The deleted or pre-update state records.
Δ_{new}	The inserted or updated records.
R'	The after update state of R, usual R'=R+ΔR.
$SPJG$	Selection, projection, inner join and group by in SQL.
$SPOJ$	Selection, projection and outer join in SQL.
Q_{SPJ}/Q_{SPOJ}	A query of selection, projection and inner/outer join.
$\mathbf{t}$	List of group-by columns.
$Agg_{\mathbf{t}}[\cdot]$	Aggregations (*sum*, *count*, etc) on group-by columns $\mathbf{t}$.

2 BACKGROUND AND MOTIVATION

2.1 Traditional ETL and processing methods

Extract, Transform, Load (ETL) [44] is a fundamental process that facilitates the data from various disparate sources into a unified data warehouse. The ETL process comprises three primary stages:

- *Extract:* Raw data is gathered from multiple sources such as databases, flat files, or APIs.
- *Transform:* The data undergoes cleansing, aggregation, and reformatting to align with application requirements.
- *Load:* Transformed data is deposited into a target system for subsequent analysis and reporting.

Next, we introduce some popular methods related to ETL implementations. The frequently used notations are shown in Table 1.

Data Sync Tools. The ETL process generally involves synchronizing data from multiple systems into the data warehouse. Traditional methods typically rely on tools such as DataX [16] and Apache Flume [22], which are responsible for extracting data from various sources and subsequently loading it into the warehouse.

Streaming Engine. In real-time scenarios, ETL for real-time data tends to favor stream processing engines such as Apache Flink [7, 14] and Spark Streaming [9, 48]. These engines not only facilitate data synchronization but also support the development of complex data transformation tasks using SQL or programming methods.

Batch SQL. In traditional offline data warehouses, data is loaded into the warehouse, after which the ETL process is completed by running SQL in a batch-oriented manner during off-peak hours.

Incremental View Maintenance (IVM). IVM [2, 13, 28] incrementally updates materialized views by propagating changes from base tables, avoiding full recomputation. For a view $V = Q(R_1, R_2, \cdots, R_n)$, IVM computes incremental changes $\Delta V = Q(R_1 + \Delta R_1, \cdots, R_n + \Delta R_n) - Q(R_1, \cdots, R_n)$, enabling efficient updates. However, IVM supports only a limited subset of SQL and struggles with high-write workloads due to strong consistency models, making it unsuitable for complex real-time data processing.

2.2 Architecture of AnalyticDB

Streaming View is designed as a real-time data processing system that is seamlessly integrated within a popular commercial data warehouse AnalyticDB [6]. AnalyticDB is a typical MPP-based [11] data warehouse with an architecture consisting of a *coordinator* and multiple *workers*, as shown in Figure 2. The coordinator is responsible for generating execution plans, managing distributed transactions, and handling query distribution and scheduling.

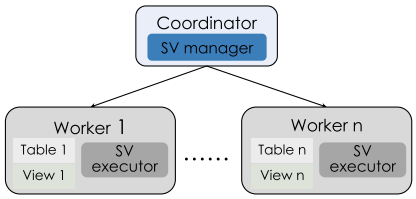

Figure 2: The architecture overview of AnalyticDB.

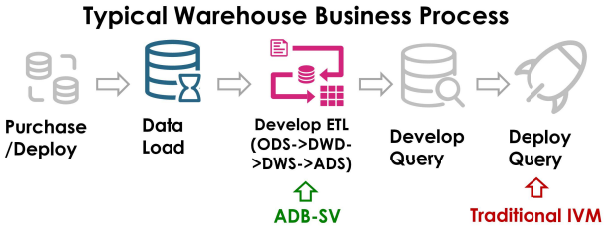

Figure 3: Data warehouse business process.

In most cases, tables in AnalyticDB are distributed across multiple workers using hash partitioning. Similarly, streaming views created by AnalyticDB are also distributed across multiple workers, just like regular tables. As a subsystem within AnalyticDB, Streaming View consists of two main components: (i) *Manager*, which runs on the Coordinator and is responsible for generating incremental execution plans and managing the view maintenance workflow, including adaptive optimization strategies. (ii) *Executor*, which runs on each worker and handles incremental computations, data merging, and other computational tasks.

2.3 Motivation

Traditional industry approaches to IVM primarily focus on view maintenance [12], rewriting [1], and recommendation [3, 10]. The key strategy involves collecting and analyzing the workload of SQL queries to identify common subsets, which are then precomputed, thus accelerating the online query performance. However, this approach has significant limitations in practical business scenarios. As shown in Figure 3, a typical data warehouse construction process involves initial setup, importing base data, developing layered ETL processes, building ad-hoc queries on top of these ETL layers, and finally deploying queries. Traditional methods are disconnected from this standard workflow because they are introduced too late—often only optimizing after queries have already been running in the data warehouse. As a result, IVM is not effective for this process.

To address this, Streaming View is designed to integrate into data warehouse operations at an earlier stage, providing real-time support for ETL activities. While certain ETL processes, such as multi-table JOIN operations and metric pre-computation, share some similarities with traditional IVM by leveraging pre-computation to enhance query performance, Streaming View handles a more comprehensive workload that requires serial management within the core data processing stream. Furthermore, for seamless integration into the ETL phase, view maintenance must achieve high throughput, low latency, comprehensive syntax support, and maintain ongoing online stability for deployment in real-world applications. These requirements extend beyond the scope of traditional IVM incremental computations and align more closely with the

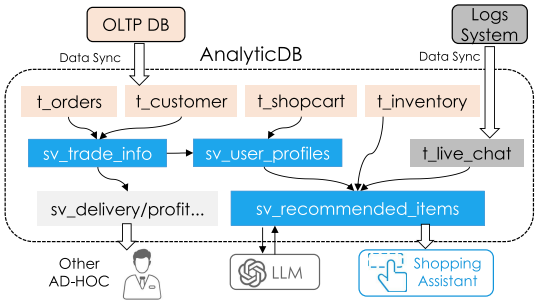

Figure 4: A real-world case.

characteristics of stream processing. Consequently, Streaming View incorporates tailored designs and optimizations that enable it to supplant traditional stream processing engines, facilitating seamless integration with ETL workloads at the early stages of the data warehouse process. This approach significantly enhances data processing performance in practical scenarios, ensuring that Streaming View meets rigorous demands of modern data environments while providing more effective operation compared to traditional IVM.

2.4 Real-world Case

Consider a popular e-commerce live-streaming scenario where large language models (LLMs) are used to automatically recommend corresponding products for real-time chat messages, alleviating the workload of hosts. In this business scenario, real-time data processing technology is leveraged to address issues related to the timeliness and customization limitations within LLM systems [31].

As shown in Figure 4, the data for this system primarily originates from two sources: the OLTP database and the logs system. There are several tables involved, including `t_orders`, `t_customer`, `t_inventory`, `t_shopcart`, and `t_live_chat`. These are processed through joins and group by operations using `sv_trade_info` and `sv_user_profiles`. The real-time processed data is then channeled through `sv_recommended_items` and input into the LLM to generate a list of recommended products. Several key challenges arise during the real-time data processing:

- *High Throughput:* Requires high write throughput, typically reaching hundreds of thousands of entries per second.
- *Complex Joins:* Traditional standalone stream processing engines demand maintaining redundant copies of all relevant tables outside the data warehouse, which consumes significant resources.
- *Data Freshness:* Data processing involves multi-layered nested pipelines. The extended data paths in standalone stream processing engines exacerbate freshness issues.

Streaming View is specifically designed and optimized for high-throughput, low-latency data processing. It is integrated within the data warehouse and conducts incremental computations based on disk storage, thereby avoiding data redundancy. This method effectively satisfies the performance and freshness requirements.

3 SYSTEM DESIGN

3.1 Design Principles

Our goal is to serve as a stream engine replacement for real-time data analytics, supporting complex ETL processing to simplify real-time data architectures. Accordingly, the design of Streaming View follows several key principles that address the limitations of traditional Incremental View Maintenance (IVM) in real-time data processing versus streaming data engines. These principles are essential for driving the necessary paradigm shifts in key dimensions: ultra-high throughput processing, incremental syntax completeness, and automatic tuning with persistent availability guarantees.

Ultra-High throughput processing: Traditional IVM systems typically operate in bypass mode under read-heavy scenarios. In contrast, Streaming View is designed for in-line deployment within the core data pipeline of real-time data processing. This requires handling high throughput while maintaining strict latency constraints, ensuring data processing tasks complete within seconds. To achieve this, Streaming View leverages distributed delta collection and advanced pre-pruning to minimize the data processed incrementally, and uses an eventually consistent write mode to merge small write tasks, enhancing throughput. Additionally, the system dynamically adjusts execution plans based on runtime information, ensuring optimal performance under varying workloads.

Incremental Syntax Completeness: The complexity of view maintenance definitions in data processing scenarios often involves multi-level nested relationships. If the SQL syntax of any layer exceeds the supported incremental syntax table, the integrity of the entire ETL pipeline will be compromised. Streaming View addresses this by providing extensive support for key SQL operations, including complex joins and aggregations. The incremental computation and merge stages of the system are designed to be extensible, allowing for the integration of new incremental algorithms to support additional SQL constructs. Empirical analysis conducted post-deployment validates the necessity of this design; in real-world deployments, over 50% of lineage graphs exhibit depths exceeding five levels and widths surpassing ten nodes. This complexity not only introduces significant performance challenges but also necessitates comprehensive syntactic coverage.

Automatic tuning and availability: In mission-critical data transformation tasks, Streaming View systems must move beyond the traditional IVM "create-tune-exit" optimization paradigm. Streaming View employs pre-optimized incremental update strategies that reduce the need for manual tuning. Furthermore, the system autonomously adapts to workload fluctuations during continuous incremental maintenance, ensuring persistent availability and stability.

To address these demands of real-time data processing, Streaming View is designed with core principles ensuring high throughput, comprehensive syntax support, and continuous online. Natively integrated in AnalyticDB, it leverages optimizations like SIMD, cost-based optimization, columnar storage, and elastic scaling to minimize overhead. With AnalyticDB transaction framework, it maintains eventual consistency for Streaming View updates while preserving base table ACID properties, using a *merge-on-write mechanism* to eliminate runtime merge overhead and ensure consistent read performance. The system provides extensive incremental syntax support, reducing syntax failure risks and enhancing applicability across business scenarios. Automatic optimization during creation and maintenance phases minimizes manual intervention, while continuous read/write operations ensure business continuity during SQL definition changes. These principles enable Streaming View to handle complex real-time data processing efficiently.

3.2 Architecture Overview

Based on the aforementioned goals and principles, we propose a novel streaming view maintenance framework as shown in Figure 5.

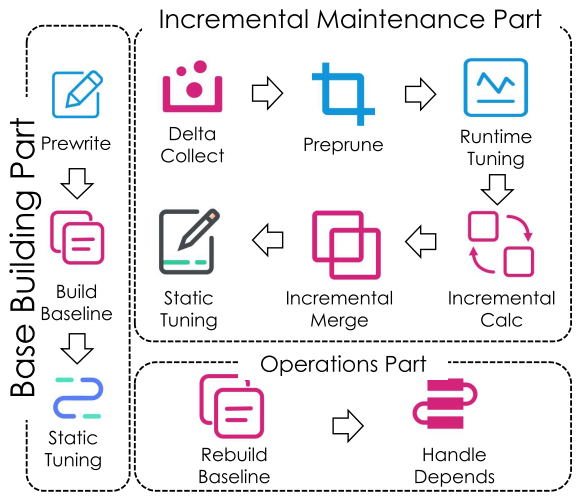

Figure 5: Architecture of Streaming View.

Our framework is structured into three main parts: Base Building part, Incremental Maintenance part, and the Operations part. Unlike traditional methods, which focuses on incremental computation and merging, our framework expands other part and includes additional stages such as Pre-write, Preprune Delta Data, Adaptive Runtime Optimizer, Runtime Information Statistics and Feedback, and the entire Operations part. These enhancements are designed to further strengthen high throughput, low latency, syntactic completeness, and continuous online processing, making Streaming View more suitable for data processing scenarios compared to traditional Incremental View. Additionally, our framework aims to unify common components across the entire view maintenance lifecycle, such as baseline building, distributed delta collection, advanced incremental data pre-pruning, and support for transactional eventual consistency. Furthermore, we design extensible interfaces at appropriate points to facilitate the continuous iteration of a wide range of incremental syntax at a low cost. Despite much research and discussion related to these topics, there remains a lack of a complete framework design capable of fulfilling all the mentioned requirements while supporting complex data processing pipeline tasks with high throughput and low latency. In Section 4, we will provide a detailed discussion of this framework.

Beyond the typical challenges and corresponding design trade-offs, during the large-scale deployment of Streaming View, we observed specific scenarios in real-world business cases. Although these scenarios may seem niche, they account for a significant portion of actual business requirements. To address these, we designed and implemented targeted optimizations that significantly improved the maintenance efficiency of views in these scenarios. We will elaborate on these optimizations in Section 5.

3.3 Design Summary

We concludes the evaluation by summarizing the design trade-offs illustrated in Figure 6. The figure provides an empirical ranking (higher is better) of several schemes across various design attributes. For each attribute, we score them based on their relative performance or our experience in implementing the designs (e.g., syntactic completeness). Streaming View achieves superior performance by leveraging incremental maintenance with built-in database capabilities, outperforming stream engine, traditional IVM, and batch-based

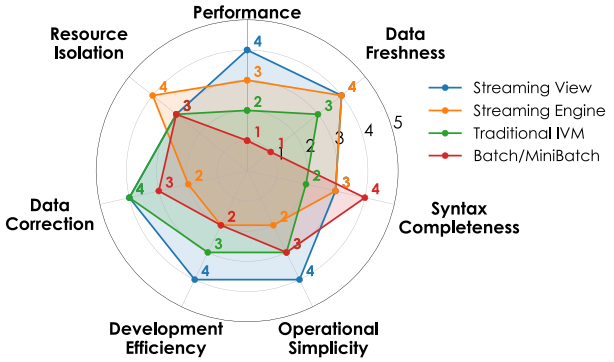

Figure 6: Design summary of the four strategies.

processing methods. It ensures high data freshness through stream-like processing within a built-in system, moreover it maintains ANSI SQL subset completeness, balancing expressiveness and efficiency. Unlike stream engines, which require multisystem coordination, Streaming View integrates built-in optimizations, improving operational efficiency, development simplicity, and data correction. While its isolation level is shared, its holistic design makes it a robust solution.

4 DESIGN AND IMPLEMENTATION

4.1 Base Building Part

For the base building part, it corresponds to the creation process of a materialized view in a typical IVM system. Typically, users need to provide a SQL statement Q to define a specific view. The syntax check is usually performed first to confirm whether Q is in the list of syntaxes that support incremental updates. After passing the check, the SQL is executed in a conventional manner to generate a set of base data. This part mainly includes three key stages: query pre-rewriting, parallel batch processing construction, and static optimization. The details of each stage will be introduced later. The pseudocode is shown in Algorithm 1. First, the query pre-rewriting stage prepares for subsequent operations by completing, splitting, or rewriting the original query (line 1). Second, the batch construction stage adopts a parallel computing strategy, significantly improving the computational efficiency of data construction (line 2). Finally, the static optimization stage generates optimization information such as indexes and data distribution based on the rewritten query and the constructed data (line 3), further enhancing data processing performance. Unlike the design of general IVM systems, Streaming View uniquely designs the first and third stages in this part to improve existing practices.

Algorithm 1: Base Building (Q)

1 $Q' \leftarrow Prewrite(Q)$;
2 $V \leftarrow BaseBuild(Q')$ in parallel;
3 $Index, Distribution, Other \leftarrow StaticOptimize(Q', V)$;
4 **return** V

Pre-write stage. Query pre-rewrite stage is aimed at aligning complex SQL syntax. Standard SQL syntax can be quite intricate, particularly when dealing with incremental updates. For example, the algorithms differ between single table operations and `JOIN` operations, as well as between inner joins and outer joins. However,

the actual situation is more complex and requires addressing various details. Here are two specific examples:

Example 1. *For SQL statements containing a* `GROUP BY` *clause, it is generally required that all* `GROUP BY` *fields are included in the* `SELECT` *columns of result. If a user's SQL does not meet these requirements, our solution will automatically complete result columns for the user.*

Example 2. *In our solution, the scenario of an outer join combined with aggregation does not support non-equi join conditions. However, scenarios with only outer joins without aggregation do support non-equi join conditions. In such cases, during this stage, we rewrite the outer join with aggregation into an outer join followed by a view nested with a single-table aggregation. This approach maximizes compliance with syntax requirements for the customer.*

There are many similar scenarios in which we try to rewrite queries as much as possible during this stage. In practice, we have observed that this capability greatly enhances syntax completeness, significantly reducing issues related to user syntax support.

Base building stage. In this stage, we execute the SQL defined by streaming view to generate a base data. However, this process cannot be completed immediately and may take some time. During the execution of the SQL, the data snapshot is captured at the moment the SQL execution begins. This means that any incremental updates to the base tables occurring during this period are not handled by the batch process and require additional attention.

A straightforward solution to ensure data consistency is to apply a write lock on the base tables during the build stage. However, this approach can significantly disrupt production environments, as large views often need several minutes or even hours to complete their construction. To address this, we developed a more efficient method that allows base tables to remain writable during the build process. At the start of the build, we employ a brief write lock to create a delta table associated with the base table, utilizing a reference counting method. If a delta table already exists due to other streaming view processes on the same base table, it can be reused; otherwise, a new one is created. Once the delta table is established, we commit the transaction and release the write lock. Consequently, the delta table records any incremental writes made to the base table, which are subsequently processed once the build is completed. This approach minimizes disruptions while maintaining data consistency.

Static optimize stage. As previously mentioned, our system is designed to ensure that users can perform maintenance on their defined streaming view with high performance without tuning. To achieve this, we have dedicated this particular stage to optimization. In this stage, a series of optimization rules are generated by comprehensively analyzing the SQL characteristics of view definitions and the statistical information from the base data produced in the previous stage. These rules are applied to enhance the efficiency of the subsequent maintenance stage, with automatic indexing and distribution key selection being particularly critical.

Automatic indexing. Our solution maintains data strictly based on SQL execution results for operations such as JOINs and aggregations, avoiding the use of data windowing methods commonly employed by many stream engines to reduce related data to ensure complete support for incremental maintenance semantics. Given the limited capacity of memory, Streaming View cannot rely on in-memory hash tables for managing JOIN and aggregation states, as is common in window-based stream engines, and instead employs disk-based mechanisms to improve efficiency. Indexing plays a pivotal role in this process. Key indexing strategies include: (i) Join Views: Indexing the join key on base tables significantly improves efficiency during incremental computation. (ii) SPJ Views: Creating a near-unique index on the view enables more efficient deletion during incremental merges. (iii) Group by Views: A unique index on the Group By fields facilitates smoother incremental merge operations. (iv) Min/Max Views: Indexing base table ensures efficient recalculation of min/max values when deletions occur.

Additionally, specific indexing strategies are applied to handle outer joins, window functions, and UNION ALL operations. However, the presence of multiple syntactic features in an view's SQL definition may necessitate multiple indexes, and an streaming view may serve dual roles (e.g., as both an streaming view and a base table for another streaming view). This can lead to index redundancy. To address this, the process is divided into a rule collection stage and a decision-making stage. The rule collection stage generates a candidate index list using extensible rules, while the decision-making stage consolidates these into a final index list.

Automatic Distribution Key. As a typical MPP distributed data warehouse, the choice of distribution keys significantly impacts efficiency of incremental computation and merge operations. For instance, aligning the distribution key of a base table with the join key in JOIN operations eliminates network overhead. Unlike automatic indexing, AnalyticDB only adjusts the distribution keys on streaming views and provides recommendations for base tables, striking a balance between maximizing maintenance efficiency and preserving stability of existing user queries. This trade-off ensures optimal performance while minimizing disruption to the system.

According to data collected from large-scale production environments, with the aforementioned optimizations, more than 99% of streaming views reached a high-performance state that did not require tuning after the build stage, and the incidence of sub-optimal conditions was less than 1%.

4.2 Incremental Maintenance Part

This part process the core of streaming view maintenance operations, where the view is continuously updated incrementally in response to changes in the read and write activities of the base tables. The conventional approach involves iteratively executing delta collection, incremental computation, and incremental merging. However, with the requirements of application scenario for high throughput, completeness, and sustained online stability, we have custom-designed several stages: Pre-prune, AdaptiveOptimize, monitoring and collection, and feedback. The pseudocode for this part is shown as Algorithm 2.

The incremental maintenance process for a set of view V is initiated by first obtaining the new and old deltas (Δ_{new} and Δ_{old}) from the updated base table in parallel (Line 1). For each view's query define $Q[i]$, the deltas are pre-pruned to eliminate invalid data based on the query constraints (Line 3), after the pre-pruning process is completed, the pruned deltas ($\Delta_{new'}$ and $\Delta_{old'}$) are obtained. Subsequently, runtime adaptive optimization rules are generated to tailor the computation to the specific characteristics of the deltas and the query (Line 4). Depending on type of query (e.g., SP, INNER JOIN, etc), the incremental data $\Delta Q[i]$ is calculated in parallel (Line

5), leveraging base table and optimization rules. The computed incremental results are then merged with the previous state $V[i]$ in parallel, producing the updated view $V[i]'$ and the corresponding delta $\Delta V[i]$ (Line 6). If the merged result has dependencies, the process may recursively handle these dependencies, ensuring consistency across the system (Line 7). Finally, the algorithm returns the fully updated views V' completing the incremental maintenance. Next, we introduce the details of involved stages.

Algorithm 2: Inc-Maintenance(Q, V, $BaseTable$)

1 $\Delta_{new}, \Delta_{old} \leftarrow ObtainDelta(BaseTable)$ in parallel;
2 **for** $i \leftarrow 1$ ***to*** n **do**
3 $\quad \Delta_{new'}, \Delta_{old'} \leftarrow PrePrune(\Delta_{new}, \Delta_{old}, Q[i])$;
4 $\quad rules \leftarrow AdaptiveOptimize(\Delta_{new'}, \Delta_{old'}, Q[i])$;
5 $\quad \Delta Q[i] \leftarrow IncCalc(\Delta_{new'}, \Delta_{old'}, Q[i], BaseTable, rules)$ in parallel;
6 $\quad V'[i], \Delta V[i] \leftarrow IncMerge(\Delta Q[i], V)$ in parallel;
7 $\quad$ **if** *HasDepend*($V'[i]$) **then** recursive $\Delta V[i]$;
8 **end**
9 **return** V';

Distributed delta collect stage. In our data warehouse, data is distributed across multiple worker nodes. To support streaming view, we introduce a *distributed delta table* mechanism. Each base table is associated with two delta tables: a *new delta* table for capturing inserted or updated records, and an *old delta* table for capturing deleted or pre-update records. Specifically, an `UPDATE` operation generates two records: one in the *old delta* table representing the pre-update state, and another in the *new delta* table representing the post-update state. Multiple views on the same base table share the same set of delta tables to minimize storage and maintenance overhead. The distribution strategy of delta tables aligns with that of the base tables, ensuring that no data movement is required during the delta collection stage.

Unlike alternative approaches that rely on subscribing to Write-Ahead Logging (WAL) for asynchronous delta collection [12], we avoid log-based methods due to their inherent limitations. Specifically, log retrieval overhead increases significantly under high-concurrency write scenarios, and using separate logs introduces additional random disk I/O operations. Instead, our delta collection mechanism is designed to handle concurrent writes efficiently by merging tuples from multiple transactions during the delta collection stage. This is enabled by the *eventual consistency* of our solution, which does not enforce strict ordering of transaction applications. Instead, the system ensures that views are updated to their final state as soon as possible after base table writes are committed.

Handling Concurrent Updates. A critical scenario involves concurrent updates to the same row by multiple transactions. For example, suppose three transactions update same row from `(1,1)` to `(1,2)`, `(1,3)`, and `(1,4)`; the old and new delta tables record `(1,1)`, `(1,2)`, `(1,3)` and `(1,2)`, `(1,3)`, `(1,4)`, respectively. Under eventual consistency, system merges deltas by retaining the net transition from `(1,1)` to `(1,4)`, discarding intermediate states, with remaining delta records merged during next preprune stage.

Periodic maintenance and throughput optimization. Our system performs periodic and prompt maintenance of delta data by default, followed by cleanup of delta tables. The system automatically merges concurrent writes during the delta collection stage, enabling high throughput even under heavy write workloads. This design is particularly advantageous for real-time data warehousing scenarios, where numerous small write transactions are common. Although it supports strong consistency modes, our production experience shows that the eventual consistency model, combined with prompt maintenance, suffices for most analytical queries. It ensures that view data only moves forward, avoiding phenomena such as phantom reads while sacrificing minimal freshness. This trade-off aligns well with the requirements of real-time analytics workloads.

Table 2: Time cost for different optimizers and delta sizes

Delta Rows	10^2	10^3	10^4	10^5	10^6
Native CBO (ms)	20	35	206	801	7,040
SV Adaptive (ms)	20	35	87	504	5,205

Preprune delta data stage. After generating delta data, we introduce a preprune stage. This stage serves two primary purposes: (1) ensuring data correctness under concurrent updates in an eventual consistency model, and (2) reducing the volume of data propagated to subsequent steps. To achieve the first goal, the following operations are performed: $\Delta'_{new} = \Delta_{new} - \Delta_{old}$, $\Delta'_{old} = \Delta_{old} - \Delta_{new}$. These operations not only guarantee correctness in concurrent update scenarios but also optimize performance for updates, which are common in real-time OLTP-to-OLAP synchronization.

In addition to the basic pruning operations, we implements advanced pruning rules for update scenarios, such as *valid column determination*. Consider an example with two tables: $R(a, b, c)$ and $S(i, j, k)$. A view $V(c, k)$ is defined as $R \bowtie_{p(R.a,S.i)} S$. In this case, only updates to columns $R.c$, $S.k$, $R.a$, and $S.i$ can affect the view, while updates to $R.b$ and $S.j$ have no impact. Thus, the pruning operations are extended as follows: $\Delta'_{new} = \Delta_{new_{(valid)}} - \Delta_{old_{(valid)}}$, $\Delta'_{old} = \Delta_{old_{(valid)}} - \Delta_{new_{(valid)}}$. The preprune stage is particularly beneficial in multi-level nested scenarios, where a significant portion of delta data generated by lower-level views can be pruned, reducing the computational load for cascading updates and improving system throughput. This is especially effective in OLAP workloads involving wide tables, where fact tables are joined with multiple dimension tables. Pruning dimension table data often significantly reduces the volume of data processed in subsequent stages.

Adaptive runtime maintenance optimizer. In Section 4.1, we discussed static optimization strategies, such as pre-building indexes and distribution keys for maintenance processes. However, real-time data processing workloads are often highly dynamic, with frequent changes in data volume and patterns. For instance, temporary data imports or significant changes in base views can lead to large volumes of delta data, complicating the selection of incremental execution plans. To address this, we employ a hybrid adaptive runtime maintenance optimizer combining cost-based and rule-based approaches. It uses cost-based optimization with adjusted operator weights for stability, caches effective incremental plans for reuse, and adapts dynamically based on delta data size thresholds. For large deltas, it shifts to bulk processing or full view regeneration. With this hybrid strategy, we can enhance stability, reduce bad cases, and improve efficiency.

Table 2 shows that Streaming View (SV) Adaptive optimizer delivers similar maintenance time as native cost-based optimizer (CBO) for small delta sizes, but achieves lower latency at moderate and large delta sizes. This indicates that the adaptive strategy is more effective than native CBO in reducing maintenance costs.

Incremental calculation and data merge stage. Generally speaking, the incremental maintenance algorithms discussed in this stage constitutes the core phase of incremental process. Due to space constraints, we provide a concise overview of the incremental update terms for various SQL syntax scenarios supported by Streaming View, leveraging bag algebra [27] and AGCA [29]. The following notations are defined in Table 1.

Standard SPJG Framework. The SPJG framework comprises selection, projection, join, and group-by operations. We outline the incremental update terms for three key scenarios:

– **Selection & Projection on a single base table** R:

$$\Delta Q_{SP}(R) = Q_{SP}(\Delta R)$$

– **Inner Join on base tables** $\{R_j\}$:

$$\Delta Q_{SPJ} = Q_{SPJ}(R_1 \bowtie \cdots \bowtie \Delta R_j \bowtie \cdots \bowtie R_n)$$

– **Inner Join with Aggregation**:

$$\Delta Q_{SPJG} = Agg_t[Q_{SPJ}(R_1 \bowtie \cdots \bowtie \Delta R_j \bowtie \cdots \bowtie R_n)],$$

Recalculation is required for *min*/*max* if deletions affect their values.

Subquery and common table expressions (CTE). We employ subquery pull-up or hierarchical nesting for non-correlated subqueries and CTEs. Assume updates on base table R result in updates on m subqueries and CTEs $\{SQ_1, \ldots, SQ_m\}$, i.e. $\Delta SQ_i(R) = SQ_i(\Delta R)$ for $i = 1, \ldots, m$, the incremental update term of single-layer nesting is:

$$\Delta Q_{SPJ}(SQ_1 \bowtie \cdots \bowtie SQ_n) = \Sigma_{i=1}^{m} Q_{SPJ}\,[SQ_i(\Delta R)]$$

where $\Sigma(\cdot)$ denotes bag union (`UNION ALL`) for all items. Multi-layer nesting is handled recursively.

Time-dependent functions. For time functions like `CURRENT_DATE`, Streaming View introduces an auto-created internal relation R_{time} to handle time-driven updates:

$$\Delta Q_{SPJ}^{time}(R) = Q_{SPJ}^{without\ time}(R \bowtie \Delta R_{time})$$

where ΔR_{time} represents the time-driven delta data, $Q_{SPJ}^{without\ time}$ denotes the query with time function removed from the original query Q_{SPJ}^{time}.

Other SQL Patterns. We also support incremental maintenance for window functions, higher-order aggregations functions (e.g., `string_agg`/`array_agg`), and `UNION ALL`, implemented via recomputation and recursive subclause processing.

Runtime info statistics and feedback optimizer. Despite the extensive optimizations integrated into the framework, bad cases are inevitable in complex real-world scenarios. As previously mentioned, the lineage graph of view can often become highly intricate, making monitoring and diagnostics crucial. For instance, when latency occurs, it is essential to pinpoint exactly which views within a lineage graph are experiencing bad or slow cases.

To address this, we deploy a comprehensive monitoring and diagnostic system that offers detailed runtime statistics for each view, such as maintenance time, data volume, and other key metrics. The system also includes rules for feedback-driven runtime optimization—for example, upon execution errors or prolonged runtimes, it invalidates and regenerates cached incremental execution plans using the cost model. This feedback loop continuously enhances execution efficiency and reliability.

4.3 Operations Part

In this part, we address the challenge of modifying the definition of view maintenance in real-world applications, such as adding fields, metrics, or altering JOIN relationships. Traditional methods often require deleting and rebuilding the entire view pipeline, which becomes particularly cumbersome in systems, where complex nested views are common. To improve this process, we designed the operations part, with pseudocode shown in Algorithm 3. It first retrieves the dependencies *Depend* associated with the current view V (Line 1). Then, it rebuilds the view V' using the new user-defined query Q' in parallel, ensuring efficiency (Line 2). Subsequently, eachA dependency $Depend[i]$ is reconstructed to maintain consistency with the updated V'. Leveraging Streaming View in-warehouse capabilities, the process ensures transactional consistency for both catalog and data changes, avoiding the challenges of maintaining consistency in stream-based architectures. The part returns the updated view V' and updated set of dependencies $Depend'[n]$, providing a seamless transition while preserving data integrity and minimizing user intervention.

Algorithm 3: Operations Part(Q', V)

```
1 Depend[n] ← GetDepend(V);
2 V' ← BaseBuild(Q') in parallel;
3 for i ← 1 to n do
4 |   Depend'[i] ← ReBuild(Depend[i], V');
5 end
6 return V',Depend'[n];
```

To address the challenges of redefining view efficiently, Streaming View introduces several innovative techniques. When a view definition changes, a new base data is created using batch SQL execution and static optimization, similar to the initial build process. Continuous delta collection is utilized to maintain write capabilities on base tables during this transformation.

A critical challenge in traditional methods is the high-level locking required for catalog changes, which can render views inaccessible for extended periods. To mitigate this, Streaming View introduces an online DDL capability, allowing catalog adjustments and view hierarchy refreshes without making upper-layer views inaccessible, thus reducing exclusive lock usage significantly. The alteration process is decomposed into multiple transactions, each managed by an independent worker. This includes creating a new view, populating it with data, correcting catalog metadata, and updating dependent views. This phased approach minimizes operational overhead and downtime, ensuring maximum accessibility and consistency while reducing the need for view dependency management by operators. This framework enhances ease of use and availability, allowing seamless integration of SQL changes in complex, nested view environments.

5 OPTIMIZATIONS FOR STREAMING VIEW

This section explores several key optimizations that enhance the performance of view maintenance in complex query scenarios. These optimizations improve the scalability, efficiency, and usability of Streaming View in real-world OLAP workloads.

5.1 Left Join Algorithm Optimization

For outer join scenarios, it is widely well-known that the incremental maintenance result consists of the following two aspects: data update caused by changing on normal tuples, and data tracking for changing on orphan tuples [20, 30]. The corresponding incremental update term ΔQ_{SPOJ} can be expressed as the following equation,

Table 3: Maintenance time of *N* left joins query (ms).

N	2	5	8	10	12	13	14	15
Traditional Algorithm	18	55	336	1424	7966	15801	38180	89640
Streaming View	18	29	45	89	186	217	241	274

$$\begin{aligned}\Delta Q_{SPOJ} &= \Delta Q^N + \Delta Q^O \\ &= Q_{SPOJ}\left(R_1 \bowtie \cdots \bowtie \Delta R_j \bowtie \cdots \bowtie R_n\right) + \Delta Q^O\end{aligned} \tag{1}$$

where ΔQ^N and ΔQ^O represent incremental update term caused by normal tuples and orphan tuples separately. Note that ΔQ^N follows the same form as the incremental update term of inner join introduced in Section 4.2. For incremental maintenance on view caused by updates on base tables, ΔQ^N and ΔQ^O show mutually inverse behavior: Insertion/deletion) on base table data always triggers insertion/deletion for ΔQ^N and ΔQ^O during incremental maintenance of view. Symbolic distinctions about this behavior will not be demonstrated in Equation.(1) or discussed in details. As described in [30] (where ΔQ^O is denoted as ΔV^I), a calculation methodology is purposed for obtaining ΔQ^O: For a SQL scenario involving N outer joins, ΔQ^O is calculated by constructing a graph consist of M outer join terms, which ranges from N to 2^N depending on the structure of outer joins in this SQL. Subsequently, ΔQ^O is derived by performing computations across different outer join terms for $1 \sim 2^N$ times according to graph configuration.

Although this methodology emphasizes comprehensive coverage of various combinations of outer join scenarios, it still may suffer from exponential explosion issue in both time and space complexity as number of outer joins N increases. Notably, over 90% outer join SQL cases involve only left joins in real-world OLAP applications. In order to optimize such cases with left join only, Streaming View introduces the conception of forward join key graph, which systematically characterizes all join key propagation paths within current left join SQL that may impact orphan tuples. When updates occur in base tables, all related left join paths that are related to orphan tuples can be directly determined easily by this graph. With such optimization, the time and space complexity of incremental update for left join only scenario can be significantly reduced as comparison: For methodology in existing literature [30], the time and space complexity of constructing outer join term graph is $O(N) \sim O(2^N)$, and outer join terms calculation are needed to be performed for at most 2^N times. In contrast, Streaming View achieves a time and space complexity of $O(N) \sim O(N^2)$ for constructing forward join key graph, and only requires to calculate once to derive ΔQ^O.

In Table 3, we evaluate a typical OLAP scenario where a fact table directly `LEFTJOIN`s multiple dimension tables, the incremental maintenance time taken (in milliseconds) of traditional algorithm and Streaming View for updating a small amount of base table data are demonstrated separately. It can be observed that as the number of tables involved in the `LEFTJOIN` increases, the advantage of our algorithm becomes more apparent. For practical OLAP wide table scenarios, it's quite common to have 10 or more `LEFTJOIN`s, Streaming View is particularly well-suited for these scenarios.

5.2 Low Cardinality Update Optimization

In OLAP scenarios, a common real-time data processing paradigm is the wide table model, where a fact table joins with multiple dimension tables to enrich attributes. A typical workload involves a fact table containing billions of records joining with dimension tables ranging from tens of thousands to hundreds of thousands of records. In real-world scenarios with Streaming View, at least half of the workloads follow this pattern. Among these, one of the most challenging cases is the update of a dimension table, as a single row update in the dimension table can propagate to hundreds of thousands or even millions of records in the view.

Table 4: Maintenance time of low cardinality algorithm (ms).

Cardinality Level	High (10000)	Medium (100)	Low (1)
Without optimization	43	309	29980
With optimization	34	32	31

As previously described, Streaming View primarily employs a merge-on-write mechanism. However, for this specific workload, a merge-on-read-like optimization is introduced to efficiently maintain low-cardinality fields in views. The core idea is to manage low-cardinality fields using a dictionary-based approach. Instead of storing repeated values directly, a dictionary is maintained where each unique value is assigned a pointer. Consequently, updating a low-cardinality field only requires modifying the dictionary entry, significantly improving update performance at the cost of minor read overhead due to dictionary lookups.

Updates are not limited to a single layer—low-cardinality fields can propagate through multiple view dependencies. While updating such fields in the first-layer view only requires modifying the dictionary, the standard process still generates a large number of incremental records that propagate upward. To address this, we propose a global dictionary referenced by all view layers. Encoding begins at the base table, and upper-layer views store only pointers to dictionary entries. The maintenance flow is as follows: initially, follow the standard maintenance workflow to ensure all dictionary fields in views use global pointers consistently across all nested views. After periodic maintenance, only the relevant dictionary values in the global dictionary repository need to be updated for the base table's dictionary columns.

Table 4 shows the execution time (in milliseconds) for incremental maintenance of a single `UPDATE` operation over 1 tuple on the dimension table under the circumstance of a fact table with 1 million tuples joining with a dimension table. It can be clearly noticed that the execution performance of our optimization methodology shows no obvious difference among 3 different cardinality levels of the fact table. In comparison to normal algorithm, our optimization method can achieve nearly 10x to 1,000x better execution performance in low and medium cardinality scenarios.

5.3 Prestate Optimization

The incremental join algorithm described in Section 4.2 primarily addresses scenarios where a single table undergoes modifications. However, in real-world database systems, multiple tables often update concurrently. Consider a three-way join scenario:

$$\begin{aligned}\Delta Q_{SPJ} =& Q_{SPJ}(\Delta R_1 \bowtie R_2 \bowtie R_3) + Q_{SPJ}(R_1' \bowtie \Delta R_2 \bowtie R_3) \\ &+ Q_{SPJ}(R_1' \bowtie R_2' \bowtie \Delta R_3)\end{aligned}$$

where R_1', R_2', and R_3' denote the after update states of R_1, R_2, and R_3, respectively. Taking R_1 as an example:

$$R_1 = R_1' + (\Delta_{old} R_1 - \Delta_{new} R_1) - (\Delta_{new} R_1 - \Delta_{old} R_1)$$

Since view maintenance is typically performed after base table updates, retrieving the post-update states R_1', R_2', and R_3' is straightforward. However, obtaining the pre-update states is non-trivial,

particularly for computing $-(\Delta_{new}R_1 - \Delta_{old}R_1)$, which requires a full-table comparison, incurring at least $O(N)$ complexity.

A common approach in previous work is to maintain a snapshot s of R_1 to facilitate retrieval of the pre-update state [52], then $R_1 = snapshot(R_1')$ where $snapshot(R)$ presents accessing table R via pre-update snapshot. However, simply recording a snapshot is insufficient, as deleted tuples must be retained until all dependent computations no longer require R_1. This retention can block the standard garbage collection mechanism of the database, leading to performance degradation, query slowdowns, and data bloat.

In cope with this issue, Streaming View introduces an optimized strategy that preserves snapshots of R without blocking garbage collection. While accessing R via a specific snapshot s, newly inserted records in R are filtered out using snapshot isolation, and deleted records are reconstructed via a `UNION ALL` operation on historical delta logs. Then we obtain R_1 as follow:

$$R_1 = snapshot(R_1') + (\Delta_{old}R_1 - \Delta_{new}R_1)$$

Since `UNION ALL` is computationally lightweight compared to `EXCEPT ALL`, which requires expensive comparisons and scans, this optimization significantly reduces the computational overhead of retrieving pre-update states while ensuring that the garbage collection mechanism remains unaffected.

6 EVALUATIONS

6.1 Evaluation Setup

Main competitors. We compare the performance of Streaming View (abbr. SV), stream processing engines, and traditional database IVMs in real-time data processing tasks. Streaming View is implemented using the standard AnalyticDB service of Alibaba Cloud. Following the previous work [28], we first anonymized all the commercial products involved due to their license agreements. Since dedicated stream processing engines typically support only incremental computation, we integrate a cloud-hosted version of a popular stream processing engine with the standard AnalyticDB service to facilitate incremental computations and result updates for a fair evaluation. We refer to this integrated system as SPY. For IVM solutions in traditional databases, we tested two widely used commercial databases, DBX and DBY. DBX offers both synchronous and asynchronous IVM modes, while DBY supports only synchronous mode. In this study, we use DBX in asynchronous mode and DBY in synchronous mode as our test benchmarks.

Workloads. Since the common workloads for real-time ETL, such as NEXMark [43], are mostly single-table scenarios, it is difficult to reflect the effect of real scenarios. We chose the standard TPC-H [42] test set as the base data, and then built workloads by constructing corresponding views or view groups for Q1-Q22 queries and generating data processing tasks. Initially, we conducted performance evaluations on a TPC-H (Scale Factor = 10) dataset across all tasks, measuring throughput under identical configuration settings and assessing the completeness of syntax support across different systems. Furthermore, we test the resource consumption at varying update load levels to evaluate system performance under different loads. Beyond these tests, we adjusted the scale factor of TPC-H to assess the scalability of all tested systems.

System configurations. All tested methods are uniformly configured with 8 CPUs and 32GB of memory or equivalent resources. Specifically, our solution utilizes Alibaba Cloud standard AnalyticDB service, configured with four nodes, each with two Intel Xeon 2.90GHz virtual cores, 8GB DDR4 DRAM, and PL1-level cloud storage. To accommodate SPY additional storage needs, it is allocated 6 CPUs and 24GB DDR4 DRAM, bringing the total with AnalyticDB to 8 Intel Xeon 2.90GHz CPUs, 32GB DDR4 DRAM, and same cloud storage, matching our solution. DBX runs on a cloud ECS server, and DBY uses a cloud-hosted version, both with the same configuration as our solution. Furthermore, AnalyticDB, SPY, DBX, and DBY use their default configurations with minor optimizations like SQL rewriting and pre-built indexing. Detailed descriptions of optimizations are provided in the subsequent experimental sections.

During the experiment, we first import the appropriate size TPCH standard dataset into each system. For each test scenario, we then construct the corresponding views or stream processing tasks. Mirroring real-world scenarios, the data update process involves base table updates and the necessary updates to the views or processing tasks' result tables within the data warehouse. This approach ensures a realistic and thorough evaluation of each system's capabilities in handling real-time data processing tasks under consistent and comparable conditions.

6.2 Evaluation Results

Throughput performance. In this section, we conduct the test for throughput of all methods. We use 10GB TPC-H to evaluate the performance of various methods, for each of which we concurrently apply enough update loads to test the highest throughput performance that each system can achieve. Figure 7 shows the throughput performance of different methods in 22 data processing tasks. If a method does not support a data processing task, the corresponding throughput is recorded as 0 and is not drawn in the figure. In particular, when performing data processing tasks, the throughput of Streaming View is 20 times higher than that of traditional IVM systems. Compared with streaming engine, Streaming View is always superior to SPY engine in various scenarios, usually showing at least twice the advantage. Especially for queries like Q18 and Q22, the Streaming View is 7 times better than SPY. By simplifying the processing path and improving the execution efficiency of the internal data warehouse, the proposed method improves the throughput performance as shown in the experiment results.

For syntax completeness, Streaming View supports all the required syntax. As mentioned earlier, DBX and DBY lack support for certain syntax elements, which show limitations on data processing tasks. In addition, Streaming View performs all queries with native SQL and configuration without tuning. For other methods of evaluation, we optimize according to the respective product guidelines to achieve the best performance. SPY requires manual connection reordering for multi-table join scenarios to prevent inconsistent connection status and excessive intermediate data, which can particularly affect certain data processing tasks, such as Q2, Q8, and Q9. Due to the lack of indexes, DBX and DBY face slower incremental updates, requiring manual index enhancement before testing is complete. To sum up, we can find that the proposed method is a favorable choice among various data processing tasks.

System usage performance. In this experiments, we test representative data processing tasks include Q1 (1-table with GROUP BY), Q2 (JOIN without GROUP BY), Q8 (8-table JOIN with GROUP BY), Q10 (4-table JOIN with GROUP BY), Q12 (2-table JOIN with GROUP BY), and Q18 (nested scenario), for further usage performance evaluation. We tested various methods under different update speeds (10K/s, 20K/s, 40K/s, K=1000) to assess system usage.

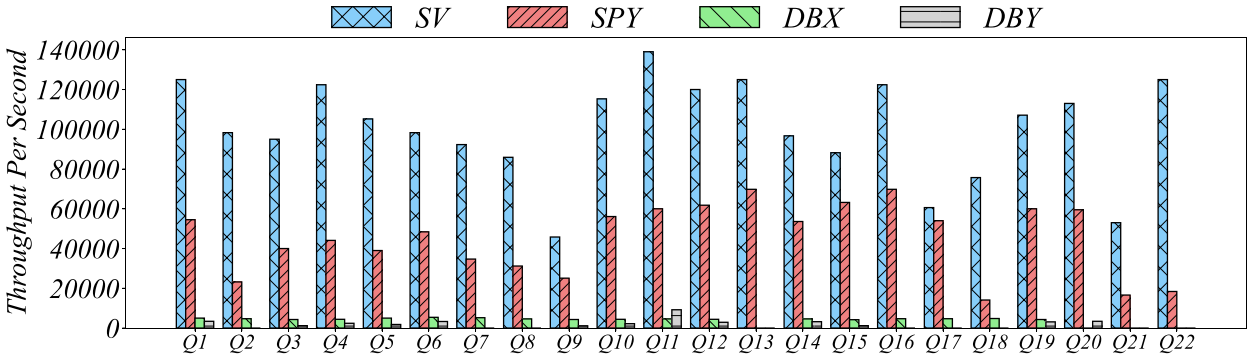

Figure 7: Throughput performance on 10GB TPC-H.

CPU usage and data delay. Figure 8 shows the CPU usage of all methods under different workloads. Given that each experimental setup is equipped with 8 CPU cores and some are distributed systems, we normalize the CPU usage to a maximum of 100%. The corresponding data processing delay is presented in Figure 9, where the delay is computed as the maximum data delay across all data processing tasks. The experimental results show that Streaming View can handle the highest workloads of all data processing tasks. Consequently, no processing delay was observed in tests conducted at three update speeds, with CPU consumption increasing linearly with load. This suggests that SV resource consumption is primarily driven by update load and the volume of data requiring stream processing, benefiting from its Merge On update architecture.

In contrast, SPY can handle update operations exceeding 40K/s for Q1, Q10, and Q12 without delay. However, in scenarios Q2, Q8, and Q18, the maximum throughput failed to reach 40K/s, resulting in delays at 40K/s. The delays in Q2 and Q8 are attributed to near-full CPU resource utilization, while for Q18, the delay may stem from the SQL complexity potentially impeding optimal resource use. Overall, SPY exhibits limitations regarding CPU consumption bottlenecks. DBX and DBY demonstrated continual delays during testing, as their maximum throughput capacity was exceeded at the initial update rate of 10K/s.

Memory Usage. Figure 10 illustrates memory consumption across different methods. SV, DBX, and DBY operate in a disk-based database mode, resulting in relatively lower memory utilization. SV requires aggregation of node usage due to its distributed architecture, its overall memory footprint is slightly higher compared to DBX and DBY. As an in-memory stream processing engine, SPY loads data into memory during processing, leading to a higher memory consumption. This experiment shows that memory consumption of SPY can be up to six times higher than memory consumption of SV. We can find that even though SV uses disk mode, benefiting from the multi-level caching mechanisms and execution efficiency optimizations built into the database, it still has lower CPU usage and higher throughput than SPY, that is, faster processing speed. Overall, SV showed an excellent trade-off in various aspects.

Scalability test. In this set of tests, we scale the TPC-H data set to observe the maximum throughput performance of each method under data sizes of 10G, 100G, and 1T. The experimental results are shown in Figure 11. For methods that take longer than 1 day to load or do not support a data processing task, the corresponding throughput is marked as 0. We find that SV, DBX, and DBY all work

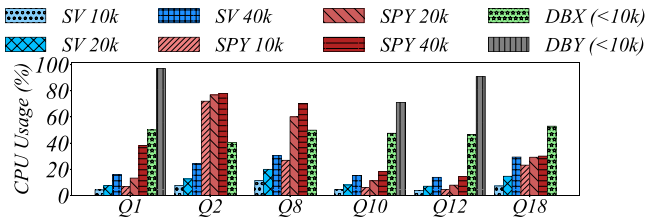

Figure 8: CPU usage for all methods under various workloads.

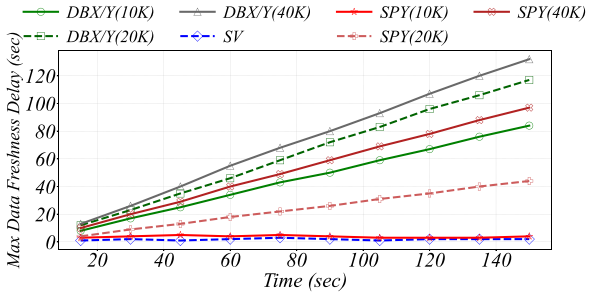

Figure 9: Maximum data freshness delay for all methods with varying the size of workloads.

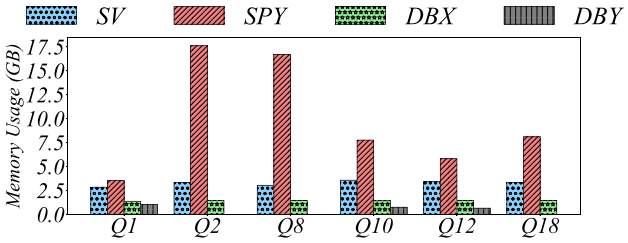

Figure 10: Memory usage for all methods under different workloads.

on data sets of all sizes, but DBX and DBY are unsupported for some tasks. SV maintains an order of magnitude advantage over DBX/DBY. For the SPY method, because it operates in-memory, the load time exceeds one day for 1T data sets. For certain tasks, such as Q10 and Q18, a noticeable performance degradation appears as data size increases, becoming evident with the 100G dataset. This is due to SPY reaching a memory bottleneck as volume grows. Additionally, for the 100GB data set, we observed that SPY performed well on specific queries. We surmise this is because, although the

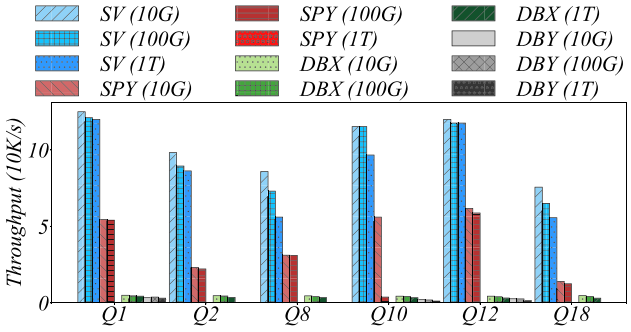

Figure 11: Throughput with varying the size of workloads.

total data set size exceeds SPY's memory limit, some SQL queries only access a subset of tables. After column pruning, actual memory consumption may not reach SPY's configuration limit. In summary, SV proves superior to other methods. Therefore, Streaming View is an excellent choice for a variety of data processing tasks.

Performance of production deployment. In the final experiment, we evaluate Streaming View in a real-world e-commerce scenario. In this application, the system builds dozens of nested views, view defines including wide tables formed by joining fact tables with 10+ dimension tables, complex scenarios with hundreds of sharded sub-tables combined via `UNION ALL`, and pre-aggregations or pre-calculated metrics on these wide tables. In practice, the width of these tables often exceeds 100 columns, with intricate business logic nesting, and the data volume typically ranges from tens to hundreds of terabytes. As a result, the AnalyticDB cluster consists of dozens or even hundreds of nodes. We focus on the metrics related to Streaming View maintenance activities of this system.

In Figure 12, we observe that during a typical e-commerce flash sale event, the real-time data transactions synchronized to the data warehouse surges from 10^4/s to $2 * 10^5$/s due to a spike in transactions, the OLAP queries also increased from 10^3 to 10^4. During this period, Streaming View effectively leverages the resources of dozens of nodes. As the transaction volume increases, CPU consumption rises proportionally, while memory usage remains stable due to the disk-based model of Streaming View. The fluctuations in memory usage are primarily caused by increased write concurrency. Additionally, the eventual consistency maintenance model ensures stable performance during peak times. The base table writes remain unaffected, and the data delay increases slightly. After the peak subsides, the data delay quickly decreases, returning to normal levels. Throughout this process, Streaming View achieves real-time updates for all views driven by updates to any table within the warehouse, with low latency and comprehensive updates.

In contrast, traditional stream processing engines struggle with complex SQL scenarios involving multi-level nesting, resulting in significantly higher resource consumption and latency compared to Streaming View. Moreover, they are prone to single-point bottlenecks during peak times. Additionally, standalone engines outside the data warehouse require redundant data in memory to fully support operations like joins, which is costly and poses significant stability challenges for datasets spanning tens of terabytes. Traditional IVM also falls short in terms of throughput, latency control, system scalability, and comprehensive syntax support, making it rarely suitable for large-scale, high-throughput, low-latency data processing scenarios. From this analysis, we can conclude that

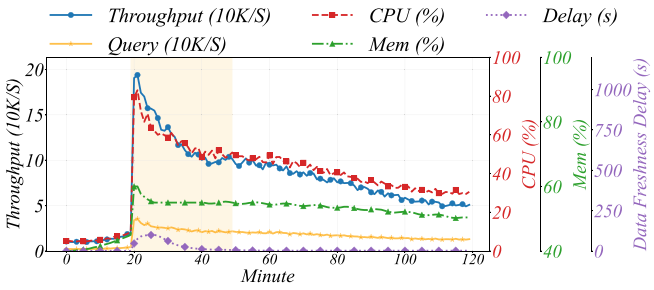

Figure 12: Performance in the real-world workload.

Streaming View is a favorable choice for handling real-time data processing tasks in practical scenarios.

7 RELATED WORK

Over the past decades, IVM-related technologies have garnered significant attention. Systems such as DBToaster[2, 28] and F-IVM[26, 38, 39] have implemented certain IVM frameworks. However, most existing prototype solutions concentrate mainly on the algorithmic aspects of incremental updates, lacking essential features required for a robust IVM system in industrial applications, such as auto-tuning, diagnostics, and online operational support mechanisms. In commercial products, Oracle[3, 12], Microsoft SQL Server[1, 51], Redshift[10], and Snowflake[4] have conducted extensive research, primarily focusing on leveraging IVM for query optimization. However, these implementations largely target light-write scenarios, and their support for incremental syntax remains limited.

With the growing urgency for real-time data processing, standalone stream processing engines like Apache Flink [7, 14] and Apache Spark [9, 48] have rapidly evolved. These engines have gained popularity due to their superior streaming performance and relatively comprehensive syntax support. However, they present challenges in terms of system integration complexity. Recently, products such as Snowflake Dynamic Tables [24] and Databricks Delta Live Tables [23] are also enhancing their capabilities in real-time data processing. Streaming View places a strong emphasis on optimizing latency, throughput, and syntactic completeness, thereby enabling it to handle larger data volumes and more complex business scenarios. Furthermore, independent streaming databases such as RisingWave [40], Materialize [36] provide real-time data processing solutions. However, they often lack the sophisticated query optimizations found in established data warehouse systems, leading to suboptimal query performance.

8 CONCLUSION

This paper introduces a modern streaming data processing system integrated into AnalyticDB of Alibaba Cloud, which not only supports query optimization in the traditional sense but also incorporates a series of enhancements to meet the demands of large-scale real-time data processing. In real-time analytics scenarios, the system demonstrates significant architectural and efficiency advantages compared to traditional approaches relying on external stream processing engines or in-warehouse batch processing. Furthermore, numerous production cases within Alibaba's internal ecosystem and on Alibaba Cloud demonstrate that this system is not an experimental prototype but a mature, stable commercial product readily available to any user on the cloud.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[2] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1566–1569.

[3] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated generation of materialized views in oracle. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3046–3058.

[4] Tyler Akidau, Fabian Hueske, Konstantinos Kloudas, Leon Papke, Niklas Semmler, and Jan Sommerfeld. 2024. Continuous Data Ingestion and Transformation in Snowflake. In *Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems*. 195–198.

[5] Amazon Web Services 2022. *AWS announces Amazon Aurora zero-ETL intergration with Amazon Redshift.* Retrieved March 16, 2025 from https://aws.amazon.com/cn/about-aws/whats-new/2022/11/amazon-aurora-zero-etl-integration-redshift/

[6] AnalyticDB for PostgreSQL 2025. *AnalyticDB for PostgreSQL: Online MPP Data Warehousing Service - Vector Database - Alibaba Cloud.* Retrieved Mar 16, 2025 from https://www.alibabacloud.com/en/product/hybriddb-postgresql

[7] Apache Flink 2025. *Apache Flink® — Stateful Computations over Data Streams.* Retrieved Mar 16, 2025 from https://flink.apache.org/

[8] Apache Spark 2025. *Apache Doris: Open source data warehouse for real time data analytics - Apache Doris.* Retrieved Mar 16, 2025 from https://doris.apache.org/

[9] Apache Spark 2025. *Apache Spark™ - Unified Engine for large-scale data analytics.* Retrieved Mar 16, 2025 from https://spark.apache.org/

[10] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[11] Batcher. 1980. Design of a massively parallel processor. *IEEE Trans. Comput.* 100, 9 (1980), 836–840.

[12] Randall G Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William D Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. 1998. Materialized views in Oracle. In *VLDB*, Vol. 98. 24–27.

[13] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[15] Edgar F Codd. 1972. Further normalization of the data base relational model. *Data base systems* 6, 1972 (1972), 33–64.

[16] Giuseppe Coviello, Kunal Rao, Murugan Sankaradas, and Srimat Chakradhar. 2021. DataX: A system for data exchange and transformation of streams. In *International Symposium on Intelligent and Distributed Computing*. Springer, 319–329.

[17] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[18] Chaoyue Dai, Feng Qian, Wei Jiang, Zhoutian Wang, and Zenghong Wu. 2014. A personalized recommendation system for netease dating site. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1760–1765.

[19] Chiara Forresi, Enrico Gallinucci, Matteo Golfarelli, and Hamdi Ben Hamadou. 2021. A dataspace-based framework for OLAP analyses in a high-variety multistore. *The VLDB Journal* 30, 6 (2021), 1017–1040.

[20] Timothy Griffin and Bharat Kumar. 1998. Algebraic change propagation for semijoin and outerjoin queries. *ACM SIGMOD Record* 27, 3 (1998), 22–27.

[21] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[22] Steve Hoffman. 2013. *Apache Flume: distributed log collection for Hadoop.* Packt Publishing Ltd.

[23] Databricks Inc. 2025. *Delta Live Tables.* Databricks Product Page. Retrieved June 2, 2025 from https://www.databricks.com/product/data-engineering/dlt

[24] Snowflake Inc. 2025. *Dynamic Tables.* Retrieved June 2, 2025 from https://docs.snowflake.com/en/user-guide/dynamic-tables-about

[25] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *VLDB*. ResearchGate GmbH, 16–25.

[26] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2024. F-IVM: analytics over relational databases under updates. *The VLDB Journal* 33, 4 (2024), 903–929.

[27] Anthony Klug. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM (JACM)* 29, 3 (1982), 699–717.

[28] 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.

[29] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 87–98.

[30] Per-Ake Larson and Jingren Zhou. 2006. Efficient maintenance of materialized outer-join views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 56–65.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[32] Yinan Li and Jignesh M Patel. 2014. Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment* 7, 10 (2014), 907–918.

[33] Feifei Li. 2023. Modernization of databases in the cloud era: Building databases that run like Legos. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4140–4151.

[34] Jimmy Lin. 2017. The lambda and the kappa. *IEEE Internet Computing* 21, 05 (2017), 60–66.

[35] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[36] Materialize. 2025. *Materialize - The Streaming SQL Database.* Materialize. Retrieved June 2, 2025 from https://materialize.com/

[37] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *VLDB DISPA Workshop 2020*.

[38] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380.

[39] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: learning over fast-evolving relational data. In *SIGMOD*. 2773–2776.

[40] RisingWave. 2025. *RisingWave: The Cloud-Native Streaming Database.* RisingWave. Retrieved June 2, 2025 from https://www.risingwave.com/

[41] Nicoleta Tantalaki, Stavros Souravlas, and Manos Roumeliotis. 2020. A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems* 35, 5 (2020), 571–601.

[42] The Transaction Processing Council 2025. *TPC-H Homepage.* Retrieved Mar 16, 2025 from https://www.tpc.org/tpch/

[43] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. Nexmark–a benchmark for queries over data streams (draft). *Technical report* (2008).

[44] Panos Vassiliadis. 2009. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)* 5, 3 (2009), 1–27.

[45] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. 2020. Apache IoTDB: Time-series database for internet of things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904.

[46] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, et al. 2020. Tempura: a general cost-based optimizer framework for incremental data processing. *Proceedings of the VLDB Endowment* 14, 1 (2020), 14–27.

[47] James Warren and Nathan Marz. 2015. *Big Data: Principles and best practices of scalable realtime data systems.*

[48] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[49] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[50] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A new benchmark for HTAP databases. *Proceedings of the VLDB Endowment* 17, 5 (2024), 939–951.

[51] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, and Luping Ding. 2006. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 526–535.

[52] Jingren Zhou, Per-Ake Larson, and Hicham G Elmongui. 2007. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*. 231–242.