Distributed Incremental Processing using Apache Flink

Sebastian Kulaj

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

Abstract

As we produce and process continuously increasing amounts of data, the need for efficient data processing is becoming even more essential. Over the years, researchers have presented many techniques to optimise data processing across many fields, including relational databases. Incremental view maintenance is one of these techniques that enables incremental processing of SQL queries and reduces the time complexity of SQL query execution. Many systems embrace this technique, including the F-IVM system [1], a state-of-the-art system for maintaining analytics over the relation data. F-IVM transforms SQL queries into a tree of views that not only enables incremental processing but also significantly reduces execution time.

However, many stream processing engines, including Apache Flink [2], do not support state-of-the-art optimisation techniques. As Nikolic et al. [3] show, the F-IVM system significantly outperforms Apache Flink and other state-of-the-art systems. Therefore, adding support for the F-IVM's transformations to Apache Flink would highly benefit its users by reducing query execution time and costs associated with the computational resources. This project aims to close this gap by introducing support for the F-IVMs transformations to Flink. We propose two solutions based on the DataStream and Table APIs that translate the F-IVM's transformations into Flink's operators. Experiments show that our solutions outperform the standard Flink SQL execution across various SQL queries.

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.

(*Sebastian Kulaj*)

Acknowledgements

I would like to convey my sincere gratitude to my supervisor, Milos Nikolic, for his continuous patience and support throughout the entire project and writing. Milos's expertise and suggestions have taught me a lot and were invaluable to the completion of this project.

Table of Contents

Chapter 1

Introduction

Over the last 20 years, we have seen an astronomic increase in the volume of data we produce and process. This increase led to the creation of tools that aimed to process large amounts of data efficiently [4], [5]. As more companies rely on real-time data analyses to drive their business decisions, the stream processing engines that process unbounded data streams have become widely used. Apache Flink became one of the most commonly used stream processing engines.

With the surge in popularity of cloud services such as AWS, many companies are moving their resources to these platforms. According to the survey conducted by the Boston Consulting Group [6], companies spend up to 80% of their cloud budget on computational resources. Therefore, processing data most efficiently becomes one of the most important optimisations to cut costs related to companies' cloud spending. The latest growth in popularity of large language models such as ChatGPT [7] has increased this need as the models require enormous amounts of data to be processed for training.

Throughout the years, researchers have proposed many techniques to address the problem of efficient computation, and incremental processing became one of the main techniques that address this issue. Incremental processing aims to recompute a part of a task upon receiving new data to avoid recomputation of an entire task every time it receives new data. This technique has been introduced to many fields, including relational databases, where the incremental view maintenance (IVM) method has significantly improved efficient computation. This technique transforms a SQL query into a tree of views, reducing its time complexity and allowing incremental processing.

F-IVM [1] is one of the state-of-the-art systems that adopts this technique. The F-IVM system transforms queries that contain sum aggregation into trees of views and highly reduces their time complexity, which results in faster and more efficient

Query	Table API Solution	DataStream Solution	Standard Flink SQL
TPC-H 10*	**241,226**	139,745	83,896
TPC-H 12*	**300,649**	147,955	119,446
TPC-H 3*	**723,108**	227,999	570,986
TPC-H 14*	**679,883**	197,497	626,385

Table 1.1: Throughput of processing slightly modified TPC-H queries.
Throughput is defined as the number of records in the relations divided by the execution time in seconds. The higher the throughput, the better.
The queries can be found in Appendix A.

execution. Nikolic et al. [3] show that their system outperforms Apache Flink and other state-of-the-art systems that use IVM in several different tasks.

Although Flink supports executing SQL queries, it lacks support for state-of-the-art optimisations that highly improve execution. Nikolic et al. [3] show in their experiments that lack of support for these optimisations significantly impacts Flink's performance, resulting in much longer and less resource-efficient execution that increases costs. The optimisations supported by F-IVM that Flink lacks are factorisation and incremental computation, among others. Factorisation refers to pushing aggregation past joins, which highly reduces the time complexity of query execution because we join aggregated values instead of records, highly decreasing the number of matching tuples. We explain the F-IVM's optimisations and their benefits in detail in Chapter 2. However, to highlight how these optimisations impact performance, Table 1.1 illustrates their impact on the throughput. The DataStream and Table API solutions that implement the F-IVM's optimisations largely outperform the standard Flink SQL execution, which does not support these optimisations, on TPC-H 10* and 12* queries that do not have WHERE clauses. For TPC-H 3* and 14* queries that have highly selective WHERE clauses, the Table API solution still outperforms the standard Flink SQL execution.

Therefore, our project aims to add support for the F-IVM's optimisations to Apache Flink, which can vastly benefit its users by reducing the time execution of SQL queries and decreasing costs related to computational resources. Our project works as a bridge between F-IVM and Apache Flink. It incorporates F-IVM's transformed query execution (a tree of views) and its optimisations using a JSON input file that encodes these transformations. The JSON input file can be created from the F-IVM's .M3 output file, which encodes transformations and works as input to the F-IVM backend, DBToaster.

We introduce two solutions that are based on two different APIs, the DataStream and Table API. Our reasons for proposing the two solutions are to benchmark these APIs and satisfy two conditions: support as many F-IVMs transformations as possible and do it most efficiently. Our initial research showed that the Table API solution is more performant, but it does not enable supporting every transformation. Thus, we stick to maintaining two solutions. Our solution takes transformed queries by F-IVM in the form of JSON files and translates them into Flink's operators. Our JSON format allows for an effortless translation from the F-IVM's .M3 output files. The DataStream solution compromises the creation of three custom Flink operators that perform the F-IVM's transformations. The custom operators are:

- Aggregate: this operator performs aggregation tasks over time windows to improve the performance of this operator.
- Join-Aggregate: this operator combines a join of two streams and the following aggregation into a single operator to boost the throughput of our solution.
- Multi-Join: this operator facilitates a join of multiple streams in a single operator, a common transformation performed by F-IVM.

While the DataStream solution involves creating custom operators, in the Table API solution, we map F-IVM's transformations into built-in Table API operators: `groupBy`, `join`, `select` and `where`. The use of built-in operators limits the number of transformations we can support, as the Table API does not enable the creation of custom operators. Thus, this solution does not support transformations that involve joining more than two streams in a single view.

Our experiments show that our solutions outperform the standard Flink SQL execution in various SQL queries, especially those that do not involve highly selective WHERE clauses. In addition, our solution shines when queries involve the multi-stream join transformation, as an increase in relations that are joined does not affect our solution, which is not the case with the standard Flink SQL execution that is highly impacted by this increase. We believe the improvement in the throughput of our solutions compared to the standard Flink SQL can significantly help Flink's users in lowering costs associated with computational resources by reducing the execution time of SQL queries.

1.1 Contributions

The contributions of this project are:

- The design of the JSON representation of the F-IVM transformations that enables a smooth translation from the F-IVM .M3 output file.
- The design and implementation of the DataStream solution that enables executing SQL queries in F-IVM's transformed fashion. The implementation includes creating three custom Flink operators that process the data.
- The design and implementation of the Table API solution that allows executing SQL queries in F-IVM's transformed fashion. The implementation involves translating the F-IVM transformations into the Table API built-in operators.
- Thorough evaluation of the performance of our two solutions, including benchmarking them against the standard Flink SQL execution and the F-IVM system.

1.2 Report Outline

Chapter 2: **Background** first provides essential knowledge about incremental computation and stream processing. Then, the chapter highlights relevant work to the F-IVM system and Apache Flink. The chapter ends by describing the essential components of the F-IVM system and Apache Flink.

Chapter 3: **Design and Implementation** starts with highlighting the system requirements of our solution and describing its design. The chapter ends by describing the implementation of both solutions.

Chapter 4: **Evaluation and Experiments** shows the performance of our solutions and compares it with the performance of standard Flink SQL execution and the F-IVM system across a number of different queries.

Chapter 5: **Conclusions** presents our results and final conclusions and highlights future work.

Chapter 2

Background

This chapter covers the essential concepts of incremental computation, stream processing, the F-IVM system and Apache Flink, which are relevant to this project. First, we address vital information about the notions of incremental computation and stream processing. Then, we dive into the related work that laid the groundwork for F-IVM and Apache Flink. In the next section, we describe the essential concepts of the F-IVM system, and we finish this chapter with an explanation of the fundamental concepts of Apache Flink.

2.1 Incremental Processing

Incremental processing refers to the approach of efficiently updating the results of the task that is processed upon input changes without the need to recompute the whole task. Suppose we have a task *f(x, y)* that consists of two inputs: x and y. When only one part of the task changes, such as x, we want to update the task's result without recomputing the part that involves y [8]. This type of computation has many use cases, especially in the streaming environment where the data is constantly coming to systems, and the constant recomputation of the entire task would be impossible in practice.

One of the methods to achieve incremental computation is to transform a program that performs the computation in a non-incremental way by dividing the computation task into subproblems, where each subproblem computes a partial result and depends on a single part of the original task. The combination of partial results yields the same results as the original computation. To further improve the partial results, we can cache them and reuse the previous values to compute updates more efficiently [8] [9].

One such example is a task that computes an average of temperature. A non-

incremental version of this program would store every incoming temperature value and recompute the average by summing all the values and dividing the sum by the number of values. It would be extremely inefficient if our system received thousands of records every minute or second as the recomputation time and the memory needed to store these values would rise indefinitely. However, by modifying the computation to maintain track of the sum and the count of incoming records, we can update the average using these two values: new average = new sum / new count. This way, we only store two values, and the update takes constant time.

2.2 Stream Computation

Stream computation relates to the processing of continuous unbounded data streams in a real-time manner. Due to the unbounded nature of data streams, it is often infeasible to perform computations on the entire stream, as this requires unbounded resources. Thus, processing is usually performed on chunks of data called windows, typically defined by a time period or a maximum number of records [10].

As the nature of stream computation differs from regular computation, its programming style also differs. Stream processing often involves three key operations: gather, operate, and scatter. This means that data is first gathered into streams before further processing. Then, kernels perform operations upon the data streams. Finally, the data is scattered back to arrays in memory. This execution style decouples computation from memory access, as the data is first read, then processed, and finally written back [11].

The Dataflow graph model, one of the most widely used stream computation models, uses these principles. The model compromises a directed graph of operators, where each operator performs a certain operation, such as a filter, and passes transformed data to the next operator. Operators that read data from a source are called source operators, and operators that save data to sinks are called sinks [12]. Most popular stream processing engines, such as Apache Flink, Apache Spark Streaming, and Apache Samza, use this execution model.

2.3 Related Work

This section covers the related work on incremental view maintenance and data stream processing, which laid the foundation for the F-IVM system and Apache Flink. We start by highlighting related work on incremental view maintenance. Then, we describe the relevant work on data stream processing.

2.3.1 Incremental Processing in Relational Databases

As the SQL databases became an industry standard to store structured data, a need arose for techniques to process data in the most efficient way. Therefore, many studies have researched methods to address this topic, including Incremental View Maintenance (IVM), upon which the F-IVM system is built.

In 1986, Blakeley et al. [13] published a paper that became one of the foundational works on the IVM topic. They proposed a framework that enables updating materialised views without requiring an entire reevaluation of the query. Their framework consists of two components. The first component detects irrelevant updates to the views that do not affect final results. The second component performs differential updates by identifying which tuples should be inserted or deleted from a view.

Ceri et al. [14] propose a system that automatically derives rules for incremental maintenance for a broad class of views. Their system conducts syntactic analysis on the view definition to determine whether the view can contain duplicates and if it is possible to produce efficient incremental maintenance rules for the tables in that view. Their proposed method cannot handle views with duplicates, but if the view cannot contain duplicates, the system produces a set of rules (triggers) for insert, delete, and update operations for each base table in that view. Their work significantly contributes to the development of triggers in the IVM world.

In the paper [15], Koch presents a new approach for IVM in the settings of a ring of databases. His method utilises key common properties of the query calculus and the ring. These properties eliminate the need for expensive query operations like joins, resulting in a constant number of operations needed for incremental maintenance of aggregate values for non-nested queries. His approach supports only aggregation queries and is expressed in the aggregate query language AGCA. This paper [15] became a foundation for the IVM system DBToaster [16].

Koch et al. present the new state-of-the-art IVM system DBToaster [16]. Their system utilises viewlet transforms, which are recursive finite differencing techniques. This

technique materialises a query along with its higher-order deltas as views. DBToaster takes an SQL query as input and produces a highly optimised procedural C++ code, where essentially the whole work comes down to low-cost updates of materialised views. In their benchmark, DBToaster outperforms two anonymised commercial systems across a set of various queries, including equi-joins and nested aggregates.

Based on the DBToaster, Nikolic et al. [17] analyse the impact of batch processing on the performance of IVM in local and distributed streaming scenarios. They propose a method for incremental processing of queries with nested aggregates for batch updates. Their implementation of distributed IVM inside DBToaster runs on top of Spark and handles tens of millions of updates with a few seconds of latency, utilising hundreds of worker nodes. Their system can outperform up to four orders of magnitude PostgreSQL in incremental batched processing in local settings.

In their innovative work, Nikolic et al. [1] introduce a factorised incremental view maintenance system (F-IVM) that supports a variety of unique applications. Their system works as an extension of DBToaster and consists of two main components. The first component utilises higher-order IVM to simplify the maintenance of the input view by reducing it to a tree of simpler views. The second component makes use of factorised computation for queries with aggregates and joins. F-IVM outperforms DBToaster and first-order IVM, achieving up to two orders of magnitude faster execution across various datasets and queries. Further publications on the F-IVM system [18], [3] describe in detail its capabilities and show its improved performance compared to other systems across a wide range of applications.

2.3.2 Data Stream Processing

The data streaming concept is not new, and researchers have been studying this concept for many even when the capabilities of network transmission and computing hardware were limited. In one of the earliest attempts, Chandrasekaran et al. [19] introduced TelegraphCQ, a processing system for continuous queries over data streams. Their approach to adaptability has distinguished their system from other systems at that time and allowed TelegraphCQ to adapt and adjust to drastic changes in the environment.

Adabi et al. [20] present Aurora, a novel system system for data stream management for monitoring applications. Aurora allows efficient processing of data from monitoring applications by introducing a data model that follows QoS specification and stream-oriented operators. Aurora uses SQuAl (stream query algebra) to implement its seven

operators, which are similar to relation algebra operators, but they differ in how they tackle specific data streaming requirements.

Based on the Aurora System, Adabi et al. [5] introduce Borealis, a new stream-processing engine. Borealis leverages Aurora's stream processing capabilities and distribution functionality from Medusa. Borealis extends append-only Aurora's data model to three types of messages: insertion, deletion and replacement, which allows users to recover from mistakes. Their system also introduces a concept of time travel, allowing users to recompute past states of the data stream, which can be beneficial during debugging or data reproduction. Features like revisions or time travel make Borealis more flexible than its predecessors.

Work on Borealis and other data stream processing engines built a foundation for the current stream processing engines. Noghabi et al. [21] present one such framework, Apache Samza, a new stateful stream-processing framework used at LinkedIn. Samza, like Apache Flink, adapts a Lambda-less design for stream and batch processing. It provides fault tolerance by utilising a changelog mechanism and a partitioned local state, allowing it to handle enormous state sizes of hundreds of terabytes per application. Noghabi et al. show that Samza achieves greater throughput compared to systems such as Spark and Hadoop.

Carbone et al. [22] describe Apache Flink, a data streaming framework that supports batch processing. The paper outlines Flink's architecture and two APIs: DataStream and DataSet. It also talks about the dataflow graph and how the data is exchanged between intermediate operators. The authors highlight how Flink uses checkpointing and partial re-execution to achieve fault tolerance. They present how Flink approaches stream processing compared to batch processing and detail mechanisms behind both types of processing.

2.4 F-IVM

The F-IVM is an incremental processing system that takes an SQL file with a query as input and produces an execution plan in the form of a tree of views that enables highly efficient incremental processing that yields the same results. Such execution form is especially advantageous in a streaming environment where the final results of queries are constantly updated, and a naive execution of such queries will result in a much longer execution time. At the moment, F-IVM supports queries that contain aggregation in the form of a SUM operator and natural joins. F-IVM consists of two

main components [3]:

- Frontend: is a module that converts a query into a tree of views and generates a .M3 file that encodes transformations performed by F-IVM. The .M3 format is used by the DBToaster system as an input file [3].
- Backend: is a DBToaster instance that converts a .M3 file into a highly optimised procedural C++ code and can produce an executable file out of it [3].

F-IVM adapts incremental view maintenance method to provide highly optimised performance, and to achieve it, F-IVM uses three main ingredients [3]:

- Higher-order incremental view maintenance: transforms a query into a hierarchy of simple views, where a view is a function that maps keys (tuples of input values) into payloads, which are elements from a ring (accumulative information, e.g. aggregate values) [3].
- Factorised computation and data representation: supports efficient computation and representation for views and updates by exploiting insights from a query to apply optimisation techniques such as pushing aggregates past joins, processing bulk updates represented as low-rank decompositions and preserving a factorised representation of query results [3].
- Ring abstraction: a ring is a set of D that can contain two binary operations (addition and multiplication) and satisfies the ring axioms. This abstraction allows F-IVM to handle seemingly different tasks in a consistent manner [3].

Example 1. Example SQL query (source [3]):

```
Q := SELECT S.A, S.C, SUM(R.B * T.D * S.E)
FROM R NATURAL JOIN S NATURAL JOIN T
GROUP BY S.A, S.C;
```

The transformed execution of a query has a significant performance improvement over a naive execution of this query. To illustrate this, a naive execution of the query presented in Example 1 first performs joins and then aggregation, resulting in $O(N^3)$ of complexity time. That high time complexity results from how a naive execution performs a join of three tables. The tables are joined one after another, so first, the records from the *R* relation are joined with the records from the *S* relation, and then the results of the join are again joined with the records from the *T* relation. The join of the

R and *S* relations takes $O(N^2)$ as every record from the *R* relation must be matched with every record from the *S* relation to check if there is a match. Then, every record from that join must be matched with every record from the *T* relation, which gives $O(N^2 * N) = O(N^3)$.

However, F-IVM transforms this query into a tree of views (shown in Figure 2.1) and optimises it by pushing aggregation past joins and decomposing aggregation into several views, resulting in O(N) of time [3].The concept of higher-order incremental view maintenance in this tree takes the form of the *Vs*, *Vt*, *Vst* and *Vr* views that perform a partial computation that results in the final values at the *Q* view. Every view maps keys, which are values of the *GROUP BY* columns, to the payloads that are values of the *SUM* aggregate. The *Vs* view maps *A* and *C* to the sum of *E*. The *Vt* view maps *C* to the sum of *E*. The *Vr* view maps *A* to the sum of B. The *Vst* view maps *A* and *C* to the sum of *Sd * Se*. The final results of the query are kept in the *Q* view that maps the values of the *A* and *C* columns to the sum of *Sb * Sc*. The factorised computation ingredient of F-IVM transforms the original query by decomposing the single *SUM(R.B * T.D * S.E)* aggregation into several aggregations and pushing them past joins, achieving the same final results. The decrease of the time complexity to O(N) is the result of pushing aggregations past joins, as now we do not match every tuple from the *R* relation with the tuples of the *S* relation and the resulting tuples with every tuple from the *T* relation. Instead, for the relation *S* joined with relation *T* on the *C* column, there is at most one matching tuple because relation *T* is grouped by the *C* column. The matched tuple is then joined with relation *R* on the *A* column, again resulting in at most one matching tuple as the *R* relation is grouped by the *A* column. That gives a constant time complexity for the *S* relation, but for the *R* and *T* relations, it is O(N) time complexity as the *S* relation is grouped by the *A* and *C* columns because when we join tuples from the *T* relation with the *S* relation on the *A* column, there can be as many as *N* matching tuples. The same applies to tuples from the *R* relation when we join them on the *A* column with the result of the join of *S* and *T* relations, as the results of that join are grouped by the *A* and *C* columns.

F-IVM uses the concept of variable orders to determine the sequence in which the variables are marginalised, which is comparable to a query plan from a classical query evaluation that specifies the sequence in which the relations are joined. Variable marginalisation is used to perform aggregation in the F-IVM system. F-IVM utilises the in-memory-hash-based approach to implement equality-based joins to ensure high performance [3].

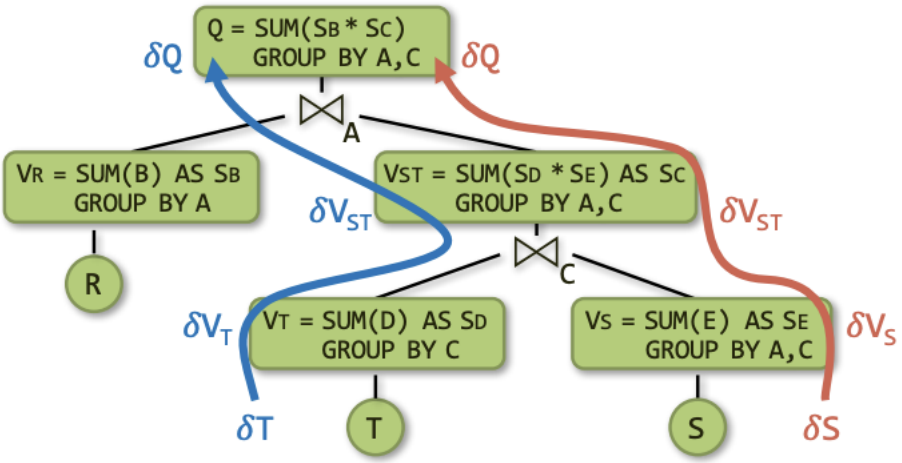

Figure 2.1: The View tree for the query in Example 1. Source [3]:

2.5 Apache Flink

Apache Flink is an open-source distributed processing framework that supports batch and stream processing, including bounded and unbounded streams [2]. Flink is designed to run in a cluster architecture that consists of three main components [10]:

- JobManager - is one of two parts of Flink's runtime that coordinates the distributed execution of Flink programs. It is responsible for scheduling tasks, managing finished or failed tasks, and coordinating checkpoints and recovery from failure. The JobManager compromises three main indigents: ResourceManager, Dispatcher and JobMaster [10].
- TaskManager - also called a worker, is the second part of Flink's runtime that executes tasks, where each task is executed by one thread [10].
- Client - is not a part of Flink's runtime, but it assembles and sends a dataflow to the JobManager. Once the client sends a dataflow, it can stay in attached mode or disconnect [10].

A Flink application takes the form of a directed graph of operators (described in Section 2.5.1), which transform data streams according to the needs. Flink provides three main APIs: DataStream, Table API and SQL. In the past, Flink offered a DataSet API for batch processing, but since version 1.12, this API has been deprecated, and its functionalities have been moved to DataStream API [10].

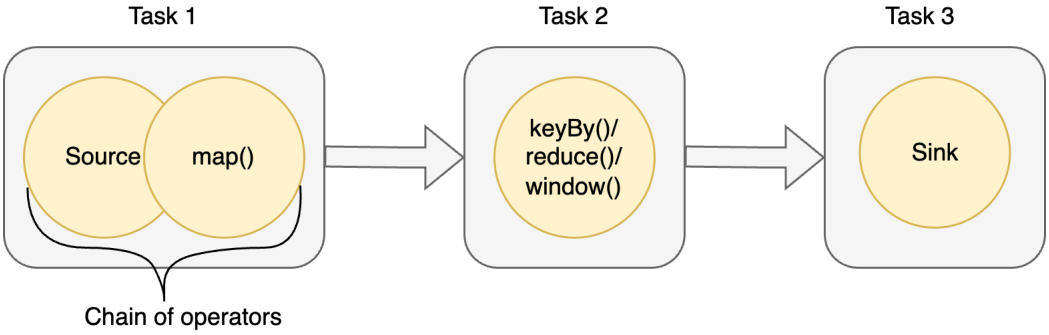

Figure 2.2: Flink's directed graph of operators. Source [10]

2.5.1 Data and Execution Model

The client transforms an application code into a chain of tasks and operators that form a directed dataflow graph presented in Figure 2.2. Flink optimises execution by putting a chain of operators into one task to reduce the overhead of buffering and exchanging data between threads, resulting in increased throughput and decreased latency [10].

Operators can transform an input stream by applying built-in or user-defined functions. They can combine different streams or fan out a single stream into multiple streams. Operators can be stateful or stateless [10].

Stateless operators do not keep previous events and only apply a specific function to the current event. Examples of such operators are a where operator from Table API or filter from DataStream API [10].

Stateful operators accumulate previous events and perform a transformation on a set of events. There are built-in operators, such as join from the DataStream API or groupBy from the Table API, and users can implement their stateful operators [10].

2.5.2 Working with Time

In many applications, such as time series analysis, time plays a significant role, and for these applications, Flink introduces a timely stream processing. Flink supports two notions of time: processing time, which is the system time of the machine performing the given operation, and event time, which is the time embedded within the records prior to their ingestion into Flink [10].

Flink uses watermarks to measure progress in event time. Watermarks track the order of events as they carry a timestamp throughout the data stream. When an event with a watermark t reaches an operator, it tells the operator that there should be no more

events with a lower watermark than t [10].

Supporting stateful operations such as aggregates over unbounded streams the same way as with bounded streams might not be possible as it eventually requires infinite resources. Therefore, Flinks uses windowing to tackle this problem. Instead of performing operations on the entire stream, Flink can perform these operations on the windows of this stream. A window is a subset of events that can be grouped by time (e.g. every 5s) or count (e.g. every 30 records). Flink distinguishes three types of windows: sliding windows (with overlaps), tumbling windows (without overlaps) and session windows (punctuated by a period of inactivity) [10].

2.5.3 DataStream API

The DataStream API is the core API that supports both finite and unbounded streams, and it gets its name from the `DataStream` class that represents a collection of data in a Flink application [10]. It supports two execution modes: streaming and batch. The streaming mode is default and is dedicated to unbounded streams, whereas the batch mode is used for bounded streams [10].

The DataStream API have a handful of built-in operators, such as `map`, `flatMap` or `keyBy`, and it also supports creating custom operators by extending one of the core classes, such as `FlatMapFunction`, `ProcessFunction` or `CoProcessFunction`. The first two classes support single-stream processing, whereas the `CoProcessFunction` processes two streams, which enables implementing a custom join operator [10].

The DataStream API supports stateful processing by introducing keyed and operator states. The operator state is associated with a single instance of a parallel operator, and a function must implement the `CheckpointedFunction` interface to use this state. The keyed state only supports keyed DataStreams, which are created with the `keyBy` operator, and it provides five primitive states: `ValueState`, `ListState`, `ReducingState`, `AggregatingState` and `MapState`. Flink manages a separate state for every key in a data stream [10].

The DataStream API distinguishes windows for keyed streams and non-keyed streams. However, these two types work similarly on the surface and support tumbling windows, sliding windows, session windows and global windows, where the global windows only work with a custom trigger[10].

2.5.4 Table API & SQL

Table API and SQL are two highly optimised relational APIs for stream and batch processing. The Table API is a language-integrated query API that enables constructing SQl queries from relational operators such as projection, selection and join. The Flink SQL offers more high-level support of SQL queries based on Apache Calcite. Flink also features an SQL client where users can SQL queries without a single line of Java or Scala code. Both APIs provide a seamless integration with one another and the DataStream API [10].

Table API offers a handful of operators known from SQL, such as `groupBy`, `where`, `distinct` and `join`, among others. Both APIs support windowing and provide a set of built-in functions for data transformations, but users can define custom functions by creating classes that extend the core classes, such as `ScalarFunction` and `TableFunction`. Flink provides options to tune the performance of Table API and SQL applications. One such option is MiniBatch aggregation, which allows collecting a group of records before performing aggregation instead of processing input records one by one, which is a default behaviour. This option mimics batch processing and can significantly increase performance [10].

Chapter 3

Design and Implementation

This chapter dives into how the application is implemented and what are the system requirements for our application. We show how we enable the F-IVM's transformations by using Flinks operators to reduce the time complexity of query execution compared to the standard Flink SQL execution that does not support these optimisations, such as factorisation and incremental processing.

The chapter starts by highlighting and explaining the system requirements that our application should meet. Next, in the Design section, we show how the application is structured, its components and the main algorithm behind this application. The chapter ends with the Implementation section, where we highlight the limitations of both solutions and explain how we implement our two solutions: the DataStream and the Table API solution.

3.1 System Requirements

The goal of this project is to adapt methods from the F-IVM system that optimise the execution of SQL queries to improve the current performance of Flink SQL applications. Our application should fulfil the following requirements to accomplish the main goals:

- Compatibility: The application should support operations and methods performed by the F-IVM system, such as a multi-join operator or a group-by operator.
- Schema agnostic: Our application should not rely on a predefined/static table schema and allow users to specify their table schemas to support various types of SQL queries.

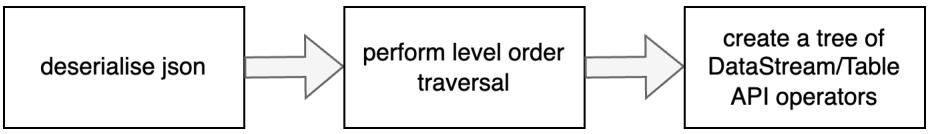

Figure 3.1: Main logic of the program

- Type agnostic: Type agnostic: Our solution should support various data types and enable a smooth introduction of new data types into the system.
- Performance: The implementation should improve the performance of the standard execution of Flink SQL programs and be comparable with the execution of the F-IVM system.

3.2 Design

Our application requires two primary input arguments: a JSON file representing a tree of views and a path specifying where the results should be written. In addition, we can specify a flag to decide which solution we want to use as we provide two different solutions: one based on the DataStream API and the other based on the Table API. Our decision to offer two solutions is motivated by the desire to compare these APIs across various aspects, such as performance, complexity, and flexibility.

However, regardless of the solutions, the main logic of the application stays the same. Figure 3.1 shows the three main parts of the application. First, a JSON file is deserialised into a Java object representing a tree of views. The F-IVM system produces a .M3 file that represents a tree of views that can be translated into the JSON format we use in our solution. At the moment, the translation is done manually, but there is a possibility of adding an add-on to the F-IVM system that will produce the JSON files automatically. We explain the structure of the JSON file in the Implementation section.

Once the JSON file is deserialised, we perform a level-order traversal to create a list of lists where each list contains nodes from that level. We need to construct a list of level-ordered nodes because a chain of operators that represents transformations of streams is created from leaf nodes to a root node, and a parent node is a transformation of its children nodes.

As the last step, our application creates a chain of operators from the list of level-ordered nodes presented by the pseudocode in Algorithm 1. The algorithm loops through the levels of a tree, starting from the bottom and for each node, it creates

Algorithm 1 The algorithm creating a chain of operators

```
root ← deserialise(jsonFile)
levels ← createListOfLevelOrderedNodes(root)
views ← new map()
for level in reversed(levels) do
    for node in level do
        if node is sourceNode then
            view ← createSourceView(node)
            views.put(node.name, view)
        else
            view ← createView(node, views)
            views.put(node.name, view)
        end if
    end for
end for
```

a view represented by Flink's operators. If a node is a source node, it constructs a connector to a file that represents the given table, and from the connector, it creates a Table/DataSource that represents this node. Otherwise, the view is constructed by transforming its children's views. We add the views to the map to reference them when we create their parent views. The final view, which is the root node, saves the query results to a file in the directory specified by the input argument. Example 3.2.1 shows how the Algorithm 1 works with the tree of views from Figure 2.1.

Example 3.2.1. Explanation of Algorithm 1 on the tree of views from Figure 2.1.
In the first iteration, the nodes *T* and *S* are translated into source views. The next iteration translates the *R* node into a source view, the node *Vt* into a view that calculates the aggregation by transforming the corresponding source view using the appropriate operator, and the node *Vs* to a view that calculates the aggregation similarly to the node *Vt*. The third iteration translates the *Vr* node in a similar way to the *Vt* and *Vs* nodes, and the node *Vst* is created by joining the views that correspond to the *Vt* and *Vs* nodes and performing appropriate aggregation. The last iteration creates the final view that saves results into a file by joining the views corresponding to the *Vst* and *Vr* nodes and performing appropriate aggregation.

3.3 Implementation

The structure that represents a tree of views contains the following information about each view: the view name, names of output columns, data types of output columns, list of operations performed in this view, references to its children's views and a boolean value indicating if this view is a source node. If a view is a source node, it also contains a path to a .csv file that represents a table, and then the names and data types of output columns represent a schema of this table, and the list of operations is empty.

The F-IVM system supports single aggregation queries that contain a sum operation and transforms them into a tree of views where each performs aggregation. Therefore, our application should implement the following operators to support the optimisations performed by the F-IVM system:

- Aggregation: One of the main optimisations of the F-IVM system is pushing aggregation past joins. Thus, every source node's parent view performs aggregation. By default, Flink processes records one by one, which is inefficient for aggregation tasks, as for every input tuple in the operator, the operator outputs the updated aggregation value of this tuple's group. However, if we pass a batch of tuples instead of a single tuple to the aggregation operator, we can lower the number of tuples that the operator outputs. This is because the operator only outputs an updated aggregation value for each group in the batch rather than for every tuple, as multiple tuples can be in the same aggregation group.

- Multi-Join: Since the F-IVM system relies on pushing aggregation past joins, there is an opportunity to perform an efficient join of more than two streams when the aggregations pushed past joins contain the same group-by columns for every joined stream, as there is always at most one matching tuple in every stream.

- Join: In many cases, a parent's view joins its left and right children and performs aggregation. Thus, it is possible to combine join and aggregation into a single join-aggregate operator.

- Where: The F-IVM system supports a WHERE clause that can contain conjugation. Therefore, our application should also enable selection.

The following subsection introduces the DataStream solution and explains in detail how we create custom operators and the optimisation we introduce to improve performance. Next, we introduce our Table API solution and explain how we use the

built-in operators to represent a tree of views. We finish this section by highlighting the limitations of our solutions.

3.3.1 DataStream Solution

The DataStream solution uses the DataStream API to implement its three main operators: aggregation, multi-join and join-aggregate. It maps nodes (views) into these operators based on the node type and number of children nodes the node has. If a node is a source node, then the node is mapped into a source view. When a node has one child, it is translated into the aggregation operator as its child is a source node. In case a node has two children, the solution maps it into the join-aggregate operator. Finally, when a node has more than two children, it is translated into the multi-join operator. This solution also introduces batching between source views and aggregation operators by leveraging time windows.

3.3.1.1 Source Views

A source view is created using a `FileSource` connector, and in order to use time windows, we assign a watermark to each record from the file. We utilise the `Row` type provided by Flink to handle various data types and to access fields by their column names. Otherwise, we would need to create a new class to handle these aspects.

3.3.1.2 Time Windows

We implement batching by introducing custom windows based on tumbling event time windows. A batch is usually associated with a set of records of fixed size, and Flink provides such options with count windows. However, there is one major problem with count windows in Flink. When a task finishes and a count window is not fully filled, the remaining records in the window are not processed, and the job finishes without processing these remaining records. However, this is not the case with time windows because when the last record from a file is read, a watermark with a maximum timestamp is assigned to this record, which causes a window to complete and pass records to the next operator.

Although relying solely on time windows could be sufficient, we wouldn't have any control over the size of batches. Therefore, we implement a custom trigger that works based on time windows but also keeps control over the maximum length of a window. If the size of a window reaches its maximum limit or a record with a timestamp older

than the maximum allowed timestamp for this window enters, the records from this window are pushed to the next operator. Listing 3.1 shows how windowing is added between a source view and the aggregation operator.

Listing 3.1: Windowing

```
1 ...
2 DataStream<Row> child = ... //a child view of the node
3 child.windowAll(TumblingEventTimeWindows.of(Duration.ofMillis(500)))
4     .trigger(new CountOrTimeTrigger(100000))
5     .process(windowedAggregate);
```

3.3.1.3 Aggregation Operator

DataStream API does not contain any operators that can perform group-by and sum operations. Thus, we implement a custom aggregation operator that extends the `ProcessAllWindowFunction` class. We extend this class as we work on time windows to optimise aggregation by limiting the number of tuples that the operator outputs.

In this custom operator, whose pseudocode is shown in Listing 3.2, we keep a `HashMap` that has a global state (independent from a current window), and it maps groups to their aggregation values, shown in lines 11 to 15. In addition, for a current window, we create a `Set` at line 8 to keep track of groups that are in this window and eliminate duplicates. This way, in lines 19 to 23, we only output tuples with the final aggregation values for every group in this window, which limits the number of tuples that the operator outputs.

Listing 3.2: Aggregation operator

```
1  public class WindowedAggregate extends
2          ProcessAllWindowFunction<Row, Row, TimeWindow> {
3      private Map<Row, Double> aggregation;
4      ***
5      public void process(Context context, Iterable<Row> iterable,
6          Collector<Row> collector) throws Exception {
7          Set<Row> processed = new HashSet<>();
8          for(Row tuple: iterable) {
9              if(checkWhere(tuple)){
10                 Row group = getGroupByColumn(tuple);
11                 Double sm = calculate(tuple);
```

```
12                  if(aggregation.containsKey(group))
13                      sm += aggregation.get(group);
14                  aggregation.put(group, sm);
15                  processed.add(group); // get rid of duplicated groups
16              }
17          }
18          for(Row group: processed) {
19              Double sm = aggregation.get(group);
20              Row output = createOutputTuple(group, sm);
21              collector.collect(output);
22              }
23      }
24      ***
```

3.3.1.4 Join-Aggregation Operator

Every view that is not a source node performs aggregation. Thus, it is possible to combine a join operation and a following aggregation operation into a single custom operator, and the join-aggregation operator does that. Combining these two operations into a single operator reduces the communication time that would otherwise be required if they were separate.

We create a `JoinAggregate` class that extends `CoProcessFunction` to implement this operator. Our operator utilises a hash join approach over other approaches like nested-loop or merge joins to provide the best performance in terms of time complexity. In addition, this operator implements aggregation in a similar fashion to our aggregation operator but without using time windows, as Flink does not provide a class like `CoProcessFunction` that supports windowing. Listing 3.3 shows how the join-aggregation operator is involved.

Listing 3.3: Join-aggregation operator

```
1 JoinAggregate joinAggregate = new JoinAggregate(leftChildColumns,
2                     rightChildColumns, outputColumns, groupBy,
3                     sumComponents, node.getJoin());
4 DataStream<Row> left = ...// get view of the left child;
5 DataStream<Row> right = ...//get view of the right child ;
6 return left.connect(right).process(joinAggregate);
```

3.3.1.5 Multi-Join Operator

Flink does not allow joining more than two streams in a straightforward way. Therefore, we utilise the `union` operator from the DataStream API to solve this task. But before performing a union operation on the streams we want to join, we map the `Row` type `DataStreams` to a custom `EitherOneOfN` type `DataStream` that allows us to perform the multi-join operation on the unioned streams later, which is shown in Listing 3.4 in lines 2 to 14. The `EitherOneOfN` wraps the `Row` type with an integer value indicating the stream from which a tuple comes. Then, at line 15, we perform mapping using a custom `JoinNStreams` class that extends the `RichFlatMapFunction` class. This operator groups the incoming records into groups based on their origin stream and then conducts the join operation. It outputs joined results of the `Row` type. Listing 3.4 illustrates the process of mapping streams to the `EitherOneOfN` type and performing the multistream join.

Listing 3.4: Multi-join operator

```
1  ...
2  DataStream<EitherOneOfN> first = firstChildView.map(tuple -> new EitherOneOfN(
3                                  numberOfStreams, 0, tuple, aggColumn));
4  List<DataStream<EitherOneOfN>> children = new ArrayList<>();
5  for(int i = 1; i < numberOfChildren; i++) {
6      int position = i;
7      ...
8      DataStream<EitherOneOfN> mapped = temp.map(tuple -> new EitherOneOfN(
9                  numberOfStreams, position, tuple, aggregateColumn));
10     children.add(mapped);
11 }
12 JoinNStreams joinNStreams = new JoinNStreams(outputColumns, sumComponents,
13                                 node.getJoin(), numberOfStreams);
14 DataStream<EitherOneOfN>[] dataStreamArray = children.toArray(new DataStream[0]);
15 DataStream<Row> view = first.union(dataStreamArray).flatMap(joinNStreams);
```

3.3.2 Table API Solution

The Table API solution utilises more abstract built-in operators from the Table API, which are `select`, `join`, `groupBy` and `where`. Unlike the DataStream API, the Table API offers more abstract but less flexible relation operators. Thus, this solution does not offer a way that would allow us to perform a multi-stream join operation, as we cannot implement custom operators here. Therefore, this solution does not cover queries that require joining more than two streams in a single operation. Akin to the DataStream solution, this solution relies on a level order traversing and mapping nodes to particular operators. If it is a source node, a source table is created. When a node has only one child, then the `groupBy` operator followed by the `select` operator is used and if the node has a where condition, the `where` operator is used additionally. Finally, if a node has two children, the `join` operator is used, followed by the `where`, `groupBy` and `select` operators.

3.3.2.1 Source Tables

We create source tables using a method the `StreamTableEnvironment` class. This method creates an in-memory table from a .csv file using a schema that contains table names and their data types as one of the arguments. Listing 3.5 shows how the source tables are created from the source nodes.

Listing 3.5: Source tables

```
1 StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
2 ...
3 Schema schema = SchemaUtils.getSchema(node);
4 tableEnv.createTable(tableName, TableDescriptor.forConnector(”filesystem”)
5         .option(”path”, tablePath).schema(schema)
6         .format(”csv”)
7         .build());
8 Table sourceTable = tableEnv.from(tableName);
9 ...
```

3.3.2.2 MiniBatch Aggregation

Similar to the time window approach used in the DataStream solution, we introduce batching here to improve significantly the performance of our application. We utilise

the mini-batch aggregation settings to perform aggregation operations based on a batch of tuples rather than on single tuples, which highly reduces the number of tuples that are passed to the next operators and the time that is spent on serialisation and deserialisation of records.

3.3.2.3 Joining Two Tables

The join operator in the Table API allows joining two tables, but the join column in each table must have different names. However, the F-IVM system works with natural joins, which means it performs join on the columns with the same name. Thus, before performing a join, we need to rename one of the columns to enable the join. To join tables, we use the built-in `join` operator followed by the `where` operator. Listing 3.6 shows how the join operation is performed.

Listing 3.6: Joining two tables

```
1 ...
2 String newJoinColumnName = node.getJoin() + "_r";
3 Table left = ... // get a left table
4 Table right = ... // get a right table
5 Expression [] expressions = ... // get renamed right table names
6 right = right.select(expressions); // rename the join column
7 Table result = left.join(right).where($(node.getJoin()).isEqual($(newJoinColumnName)));
```

3.3.2.4 Aggregation

The aggregation operation is performed using two operators: a `sum` operator that is a part of the `select` operator and the `groupBy` operator. By default, the sum operator only performs sum on a single variable and cannot handle a sum that contains more variables like SUM(S.A * S.B). Thus, we implement a `Multiply` class to handle this scenario. The `Multiply` class extends `ScalarFunction`, and its `eval` function returns a multiplication value of its input arguments. We convert output column names of `string` types to `Expression` types used by Table API operators at line 2 in Listing 3.7, and by convention, we keep a sum column as the last column in the list. If the sum column involves multiplication, we call the custom `multiply` class. Otherwise, only the `sum` operator is used on the sum column. These operations are shown in lines 7 to 11. Listing 3.7 illustrates how the output columns with the sum are created and how the aggregation is performed.

Listing 3.7: Aggregation

```
1  ...
2  Expression [] groupByCols = Utils.stringsToExpressions(groupByColumns);
3  Expression [] selects = Utils.stringsToExpressions(outputColumns);
4  if (ViewUtils.containsSum(node) != -1){
5      Operation sum = node.getOperations().get(ViewUtils.containsSum(node));
6      String alias = node.getColumnNames().get(node.getColumnNames().size() - 1);
7      if(sum.getExpression().contains("*")){
8          Expression[] cols = Utils.stringsToExpressions(sum.getExpression().split("\\*"));
9          selects[selects.length-1] = call("multiply", cols).sum().as(alias);
10     } else {
11         selects[selects.length-1] = $(sum.getExpression()).sum().as(alias);
12 }
13 ..
14 return view.groupBy(groupByCols).select(selects);
```

3.3.3 Limitations

The current implementation of our solution has some limitations, which are caused by various reasons, including API limitations and time constraints. The limitations are:

- We do not support SQL queries that use the SUM aggregation with operators other than multiplication or have more than one SUM aggregation.
- The Table API solution does not support SQL queries containing the SUM aggregation with constant values other than one due to the Table API limitations.
- We only support the following operators in the WHERE clause: AND, EQUAL, NOT EQUAL, LESS THAN, LESS THAN EQUAL, GREATER THAN, and GREATER THAN EQUAL.
- The Table API only supports view transformations that contain up to one AND operator in a single view due to the Table API limitations.
- The Table API does not support view transformations that contain a join of more than two data streams in a single operation because of the Table API limitations.

Chapter 4

Experiments and Evaluation

This chapter evaluates the performance of the DataStream and Table API solutions and investigates the impact of batch optimisation on the performance of both solutions. We compare the throughput of these two solutions with the throughput of the standard Flink SQL execution and the F-IVM system. The experiments show:

- Significant improvement in throughput of our solutions compared to the standard Flink SQL execution on queries that do not contain highly selective WHERE clauses.
- Drastic improvement in throughput of the DataStream solution compared to the standard Flink SQL execution on queries that allow for the multi-join transformation. The increase in join relations does not impact the throughput of our solution but prevents the standard Flink SQL execution from executing queries in a reasonable time.
- The Table API solution outperforms the DataStream solution but is less flexible.
- The batch optimisation can significantly improve the throughput of both solutions on queries that do not have highly selective WHERE clauses.

The rest of this chapter dives into details of our experiments. We start by highlighting the settings of experiments and describing the datasets we use in our experiments. Then, we evaluate the performance of the custom join-aggregate operator and its equivalent in the Table API solution on two different datasets and five queries. Next, we assess the effectiveness of the multi-join operator on two different datasets and four queries. Then, the subsequent section analyses how the batching optimisation influences the performance of our solutions. The chapter ends with a detailed summary of our findings.

4.1 Settings

Experiments compare the performance of the DataStream and Table API solutions against the standard Flink SQL execution and the F-IVM system. We run experiments on Apache Flink version 1.19.0 and use the default batch in our solutions (100000) and the F-IVM system (1000) unless stated otherwise, and we conduct these experiments on the MacBook M1 Pro 16 GB. The experiments are performed on four datasets:

- Housing: it is a synthetic dataset similar to the one used by Nikolic et al. [3]. The dataset comprises 1800000 records and six relations: Demographics, House, Institution, Shop, Restaurant and Transport.
- Retailer: a synthetic dataset created in the likeness of the dataset used by Nikolic et al. [3]. This dataset consists of 900000 records and three relations: inventory, location and weather.
- TPC-H: it is a benchmark dataset [23] used for database evaluation. We create the dataset with a scaling factor of one, and we use five relations: Orders, Lineitem, Customer, Nation and Part.
- S-T-R: it is a synthetic dataset that the Example 1 query uses. The dataset has 2200000 records and three relations: S(A, C, E), T(C, D) and R(A, B). We generate the dataset by sampling random numbers between 0 and 10000.

We measure the performance of the applications using throughput, which we define as the number of records in the relations divided by the execution time of the program in seconds. The higher the throughput the better.

4.2 Join-Aggregate Evaluation

Experiments in this section aim to evaluate the performance of the custom Join-Aggregate operator and its Table API solution equivalent. We measure the performance of our solutions against the standard Flink SQL execution and the F-IVM system on five queries and two datasets. The first query we examine is the query from Example 1 on the S-T-R dataset. Then, we examine two queries with a highly selective WHERE clause that are slightly modified versions of the TPC-H 3* and TPC-H 14* queries to make them compatible with our solutions. The last two queries are somewhat modified versions of the TPC-H 10* and TPC-H 12* queries with removed WHERE clauses.

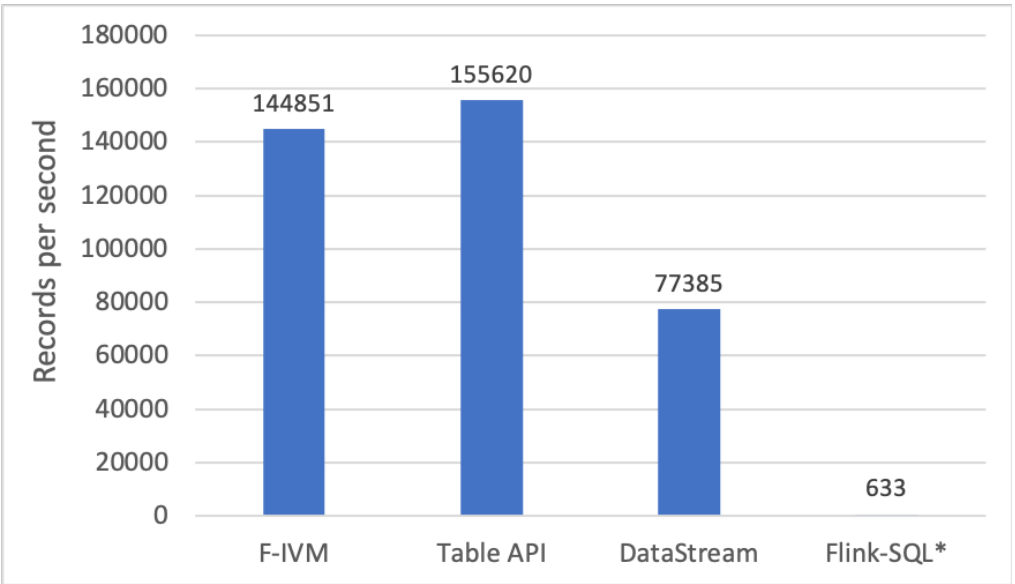

Figure 4.1: The throughput of processing the Example 1 query.

We do not run other TPC-H queries as they are either very similar to these queries or are not compatible with our solutions. The TPC-H queries are executed on the TPC-H dataset, and the queries can be found in Appendix A. We divide the TPC-H queries into two groups, one with a WHERE clause and one without it, to examine how the number of tuples in join operators affects the performance of the four solutions. Lastly, to investigate further how the selectivity of the WHERE clause impacts the throughput of our solutions and the standard Flink SQL, we run the TPC-H 12* query with four different selectivities of a predicate: 100% (the original TPC-H 12* query without a WHERE clause), 75%, 50% and 25%. The selectivity of the predicates here is the opposite of 'highly selective.' A selectivity of 100% means that every record matches the predicate, while a selectivity of 0% means that no records match the predicate. The TPC-H 12* query with these different selectivities and the method used to calculate selectivity are detailed in Appendix A. We run the TPC-H queries with the maximum batch size set to 5000 in the Table API and DataStream solutions.

Figure 4.1 shows the performance of the four applications on the Example 1 query. The standard Flink SQL program does not finish its execution, and its result represents the number of records processed in 30 minutes. The performance of the Table API matches the F-IVM system's performance, but the throughput of the DataStream solution is lower. However, its performance is still comparable with the performance of Table API and F-IVM.

The difference in performance between the Table API and the DataStream solutions

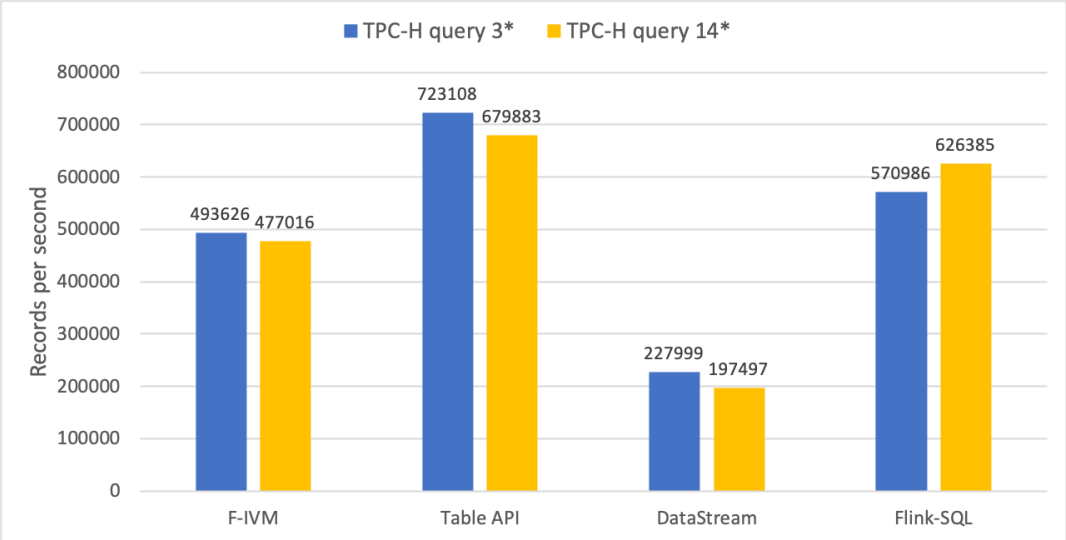

Figure 4.2: The throughput of processing the TPC-H 3* and 14* queries that have highly selective WHERE clauses.

might be due to at least two reasons. First, the Table API is highly optimised for relation tasks, and its operators are written specifically for such tasks. In the DataStream solution, we create custom operators that might not be as optimised as those in the Table API in terms of data serialisation/deserialisation and how the abstraction of different data types is handled. Second, the Table API performs mini-batch aggregation every time the aggregation is done. In contrast, the DataStream solution only performs the batch aggregation immediately after source nodes in the custom aggregation operator because the next aggregations are combined with joins in the custom join-aggregate operator, which prevents using time windows.

This experiment shows how pushing aggregation past joins and performing incremental view maintenance can impact performance. The lack of support for these optimisations in Flink prevents it from executing such queries in a reasonable time.

Figure 4.2 shows the performance of these four programs on the two TPC-H queries that have highly selective WHERE clauses. The performance of the standard Flink SQL, the Table API solution and the F-IVM system is similar, but the DataStream solution's performance is lower. The reasons why the throughput of the standard Flink SQL execution is akin to others are the selectivity of the queries and, thus, the small number of join-matching tuples. The execution time of the F-IVM system confirms that hypothesis as over 95% of the time is spent on loading relations of size around 0.9 GB in both queries, which is not the case with the previous query, as the time spent on loading relations is negligible. The reason behind the lower performance of the

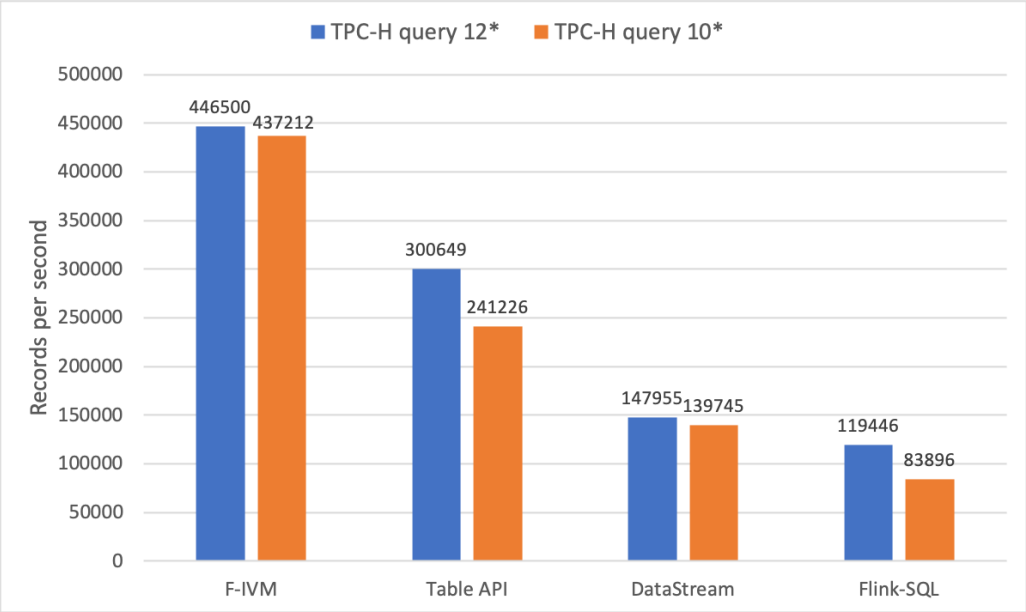

Figure 4.3: The throughput of processing the TPC-H 10* and 12* queries.

DataStream solution compared to the Table API solution can be the less optimised implementation of the custom operators, as the batching plays a marginal role because we lower the maximum batch size to 5000, and it does not impact performance because few tuples are in the same group-by group.

Figure 4.3 illustrates the throughput of the four applications on the TPC-H queries that do not have WHERE clauses. The throughput of the standard Flink SQL execution dropped significantly, making it the least performant solution. The throughput of the F-IVM system drops only marginally. Although the throughput of the Table API and DataStream solution decreases, it is not that significant, and both solutions outperform the standard Flink SQL execution.

The impact of the selectivity of the WHERE clause on the throughput of our solutions and the standard Flink SQL execution is shown in Figure 4.4. As we have a more selective WHERE clause, the throughput of standard Flink SQL execution rapidly increases, which is not the case with our solutions. This chart clearly shows the impact of the number of matching records on the standard Flink SQL compared to our solutions. When the selectivity of the predicate is 100%, the Flink SQL performs the worst. When we decrease the selectivity of the predicate to 50%, the Flink SQL outperforms the DataStream solution as a large portion of time is spent on loading relations and not computation in the case of the DataStream solution.

Figure 4.4 also illustrates how pushing aggregation past joins improves the perfor-

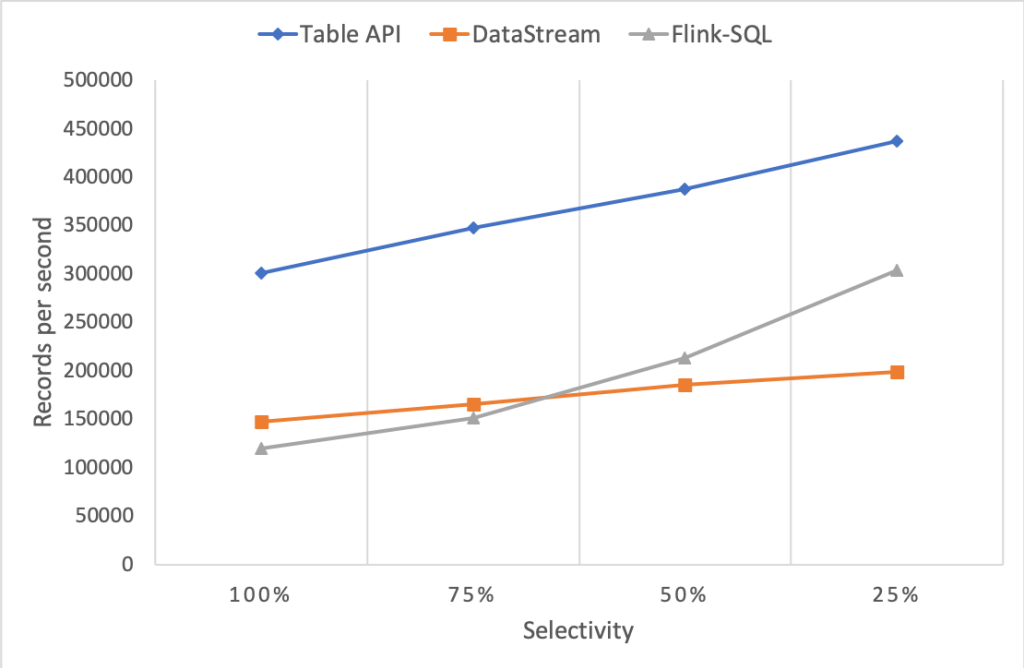

Figure 4.4: The throughput of processing the TPC-H 12* query with four different selectivities of a predicate.

mance of our solutions compared to Flink SQL. As we have a less selective WHERE clause, we have more matching tuples in the join operators, and with the pushed aggregation past joins, the number of matching records significantly decreases because the records are grouped before the join, which is not the case with the standard Flink SQL.

The experiments with the TPC-H queries and the selectivity of the WHERE clause show that as the number of matching tuples in the joins increases, the performance of the standard Flink SQL execution is increasingly affected, and the performance of the DataStream and Table API solution are more robust due to the pushing of aggregation past joins. The Example 1 query clearly shows it because the S-T-R dataset has only 10000 unique values, significantly less than the number of unique values in the join columns in the TPC-H dataset, which results in much more matching tuples in the joins that highly affects the standard Flink SQL execution.

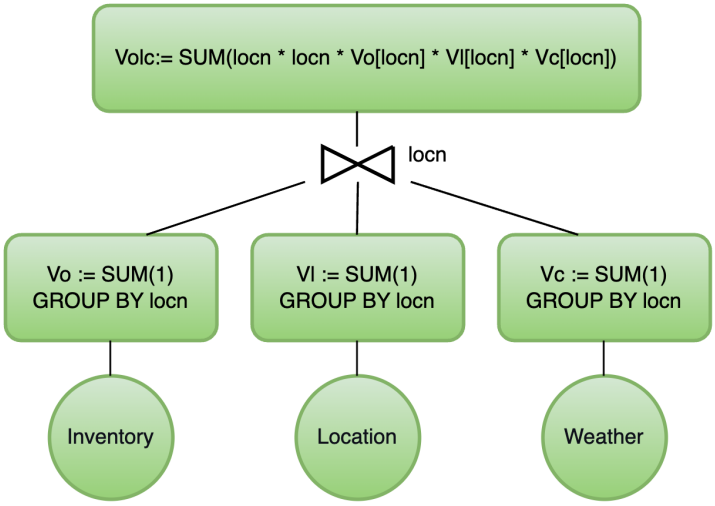

Figure 4.5: The tree of views for the retailer query.

4.3 Multi-Join Evaluation

In these experiments, we seek to assess the performance of the multi-join operator from the DataStream solution against the standard Flink SQL execution and the F-IVM system. These experiments do not evaluate the Table API solution since it lacks the support for the multi-join operation. We measure the performance of these three applications based on four queries that can be found in Appendix A:

- The first query joins three relations of the Retailer dataset based on the locn column and calculates the sum of the square values of the join column. Although that query might not look sensible at first, it can have use cases in other areas like machine learning.

- The next three queries calculate the sum of squares of the postcode column, which also is the join column of six relations of the Housing dataset. The queries differ in the number of joins: the second query joins four relations, the third joins five, and the fourth joins six. As with the first query, these queries might not look sensible at first, but they can have use cases in other areas like machine learning.

We selected the four queries with different numbers of join columns to demonstrate how increasing the number of join relations impacts the performance of our solution and the F-IVM system compared to standard Flink SQL.

Figure 4.6 shows the throughput of the three solutions. The F-IVM system achieves the best performance, followed by the DataStream solution, which has around two

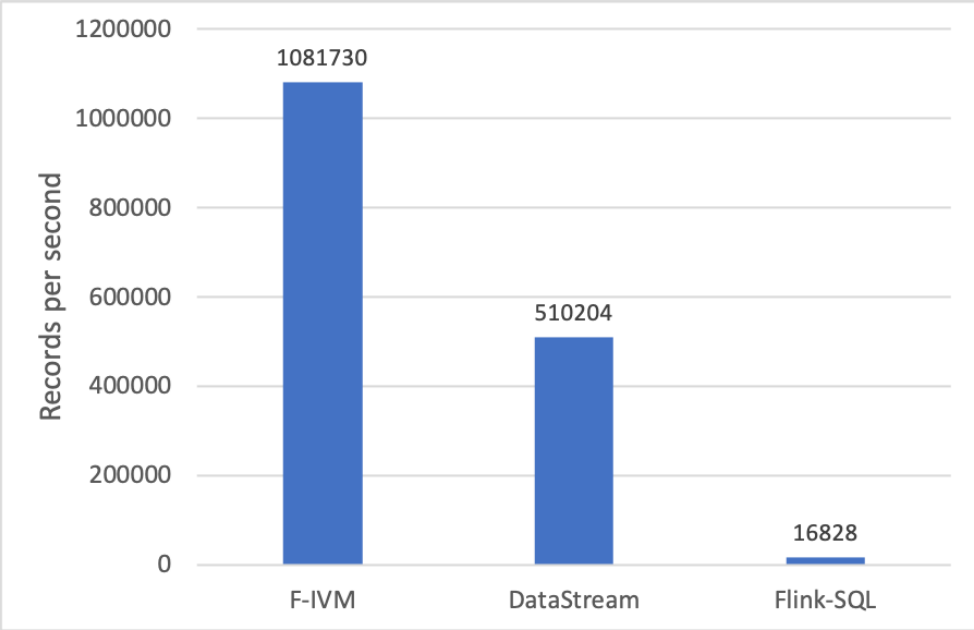

Figure 4.6: The throughput of processing the retailer query.

times lower throughput. The standard Flink SQL execution places last with over 30 times lower throughput than the DataStream solution. The results clearly show how the query transformation shown in Figure 4.5 impacts performance. The transformation introduces aggregation before the join that counts the same values of the locn column and groups by this column. This aggregation enables joining all three relations in the single multi-join operator.

Figure 4.7 illustrates the performance of the three systems on the housing queries that contain four to six join relations. The increase in the number of join relations does not negatively impact the throughput of the F-IVM and the DataStream solution. However, the impact on the standard Flink SQL execution is very high as Flink SQL does not finish processing any of these queries in a reasonable time, and its throughput represents the number of records processed in 30 minutes. This experiment highlights the impact of query transformations and joining many tables in a single operation and how the lack of support for these optimisations affects Flink's performance.

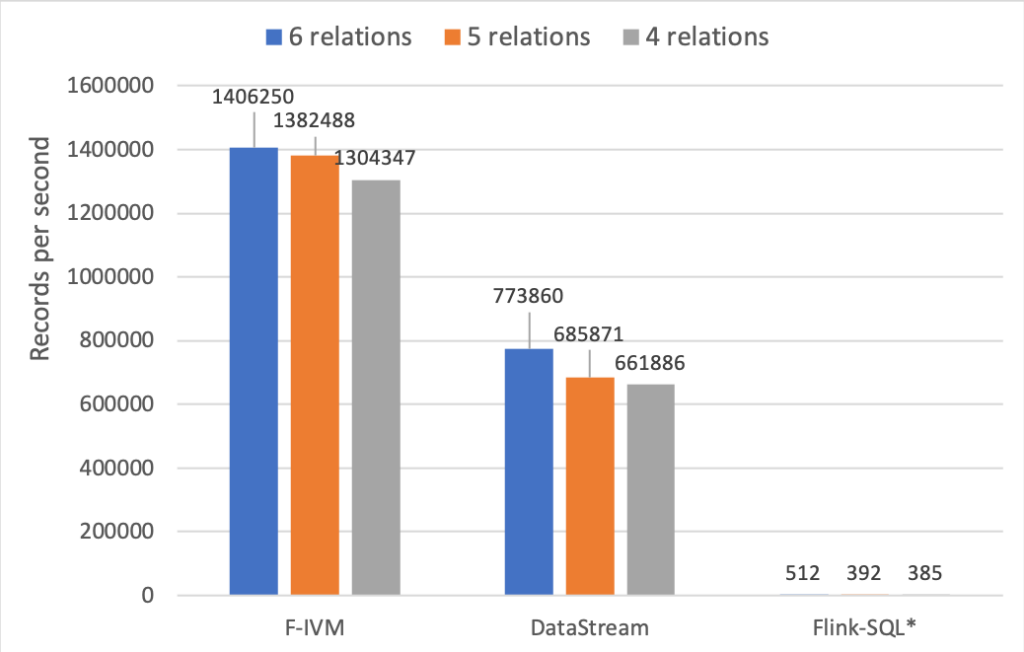

Figure 4.7: The throughput of processing the housing queries.

4.4 Batch Optimisation

This section seeks to evaluate how the introduction of batching optimisation impacts the performance of the DataStream and Table API solutions. By the batching optimisation, we refer to the use of time windows in the DataStream solution and mini-batch aggregation in the Table API solution. We measure the impact on performance by running two queries with four different maximum batch sizes: 1000, 5000, 10000 and 100000. The first query is the Example 1 query, which we run on the subset of the S-T-R dataset containing 500000 records. The second query is a slightly modified version of the TPC-H query 14 with a highly selective WHERE clause. We perform the experiment using two queries, one with a highly selective WHERE clause and the other without, to assess how batch optimisation impacts performance in these two scenarios.

Figure 4.8 highlights the impact of the batch size on throughput when there are many tuples that can be aggregated to lower the number of tuples that are passed to the next operators. The results indicate a significant decrease in throughput as the batch size decreases in both solutions, with the Table API solution being slightly more influenced by the change in the batch size. The outcomes show how batching optimisation improves the performance of both solutions and that without the optimisation, the throughput would considerably decrease if the solutions processed data record by record.

However, 4.9 shows the impact of the batch size on throughput when there is a

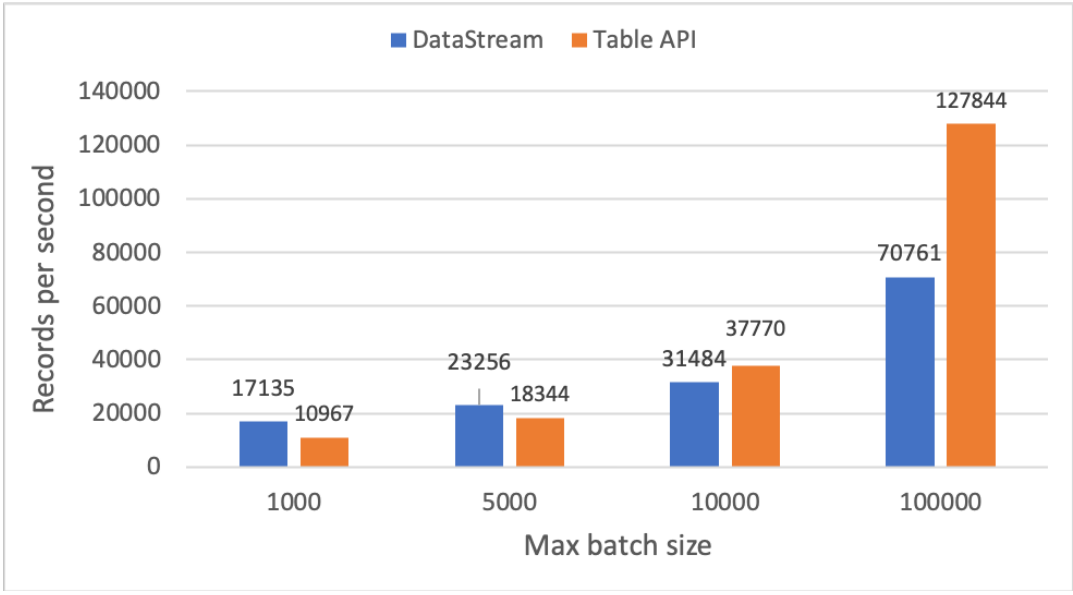

Figure 4.8: The impact of the batch size on the throughput on the Example 1 query.

highly selective WHERE clause that limits the number of tuples to aggregate. Thus, the batch size does not affect the throughput in both solutions because there are few tuples to aggregate. Therefore, the benefits that come from this optimisation are marginal.

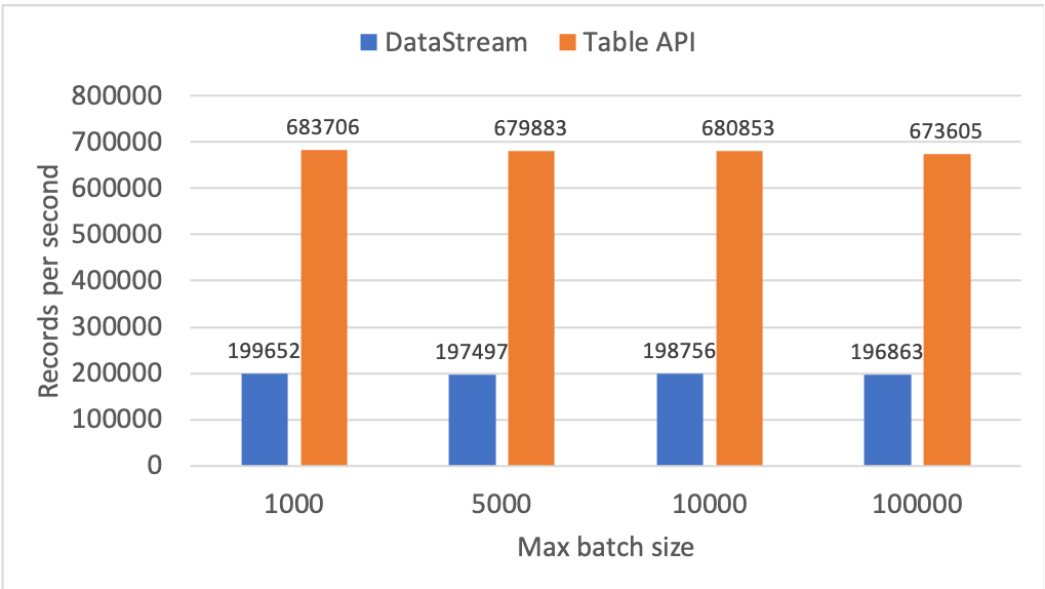

Figure 4.9: The impact of the batch size on the throughput on the TPC-H 14* query.

4.5 Overview of Findings

The experiments clearly show how our two solutions significantly improve the performance of Flink's applications compared to the standard Flink SQL execution in the queries that involve joins of a large number of records. The Flink SQL cannot process the housing and S-T-R queries within a reasonable period, while our two solutions process these queries in a comparable time to the F-IVM system. In the case of queries with highly selective WHERE clauses that have few matching tuples, the performance of the standard Flink SQL execution is comparable with the performance of our solutions and the F-IVM system.

The Table API solution is more performant than the DataStream solution for at least two reasons, which are the more optimised implementation of the built-in Table API operator than our custom operators and the fact that the batching optimisation is used with every aggregation operation in the Table API solution, which is not the case in the DataStream solution.

The experiments also demonstrate how our batching optimisation impacts the performance of both solutions, depending on whether a query includes a highly selective WHERE clause. When a query has many records to aggregate, the batching optimisation significantly increases throughput in both solutions. However, when there is a highly selective WHERE clause that greatly lowers the number of records to aggregate, then the batching optimisation does not impact the throughput of both solutions.

Chapter 5

Conclusions

Our project aimed to extend the current capabilities of Apache Flink by introducing support for the query transformations performed by the F-IVM system. The transformed version of queries significantly reduces the time complexity of their execution compared to the standard Flink SQL execution in many SQL queries that involve sum aggregation. Thus, a more efficient execution of these queries can significantly reduce costs, as one of the largest costs in companies' cloud spending is related to computational resources.

To allow support for the transformed queries by the F-IVM system, we propose two solutions based on the DataStream API and Table API. We decided to introduce two solutions based on these two APIs to compare their performance, flexibility and simplicity of implementation. Our solutions take a JSON representation of a tree of views that can be effortlessly created from an output file of the F-IVM system. At the moment, the transformation to the JSON file is done manually. However, it is possible to develop an add-on to the F-IVM system that will automatically generate this file.

The DataStream solution involved implementing three custom operators that perform aggregation, joining more than two streams in a single operator and joining two streams combined with aggregation. In addition, we create a custom class that evaluates WHERE clauses. The Table API solution involves mapping F-IVM's transformations into the built-in operators. Although the Table API solution enables faster implementation because we use the built-in operators, it lacks flexibility. The limitations of the Table API prevent us from supporting the F-IVM transformation that involves a join of more than two views. Therefore, this solution does not support queries that require that transformation. Furthermore, the Table API prevented us from supporting queries that involve a sum aggregation with a constant value that is different from one.

The experiments show that our solutions outperform the standard Flink SQL exe-

cution when queries do not have highly selective WHERE clauses and produce many join-matching tuples. However, for queries that involve highly selective where clauses, the performance of standard Flink SQL is comparable with the performance of our solutions. When queries allow joining many tables in a single operator, the DataStream solution significantly outperforms the standard Flink SQL execution. The number of join relations, in this case, does not affect the performance of the DataStream solution, but the standard Flink SQL execution cannot execute queries that involve more than three relations in a reasonable time. The experiments show that the Table API solution is more performant than the DataStream solution and that our batch optimisation highly improves throughput for queries that do not have highly selective WHERE clauses.

We believe that our system offers a significant attribution to Apache Flink that enables more efficient and performant execution of SQL queries that involve sum aggregation than the standard Flink SQL execution. With the increasing volume of processed data, efficient data processing becomes more and more crucial for reducing costs associated with computational resources. Our system tackles this problem by providing F-IVM's optimisations to Apache Flink, which results in a more efficient SQL query execution.

5.1 Future Work

For future work, we recommend adding an add-on to the F-IVM system that will produce JSON input files to automate the creation of input files. Further, we recommend extending support for more WHERE operators to extend the range of queries our system can support. In addition to that, expanding support for sum aggregation that involves different operations than multiplication will be highly beneficial. On top of that, introducing optimisation for hierarchical queries that involve the `keyBy` operator in the DataStream solution can significantly improve performance for hierarchical queries. Lastly, we recommend further testing of our solutions on datasets that involve many gigabytes of data.

Bibliography

[1] Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorization benefits. SIGMOD '18, page 365–380, 2018.

[2] Apache Software Foundation. Apache flink. https://flink.apache.org/.

[3] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. F-ivm: analytics over relational databases under updates. *The VLDB Journal*, November 2023.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[5] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, and et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[6] Boston Consulting Group. A new architecture to manage data costs and complexity. https://www.bcg.com/publications/2023/new-data-architectures-can-help-manage-data-costs-and-complexity.

[7] OpenAI. Chat gpt. https://chat.openai.com/.

[8] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, may 1998.

[9] Yanhong A Liu. Efficient computation via incremental computation. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, volume 1574, pages 194–203, 1998.

[10] Apache Software Foundation. Apache flink v1.19 documentation. https://nightlies.apache.org/flink/flink-docs-stable/.

[11] Jayanth Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–354, 2005.

[12] Fabian Hueske and Vasiliki Kalavri. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. O'Reilly Media, Inc., 2019.

[13] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, jun 1986.

[14] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, page 577–589, 1991.

[15] Christoph Koch. Incremental query evaluation in a ring of databases. PODS '10, page 87–98, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *CoRR*, abs/1207.0137, 2012.

[17] Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. SIGMOD '16, page 511–526, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. F-IVM: learning over fast-evolving relational data. *CoRR*, abs/2006.00694, 2020.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[20] Daniel J. Abadi, Don Carney, Ugur Çetintemel, and et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, aug 2003.

[21] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, aug 2017.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.

[23] TPC. Tpc website. `https://www.tpc.org/`.

Appendix A

SQL queries used in the experiments

Listing A.1: The slightly modified TPC-H 3* query

```
SELECT orderkey, o_orderdate, o_shippriority,
SUM(l_extendedprice*l_discount)
FROM CUSTOMER NATURAL JOIN ORDERS NATURAL JOIN LINEITEM
WHERE c_mktsegment = 'BUILDING' AND o_orderdate < '1995-03-15'
AND l_shipdate > '1995-03-15'
GROUP BY orderkey, o_orderdate, o_shippriority;
```

Listing A.2: The slightly modified TPC-H 10* without a WHERE clause

```
SELECT custkey, c_name,
       c_acctbal,
       n_name,
       c_address,
       c_phone,
       SUM(l_extendedprice * l_discount)
FROM customer NATURAL JOIN orders
NATURAL JOIN lineitem NATURAL JOIN nation
GROUP BY custkey, c_name, c_acctbal, c_phone, n_name, c_address;
```

Listing A.3: The slightly modified TPC-H 12* query without a WHERE clause

```
SELECT l_shipmode, SUM(o_totalprice)
FROM lineitem NATURAL JOIN orders
GROUP BY l_shipmode;
```

Listing A.4: The slightly modified TPC-H 14* query

```
SELECT p_brand, SUM(l_extendedprice * l_discount)
FROM lineitem NATURAL JOIN part
WHERE l_shipdate >= '1995-09-01'
  AND l_shipdate < '1995-10-01'
GROUP BY p_brand;
```

We measure the selectivity based on the l_shipmode column, whose values range from '1992-01-02' to '1998-12-01'. Therefore, the selectivity of 75% covers 75% of dates from this range, the selectivity of 50% covers 50% of dates from this range and the selectivity of 25% covers 25% of dates from this range.

Listing A.5: The slightly modified TPC-H 12* query with a selectivity of 75%

```
SELECT l_shipmode, SUM(o_totalprice)
FROM lineitem NATURAL JOIN orders
WHERE l_shipdate >= '1993-05-01'
GROUP BY l_shipmode;
```

Listing A.6: The slightly modified TPC-H 12* query with a selectivity of 50%

```
SELECT l_shipmode, SUM(o_totalprice)
FROM lineitem NATURAL JOIN orders
WHERE l_shipdate >= '1995-01-01'
GROUP BY l_shipmode;
```

Listing A.7: The slightly modified TPC-H 12* query with a selectivity of 25%

```
SELECT l_shipmode, SUM(o_totalprice)
FROM lineitem NATURAL JOIN orders
WHERE l_shipdate >= '1996-05-01'
GROUP BY l_shipmode;
```

Listing A.8: The three relations join query on the Retailer dataset

```
SELECT SUM(locn * locn) FROM INVENTORY
JOIN LOCATION ON INVENTORY.locn = LOCATION.locn
JOIN WEATHER ON INVENTORY.locn = WEATHER.locn;
```

Listing A.9: The six relations join query on the Housing dataset

```
SELECT SUM(HOUSE.postcode*HOUSE.postcode) FROM HOUSE
NATURAL JOIN SHOP NATURAL JOIN INSTITUTION
NATURAL JOIN RESTAURANT NATURAL JOIN DEMOGRAPHICS
NATURAL JOIN TRANSPORT;
```

Listing A.10: The five relations join query on the Housing dataset

```
SELECT SUM(HOUSE.postcode*HOUSE.postcode) FROM HOUSE
NATURAL JOIN SHOP NATURAL JOIN INSTITUTION
NATURAL JOIN RESTAURANT NATURAL JOIN DEMOGRAPHICS;
```

Listing A.11: The four relations join query on the Housing dataset

```
SELECT SUM(HOUSE.postcode*HOUSE.postcode) FROM HOUSE
NATURAL JOIN SHOP NATURAL JOIN INSTITUTION
NATURAL JOIN RESTAURANT;
```