PreView: Expediting Query Performance via Predictive View Materialization

Anastasiia Tkachenko
University of Utah
Salt Lake City, Utah, USA
anastasiia.tkachenko@utah.edu

Giang Ta
University of Utah
Salt Lake City, Utah, USA
u1589956@utah.edu

Shamit Fatin
University of Utah
Salt Lake City, Utah, USA
shamit.f@utah.com

Anna Fariha
University of Utah
Salt Lake City, Utah, USA
afariha@cs.utah.edu

ABSTRACT

Analytical queries within a session often share expensive intermediate computations (e.g., joins over large tables and filters). Materializing these as reusable views and rewriting a new query in terms of these materialized views can significantly improve query performance by avoiding redundant computation. However, a key challenge here is deciding *which* views to materialize under resource constraints such as limited storage and materialization time. This yields an optimization problem: given a query workload, identify reusable substructures to materialize as views that maximize the expected performance within a storage budget. Traditional approaches assume a static workload and perform view selection and materialization during offline database tuning, failing to adapt to user's dynamic analytical query sessions. In practice, queries arrive sequentially with temporal gaps in between. These gaps can be exploited to materialize views on the fly, requiring view selection strategies that adapt to recent query patterns. Crucially, such strategies must also account for materialization latency to ensure views are fully materialized before the user issues their next query.

We present **PreView**, a personalized system for expediting query performance during a user's analytical query session through **Pre**dictive **View** materialization. PreView observes a user's recent query history to predict the next queries, along with their likelihood and time to appear. Guided by this prediction, it identifies and materializes views that maximize the expected future query speedup subject to storage and materialization-time constraints. PreView further supports cross-user view reuse by leveraging relevant views already materialized by others. We will demonstrate how PreView significantly reduces query latency in a real-world dataset over a user's analytical query session.

1 INTRODUCTION

During an analytical query session, queries often exhibit predictable patterns [6] and share substantial computation that overlap with previously executed queries. Materializing these shared substructures as views—such as joins over large tables or selective filters—can significantly expedite subsequent queries by avoiding expensive redundant computation. However, materializing all potentially useful views is impractical. Real systems operate under limited storage and computational budgets, and an excessive number of materialized views introduces additional overheads, including view maintenance costs and increased complexity for the query optimizer when selecting among candidate views. Consequently, the key challenge is to determine *which* views to materialize so as to maximize their overall benefit to query performance subject to resource constraints.

Example 1. *Consider Kruti, an analyst exploring the IMDb database [5] through interactive analytical SQL queries. In a typical session, she may start with actor-centric exploration, such as finding Action movies featuring* `ACTOR` *after* `DATE`*; then move to genre-level analysis, such as identifying the top 10* `GENRES` *since* `DATE`*; and later examine director- and production-oriented questions, such as retrieving movies directed by* `DIRECTOR` *within a given time range. Although the concrete predicates and parameter values may change from one session to another, Kruti's sessions often follow similar exploratory patterns and repeatedly involve the same families of queries over the same relational structure. Consequently, her analytical queries contain recurring sub-queries both within individual sessions and across sessions, such as repeated expensive joins.*

An opportunity for computation reuse arises with the join among `actors`, `cast`, `movies`, *and* `genres`. *For instance, her first query is:*

```
SELECT m.title FROM actors a JOIN cast c
JOIN movies m JOIN movie_genres mg JOIN gen
res g WHERE a.name='ACTOR' AND g.name='Action'
AND m.year > 'DATE';
```

After observing the result of the above query for 5 minutes, she issues another query, which repeats the same join structure:

```
SELECT g.name, COUNT(*) FROM actors a JOIN
cast c JOIN movies m JOIN movie_genres
mg JOIN genres g WHERE a.name='ACTOR' AND
m.year > 'DATE';
```

Instead of recomputing this expensive multi-way join for each query, we can materialize the shared result:

```
CREATE MATERIALIZED VIEW actor_movie_genre
AS SELECT a.name, m.title, m.year, g.name
FROM actors a JOIN cast c JOIN movies m
JOIN movie_genres mg JOIN genres g;
```

The above materialization completes in under 3 minutes. Subsequent queries arriving after this point can be rewritten to operate directly on the materialized view `actor_movie_genre`*, allowing the optimizer to avoid repeatedly executing the expensive join and thereby significantly reducing query latency.*

A simple approach to improve computation reuse is to aggressively materialize all intermediate results that might benefit future queries. However, this is infeasible due to the overhead of storing and maintaining an excessive number of views and overwhelming the query optimizer with too many options. A better strategy is to materialize a small set of high-utility views that are likely to benefit future queries. Common approaches to estimate the utility of candidate views is to use query frequency, materialization cost, expected workload benefit, and normalized measures like benefit-to-cost ratio [2]. However, these approaches struggle to adapt to dynamic environments, where user query sessions evolve, making the materialized views stale.

Recent works on the materialized view selection problem employ more sophisticated selection criteria to enable automatic and adaptive view generation [6, 10]. However, the candidate views are drawn from subqueries in the historical workload, without any prediction of future queries. Their optimization objective is "retrospective": maximize the total speedup on recent queries using a selected set of views V while accounting for the cost of materializing V. The rationale is that future queries will follow the same distribution as past workloads. However, this assumption is misaligned with practical requirements: (1) future queries may follow a different distribution than past queries, and (2) for a query to benefit from a view, that view must already be materialized before the query is issued, enabling immediate rewriting at submission time. Existing approaches ignore these constraints. As a result, even if the selected views are optimal, ignoring materialization time relative to the estimated arrival of the next query undermines the purpose of materialization.

PreView. In this paper, we present PreView, a personalized system for expediting query performance during a user's analytical query session through Predictive View materialization. Given a user's recent query history, PreView applies a predictive model to estimate a distribution over possible next queries, along with their likelihood and estimated time of arrival. Unlike prior works [6, 10], PreView uses predicted *future* queries to generate the candidate space of possible views. PreView then selects a small set of views that maximizes the expected speedup of future queries subject to storage constraints, while accounting for the time to materialize the views. To ensure views are materialized *before* future queries arrive, PreView considers the earliest predicted arrival time for each possible future query and computes the expected next-query-arrival time from these conservative estimates. PreView also supports view sharing across multiple users sharing the same database. This is particularly important in emerging analytics-as-a-service concept, where multiple users issue related analytical jobs over shared data, with overlapping intermediate computations [7]. To reuse cross-user computations, PreView exploits relevant views materialized by other users instead of re-materializing them.

While view materialization is automatic, PreView can operate in a recommendation mode in which the user chooses which suggested views to materialize. Unlike approaches that target a single global workload [10], PreView enables personalized view materialization.

Related Work. Materialized-view selection has long been studied in database management, aiming to choose views for a given workload under storage and maintenance constraints [1]. More recent systems make this process dynamic, using machine learning or online controllers to adapt the view set as workloads change [3, 4, 9], while self-driving database research has shown that workload forecasting can enable proactive tuning [8]. However, these methods largely assume a fixed or slowly changing workload; dynamic view materialization systems are mostly reactive, and workload forecasting does not directly determine which views to materialize before the next query arrives. No prior system unifies query prediction, candidate-view generation, and optimal view selection under storage constraints, while ensuring views are materialized in time for interactive analytical query sessions. PreView addresses these gaps by predicting future queries and materializing only views that are both useful and can be completed before the next query arrives.

Demonstration. We demonstrate PreView on the IMDB dataset [5], following Kruti's interactive analytical session. Participants will explore how PreView automatically predicts future queries, materializes views under storage and build-time constraints, and rewrites incoming queries on the fly. We show that PostgreSQL combined with PreView achieves a median speedup of 10.83× over a baseline system without view materialization on a realistic analytical query session, with some queries accelerated by up to 30,000×.

We provide an overview of PreView in Section 2 and a detailed walkthrough of our demonstration scenario in Section 3

2 SYSTEM OVERVIEW

Our goal is to reduce the latency of a user's future queries by observing their past query workload and proactively deciding which views should be materialized before the next query arrives. Below, we discuss how we predict the future queries, how we generate candidate views whose materialization may benefit the predicted queries, and how we select an optimal subset of views to materialize.

Query Prediction. Given a user's recent query workload as a time-ordered stream of queries $\mathcal{W} = \{(q_i, \tau_i)\}_{i=1}^{n}$, where q_i is the i-th query and $\tau_i \in \mathbb{R}^+$ is its submission time, the first task is to predict which queries the user is likely to issue next. We capture the notion of a future query at two levels of abstractions. For some workloads, it might be preferable to predict the *template* of the next query, that is, its structural pattern such as joins, group-bys, or aggregations. For other workloads, it is more informative to predict the *parameters* of the next query, such as a date range, genre, or user-specified entity. Therefore, each predicted future query is represented at one or both of the following abstraction levels:

1. $\mathcal{T}$: query templates (structural patterns, e.g., joins/group-bys),
2. $\mathcal{P}$: query parameters (e.g., ranges such as year intervals).

We denote the predicted query abstraction by z, where $z \in \mathcal{Z}$=$\mathcal{T} \cup \mathcal{P}$. Using the workload history up to time τ_n, the predictive model outputs a distribution over likely future queries:

$$Q = \left\{(z_j, p_j, \hat{\tau}_j)\right\}_{j=1}^{m},$$

where $z_j \in \mathcal{Z}$ is a future query representation, p_j is the probability of the query to arrive, and $\hat{\tau}_j$ is its estimated arrival time.

We first canonicalize queries by replacing all literal constants with placeholders, so that structurally identical queries with different parameters map to the same template. We then embed the canonical forms and cluster them to identify recurring query patterns. Each cluster's probability p_j is a linear combination of its relative frequency and a recency score based on exponential temporal decay, giving more weight to patterns that appeared more recently in the workload. We predict the arrival time $\hat{\tau}_j$ from per-cluster Poisson arrival rates, estimated from the query workload $\mathcal{W}$.

Candidate View Generation. The next step is to generate candidate views that are likely to expedite the predicted queries. Specifically, if a future query can be rewritten in terms of a materialized view, then the DBMS can reuse precomputed results instead of recomputing the corresponding joins or aggregations from scratch, substantially reducing query latency. Given a distribution over future queries Q, a view-generation model constructs a candidate set $\mathcal{V} = \{v_1, \ldots, v_r\}$, where each $v_i \in \mathcal{V}$ is a candidate view that can benefit one or more predicted queries in Q.

We generate candidate views with the assistance of OpenAI GPT-4o language model: given $\mathcal{Q}$, we prompt the LLM with each cluster's structural summary—including its tables, join conditions, aggregations, and representative example queries—and request SQL definitions for up to 8 candidate views, each annotated with the clusters it is expected to benefit. While we use an LLM for this step, the approach is not specific to PreView; any method for generating candidate views from a query workload could be used instead.

Selecting a set of Views to Materialize. Once the candidate views are generated, the final task is to choose which of them should actually be materialized. This is the Materialized View Selection (MVS) problem, where the objective is to select the subset of views that maximizes the benefit, in terms of the expected gain of runtimes over the predicted future queries.

View Benefit Model. Let $\text{Latency}(z \mid V)$ denote the execution latency of query abstraction z rewritten in terms of the set of materialized views $V \subseteq \mathcal{V}$. Then the benefit, in terms of runtime gain, obtained from using V is defined as the latency reduction relative to the case where no views are materialized:

$$\text{Gain}(z \mid V) = \text{Latency}(z \mid \varnothing) - \text{Latency}(z \mid V).$$

Since z is an abstraction rather than a concrete query, evaluating $\text{Latency}(z \mid V)$ exactly is infeasible. We therefore use a heuristic that measures the structural overlap between z and views in V to approximate the expected runtime reduction. Given a view v, we compute $\text{overlap}(v, z) \cdot \text{Latency}(z)$ to estimate $\text{Latency}(z \mid v)$, where $\text{overlap}(v, z) \in [0, 1]$ is the fraction of z's tables, join conditions, and aggregates covered by the view v, and $\text{Latency}(z)$ is the average latency of three representative instantiations of z measured via PostgreSQL's `EXPLAIN ANALYZE`. We then compute $\text{Latency}(z \mid V) = \min_{v \in V} \text{Latency}(z \mid v)$

Storage and Time Constraints. Two key constraints while selecting which views to materialize are: (1) ensuring that too many views are not being materialized, respecting a *storage budget* S and (2) the time required to materialize the views must be *less than* the expected next-query arrival time T, to ensure practical benefit.

Let $\text{Size}(V)$ denote the total storage required by the selected views in V. Thus, $\text{Size}(V) = \sum_{v \in V} \text{Size}(v)$, where $\text{Size}(v)$ denotes size of the view $v \in V$, when materialized. Let $\text{Time}(V)$ denote the total time needed to materialize the set V. If up to k views can be materialized in parallel, then $\text{Time}(V)$ depends on both the materialization times of individual views, $\text{Time}(v)$, and the materialization schedule across the k available workers. Thus, $\text{Time}(V) = \sum_{i=1}^{|V|/k} \text{Time}(v_{1+k(i-1)})$, assuming that V is sorted in descending order of $\text{Time}(v)$. Since exact evaluation of $\text{Size}(V)$ and $\text{Time}(V)$ is not feasible, we estimate $\text{Time}(v)$ using PostgreSQL's `EXPLAIN ANALYZE` on the view definition, and approximate $\text{Size}(v)$ from the query's row count and row width, estimated from column types. However, these estimates can be improved from historical data.

While the storage budget S is a user-defined parameter, the available materialization-time budget T depends on the prediction of the future queries. To model T, we consider two estimators:

$$T_{\min} = \min_{1 \le j \le m} (\hat{\tau}_j - \tau_n) \qquad T_{\exp} = \textstyle\sum_{1 \le j \le m} p_j (\hat{\tau}_j - \tau_n),$$

where $\hat{\tau}_j$ is the predicted arrival time of the future query q_j, τ_n is the arrival time of the most recent query in the workload $\mathcal{W}$, and p_j is the probability of q_j. $T_{\min}$ is conservative: it uses the earliest predicted arrival time, while $T_{\exp}$ is probability-weighted and represents the expected time until the next predicted query, giving higher weight to more likely queries. Finally, we use a linear combination of these two estimators to compute $T = \beta T_{\min} + (1 - \beta) T_{\exp}$ where the user can tune β to prioritize either of the estimators.

Optimization Problem. Combining all components above, the overall optimization problem that we aim to solve is:

$$M^* = \arg\max_{V \subseteq \mathcal{V}} \textstyle\sum_{j=1}^{m} p_j \, \text{Gain}(z_j \mid V)$$

subject to $\text{Size}(V) \le S$ and $\text{Time}(V) \le T$.

Since we limit $|\mathcal{V}|$ to at most 8, we solve the optimization via exhaustive enumeration of all subsets. For larger candidate sets, a greedy or ILP-based method is needed.

Preliminary Results. We evaluate PreView on IMDB [5] using a synthetic workload of 200 queries that simulates a 9-hour interactive session divided into three behavioral phases: an actor-centric morning phase, a genre-centric midday phase, and a director/production-centric evening phase. We benchmark 20 queries sampled uniformly across the three behavioral phases, and reset cache between experiments, averaging over three runs with a 100MB storage budget and latency estimate using $T_{\exp}$ ($\beta = 0$). PreView identifies up to 14 clusters for predicted queries, generates up to 8 candidate views, and materializes up to 7 under 4 seconds. On PostgreSQL, this yields a median speedup of 10.83× across the 20 benchmark queries, with individual speedups up to 30,000× (for cast-aggregation queries).

3 DEMONSTRATION

We demonstrate PreView over the IMDB dataset, following Kruti's scenario from Example 1, which consist of joins involving actors, movies, and genres with various parameters. We guide participants through ten steps (annotated in Figure 1), impersonating Kruti, who wants her queries to run faster.

Steps Ⓐ & Ⓑ (Loading the database and adjusting session context). Kruti enables *Baseline Comparison* in Ⓐ to see side-by-side latency comparisons during her session. She then selects the database type, uploads the data, and provides a historical query workload in Ⓑ. PreView loads the database schema and displays the available tables.

Steps Ⓒ & Ⓓ (Issuing and executing queries). Kruti then begins exploring the database by issuing SQL queries in the input panel in Ⓒ and executing them in Ⓓ. She does not need to manually materialize any view or rewrite her SQL queries, PreView does that automatically for her.

Steps Ⓔ & Ⓕ (Predicting future queries). As Kruti issues more queries over time, PreView identifies recurring patterns from both the historical workload and the live query session. It uses these patterns to predict likely future queries, shown in Ⓔ, with an estimated probability and expected time of arrival in Ⓕ.

Step Ⓖ (Materializing views). Using the predicted future queries, PreView automatically materializes views that are likely to accelerate upcoming queries, as shown in the *Materialized Views* panel in Ⓖ, along with their defining SQL, estimated size, build (materialization) time, and expected speedup.

Steps Ⓗ, Ⓘ & Ⓙ (Explaining how a query is re-written). The rewritten query is shown in Ⓗ, highlighting the sub-query

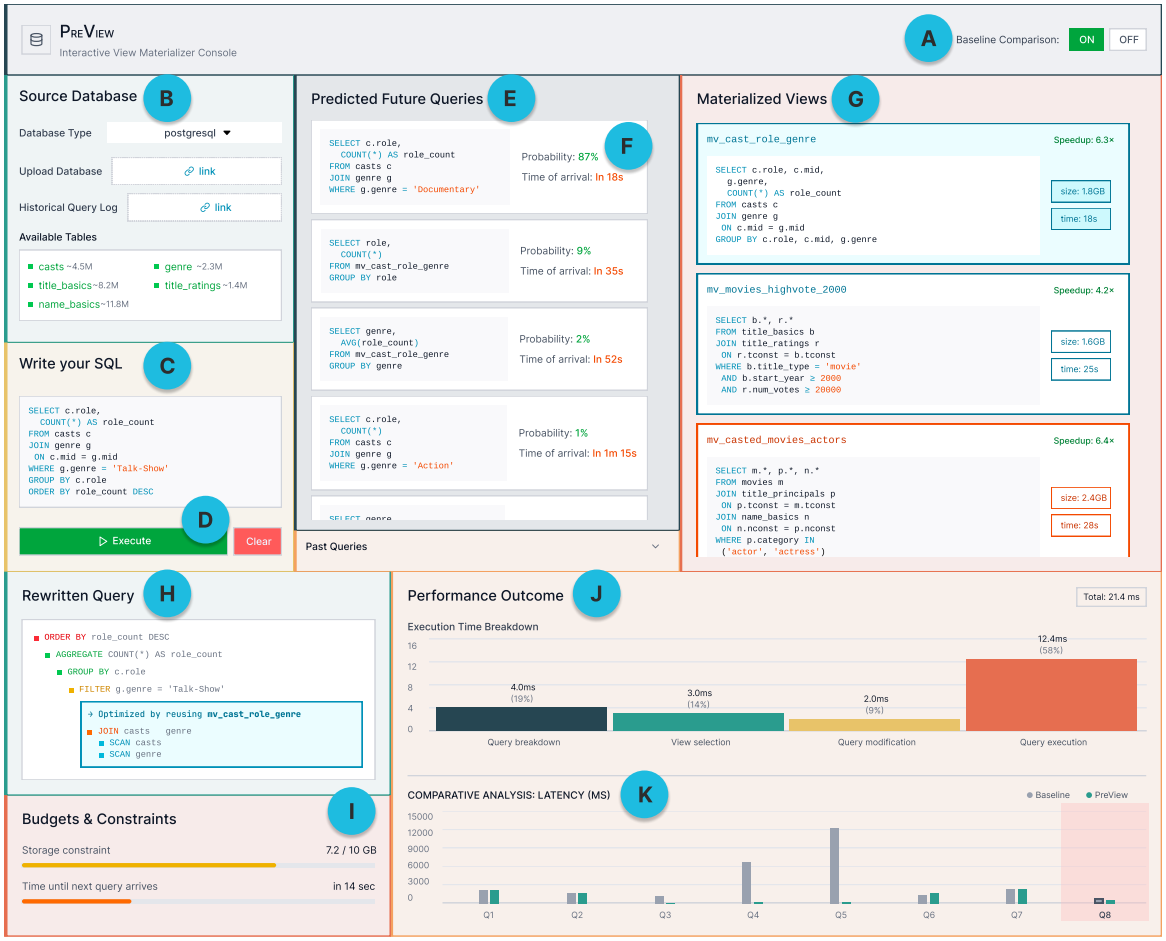

Figure 1: PreView's interface during Kruti's interactive exploration session: Ⓐ & Ⓑ loading the database and session context, Ⓒ & Ⓓ issuing and executing queries, Ⓔ & Ⓕ predicting future queries, Ⓖ & Ⓗ materializing views under budget constraints, Ⓘ & Ⓙ explaining query optimization and performance gains, and Ⓚ contrasting against the baseline latency throughout the session.

that was replaced by a materialized view. The *Budgets & Constraints* panel in Ⓘ shows the remaining storage budget and the time left before the next predicted query arrives. The *Performance Outcome* panel in Ⓙ shows a detailed execution-time breakdown across the major stages of PreView's pipeline, including query breakdown, view selection, query modification, and query execution. For the query shown in Ⓙ (Query 8), PreView reduces latency from a baseline of 836.7 ms to 21.4 ms, yielding a 39.1× speedup.

Step Ⓚ (Tracking speedup over the session). As the session progresses, the *Comparative Analysis* panel in Ⓚ updates continuously to report query runtimes of PreView against the baseline across all queries in the session.

Our target users are data analysts who want to accelerate their interactive workflow without manual tuning. After a guided walkthrough, participants can explore PreView with additional datasets. The key takeaway is that the natural pause between queries in an interactive session provides ample opportunity to proactively predict future queries and materialize views to reduce their latency. PreView demonstrates that this idle time, often overlooked, can be be leveraged for expediting query performance.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*.

[2] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*.

[3] Yue Han, Chengliang Chai, Jiabin Liu, Guoliang Li, Chuangxian Wei, and Chaoqun Zhan. 2023. Dynamic Materialized View Management using Graph Neural Network. In *ICDE*.

[4] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2023. AutoView: An Autonomous Materialized View Management System With Encoder-Reducer. *IEEE TKDE* 35, 6 (2023).

[5] IMDb. 2024. Non-Commercial Datasets. https://developer.imdb.com/non-commercial-datasets/. Accessed: 2025.

[6] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB* 11, 7 (2018).

[7] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc T. Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*.

[8] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*.

[9] Zhenrong Xu, Pengfei Wang, Guoze Xue, Qitong Yan, Shenghao Gong, Yelan Jiang, Yuren Mao, Yunjun Gao, Shu Shen, Wei Zhang, Dan Luo, and Lu Chen. 2024. UniView: A Unified Autonomous Materialized View Management System for Various Databases. *PVLDB* 17, 12 (2024).

[10] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE*.