SQL-ML
A SQL-Centric Framework for Building Efficient Feature Store

Ahmad Ghazal
PingCap
USA
ahmad.ghazal@pingcap.com

Hanumath Rao Maduri[*]
Workday
USA
hanu.ncr@gmail.com

Pekka Kostamaa
Teradata
USA
pekka.kostamaa@teradata.com

Abstract

Feature stores are centralized repositories for managing and serving ML model features, ensuring consistent data access during both training and inference. However, current solutions are often integrated into the ML stack and rely on proprietary systems, adding complexity. We argue that feature stores should be treated as a database problem due to their conceptual similarity to derived tables and materialized views, as current approaches are effectively reinventing well-established database concepts.

In this paper, we present SQL-ML, a SQL extension designed to support the creation, management, and backfilling of feature stores. SQL-ML treats feature stores as user-defined databases and features as materialized views, seamlessly integrating into existing SQL environments. While SQL-ML can be embedded in any SQL database, a more adaptable solution is to deploy it as middleware, interfacing with a host SQL database and leveraging its source data for feature storage. SQL-ML manages metadata independently while delegating data-related tasks—such as materialization, updates, global optimizations, and serving—to the host database. Global optimizations in SQL-ML minimize redundant scans and computations across features, significantly enhancing performance when thousands of features share common source data. Also, SQL-ML scales to large feature sets by leveraging database capabilities—for example, PostgreSQL supports up to 1,600 columns per table and imposes no practical limit on the number of tables.

We demonstrate SQL-ML's potential through a proof-of-concept implementation, extending PostgreSQL's functionality and using another PostgreSQL instance as the host database. Our experiments show that SQL-ML significantly improves both user experience—reducing the time required to create, manage, and backfill feature stores—and system efficiency, lowering resource consumption compared to other feature store solutions.

Keywords

SQL, Feature Store, Feature, PostgreSQL, Middleware

VLDB Workshop Reference Format:
Ahmad Ghazal, Hanumath Rao Maduri, and Pekka Kostamaa. SQL-ML A SQL-Centric Framework for Building Efficient Feature Store. VLDB 2025 Workshop: DATAI.

[*]The first two authors contributed equally to this paper.

1 Introduction

Machine learning, a branch of AI, enables systems to learn from data and improve over time without explicit programming. It uses algorithms and statistical models to analyze data patterns, allowing machines to make predictions or decisions. Initially focused on simple tasks, machine learning is now widely applied across industries like retail, finance, and technology. For optimal performance, models need extensive training and testing, but since real-world data often contains flaws, careful curation and cleansing are essential.

To address this challenge, feature stores were developed to streamline the cleaning and preparation of data for machine learning tasks. In this context, a feature is a measurable property or characteristic, often referred to as a variable or attribute. Features are essential for capturing the traits of the data being analyzed, enabling models to make accurate predictions or classifications. They can take various forms, including numerical, categorical, ordinal, binary, or text. A feature store serves as a centralized platform for the development, storage, modification, and reuse of these machine-learning features.

Features are derived from raw data, similar to summary tables or materialized views. However, modern feature stores—like Tecton, Feast, AWS SageMaker, Databricks, and Snowflake—treat feature management as an ML-specific task external to the database, using it only as a data source. This separation leads to redundant efforts in defining languages, managing metadata, and executing features. Most systems depend on custom, Python-based APIs, which hampers usability, extensibility, and portability. While some use SQL for feature computation, it's often embedded and lacks a true abstraction. Repurposing materialized views or Delta tables for features still requires custom pipelines and orchestration. A native SQL extension provides a cleaner, standardized, and declarative way to manage features without this complexity.

Some of the current feature stores use ad-hoc domain-specific languages (DSLs) and complex feature management life cycles. They often require proprietary storage for features, which can hinder their adaptability to different compute-engines. Additionally, their reliance on Python for relational operations—such as joins and aggregations—lacks standardization and can be cumbersome, presenting a steep learning curve for users.

Embedding feature stores directly within the database core is a more intuitive approach. By treating features as first-class database entities, SQL-ML provides a SQL-centric, efficient framework for feature store management. Key capabilities, such as backfill operations for retroactively populating feature data and global optimizations to minimize redundant computations, enhance SQL-ML's flexibility and performance. This paper makes two primary contributions: (1) a specification for extending SQL engines to support feature stores, and (2) an architecture for implementing SQL-ML as a middleware solution based on PostgreSQL. SQL-ML processes feature store requests, manages backfilling and optimizations, and integrates with any SQL database storing the source data. For simplicity, we refer to the middleware as SQL-ML throughout the paper.

The SQL extension introduced in SQL-ML includes commands for creating and dropping feature stores and individual features. Additionally, SQL-ML supports backfilling (UPDATE FEATURE) and describing (DESCRIBE FEATURE) features. Creating a feature store is analogous to creating a database (e.g., using *CREATE DATABASE* in PostgreSQL), while creating a feature is similar to creating a view (e.g., using *CREATE VIEW* in PostgreSQL). Feature stores and features are represented as new database objects, which requires extensions to the database metadata.

A SQL-ML-based feature store integrates seamlessly with the host database, requiring no changes to its syntax or metadata. SQL-ML manages the materialization of the feature store by representing it as a database or schema and its features as tables within the host database. This process is pushed down to the host database without altering its core functionality, ensuring that the feature store remains transparent and fully compatible with existing database operations.

SQL-ML interacts with the host database via SQL to implement feature store functionalities. For instance, *CREATE FEATURE-STORE...* in SQL-ML translates to *CREATE DATABASE...* on the host DB. Feature updates and materialization are handled by issuing insert-select queries, optimized by the host DB. However, SQL-ML can apply global optimizations using its knowledge of feature definitions, which is crucial when thousands of features share common source data. These optimizations, implemented through SQL rewrites, will be detailed in the paper.

Mature feature stores—often store features in the tens or hundreds of thousands. A common misconception is that relational databases cannot scale to support the large number of tables and columns. In practice, databases like PostgreSQL support up to 1,600 columns per table with no limit on the number of tables. TiDB also supports scalable and quick creation/migration of millions of tables demonstrating that relational systems are well-suited for enterprise-scale feature stores.

We built an initial prototype of SQL-ML using the open-source PostgreSQL. To support extended SQL, we modified the parser and added new metadata to handle feature stores, including information about the host database and specific features. While our implementation currently operates on another PostgreSQL instance as the host database, it is designed to be extensible to other host databases. Feature store and feature-related commands are executed by generating appropriate SQL queries and sending them to the host database.

We also implemented feature refresh through a background process, which periodically sends insert-select queries to the host based on the time window of the features. To evaluate our SQL-ML prototype, we used a sample feature store for ML models diagnosing heart disease. The results showed that users could create and start using the feature store within minutes, a significant improvement compared to the hours required when using Feast.

While model training and inference are essential stages in the ML lifecycle, they are downstream consumers of the feature store. SQL-ML focuses on efficient and scalable feature definition, maintenance, and serving—core responsibilities of any feature store. Demonstrating training or inference pipelines would conflate concerns and distract from the contributions of SQL-ML. Nonetheless, we provide a full example in Section 5 showing how features created by SQL-ML are directly consumable by predictive models, emphasizing its compatibility with model scoring workflows.

For the rest of the paper, we provide an overview of existing feature store solutions and their limitations (Section 2), followed by a detailed description of SQL-ML's architecture, including its SQL extensions, metadata management, and execution (Section 3). We then discuss feature maintenance processes, such as updates and backfilling, and the integration of global optimizations (Section 4). Next, we present SQL-ML's approach to feature serving (Section 5) and evaluate its performance compared to existing solutions through user experience and optimization experiments (Section 6). Finally, we outline future directions for SQL-ML and conclude with a summary of our contributions (Section 7).

2 Related Work

SQL has been extended for tasks such as ML training and prediction through frameworks like SQLFlow [25], which enables ML workflows directly in SQL by integrating with databases and ML engines such as TensorFlow and XGBoost. Other examples include Apache MADlib [13], which provides in-database analytics for PostgreSQL and Greenplum; PostgresML [14], which supports native ML model training and execution within PostgreSQL; RedshiftML [2], which enables creating, training, and applying machine learning models directly in Amazon Redshift using familiar SQL commands; and Microsoft SQL Server Machine Learning Services [19], which integrates Python and R scripts for predictive analytics directly within SQL Server.

While these extensions illustrate the versatility of SQL in bridging ML workflows with databases, they fall outside the scope of this paper. Our focus is solely on feature store management rather than the downstream consumers of feature stores, emphasizing efficient feature creation, maintenance, and optimization within SQL-centric workflows.

Broadly, feature stores can be categorized into two types: general-purpose feature stores, which are designed to cater to a diverse range of use cases across industries, and platform-specific feature stores, which are deeply integrated with specific cloud environments or ML frameworks. SQL-ML belongs to the general-purpose category because it operates independently of any specific platform, leveraging standard SQL to integrate seamlessly with a wide variety of databases and workflows. Thus, our primary focus will be on examining existing feature stores within this category, particularly Feast [5] and Feathr [6]. Our analysis is structured around three core dimensions: APIs, metadata management, and optimization strategies. Additionally, we provide a brief overview of platform-specific feature stores, such as SageMaker, Vertex AI, Snowflake, and Databricks, which are tightly coupled with their respective cloud or data platforms.

2.1 General Purpose Feature Stores

The two most popular general-purpose feature stores are Feast and Feathr. Both can be deployed on-premises or in the cloud.

2.1.1 *Feast*

Feast is an open-source feature store designed for managing and discovering machine learning features, and can be deployed both on-premises and across major cloud platforms, including AWS, GCP, and Azure. Its key capabilities include:

- **API:** A declarative DSL for defining features in Python (with embedded SQL) is provided, along with a Python CLI SDK for interacting with entities, feature views, and stores. Users can retrieve both historical and real-time features using functions like *get_historical_features* and *get_online_features*, with support for various data stores (e.g., Snowflake, Redshift, Postgres) and formats such as Parquet and CSV.
- **Metadata:** It utilizes a proprietary catalog to register feature definitions and metadata, enabling feature discovery and collaboration. The registry is file-based and compatible with both local and cloud storage systems like S3, GCS, and Azure, with automatic updates reflecting any changes.
- **Optimizations:** It relies on the underlying host databases to optimize the execution of SQL queries that are pushed down to them.

Tecton [9] is an enterprise-focused alternative that provides a similar feature retrieval API to Feast but is built on a distinct codebase. As a managed service, Tecton simplifies the creation and management of features by minimizing operational overhead while delivering enhanced functionality and scalability. Both Feast and Tecton include feature registries (supporting both online and offline use cases) and leverage pushdown optimizations to the host database. For the purposes of this paper, we treat Tecton and Feast as conceptually and architecturally similar [23], and therefore, we do not discuss Tecton further.

2.1.2 *Feathr*

Feathr is a scalable and unified platform designed for enterprise data and AI engineering, utilized by companies like LinkedIn and Microsoft Azure. Its core features include:

- **API:** It supports user-friendly Python APIs and customizable user-defined functions (UDFs) with support for PySpark and Spark SQL, facilitating easy use for data scientists across offline, streaming, and online environments.
- **Metadata:** Feathr supports a built-in proprietary catalog for creating and managing features. Users can create features on raw data from myriad data sources, can also search for features, explore metadata, manage access control, and share features within teams.
- **Optimizations:** Feathr primarily utilizes the Spark engine's optimizer. It has custom join implementations like Point in time joins which are useful for the feature engineering workloads [17].

Table 1 presents a comparison between SQL-ML and two leading general-purpose feature stores, Feathr and Feast. It highlights how current feature stores are reinventing the wheel when it comes to language (user interface) and metadata management, whereas SQL-ML builds on mature database technology for these crucial components. In terms of optimization, Feast, Feathr, and SQL-ML all rely on underlying databases to optimize feature computation and updates. However, SQL-ML goes a step further by leveraging its understanding of feature definitions to apply global optimizations, reducing redundant computation of shared source data.

2.2 Platform Specific Feature Stores

Many feature stores fall into this category, and we will briefly highlight a few key examples. Notable ones include F3 Feature Store [18], Google Vertex AI [11], AWS SageMaker [3], Databricks Feature Store [4], and Snowflake Feature Store [10]. These feature stores offer APIs for defining data objects within their ecosystem, such as entities, feature stores, feature registries, and support for creating and retrieving both offline and online features. While Python is the primary API, they also offer bindings for other programming languages. Although these stores allow for the specification of SQL dialects based on their underlying data sources, they generally lack the database abstraction that SQL-ML aims to provide. Furthermore, their optimization capabilities—such as global optimizations or advanced query-based techniques—are often limited.

Table 1: Comparison of Feature Stores

Feature Store	Language	Metadata	Optimizations
SQL-ML	SQL	SQL engine extensions	Host DB's optimizations + global optimizations
Feast	Python API with embedded SQL	Proprietary	Host DB's optimizations
Feathr	Python API with embedded SQL	Proprietary	Spark optimizer

3 SQL-ML

This section describes the architecture and design of the key components of SQL-ML. As mentioned earlier, SQL-ML is a middleware that extends PostgreSQL and is designed to operate on source data from any SQL-based database. While the middleware architecture offers broad compatibility and ease of deployment, database systems can alternatively choose to integrate SQL-ML functionality directly into their engines by extending metadata catalogs and translating SQL-ML commands internally. This direct integration is essentially a simplified special case of the middleware design, where the SQL-ML layer is embedded within the database itself. However, we do not elaborate on this variant in the paper and instead focus on the middleware-based approach, as it represents the more general and portable solution across diverse database backends. The overall architecture of SQL-ML and its core components is shown in Figure 1.

We built SQL-ML by extending the SQL language of PostgreSQL to allow creating feature stores and features. This extension is implemented by adding new grammar for handling the new objects (Parser component in Figure 1). The analysis phase (semantic analyzer) in PostgreSQL is also extended to support the new parse trees for feature stores and features (Analyzer component in Figure 1). Most of the planner work in PostgreSQL is not needed for SQL-ML and the new planner is just to apply global optimizations for feature refresh and produce the execution code which include: metadata update and generated SQL to be executed on the host DB.

The rest of this section is structured as follows: Section 3.1 presents the SQL extensions that form the core of SQL-ML, including commands for creating, updating, and describing features and feature stores. Section 3.2 explains the metadata layer and

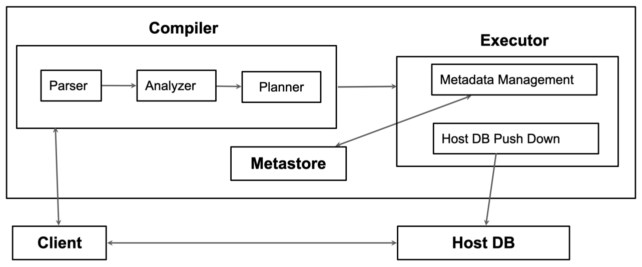

Figure 1: SQL-ML Architecture

its integration with the existing database catalog, enabling seamless management of feature-related metadata. Finally, Section 3.3 discusses the processing of SQL-ML commands, including parsing, semantic analysis, and execution, where commands such as creating or dropping feature stores and features are mapped to equivalent operations in the host database, ensuring efficient integration with SQL-ML's metadata and execution framework.

3.1 SQL Extensions

Existing SQL constructs like materialized views or Delta tables can be used to approximate feature store functionality, doing so typically requires ad hoc conventions, custom metadata tracking, and external orchestration logic for updates, backfills, and serving. These approaches lack a standardized abstraction, forcing users to build and maintain bespoke pipelines and tooling. By introducing a SQL extension, SQL-ML elevates features and feature stores to first-class database objects with declarative semantics for creation, maintenance, and introspection—removing the need for external glue code and enabling native optimizations within the SQL engine.

SQL-ML language is an extension of PostgreSQL's SQL, a widely used and highly standards-compliant dialect. The extension cover creating/dropping feature stores and features. The metadata management and executions of these new commands are covered in the next two sections. In this section, we cover the new grammar with some examples.

Creating a feature store is simple and similar to creating a database in PostgreSQL but most of the options are not related to our context. The SQL commands to create or drop feature stores are below.

```
CREATE FEATURESTORE name
    [ WITH ] [ OWNER [=] user_name ]
DROP FEATURESTORE name
```

where name is the name of the feature store in both cases. For the create statement, there is an optional specification of feature store owner (default is current user). For example, *CREATE FEATURESTORE ride_share;, DROP FEATURESTORE ride_share;* and creates and drops the ride_share feature store respectively.

The feature concept in SQL-ML corresponds to multiple values(columns) used by ML and therefore it is represented and implemented as a table. The syntax of creating/dropping features is below. For simplicity, we are not including *ALTER FEATURE* command.

```
CREATE FEATURE [IF NOT EXISTS] [feature_store_name.]
     feature_name <column_list> AS <sql_query>
PARTITION BY <date_time_sql_expression> BY <granuality>
```

```
DROP FEATURE [IF EXISTS] [feature_store_name.]
     feature_name
```

Features only exist in feature stores and feature_store_name is optional. *<column_list>* is list of column names for the feature. Also, *<sql_query>* is a *SELECT* statement written using PostgreSQL syntax and it is the same used for *CREATE VIEW* or *CREATE TABLE AS*. ML typically extracts features based on a time window like "daily active user count in Facebook" and *<date_time_sql_expression>* is used to represent that time window. *<date_time_sql_expression>* could be a date/timestamp column or expression and it is also used by the refresh background process discussed in Section 3.3. The <granularity> expression specifies the interval used to compute partitions for features, and it can be set to HOUR, DAY, WEEK, MONTH, or YEAR.

The following is a simple example of creating a feature named trip_rollup:

```
CREATE FEATURE IF NOT EXISTS rideshare.trip_rollup AS
SELECT driver_id, COUNT(*) AS total_trips
FROM rideshare_rawdata.driver_stats
GROUP BY driver_id
PARTITION BY creation_timestamp BY DAY;
```

The *trip_rollup* feature is defined in the *ride_share* feature store and is designed to calculate the total number of trips per driver daily. The data source is the *driver_stats* table in the *ride_share_rawdata* database. This table includes, among other fields, *driver_id* (which uniquely identifies each driver) and *creation_timestamp*, which records the time a trip was logged. The feature uses the column *creation_timestamp* to compute daily aggregates, as specified by the PARTITION BY ... BY DAY clause in the feature definition.

SQL-ML also supports on-demand backfilling of feature data using the UPDATE command with the following syntax:

```
UPDATE FEATURE [feature-store-name.]<feature-name> WHERE
     <partition-by-col-predicate>
```

In the UPDATE FEATURE command, the WHERE clause is composed of range predicates on the partition-by field. SQL-ML computes and stores the specified partitions if they do not already exist. Further details on the operation of this command are provided in Section 4.2.

Users can consume feature data directly from the host and to facilitate that we provide the DESCRIBE FEATURE command that shows the table and its partitions information on the host DB. The syntax is:

```
DESCRIBE FEATURE [feature-store-name.]<feature-name>
```

Finally, the SHOW FEATURE command can be used to show the feature DDL (Data Definition Language) statement that was used to create the feature.

3.2 Metadata Extensions

This section outlines the metadata layer for registering FEATURESTORE and FEATURE entities. By seamlessly integrating these FEATURE abstractions into the database, they are stored alongside other database objects like TABLEs and VIEWs within the same catalog tables. This strategy allows us to make only minimal modifications to the system catalog in order to register the metadata objects associated with these new entities.

In our SQL-ML implementation, we implemented the following changes to the PostgreSQL database catalog:

3.2.1 FEATURESTORE metadata: PostgreSQL includes the metadata table *pg_namespace* to capture SCHEMA information. Due to the conceptual and structural similarities between FEATURESTORE and SCHEMA, we utilize *pg_namespace* to represent feature stores by adding a new column, nspkind, to distinguish between the two types of objects.

3.2.2 FEATURE metadata: In SQL-ML, a FEATURE is analogous to database views, and we leverage PostgreSQL's system catalog table *pg_class*, which contains information about all tables and views. We made a minor extension to this table to distinguish FEATURES from traditional tables and views.

3.2.3 System views: System views provide a secure way to access relevant metadata. PostgreSQL includes several built-in views, such as *pg_tables*, for querying existing tables. Similarly, we have introduced two new system views, *pg_feature* and *pg_featurestore*, that are specifically related to FEATURE and FEATURESTORE objects. Below is an example of how to use the *pg_feature* view.

```
SELECT * FROM pg_feature;
```

Table 2: Sample output from `pg_feature`.

feature name	feature store	partition column	partition by granularity	last_updated
heart_data	health	record_dt	day	01-22-2023 20:07:40

The output in Table 2 displays a FEATURE called *heart_data* which is part of the FEATURESTORE *health*. This FEATURE is partitioned on a column called *record_dt* with partition_by_granularity of DAY. The column *last_update* indicates the time stamp when the feature was last updated.

3.3 Planner & Execution Extensions

This section discusses the processing of the CREATE and DROP commands for FEATURE and FEATURESTORE in SQL-ML. The process involves parsing and analyzing these commands in SQL-ML, followed by the materialization of the objects in the host database and the updating of SQL-ML metadata. To illustrate the end-to-end process, we use the previous example of the feature *trip_rollup* in the *rideshare* feature store. It is important to note that SQL-ML does not modify or constrain how features are consumed. The tables created and maintained by SQL-ML follow standard SQL semantics and are readily accessible by any external ML framework or in-database ML system. This makes SQL-ML a transparent and interoperable feature store backend, capable of serving training and inference workflows without requiring additional APIs or glue code.

Our initial implementation of SQL-ML uses PostgreSQL as the host database, which influences how we materialize and optimize feature processing. However, this approach can be easily extended to other host databases, a step we plan to take in the future.

3.3.1 Parser: The SQL-ML parser is based on the PostgreSQL dialect, as PostgreSQL is widely used and shares grammar similarities with many newer databases [12]. We added few more grammar rules to support in addition to the syntax extension, we extended its grammar to support SQL-ML commands like CREATE FEATURESTORE, CREATE FEATURE, and their corresponding DROP commands.

When a user submits an SQL statement, the SQL-ML parser generates an abstract syntax tree (AST) that captures essential details. For example, the AST for a CREATE FEATURESTORE command includes: 1) whether to create it if it doesn't exist, 2) the configuration file for HostDB, and 3) the FEATURESTORE name.

Similarly, for the CREATE FEATURE command (e.g., *trip_rollup*), the AST captures: 1) creation conditions, 2) FEATURE name (with optional FEATURESTORE prefix), 3) the SELECT query, 4) the partition column, and 5) refresh granularity.

3.3.2 Analyzer: This phase primarily involves semantic checks. The input for this phase is the abstract syntax tree (AST) generated during the Parser phase (see Fig 1). For the SQL-ML extensions, we conduct two types of semantic checks: one concerning the SQL-ML metastore and the other regarding the host database which are part of execution phase. Some of the semantic checks specific to the SQL-ML layer include verifying whether a feature already exists, whether the feature has a valid partition-by column etc.

3.3.3 Execute: This phase involves executing the corresponding command for each SQL-ML extension DDL. It also identifies and reports semantic check errors related to the host database. For instance, if a database called *rideshare* for the underlying FEATURESTORE already exists in the host database, then an error will be reported.

Each SQL-ML command corresponds to a command in the host database, and it is during this phase that the relevant command in hostDB is executed. Table 3 provides a mapping for the CREATE and DROP commands related to FEATURE and FEATURESTORE. For example, when a user executes a CREATE FEATURESTORE command, the corresponding *CREATE DATABASE <fs-name>* command is executed after the Abstract Syntax Tree (AST) is generated by the parser and validated during the analyzer phase.

Table 3: Mapping of SQL-ML Commands to Host Database Commands

SQL-ML Command	HostDB Command
create featurestore <fs-name>	create database <fs-name>
drop featurestore <fs-name>	drop database <fs-name>
create feature <f-name> <query>	create table <f-name>
drop feature <f-name>	drop table <f-name>

To better understand the execution logic, let us use an example of creating a feature store named *rideshare* along with the feature *trip_rollup* (as described earlier). Figure 2 depicts the execution logic of `CREATE FEATURESTORE rideshare` where the command goes through the parser and semantic analyzer that checks if `rideshare` exists and if the user has the proper access rights. If no semantic

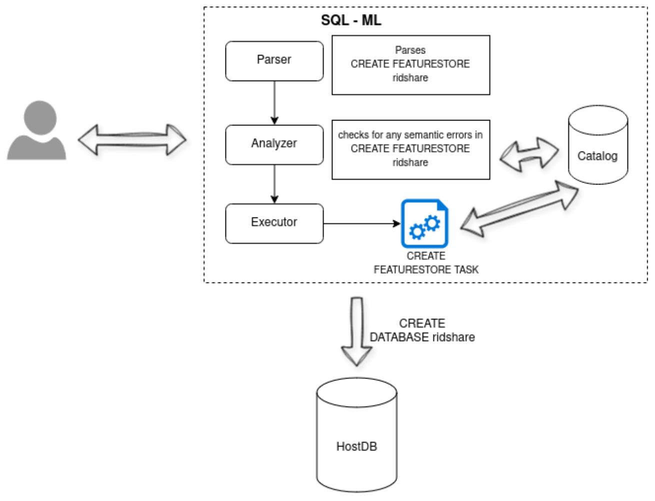
SQL - ML
Parser
Parses CREATE FEATURESTORE ridshare
Analyzer
checks for any semantic errors in CREATE FEATURESTORE ridshare
Catalog
Executor
CREATE FEATURESTORE TASK
CREATE DATABASE ridshare
HostDB

Figure 2: CREATE FEATURESTORE

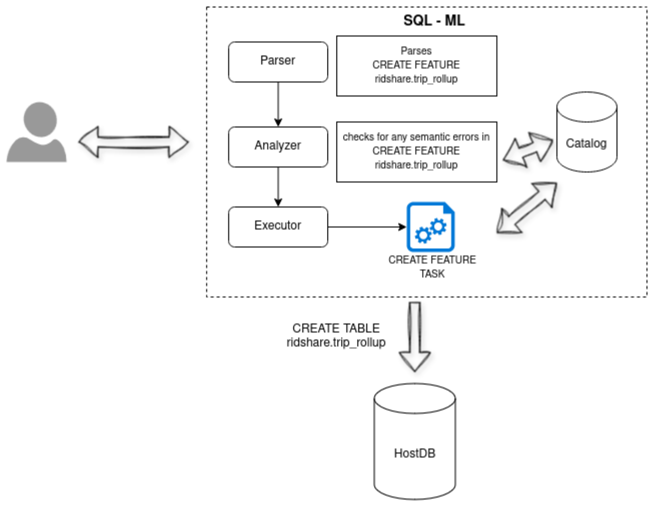
SQL - ML
Parser
Parses CREATE FEATURE ridshare.trip_rollup
Analyzer
checks for any semantic errors in CREATE FEATURE ridshare.trip_rollup
Catalog
Executor
CREATE FEATURE TASK
CREATE TABLE ridshare.trip_rollup
HostDB

Figure 3: CREATE FEATURE

errors, SQL-ML creates a connection with the PostgreSQL host database and submits `CREATE DATABASE rideshare`. If the command is successful then SQL-ML adds it to its metadata. Otherwise, SQL-ML issues an error to the user. This paper does not show the details of the error checking and reporting.

Similarly, Figure 3 illustrates the process of creating the FEATURE *trip_rollup*. The command undergoes parsing and resolution, where access rights are verified and the existence of the object is checked. As with creating a feature store, SQL-ML establishes a connection and submits `CREATE TABLE trip_rollup` to the host database. Upon successful table creation, SQL-ML registers `trip_rollup` in its metadata and reports an error otherwise. Below is the SQL command that is submitted to the PostgreSQL host database.

```
CREATE TABLE rideshare.trip_rollup (
    driver_id INT,
    total_trips INT,
    creation_timestamp TIMESTAMP
) PARTITION BY RANGE (creation_timestamp);
```

Creating the feature does not require any materialization, as this is managed by either the periodic feature update or an explicit feature backfill request from the user. We discuss both of these processes in the following sections.

4 Feature Maintenance

Efficiently managing the lifecycle of machine learning features is crucial for maintaining accurate and up-to-date models. In SQL-ML, feature updates and backfills are integrated into a streamlined process designed to handle large-scale data while minimizing manual intervention. This section describes SQL-ML's approach to updating features with automated, partition-based refresh cycles, allowing for timely feature maintenance aligned with common ML requirements. Additionally, SQL-ML introduces a powerful global optimization capability that leverages shared computations across features, significantly reducing redundancy and improving performance.

4.1 Feature Update

In SQL-ML, features are partition-based and can use one of five partition granularities: hourly, daily, weekly, monthly, or yearly, with hourly and daily being the most common. SQL-ML refreshes and updates features automatically through background processes. To streamline updates, five main processes are configured—one for each partition type. These processes activate at specific partition boundaries: the start of each new hour, day (midnight daily), week (midnight every Sunday), month (midnight on the first day of each month), and year (midnight on New Year's Day). At each boundary, the process identifies all relevant features with matching partition boundaries and initiates an update process for each feature.

For instance, the trip_rollup feature is partitioned daily using the creation_timestamp column. The daily update process involves generating a new partition and inserting data into it. To maintain consistency, the new partition table is named using the original table name as a prefix and the corresponding timestamp as a suffix. For example, updating trip_rollup on 11/09/2024, would create a table with a suffix based on the timestamp value 1731202290, which represents November 9, 2024, at 00:00:00. Note that both the creation_timestamp and driver_id fields are sourced from the raw dataset driver_stats.

```
CREATE TABLE rideshare.trip_rollup_1731202290
PARTITION OF rideshare.trip_rollup
FOR VALUES FROM '11-08-2024␣00:00:00' TO '11-09-2024␣
    00:00:00'
```

```
INSERT INTO rideshare.trip_rollup_1731202290 (driver_id,
    total_trips)
SELECT driver_id, COUNT(*) AS total_trips
FROM rideshare_rawdata.driver_stats
WHERE creation_timestamp >= '11-08-2024␣00:00:00' AND
    creation_timestamp < '11-09-2024␣00:00:00'
GROUP BY driver_id;
```

SQL-ML simplifies feature updates and maintenance by supporting a straightforward UPDATE command, intentionally limiting the use of complex subqueries or advanced functions like PIVOT during the update process. However, these advanced operations can still be leveraged during feature creation. Furthermore, global optimizations for each of the 5 partition-based update processes that leverage common intermediate results among features are also performed. The cross-feature or global optimizations are covered in Section 4.3.

4.2 Feature Backfill

Feature backfill involves adding data to an existing feature, specifically for partitions that have not yet been computed. SQL-ML computes feature partitions based solely on the current date, time, or timestamp. Users can use the UPDATE FEATURE command, as described in the syntax extension section, to backfill partitions that does not exists and ignores those that is already created. The example below uses UPDATE to backfill *rideshare.trip_rollup* for first day of November 2024.

```
UPDATE FEATURE rideshare.trip_rollup
WHERE creation_timestamp BETWEEN '11-01-2024 00:00:00'
    AND '11-02-2024 00:00:00';
```

The UPDATE execution logic starts by identifying all partitions that match the WHERE clause, skipping non-empty partitions, and then applies the feature update logic to the empty ones. The backfilling process for each empty partition follows the same steps as creating new partitions and inserting data, as described in the previous section.

4.3 Global Optimization

Global optimization seeks to identify the best possible solution, or global optimum, across the entire feasible solution space, as opposed to settling for a locally optimal solution within a constrained region. In the context of database research, this concept is typically referred to as multi-query optimization [8, 16, 20–22, 24]. Optimal multi-query optimization is an NP-hard problem [22] which requires exponential time.

Global optimization in SQL-ML focuses on identifying common table scans across features, which is particularly effective since the number of features often far exceeds the number of source tables. The commonality between two features is determined at each table scan by checking if they share the same filter, a common scenario when data is pulled from source tables based on partition boundaries. The complexity of finding common scans between two feature plans in SQL-ML is $O(n_1 + n_2)$, using hash-based indexing, where n_1 and n_2 represent the number of scan nodes in each plan. This complexity simplifies to $O(n)$ when n is the maximum number of scans in both features. Extending this to all k features, the complexity of identifying common scans becomes $O(k^2)$, resulting in an overall complexity for the global optimizer of $O(nk^2)$.

For example, in our evaluation, we use two source tables—one for patients and another for health information—to construct 20 distinct features. The SQL-ML global optimizer operates within the feature update component, which includes five main processes (hourly, daily, weekly, monthly, and yearly) discussed earlier. Each process invokes the global optimizer with insert-select plans as input, represented as directed acyclic graphs (DAGs). The optimizer examines these DAGs, identifying opportunities where features share identical scan filters. It then rewrites the DAGs to consolidate these common components by constructing temporary results for the shared portions. However, this rewrite may be less efficient when the source data is indexed or partitioned by timestamp. To handle this, the SQL-ML optimizer compares the cost of executing the query with and without the rewrite for each common set of features. It does this by using "explain" query on the host database to estimate the cost and then chooses the lower-cost option.

The algorithm below summarizes how feature updates work in sync with the SQL-ML global optimizer. The algorithm identifies features that match specific partition boundaries, generates and optimizes insert-select plans for each feature, creates new partitions, and then executes the optimized plans to update each partition efficiently.

Algorithm 1 Feature Update Process in SQL-ML

```
1: Identify features with matching partition granularity
2: Create create table DDL for new partitions
3: Generate insert-select plan for each partition
4: Send all plans to the global optimizer
5: Execute DDL to create new table partitions
6: Execute insert-select statements into table partitions based on
   optimized plans
```

We illustrate the details of the global optimizer through an example. Consider the two features: *revenue_rollup* and *trip_rollup* (described earlier). The *revenue_rollup* feature captures the total revenue per driver per day, thereby enhancing the model's predictive power by incorporating both the number of trips and the associated revenue for each driver. The definition of this feature is provided below which shows that both features are based on *ride_share_rawdata.driver_stats* and partitioned the same way on *creation_timestamp* daily.

```
CREATE FEATURE IF NOT EXISTS rideshare.revenue_rollup AS
select driver_id, sum(revenue) as total_revenue from
    rideshare_rawdata.driver_stats group by driver_id
PARTITION BY creation_timestamp BY DAY;
```

Let us assume that the update module built the insert select part of each of the features for 11/11/2024. The DAGs are shown in Figures 4 and 5, respectively.

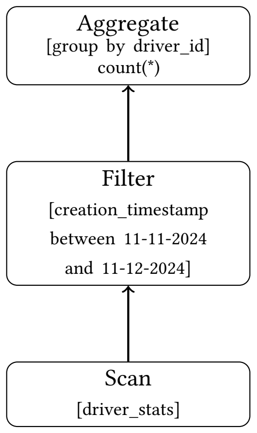

Figure 4: DAG for the trip_rollup

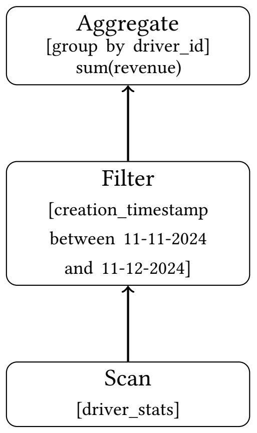

Figure 5: DAG for the revenue_rollup

Both features share the same source, partition column, and filter conditions which allows sharing the table scan and filter operations. The optimized plan in this example is shown in Figure 6 which computes the common results. The resulting data is written to a temporary table via a spool operator. Let us assume that the global optimizer finds out this plan is cheaper than the individual plans which can be due to lack of good access paths for the *driver_stats* table.

Subsequent operators, such as aggregation, which are specific to each feature, are processed by reading from this temporary table. After the aggregation operations are completed, the resulting feature values—specific to each feature—are inserted into their respective feature tables. In the next section, the process of serving these features is explained, where the feature values are retrieved from these tables for further use.

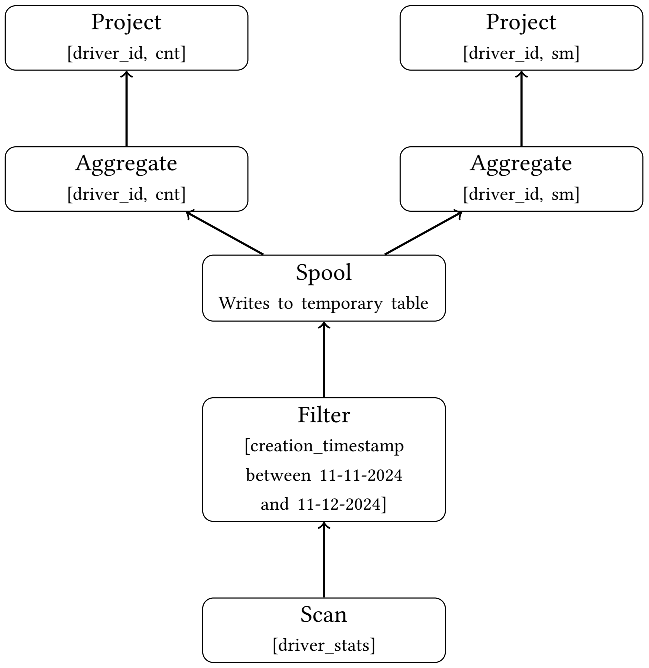

Figure 6: DAG after the Global Optimization

We conclude this section by demonstrating how we separated common components, such as creating new partitions (e.g., create table partition and insert-select into new partitions), between feature updates and backfilling, as shown in Figure 7. The figure also illustrates global optimization as a subcomponent of the feature update process.

4.4 Feature Maintenance Summary

In this section, SQL-ML's approach to efficiently managing feature updates and backfilling is detailed, emphasizing the system's reliance on partition-based refresh processes. By leveraging granularity settings (e.g., hourly, daily) for feature partitions, SQL-ML automates periodic updates, ensuring features are timely and require minimal manual intervention. SQL-ML also supports on-demand

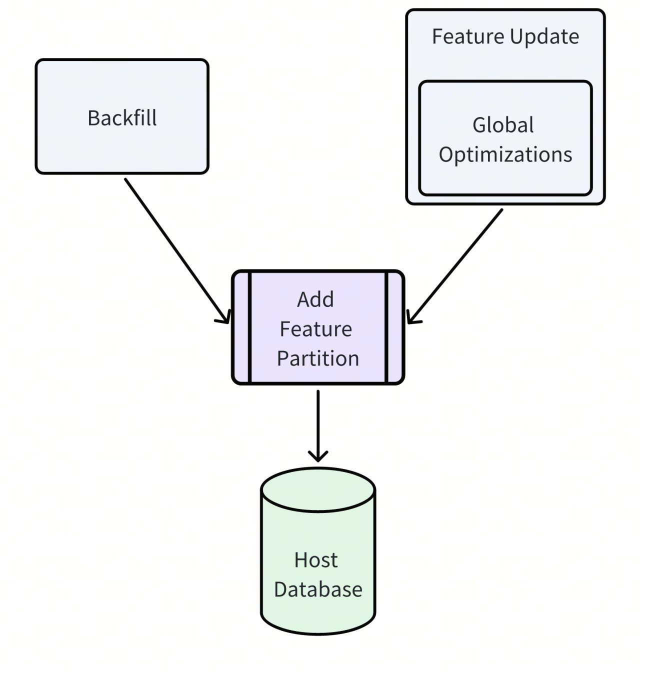

Figure 7: FEATURE MAINTENANCE

backfilling, allowing users to retroactively populate feature data based on specified date or time ranges.

A key advantage of SQL-ML is its global optimization capability, which minimizes redundant computations across features that share similar source data. The optimizer consolidates common table scans, reducing resource usage and improving performance significantly, especially in large-scale ML applications. This optimization is showcased through examples, demonstrating how SQL-ML reduces overhead by sharing common intermediate results for multiple features, unlike conventional feature stores that lack such query optimization.

5 Feature Serving

Feature serving in SQL-ML is straightforward, as it simply directs the user to the location where the feature results are stored. Our approach is effective because all feature computation is offloaded to the host database, making it logical to access the feature data directly from its storage location. We have introduced the DESCRIBE FEATURE command in SQL-ML, which offers a detailed description of the table names and partitions that can be scanned to generate the corresponding feature values. This command is briefly discussed in the section 3.1.

For example, here's the output of this command for an existing feature called *trip_rollup*:

```
DESCRIBE FEATURE rideshare.trip_rollup;
```

The output from the above SQL-ML describe command for a feature with three partitions would look like this:

ML prediction and learning are the most common consumers of features, though they are outside the primary scope of feature

Table 4: Output for describe trip_rollup

table-name	partition-name
trip_rollup	trip_rollup_1727654400
trip_rollup	trip_rollup_1727481600
trip_rollup	trip_rollup_1727395200

stores. However, a simple example illustrating how ML can leverage features created and maintained by SQL-ML can help readers understand this relationship. We use a prediction example that utilizes *trip_rollup* and *trip_revenue* features to rank drivers based on their activity and revenue. This example, built in SQL, supports the increasingly prevalent practice of running ML directly within databases [1]. SQL-ML provides a distinct advantage by keeping raw data, feature computation, and consumption closely integrated.

For this example, let us first define what is the ranking function for drivers. A simple PLSQL UDF function is defined below that combines the two features into a single rank value. Below is the function definition:

```
CREATE OR REPLACE FUNCTION rank_driver(total_trip_count
     NUMERIC, total_revenue NUMERIC, trip_count NUMERIC,
     trip_revenue NUMERIC)
RETURNS NUMERIC AS $$
DECLARE
    alpha NUMERIC := 0.5;
    beta NUMERIC := 0.5;
    rank_score NUMERIC;
BEGIN
    rank_score := (alpha * trip_count/total_trip_count) +
         (beta * trip_revenue/total_trip_revenue);

    RETURN rank_score;
END;
$$ LANGUAGE plpgsql;
```

The following SQL query ranks the drivers on a specific day based on their performance score defined above by rank_driver. The example uses the day with timestamp 1727654400 which leads to using the partitioned tables *trip_rollup_1727654400* *trip_revenue_1727654400*

```
WITH total_stats AS (
    SELECT
        COUNT(*) total_trip_count,
        SUM(revenue) total_trip_revenue
    FROM driver_stats
)
SELECT
    driver_id,
    rank_driver(total_trip_count, total_trip_revenue, SUM
         (trip_rollup), SUM(trip_revenue)) AS
         performance_score
FROM trip_rollup_1727654400 as trol inner join
     trip_revenue_1727654400 as trev on trol.driver_id =
     trev.driver_id, total_stats
group by driver_id
ORDER BY performance_score DESC;
```

While the current SQL-ML prototype requires users to query feature partitions directly, this design prioritizes simplicity and transparency in initial deployments. However, we acknowledge that querying partition tables may be unintuitive for some users. As part of future work, we plan to support direct SQL access to FEATURE objects (e.g., SELECT ... FROM feature_name) by introducing logical views that unify partitions. These views will internally rewrite timestamp filters to access the appropriate partition(s), providing a more intuitive and streamlined querying experience for both training and inference workloads.

6 Evaluations

In this section, we present the evaluation results for SQL-ML through two experiments. The first experiment demonstrates the user experience of creating a simple feature store with SQL-ML and compares it to Feast. The second experiment highlights the potential of the SQL-ML global optimizer by evaluating the performance of a model using a SQL-ML feature store with 20 features, both with and without optimization.

Evaluations are carried out using publicly available heart disease data [[7], [15]], with two imported tables: `patient_details` [A.2] and `heart_data` [A.1], which capture patient information and heart-related data, respectively. The original data set was relatively small, so we synthesized 100 GB of data based on it.

The SQL-ML setup consisted of a single instance of SQL-ML and a PostgreSQL database as a host. Both the SQL-ML platform and the host PostgreSQL shared a similar configuration, running on a basic Linux system with an AMD RYZEN 7 4700U processor, 16 GB of RAM, 256 GB SSD storage, and Linux (Ubuntu 22.04 LTS) as the operating system.

6.1 User Experience Test

SQL-ML Evaluation This test represents a feature store with two features: cholesterol level and heart rate for patients. For SQL-ML, a feature store named `health` [B.1] was defined to store these features. We created two features: `cholesterol` (representing patients' cholesterol levels) [B.2] and `heart_rate` (representing patients' heart rates) [B.2]. A heart disease prediction model was added, utilizing our feature store. The prediction model is implemented as a PostgreSQL UDF [B.3] that declares a heat disease if both cholesterol and heart_rate are above a certain level. The prediction for all patients is done through a SQL query [B.3] that invokes the UDF to calculate the risk of heart disease for all patients based on data from a specific day's data source.

The entire process—including feature definition, materialization, and predictions—was completed within **1 hour**. Detailed feature definitions can be found in the Appendix.

Feast Evaluation We conducted the same heart disease prediction test above. We used the same configuration on Feast version 0.29.0. The creation of features in Feast required several steps: (1) adding the source data using the `Entity` abstraction [C.1], (2) creating data sources based on the entities [C.2], (3) defining and registering features [C.3], and (4) configuring a materialization schedule for the features [C.4]. All these steps were implemented in Python using the custom APIs provided by Feast, which abstract data sources, features, and materialization windows (analogous to partition granularity in SQL-ML). Feature serving in Feast also relies on Python APIs, which we used to implement the heart disease prediction code.

Implementing the heart disease prediction workflow with Feast took over 10 hours of coding and testing, compared to just 1 hour using SQL-ML. Both estimates exclude time spent on installation, setup, or learning the system. While the comparison is not a formal user study and depends on the user's familiarity with SQL or Python, the difference reflects the nature of each system. Feast requires multiple configuration steps using custom Python SDKs and CLI tools, whereas SQL-ML relies solely on declarative SQL. This streamlined approach reduces boilerplate and lowers the barrier to entry for data analysts who are already familiar with SQL. Overall, the comparison illustrates how SQL-ML simplifies feature engineering by leveraging the native capabilities of the database.

6.2 Global Optimizations Test

On average, prediction models in healthcare utilize between 10 and 20 features, with the exact number depending on the complexity of the condition and the availability of data. To explore global optimizations, we extended the heart disease prediction model to include a total of 20 features. Of these, 13 are health factors from the original heart dataset, such as age, sex, chest pain type, resting blood pressure, serum cholesterol levels, fasting blood sugar, and other attributes listed in the original health data table [A.1]. The remaining 7 features were synthetically generated to augment the feature store, resulting in a dataset with 20 features in total. The additional 18 feature definitions are analogous to `cholesterol` and `heart_rate` used in the user experience test described above and are omitted here for brevity.

We also extended the prediction model to leverage all 20 features. While our test primarily focuses on measuring the performance of feature materialization with and without global optimization, we describe the prediction model here for completeness. The model is based on logistic regression, which produces a binary output indicating the presence or absence of heart disease. The prediction is defined by the function:

$$P(\text{Heart Disease}) = \frac{1}{1 + e^{-z}}$$

Heart disease is predicted if $P(\text{Heart Disease}) > 0.5$, and no heart disease is predicted otherwise. The value of z, known as the linear predictor, is defined as:

$$z = \sum_{i=1}^{20} \beta_i \cdot f_i$$

Where:

- β_i: The coefficient for feature f_i, representing its contribution to the prediction.
- f_i: The i-th feature, such as age (f_1) or sex (f_2).

We used the same hardware, and we added 7 columns to the original heart_date table to get the raw data for all the 20 features resulting in 150 GB total source data on the host database. The source data covers a span of 60 days of health information for the 100 million patients. The test measures the performance of feature materialization for a specific day for all patients with (per section 4.3) and without global optimization where features are materialized independently and without common temporary results.

Figure 8 compares the performance of global optimization versus no global optimization for different numbers of features (2, 10, and

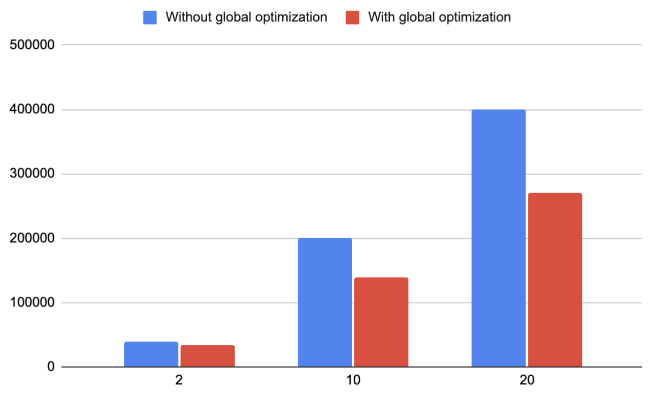

Figure 8: With & Without Global Optimization- No Index

20), using the heart dataset without indexing. The results demonstrate that global optimization provides significant performance improvements as the number of features increases, with reductions ranging from 6 seconds in the 2-feature test to 129 seconds in the 20-feature test. This increase in benefit is logical, since the cost of building the common result is incurred only once, and the advantages grow as more features are processed.

We also explored a scenario where indexing might reduce the advantages of global optimization by limiting the benefits of shared access to common table data. Our objective was to examine a negative case in which the cost of constructing common results exceeds their benefits, prompting the cost-based global optimizer to reject this strategy. However, this proved challenging, as the construction of common results remained advantageous even with an index present in the scope. This is because, within the initial scope, the common results align precisely with table scans and filters, making them inherently efficient. Future work involves broadening the scope of global optimizations to create more opportunities for constructing shared results. However, this also requires the global optimizer to make cost-based decisions, as the overhead of shared results may, in some cases, outweigh their benefits.

Figure 9 illustrates the results of our experiment, which utilized an index on the timestamp column of the patient data. The index significantly reduced the data accessed to approximately 1.7% of the original size, or about 1/60th. Despite this, the index still accounted for roughly 15% of the overall query performance improvement. Consequently, the SQL-ML optimizer continued to employ global optimizations, achieving performance gains ranging from 3 to 48 seconds.

7 Conclusion and Future Work

In this paper, we presented SQL-ML, a SQL-centric framework that simplifies the management and retrieval of machine learning features by embedding feature store capabilities directly into the SQL ecosystem. By treating features as database objects, SQL-ML streamlines feature storage and management, enabling seamless integration with existing SQL workflows. Our prototype, built on PostgreSQL, demonstrated SQL-ML's potential to enhance user experience and system efficiency by minimizing redundant computations

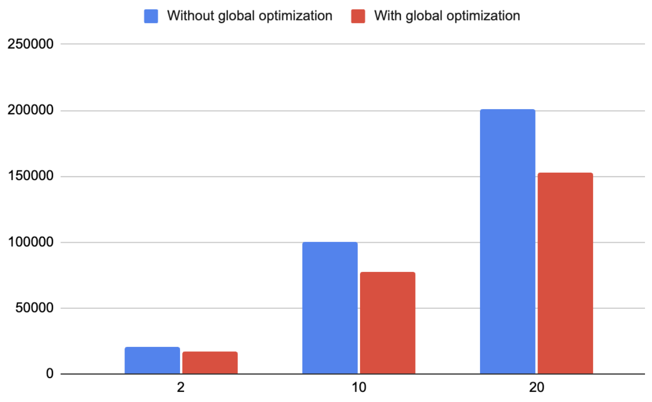

Figure 9: With & Without Global Optimization- Index

and leveraging SQL for robust feature optimizations. Compared to conventional feature stores, SQL-ML reduces the operational overhead, achieves faster feature creation, and improves performance by incorporating multi-query optimizations and periodic feature refreshes.

To further enhance SQL-ML's applicability and efficiency, future work will explore: (1) adding support for additional host databases, such as MySQL, to increase SQL-ML's adaptability across various database environments, (2) extending the global optimization capabilities for joins and aggregations by leveraging current multi-query optimizations, which will improve query efficiency for more complex feature engineering tasks, and (3) expanding SQL-ML to support streaming engines, enabling real-time feature updates and maintaining freshness for time-sensitive applications. This evolution will empower SQL-ML to become a versatile, highly efficient tool in feature store management for diverse use cases in machine learning.

References

[1] [n.d.]. *10 databases supporting in-database machine learning | InfoWorld.* https://www.infoworld.com/article/2262611/10-databases-supporting-in-database-machine-learning.html

[2] [n.d.]. *Amazon Redshift | Redshift ML - Amazon Web Services.* https://aws.amazon.com/redshift/features/redshift-ml/

[3] [n.d.]. *Amazon SageMaker Feature Store for machine learning (ML) – Amazon Web Services.* https://aws.amazon.com/sagemaker/feature-store/

[4] [n.d.]. *Databricks Feature Store.* https://www.databricks.com/product/feature-store

[5] [n.d.]. *Feast: Feature Store for Machine Learning.* https://feast.dev/

[6] [n.d.]. *feathr-ai/feathr: Feathr – A scalable, unified data and AI engineering platform for enterprise.* https://github.com/feathr-ai/feathr

[7] [n.d.]. *Heart Disease Dataset.* https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset

[8] [n.d.]. *Multi-query optimization-Apache Mail Archives.* https://lists.apache.org/thread/mcdqwrtpx0os54t2nn9vtk17spkp5o5k

[9] [n.d.]. *Power Every Experience with AI.* https://www.tecton.ai/

[10] [n.d.]. *Snowflake Feature Store | Snowflake Documentation.* https://docs.snowflake.com/en/developer-guide/snowflake-ml/feature-store/overview

[11] [n.d.]. *Vertex AI with Gemini 1.5 Pro and Gemini 1.5 Flash.* https://cloud.google.com/vertex-ai

[12] [n.d.]. *The world of PostgreSQL wire compatibility.* https://datastation.multiprocess.io/blog/2022-02-08-the-world-of-postgresql-wire-compatibility.html

[13] 2024. *Apache MADlib: Scalable In-database Machine Learning.* https://madlib.apache.org/ Accessed: 2024-11-26.

[14] 2024. *PostgresML: Machine Learning in PostgreSQL.* https://www.postgresml.org/ Accessed: 2024-11-26.

[15] William Steinbrunn Andras Janosi. [n.d.]. Heart Disease. https://doi.org/10.24432/C52P4X

[16] Panos Kalnis and Dimitris Papadias. 2003. Multi-query optimization for on-line analytical processing. *Inf. Syst.* 28, 5 (July 2003), 457–473. https://doi.org/10.1016/S0306-4379(02)00026-1

[17] Rui Liu, Kwanghyun Park, Fotis Psallidas, Xiaoyong Zhu, Jinghui Mo, Rathijit Sen, Matteo Interlandi, Konstantinos Karanasos, Yuanyuan Tian, and Jesús Camacho-Rodríguez. 2023. Optimizing Data Pipelines for Machine Learning in Feature Stores. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4230–4239. https://doi.org/10.14778/3625054.3625060

[18] Weixi Ma, Siyu Wang, Arnaud Venet, Junhua Gu, Subbu Subramanian, Rocky Liu, Yafei Yang, and Daniel P. Friedman. 2024. F3: A Compiler for Feature Engineering. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture* (Milan, Italy) *(FUNARCH 2024)*. Association for Computing Machinery, New York, NY, USA, 3–9. https://doi.org/10.1145/3677998.3678220

[19] Microsoft 2024. *Microsoft SQL Server Machine Learning Services.* Microsoft. https://learn.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services Accessed: 2024-11-26.

[20] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.* 29, 2 (May 2000), 249–260. https://doi.org/10.1145/335191.335419

[21] Prasan Roy and S. Sudarshan. 2009. *Multi-Query Optimization.* Springer US, Boston, MA, 1849–1852. https://doi.org/10.1007/978-0-387-39940-9_239

[22] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[23] Mugdha Somani. [n.d.]. Top 3 Feature Stores To Ease Feature Management in Machine Learning. https://censius.ai/blogs/top-3-feature-stores-to-ease-feature-management-in-machine-learning

[24] Yicheng Tu, Mehrad Eslami, Zichen Xu, and Hadi Charkhgard. 2022. Multi-Query Optimization Revisited: A Full-Query Algebraic Method. In *2022 IEEE International Conference on Big Data (Big Data).* 252–261. https://doi.org/10.1109/BigData55660.2022.10020338

[25] Yi Wang, Yang Yang, Weiguo Zhu, Tao Gu, Yu Ji, Xiaoyu Zhu, Weichen Yang, and Yuan He. 2020. SQLFlow: A Bridge between SQL and Machine Learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* Association for Computing Machinery, New York, NY, USA, 1241–1255. https://doi.org/10.1145/3318464.3389773

A Heart Disease Source Data

Two base tables store the raw data.

A.1 heart_data table definition

```sql

CREATE TABLE rawdata.heart_data(
    patient_id              CHAR(10),
    record_dt               CHAR(100),
    chest_pain_type         CHAR(10),
    resting_blood_pressure  CHAR(10),
    serum_cholestoral       CHAR(10),
    fasting_blood_sugar     CHAR(10),
    resting_electoral_res   CHAR(10),
    max_heart_rate          CHAR(10),
    exercise_ind_angina     CHAR(10),
    oldpeak                 CHAR(10),
    slope_of_peak_exercise  CHAR(10),
    no_of_major_vessels     CHAR(10),
    thal                    CHAR(10),
    target                  CHAR(10)
);
```

A.2 patient_details table definition

```sql
CREATE TABLE rawdata.patient_details(
    patient_id              CHAR(10),
    age                     CHAR(10),
    sex                     CHAR(10)
);
```

B Heart Disease Feature Store

SQL-ML feature definitions used in the evaluations.

B.1 FEATURE STORE

```sql
CREATE FEATURESTORE health;
```

B.2 FEATURE DEFINITIONS

Definitions of features for cholesterol and heart rate in a patient.

```sql
CREATE FEATURE health.cholesterol_for_patient(
  patient_id, record_dt,
  cholesterol
) AS
SELECT
  patient_details.patient_id,
  TO_DATE(record_dt, 'YYYYMMDD') AS record_dt,
  (
    cast(serum_cholestoral as int)- min_chol
  )/ min_max_chol AS cholesterol
FROM
  rawdata.heart_data,
  rawdata.patient_details,
  (
    SELECT
      min_chol,
      max_chol - min_chol as min_max_chol
    FROM
      (
        SELECT
          min(
            cast(serum_cholestoral as int)
          ) min_chol,
          max(
            cast(serum_cholestoral as int)
          ) max_chol
        from
          rawdata.heart_data
      )
  ) AS min_max_cholestoral
WHERE
  heart_data.patient_id = patient_details.patient_id
      PARTITION BY record_dt BY DAY;
```

```sql
CREATE FEATURE health.heart_rate_for_patient(
  patient_id, record_dt,
  heart_rate
) AS
SELECT
  patient_details.patient_id,
  TO_DATE(record_dt, 'YYYYMMDD') AS record_dt,
  (
    cast(max_heart_rate as int) - min_hr
  )/ min_max_hr AS heart_rate
FROM
  rawdata.heart_data,
  rawdata.patient_details,
  (
    SELECT
      min_hr,
      max_hr - min_hr as min_max_hr
    FROM
      (
        SELECT
          min(
            cast(max_heart_rate as int)
          ) min_hr,
          max(
            cast(max_heart_rate as int)
          ) max_hr
        from
          rawdata.heart_data
      )
  ) AS min_max_heart_rate
WHERE
  heart_data.patient_id = patient_details.patient_id
      PARTITION BY record_dt BY DAY;
```

B.3 Prediction query using UDF

Prediction function used in a query for predicting heart disease of a patient.

```sql
CREATE FUNCTION predict_heart_disease(cholesterol real,
     heart_rate real) RETURNS integer AS $$
BEGIN
    if cholesterol > 0.9 and heart_rate > 0.8 THEN
        return 1;
    ELSE
        return 0;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Query to predict the heart disease using the above UDF

```sql
SELECT
  predict_heart_disease(cholesterol, heart_rate)
FROM
  health.heart_rate_for_patient hr
  inner join health.cholesterol_for_patient ch on hr.
      patient_id = ch.patient_id;
```

C Feast API

C.1 Entity

Feast API for entity creation.

```
class feast.entity.Entity(name: str, description: str,
  value_type: feast.value_type.ValueType,
  labels: Optional[MutableMapping[str, str]] = None)
```

This class represents a collection of entities and their associated metadata.

C.1.1 Properties

- **`created_timestamp`**: Retrieves the `created_timestamp` of the entity.
- **`description`**: Retrieves the description of the entity.
- **`labels`**: Retrieves the labels associated with the entity, represented as a dictionary of user-defined metadata.
- **`last_updated_timestamp`**: Retrieves the `last_updated_timestamp` of the entity.
- **`name`**: Retrieves the name of the entity.
- **`value_type`**: Retrieves the type of the entity.

C.1.2 Methods

- **`from_dict(entity_dict)`**: Creates an entity from a dictionary.
  - **Parameters:** `entity_dict` – A dictionary representing the entity.
  - **Returns:** An `EntityV2` object created from the dictionary.
- **`from_proto(entity_proto)`**: Creates an entity from its protobuf representation.
  - **Parameters:** `entity_proto` – A protobuf representation of the entity.
  - **Returns:** An `EntityV2` object created from the protobuf.
- **`from_yaml(yml)`**: Creates an entity from a YAML string or file path.
  - **Parameters:** `yml` – A YAML string or a file path containing the YAML data.
  - **Returns:** An `EntityV2` object created from the YAML data.
- **`is_valid()`**: Validates the entity's state locally. Raises an exception if the entity is invalid.
- **`to_dict()`**: Converts the entity to a dictionary.
  - **Returns:** A dictionary representation of the entity.
- **`to_proto()`**: Converts the entity to its protobuf representation.
  - **Returns:** An `EntityV2Proto` protobuf object.

- **to_spec_proto()**: Converts the entity to an `EntitySpecV2` protobuf representation, suitable for passing to Feast requests.
    - **Returns:** An `EntitySpecV2` protobuf object.
- **to_yaml()**: Converts the entity to a YAML string.
    - **Returns:** A YAML-formatted string representing the entity.

C.2 FileSource

Feast source API for file based sources.

```
class feast.data_source.FileSource(event_timestamp_column: str,
   file_format: feast.data_format.FileFormat,
   file_url: str,
   created_timestamp_column: Optional[str] = '',
   field_mapping: Optional[Dict[str, str]] = None,
   date_partition_column: Optional[str] = '')
```

C.2.1 *Description*

Represents a file-based data source for Feast.

C.2.2 *Properties*

- **file_options**: Retrieves the file options of this data source.

C.2.3 *Methods*

- **to_proto()**: Converts a `DataSourceProto` object to its protobuf representation.
    - **Returns:** A `feast.core.DataSource_pb2.DataSource` protobuf object.

C.3 Feature

Feast API for creating feature.

```
class feast.feature.Feature(name: str,
    dtype: feast.value_type.ValueType,
    labels: Optional[MutableMapping[str, str]] = None)
```

C.3.1 *Description*

Represents a feature field type in Feast.

C.3.2 *Properties*

- **dtype**: Getter for the data type of this field.
- **labels**: Getter for the labels associated with this field.
- **name**: Getter for the name of this field.

C.3.3 *Methods*

- **from_proto(feature_proto)**: Creates a `Feature` object from a `FeatureSpecV2` Protocol Buffer object.
    - **Parameters:**
        - `feature_proto` – A `FeatureSpecV2` Protocol Buffer object.
    - **Returns:** A `Feature` object.
- **to_proto()**: Converts the `Feature` object to its Protocol Buffer representation.
    - **Returns:** A `feast.core.Feature_pb2.FeatureSpecV2` Protocol Buffer object.

C.4 Feast Materialize

Materialize API for Feast

- **feast materialize**: Materializes given from_date and to_date