Efficient Multi-Stream Machine Learning on Apache Flink

Harsh

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2024

Abstract

Across numerous sectors, a pressing demand has risen for real-time machine learning model training, a pivotal requirement for swift decision-making. Extensive research has been conducted to address challenges such as concept drift, class imbalance, and other pertinent issues. A critical challenge within this domain is the effective joining of the pipeline with streams containing correlated data, which enriches the dataset with more correlations and thereby enables algorithms to learn better Multivariate Linear Regression models. However, the computational cost associated with join and aggregation operations presents a significant obstacle, rendering real-time learning difficult to achieve. This project seeks to optimize training times for incremental learning within data pipelines by utilizing the ideas of the Factorized incremental view maintenance (F-IVM) framework to function within distributed and multi-threaded environments provided by Apache Flink. By leveraging the factorization capabilities inherent in the F-IVM framework, the project aims to substantially reduce computational overhead, thereby advancing the feasibility of real-time machine learning.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Harsh*)

Acknowledgements

The most valuable lessons are learned through the process of making decisions, experimenting, and understanding where mistakes were made. It is only through recognizing what doesn't work that one becomes capable of discerning what does.

I am deeply grateful to my supervisor for his unwavering guidance and support, and also for providing me with the opportunity to work on a project with significant potential for exploration and real-world application that has enriched me with many lessons.

I would also like to express my heartfelt gratitude to my parents, grandparents, and sister for their continuous encouragement and inspiration, which have been instrumental in helping me achieve my goals.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The volume of data being collected today has long surpassed the capacity of humans to manually analyze it. Consequently, machine learning is increasingly employed to bridge the gap between the rate at which data is gathered and the speed at which it is analyzed. This shift underscores the necessity for real-time analytics, which enables swift responses to evolving patterns within the data. In most cases, data is sourced from multiple streams, often containing correlated data, which, when integrated, enrich the dataset and enhance the ability of machine learning algorithms to identify and learn from these correlations. This integration is typically achieved by executing SQL queries where streams function as relational tables subject to continuous, but small, updates.
Though originally designed for static databases, the prevailing approach in practice is to first generate the query result and subsequently run the machine learning training on it. However, this process introduces a costly export/import step, which significantly hampers real-time analytics. Recent advancements, such as Morpheus [23] and F[32], address this issue by factorizing computations and pushing the operations below arbitrary joins, effectively reducing the number of tuples involved in the joins and improving performance.
Given this expectation of real-time analytics, re-executing queries from scratch on updates is an untenable solution. Thus, to meet the demands for low latency and high throughput in model updates, incremental computation techniques are employed. Incremental View Maintenance (IVM)[13, 21] is one such technique, where queries are decomposed into smaller subqueries that generate partial results or views. Rather than re-computing from scratch, it focuses on updating only the affected views, thereby

enhancing efficiency. One such framework that leverages IVM is Factorised Incremental View Maintenance (F-IVM) [19]. It integrates both IVM and push-down techniques to achieve orders of magnitude faster performance compared to older methods.
However, these algorithms have been demonstrated only on single-core systems. Meanwhile, in the real world, the scale of data is so vast that it far exceeds the processing capabilities of single-core systems, necessitating the use of both vertical and horizontal scaling. Earlier attempts usually required complex implementations for queries or worked only on a window of data. In contrast, Apache Flink[10] offers a more advanced solution by supporting the execution of SQL queries over streaming data without the need for windowing, and it also supports Incremental View Maintenance. However, the techniques currently used in Flink for IVM are relatively primitive.
This project seeks to bridge this gap by porting concepts from the state-of-the-art F-IVM approach to a scalable, distributed platform like Apache Flink. By doing so, it aims to advance the field of real-time analytics and bring database processing into the realm of extreme computing.

1.2 Research Objective

This research investigates an innovative approach for deploying a real-time supervised machine learning algorithm on integrated data streams by borrowing concepts from the state-of-the-art Factorised Incremental View Maintenance (F-IVM) technique to produce the improvements. By the means of the following, the project aims to ascertain the benefits achieved by the integration of F-IVM with stream processing and parallel processing over the inbuilt IVM implementation of Flink and an approach having SQL query with quadratic aggregates:

1. Development of a scalable architecture for processing joined data streams in real-time by an efficient implementation of F-IVM.

2. Creation of a query plan by conversion of variable orders into View Trees and Delta Trees.

3. Enhance the computational efficiency of F-IVM by incorporating parallel processing capabilities, leveraging the multi-threaded execution model inherent in Apache Flink's architecture.

By doing the above, the project aims to explore impacts and answer the following:-

- **RQ1:** How does our pipeline, which uses FIVM to train a Multivariate Linear Regression model, compare to the pipeline that uses Flink's View Maintenance algorithm for the same purpose?
- **RQ2:** Does moving to multicore provide any meaningful improvement?
- **RQ3:** If successful, what are the limitations of multi-threading?

1.3 Project Overview

This project is composed of three key components. First, a pair of tools—exporter and importer for Variable Order, a concept analogous to logical query plans. These tools enable users to export variable orders into an Object file and subsequently load any persisted variable order. Second, the ViewTree constructor - optimizes the variable order to generate View Trees which are a hierarchy of Views depicting a physical plan for query evaluation. Third, the DeltaTree constructor - materializes the views in the ViewTree to handle updates efficiently. It is also the component that implements the generated plan and also integrates the pre-computations for linear model training.

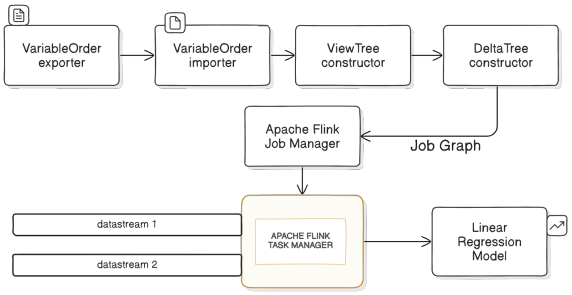

Figure 1.1: Overview

1.4 Overview of Contributions

F-IVM, originally designed for DBToaster, serves as a compiler that optimizes the tasks by generating specialized code. Thus, adapting F-IVM to stream-based processing systems like Apache Flink presented unique challenges due to differing constraints and coding paradigms. Significant effort was invested in tailoring F-IVM concepts to Flink. Firstly, the ViewTree algorithm was modified to support query merging to reduce the

number of costly GroupBy operations. Secondly, a custom operator (Section 4.2) was created which merged the group-by and join operations to significantly reduce memory consumption. However, this involved a significant effort to guarantee consistency, thus a concept of retraction was implemented to maintain the guarantee. Thirdly, as F-IVM maps each tuple to a payload containing intermediate computations that lead to potentially large cumulative sizes, thus techniques such as local-global context and delayed lifting (Section 4.3) were introduced. Lastly, a close competitor - F/SQL was also implemented for benchmarking.

1.5 Key Takeaways

The evaluation of F-IVM within Apache Flink underscores its clear advantages over traditional methods. F-IVM consistently outperforms both the Naive approach and F/SQL in processing speed and memory efficiency, making it a more effective solution for real-time data stream processing. This success is driven by its optimized use of ring payloads, variable ordering, and efficient state management.
In contrast, even though F/SQL benefits from SQL query optimizations, its performance is significantly hindered by the complexity of its numeric aggregation operations, leading to substantial delays in job graph creation and higher memory usage.
F-IVM also scales well with multi-core processors, though performance gains plateau beyond 32 cores. This limitation is largely dependent on the nature of the data rather than being specific to F-IVM. Additionally, we observed that stream-based solutions, including F-IVM, generally require considerably more memory, highlighting a common challenge in real-time data processing.

1.6 Dissertation Structure

The rest of the report is organized into four chapters: Chapter 2 provides essential background knowledge, reviews the field's history, and discusses the limitations of previous approaches. Chapter 3 details the integration of F-IVM concepts into Apache Flink, focusing on the core components. Chapter 4 covers intuitions behind calculations and optimizations. Chapter 5 evaluates the implementation's strengths and weaknesses through comparisons with other techniques. Finally, Chapter 6 concludes with a summary and recommendations for future work.

Chapter 2

Background

This chapter goes over the basic concepts in this field and explores different approaches, explaining how they connect to or differ from this work. This is later augmented with project-specific knowledge.

2.1 Preliminaries

This section provides foundational concepts and techniques essential for understanding the subsequent discussions. It begins by introducing techniques for storing the data in databases.

Star and Snowflake Schemas are two such ways of arranging the data that aim to reduce redundancy, optimize storage, and simplify tuple updates by means of data normalization [33].

Redundancy is common in databases, often manifesting as strongly correlated attributes. This redundancy not only takes up space but also makes updates more complex, requiring changes to all rows that share the same attribute. Normalization resolves this by retaining only the unique attributes and storing the rest in a separate dimension table, linked to the main fact table, exemplifying a star or a snowflake pattern. In turn, this method allows updates to be made in a single row only i.e. the dimension table.

The star schema is a straightforward method of organizing data, where the fact table is directly linked to the dimension tables. This single level of normalisation enhances query performance by reducing the number of joins. However, this efficiency comes at the cost of storage optimization, as dimension tables may contain redundant data.

In contrast, the snowflake schema further normalizes dimension tables into sub-dimension tables, reducing redundancy and improving storage efficiency. However, this hierarchi-

cal structure introduces additional complexity. To retrieve enriched tuples, dimension tables must be reconstructed from their sub-dimension tables before joining with the fact table. This extra step can lead to slower query performance due to the increased number of joins.

Once a choice is made and the database is established, the data inside it can then be utilized for many purposes. One such application is Machine Learning.

Machine Learning [7] is a field of study in Artificial Intelligence concerned with the use and development of algorithms that learn and improve on a performance metric on a task, as it gains more experience i.e. as it iterates through data/feedback. **Multivariable Linear Regression** being one of the most prominent examples. It is a technique of predicting the outcome of a dependent variable given two or more independent variables, under the assumption of low correlation of dependent variables with each other. The aim is to figure out some mathematical relation that the data contains, which is achieved by assigning coefficients to the independent variables and learning those coefficients. However, to be able to execute such tasks, first data is required to be extracted from databases by means of queries. These queries can be expensive and are usually optimized by using views.

A **View** is a table in a database that is created by combining data from other tables, known as base relations. Views can be materialized, which means their results are stored in the database instead of being recalculated every time they're needed. This allows for the creation of index structures on the materialized view, making data access much faster. However, when the underlying tables are modified—through deletions, insertions, or updates—the content of the view can be affected. Recalculating the entire view from scratch can be inefficient, so a more practical approach is based on the principle of inertia. This method assumes that only a small part of the view changes when the base tables are updated, making it cheaper to compute just these changes. Algorithms that focus on updating only the affected parts of a view in response to changes in the base tables are known as **Incremental View Maintenance algorithms.**

One example of such an algorithm is F-IVM which under the hood uses the concept of rings.

Rings [3] are algebraic structures which are an abelian group of sets each of which is accompanied by 4 components - two binary operations $+$ & $*$ that are analogous to addition and multiplication operations on numbers, an additive identity **0**, a multiplicative identity **1**; satisfying the following axioms for a set R s.t. $a, b, c \in R$:

1. commutativity: a + b = b + a

2. associativity: (a + b) + c = a + (b + c) and $(a*b)*c = a*(b*c)$

3. identity: $a+0 = 0+a = a$ and $a*1 = 1*a = a$

4. distributivity: $a*(b+c) = a*b+a*c$ and $(a+b)*c = a*c+b*c$

5. additive inverse: for each $a \in R, \exists -a \in R$ such that $a+(-a) = (-a)+a = 0$

2.2 Previous Works

This section covers the predecessors of F-IVM and how attempts are being made to train Machine Learning models by executing queries on streams of data.

Stream query processing has been a long ongoing journey. Originally, attempts were made to use traditional databases, but they suffer from high query latency despite having support for Incremental View Maintenance (IVM). The journey began with D. Carney et al.[11] creating a system that can process streams of data. Further attempts followed, notably by D. J. Abadi et al.[1] and S. R. Madden (tinyDB) [24], who created Stream Processing Engines to optimize the resource usage for processing. However, since their memory is limited to fixed window size, they are unable to manage long-lived data. Meanwhile, tinyDB doesn't even support ACID properties. These efforts are then succeeded by DBToaster[21], a compiler for SQL queries that delivered a significant performance boost, with improvements ranging from 3 to 6 orders of magnitude. However, just like others it also shared a common issue with scalability - all these engines are essentially restricted to a single core.

The data ingestion rates have already outpaced the capabilities of single-threaded computation, thus inducing the need for **Scalable Stream Query Processing**. Projects like MillWheel [2], S4 [26], and Heron [22] introduced multithreaded, multi-computer continuous query processing. Unfortunately, they offer only low-level processing which means the burden of expression and implementation of complex queries lies on the developers. Meanwhile works like Naiad[25] and Trill[12] do support them but use limited flat LINQ-style queries and additionally also support incremental computation of queries but leave its implementation to the developer. Spark Streaming [4] does support SQL queries but only simpler ones and just like tinyDB[24] etc, works only on windows of data. Apache Flink [10], a more modern framework, has support for more complex queries and unlike Spark streaming, which treats streams as micro-batches, it is a true streaming processing framework. Lastly, the latest development in this

area is SepJoin [37], which focuses on enhancing performance by optimizing the data placement strategy used by Flink, a direction that differs from our primary focus.
This ability of databases to store and process large-scale data has invoked a curiosity about their applicability in the field of Machine Learning. Originally, the journey began from static databases, with foundational contributions from [8] and [18]. These systems integrate statistical packages with the database, performing all joins and aggregations before passing the data to the statistical package. However, the computational cost of these joins, particularly cross-products, made real-time evaluations impractical [5]. To address this challenge, efforts like Morpheus [23] and LMFAO [31] were developed, focusing on factorizing and optimizing the computations of statistical algorithms by pushing them below the joins and aggregations. This approach significantly reduced unnecessary computations, such as cartesian products, and thereby decreased the overall computational complexity [5, 28]. Another related field of research is machine learning in streaming scenarios, as explored in works like [6]. This research direction addresses challenges such as concept drift [36] and resource management in streaming contexts, which is orthogonal to our work. Our focus is primarily on eliminating redundant computations that arise from joining and aggregating multiple data streams, and secondarily, on removing redundant computations within the core logic of a specific class of machine learning algorithms, namely linear regression.
The data is being collected at such a vast scale that databases are continuously under updations. Thus, simultaneously research efforts are being made for 3 decades to address the challenge of evolving databases through Incremental View Maintenance (IVM). IVM comprises a set of algorithms and techniques initially introduced in foundational papers such as [17, 9, 34, 13], collectively known as First-order IVM. Subsequent advancements led to the development of recursive IVM, exemplified by DBToaster [21], with these efforts culminating in the state-of-the-art approach represented by F-IVM[19]. Recursive IVM uses one materialization hierarchy per relation in the query, whereas, on the other side, FIVM uses one view tree for all relations. The former can thus have much larger space requirements and update times. A key advantage of F-IVM is its ability to share computations. When dealing with complex aggregations, F-IVM breaks down the process by pushing the aggregation past the join operations. Instead of recomputing everything from scratch at the parent level, intermediate computations are shared, improving efficiency. Unfortunately, other approaches lack this computation sharing and rely on numerous additional queries to maintain intermediate results. In constrast, First-order IVM does not take advantage of data factorization at all[38]. Addi-

tionally, while First-order IVM computes changes on the fly without using extra views, Recursive IVM tends to rely on too many additional views. Meanwhile, F-IVM strikes a balance by carefully selecting a minimal set of views for computation, optimizing both performance and resource usage. Hence, is considered to be superior.

2.3 FIVM

Factorised-Incremental View Maintenance (F-IVM) [19] is an approach for view maintenance designed to keep computed analytics in synchronisation with the changes in the underlying database. This section covers 3 fundamentals behind its superior performance and how they are realised.

2.3.1 Fundamentals

The performance of F-IVM has been attributed to 3 components, namely: Higher-order Incremental View Maintenance, which simplifies and accelerates complex query handling; Factorized Computation and Data Representation, which optimizes query algorithms; and Ring Abstraction, which offers flexible computation through diverse algebraic structures.

1. Higher-order Incremental View Maintenance: The first-order IVM algorithms are essentially simple and direct formulas designed to identify the adjustments required in the View in response to an incoming change. However, due to their simplicity, they are limited in how complex queries they can handle and in such cases are not efficient. Meanwhile, as a higher-order algorithm, F-IVM decomposes the task into a tree of views that progressively simplifies towards the leaf nodes. Consequently, the complexity of maintenance is reduced which can substantially speed up the task. This behaviour is shared by Fully recursive IVM algorithms as well. However, they decompose the task until only the most primitive views are left, while FIVM achieves a balance between the (storage and computational) overhead added by more but simpler views and complexity in updating fewer but complicated views. In practice, FIVM achieves two orders of magnitude better performance.

2. Factorized Computation and Data Representation: Instead of performing analytical computation on the result of the query, FIVM utilizes the query algorithms

([20], [5], [28]) that provide the best complexities such as pushing aggregates past join and factorized representations.

3. Ring abstraction: This is the major benefit of FIVM over others. It maps keys (tuples) to payload (elements of Ring). In key space, two tasks could be using the same join and aggregation operations however, in payload space different ring elements and operations can be selected which enables the same query to perform distinct tasks. For instance, on the same query, one Ring can be used to compute matrix multiplication where the Ring element is the matrix represented by the tuple, on the other hand, a degree-m based matrix Ring can be used to compute the covariance matrix. As a result, FIVM can be used to perform any analytical computation as long as it is attached with an appropriate sum and ring product operation.

Further, F-IVM also leverages distributed computation of aggregates. Often, queries compute the duplicate computations independently, thus wasting resources. However, F-IVM in such cases only computes the common sub-expression once and re-uses them whenever required.
These capabilities are effectively implemented through View Trees and Delta Trees, which will be further elaborated in the following sections.

2.3.2 Variable Orders, View Tree and Delta Tree

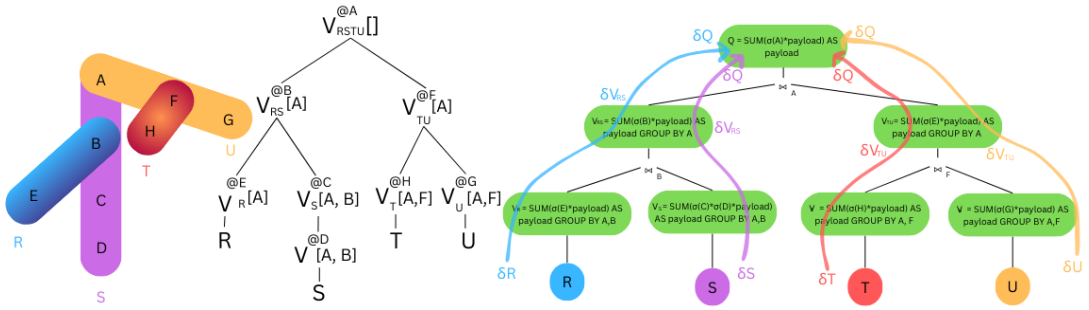

Figure 2.1: (Left) Variable order ω of the natural join of the relations R[B,E], S[A,B,C,D], T[E,G], U[E,F]; (middle) View Tree over ω; (right) Delta Tree where coloured lines represent update paths.

Traditional query evaluation relies on query plans that determine the sequence in which relations are joined. In contrast, variable orders [28], a concept similar to query plans, dictate the sequence in which variables should be marginalized instead of

focusing on relations first. As compared to the former, for every join variable, it joins all the relations sharing that variable, usually preferring to marginalize those variables first that are not involved in a join, as can be seen in Fig. 2.1. Further, this strategy has been shown to be worst-case optimal by [27] and hence, F-IVM employs variable orders instead.

Formally, a variable order ω is a pair of 1) a rooted forest, with one node per variable 2) a dependency function (dep) that identifies the ancestors sharing a relation with at least one variable in the subtree rooted at a given node. For instance in Fig. 2.1: $dep(A) = \emptyset; dep(B) = \{A\}; dep(C) = \{A,B\}; dep(D) = \{A,B,C,D\}; dep(E) = \{B\}; dep(F) = \{A\}; dep(G) = \{A,F\}; dep(H) = \{F\}$. Intuitively, these dependencies determine which variables must be preserved for use in ancestor nodes.

In the next step, F-IVM produces a View Tree which is a tree of queries (views) that are created to optimize the computation of the query. The View Tree utilize the information carried by variable order to describe the structure of marginalization and to decide which attribute computation should be pushed below joins. As can be seen Fig. 2.1 in the middle, it decides to create a View V_{RS} that marginalises variable 'B' while grouping by 'A', denoted by $V_{RS}^{@B}[A]$. To ensure that the query can directly access data from the tables, F-IVM employs an Extended Variable Order, which augments the leaves with input relations, enabling the View Tree to include queries for those relations.

Now, when a relation is updated, all views along the path from that relation to the root are affected. Thus, unlike traditional query evaluation, incremental maintenance necessitates the materialization of these queries (or views) to avoid recomputing the entire intermediate result. Consequently, F-IVM transforms the View Tree into a Delta Tree, where all affected nodes along the path are replaced by Delta views. These delta views support incremental maintenance by leveraging materialization, potentially reducing the update time for each view from quadratic to constant. In Fig. 2.1 the update paths depict that an update to one relation only affects the views along its update path and for the rest, materialised views are used. Along the path, each view is replaced by delta views, annotated by δ, which are small tables containing just the updates.

2.4 Batch Gradient Descent Using Covariance Matrix

This section explores the theory behind training a machine learning model in a single pass over a dataset using the concept of a covariance matrix which was originally presented by [32]. It also highlights key properties of the covariance matrix that are

utilized in this project to achieve tight integration of machine learning and streams. Consider a set of tables that, when joined, form a training dataset of size m:

$$\{(y^{(1)}, x_1^{(1)}, ..., x_n^{(1)}), ..., (y^{(m)}, x_1^{(m)}, ..., x_n^{(m)})\}$$

wherein, the value $y^{(i)}$ represents the target variable and $x_j^{(i)}$ represents the n feature/dependent variables.

The goal of a Linear Regression model is to learn the parameters $\theta = \{\theta_0, \theta_1 ... \theta_n\}$ of a linear function:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n$$

such that given some unseen values for $\{x_1 ... x_n\}$ it can predict the target variable $y = h_\theta(x)$.

Usually, to simplify and facilitate computation through vectorization, a constant feature, $x_0 = 1$ is introduced which converts the formula into $h_\theta(x) = \Sigma_{j=0}^{n} \theta_j x_j$.

To learn this linear function, the process begins with random initial values for the parameters, followed by iterative minimization of the loss, which measures the difference between the actual target values and the predicted values. There are several methods to compute this loss, with the most prominent being the Mean Squared Error (MSE):

$$J(\theta) = \frac{1}{2m} \Sigma_{i=1}^{m} (h_\theta(x^{(i)}) - y_i)^2$$

This learning process involves traversing the loss function space to locate its minima by consistently following the gradients at the current position on the surface. This approach is known as gradient descent (GD):

$$\forall 0 \leq l \leq n : \theta_j := \theta_l - \alpha \frac{\delta}{\delta \theta_l} J(\theta) \qquad \{where\ \alpha = learningrate\}$$

$$:= \theta_l - \alpha \sum_{i=1}^{m} (\sum_{j=0}^{n} \theta_j x_j^{(i)} - y_i) x_l^{(i)}$$

It is of two main varieties, namely batch GD and stochastic GD. While the former updates the parameter only after iterating through all the data, the latter updates the parameters after every few examples, thus instead of updating once, updates happen multiple times when iterating through the dataset. However, both do iterate over the entire dataset multiple times.

Gradient descent can be conceptualized as consisting of two main components: (1) the computation of sum aggregates, and (2) the convergence of the parameters. As a result, these processes can be executed independently. But first further simplification can be

made by considering target variable y as the $n+1^{th}$ feature and assigning a constant parameter $\theta_{n+1} = -1$ to it:

$$\forall 0 \leq l \leq n : \theta_j := \theta_l - \alpha \sum_{i=1}^{m} \sum_{j=0}^{1+n} \theta_j x_j^{(i)} x_l^{(i)}$$

$$:= \theta_l - \alpha \sum_{j=0}^{1+n} \theta_j \sum_{i=1}^{m} x_j^{(i)} x_l^{(i)} \qquad \text{[as } \theta_j \text{ is independent of i]}$$

$$:= \theta_l - \alpha \sum_{j=0}^{1+n} \theta_j \times Covariance[j,l]$$

As a result of this decomposition, instead of iterating over the entire dataset, we are now required to compute the covariance matrix only once. This covariance matrix accounts for the interactions $\text{SUM}(x_j * x_l)$ of variables x_j and x_l with continuous domains. More precisely, it quantifies the degree of correlation for each pair of features (or feature and label) in the data. Thus, by using covariant matrix a $O(m(n+1))$ operation is converted to an $O((n+1)^2)$ where $n << m$.
Further, this covariance matrix also carries 3 desirable properties:-

1. covariance matrix is symmetric across primary diagonal:

$$Covariance[j,l] = Covariance[j,l] \quad \forall\, j,l$$

2. The computation for populating the covariance matrix commutes with the union of K partitions of the dataset:

$$Covariance = \sum_{k}^{K} Covariance_k$$

where $Covariance_k$ is the matrix for k^{th} partition

3. The computation also commutes with projection i.e. if a model needs to be trained only on a subset of features, say from j' to j'', then, new covariance matrix $Covariance'$ is just a sub-matrix from row j' to j'' and columns j' to j'':

$$Covariance' = Covariance[j' : j'', j' : j'']$$

These latter two properties are particularly advantageous for distributed processing. By partitioning streams based on certain attributes or the number of rows, the covariance computations can be parallelized across different machines. Subsequently, the results from these parallel computations can be combined to form the final covariance matrix.

2.5 Apache Flink and Continuous Queries

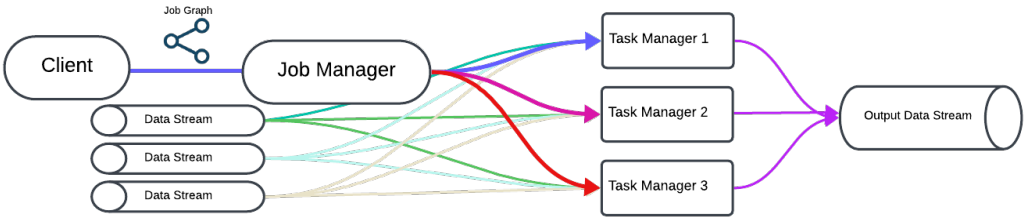

Figure 2.2: Flink components

Apache Flink is a distributed processing engine specifically designed to manage both bounded and unbounded data streams, with the expectation of near-real-time processing. An unbounded stream is characterized by the absence of a definitive endpoint, precluding the assumption that all requisite data has been received at any given moment. Consequently, Flink employs a processing model that promptly handles the incoming data and subsequently refines the output. In contrast, bounded streams, which have a finite endpoint, are processed only once all the data has arrived.

Flink also supports database-like operations, such as joins and aggregations, within a streaming context. However, unlike traditional databases, Flink treats tables as infinite sequences of tuples, known as Dynamic Tables, where each stream record is interpreted as an insert or modification. Consequently, queries in Flink are continuous, meaning the results are updated perpetually. To bridge the gap between SQL queries on streams and traditional database tables, Flink uses a concept analogous to Materialized Views and updates the results using an IVM algorithm called Eager View Maintenance. This capability, along with support for dynamic tables, makes Flink suitable for streaming applications.

Now, broadly, Flink's processing is managed by three main components (Fig. 2.2): the Job Manager, which distributes work and coordinates communication, and the Task Manager, which executes computations. Lastly, the Client, which submits the user-defined program, optimizes the Job Graph—representing the dataflow of sources, transformations, and sinks—and sends it to the Job Manager for execution which then transfers it to the Task Managers. They further then connect to data sources, subsequent tasks, or sinks to collectively execute the job.

Chapter 3

Fundamental Blocks

This chapter outlines the integration of F-IVM into Apache Flink, emphasizing the key components involved in its implementation. The chapter begins with an overview of the system architecture, highlighting the implementation choices made within the Flink platform, particularly the use of the DataStream API. It then delves into the role of Variable Orders in optimizing query plans and managing incremental updates. The View Tree structure is examined next, illustrating its role in organizing and maintaining a hierarchical query framework for efficient computations. Finally, the Delta Tree is discussed, emphasizing its function to compute covariance matrix by minimizing recomputation and propagating changes effectively. Each section provides a detailed analysis of these components and their integration within the system.

3.1 Architecture

Apache Flink offers two primary methods for defining processing pipelines: the Relational API and the DataStream API. The former provides an SQL-like interface, thus providing access to a lot of inbuilt operations such as join, group-by etc. Further, it also includes a Query planner which incorporates optimizations for query planning, all a user has to do is to just define the task. However, due to this declarative nature, a lot of low-level control has to be relinquished. The Datastream API is Flink's alternative means to attain complete control over the data processing but it comes at the cost of losing the Relational API's optimizations.

In this project, both approaches were explored. Since Flink employs classical query plans, while F-IVM utilizes Variable Order and additionally, also involves mapping of tuples to elements of a Ring, thus both APIs required substantial adaptation. However,

table API uses an inefficient combination of join and group-by operation (covered in Section 3) and didn't support the use of dual input operators which are essential for any custom join implementation. Further, it uses boxed Java Data types instead of primitive causing large memory use. Data types are extremely critical components considering the number of tuples. We originally worked with BigDecimal type, and after moving to the primitive *double* type we saw a 2/3rd decrease in memory consumption and 10 times faster performance. Considering the limitations the move to custom operator was necessary.
The system developed in this project comprises four main components to create a Job Graph: a Source, a Variable Order importer, a View Tree Generator, and a Delta Tree Generator. Additionally, it includes an operator for computing Linear Regression.

3.2 Source

Any Apache Flink project begins with data ingestion from a source. This project offers two approaches:

1. **JDBC Connection**: Flink's JDBC connector for the Table API allows for the creation of virtual views linked to a PostgreSQL [35] database, mirroring the schema of the underlying tables. Once created, these are then accessed by executing a "SELECT *" query on them.

2. **Postgres-CDC**: Unlike JDBC, which reads data only at query time, this method continuously monitors and captures ongoing changes by using Debezium [14] to read PostgreSQL's Write-Ahead Log (WAL).

Postgres-CDC requires an expensive initial database snapshot to read existing data protecting it from changes until ingestion, but since this project simulates streams from static datasets without updates, the JDBC option was chosen to avoid the overhead.
With data ingestion methods set, the next focus is on importing Variable Orders.

3.3 Variable Orders

As discussed in section 2.3, the variable orders are an alternative to classical query plans used by F-IVM due to their worst-case optimal time complexity. However, finding a good variable order is a hard problem, but since there exist systems like DB-Toaster

[21] that provide a robust implementation to identify the variable order, we are taking variable orders as an input to our system.

VariableOrder
Attributes
+ isRelation : boolean
+ variableName : String
+ relations : List<String>
+ children : List<VariableOrder>
+ shouldStopPushdown : boolean

Figure 3.1: UML for Variable Order Node

To facilitate this, two utilities have been provided - Variable Order Exporter and Importer. The former can be used to provide a variable order and it then exports the Java class hierarchy in a txt file. Later while creating the Job Graph, the importer can be used to import these txt files and pass it as an input to the View Tree creation algorithm. The class hierarchy is a tree of VariableOrder nodes containing the structure mentioned in Figure 3.1. If required, the generator component of other systems such as DBToaster can be utilized by extending the exporter.

3.4 View Tree

The View Tree Construction Algorithm [Appendix:1] is designed to recursively build a hierarchical structure known as the View Tree, using a Depth First Search approach. This View Tree captures the dependencies and relationships among variables within a given query plan. While the algorithm adheres to the core principles outlined in the F-IVM framework, it has been significantly adapted to integrate with our Variable Order and Delta Tree structures. These modifications not only ensure compatibility but also introduce several optimizations, enhancing the overall efficiency of the view maintenance process. It's recursion body is explained below and the visualisation of the produced tree is presented in Fig. 2.1.

1. Base Case - Leaf Node Construction: A leaf node represents a relation in the extended variable order, thus a ViewTree node is created with its reference.

2. Recursive Case - Non-Leaf Node Construction: When `VariableOrder w` has children, it indicates that the node could be a target for marginalisation, the algorithm then proceeds to create a `ViewTree` node but this time copies relevant metadata, including the variable name, relations involved, and any child nodes.

- **Child Node Processing:** Each node in the ViewTree maintains a mapping of ancestor variables and their associated relations. This algorithm creates such a map, copies the ancestor information received from its parent and adds the variable and relation represented by itself. The updated map is then passed to the child nodes, enabling them to track their ancestors. The algorithm recursively processes each child node to build their respective ViewTree nodes. Additionally, each node also tracks all marginalized variables from its descendants, which is further explained in Section 4.3.

3. Determining Free and Marginalized Variables: After processing all child nodes, the algorithm identifies the set of free variables for the current node—those needed for further computation and not marginalized by the query. This determination is crucial for the F-IVM algorithm as it impacts how views are incrementally maintained. At any node, these are those variables that are returned by the result of queries at children and are either required by some ancestor belonging to the same relation as a child (identified by the getDeps method) or are free at the root itself. All the variables that are not free are then marked for marginalization. Additionally, a special flag, `shouldStopPushdown`, is introduced to prevent unnecessary marginalization at this and child nodes, with detailed reasoning present in Section 4.6.

4. Tree Compression: To optimize the View Tree, the algorithm compresses the tree structure by merging nodes. If a node has a single child that is not a leaf node (i.e., does not represent a relation directly), the algorithm consolidates the child node into the current node. This compression reduces the tree's depth and simplifies its structure, improving the efficiency of the F-IVM process by minimizing the number of intermediate nodes which in turn reduces the nodes that are required to be maintained. As can been in Fig. 2.1, the view tree contains two nodes $V_S^{@C}[A,B]$ and $V_S^{@D}[A,B]$ which only perform marginalization, thus there's no need to maintain these two independently, hence are compressed to form the resulting ViewTree shown in Fig. 3.2

5. Returning the Constructed Node: Finally, after processing all child nodes, determining free and marginalized variables, and applying tree compression where applicable, the algorithm returns the constructed `ViewTree` node. This node represents the entire subtree rooted at the current `VariableOrder` node, containing all necessary information for efficient view maintenance.

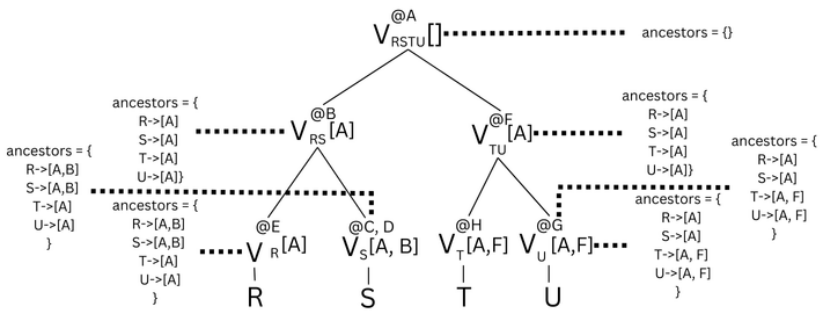

Figure 3.2: Compressed View Tree with ancestors map as annotation.

6. The `getDeps` Function: The `getDeps` function, used within the main construction algorithm, calculates the dependencies of a node by traversing the subtree rooted at that node. It identifies all relations in the subtree and uses the ancestors map to determine which variables the current node depends on. This function is crucial for ensuring that the View Tree accurately captures all dependencies necessary for correct incremental maintenance. For instance, in Fig. 3.2, $V_{RS}^{@B}[A]$ has relation R & S as its children, so it will look into its ancestor map to find A as its dependency.

Once the view tree is constructed it is then forwarded to the DeltaView constructor for the construction of the actual updation queries.

3.5 Updates and Delta Trees

A typical query processing lifecycle involves two key phases: evaluation and maintenance. The evaluation phase operates on a top-down, pull-based model, where the query is executed at the root node, which in turn recursively pulls data from child nodes. Conversely, the maintenance phase follows a bottom-up, push-based model, where updates originate at the leaves and propagate upward to their parent nodes. In the context of streaming data, streams are connected directly to the leaf nodes, and each incoming data packet is treated as an update. As a result, in a streaming environment, the evaluation phase is absent; every data packet is considered an update, with the first packet acting as an update to an initially empty table. Consequently, the ViewTree phase lacked any query creation and thus, only delta queries exist in this implementation. The following passage theoretically covers delta tree construction while the next subsection covers it practically.

While constructing a delta tree, a view node V representing a relation R under updates is simply replaced by a delta node δV representing the stream of update δR i.e. $\delta V = \delta R$.

In case, there are no updates to a view then its delta view is assigned to be empty i.e. $\delta R = \emptyset$. For others i.e. views defined using operators, [19] mentions the following formulas for deriving the delta view:

$$\begin{aligned} \delta(V_1 \uplus V_2) &= \delta V_1 \uplus \delta V_2 \\ \delta(V_1 \otimes V_2) &= (\delta V_1 \otimes V_2) \uplus (V_1 \otimes \delta V_2) \uplus (\delta V_1 \otimes \delta V_2) \\ \delta(\oplus_X) &= \oplus_X \delta V \end{aligned} \tag{3.1}$$

However, in Flink, an operator such as 'connect', which connects two streams together, only processes data from one side at a time. As a result, at any point in time, one of the two delta views will be null (say $\delta V_2 = \emptyset$). Thus, By using the identities $\emptyset \uplus V = V \uplus \emptyset = V$ and $\emptyset \otimes V = V \otimes \emptyset = \emptyset$, equations 3.1 gets simplified to:-

$$\delta(V_1 \uplus V_2) = \delta V_1 \text{ and } \delta(V_1 \otimes V_2) = \delta V_1 \otimes V_2 \text{ and } \delta(\oplus_X) = \oplus_X \delta V \tag{3.2}$$

This simplification reduces the operations to a straightforward application of the ring product and addition operations (Section 4.1) to delta views.

3.5.1 The Delta Tree Construction

This section covers the implementation of DeltaTree (Alg. 2) which is encapsulated in the 'DeltaTreeForStreams' class. The primary role of this class is to construct a delta tree that represents the hierarchical structure of a relational query plan. Each node in this tree can either correspond to a base relation or an intermediate result derived from joining multiple relations. The construction function is a recursive function that accepts 4 arguments - (1) tau: the node of the view tree for which a delta view needs to be created (2) relations: a map of relations names and the Table object that provides access to the respective table (3) attributeIdx: a map between attribute and the index that should be assigned to it in the covariance matrix (4) tEnv: the Flink's StreamTableEnvironment that facilitates the conversion between Table object and DataStream.
These arguments are then processed as follows:

1. Base Case- The construction process begins by checking whether a node directly corresponds to a relation. In this scenario, the node is initialized with the corresponding relation's stream (fetched from the 'relations' map), and converted into a DataStream using the 'StreamTableEnvironment'. This stream serves as the foundation upon which subsequent operations are built.

2. Recursive case- For non-base cases, where the node represents a more complex operation (such as a join or aggregation), the algorithm recursively constructs child nodes. These children represent the sub-queries or sub-relations involved in the operation. The recursive construction ensures that the tree accurately captures the hierarchical nature of the query, where each node's output may serve as input to its parent node.

3. Management of Free and Marginalized Variables - One of the key features of the algorithm is its ability to manage free and marginalized variables efficiently via the use of information provided by the ViewTree nodes. Marginalization ensures only the necessary data is propagated through the tree. This careful management reduces the computational overhead by minimizing the amount of data that needs to be maintained and processed at each step. At the same time, it ensures that all necessary variables are preserved until they are no longer needed, at which point they are eliminated. For instance, the delta tree in Fig. 2.1 contains marginalisation queries at each node.

4. Delta View Creation and join Operations- The creation of delta views, as implemented in the 'createDeltaView' method [Algorithm 2], is where the actual stream processing logic is applied. This method handles the task of joining multiple data streams, which may represent different parts of the query. The algorithm uses a custom implementation of Equi-join and Group-by operations [Section: 4.2] that merges the stream and the carried payload. By means of the ring operations of a covariance structure, these operations are performed in a manner such that not only the partial computations of both sides are preserved but the cross-correlation between the variables is also computed.

5. Novel Optimization: At each node, rather than creating a large covariance matrix that incorporates the knowledge about all the variables, we create a smaller covariance matrix that only has a local context. In order to do so, we create an 'attributeIdx' like HashMap that consists only of local mappings and also a HashMap that maps these new local indexes to the indexes of global(i.e. parent) context, provided by 'attributeIdx'. Section 4.3 explains it in more detail.

Once all the operations on the streams are configured for covariance computation, the final stream from the root delta node is attached to the linear regression model that learns the hyperparameter using concepts defined in Section 2.4.

Chapter 4

Optimisations

This chapter delves deeper into various concepts utilized to incrementally and optimally compute the covariance matrix. It begins by discussing which ring was picked and how it computes the matrix. Then, the discussion moves to the custom operator created to improve speed and reduce memory consumption. Afterwards, the focus shifts to how payloads themselves were compressed to consume significantly less space. This is followed by an intuition to minimise the number of intermediate tuples by means of a new query plan strategy. Later, it moves on to discuss how redundant computations were removed and finally, how multithreading works in this project.

4.1 Covariance Computation With Sharing

The process of computing any task in F-IVM begins by selecting a Ring and mapping all the tuples (called keys) to the elements of that Ring (called payload). Later, queries are defined using join and aggregation functions. These aggregations are computations on the payloads by means of Ring operations and primitive SQL operations such as projections on keys. This project trains Linear Regression models through a covariance matrix which is computed by using a covariance structure of degree-m over Real numbers R [19].

This Ring of covariance structure is a set where elements are a triple of $(R, R^m, R^{m\times m})$ where R^m represents a vector of size m and $R^{m\times m}$ represents a $m \times m$ matrix of real numbers. It carries the additive identity $\mathbf{0} = (0, 0^m, 0^{m\times m})$, where the first index carries the real number zero, 2nd and 3rd indexes are a vector and a matrix of size m and $m \times m$, respectively, of the real number 0. Further, it carries the multiplicative identity $\mathbf{1} = (1, 0^m, 0^{m\times m})$ where 1st index instead carries the real number 1. Finally, the sum

(denoted as $+^{ring}$) and product (denoted as $*^{ring}$) operations between two elements $a = (c_a, s_a, Q_a)$ and $b = (c_b, s_b, Q_b)$ are defined as:

$$\begin{aligned} a +^{ring} b &= (c_a + c_b, s_a + s_b, q_a + q_b) \\ a *^{ring} b &= (c_a * c_b, c_b * s_a + c_a * s_b, c_b * Q_a + c_a * Q_b + s_a * s_b^T + s_b * s_a^T) \end{aligned} \tag{4.1}$$

Here, $+$ and $*$ represent scalar addition, element-wise matrix addition, scalar multiplication, matrix multiplication and cross-product operations on real numbers, vectors and matrices accordingly.

The first element of this triple is a scaler which represents the number of rows, the second is called linear aggregate where the i^{th} element represents the sum of i^{th} attribute across all tuples and finally, the last one is the quadratic aggregate which is the covariance matrix itself and contains the correlation between all pairs of the attributes.

Now, the second requirement i.e. SQL query to compute covariances for n+1 variable across t tables becomes:

```
SELECT +^c (g_0(bias) *^ring g_1(X_1) *^ring ... * g_n(X_n)) FROM table_1 NATURAL
    JOIN table_2 ..... NATURAL JOIN table_t;
```

where bias will have a value of 1 (real number) for all the tuples. Further, g_i represents a lifting function. The role of this lifting function is to convert the given value from the domain of the dataset into the domain of our Ring. It is defined as $g_i(x) = (1, s_i = x, Q_i = x * x)$; where $s_i = x$ represents setting value x at i^{th} index of 0^m and $Q_i = x * x$ implies setting x^2 at ith column of ith row in matrix $0^{m \times m}$.

4.1.1 Intuition

This section intuitively explains how the above two elements - Ring and the Query, enable training of the Machine Learning model.

The application of the lifting operation can be seen as computing the covariance over a table containing just that one tuple have one attribute carrying value x. Hence the first entry in the tuple is 1. Now, the sum of the i^{th} attribute would be x itself and since no other attribute exists, thus their sums are registered as 0. Finally, the last entry represents the correlation of i^{th} feature with itself thus, x^2 is registered at index (i, i). Further, since no other variable exists in the context of this marginalization, the correlation with other features is marked with 0. These values are then passed to Ring operations.

First, the sum operation can be seen as utilizing the distributive property of the covariance matrix (Section 2.4) to merge the covariance matrices of all the tuples in the group.

Meanwhile, the count and sum of the table resulting from the union of two tables would simply be the sum of counts and the sum of the sum of both tables.
Second, the product operation consists of three components:-

1. **Scalar multiplication** $c_i * Q_j$: It is obvious that a tuple belonging to the second table is present in the result of the equi-join operation as many times as the number of tuples in the first table having the same value for the join attribute (say c_a). Thus, if a tuple is present c_b times in the second table. In the join result, the covariance for two attributes x_i & x_j would be $\sum_i^{c_a * c_b} x_j x_l$. Now, we know an entry in the quadratic aggregate represents the product of value for two attributes j & l summed across c_b rows i.e. $Q[j,l] = \sum_{i=1}^{c_b} x_j x_l$. Thus, the duplicates created as a result of join can be integrated by using the count from the left table i.e. $c_a * \sum_{i=1}^{c_b} x_i x_j = \sum_{i=1}^{c_a * c_b} x_i x_j$ which is the same as before.

2. **Outer product** $s_i * s_j^T$: The above only corrected existing covariances but didn't calculate the covariance between variables from the left and right tables. Thus, the outer product is utilized to introduce these cross-correlations. The intuition can be gained mathematically by writing linear aggregates as $s_a[j] = \sum_{i=1}^{c_a} x_j = c_a * x_j$ and $s_b[l] = \sum_{i=1}^{c_b} x_l = c_b * x_l$. This means that their product in outer-product would be $c_a * x_j * c_b * x_l = c_a * c_b * x_j * x_l$ = (number tuples after joining $* x_j * x_l$), which is the formula for covariance.

3. The **element-wise sum** of all the components: Scalar multiples $c_a Q_b$ and $c_b Q_a$ only introduce covariants from their own tables but lack correlation across the tables. Meanwhile, outer products $s_a s_b^T$ and $s_b s_a^T$ introduce correlation across the tables but lack the former. Thus, the distributive property of the covariance matrix is again utilized to merge them together to create a matrix containing all.

Once generated, this matrix is then used to train the Linear Regression model as explained in Section 2.4.

4.2 Custom GROUP BY and JOIN Operation

Joins are already costly in databases, but in streaming environments, they become even more expensive. Since streams are continuous, it's impossible to predict when—or if—the counterpart tuple from stream 2 will appear to match a tuple in stream 1. For instance, if tuple A with value X1 is at the front of stream 1, and stream 2 has a different

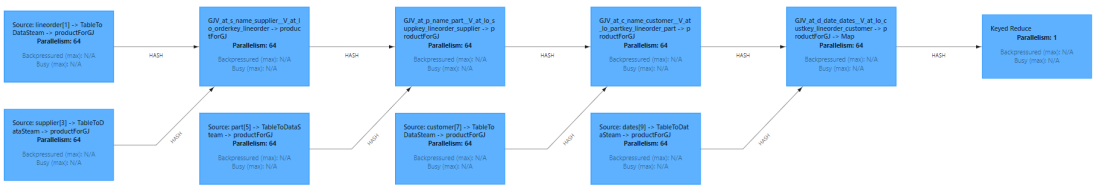

Figure 4.1: Apache Flink Job Graph

value at the front, we can't be certain that no counterpart for A exists. Tuple B, matching X1, might appear later. Additionally, even if tuple B is present, another tuple with X1 might arrive in the future, making it a candidate for joining. Therefore, tuple A must be stored in memory to ensure it's available for matching, and the same applies to tuples in stream 2. This necessitates holding both streams entirely in memory, leading to significant memory demands. Similarly, the Group By operation must store all groups in memory since it's uncertain whether all tuples in a group have been received, further increasing the cost.

On examining the View/Delta Tree creation (e.g. Figure 4.1), it's clear that a join operation is always preceded by a group-by operation. As noted, a Group By stores its output, while a join stores its input, leading to redundant data storage since the output of one is the input for the other. This redundancy burdens the memory. So, to resolve this, a custom implementation was developed to merge these operations, eliminating duplication and enhancing efficiency by avoiding the transfer of tuples between the two, thus reducing distribution, serialization, and deserialization overhead.

Further improvements were achieved by leveraging specific knowledge about the queries, which consisted solely of equi-joins. First, a hashmap was used to partition tuples by the join key, allowing for O(1) lookup operations. Second, the aggregation process was then divided into two parts. First, a partial marginalization operator called the "compression operator" [Algorithm 5], was introduced. This operator lifts and projects out any variables requiring marginalization before the group-by operation and applies the ring product to merge all existing covariance structures, leading to significant savings in transmission costs. Second, a leaner group-by operator was created, focusing solely on grouping and using the ring sum to merge structures across all its tuples.

However, there exists one latent problem with the streaming scenario. Due to the uncertainty about the future, the group-by operation cannot wait for all tuples to be collected and thus immediately outputs intermediate aggregation results. Consequently, all subsequent aggregations, joins, and operations are based on these interim results. When the group-by operation receives a new tuple for an existing group, it must update the result,

transmit the update, and notify downstream operators to revert the previous effect. This is achieved by tagging the tuples as either UPDATE_BEFORE or UPDATE_AFTER. The former maps the current tuple to a negative payload (element-wise multiplication with -1) to retract the side effect while the latter carries the new updated value. This algorithm [Algorithm 3] is an follows:-

The operation starts by initializing key variables, including the join attribute and grouping attributes (free variables). To avoid conflicts, different names are assigned to the payload depending on whether the data is from the left or right side of the join. Then, the row is processed to generate a key for the HashMap, based on the group-by attributes if present; otherwise, the key is derived from all attributes except the payload.

Once the key is generated, the algorithm checks for existing tuples with the same values for free variables to retrieve old payloads for the current group. It then finds the counterpart for the join operation, which is always preceded by a KeyBy operation on the join attribute, ensuring that the state is partitioned accordingly. Flink is smart enough that it only provides access to the relevant partition of state. This means all the entries in the current state will be the counterpart for the input tuple, thus the algorithm doesn't need any special checks and so, it simply iterates over all the retrieved rows from the right table's state, joining them with the left side's tuple.

If the group-by operation is enabled and an existing mapped value is found, an intermediate output with RowKind set to UPDATE_BEFORE is generated to retract the previous state. This involves re-joining old tuples with counterparts from the other table, ensuring exact replication of tuples. Note that it can never be the case that the other side table has a new tuple which was never joined with the current table's outdated tuple. This is conjectured due to the fact that this said tuple can only be present in the other table's state if it got joined with all the tuples in this current table's state.

The algorithm then computes the payload using either an accumulator or a retractor, depending on whether the current row represents an insertion (INSERT or UPDATE_AFTER) or a retraction (DELETE or UPDATE_BEFORE). The latter is performed by leveraging the distributivity over the union property (Section 2.4) of the covariance matrix to perform element-wise subtraction of the new payload from the existing one. This updated state is then stored back in the current table's state, which then can be used for both - to find the computed value of the state and to maintain the current table for performing joins with newer tuples. Afterwards, the new row is joined with tuples of the other side to produce a new UPDATE_AFTER row which contains the new updated value. This new row is then transmitted.

In the case of join-only operation, the functionality is very different. No payload computation takes place during the join operation, thus in this case there exists no concept of old and new payload values. Hence, no special UPDATE_BEFORE request is created. The algorithm just performs the join operation, producing the final output row which is then emitted. This emitted row carries the same kind as the input. If it was a retraction message, then it behaves as if replicating whatever was emitted in past, but with a payload meant for retraction. It then goes on to delete such a tuple from its state as retraction represents erasing what has been done in the past. In other cases, the join operation behaves normally according to what has been discussed previously.

4.3 Condensed Payloads

As outlined in Section 2.4, the covariance matrix is an $n \times n$ matrix, but due to its symmetry, much of the data is redundant. While the matrix size increases quadratically with each added feature, the memory requirement grows cubically since every tuple stores this matrix. To mitigate this inefficiency, we implemented a flattened structure inspired by a C++-like style, storing only the lower triangular part of the covariance matrix. This reduces the storage requirement from n^2 to $\frac{n(n+1)}{2}$ space.
Further, Java stores 2D arrays as an array of pointers to 1D arrays, requiring two heap accesses (AALOAD and IASTORE) per index operation. While flattened arrays may seem to negate these benefits due to the arithmetic needed for indexing, by leveraging the pattern in our ring, the algorithms were written in a way so that only sequential access is performed as much as possible. Also enabling it to benefit from spatial locality i.e. a process where CPUs while fetching anything from the memory, fetch the nearby data as well. To measure the difference, a simple script was executed that simulated the number of operations of one of our experiments. It consisted of 40 million accesses to two arrays containing 2500 elements, one being 1D and the other being 2D. After the test, we recorded that 2D arrays were 100 times slower. Further, as the covariance structure is a Triple, thus wrapping it in a Tuple class would have also added a lot of memory overhead that is associated with all non-primitive data structures. To address this, we integrated constant and linear aggregates into our flattened array, structured as follows: the 0th index for the constant, the next n items for linear aggregates, and the remainder for quadratic aggregates stored in row-major order.
Another novel finding is that F-IVM always uses a constant payload size which implies that even if only a single variable has been marginalized, the key will be mapped to a

tuple containing a correlation of all the pairs. Since we haven't marginalized others, these correlations will be zero. As it's usually the case that the starting stages always contain the largest number of tuples, a lot of precious memory space is wasted on useless data. To address this, we introduced a local-global context, treating each subtree in a Delta Tree as an independent dataset. A subtree marginalizing 10 variables assumes only 10 exist for model training, utilizing a degree-10 covariance structure. As the flow progresses to the parent tree, which marginalizes 5 more variables, the covariance structure expands to degree-15. This approach reduces data usage by 4.5 times compared to using a degree-15 ring throughout, which is particularly beneficial in complex schemas like Star and Snowflake where the final join result schema is much larger than individual sub-dimension tables. The pseudocode for this algorithm is provided in Algorithms 1,2. In Algorithm 1, during the backtracking step of the top-down recursion (Lines 20-23), the number of variables marginalized across all descendant subtrees is captured, which is essential for adjusting the covariance structures as the process moves up the tree.
Algorithm 2 then maps each attribute to its index in the subtree's covariance structure (Lines 12-17). The parent node then instructs its child nodes via the localAttributeIdx map, guiding the organization of the covariance matrix. This ensures that variables marginalized at the parent node do not interfere with others, allowing accurate covariance computation with previously marginalized variables (Lines 22-34).

4.4 Computation on the Condensed Payload

When scaling up existing payloads to a larger covariance structure, the typical method involves creating an empty array for each payload and transferring all data into it. Additionally, whenever a variable is marginalized, a new payload must be created first. Creating another such temporary array solely to store the scaled-up version can lead to substantial memory spikes, increasing memory demands. To address this, the distributive property of the covariance matrix can be leveraged to directly integrate the scaled result into the final array, thereby conserving valuable memory resources.
Another notable issue is that the covariance matrix is physically a flattened array, whereas logically, it functions as a 2D matrix. To reconcile this discrepancy, it is necessary to map the indexes appropriately to the flattened structure. Thus, Ring operations are required to be modified to support this. However, since Ring Sum is used as an aggregate operation, it only works with the same dimensional data thus, only the ring product is modified. The approach is as follows:

For the multiplication of scalars c, the ring product operation is straightforward. However, for linear aggregates, two operations are involved. The first is a scalar multiplication with the aggregate, which can be performed on the fly and is trivial. The second involves the element-wise addition of the linear aggregates of both operands. However, before performing this addition, the indexes must be mapped to the correct location. This mapping can be achieved using the formula:

$$\text{destination\_idx} = \text{globalMapping}[\text{attribute number}] + 1$$

The value is incremented by one to account for the fact that the 0th index stores the scalar c for the structure, causing all elements to be shifted to the right by one.
In contrast, the product for quadratic aggregates is more complex. This computation can be divided into three components: 1) adding $c_a Q_b$ to the result, 2) computing $s_a s_b^T$, and 3) computing $s_b^T Q_a$; which are discussed in the next subsections.

4.4.1 Scalar Multiplication

For the $c_a Q_b$ term, the challenge arises from the logical 2D indexing. The global mapping identifies which row and column should correspond to which row and column in the result, but since the underlying structure is 1D, direct usage is not feasible. However, by utilizing the row-major order formula employed in C++, the destination index can be identified as follows:

$$\text{dest\_i}^{\text{th}}\text{\_row} = \text{globalMapping[row number]}$$

$$\text{dest\_j}^{\text{th}}\text{\_column} = \text{globalMapping[column number]}$$

$$\text{offset} = 1 + \text{len\_of\_linear\_aggregates}$$

$$\text{destination\_idx} = \text{offset} + \text{dest\_j}^{\text{th}}\text{\_column} + \text{dest\_i}^{\text{th}}\text{\_row} \times (\text{dest\_i}^{\text{th}}\text{\_row} + 1)/2$$

Here, the offset denotes the index from which the quadratic aggregates begin.

4.4.2 Reducing Redundancy in Cross-Product Calculation

Initially, s_a and s_b are n-dimensional vectors, and their cross-product yields an $n \times n$ matrix. The ring product operation involves two such cross-products. However, by exploiting the matrix properties $(A^T)^T = A$ and $(AB)^T = B^T A^T$, it can be seen that $s_a s_b^T = (s_b s_a^T)^T$. Thus, only one cross-product needs to be computed. Furthermore, since the cross-product involves only two elements at a time to produce a single element of the result, the computation can be performed on the fly, eliminating the need to create

a temporary $n \times n$ matrix. However, this introduces a complication with the reduced covariance structure. The linear aggregate s_b is of size m, where $m \subset n$, their outer product would be of incorrect length. Creating a temporary array to scale it to size n would add unnecessary computational costs. Therefore, an algorithm was devised to calculate the product without generating a temporary array. Since a matrix is being element-wise added with its transpose, the result will be a symmetric matrix. This makes the upper triangle along the primary diagonal, redundant.
Considering all of the above points only $s_b s_a^T$ is computed. When its transpose is taken, the upper triangle becomes the lower triangle. Now, when $s_b s_a^T$ and $(s_b s_a^T)^T$ are summed up, the lower triangle of the final result is the sum of the lower triangle of $s_b s_a^T$ and the mirror image of its upper triangle. As both halves function differently, the computation is divided into two parts - 1) computing the lower triangle, adding it to the result and 2) computing the upper triangle, swapping its index and adding in the result. During iteration, since s_b is smaller, the missing variable is assumed to be zero. Thus, in the cross-product, the respective rows contribute nothing to the computation and can be disregarded. This allows iteration over s_b using pointer i, while the product is computed with elements of s_a one at a time and added to the row given by the 'globalMap' (say k) instead of the ith index. Further, instead of iterating over s_a only until ith index, iteration will continue until globalMap[i] as in the final covariance matrix, kth attribute needs to have covariances with first k attributes. The pseudo-code for the algorithm can be referenced in Algorithm 4.

4.5 Delayed and Combined Lifting

One of the main issues with F-IVM is that during ingestion, it maps keys (tuples) to the payload **1** from the Ring, as discussed in Section 4.1. This payload, represented as $(1, 0^m, 0^{m \times m})$, essentially stores nothing but consumes memory on the order of $O(1 + m + m \times (m+1)/2)$. Initially, since no operations exist in the tables, the first operator stores the entire payload in its state, leading to significant memory usage. However, as this payload is the ring's multiplicative identity, when the first lifting variable merges with it via the ring product operation, the result is just the lifted value itself. This suggests that assigning such a payload is unnecessary. Consequently, the first marginalization operation was modified to not expect any existing payload, resulting in memory improvements on the order of $O(m^2 t)$ where t is number of tuples.
Further, in the original view tree algorithm, each view marginalizes one variable at a

time. However, with the compression optimization, a view now marginalizes multiple variables simultaneously. Thus, Lifting each variable individually can create large intermediate structures, which are later merged to form a single structure representing covariances. Now, each lifting operation can be seen as finding covariances of a dataset consisting of only a single feature. Similarly, all these variables can be taken together as one dataset and covariances can be computed afterwards. Thus, we alter the Ring product function to accept any number of arguments, if those are variables to be lifted then a count function is applied to find the constant of our covariance structures, 'sum' operation is applied, to each variable independently, to get an array of sums i.e. linear aggregates and lastly, the covariance matrix is calculated using the covariance formula $\text{Cov}(X,Y) = \frac{1}{n}\sum_{i=1}^{n} X_i * Y_i$ (with division at the time of linear regression operation).
In DBToaster's original F-IVM implementation, the recommended approach was to maintain a HashMap of keys and payloads. This project, however, incorporates the payload directly into the rows themselves. Since Flink creates new rows after every operation, adding the payload does not incur additional costs and avoids the overhead of accessing and storing the HashMap. This approach also aligns with Flink's multi-threaded nature, where operators cannot share states with one another.

4.6 A New Query Planning Strategy

A marginalization operation involves three key steps: lifting the variable to produce a Ring element, merging this element with the tuple's payload, and performing aggregation and grouping. Consider a scenario where a batch of n variables is marginalized in the initial operation. The new Ring element will then have a size of $O(1+n+n\times\frac{n+1}{2})$. Further, If the grouping operation produces g groups, instead of saving $O(gn)$ by removing attributes, the operation results in an increase of $O(g\times n\times(n+1)/2)$. Therefore, if memory usage is a concern, the following formula can be used at each marginalization step to assess its advantage:

$$\begin{aligned} cost_1 + cost_2 &< r_1 * (n_1 + f_1) + r_2 * (n_2 + f_2) \\ \text{where } cost_1 &= g_1 * (1 + n_1 + n_1 * (n_1 + 1)/2 + f_1), \\ cost_2 &= g_2 * (1 + n_2 + n_1 * (n_1 + 2)/2 + f_2) \end{aligned}$$

where for stream i: $g_i = |\text{groups}|$ $n_i = |\text{marginalized variables}|$ $f_i = |\text{free variables}|$ $r_i = |\text{rows}|$. The formula is based on the intuitive idea that after marginalization, each stream produces a tuple with a size equal to the sum of the covariance structure and

the remaining attributes. Now, since a join operation is involved, both outputs will have to be stored. Instead, if no group by operation is performed, both inputs will be stored. Thus, the size of states is compared in both scenarios, with the benefit essentially depending on how many groups the group-by operation produces. However, by not pushing down the aggregation, any underlying cartesian products can cause the result to become costly, thus it should also be a consideration. In our case, this is not an issue because we use equi-joins.
As can be seen in the UML of the variable orders, each node has a STOPPUSHDOWN flag. After analysis, the end user can set this flag to True, instructing the system not to perform the group-by operation on the children below.

4.7 Multi-Threading

Apache Flink is a framework designed to handle data streams at any scale, making it a distributed processing engine by nature. Flink executes each operator in a job graph on an independent Java thread, which enables parallelism and distribution across multiple machines. To be precise, rather than assigning a thread to each operator, it assigns a thread to each Task, where a task is a combination of operators that are executed sequentially. Often, a stateless operator succeeds a stateful one and since stateless transformations are usually lightweight, Flink merges two or more operators to increase throughput by reducing thread-to-thread handover and buffering. This project availed the benefits of this by dividing our custom operator (Section 4.2) into a compression operation and the main operator. The compress operation is defined using the stateless ‘MapFunction‘, which allows merging payloads and reducing transmission size. Further, by being stateless, it gets chained to previous operations providing more savings.
Flink also introduces parallelism by splitting operators into **operator subtasks**, allowing stream partitions to be processed independently. Operators like ‘KeyBy‘ partition a stream into sub-streams, each handled by different threads. However, there are cases where an operator should be divided into only a limited number of instances. This can be controlled by explicitly setting the parallelism level of the operator. In this project, operator parallelism was carefully configured for accuracy. For instance, the final custom group-by/join operator was set to a parallelism of one to ensure correct covariance calculations, with various levels tested to balance performance, as discussed in the next chapter.

Chapter 5

Evaluation

This section provides a thorough evaluation of the proposed methodology for integrating F-IVM strategies within Apache Flink. Its primary objective is to evaluate the effectiveness, performance, and scalability of F-IVM in real-time data stream processing through a series of experiments on Flink V1.19.1.

These experiments are performed on a machine with 64 threads on 3 datasets -

The **Housing** [32] dataset, which is structured in a Star Schema, consists of six relations: House, Shop, Institution, Restaurant, Demographics, and Transport, with a single join variable and 26 non-join variables. The results are computed using a variable order where each path includes variables from only one relation, except at the root, where all paths intersect on a join variable.

Next, the **TPC-H** [15] dataset, originally created for the industry-standard TPC-H benchmark, includes eight relations: Lineitem, Orders, Customer, Nation, Region, Partsupp, Part, and Supplier, organized in a Snowflake schema. However, given that the dataset initially had one-to-many relations with joins primarily on keys, it did not benefit from pushdown operations. Thus, to adapt this dataset for the experiments, it is first generated via dbgen [16] with a scale factor of 0.5 and later scaled up by duplicating the dimension tables. However, later the number of attributes were reduced from 53 to 37 to minimize computational requirements (refer to Section 5.1). Lastly, the variable order (Appendix A.2) was created according to intuition in 4.6, which resulted in a 10% reduction in the number of intermediate tuples and 20% in execution time.

Finally, **SSB** [29], a dataset based on TPC-H but adapted for Star Schema, consists of 5 tables - Customer, Dates, Lineorder, Part, Supplier. This was generated using ssb-dbgen [30] with a scale factor of 2. Similar to TPC-H, it faced the same challenges and was treated with the same duplication strategy and reduction of attributes from 54 to 37.

Here also, the variable order was chosen with a similar strategy as for TPC-H (Appendix A.2.2).
Lastly, two restrictions were applied to these experiments. First, the focus was solely directed towards covariance matrix computation, as the Linear Regression sub-pipeline is common across all methods and is usually fine-tuned according to specific requirements, such as placement on different machines capable of GPU-based computing. Additionally, since these datasets are synthetic, the learned parameters either way would not be representative of real-world scenarios. Second, these experiments were performed with Mini-Batching turned off, which is a feature that is used to remove redundant pairs of data (Section 6.1). Since it offers general improvements, Mini-Batching could be applied to any pipeline containing an aggregation operation, thus uniformly reducing the number of records across all 3 methods. Further, since F/SQL and F-IVM produce the same number of tuples, the effect will be the same for both. Consequently, the timing data is presented only for comparison purposes and is not indicative of the extact execution time. So, it should only be used as a measure of the ratio of improvement.
To answer the research questions, the experiments are divided into two categories - Comparison with other implementations and Multi-threading.

5.1 Comparison With Other Implementations

This section addresses RQ1 by comparing our implementation of F-IVM with both a Naive approach and F/SQL [32] across the three datasets. Alongside F-IVM, we also implemented F/SQL as a sub-project to serve as a benchmark for comparison. F/SQL, like F-IVM, leverages the symmetric and distributive properties of the covariance matrix to generate SQL queries with primitive aggregation functions, thus, benefiting from query optimization techniques such as pushdown operations. This makes F/SQL the closest competitor to the F-IVM approach. In contrast, the Naive implementation is very different as it computes the covariance matrix only after the join operation.

	Housing	**SSB**	**TPC-H**
Naive	15.34 s	35.26 min	28.97 min
F/SQL	34.24 s	91 min - 65%	24.22 min
F-IVM-Flink	7.16 s	6.57 min	2.26 min

Figure 5.1: Performance on three datasets

Table 5.1 presents a comparison of the Job completion time for three methods. As can be seen, F-IVM significantly outperforms the other methods. Surprisingly, F/SQL fared even worse than Naive implementation. This could be attributed to the

fact that F/SQL contain a quadratic number of aggregates which Flink then maintains in separate states, resulting in increased state access times. This, combined with the exponential number of records generated by each group-by operator update (Section 6.1), leads to slower computation and hence, lower throughput. While F-IVM also faces this issue, it incurs significantly lower computational overhead due to the use of ring payloads and other optimizations (Chapter 4), resulting in orders of magnitude higher throughput (Fig. 5.5). On the other side, the Naive approach also enjoys a significantly higher throughput, however since it needs to perform computations on the join results, the gained performance is negated by the large number of tuples needed to be processed. In contrast, F/SQL fairs better in TPC-H due to the lower number of tuples, which reduces intermediate tuples to 1/3rd.
It is also worth highlighting that, in the case of F/SQL, the Job Graph creation is a very expensive process, consuming up to 30 minutes (for 51 attributes) to even begin the task and sometimes failing altogether. Thus, the number of features was reduced to 37, even after which it took a few minutes. In contrast, both the Naive approach and F-IVM create their job graphs in seconds. This substantial overhead for F/SQL arises from the quadratic number of aggregates in each query, which exponentially increases the plan search space and complicates the job graph generation.

The trend observed in job completion time is also reflected in heap usage (Fig. 5.2). Although both F-IVM and F/SQL handle the same number of aggregate computations, F-IVM performs much better and is closer to the Naive implementation. Initially, both F/SQL and F-IVM shared a common issue discussed in Section 4.2, where the group-by and join operators stored their states independently, despite containing the same data. However, by introducing a custom join operator and storing the payload in a primitive format, Flink's memory usage decreased significantly. This problem of duplicacy is exacerbated by redundant tuples, leading to sudden memory peaks, which represent intermediate storage used during computation. Since F/SQL already consumes more memory, sudden peaks can cause out-of-memory errors. In comparison, the Naive implementation had the least memory usage as it does not suffer from this problem due to not involving aggregation operations. However, particularly in the case of the TPCH dataset, it consumed significantly more heap space than F/SQL itself. Unsurprisingly, the minima are still the least amongst all three due to the same reasoning as above. Meanwhile, the peaks shoots up due to the high throughput of the join operations that delay garbage cleaning of the intermediate covariance matrices. This issue is only visible in TPC-H dataset as it produces 211 million rows from a join of 5 million rows,

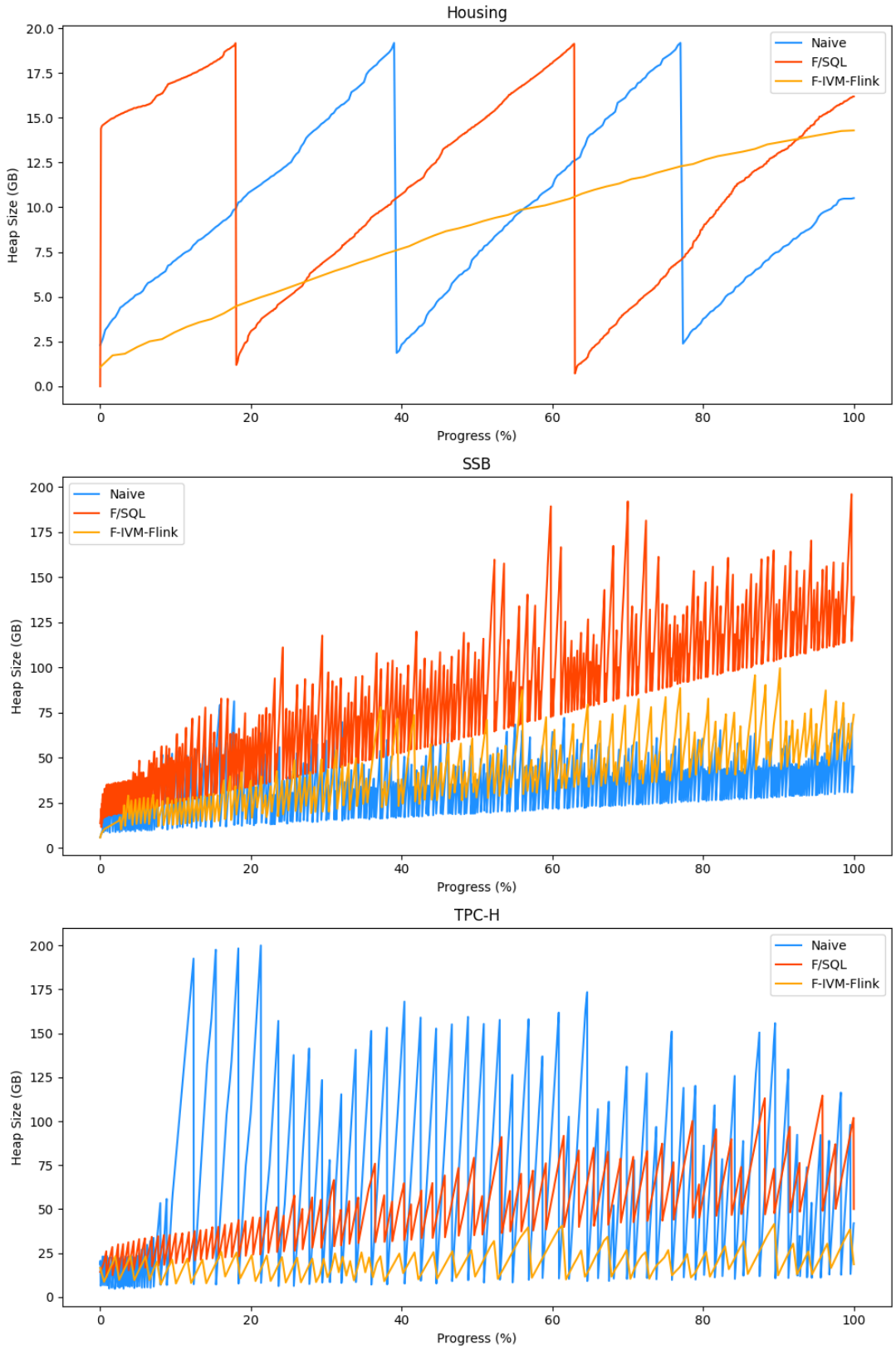

Figure 5.2: Heap Usage (GB) of the three algorithms for Housing, SSB and TPC-H Dataset

a factor 38.77 times while SSB scales by 14.76 times. Lastly, F-IVM depicted better improvements in TPC-H, as compared to other datasets, due to the use of variable order which produced a better query plan than Flink's own query planner [Appendix: A.2.3]. To summarise, F-IVM outperforms the other methods in terms of throughput and processing time, though it incurs a comparatively small memory cost but still significant compared to the Naive implementation. For instance, SSB originally is 2 GB in size while F-IVM consumes $\tilde{8}$0GB for computation. However, this is much better than storing the entire join result, which would have been >200GB.

5.2 Multi-Threading

This section addresses RQ2 and RQ3 by exploring the performance improvements from increasing the number of cores. The experiments were conducted using the *taskset* instruction to restrict CPU affinity. Additionally, since Apache Flink creates instances of an operator based on the number of task slots in the task managers, the number of task slots was matched to the number of cores.

	Housing	**SSB**	**TPC-H**
1	25.33 s	51.03 min	29.56 min
2	18.52 s	63.09 min	23.84 min
4	11.15 s	20.98 min	10.82 min
8	8.49 s	12.37 min	3.80 min
16	7.21 s	9.57 min	2.68 min
32	6.98 s	7.18 min	2.28 min
64	7.3 s	6.57 min	2.26 min

Figure 5.3: Performance with different parallelism

As apparent from Table 5.3, the implementation scales effectively from a single core version to 64 cores, creating an improvement of 3.6, 7.8, and 13.0 times for Housing, SSB & TPCH datasets (respectively). However, the performance gains appear to plateau when moving from 32 cores to 64 cores. Upon further investigation, it was found that this depends significantly on the data as well as the hardware.

By the nature of the distributed processing, there's always a requirement for a single-threaded aggregator that collates all the parallel partial computations, which in our case is the final group-by operator (Fig. 4.1). It had 100% utilization of the assigned CPU core but still was not able to keep up with the influx of tuples caused by the increase in parallelism of the previous operators. Thus, it started producing backpressure to slow down the input. Therefore signifying the importance of single-core performance.

To assess the increase in performance of the preceding operator due to an increase in parallelism without the bottleneck operators, experiments were conducted after re-

moving the last aggregation operator and with a parallelism of 32. This analysis, as shown in Figure 5.4, reveals that after the initialization phase, all the operators were not under 100% load, implying low usage even with 32 cores. Consequently explaining why a move towards 64 cores didn't yield improvements. Surprisingly, at first look the Housing dataset seems to disobey the trend. However, the resulting graph is only like such because the task duration was small enough that all operators finished during the initialization phase itself, which itself becomes longer due to the increase in parallelism. Finally, since Flink assigns one operator instance to one core and one or more groups to one operator instance, thus to optimally use multithreading, it is essential that the load is well-distributed across the operators i.e. all groups should have a similar number of tuples. In our case the standard deviation of load across all the operators is very low (Appendix: A.3) which reinforces that 32 cores is the final limit for these datasets.

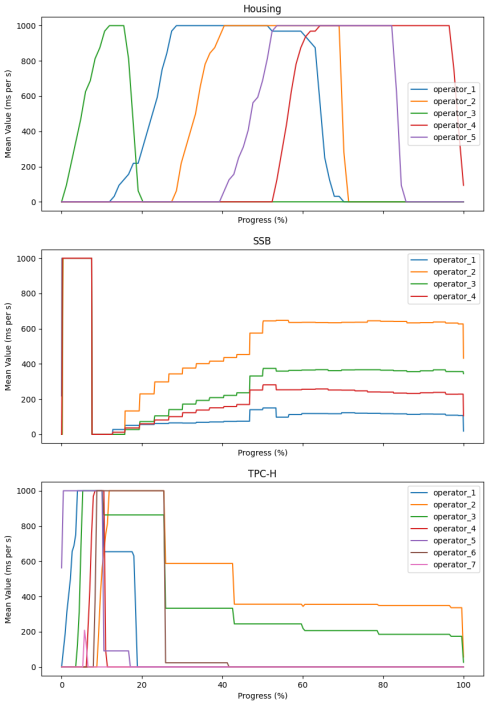

Figure 5.4: Mean busy times (ms per sec) for each operator

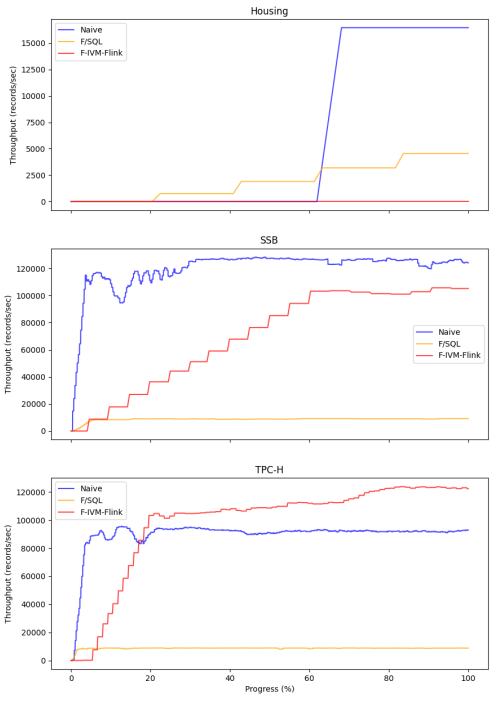

Figure 5.5: Throughput comparison across the 3 datasets

Chapter 6

Conclusion

This project was undertaken with the vision of reducing the overhead incurred on the stream processing side to train machine learning models using join operations. The focus was primarily on eliminating redundant computations that arise from joining and aggregating multiple data streams and secondarily, on removing redundant computations within the core logic of linear regression. The former was achieved by porting concepts of the Factorised-Incremental View Maintenance framework, while the latter was achieved by using covariance matrices to reduce the iterations over the data.
The project initially explored implementation using the Table-API and User-Defined Functions (UDFs). However, due to limitations such as high memory usage and less freedom in the implementation of UDFs, the project moved to the DataStream API, which allowed for greater flexibility. Herein, a custom join/group-by operation was created and many other optimizations such as reducing the payload size, sequential access, delaying and combining lifting operations etc were made. Throughout the project as minute things as the datatypes were carefully selected to reduce the overhead. This transition improved performance significantly but required manual implementation of many features, some of which had to be left for future work ex mini-batching.
The experimentation consisted of 3 datasets - Housing, TPC-H and SSB. All vary either in size, attributes or schema. These experiments depicted significant improvements in terms of memory pressure and execution time. Surprisingly, though Naive generally had lower constant memory required, in some cases its intermediate consumption peaked even above the rest of the two. The F-IVM-Flink had mostly limited peaks, while F/SQL had the highest minimas indicating that its intermediate results are larger in size (in case the Mini-Batching feature is disabled). Despite the significant improvements, it can be concluded that the streaming process inherently is a very expensive operation in

terms of memory requirement due to intermediate results being required to be stored at every join/group-by operation, consuming memory up to 20 times the dataset.
Lastly, in terms of scalability, the performance increased significantly creating up to 13x improvements. However, there exists a plateau at 32 cores, which on further investigation revealed that the system was already easily handling the data with 32 cores which obviously means increasing core count further will not be beneficial. It was also discussed how data needs to be well-balanced across groups to gain from increasing core counts. Further experimentation revealed a limitation around the last aggregation operation, where the results from all the pipelines are merged together. For now, this last operator can only be processed by a single thread and thus has an upper limit on how many tuples it can process.
Despite these limitations, the system demonstrated major improvements and is expected to continue improving once the enhancements discussed in the next section are implemented.

6.1 Future Work

Throughout the execution of this project, multiple areas of improvement were identified. Most prominent being mini-batching, more particularly the concept of **Folding**.
Folding is a technique that uses a buffer to detect and disregard UPDATE_AFTER requests that are followed by UPDATE_BEFORE requests. This situation often arises when an update to a group is quickly followed by another update to the same group. Since each update carries the full payload rather than just the deltas, the initial update request can be ignored, which saves a pair of UPDATE_BEFORE and UPDATE_AFTER, each of which further creates 2 more records, thus creating exponentially many requests. Hence, this implementation can save a significant amount of requests.
making it difficult to identify the necessary changes for the next operator. To address this, a retraction request is typically made to clear the state of the following operator. However, exploring the pattern of ring product operations could allow the system to send only delta-updates, eliminating the need for UPDATE_BEFORE.
Finally, as noted in the section 5.2, the last aggregation operation presents a significant bottleneck in Group-By implementations. In Flink, this operation essentially computes just the element-wise sum. A custom multithreaded operator could be developed to distribute certain partitions across different threads, though the challenge would lie in rejoining them. To manage this, order metadata would need to be appended.

Bibliography

[1] Abadi, D. J., Ahmad, Y., Balazinska, M., et al. : The Design of the borealis stream processing engine. In: CIDR, pp. 277–289 (2005).

[2] Akidau, T., Balikov, A., Bekiroglu, K., et al. : Millwheel: fault-tolerant stream processing at internet scale. PVLDB**. 6**(11), pp. 1033–1044 (2013).

[3] Anderson, M., Feil, T. : A first course in abstract algebra: rings, groups, and fields. Chapman and Hall/CRC (2005).

[4] Armbrust, M., Xin, R. S., Lian, C., et al. : Spark SQL: relational data processing in spark. In: SIGMOD, pp. 1383–1394 (2015).

[5] Bakibayev, N., Kočiský, T., Olteanu, D., Závodný, J. : Aggregation and ordering in factorised databases. PVLDB**. 6**(14), 1990–2001 (2013).

[6] Bifet, A., Gavalda, R., Holmes, G., Pfahringer, B. : Machine learning for data streams: with practical examples in moa. MIT press (2023).

[7] Bishop, C. M. : Pattern recognition and machine learning. Springer (2006).

[8] Brown, P. G. : Overview of SciDB: large scale array storage, processing and analysis. In: SIGMOID, pp. 963–968 (2010).

[9] Buneman, P., Clemons, E. K. : Efficiently monitoring relational databases. ACM Trans. Database Syst. **4**(3), pp. 368–382 (1979).

[10] Carbone, P., Katsifodimos, A., Ewen, S., et al. : Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. **38**(4), pp. 28–38 (2015).

[11] Carney, D., Çetintemel, U., Cherniack, M., et al. : Monitoring streams - a new class of data management applications. In: PVLDB, pp. 215–226 (2002).

[12] Chandramouli, B., Goldstein, J., Barnett, M., et al. : Trill: a high-performance incremental query processor for diverse analytics. In: PVLDB, pp. 401–412 (2014).

[13] Chirkova, R., Yang, J. : Materialized views. Found. & Trends in DB**.** **4**(4), pp. 295–405 (2012).

[14] Community, D.: Debezium. Accessed: 2024-08-15. 2024. URL: `https://debezium.io/`.

[15] Council, T. P. P.: TPC Benchmark™ H (TPC-H) Standard Specification, Revision 2.17.3. `http://www.tpc.org/tpch/`. Last accessed on 2024-08-26. 2021.

[16] Electrum, Tpc-h tools. `https://github.com/electrum/tpch-dbgen`. Accessed: 2024-08-23. 2024.

[17] Gupta, A., Mumick, I. S., Subrahmanian, V. S. : Maintaining views incrementally. In: SIGMOID, pp. 157–166 (1993).

[18] Hellerstein, J. M., Ré, C., Schoppmann, F., et al. : The MADlib analytics library: or mad skills, the sql. PVLDB**.** **5**(12), 1700–1711 (2012).

[19] Kara, A., Nikolic, M., Olteanu, D., Zhang, H. : F-IVM: analytics over relational databases under updates. VLDB J. **33**(4), pp. 903–929 (2023).

[20] Khamis, M. A., Ngo, H. Q., Rudra, A. : FAQ: questions asked frequently. In: PODS, pp. 13–28 (2016).

[21] Koch, C., Ahmad, Y., Kennedy, O., et al. : DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), pp. 253–278 (2014).

[22] Kulkarni, S., Bhagat, N., Fu, M., et al. : Twitter Heron: stream processing at scale. In: SIGMOID, pp. 239–250 (2015).

[23] Kumar, A., Naughton, J., Patel, J. M. : Learning generalized linear models over normalized data. In: SIGMOD, pp. 1969–1984 (2015).

[24] Madden, S. R., Franklin, M. J., Hellerstein, J. M., Hong, W. : TinyDB: an acquisitional query processing system for sensor networks. TODS**. 30**(1), 122–173 (2005).

[25] Murray, D. G., McSherry, F., Isaacs, R., et al. : Naiad: a timely dataflow system. In: SOSP, pp. 439–455 (2013).

[26] Neumeyer, L., Robbins, B., Nair, A., Kesari, A. : S4: distributed stream computing platform. In: ICDW, pp. 170–177 (2010).

[27] Ngo, H. Q., Ré, C., Rudra, A. : Skew strikes back: new developments in the theory of join algorithms. SIGMOD Rec. **42**(4), pp. 5–16 (2014).

[28] Olteanu, D., Závodný, J. : Size bounds for factorised representations of query results. TODS**. 40**(1), 2:1–2:44 (2015).

[29] O'Neil, P., O'Neil, E., Chen, X. : The Star Schema Benchmark and Augmented Fact Table Indexing. In: TPCTC, pp. 237–252 (2009).

[30] Rozenberg, E.: Ssb-dbgen: star schema benchmark data set generator. `https://github.com/eyalroz/ssb-dbgen.git`. Version 1.0. 2024.

[31] Schleich, M., Olteanu, D., Abo Khamis, M., Ngo, H. Q., Nguyen, X. : A layered aggregate engine for analytics workloads. In: SIGMOD, pp. 1642–1659 (2019).

[32] Schleich, M., Olteanu, D., Ciucanu, R. : Learning linear regression models over factorized joins. In: SIGMOID, pp. 3–18 (2016).

[33] Shin, S., Sanders, G. L. : Denormalization strategies for data retrieval from data warehouses. Decis. Support Syst. **42**(1), pp. 267–282 (2006).

[34] Shmueli, O., Itai, A. : Maintenance of views. In: SIGMOID, pp. 240–255 (1984).

[35] The PostgreSQL Global Development Group, Postgresql. Accessed: 2024-08-19. 2024. URL: https://www.postgresql.org/.

[36] Tsymbal, A. : The problem of concept drift: definitions and related work. SCSS**.** **106**(2), p. 58 (2004).

[37] Wang, Q., Zuo, D., Zhang, Z., Chen, S., Liu, T. : SepJoin: a distributed stream join system with low latency and high throughput. In: ICPADS, pp. 633–640 (2022).

[38] Zhao, W., Rusu, F., Dong, B., Wu, K., Nugent, P. : Incremental view maintenance over array data. In: SIGMOD, pp. 139–154 (2017).

Appendix A

First appendix

A.1 Specifications

This section covers the detailed specifications about the configuration of the machine and the dataset used to perform the experiments.

A.1.1 Machine

The project utilized the University of Edinburgh's DICE system to gain access to one of their compute machines. This machine consisted of 2 CPUs attached in NUMA configuration with 540GB of memory in total. Each of these CPUs had the following configuration:-

Specification	**Details**
Model	AMD EPYC 7302
Cores	16
Threads	32
Base Clock	3.0 GHz
Boost Clock	3.3 GHz
L3 Cache	128 MB
TDP	155 W
Process Technology	7 nm

Table A.1: Basic Specifications of the used CPU

A.1.2 Dataset

The project worked with 3 datasets, namely - Housing, TPC-H and SSB datasets. The following table provides further information.

Dataset	#Tuples	#Relations	#Join Vars	#Non-Join Vars
Housing	1.6M	6	1	26
SSB	192M	5	4	33
TPC-H	211M	8	6	31

Table A.2: Description about the 3 dataset used

A.2 Job Graphs

This section presents the final Job Graphs created by all three techniques for all three datasets. The Job graphs for F-IVM-Flink closely replicate the Variable orders. Thus, can be used to deduce the variable orders.

A.2.1 Housing dataset

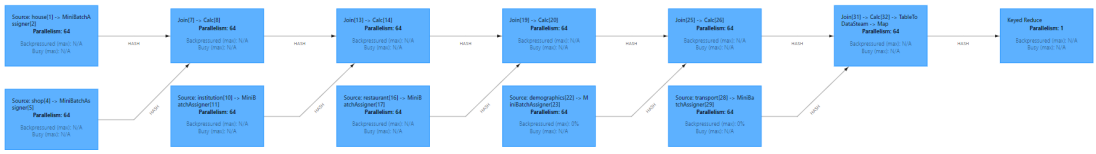

Figure A.1: Job Graph Created by the Naive algorithm for the Housing dataset

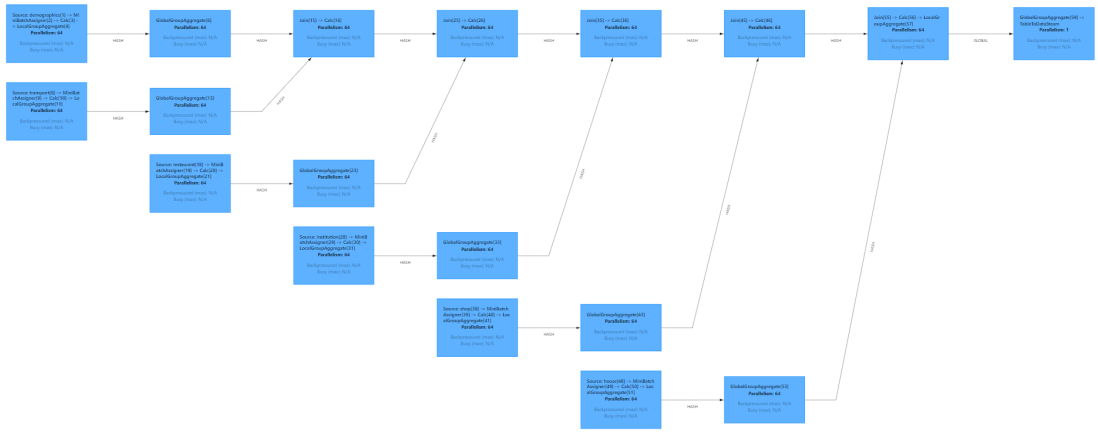

Figure A.2: Job Graph Created by the F/SQL algorithm for the Housing dataset

Figure A.3: Job Graph Created by the F-IVM-Flink algorithm for the Housing dataset

A.2.2 SSB dataset

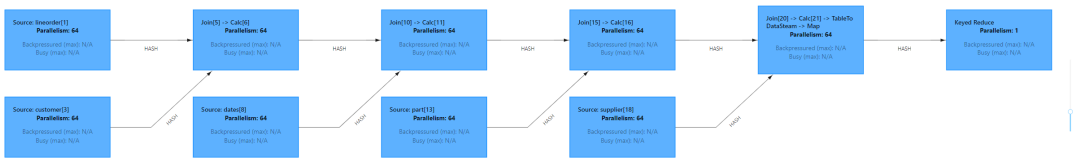

Figure A.4: Job Graph Created by the Naive algorithm for the SSB dataset

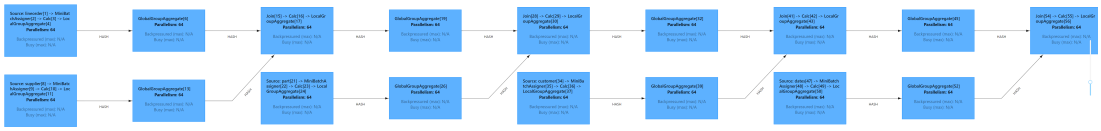

Figure A.5: Job Graph Created by the F/SQL algorithm for the SSB dataset

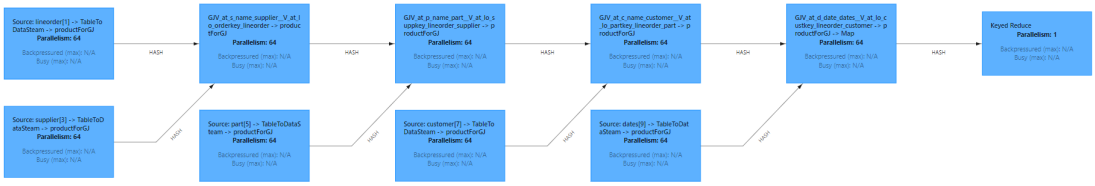

Figure A.6: Job Graph Created by the F-IVM-Flink algorithm for the SSB dataset

A.2.3 TPC-H dataset

Figure A.7: Job Graph Created by the Naive algorithm for the TPC-H dataset

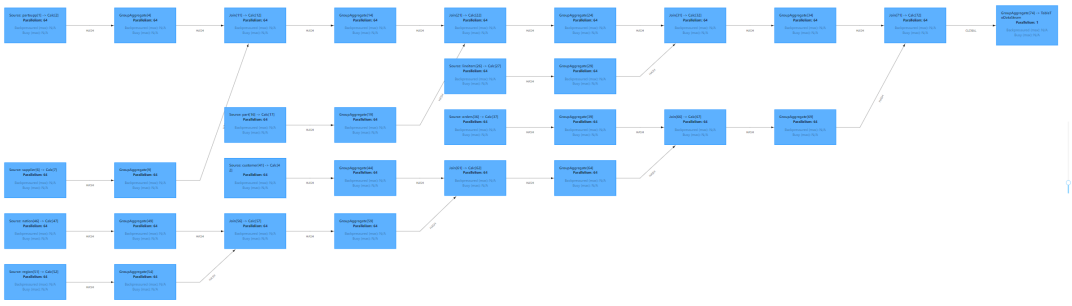

Figure A.8: Job Graph Created by the F/SQL algorithm for the TPC-H dataset

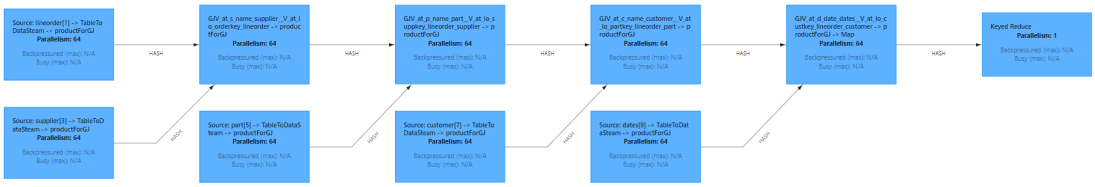

Figure A.9: Job Graph Created by the F-IVM-Flink algorithm for the TPC-H dataset

A.3 Standard Deviations

This section presents the standard deviation between the time that each sub-task of operators spends computing per second. As discussed earlier, all operators are not under full load except the Housing dataset, however, since it's a small dataset, it gets processed very close to the initial phase itself. Since the values are very low, this implies that the load is well-balanced within an operator.

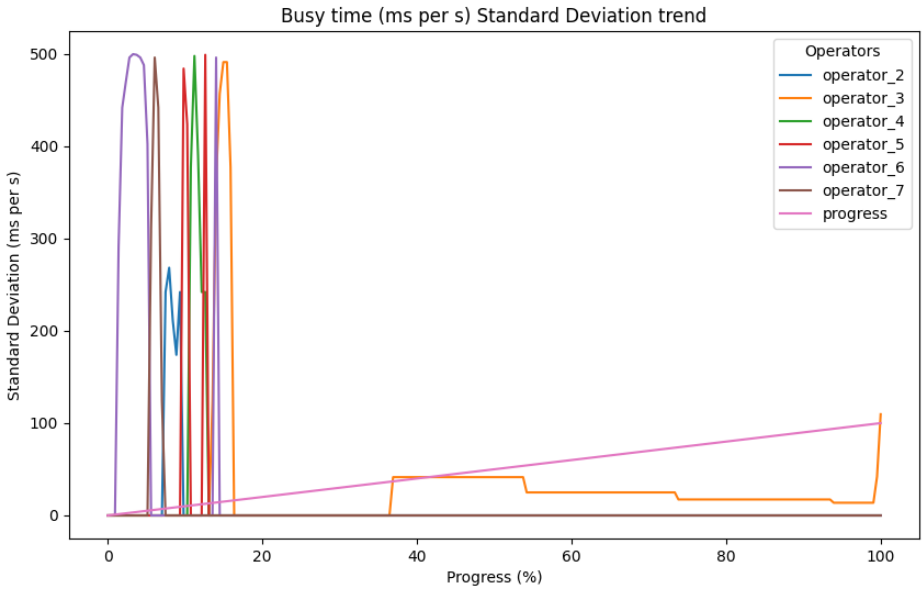

Figure A.10: Standard Deviation of all the operators for the Housing dataset with parallelism of 64

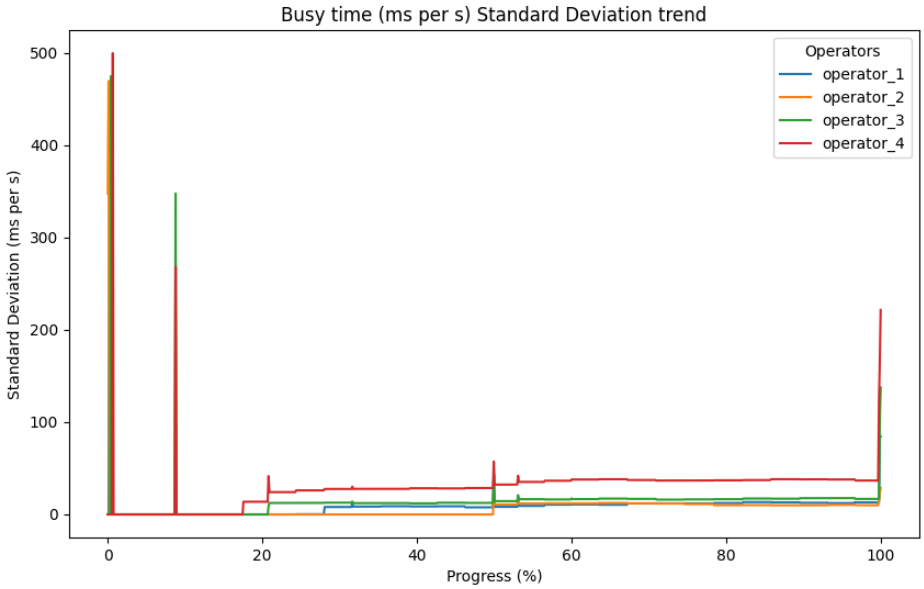

Figure A.11: Standard Deviation of all the operators for the SSB dataset with parallelism of 64

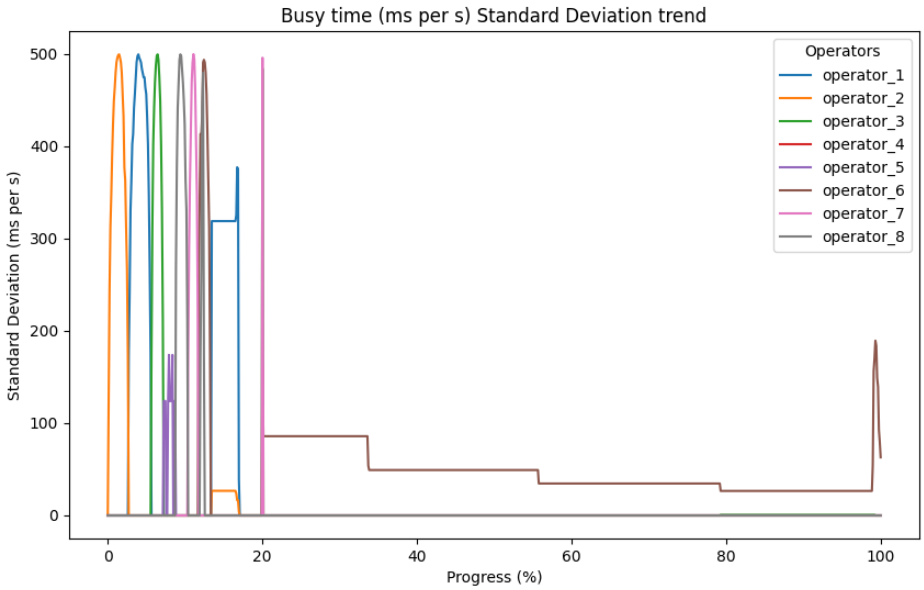

Figure A.12: Standard Deviation of all the operators for the TPC-H dataset with parallelism of 64

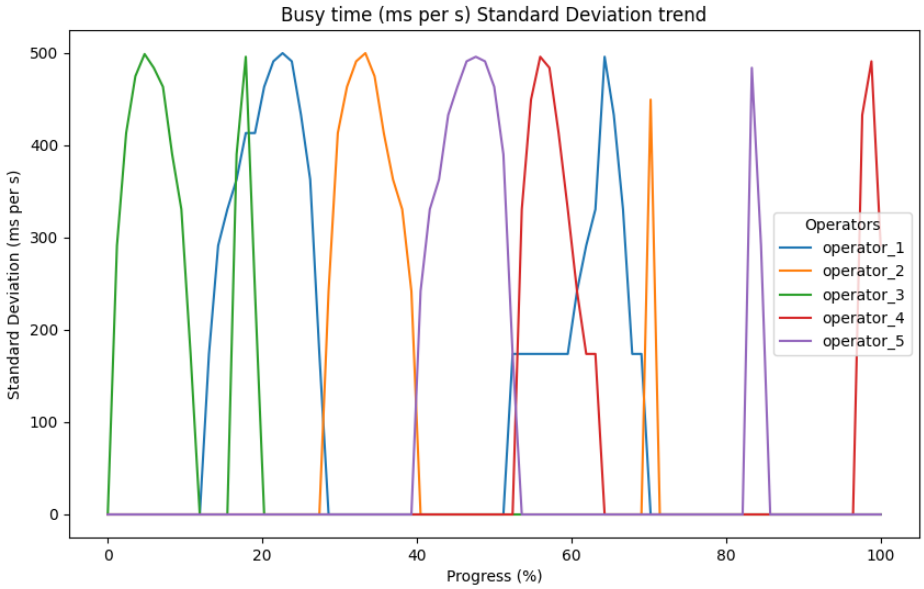

Figure A.13: Standard Deviation of all the operators for the Housing dataset with parallelism of 32

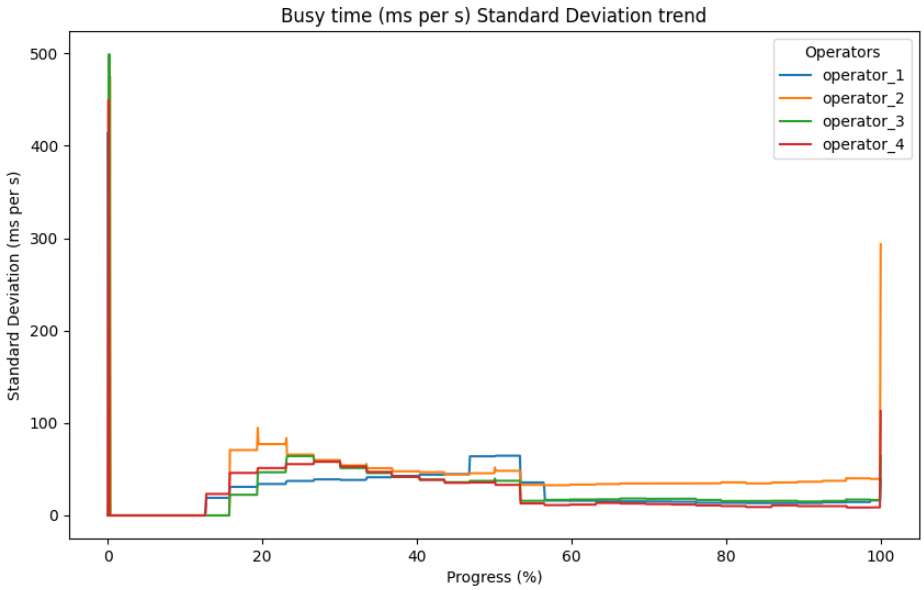

Figure A.14: Standard Deviation of all the operators for the SSB dataset with parallelism of 32

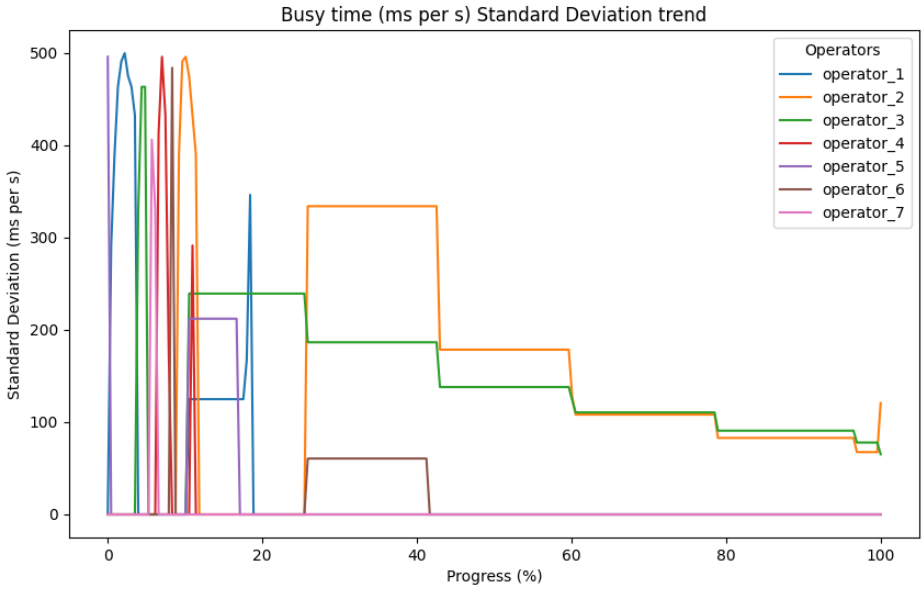

Figure A.15: Standard Deviation of all the operators for the TPC-H dataset with parallelism of 32

A.4 Mean

This section presents the mean values for the time that each operator spends computing per second. As discussed earlier, all operators are not under full load except the Housing dataset, however, since it's a small dataset, it gets processed very close to the initial phase itself.

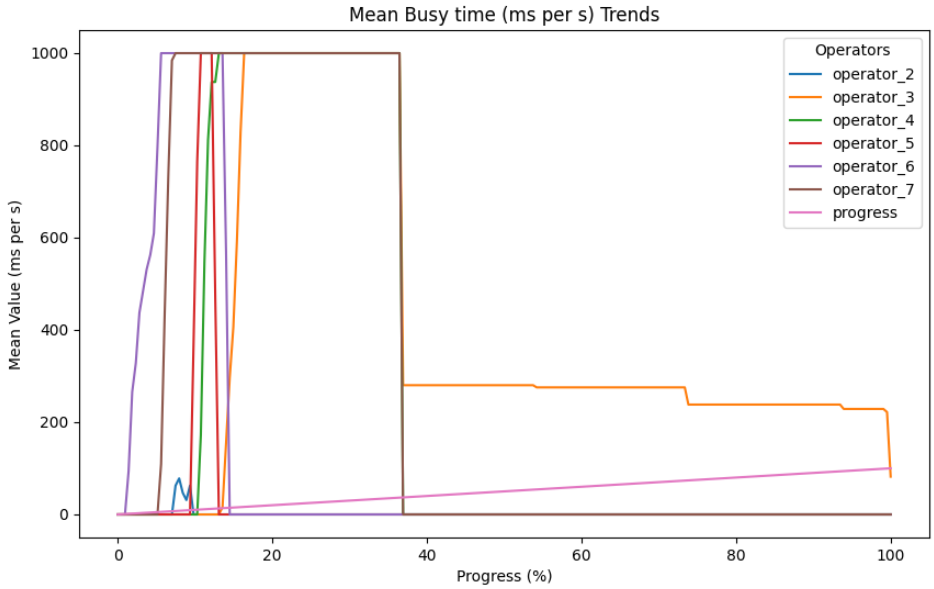

Figure A.16: Mean busy time (ms per s) of all the operators for the Housing dataset with parallelism of 64

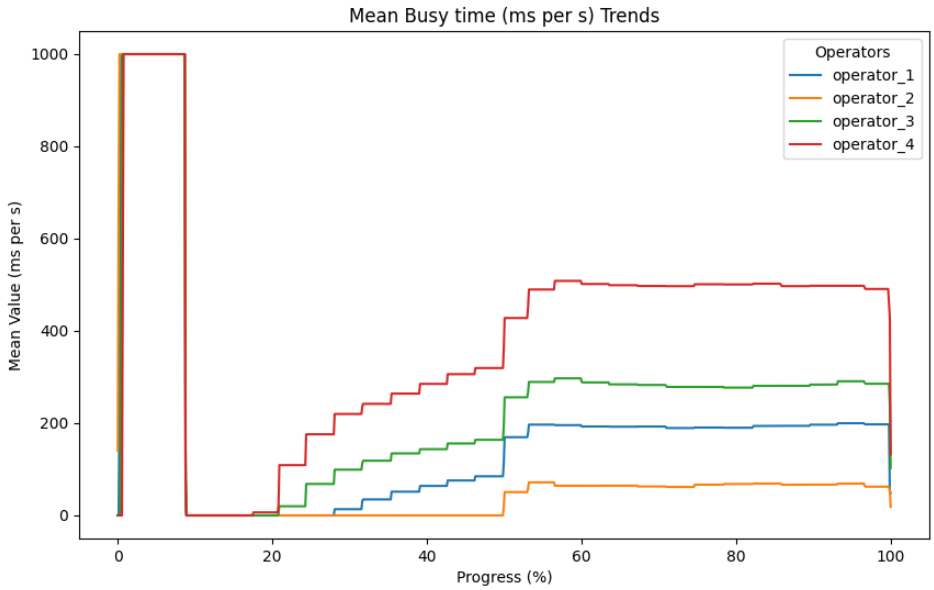

Figure A.17: Mean busy time (ms per s) of all the operators for the SSB dataset with parallelism of 64

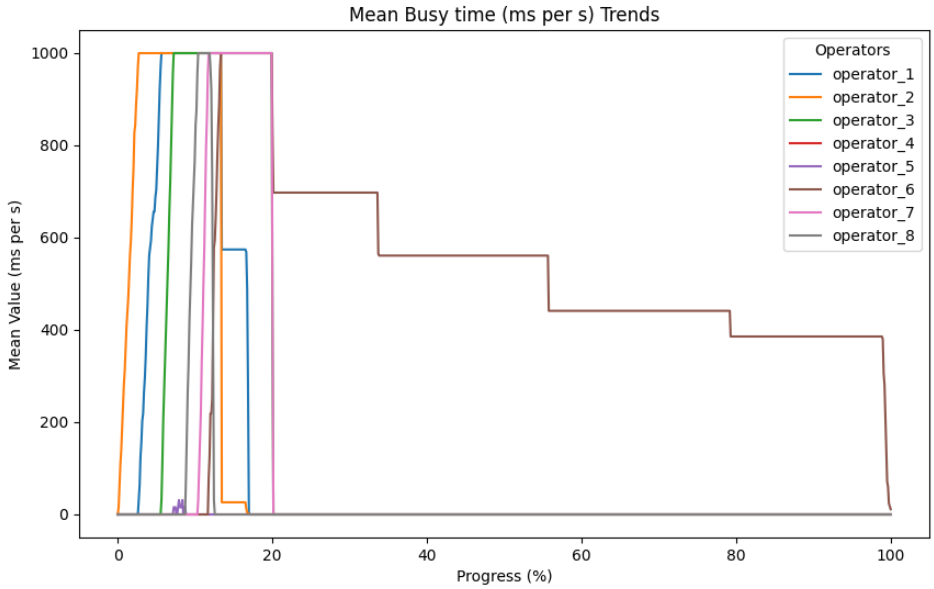

Figure A.18: Mean busy time (ms per s) of all the operators for the TPC-H dataset with parallelism of 64

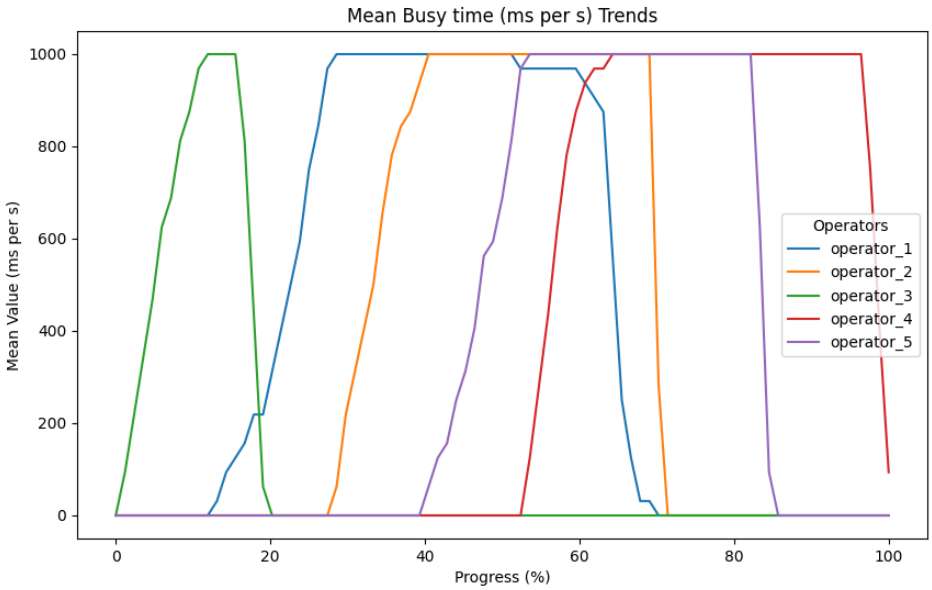

Figure A.19: Mean busy time (ms per s) of all the operators for the Housing dataset with parallelism of 32

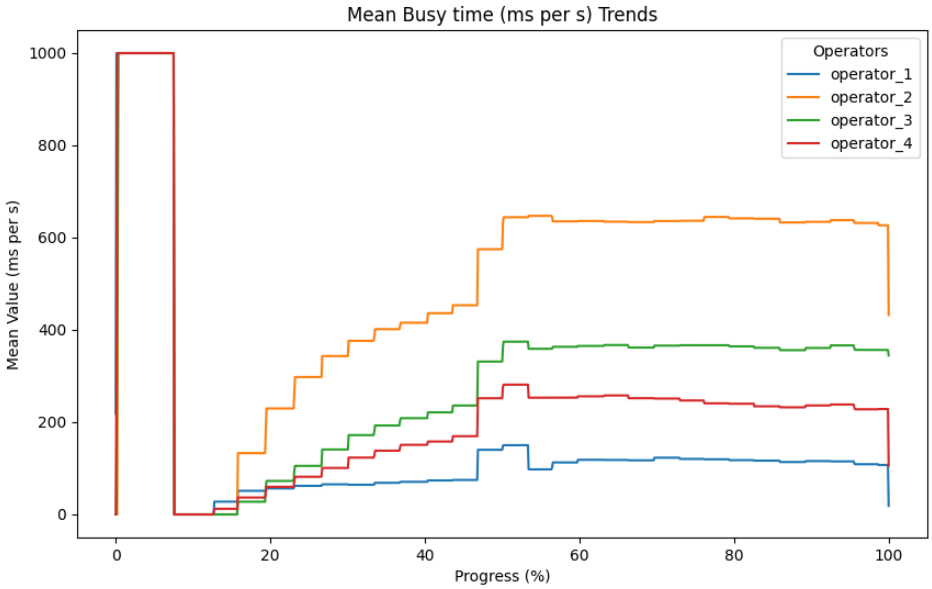

Figure A.20: Mean busy time (ms per s) of all the operators for the SSB dataset with parallelism of 32

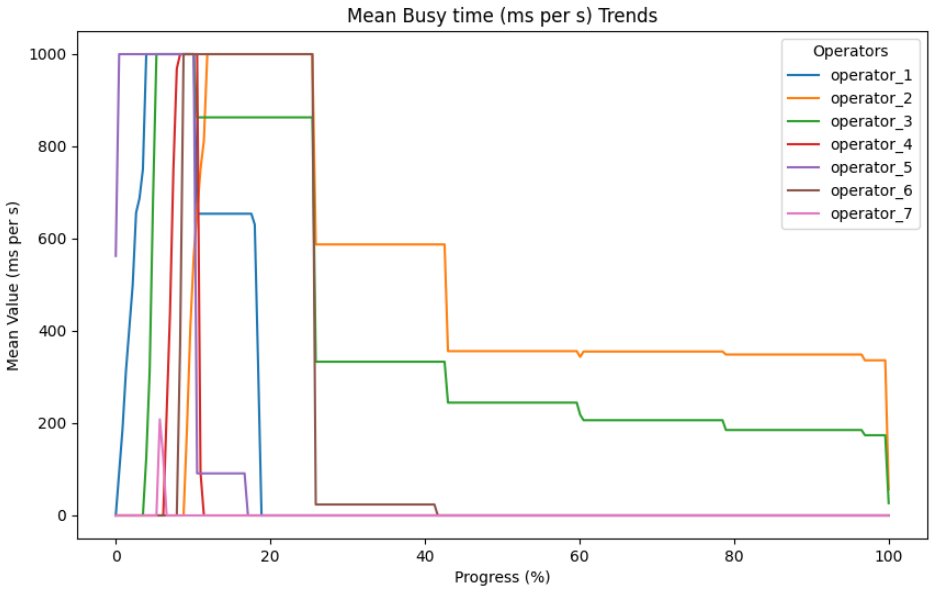

Figure A.21: Mean busy time (ms per s) of all the operators for the TPC-H dataset with parallelism of 32

A.5 Algorithms

This section presents the pseudocode for the algorithms mentioned in the previous sections.

Algorithm 1 View Tree Construction Algorithm

```
1:  function CONSTRUCT(w, F, ancestors)
2:      if w.isRelation then
3:          node ← new ViewTree()
4:          node.relations ← {w.relations[0]}
5:          node.id ← "R[" + w.relations[0] + "]"
6:          return node
7:      end if
8:      node ← new ViewTree(w.variableName if |w.children| > 1 else ∅)
9:      node.id ← "V_at_" + w.variableName + "_" + join(w.relations)
10:     R ← ∅
11:     for all child ∈ w.children do
12:         ancestorsClone ← Clone(ancestors)
13:         for all r ∈ w.relations do
14:             ancestorsClone[r] ← ancestorsClone[r] ∪ {w.variableName}
15:         end for
16:         childNode ← Construct(child, F, ancestorsClone)
17:         childNode.parent ← node
18:         node.children ← node.children ∪ {childNode}
19:         R ← R ∪ child.relations
20:         node.variablesBelowMe←
              node.variablesBelowMe∪childNode.variablesBelowMe
21:         if childNode.marginalizedVariables ≠ ∅ then
22:             node.variablesBelowMe ←
              node.variablesBelowMe ∪ childNode.marginalizedVariables
23:         end if
24:     end for
```

```
25:     if |w.children| > 1 then
26:         node.isJoin ← True
27:     end if
28:     K ← ⋃_{child∈node.children} child.freeVariables
29:     L ← K
30:     if w.shouldStopPushdown then
31:         K ← K ∪ {w.variableName}
32:     else
33:         K ← K ∩ F
34:     end if
35:     K ← K ∪ getDeps(w, ancestors)
36:     L ← L \ K
37:     node.freeVariables ← List(K)
38:     node.targetVariables ← {w.variableName}
39:     node.relations ← List(R)
40:     if ¬w.shouldStopPushdown ∧ w.variableName ∉ F then
41:         L ← L ∪ {w.variableName}
42:         node.marginalizedVariables ← List(L)
43:     end if
44:     if |node.children| = 1 ∧ ¬node.children[0].doesRepresentARelation() then
45:         child ← node.children[0]
46:         if child.isJoin then
47:             node.children ← {child}
48:         else
49:             node.marginalizedVariables←
            node.marginalizedVariables∪child.marginalizedVariables
50:             node.targetVariables ← node.targetVariables ∪ child.targetVariables
51:             if |node.children| > 1 then
52:                 node.relations ← node.relations ∪ child.relations
53:             end if
54:             node.children ← node.children ∪ child.children
55:         end if
56:     end if
57:     return node
58: end function
```

```
59: function GETDEPS(w, ancestors)
60:     R_s ← ∅
61:     Q ← Queue()
62:     Q.add(w)
63:     while ¬Q.isEmpty() do
64:         current ← Q.remove()
65:         R_s ← R_s ∪ current.relations
66:         if ¬current.isRelation then
67:             Q.addAll(current.children)
68:         end if
69:     end while
70:     R_s ← R_s ∩ ancestors.keySet()
71:     deps ← ∅
72:     for all r ∈ R_s do
73:         deps ← deps ∪ ancestors[r]
74:     end for
75:     return deps
76: end function
```

Algorithm 2 Constructing Delta Tree for Stream Processing

```
1: function CONSTRUCTTREE(τ, r, attributeIdx, tEnv)
2:     if τ represents a relation then
3:         Create a new node node of type DeltaTreeForStream
4:         node.stream ← TODATASTREAM(r[τ.relations[0] + "Streamed"].stream)
5:         return node
6:     end if
7:     deltaTree ← DELTATREEFORSTREAM(τ)
8:     deltaTree.freeVariables ← τ.freeVariables
9:     Initialize localToGlobalAttributeMapping ← {}
10:    Initialize attrIdxInCurrContext ← {}
11:    idxForAttrIdxInCurrContext ← 0
12:    for all attr ∈ τ.marginalizedVariables do
13:        attrIdxInCurrContext[attr] ← idxForAttrIdxInCurrContext++
14:    end for
15:    for all attr ∈ τ.variablesBelowMe do
16:        attrIdxInCurrContext[attr] ← idxForAttrIdxInCurrContext++
17:    end for
18:    hasDeltaChild ← false
19:    childIdx ← 0
20:    for all child ∈ τ.children do
21:        hasDeltaChild ← true
22:        Initialize localAttributeIdx ← {}
23:        if child.marginalizedVariables ≠ null then
24:            currChildVariablesToIndex ←
           SORTEDSET(child.variablesBelowMe.marginalizedVariables)
25:            currLocalToGlobalMapping ←
           new array of size |currChildVariablesToIndex|
26:            idx ← 0
27:            for all var ∈ currChildVariablesToIndex do
28:                if var ∈ child.marginalizedVariables then
29:                    localAttributeIdx[var] ← idx
30:                end if
31:                currLocalToGlobalMapping[idx++] ←
           attrIdxInCurrContext[var]
32:            end for
33:            localToGlobalAttributeMapping[childIdx] ←
           1.5emcurrLocalToGlobalMapping
34:        end if
```

```
35:         streamingChild ← CONSTRUCTTREE(child, r, localAttributeIdx, tEnv)
36:         Add streamingChild to deltaTree.children
37:         childIdx++
38:     end for
39:     if hasDeltaChild then
40:         deltaTree.stream ←CREATEDELTAVIEW(deltaTree, attrIdxInCurrContext,
    localToGlobalAttributeMapping, |attrIdxInCurrContext|, tEnv)
41:     end if
42:     return deltaTree
43: end function

44: function CREATEDELTAVIEW(root, attributeIdx, localToGlobalAttributeMapping,
    noOfFeatures, tEnv)
45:     tablesToJoin ← [child.id for each child ∈ root.children]
46:     if |root.children| = 1 then
47:         stream ← root.children[0].stream
48:     else
49:         stream ← root.children[0].stream
50:         freeVariables1 ← root.children[0].freeVariables
51:         for i ← 1 to |root.children| − 1 do
52:             stream ← stream.keyBy(new JoinKeySelector(root.joinVariable))
53:
54:             stream ← stream.connect(
                    root.children[i].stream.keyBy(new JoinKeySelector(root.joinVariable))
                    )
55:             stream ← stream.flatMap(
                    new UDFsForStream.customJoinAndGoupBy(
                    root.joinVariable, freeVariables1, root.children[i].freeVariables, i
                    ))
56:             freeVariables1 ← null
57:         end for
58:     end if
59:     stream ← stream.map(new MarginalisationFirstHalf(
        root.marginalizedVariables, root.freeVariables, |root.children|, attributeIdx,
        localToGlobalAttributeMapping, noOfFeatures))
60:     return stream
61: end function
```

Algorithm 3 Custom Join and GroupBy Operation

Input:
row: New tuple from either stream
out: Collector to output results
stateForTable1: MapState for table 1
stateForTable2: MapState for table 2
groupByAttributes: Optional group-by attributes
isLeft: Boolean flag indicating the left table
Initialization:
myMappedValueName ← if isLeft then "*mappedValue*" else "*mappedValue*" + indexFor2ndTable
otherSideMappedValueName←if isLeft then "*mappedValue*"+ indexFor2ndTable else "*mappedValue*"
key ← Concatenate relevant attributes of row
inputRowKind ← row.getKind()
row.setKind(INSERT)
oldMappedValue ← stateForTable1.get(key).getField("mappedValue") if exists
leftHalf ← Row with selected attributes from row
rightHalf ← {}
while otherSideRow ← stateForTable2.values().next() **do**
    output ← Row with fields from otherSideRow
    if groupByAttributes ≠ null and oldMappedValue ≠ null **then**
        output ← join(output, leftHalf)
        output.setField(myMappedValueName, oldMappedValue)
        output.setKind(UPDATE_BEFORE)
        out.collect(output)
    end if
    rightHalf.add(output)
end while
if groupByAttributes==null and oldMappedValue==null and isRetract(inputRowKind) **then**
    throw Error: "Retraction for unseen tuple in Join operation."
end if

```
24: Update oldMappedValue:
25: if groupByAttributes ≠ null then
26:     incomingMappedValue ← row.getField("mappedValue")
27:     if oldMappedValue = null then
28:         oldMappedValue ← new accumulator array
29:     end if
30:     if isRetract(inputRowKind) then
31:         UDFsForStream.ringSum.retract(
            oldMappedValue, incomingMappedValue)
32:     else
33:         UDFsForStream.ringSum.accumulate(
            oldMappedValue, incomingMappedValue)
34:     end if
35:     newMappedValue ← oldMappedValue
36:     leftHalf.setField("mappedValue", newMappedValue)
37:     stateForTable1.put(key, Row.copy(leftHalf))
38: else
39:     if isRetract(inputRowKind) then
40:         stateForTable1.remove(key)
41:     else
42:         stateForTable1.put(key, Row.copy(leftHalf))
43:     end if
44: end if
45: if !isLeft then
46:     leftHalf.setField(myMappedValueName, leftHalf.getField("mappedValue"))
47: end if
48: for all rowTable2 ∈ rightHalf do
49:     output ← join(leftHalf, rowTable2)
50:     output.setKind(UPDATE_AFTER if groupByAttributes else inputRowKind)
51:     out.collect(output)
52: end for
```

Algorithm 4 Compute Cross-Correlation for Quadratic Aggregates

function COMPUTECROSSCORRELATION($\mathbf{a}, \mathbf{b}, stA, endA, stB, endB, \mathbf{res}, \mathbf{g}$)
    Input:
        a: Covariance structure a
        b: Covariance structure b
        stA, $endA$: Start and end+1 indices of linear aggregates in **a**
        stB, $endB$: Start and end+1 indices of linear aggregates in **b**
        res: Covariance structure storing the result of the Ring product operation
        g: Mapping of indices of attributes in child covariance structure to those of parents
    $aLen \leftarrow endA - stA$
    $offset \leftarrow 1 + aLen$
    $bLen \leftarrow endB - stB$
    for $i \leftarrow 0$ to $bLen - 1$ **do**
        $mappedIdx \leftarrow g[i]$
        $flattendedRow \leftarrow offset + \frac{mappedIdx \times (mappedIdx+1)}{2}$
        for $j \leftarrow 0$ to $mappedIdx - 1$ **do**
            $flattenedLocation \leftarrow flattendedRow + j$
            $\mathbf{res}[flattenedLocation] \leftarrow \mathbf{res}[flattenedLocation] + \mathbf{b}[stB + i] \times \mathbf{a}[stA + j]$
        end for
    end for
    for $i \leftarrow 0$ to $bLen - 1$ **do**
        $mappedIdx \leftarrow g[i]$
        for $j \leftarrow mappedIdx$ to $aLen - 1$ **do**
            $flattenedLocation \leftarrow offset + \frac{j \times (j+1)}{2} + mappedIdx$
            $\mathbf{res}[flattenedLocation] \leftarrow \mathbf{res}[flattenedLocation] + \mathbf{b}[stB + i] \times \mathbf{a}[stA + j]$
        end for
    end for
    return res
end function

Algorithm 5 Compression Operator for Marginalization and Ring Product Calculation

Require: marginalizedAttributes: Attributes to be marginalized
Require: freeVariables: Variables to output
Require: numInstancesMapped: Instances of mapped attributes
Require: attributeIdx: Mapping of attribute names to indices
Require: localToGlobalMapping: Local to global attribute index mapping
Require: numFeatures: Number of features

```
1: function COMPRESSIONOPERATOR(marginalizedAttributes, freeVariables,
   numInstancesMapped, attributeIdx, localToGlobalMapping, numFeatures)
2:     if marginalizedAttributes ≠ null then
3:         marginalisedAttributes ← marginalizedAttributes.toArray()
4:         numAttributesToMarginalize ← marginalizedAttributes.size()
5:     end if
6:     if freeVariables ≠ null then
7:         variablesToOutput ← freeVariables.toArray()
8:     end if
9:     numInstancesMapped ← numInstancesMapped
10:    localAttributeIdx ← attributeIdx
11:    localToGlobalMapping ← localToGlobalMapping
12:    numFeatures ← numFeatures
13:
14:    function MAP(val)
15:        originalSize ← val.getArity()
16:        inputRowKind ← val.getKind()
17:        hasMappedValue ← val.getFieldNames(true).contains(”mappedValue”)
18:        valuesToMarginalize ← new double[numAttributesToMarginalize]
19:        idxForMarginalizedValues ← new int[numAttributesToMarginalize]
20:        for i ← 0 to numAttributesToMarginalize −1 do
21:            idxForMarginalizedValues[i] ←
           localAttributeIdx.get(marginalisedAttributes[i])
22:            valuesToMarginalize[i] ←
           (val.getField(marginalisedAttributes[i]) instanceof Integer
           ? (int)val.getField(marginalisedAttributes[i])
           : (double)val.getField(marginalisedAttributes[i])
23:        end for
```

```
24:     if hasMappedValue then
25:         allMappedValueAttributes ← new double[numInstancesMapped][]
26:         allMappedValueAttributes[0] ←
        (double[]) val.getField(”mappedValue”)
27:         for j ← 1 to numInstancesMapped −1 do

28:             allMappedValueAttributes[j] ←(double[]) val.getField(
        ”mappedValue_” + j)
29:         end for
30:     else
31:         allMappedValueAttributes ← null
32:     end if
33:     mappedValues ← UDFsForStream.ringProduct.eval(
        idxForMarginalizedValues, valuesToMarginalize,
        allMappedValueAttributes, localToGlobalMapping, numFeatures)
34:     output ← Row.withNames(inputRowKind)
35:     for each key in variablesToOutput do
36:         output.setField(key, (double)val.getField(key))
37:     end for
38:     output.setField(”mappedValue”, mappedValues)
39:     return output
40: end function
```

Algorithm 6 Ring Product

1: **function** PRODUCTFORMAPPEDVALUESOFCHILDREN(noOfFeaturesInCurrContext, localToGlobalAttributeMapping, payloads)
2: $aggregateLen \leftarrow 1 + noOfFeaturesInCurrContext + \frac{noOfFeaturesInCurrContext \cdot (noOfFeaturesInCurrContext+1)}{2}$
3: $a \leftarrow payloads[0]$
4: $len \leftarrow$ length of $payloads$
5: **for** $childIdx \leftarrow 1$ **to** $len - 1$ **do**
6: $res \leftarrow$ new array of size $aggregateLen$
7: $b \leftarrow payloads[childIdx]$
8: $globalMappingsForChild \leftarrow localToGlobalAttributeMapping[childIdx - 1]$
9: $numberOfFeaturesInCurrChild \leftarrow$ length of $globalMappingsForChild$
10: $res[0] \leftarrow a[0] \cdot b[0]$
11: $idxInCurr \leftarrow 1$
12: **for** $j \leftarrow 0$ **to** $numberOfFeaturesInCurrChild - 1$ **do**
13: $destMapping \leftarrow globalMappingsForChild[j] + 1$
14: $res[destMapping] \leftarrow a[0] \cdot b[idxInCurr++]$
15: **end for**
16: **for** $j \leftarrow 1$ **to** $noOfFeaturesInCurrContext$ **do**
17: $res[j] \leftarrow res[j] + b[0] \cdot a[j]$
18: **end for**
19: **// Computing** $-b_C \cdot a_Q + a_C \cdot b_Q$
20: **for** $i \leftarrow 0$ **to** $numberOfFeaturesInCurrChild - 1$ **do**
21: $destI \leftarrow globalMappingsForChild[i]$
22: **for** $j \leftarrow 0$ **to** i **do**
23: $destJ \leftarrow globalMappingsForChild[j]$
24: $res\left[\frac{destI \cdot (destI+1)}{2} + destJ + 1 + noOfFeaturesInCurrContext\right] \leftarrow a[0] \cdot b[idxInCurr++]$
25: **end for**
26: **end for**
27: $lim \leftarrow aggregateLen$
28: **for** $j \leftarrow 1 + noOfFeaturesInCurrContext$ **to** $lim - 1$ **do**
29: $res[j] \leftarrow res[j] + b[0] \cdot a[j]$
30: **end for**
31: **// Adding to previous** $-a_S \cdot b_S^T + s_b \cdot s_a^T = 0$

32: $res \leftarrow$ Matrix.elementWiseSum(
res,
Matrix.computeCrossCoRelationForQuadraticAggregates(
$a, b, 1, 1+$noOfFeaturesInCurrContext$, 1, 1+numberOfFeaturesInCurrChild$,
$globalMappingsForChild$),
$1 + noOfFeaturesInCurrContext$,
!App.useObjectReuse
)
33: $a \leftarrow res$
34: **end for**
35: **return** a
36: **end function**
37: **function** EVAL(idxForMarginalisedValues, row, allMappedValueAttributes, localToGlobalAttributeMapping, noOfFeatures)
38: $n \leftarrow 1 + noOfFeatures + \frac{noOfFeatures \cdot (noOfFeatures+1)}{2}$
39: $aggregateForLifted \leftarrow$ new array of size n
40: $len \leftarrow$ length of row
41: **for** $idx \leftarrow 0$ **to** $len - 1$ **do**
42: $attributeIdx \leftarrow idxForMarginalisedValues[idx]$
43: $value \leftarrow row[idx]$
44: $aggregateForLifted[1 + attributeIdx] \leftarrow value$
45: **end for**
46: $aggregateForLifted[0] \leftarrow 1.0$
47: **// Compute** $s1^T \cdot S2 + s2^T \cdot S1$
48: $idxForQuad \leftarrow 1 + noOfFeatures$
49: **for** $j \leftarrow 1$ **to** $noOfFeatures$ **do**
50: **for** $k \leftarrow 1$ **to** j **do**
51: $aggregateForLifted[idxForQuad\ +\ +] \leftarrow aggregateForLifted[j] \cdot aggregateForLifted[k]$
52: **end for**
53: **end for**

54: **if** $allMappedValueAttributes \neq$ null **and**
length of $allMappedValueAttributes \neq 0$ **then**
55: $payloads \leftarrow$ new array of size $(1 +$ length of $allMappedValueAttributes)$
56: $payloads[0] \leftarrow aggregateForLifted$
57: **for** $i \leftarrow 0$ **to** length of allMappedValueAttributes - 1 **do**
58: $payloads[i+1] \leftarrow$ allMappedValueAttributes$[i]$
59: **end for**
60: $finalres \leftarrow$ productForMappedValuesOfChildren(
$noOfFeatures, localToGlobalAttributeMapping, payloads$
)
61: **else**
62: $finalres \leftarrow aggregateForLifted$
63: **end if**
64: **return** $finalres$
65: **end function**

Algorithm 7 Gradient Descent for Parameter Optimization

Require: covarianceMatrix: Covariance matrix computed by the project

Require: θ: Previously learned values for the parameters

Require: learning_rate: Learning rate α

Require: num_epochs: Number of epochs N

function GRADIENTDESCENT(covarianceMatrix, θ, learning_rate, num_epochs)
    $n \leftarrow$ length(θ)
    for epoch $\leftarrow 0$ to num_epochs $- 1$ **do**
        $\Delta\theta \leftarrow \mathbf{0}$
        for thetaIdx $\leftarrow 0$ to $n - 1$ **do**
            if thetaIdx $=$ this.attributeToPredict **then**
                $\Delta\theta$[thetaIdx] $\leftarrow 0$
                continue
            end if
            $val \leftarrow 0$
            for featureIdx $\leftarrow 0$ to $n - 1$ **do**
                $i \leftarrow$ thetaIdx
                $j \leftarrow$ featureIdx
                if $j > i$ **then**
                    swap(i, j)
                end if
                $location \leftarrow \frac{i \cdot (i+1)}{2} + j$
                $val \leftarrow val + \theta$[featureIdx] $\cdot$ covarianceMatrix[$location$]
            end for
            $\Delta\theta$[thetaIdx] $\leftarrow val$
        end for
        for $i \leftarrow 0$ to $n - 1$ **do**
            if $i =$ this.attributeToPredict **then**
                continue
            end if
            $step \leftarrow \Delta\theta[i] \cdot$ learning_rate
            $\theta[i] \leftarrow \theta[i] - step$
        end for
    end for
    return θ
end function