ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

BENCHMARKING MATERIALIZED VIEWS OF SQL-BASED STREAM PROCESSING SYSTEMS

Tesi in
Big Data

Relatore
Prof. GALLINUCCI ENRICO

Presentata da
PARRINELLO ANGELO

ANNO ACCADEMICO 2022–2023
Quarta Sessione di Laurea

Keywords

Stream Processing System
Materialized View
Big Data
Benchmarking
Apache Flink
Materialize
ksqlDB
Apache Kafka
Nexmark

Abstract

With the growing importance of real-time analytics, streaming processing systems are becoming an essential tool; in particular, the paradigm of the materialized view is gaining traction due to its simplicity and power. Moreover, a standard benchmark for these systems is still missing, although multiple benchmarks have been proposed for the more general field of stream processing. Many tools can support this kind of processing, both well-established and newer to the scene with different feature sets and technical choices. The goal of this thesis is to thoroughly evaluate and compare these tools in order to gain a deep understanding of their performances, characteristics, and limits. The competitors considered in this evaluation are *Flink*, *Materialize*, and *ksqlDB* (previously KSQL). Multiple aspects will be evaluated, from more abstract ones such as architecture, features, and technical choices to more concrete ones revolving around performance, scalability, and ease of use. This thesis proposes a baseline analysis of the three systems aim to exploit the advantages and disadvantages of each system and provide a comparison between them. As a starting point, we will use the *Nexmark* benchmark, which is a relatively simple and uncharted benchmark for stream processing systems.

Dedicated to all those whose constant support and encouragement made this journey possible.

Acknowledgements

First of all, I would like to thank the entire Agile Lab for allowing me to work on this project. I had the honor of knowing this incredible company and working with a special person, Nicolò Bidotti, my internship supervisor. His patience, support, motivation, and immense knowledge have been a great source of inspiration for me. I would also like to express my deepest gratitude to my supervisor, Prof. Enrico Gallinucci. I am extremely grateful for his guidance and support in a completely new field for me: without his help, this work would not have been possible. Huge thanks to my family, who have always been there for me in every moment of this journey, my mother Marina, my father Carlo, my sister Giorgia, and my girlfriend Sofia. Particular thanks to my friends and colleagues of study and work, Davide, Paolo, Francesco, Angela, Eddie, Leonardo and Filippo. Thanks to them I have been able to overcome also the most difficult days of study. I had the luck of living one of the most beautiful experiences of my life thanks to the Erasmus program, and I would like to thank all the people I met in that period, especially Federica, Mattia and Nick. Finally, I would like to thank my friends, who have always been by my side, Kevin, Alessandro, Andrea, Lucia, Anna, Clarissa, Ginevra, Chiara, Alessandro, Martina, Beatrice, Valentina, Giacomo, Nicola, Albi, Nikolas, Maicol, Riccardo, Luca and all the others.

Contents

Chapter 1

Introduction

In response to the increasing demand for fast, reliable and scalable data processing, the field of *stream processing systems* has seen a surge in popularity. The traditional paradigm of batch processing, where data is collected and processed in large chunks, is no longer sufficient for many use cases. In contrast, stream processing is a processing model that aims to compute data on-the-fly, as it arrives, and creates a continuous stream of results. This model is particularly well-suited for applications that require low-latency processing, such as fraud detection, monitoring, and real-time analytics. Many widely popular frameworks like *Apache Flink* [15] or *Apache Spark* [121] have been developed to support enterprises in this kind of processing. While most of these tools have first implemented a core API in Java or Scala, newer systems mostly focus on providing abstractions for running SQL queries. SQL is a standard language for performing data analysis and is widely used in the industry from all those personas most interested in data processing, such as data engineers, data scientists, and business analysts. In this context, systems like *ksqlDB* [65] or *Materialize* [21] are increasingly used to provide a more user-friendly interface to the stream processing model. One of the main concepts that these systems support is the *materialized view*. Contrary to the traditional view, which is a virtual table that is not stored physically, a materialized view is a physical copy of the data, which is updated periodically. The development of new systems and paradigms in this area has brought challenges to the testing of these

big data systems. First, the lack of a standard benchmark for stream processing systems has made it difficult to compare the performance of different systems. Second, the variety of tools available makes it difficult to choose the right one for a specific use case. Finally, the complexity of the systems leads to a challenging metrics selection process, as well as the need for a standardized way to evaluate the performance of these systems. How to reasonably compare and evaluate the performance of stream processing systems has become a new direction of research in the field of big data systems.

This thesis aims to take an initial step toward addressing the challenges of testing stream processing systems that support materialized views utilizing SQL queries. To this end, we start by identifying the main challenges in testing these systems and the current state of the art in benchmarking stream processing systems. We then propose a benchmark for testing stream processing systems that support materialized views using SQL queries. The benchmark will be based on the *Nexmark* [69] benchmark, which already provides a set of queries, specific to streaming operations. We will extend the benchmark to include three systems that support materialized views using SQL queries: ksqlDB, Materialize, and Apache Flink. We will then evaluate the performance and qualitative properties of these systems using the proposed benchmark. Finally, we will discuss the results and conclude the performance of these systems.

The thesis is structured as follows. In Chapter 2, we provide an overview of the state of the art in stream processing systems and a summary of the theoretical background. In Chapter 3, we present the current state of the art in benchmarking stream processing systems, the challenges in testing these systems, and the benchmark on which the baseline of our work is based. In Chapter 4, we describe the implementation of the benchmark and the technologies used. In Chapter 5, we present and discuss the results of the comparison. Finally, in Chapter 6 we draw conclusions and discuss future work.

Chapter 2

Stream Processing System

A *stream processing system* is a distributed system that allows the processing of data in motion, that natively supports requirements such as scalability, fault tolerance, and extensibility [3]. Due to its nature, in the last years, stream processing systems have become more and more popular, and companies have started to use them to process data as soon as they are gathered, to have a real-time view of the business. This behavior is the main difference between streaming and batch systems: the latter does not allow the handling of a continuous flow of data. Many are the challenges that a stream process engine should address. One of the biggest is the capability to act *stateful operations*. Since real-time streaming is theoretically unbounded, a system of this kind must process information on-the-fly (involving a narrow memory). Nowadays, we can go beyond traditional analytics by leveraging the internal mechanisms of these engines. Indeed, one of the biggest advantages of stream processing systems is the ability to generate a *streaming materialized view* (i.e. the results of a continuous query) [4] that can be used to feed other applications or to trigger some actions.
After a brief introduction to what stream processing systems are, the chapter will introduce research on the current state of the art, produced during the first phase of the project (candidate systems selection), and it will conclude with a description of the three systems that we have chosen for our benchmark: Apache Flink, ksqlDB, and Materialize.

2.1 State of the art

Streaming big data architecture is typically composed of two main layers: the *message queuing* layer and the *stream processing* layer. The first one is responsible for ingesting data from different sources, maintaining them, and serving the information to the other systems such as the stream processing layer. Often, these systems are called *message brokers* and they are used to decouple the data producers from the data consumers. Starting from the 00s, many message brokers have been developed. The first ones were centralized systems, such as ActiveMQ [181] and RabbitMQ [182], but with the advent of cloud computing, distributed systems have become more and more popular. Nowadays, some of the most used message brokers are Apache Kafka [66], and Apache Flume [183]. In this thesis, we will use Apache Kafka as a message broker. Kafka is an open-source distributed event streaming platform, which is able to handle real-time data streams. It provides a scalable and fault-tolerant architecture as well as huge integration capabilities, at the cost of a more complex setup and steeper learning curve. However, the message broker is not the focus of this work, so we will not go into details, leaving aside technical aspects and architectural choices. We will focus on the second layer, the stream processing layer.

The concept of stream processing systems (from now we will refer to them as *SPSs*) is now well known and both open and closed solutions, are available. Many big actors in the market have developed their own SPS, such as IBM with IBM Streams[5] and Google with Cloud Dataflow [6], but virtually all cloud vendors have their solution. As stated in [7], this concept is not new and it has been around starting from the 90s. Through the years, the focus on these products has changed. In the beginning, the main goal was to provide a system that could handle a continuous flow of data. We assist in the creation of prototypes such as TelegraphCQ [9], NiagaraCQ [10], and Aurora [11]. These models highly inspired the first wave of commercial SPS, for instance, IBM System S [12] and Oracle CQL/CEP [13], which aims to support streaming window queries and complex event processing (CEP).

Then, cloud computing and the introduction of MapReduce [8] have changed

the way we process data. The focus has shifted towards the scale-out ability, so distributed and fault-tolerant systems, but we also witness the development of projects that could facilitate the creation of User-Defined Functions (UDF) - based programming models, supporting the widely used MapReduce paradigm. Throughout this period, many big-data streaming architectures have been proposed, such as the Lambda Architecture [51] and the Kappa Architecture [52]. Both of them are based on the idea of handling batch and stream business problems, but the former uses two separate layers to manage different types of processing, while the latter uses a single layer to manage both batch and stream processing.

Another important turning point in this second season, was the introduction of the Google Dataflow Model [18], which is a unified model for batch and stream processing (now incorporated in Google Cloud Dataflow [6]). The idea behind this concept is to provide a single programming model, for both batch/stream processing and un/bounded stream of data, that can easily deal with massive out-of-order streams. Furthermore, this approach is engine and language agnostics, as showed by Apache Beam [19], which is an open-source implementation of the Dataflow Model. All these features have been further developed by many open-source projects, such as Apache Storm [14], Apache Spark [17], Apache Flink [15] and Apache Kafka Stream [16], which are still extremely popular and have introduced new paradigms such as processing guarantees (i.e. exactly-once processing), reconfiguration (i.e. adding new workers to a running computation), state management(i.e. capturing all the changes in the state of a computation), and SQL support.

The third wave of SPS is the one that we are currently living in. The market is concentrating on incorporating data streaming with either cloud services and serverless applications or edge computing and specialized hardware [7]. On the other hand, we observe the foundation of new streaming database systems (also cloud-based), such as Materialize [21] and Bytewax [195] (both based on Timely Dataflow), Arroyo [23], and RisingWave [24], which tend to fully support SQL or new lightweight systems, such as Portals [22], Faust [62], and Quix [196].

While new systems are being developed, new features for the existing ones are being introduced. Internal state management is one of the most important aspects of the last decade of SPS. Although those mechanisms have reached a certain grade of maturity, most of the time they are being used as a black box. Indeed, either the user cannot interact with the internal state of the system or the only way to do it is through a complicated process. S-Query [25] points out the importance of exposing the internal state of an SPS to external applications as a materialized view. The authors show how this prototype introduces minimal overhead in the tested systems (primarily Hazelcast [26] but also on Flink has been proposed) while providing a way to access the information stored in the system for debugging, monitoring, and analytics purposes.
Figure 2.1, which is a variant of that showed in [7], illustrates the evolution of the SPS through the years, underlying the main features introduced in each wave.

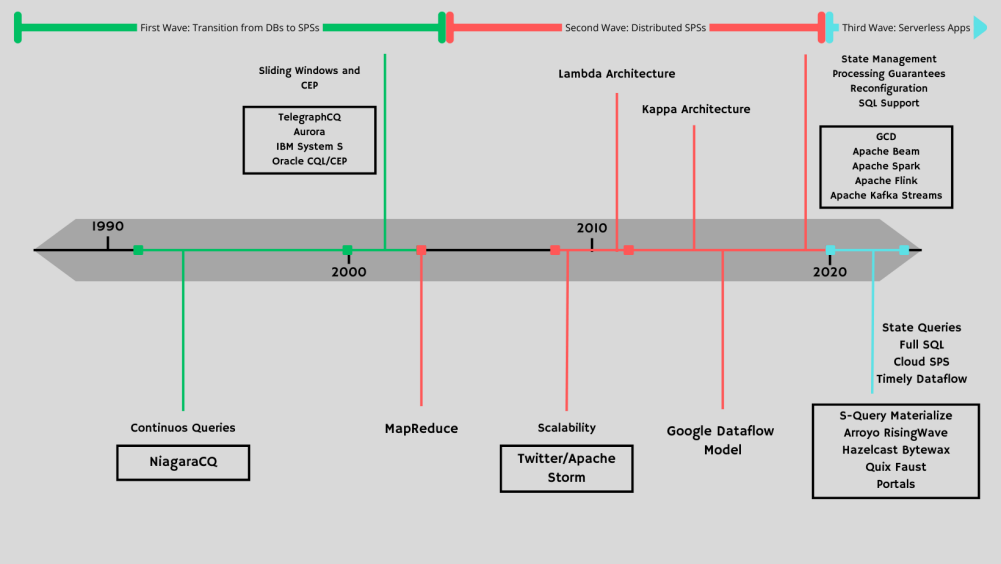

Figure 2.1: Evolution of stream processing systems

Along with the plethora of SPS, three of them have attracted our attention: Apache Flink, ksqlDB, and Materialize. Apache Flink and ksqlDB are two of the most popular open-source SPS, while Materialize is a new entry that was released

in 2020. To better comprehend the current landscape of SPS, we have chosen to benchmark these three systems: Flink is a well-established SPS that has been around for a while, ksqlDB is quite new but the technology under the hood is not (is built on top of Apache Kafka Stream engine [27]) and Materialize is one of the most promising new SPS. In the next sections, we will introduce these three systems, highlighting their main features and the differences between them.

2.2 Background

Dataflow programming model

The *dataflow model* [18], represents a streaming computation as a logical directed graph $G = (V, E)$, where vertices in V are operators and edges in E denote data streams. After the deployment, the logical graph begins a *physical execution plan*, $G' = (V', E')$, which maps operators to provisioned workers (i.e. threads). The vertices in V' are called tasks or instances of a logical operator in V and edges in E' physical data channels. Tasks are usually scheduled once and have extended durations. Each task belongs to a single worker, and each worker can execute one or more tasks, whether they are owned by the same or different operators. The internal mechanism of the assignment is system-specific. It is computed at deployment time and remains constant during the whole execution unless a reconfiguration occurs. In most of the cases, the execution is *data-parallel*, meaning that all the tasks of an operator perform the same logic on disjoint partitions of the input stream. Then, an operator must communicate with the tasks of upstream and downstream operators through messages.

The *dataflow programming model* is a programming paradigm that models a program as a directed graph of the data flowing between operations. Essentially, dataflow programming involves defining your program as autonomous components that respond to the presence of input data and specify the relationships and connections among these components. A dataflow program can be represented as a direct graph where the nodes are the components/operators and the edges are the connections between them.

What nowadays happens in a large number of SPS is that the dataflow model is used as a programming model. In other words, the system offers the possibility to the user to compose applications in several languages (i.e. Java, Python, SQL, etc.) and then it will be translated into a dataflow graph by the underlying engine that will also distribute the computation across the cluster and subsequently run the transformed application over incoming data streams. A graphical representation of the dataflow programming model is shown in Figure 2.2.

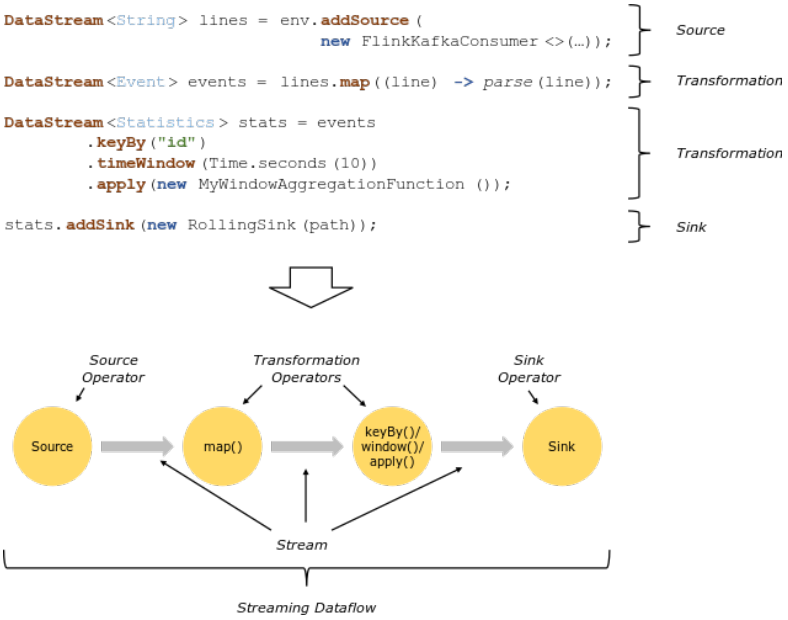

Figure 2.2: Dataflow programming model example [20].

Delivering Guarantees

During a critical situation, such as a failure, the system could have lost, or duplicated some events. There are three kinds of guarantees that a stream processing system can provide: *at-most-once*, *at-least-once*, and *exactly-once*. The first one is the weakest guarantee, meaning that the system can lose some events, the system will not try to recover the data nor check whether the events have been processed or not. The second guarantee is slightly stronger than the previous one, meaning that the system will not lose any event but it could process some of them more than once: it will be an engineer's responsibility to handle the duplicates. The last one is the strongest possibility and it means that the system will not lose any event and will process each event exactly once [59]. What is important to say about EOS (Exactly-Once semantic), is that does not deduplicate messages created by an application (i.e. producer) or an external source. So, for example, if the client code produces duplicate messages, it is a problem of the client code, not of the EOS and so, it will not remove those events. What EOS does is to ensure that all the messages produced and consumed by the same producer/consumer are

marked as committed atomically (i.e. ensure that all the messages are processed and delivered correctly) or aborted (i.e. ensure that no message is processed and delivered).

Event Time and Processing Time

Within the domain of stream processing, there are two main definitions of time:

- **Event Time**: it is the time at which the event occurred on the source device [6]. This time is usually melted into the event itself and we can extract it as *event timestamp*. Ideally, the processing of the events based on the timestamp would yield deterministic results regardless of the order or when the events arrive. But in reality, event time processing will include some latency waiting for late events to arrive. Thus, we can only apply a heuristic method to deal with late events, such as waiting for a certain amount of time before closing a window. Most of the time the event time-based processing will eventually produce the correct results, but it is not guaranteed. For instance, assuming that all the events have arrived and in the context of an hourly event time window, all records with event timestamps within that hour will be included, regardless of the order of their arrival or when they undergo processing.

- **Processing/Wall-clock Time**: it is the time at which the entire processing of an event occurs [6]. When a pipeline is working with the processing time, all the time-based operators will use the system clock of the machine that is executing the operation. For example, if we have a window of 10 minutes, and the application starts at 12:03, the first window will contain all the events that have arrived between 12:03 and 12:10, the further one between 12:10 and 12:20, and so on. While this approach is simple and provides the best performance and lowest latency as well, it is not reliable in terms of determinism due to factors like varying arrival rates of records, differences in processing speeds between system operators, and the impact of outages.

An important (and pragmatic) take-home message is that none of the two approaches is better than the other. Indeed, the choice of the time model depends on the use case and the requirements of the application and, sometimes, it is possible to use both of them in order to ensure the correctness of the results and the responsiveness of the system [44]. As a rule of thumb, we can say that event time is generally preferred because it is consistent, and working with it should produce deterministic results even though working with processing time is faster and easier.
In an ideal world, the event time and the processing time would be the same. However, the reality is that the skew between the two times is not even zero but a variable function, depending on several factors such as network latency, software delays, and so on. If we try to plot the difference between the two times, we will obtain a graph like the one in Figure 2.3.

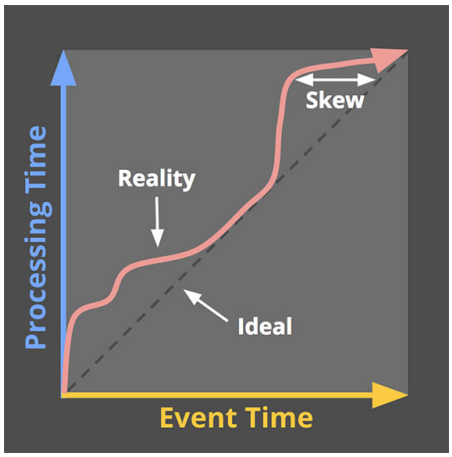

Figure 2.3: Skew Time Difference from [45]. The X-axis represents the completeness event time and the Y-axis the progress of the processing time.

In the figure above, the X-axis represents the completeness of the event time (i.e. time X is the point in event time where all data with event times preceding X have been recorded or observed) and the Y-axis is the progress of the processing time. The black dashed line shows the ideal case where the processing and event time are the same. The red line represents the actual skew between the two times. To visually calculate the skew, we can take the horizontal distance between these

two lines: the greater the distance, the greater the skew. In other words, the skew is the *latency* resulting from the processing pipeline. The concept of the skew is important because introduces some challenges in analyzing unbounded data streams. Instead of relying on either event time or processing time and their respective drawbacks, we should use systems that can handle the complex and uncertain nature of this information, supporting data retracting and data updates [45].

Windowing

The most common approach for an unbounded stream of data is *windowing*. The technique of windowing is to divide a data source (either bounded or unbounded) into a finite number of sets, called windows, based on certain temporal constraints. Along with the idea of windowing, we have the concept of *triggers*, which is a mechanism proposed in [18] to determine when a window is ready to be processed. There are three main different windowing patterns (Figure 2.4 tries to sum up them):

- **Fixed/Tumbling Windows**: in this case, the windows do not overlap and slice up time into chunks of fixed length [46]. The most common use of this pattern is with an *aligned window* that starts at regular intervals (i.e. every 5 minutes) and processes all the events that fall in that time frame. The main pros of this technique is that we ensure consistency but we need to pay attention to the uneven distribution of the workload. In *unaligned windows*, the interval may vary for different parts of data, usually based on some criteria like a key. While by this approach we can solve the problem of uneven distribution, we surely introduce some complexity in the system.

- **Sliding Windows**: in this case, we have two parameters that define the window's behavior: the window size (i.e. the length of the window) and the slide interval (i.e. the frequency at which the window operation is performed) [46]. For example, we would like to compute every thirty seconds the number of cars passed in the last minute. What emerges from this example is that if

the period of the slide interval is smaller than the window size, the windows will overlap. Thus, if the period is equal to the window size, we will have a tumbling window. As with the previous mechanism we can have aligned and unaligned windows, with the latter that is mostly used in real-world cases.

- **Session Windows**: in this case, the windows are created dynamically and the sessions are composed of events that are separated by a gap of inactivity (i.e. timeout) [46]. They are useful when we want to group events that are related to each other by certain criteria (i.e. user session). The main difference with the previous two patterns is that the windows are not aligned (indeed, they are the standard unaligned windows) to a certain point in time, but they are created on the fly.

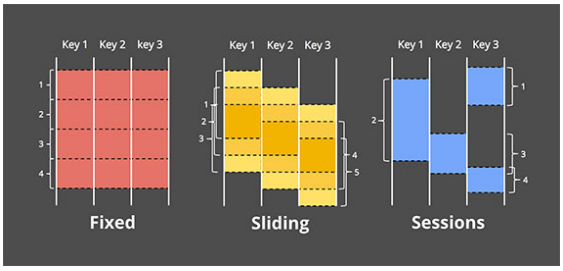

Figure 2.4: The three main Windowing Patterns [45]. These three strategies show the differences between aligned (i.e. all the datasets are processed at the same time) and unaligned (i.e. subsets are processed at different times) windows.

The concept of windowing is widely applied both in the processing by processing time and event time. In the first case, a system will buffer the data into windows until the processing time reaches the end of the window. For example, if we have a 5-minute window, a system will buffer the data until the processing time reaches the end of the 5-minute window and then will process the data. The three main benefits of this approach are the low overall complexity of a system, determining the window completion is easy, and inferring information about the health or the status of a source is straightforward. On the other hand, the main drawback is that if the incoming data have a timestamp framed in itself, those

data shall arrive in time-order to reflect the reality. As stated before, this is not always the case.
In the second case, a system will mirror what happened in the real world. Hypothetically, this approach is the best one because it is the most accurate yet not exempt from drawbacks. Since a system has to handle late events, it needs to buffer the data. It must be mentioned that persistent storage is one of the cheapest resources in a data center and many operators could work incrementally (i.e. sum), so the cost of buffering could be mitigated. However, the problem of determining when a window is complete is not trivial but we can deal with it.

Watermarks

Many challenges have arisen from the introduction of time in stream processing. One of the most fascinating is the managing of window completeness. As explained before, we cannot know when all the events have arrived, especially when processing events on their event time. What we should do is define a heuristic that allows us to determine when a window can be considered complete. A popular heuristic method is called *watermark*.
The concept of using watermarks in streaming processing systems was introduced by Google in [47] and [18]. A watermark can be defined as a metric that signals the completeness of streaming progress. To put this in easier words, a watermark with a value of X means that all the input data with event times less than X have been processed [48]. By saying that, we are assuming that events are at most X minutes out of order. Earlier events are still possible, but they are not expected. If they do occur, they violate our assumption that the stream is no more than X minutes out of order and such events will be considered late. Depending on what we are doing, late events might be either ignored or not. Through the watermarks, it is possible to measure the progress in event time. The watermark bundle is included in the stream and piggybacks the timestamp t, as illustrated in Figure 2.5. In the figure 2.5 we assume that the stream is composed of ordered events and watermarks flowing inline. But the watermarks are crucial particularly when the stream is out-of-order. In this case, once a watermark anchor reaches an operator,

the operator will increase its own event time clock to the watermark's value.

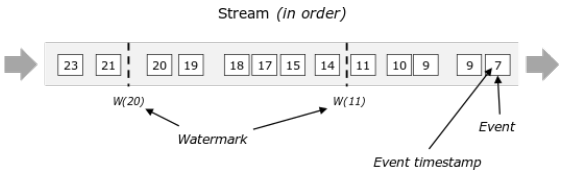

Figure 2.5: Watermark mechanism [49]

Persistent, Push and Pull Queries

Within the domain of stream processing, there are three types of possible queries: *persistent*, *push*, and *pull*.

- **Persistent Queries**: these kinds of queries are server-side and extremely common in stream processing systems. They continuously update themselves as soon as new data arrives. The term "persistent", indicates that the query will run indefinitely and remain active as new data arrives. The results of these queries need to be stored in a persistent and durable storage, making them resilient to failures, and ready to be used by other queries or applications.

- **Push Queries**: these are a form of client-side queries, that follows the Publish-Subscribe pattern [76]. In this case, the client subscribes to a query and receives the results as soon as they are available, in a real-time fashion [74]. Contrary to the persistent queries, the push queries are not stored in durable storage, but they are sent directly to the client.

- **Pull Queries**: pull queries try to mimic the behavior of a traditional database synchronous control flow[75]. The client sends a query to the server and receives the results as soon as they are available. Thus, the client is responsible for polling the server for new results and thus, it will receive the results currently available.

Figure 2.6 shows the differences between the three types of queries.
A *materialized view* is an object that stores the result of a precomputed query in

a persistent, physical form. It serves as a precomputed snapshot of data, allowing for faster query performance by avoiding the need to repeatedly compute complex or resource-intensive queries. In this sense, a materialized view has to save some results in a reliable memory, so it is typically the result of a persistent query, while the push and pull queries are used to interrogate the system.

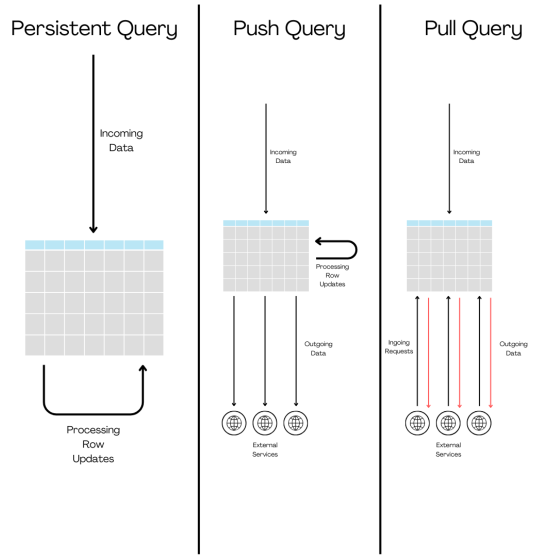

Figure 2.6: During an interrogation through SQL, in a stream processing system we can find three types of queries.

2.3 Apache Flink

Apache Flink [15] is an open-source stream processing (and batch) engine that was initially developed as a research project by a group of German universities [28]. The project, named Stratosphere and started in 2011, had to create a new generation of Big Data Analytics platform that could overcome the limitations of the existing ones. In 2014, the project was renamed Apache Flink and it became a top-level project of the Apache Software Foundation.

Overview

Figure 2.7: Apache Flink logo

Flink's core is a distributed streaming data-flow engine written in Java and Scala [29]. Its engine is attempting to tackle some challenges such as reducing complexity through declarative languages and parallel in-memory and out-of-core algorithms [30]. Moreover, Flink supports a query optimizer that automatically improves the programs, leaving the engineer free to focus on the business logic. Its runtime system supports high-throughput and low-latency data processing applications [31], with a feather-light fault-tolerant algorithm [32]. As shown in Figure 2.8, Flink is composed of four layers [33]:

- **Stateful Stream Processing**: this is the lowest layer of abstraction that Flink offers. Withing this block, the user can process events from one or more streams realizing complex operations.
- **DataStream/DataSet API**: most of the time, the user does not need to interact with the previous layer. However, the user is free to use that low-

level mechanism through this API that embeds the stateful stream processing layer. In addition, this API offers a set of common operations for data processing (i.e. map, filter, join, windows, etc.) both bounded (DataSet) and unbounded (DataStream). This API is available in Java, Scala, and Python.

- **Table API**: this layer is built on top of the DataStream/DataSet API as a declarative DSL. The main concept behind this API is the *table*, which is a structured view of a stream or a batch of data. Although this API does not allow as much expressivity as the previous one, it is easier to use and more concise. Another pro of this layer is the possibility to easily translate tables in datastream/dataset objects and vice versa. You can use this API in Java, Scala, Python, and SQL.

- **SQL**: this is the highest layer of abstraction that Flink offers. The SQL abstraction interoperates with the Table API, in the same way as we would expected: the user can write SQL queries that run against tables/streams.

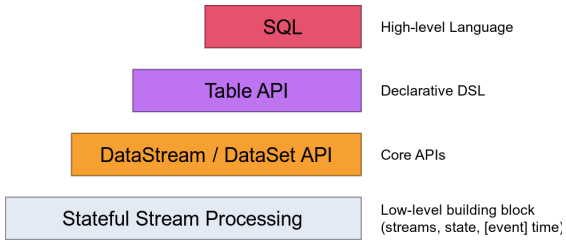

Figure 2.8: Apache Flink layers of abstraction [33]

Stateful Stream Processing

Flink's State

There are many reasons for Flink's success, but one of the most important is the ability to perform *stateful stream processing*. First of all, we need to understand what is a state in a stream processing system. In a typical data flow, many of the operations are stateless, meaning that those operators transform the input at one

individual event at a time. On the other hand, stateful operators need to keep track of the previous events to produce the output [34]. For instance, a stateful operator could be a window that aggregates the events that fall in a certain time frame. Let us consider an application that is looking for a specific pattern in a stream of events. In this case, the system needs to keep track of the previous events to understand if the pattern is present in the stream. By the use of a state, Flink can reach fault tolerance, scalability, and consistency.

Checkpoints and Savepoints

The state of a Flink's operator or function is a critical piece of information that needs to be stored in a fault-tolerant way. To achieve this goal, Flink provides two mechanisms: *checkpoints* and *savepoints*. A checkpoint is a consistent snapshot of the state of the application at a certain point in time whereby Flink might recover the state and positions in the input streams in case of failure [35]. During the lifetime of an application, Flink periodically takes checkpoints (in an asynchronous way) that must be stored in a persistent storage system (i.e. HDFS, S3, etc.) called *state backend*. Flink already provides some out-of-the-box state backends, for instance, an embedded RocksDB [149] one or the default filesystem one.
A savepoint creates a representation of the execution state of a job, via the checkpointing facility [36]. Hence, you can trigger manually a savepoint and then use it, for example, to stop and resume or to update a job. Theoretically, a savepoint is a special type of checkpoint, yet with different meanings [37]. The purpose of a checkpoint is to provide a safe recovery method after an unexpected failure. The whole checkpoints' lifecycle is automatically managed by Flink, and since are often created, they are lightweight. On the contrary, savepoints are typically planned (i.e. before updating a job) and more expensive to create. Moreover, Flink does not take care of whatsoever savepoints' lifecycle (i.e. creation or deletion), so the software engineer has to handle it.
As stated before, the backbone of Flink's fault tolerance engine is checkpointing. The model [32] is deeply inspired by the Chandy-Lamport algorithm [41], which is a distributed algorithm that allows the creation of a consistent snapshot of a

distributed system. With the term *barrier*, we refer to a special type of event that is injected into the data stream and that triggers the checkpointing mechanism [42], as shown in Figure 2.9.

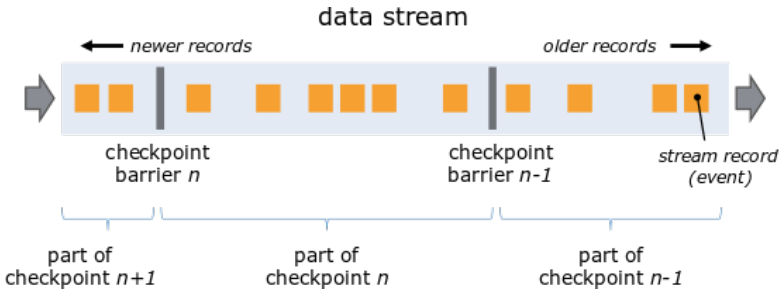

Figure 2.9: Apache Flink Stream Barriers [42]

The barriers are propagated through the data stream and when an operator receives a barrier, it stops processing new events and starts to finalize the current ones. When all the operators have received the barrier, the checkpoint is considered completed. When an operator has multiple input streams, it waits for all the barriers to arrive before finalizing (i.e. writes the state to the state backends) the events. This mechanism is called *aligned checkpointing*. Flink also allows the use of *unaligned checkpointing*, which is a more relaxed version of the previous one [43]. In this case, the barriers go through the stream as fast as possible, at the cost of increasing I/O operations.

Keyed and Operator State

States in Flink can assume two main different forms: *keyed* and *operator*. In the domain of stream processing, the data are usually associated with keys, and many operations are performed on a per-key basis. Flink supports a key-value processing paradigm, through an embedded key-value store, enabling efficient state management [38]. Furthermore, the keys are used to partition the data across the parallel instances of the operators. Ensuring these keys match means the state can be kept local, leading to in-memory caching and speeding up disk access. But this process is as powerful as dangerous: if your key space is very large or not well-distributed, you can end up with a skewed workload or a huge state that cannot

be stored in memory. However, there are some workarounds to this problem, such as using a hash function to distribute the keys across the parallel instances of the operators or implementing a time-to-live (TTL) setting. The keyed state is further divided into groups, named *key-groups*, which are the unit of parallelism (i.e. the number of parallel instances of the operator). In Figure 2.10, we can see how the keyed state is partitioned across the parallel instances of the operator.

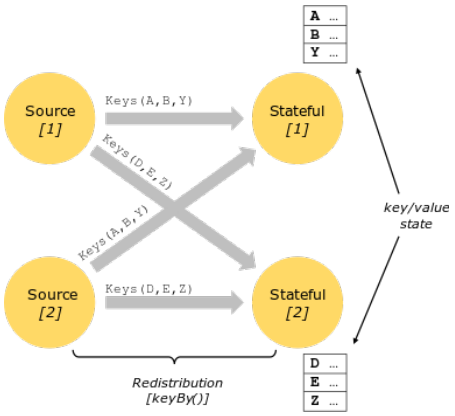

Figure 2.10: Apache Flink Keyed State mechanism [38]

On the other hand, the operator state is a state that is not associated with keys and it is bound to a single parallel instance of an operator [39]. Although the usual stateful Flink application does not require the use of the operator state, it can be useful in some cases, for instance when you do not have a key to partition the data. Since version 1.5.0, Flink has extended the concept of operator state, introducing the *broadcast state*. This new type of state is a special type of operator state that is broadcasted to all the parallel instances of the operator [40]. This mechanism is useful when you need to enrich the data with some information that is not part of the stream. A use case could be when you have a low-throughput stream (i.e. a stream that contains configuration updates) and a high-throughput stream (i.e. a stream that contains data to be processed). Via this broadcasting trick, it is easier to send to all the parallel instances the updates required and consequentially update the processing logic for the high-throughput stream.

Flink Guarantees

Flink supports all three delivery guarantees, with a major focus on the strongest one: *exactly-once* guarantee. Indeed, Flink can provide that guarantee even in the presence of failures. Despite that, this does not mean that each event will be put on trial only once, it means that every event will affect the state of the system only once. The checkpointing mechanism is the key to this guarantee. A distinguished speech must be made for the achievement of exactly one end-to-end delivery. To reach this kind of constraint, every event from the sources has to influence the state of the sinks exactly once. Unfortunately, this is not always possible. To provide exactly-once end-to-end delivery it must be guaranteed that: the sources must be replayable (i.e. the events must be stored in a persistent storage), and the sinks must be idempotent (i.e. the result of applying a function multiple times is the same as applying it once). From the Flink side, it already supplies some out-of-the-box connectors for both sources and sinks that are replayable and idempotent, respectively [60].
In a typical Flink application, we have data sources, data pipelines, and data sinks. A failure can occur in any of these three components. Flink provides a mechanism (i.e. checkpointing) to recover from a failure providing consistent views across three important things: the internal compute state (i.e. the state of a windowed operation), the external transaction identifiers (i.e. which part of the stream I have already processed), and the offset (i.e. track the progress of the consumer from the source). What checkpointing does is to apply the *Two Phase Commit (2PC)* protocol. In brief, the 2PC protocol is a distributed algorithm that coordinates all the processes involved in a distributed transaction to reach a consensus. Then, Flink extended it in an *asynchronous* way [184]. In this sense, Flink is the transaction coordinator so it will decide when to commit or abort the transaction.

Timely Stream Processing

Whenever an operation during a Flink's processing pipeline requires the use of time, that processing is called *timely stream processing*. Among the most common operations that engage the concept of time, we can find time series analysis, event time processing, and windowing.

Time in Flink

Flink supports both of the two main notions of time, including a third one called *ingestion time*. The ingestion time is the moment when the event arrives in the processing pipeline [61] or in other words, the time when the event is ingested by the source Flink's operator. However, to be sure that a result is reproducible, the concept of event time is the most suitable one. In this way, we ensure that the result will depend on when the events actually occurred and not when they were processed by the system. From the Flink 1.12 version, the event time is the default one, but the user can still use different time notions.
Indistinctly from the time notion used, Flink supports data updates and retraction. The mechanism of the watermark is useful in order to support these operations and many others. These watermarks are generated directly from the sources and automatically forwarded through the distributed data flow. By default, Flink will look at the timestamp embedded in each message to generate watermarks. To define the watermark value, Flink analyzes the maximum timestamp of the events seen so far. Then, it subtracts a certain amount of time, the out-of-order estimation assumed before. The result is the watermark value for that specific source. This case can be expanded to multiple sources, that are flowing into a single operator, by taking the minimum watermark value among the sources. To better explain this concept, let us consider the example in Figure 2.11. In this example, we have multiple sources (and so multiple levels of parallelism) that are flowing into a single window operator. Each source belongs to a Kafka partition, and for each partition, a watermark has been emitted (i.e. P2 1:30). The final watermark value for the window operator is the minimum watermark value among

the sources, which will help the operator to resolve when to produce a result.

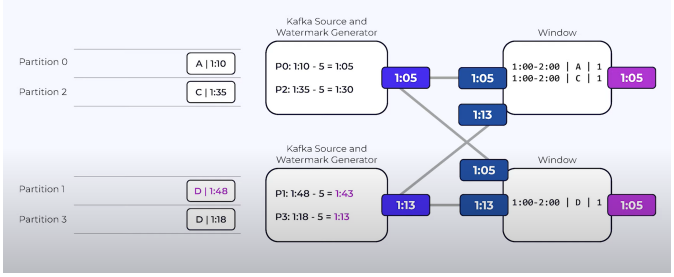

Figure 2.11: Apache Flink Watermark example [50]

Flink's watermark structure is highly related to windowing capabilities, which are one of the main mechanisms to perform time-based operations. Flink supports all the types of windowing mechanisms enumerated in Section 2.2. In addition, Flink features a very expressive windowing API, which allows the user to handle those events that arrive late (i.e. after the watermark has passed the end of the window) and usually are dropped, through a way called *side output* [130].

Architecture

Flink can operate as a local or distributed system. In the first case, the system can run as a standalone cluster (i.e. a single JVM) or as a library (i.e. embedded in a JVM application). In the second case, Flink can integrate with popular cluster resource managers like Hadoop YARN and Kubernetes or even run as a standalone cluster likewise in the first case. In Figure 2.12, we can see the architecture of a Flink cluster.

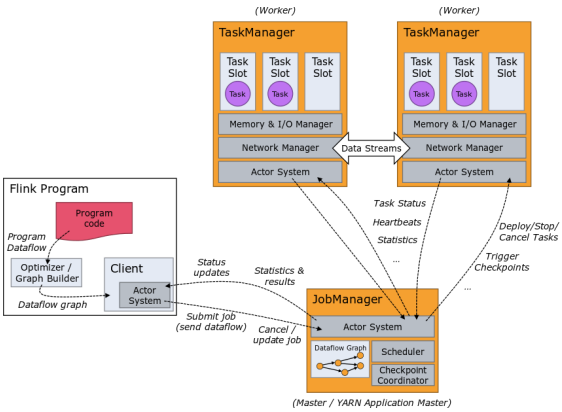

Figure 2.12: Apache Flink Architecture [53].

The implementation of Flink's distributed runtime is based on Akka [58] a framework for building distributed, message-driven applications [56]. The toolkit provides a high-level abstraction of the actor model [57] where all the components of the system are actors and they communicate with each other via messages. With this in mind, we can locate three main actors in Flink's architecture: the *Client*, the *JobManager*, and the *TaskManager*. However, the *Client* is not directly part of the runtime cluster but it is used to prepare and send the dataflow to the JobManager. The architectural pattern followed by Flink is the *master-worker* one, where the JobManager is the master and the TaskManager are the workers.

JobManager

The JobManager coordinates the distributed execution of the data flow (from now we will refer to a submitted data flow as a *job*). Amongst its tasks, we can find task scheduling, failure recovery, coordinating checkpoints, and resource management. In a Flink cluster there always must be one active JobManager, which is the *JobManager Leader*. Flink delivers *high availability* (i.e. will continue executing the ongoing jobs) by having multiple JobManager instances in the cluster, where one of them is the leader and the others are the standby ones. In those cases, the standby JobManagers are responsible for monitoring the leader and taking over its role in case of failure. The JobManager is composed of:

- **JobMaster**: this component is in charge of taking care of a single execution of a JobGraph. A JobGraph is created when submitting a job and represents the job itself via a directed acyclic graph (DAG), with the operators as nodes and the datastreams as edges [54]. Flink can administrate multiple jobs at the same time, generating a JobMaster for each of them.

- **Dispatcher**: it is the component that receives the dataflow from the client and then summons a new JobMaster for each submitted job.

- **ResourceManager**: govern the resources of the cluster is a task of the ResourceManager. Notably, it manages *task slots*, which are the atomic part of resource scheduling in a Flink cluster [55].

TaskManager

The TaskManager, which is basically a JVM process, operates as a worker in the cluster and it is responsible for executing the *tasks* of a dataflow as well as exchanging data streams with other TaskManagers. A task is a unit of work in Flink that composes the job, called *sub-task* when processing a partition of a datastream. One task encapsulates exactly one parallel instance of an operator and hence, each task is executed by one thread. Without at least one TaskManager, the JobManager cannot execute any job.

2.4 ksqlDB

Confluent [64] defines ksqlDB (ex KSQL [63]) as a *database purpose-built for stream processing applications* [65]. Basically, ksqlDB is a stream processing engine that is built on top of Apache Kafka [66] and Kafka Streams [16], and offers a SQL-like language to query the data streams (instead of using Java or Scala). The project was started in 2017 by Confluent and it was released in 2018.

Overview

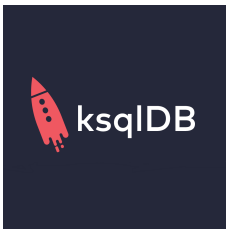

Figure 2.13: ksqlDB logo

ksqlDB is a distributed, scalable, fault-tolerant, and real-time SQL engine that allows the user to query the data streams and craft materialized views over those streams. This system, like Flink, lets the user the possibility to acquire data from different sources (i.e. Kafka, Kinesis, etc.) and then process them in real-time. However, ksqlDB tries to simplify the process of stream processing by offering an environment where the user can manipulate the data (i.e. ETL operations), without writing a single line of code. This advantage is given by the SQL-like language that ksqlDB offers. Processing data through SQL is beneficial for both technical and non-technical users. For instance, an analyst who has to create a report could easily use ksqlDB to analyze in real-time a specific piece of information that is needed.
ksqlDB seeks to serve also as a storage layer, by its Kafka backbone. The dependency on the latter is a double-edged sword: on the one hand, it is a great advantage because Kafka is a battle-tested system, on the other hand, it is a disadvantage because the user is forced to use Kafka as a middleware message

broker. In other words, ksqlDB is not *purely* standalone but runs natively of Kafka.

Dual Streaming Model

ksqlDB and Kafka Streams are based on the same streaming model, which is called *Dual Streaming Model* [67]. Formally, ksqlDB defines a stream's record as a tuple of $\langle o, t, k, v \rangle \in \mathbb{N}_0 \times \mathbb{N}_0 \times \mathrm{D}_V \times \mathrm{D}_V$, where the o is the offset of the record in the stream (i.e. the number of records that precede it), the t is the timestamp of the record and the payload is composed of a k key and a v value, within the domain of the data types D_K and D_V, respectively. The values could be of multiple types even also a composite data type.
Let the datastream be $S = (r_0, r_1, r_2, \ldots)$ with $r_i \wedge r_i.o = i$. Every single record inside the stream must respect the same schema $\mathbb{S} = \mathrm{D}_V \times \mathrm{D}_V$, which defines how the data is structured. A data stream S with schema $\mathbb{S}$ is denoted as $S[\mathbb{S}]$.
Let's now consider the following example depicted in Figure 2.14.

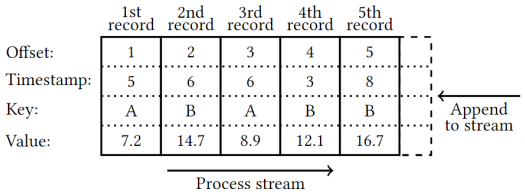

Figure 2.14: Data stream example from [67] with five records.

The stream S is composed of five records, each one with a different offset. The respective domains of key and value are $\mathrm{D}_K = \{\mathrm{A}, \mathrm{B}\}$ and $\mathrm{D}_V = \mathbb{Q}$. The records are ordered by the offset, but not by the timestamp. For instance, the record r_3 is described as $r_3 = \langle 3, 6, \mathrm{A}, 8.9 \rangle$.
The *logical order* is extrapolated by the offsets of the records, while the *physical order* is from the timestamps of the records. Considering r and r' two records of the stream S, we can say that r is *before* r' in the logical order if and only if $r_t < r'_t \vee (r_t = r'_t \wedge r_o < r'_o)$.
On the other hand, the physical order (i.e. records appended to the stream), may

not always align with their intended logical order, as illustrated in Figure 2.14. This inconsistency often arises due to factors such as imperfect clock synchronization, network delays, or data buffering, especially in distributed applications, as explained also in one of the previous sections 2.3. Streams that exhibit this out-of-order characteristic are referred to as *unordered streams*, distinguishing them from *ordered streams*.

The challenges introduced by the inconsistency between the physical and logical order may affect the computation of an operator state and output stream. Specifically, the latency of computing the output stream is dominated by the characteristics of the data stream rather than the semantic and implementation of the operator. These challenges have been addressed by the ksqlDB team along with the University of Berlin [67] and the result is the *Dual Streaming Model*, which includes the concepts of a *table*, a *table changelog stream*, and a *record stream*. The model proposes a continuous update of operator results, allowing for the handling of out-of-order records without increasing latency: updating an out-of-order event is as expensive as updating an in-order event because both are a key-value lookup in a hash table.

It presents a static view of the operator result as a table, continuously updated for each processed input record, and a dynamic view as a table changelog stream, consisting of record updates to a table (Figure 2.15).

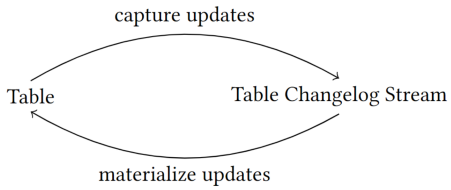

Figure 2.15: Table and (Table Changelog) Stream Duality from [67].

To sum up, the Dual Streaming Model separates the treatment of data arriving out of order from concerns about latency. This creates a design space (Figure 2.16) that allows users to navigate a balance between processing cost, acceptable latency, and the completeness of results. This result offers an alternative solution to other

models such as watermarks [68] or buffering-and-reordering [70].

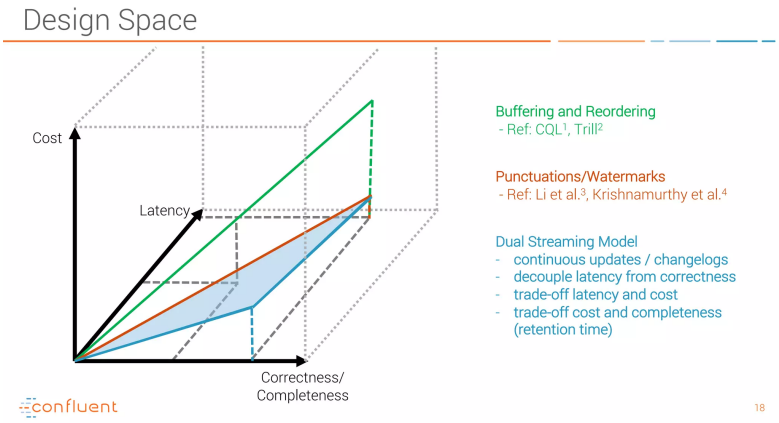

Figure 2.16: Design space of Dual Streaming Model from [71].

Domain Elements

Streams and Tables in ksqlDB

What we have seen so far is the theoretical background of the Dual Streaming Model. The next step is to understand how ksqlDB implements this model.

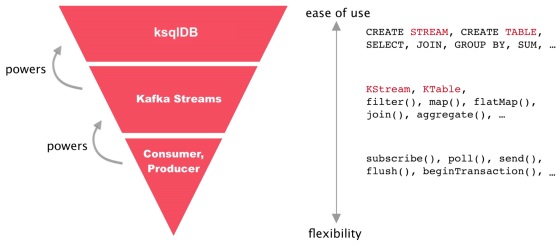

Figure 2.17: ksqlDB stack from [72].

As stated before, ksqlDB relies on Kafka (see Figure 2.17), which is a distributed streaming platform. One of the main components of Kafka is the *topic*, which is a category/feed name to which records are published, a set of key/value bytes. The DSL on top of Kafka for streaming purposes is Kafka Streams. Two of the main abstractions of this library are the *KStream* and the *KTable*. A KStream

is a stream of records, unbounded and continuously updated (i.e. a stream of events), while a KTable is a table that is continuously updated, basically a materialized view. Over these two abstractions, ksqlDB defines the concepts of *stream* and *table*.

In ksqlDB, a stream is a Kafka topic (i.e. it is partitioned and replicated across the cluster), an immutable, append-only collection of records. In addition, a stream has a *data schema* that defines the structure of the data contained in the stream [67]. Whereas a stream is an immutable, append-only collection of records, a table is a mutable, continuously updated collection of records. By contrast with a stream, a table shows the current value for each key (i.e. the last record arrived into the stream for that key).

In Figure 2.18, we can see the difference between a stream and a table.

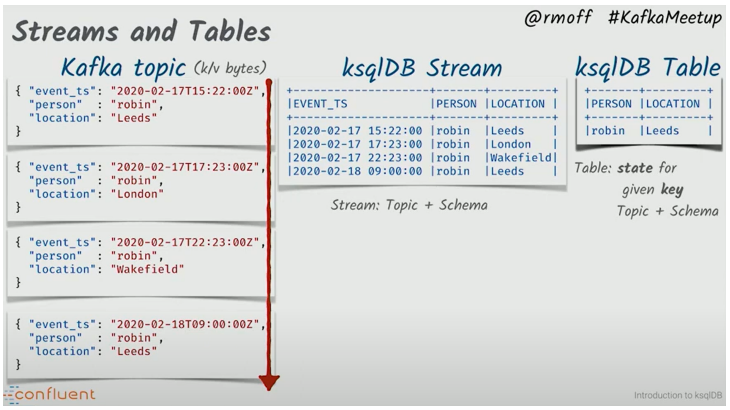

Figure 2.18: ksqlDB Stream and Table [65]

Querying Streams and Tables

ksqlDB offers a SQL-like language to query the data streams and craft materialized views over those streams. ksqlDB's SQL grammar [79] is based on the SQL-92 standard, with some extensions to support streaming semantics. Streaming features like persistent queries, leverage Kafka topics as the underlying storage layer. This means that the results of these queries are stored in a Kafka topic, inheriting the same properties of the topic (i.e. partitioning, replication, etc.). In particular, DDL statements (i.e. CREATE STREAM/TABLE) are used to add,

remove, or modify streams and tables. On the other hand, DML statements (i.e. SELECT, INSERT INTO, etc.) are used to query the data streams and craft materialized views over those streams. Since most of the time, the user needs to both create and query streams/tables, ksqlDB offers two operations for creating and querying a stream or a table called *CSAS (Create Stream As Select)* and *CTAS (Create Table As Select)*. These two statements occupy both the roles of DDL and DML statements because they perform an operation over the metadata layer, like adding a new table, and over the data layer, by creating a new object from existing records.

Architecture

ksqlDB Components

ksqlDB is composed of two main components: the *ksqlDB Server* and the *ksqlDB Client*. The ksqlDB Server is what ties the critical parts together: it contains the *ksqlDB engine* and the *REST API*. Then, the ksqlDB Client is the interface that allows the user to interact with the server. The user has only one way to apply to the server, particularly to the engine, and it is through the REST API. The REST API is typically used by the *ksqlDB CLI* and the *ksqlDB UI*.

- *ksqlDB Engine*: the core of this system, processes the SQL queries and statements. The engine, employing the DSL offered by the SQL-like language translates the logic into a Kafka Streams application. Afterward, the engine deploys the applications against the other ksqlDB server and the Kafka cluster. One engine belongs to one ksqlDB server.

- *REST Interface*: this component is in charge of receiving the requests from the clients and then forwarding them to the ksqlDB engine. The REST API is the only way to interact with the ksqlDB server.

- *ksqlDB CLI*: ksqlDB provides a command-line interface (CLI) that connects the user/client to the server. Via the CLI, it is possible to define the application logic, such as creating streams and tables, and then deploy them to

the server.

- *ksqlDB UI*: the UI is a web-based interface that allows the user to interact with the ksqlDB server. It is a graphical alternative to the CLI, directly integrated into the Confluent Control Center.

Figure 2.19 shows the components of ksqlDB and how they interact with each other.

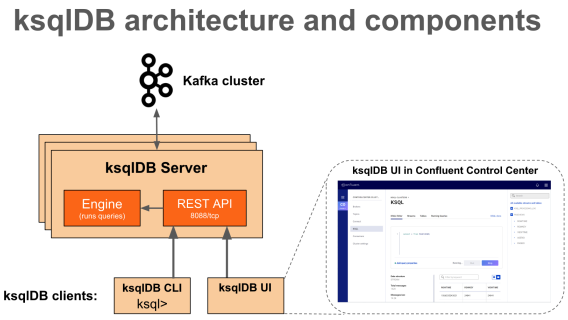

Figure 2.19: ksqlDB Architecture [77].

Query Lifecycle

Once a query/statement is submitted to ksqlDB, will start a flow (analyzed in Figure 2.20) that will allow the user to manage the application.

1. You create a stream or table above an existing Kafka topic, via a DDL statement. Each server has its internal memory (called *metastore*) that keeps track of the streams and tables created.

2. Now that the stream/table is created, you can express your business logic by using an SQL query, via a DML statement.

3. Now the engine has to translate the query into a Kafka Streams application. By doing so, the engine uses a parser (ANTLR [78]), which creates an abstract syntax tree (AST) from the query.

4. The resulting AST is suddenly used to create the logical plan of the query. The logical plan is a tree of operators that represent the query's logic.

5. The last step involves the creation of the physical plan, which is a Kafka Stream application. Starting from the logical plan, the engine traverses the nodes and emits the corresponding API calls. Often, the process ends with a persistent query that writes continuously to its output topic until the query is terminated.

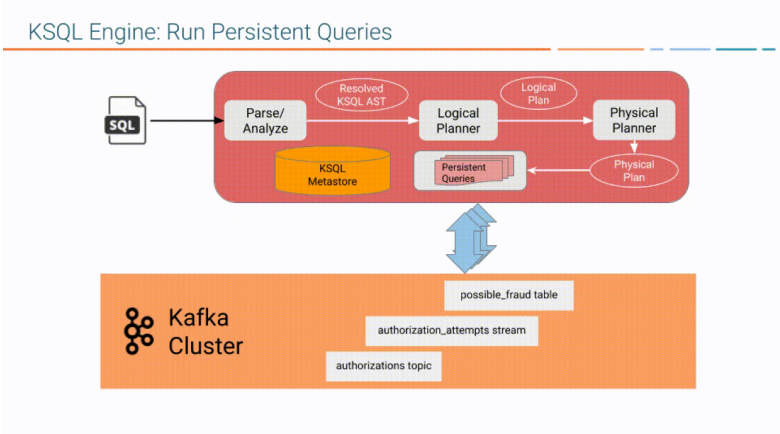

Figure 2.20: ksqlDB Query Lifecycle [77].

Scaling Workloads and Fault Tolerance

You can scale the workload of a ksqlDB application by adding more servers to the cluster (i.e. horizontal scaling) or by adding more resources to the existing servers (i.e. vertical scaling). In both cases, ksqlDB will automatically redistribute the workload without losing any data.
Like Kafka Streams, ksqlDB can well-scale horizontally when resources are allocated and Kafka topics have an adequate number of partitions to enhance parallelism. For instance, if you have a topic with 10 partitions, you can achieve a maximum parallelism of 10. Continuing with this example, the query in question will be executed by (at most) 10 threads concurrently. If you want to increase the parallelism, it is necessary to augment the number of partitions of the undergoing

topic otherwise, if we have more threads than partitions, the extra threads will be idle.

Under the hood, ksqlDB can distribute the workload across multiple servers using the *service id* configuration. This field is used as an identification for the ksqlDB server and it is used to specify which server belongs to which ksqlDB cluster. When a new server tries to join a cluster, it must have the same ID as the other servers. Once this condition is satisfied, the new server will join the cluster and they will start to split the tasks. The mechanism that controls this aspect of ksqlDB, is the same as Kafka client's one. In Kafka, the actor that consumes data (i.e. Kafka Consumer) has two ways to read data from partitions. We can either assign a consumer to a partition or it can join a *consumer group*. A consumer group is a set of consumers (i.e. a daemon that continuously processes data) that work together to consume a topic, under the same ID. Since ksqlDB is based on Kafka, it uses the same abstraction to achieve scale-out. The ability to continuously balance the workload, as soon as a new server joins/leaves the cluster is called *dynamic load balancing* [82].

When a DDL statement is submitted to the server, the metastore is updated and then every new statement is enqueued into a Kafka topic, called *command topic*. It is important to underline the fact that when a client submits a DDL statement, it is sent only to the server linked to that client. Hence, the command topic is used to propagate the statement to the other servers and remain in sync. From this specific, we deduct that every node, even the new ones, in a ksqlDB cluster, must have the same view of the current stream/table. The ksqlDB's protocol for ensuring the same view of the streams/tables across the cluster has three main concepts. Firstly, considers the command topic as the source of truth and includes all the statements that must be executed by the servers to stay up-to-date. Then, the statements are validated before being produced to the command topic. Finally, because of the possibility of race conditions (i.e. two servers that try to write to the command topic at the same time), servers are isolated allowing only one server at a time to validate and produce statements to the command topic. Once again Kafka has been used to solve this problem, by using the *transactions* [81] feature.

In a nutshell, the transactions are designed to ensure the atomicity and consistency of message production and consumption across multiple partitions or topics.
So far, we have seen that ksqlDB is tightly integrated with Kafka, and its fault tolerance is closely tied to Kafka's durability and replication features. As we have seen before, a ksqlDB server is part of a consumer group and it consumes data from a Kafka topic. These consumers tell Kafka which messages they have read, by *committing* their offsets (i.e. the position of the last record read). In order to do so, in case of failure, the broker can command consumers where to resume the processing. However, during stateful operations, it is relevant to keep track of the state of the application (i.e. during an average aggregation). For instance, let us consider the case of a materialized view. For every new incoming row, ksqlDB will update the view and emit the row to a changelog topic (i.e. a topic that contains the history of the changes). Kafka Streams has already implemented this type of operation using a key/value store (RocksDB, which also Flink uses) that contains all the valuable information about the materialized view, in this case. When a server stops working, the new server will take over the changelog topic and will copy the data into its own key/value store. Thus, the new server will be able to resume the processing from the last committed offset. This style of fault tolerance is often called *cold recovery* [80]. With this kind of recovery, ksqlDB can optimize the recovery time using the concept of *compacted topic* [159]. In this way, a Kafka topic is configured to keep only the last record for each key and will not grow indefinitely.

2.5 Materialize

Materialize is a distributed SQL database built on streaming internals. In a similar manner to ksqlDB, this *streaming datawarehouse* [85] has been built for view maintenance (i.e. materialized view) and you can use SQL to build stream processing capabilities. The project saw the light for the first time in early 2020 as a single binary [83] that you could run on a single node. Nowadays, the binary has evolved into a cloud-native platform [84] with built-in horizontal scalability.

Overview

Materialize is a streaming data warehouse built on the principles of interoperability and consistency.
Materialize lets you perform complex computations over streaming data using the PostgreSQL language, giving you the declarativity of a SQL data warehouse, in an almost real-time fashion, abstracting away the complexity of the underlying streaming services. To address this, Materialize needs to be connected to the sources of data you want to query. Subsequently, you can define materialized views using the PostgreSQL syntax. These views could be seen as queries that you want to repeatedly perform. Finally, you can operate over these views always with SQL. Once you have defined the views, Materialize will continuously update them as new data arrives. The name "streaming data warehouse" derives from the fact that Materialize reflects data that was originally written elsewhere. Although Materialize supports INSERT statement (in a limited way [97]), it is not designed to be a system of record, like an OLTP database. Materialize relies on external sources to provide data that are already ordered, for example from Apache Kafka, AWS Kinesis, or local files.
Views are the main concept of Materialize. In classical SQL databases, views are a way to create a named reference for a SELECT statement, making it easier to use complex queries. In most systems, this is essentially a workaround for executing the underlying SELECT statement without providing performance benefits. The goal of Materialize is to provide a way to maintain views in a streaming context,

without the need to recompute the view every time the data changes, overcoming the limitations of traditional databases (i.e. limitations in terms of refresh rate and supported syntax). The method designed by Materialize engineers to achieve this goal is to leverage a powerful computational framework known as *Differential Dataflow* [88].

This framework allows Materialize to react to changes in the data, and incrementally update the views. Differential Dataflow is a Rust library that provides a way to express incremental computation in a timely dataflow system. Indeed, the Differential Dataflow library is built on top of *Timely Dataflow* [87], a powerful stateful stream processing framework.

Timely Dataflow is a distributed stream processing framework, which enables the definition of dataflow graphs of arbitrary stateful operators that receive inputs and produce outputs. This framework takes care of scheduling, message-passing, scaling, and advancing operators' progress as the data changes. Despite timely dataflow being a powerful framework, it does not provide a way to express incremental computation: for this reason, it has been implemented on top of, a library that extends the capabilities of Timely Dataflow. One of the most important Differential Dataflow's key design decisions is to send only data differences, making it highly efficient. It reacts to changes, handles bursts of data, and conserves resources when there are no changes. This approach enables support for arbitrary updates and deletes in input streams.

The data structure used by Differential Dataflow is called *arrangement* [98]. After the user has defined the views through SQL, Materialize will transform it into a Differential Dataflow graph. As data flows in from the sources, the Differential Dataflow graph will be updated, and the views will be maintained. Then, at the moment of the user's request, Materialize will serve the results directly from the dataflow's arrangements, decreasing the latency.

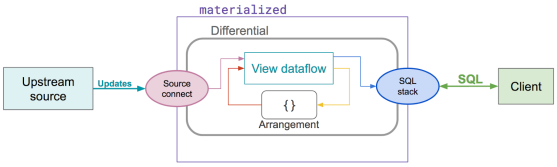

Figure 2.21: Mechanism overview of Materialize's materialized views [96]

In the Figure above 2.21, we can see a high-level overview of Materialize's mechanism for maintaining materialized views. In this example, we have only one source, that continuously sends data, and a view, that encapsulates the data flow and its arrangement.

Timely Dataflow

Timely Dataflow is a low-latency cyclic dataflow computational model, firstly introduced in [86], from internal research in Microsoft, and then implemented in [87], in Rust. The main goal of Timely Dataflow is to provide a way to express those kinds of computations in a declarative way, without worrying about the parallelization and distribution of the computation. This programming model is mainly designed for distributed and parallel data processing. As reported by [89], this dataflow could be a suitable solution for computations that can be parallelized/scaled (either multiple threads in a single node or multiple nodes in a cluster), and you have a stream of data that needs to be processed as soon as they arrive. Hypothetically, this project aims to be the solution for expressing anything that can be parallelized. However, have some limitations especially when you do not have to move data around [89].
What distinguishes Timely Dataflow from other dataflow models is the ability to express much more complex computations (i.e. iterative computations, dataflow with cycles), and a different approach to the dataflow execution strategy. This methodology proposes to slice the dataflow up so that all workers have a view of the entire dataflow graph and each worker is responsible for a proportional fraction of each of the operators in the graph. In this way, workers share responsibility and even the resources of the system.

This concept is graphically represented in Figure 2.22.

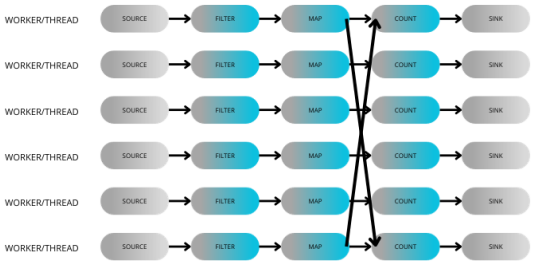

Figure 2.22: Sharing responsibility principle in Timely Dataflow graph. As it is possible to see, each worker has the same view of the dataflow graph, and some workers need to communicate with each other to complete the computation.

There are two critical concepts in this architecture: *timestamps* and *dataflow*, that bring together lead to the concept of *progress*. While the first two notions have already been discussed, the last concept, progress is the ability to reason about the completeness of the computation, using both timestamps and the dataflow paradigm. In Timely Dataflow, what provides the correctness of the computation is the *progress tracking protocol* [86].

Progress Tracking Protocol

Similarly to the watermarks, the progress tracking protocol is a way to track the progress of the computation. A Timely Dataflow computation can be described as a graph of operators, linked by channels. In a system (it could be also a cluster), each worker runs an instance of the entire dataflow graph. Every instance of an operator is in charge of a subset of the data under processing. Workers operate autonomously and solely exchange information through message queues, essentially functioning as communicating sequential processes [90]. Each worker alternately executes the progress tracking protocol and the operator's processing logic.
Pointstamps are the mechanism used to uniquely identify and timestamp data in a dataflow system. It is identified as the tuple $\langle l, t \rangle$ where l is a location in the dataflow and t is a timestamp.

An operator has a variable number of input and output ports, which are *locations*. Typically, the operator receives data through its input ports (target locations), performs some processing, and produces data through its output ports (source locations). A dataflow channel is an edge from a source to a target. Internal operator connections are edges from a target to a source, which are additionally described by one or more summaries: the minimal increment to timestamps applied to data processed by the operator. Throughout the processing of a dataflow graph, all the instances must be informed of which timestamps they may still receive from their incoming channels, to determine when they have a complete view of data associated with a certain timestamp. The progress tracking protocol tracks the system's pointstamps and summarizes them to one frontier per operator port. A *frontier* is a lower bound on the pointstamp that may appear at the operator instance inputs. Progress tracking computes frontiers in two steps. A distributed component exchanges pointstamp changes to construct an approximate, conservative view of all the pointstamps present in the system. Workers use this global view to locally propagate changes on the dataflow graph and update the frontiers at the operator input ports. The combined protocol asynchronously executes these two components [91].
Different from Flink, which continually asks operators if their output contains certain timestamps (check whether their output contains events with timestamps up to a certain watermark), this method reduces the amount of communication between the system and operators.

Differential Dataflow

On top of Timely Dataflow, Materialize has built Differential Dataflow [88]. Differential Dataflow was first created to address the problem of incremental computation in a (timely) dataflow system. As we have seen in the previous section, Timely Dataflow is a powerful computational model. However, it does not provide a way to express incremental computation. Differential Dataflow is a way to express incremental computation in a timely dataflow system. It provides a framework that specifically focuses on handling incremental updates to data, minimizing

the amount of recomputation needed. Differential Dataflow has been designed to work within scenarios where data changes are frequent, and it is required to update results based on these changes. The changes are depicted as multisets of records that store information in terms of $\langle data, time, multiplicity \rangle$ where *data* is the data itself, *time* is the timestamp, and *multiplicity* is the number of copies. For example, if we have a record $\langle 1, 2, 3 \rangle$, it means that we have three copies of the record 1 at time 2. The value of multiplicity could be negative, which means that we have to remove the record from the system. This triple-indexed structure is called *arrangements* [98], and they are used to maintain an intermediate state in memory, grouping key-value pairs for optimal use by data consumers. Arrangements can be shared among views, significantly enhancing the efficiency of a collection of views. This in-memory data structure, the *arrange*, accepts an immutable block of data as input and compacts them.

Architecture

Materialize's first architecture was a single binary that you could run on a single node. This first version comprehended the SQL client shell, the Materialize daemon, and the source connectors. Figure 2.23 shows the architecture of the first version of Materialize.

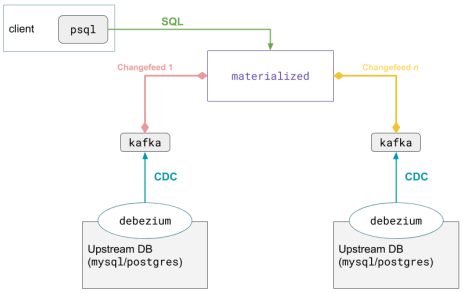

Figure 2.23: Materialize architecture [92]

Starting from the SQL Shell client, you can send SQL queries to the Materialize daemon, defining sources and views. Although not as powerful as the SQL Shell client, there are various library clients to interact with Materialize, such as a Java

client libray [93]. The sources are the data sources that you want to ingest into the system. Then, in Figure 2.24, we can see the internal structure of the Materialize daemon.

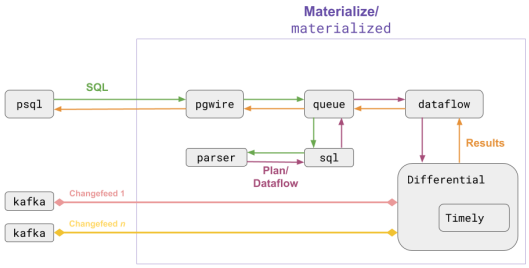

Figure 2.24: Materialize architecture zoom [92]

The interactivity layer is provided by *psql*, which runs on the client side and uses the PostgreSQL wire protocol (*pgwire*) to communicate with the running daemon. After the statement has been enqueued in the *command queue*, a *sql* thread parses the statement and transforms the definition into a differential dataflow graph by the query engine. The dataflow, that represents the query plan, is then sent to the *dataflow layer*, which is responsible for executing the query. The *dataflow layer* is composed of two main components: the *Differential Dataflow layer* and the *Timely Dataflow layer*. Those two layers are responsible for the actual execution of the query. In particular, as soon as the data arrives, the Differential workers will decide whether to process the new data or not, and if so, they will update the maintained views.

The collection that is maintained by the Materialize dataflow layer, is a multiset of records that store information in terms of $\langle data, time, diff \rangle$ where $data$ is the record update, $time$ is the logical timestamp of the update, and $diff$ is the change in the number of copies of the record (i.e. -1 for a deletion, 0 for no change, and 1 for an insertion).

The data can be quickly accessed by means of an arrangement. In this situation, the collections are indexed $data$ to present the sequence of changes $\langle time, diff \rangle$ that the collection has undergone.

State Storage

A crucial component of a streaming platform is the *state storage engine*. In brief, it is an engine responsible for managing and storing the state of computations. A state is any relevant information that needs to be retained and accessed through time during the processing. For instance, in a stream of user events, the state could be the number of events per user. Having a storage system of this kind is the default choice for state management in streaming systems. However, Materialize has a different approach, and one of the main differences between Materialize and the other two SPSs resides here.

Materialize does not have a state storage system. In particular, does not use RocksDB [149] as a state storage engine, as Flink and Kafka Streams (so, ksqlDB) do. RocksDB uses a log-structured merge-tree (LSM-tree) [141] as the underlying data structure. It supports strong isolation guarantees when performing reads and writes, and does so during multiple concurrent readers and writers while continuing to create (and compacting) the LSM with additional (background) threads [151]. Even though Flink allows slot sharing [150] (meaning that one core may hold an entire pipeline of the job), Flink and ksqlDB resource management permits dividing the available CPU cores to operators and sharding streams across a cluster of cores. By contrast, at a glance, this minimizes data exchange between cores. Materialize shares cores differently. Each core in a Timely Dataflow has a whole copy of the dataflow plan, and each core has a copy of every operator (as depicted in Figure 2.22). This, In contrast, timely dataflow cores are shared differently, intentionally to minimize cross-core data movement at the cost of an increasing engineering complexity on the cooperative scheduling of the cores.

Thus, Materialize has to use all the available cores to process the data and schedule the operators. For this reason, it is not possible to use RocksDB as a state storage engine. Indeed, the compaction phase of RocksDB would be an issue since it would require additional background threads, and the dataflow model of Materialize does not allow this, which means compaction happens in the foreground [152].

Differential Dataflow leverages a custom LSM-like operator called *arrange*. While what the arranges are, has already been discussed in 2.5, the main point here is

that the arranges are sharded among the workers and scheduled cooperatively like any other operator in the dataflow, but with the trade-off of not being durable [153]. To address the persistency issue, it has already been shown that Flink uses the checkpoint mechanism, on the other hand, ksqlDB recreates the entire state from the changelog topic (although it can optimize this process with compacted topics [158]). The solution addressed by Materialize to implement persistence is to write the raw streams to append-only files (and then replicated) [152], an approach favorable to Materialize's dataflow model and the principle of separating compute from storage [154].

Materialize Cloud

In the last quarter of 2022, Materialize announced the cloud version of the platform [84]. The cloud version is a fully managed service that provides a way to run Materialize in the cloud. The main goal of this version is to provide a way to scale the platform horizontally, without worrying about the infrastructure. Since the binary version, which included all the necessary for running a Materialize node, the cloud version has been split into three main components: the adapter, the compute, and the storage (as shown in Figure 2.25).

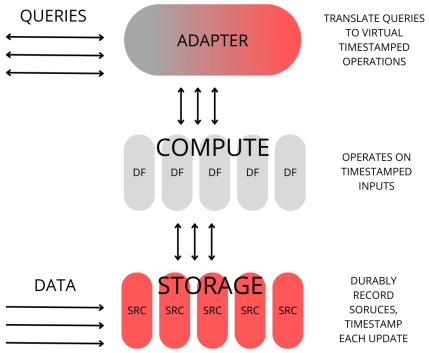

Figure 2.25: Materialize cloud architecture

- **Adapter**: the adapter layer is still where the user interacts with the system.

Likewise the binary version, the user can interact with the system employing the SQL Shell client.

- **Compute**: the compute layer is where the core of Materialize cloud is. It transforms and arranges timestamped inputs, allowing horizontal scaling. It is independent from the storage layer, and thus it can be scaled independently. Moreover, this layer introduces a new concept, called *cluster* that is an independent pool of compute resources, including a standalone dataflow process. For instance, you can have a production cluster and a development cluster, both for different reasons, that will not interfere with each other (i.e. they do not compete for resources). Another property of this "cluster" is that it is fault-tolerant, by increasing the number of replication factors of the underlying data. Finally, you can upscale and downscale the cluster, depending on the workload in a zero-downtime fashion.

- **Storage**: the storage layer is where you pull in the data, it is responsible for data ingestion (the ownership of the data is taken) and data storing (in a compacted form). Along with the responsibilities of the storage layer, there is also to provide a way to durably record the data source that flows in and apply durably timestamps to those data. Each source inside this layer is independent of the others, and it is responsible for its data (in practice, these are stored in Amazon S3 Storage with their Kubernetes pods). This will be the source-of-truth for Materialize and its users. Ingestion occurs even if the above layer Compute is busy, or inactive.

In all three layers, the interaction always involves a *virtual time*, that provides *consistency* guarantees. Materialize cloud furnished *strict serializability* [94] guarantees, which means that the system provides the same guarantees as a single-threaded system, through the virtual time: which timestamps all the data updates that flows through the system and ensures consistency within transactions. As stated in the bullet list above, the storage layer is responsible for assigning virtual timestamps to the data [95].
In the figure below 2.26, we can see how the cloud version is presented to the user.

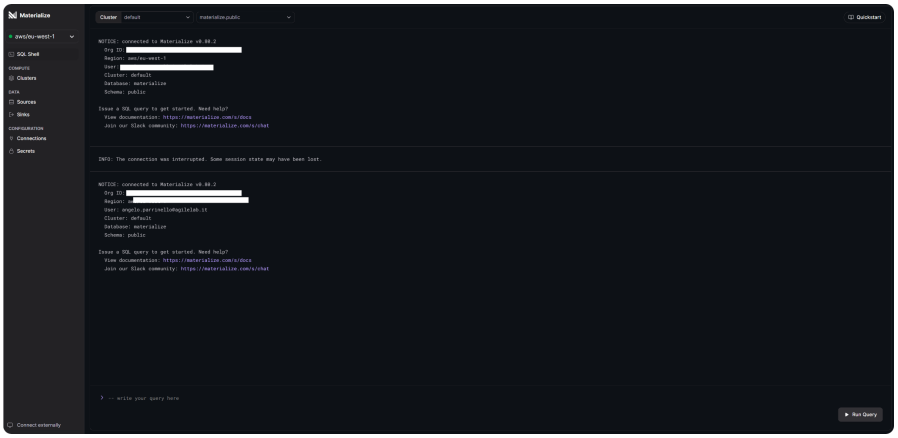

Figure 2.26: Materialize cloud user interface.

Chapter 3

Benchmark

With the term *benchmark* we usually refer to standard tools that allow the evaluation and comparison of different systems, following a set of rules and metrics such as performance, scalability, reliability, etc [1]. Both the qualitative and quantitative aspects of a system can be evaluated through a benchmark. Even though we cannot properly measure, for example, the usability or the maturity of a system, we can still evaluate the time needed to perform a certain task or the number of errors encountered during the development of a specific operation. Especially if it is a company that wants to graduate a new product, it is important to have a benchmark, and then a set of results, that takes account of all the aspects - in particular those that are not directly related to the performance of the system but can still affect the overall quality of the product. In this way, the company can have a clear idea of the strengths and weaknesses of its product, and can also compare it with the competitors.
Some of the key characteristics of a benchmark [2], which have been mostly followed during the realization of this work, are the following:

- **Repeatability**: the benchmark must be repeatable so that it can be executed multiple times and in different environments, and the results must be consistent.
- **Usability**: important to avoid as much as possible the barrier of entry, so that the benchmark can be used by a wide range of users.

- **Relevance**: either we make a benchmark for a specific class of consumers or we make it for a general purpose, the results must be relevant for the target.

This chapter aims to describe the benchmark used in this work to evaluate the three stream processing systems. We will start by describing the state of the art, and then we will introduce the chosen benchmark itself.

3.1 State of the art

Traditionally, workloads and benchmarks have been designed for Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) systems or a combination of both. While OLTP systems focus on day-to-day operations handling a large number of daily transactions (i.e. a system for an online bookstore), OLAP systems are used to analyze large amounts of data to support strategic decision-making (i.e. a system for reporting the total sales of each book genre for the past year, broken down by quarter, to identify trends and plan inventory). The growing importance of OLTP and OLAP systems has led to the development of a large number of benchmarks for these systems. Particularly, the Transaction Processing Performance Council (TPC) [106] has developed two of the most popular benchmarks for OLTP and OLAP systems: TPC-C [100] and TPC-H [99]. Then, the lack of hybrid benchmarks has led to the development of CH-benCHmark [102], which combines both OLTP and OLAP workloads.
On the other hand, we found a lack of standard tools to evaluate and compare stream processing systems. Although the popularity of stream processing systems has grown in recent years, there are only a few benchmarks for these systems.
Linear Road [103] is one of the most popular benchmarks for stream processing systems. It has been the first benchmark for stream processing systems. The benchmark models a toll collection system for a metropolitan area with multiple expressways. The benchmark defines four queries that are used to evaluate the performance of the system. However, one of the queries explained in the benchmark paper is skipped due to complexity. The metric measured is the

latency of the system. The benchmark involved two stream processing systems: Aurora and STREAM.

StreamBench [104] proposes a benchmark suite with 7 microbenchmarks (it means 7 queries) on 4 different synthetic workload suites generated from real-time weblogs and network traffic. The evaluation is based on three metrics (throughput, latency, and fault tolerance) over Storm and Spark. Even though the benchmark covers a wide range of applications, it does not provide queries with complex typical stream processing operations such as windowing. StreamBench uses Apache Kafka as a message broker for decoupling data generation and processing.

Nexmark is a benchmark suite for queries over continuous data streams. While an initial draft was proposed in [69], the benchmark was later revised and implemented with Apache Beam [101] and from an internal team at Alibaba [108]. However, in the original paper, no implementation or evaluation is provided, although the authors defined two metrics based on throughput and latency.

The benchmark aims to evaluate the processing speed and accuracy of the stream processing system under test. The benchmark defines 8 queries for 8 different use cases.

Another extremely popular benchmark, that simulates an advertisement application, is the *Yahoo Streaming Benchmark* [109]. Similarly to StreamBench, this benchmark uses Apache Kafka as a message broker. Born as an informal benchmark, it has been taken into account by many companies and research groups. The values measured are the throughput and the latency of Storm, Spark, and Flink. However, the benchmark has shown some inconsistencies [110] as well as a lack of horizontal analysis: presenting only a single query in the domain of advertisement.

In [111] it is presented a first benchmark proposal (*Open Stream Processing Benchmark*) which included Kafka Streams, Spark, and Flink. Very close to other previous benchmarks, it defines 4 workloads and uses Apache Kafka as a message broker. The benchmark evaluated how the resources are used by the system under test as well as the throughput and the latency. The benchmark's domain is between traffic and IoT. Nevertheless, the benchmark covers only partial aspects

of stream processing systems.

DSPBench [112] offers 15 different scenarios and leaves the possibility to implement new platforms on that benchmark through a common API with the support of (a little) documentation. This work also provides a good mix of generated and real data. Likewise, the previous benchmark, DSPBench calculates the throughput, the latency, and the resource utilization, furthermore, its backbone is Apache Kafka. The benchmark has been tested on Storm and Spark.

While many benchmarks found in the literature are horizontal (meaning that they are not designed for a specific system or a specific aspect), we found a benchmark called *SparkBench* [113] that is specific to Apache Spark. It is a framework-specific benchmark that includes four types of applications, but only one streaming-oriented. The main goal of this benchmark is to evaluate the performance of Apache Spark and fine-tune the system. The fine-tuning is done also by measuring the throughput of Spark under a stream of ML-like data.

In *CEPBen*[114] more focus is given to the complex event processing systems. The benchmark evaluates the latency of the whole processing pipeline and the degree of complexity of CEP operations such as filtering and pattern matching, during the processing of random synthetic data. The system under test is Esper [176] and the benchmark defines three different cases while the throughput is measured.

Then, we have *RIoTBench* [120], a benchmark for IoT stream processing systems. The suite defines 27 Internet of Things tasks that can be assembled to create micro-benchmarks. Although the project is powerful and interesting (the API has been designed to be extensible), the current evaluation is limited to a single platform (Apache Storm) and the tasks are strongly related to the IoT domain. Even though the benchmark tested only Storm, the authors proposed different metrics along with the throughput and the latency: jitter and resource utilization.

Finally, *ESPBench* [122] is a benchmark for evaluating distributed stream processing systems in the enterprise context, specifically in fields like Industry 4.0 and the Internet of Things. ESPBench covers a wide range of functionalities of SPSs and provides a set of 5 queries that can be used to evaluate the performance

of the system under test. Yet, the benchmark has only one metric (latency) even though has been tested on Flink, Spark, and Hazelcast Jet.
Along with these benchmarks, we found other projects that are not properly benchmarks but are strictly related to the topic. *NAMB* [119] and *SPBench* [123] are two frameworks for generating benchmarks for stream processing systems. NAMB, which means Not-only-A-Micro-Benchmark, is a benchmark suite for stream processing systems designed for generating generic application prototypes. Employing the definition of a simple .yaml file, NAMB can generate the code for testing a specific platform under different scenarios. Despite this, the benchmark is not maintained anymore and the documentation as well as the codebase is not so clear and fresh [117]. On the other hand, SPBench is a framework for generating custom benchmarks for stream processing systems. The goal of SPBench is to simplify the process for programmers to develop parallel code and generate a tailored version for benchmarking stream processing. Since many SPSs leverage the concept of Key-Value stores, *Gadget* [124] wants to evaluate the performance of these systems extremely useful for stateful operations.
In the table 3.1 we summarize the main characteristics of the benchmarks found in the literature.

Benchmark	Queries	Message Broker	Domain	Horizontal	Metrics	SPSs
Linear Road	4	-	Traffic	✓	Latency	Aurora, STREAM
StreamBench	7	Kafka	Weblogs, Network	✓	Throughput, Latency, Fault tolerance	Storm, Spark
Nexmark	**8**	**-**	**Auctions**	✓	**Throughput, Latency**	**-**
Yahoo Streaming	1	Kafka	Advertisement	✗	Throughput, Latency	Storm, Spark, Flink
Open Stream Processing	4	Kafka	Traffic, IoT	✗	Throughput, Latency, Resource utilization	Kafka Streams, Spark, Flink
DSPBench	15	Kafka	Finance, Social	✓	Throughput, Latency, Resource utilization	Storm, Spark
SparkBench	1	-	ML	✗	Throughput	Spark
CEPBen	3	-	Random Synthetic Events	✗	Throughput	Esper
RIoTBench	27	-	IoT	✓	Throughput, Latency, Jitter, Resource utilization	Storm
ESPBench	5	-	Industry 4.0, IoT	✓	Latency	Flink, Spark, Hazelcast Jet

Table 3.1: Comparison of the benchmarks found in the literature. The table refers to the content of the papers. The Nexmark row is highlighted because it is the benchmark used in this work.

Between the benchmarks that we have analyzed in the section before, we have chosen *Nexmark* [69] as the benchmark to use in our evaluation.
Throughout the phase of benchmark selection, it has been discovered that a specific benchmark for SPSs which is able to cover all the aspects of the domain and meanwhile is widely recognized by the community does not exist. So, after gathering insights from all the benchmarks, we have decided to choose Nexmark as the benchmark reference for these reasons:

- All the benchmarks tested Apache Flink and other common stream processing systems that were always the same ones. We wanted to test something different.
- A few of them cover multiple stream processing operations such as windowing. Nexmark shows a good variety of queries.

- The whole plethora of benchmarks is not so much documented or does not provide a real guide.
- Almost no one gives the possibility to add new platforms, nor presents a common API for the implementation of new platforms, or has designed the benchmark to be extensible. Since Nexmark offers fewer functionalities than other benchmarks, it is easier to implement new platforms.
- A relatively unexplored benchmark in the existing literature. No one has used Nexmark to evaluate the performance of a stream processing system.
- There had to be a system-agnostic platform for the whole evaluation process. However, it could be useful to use a message broker (i.e. Apache Kafka) for decoupling data generation from the system under test. This means using the same data generator for all the systems.
- The system must already have a good codebase of where to start. Our initial intention was to implement the benchmark from scratch, but we realized that it would have taken too much time. A Nexmark implementation from Alibaba [108] was found and used as a starting point.
- All the benchmarks are no longer maintained except Nexmark. At the time of writing, the last commit was 5 months ago.
- As much standard as possible. This includes the use of standard and well-established technologies but also being as close as possible to TPC levels. One of the goals of Nexmark is to be a standard TPC-DS benchmark for SPSs [128].
- If possible, the codebase should include at least one of the underlying technologies that we want to test. The repo used as a starting point for the implementation of Nexmark includes Apache Flink.

3.2 Nexmark

The benchmark [69] extends the XMark benchmark [125] to the streaming domain, so is intended for systems that execute queries over continuous data streams. XMark is a standard benchmark for XML databases. It is a synthetic benchmark that generates XML data and queries over it. The revisited version of XMark, Nexmark, is a benchmark for stream processing systems and includes the data model (also other data formats can be used) for streams of an online auction system and provides a set of queries to be executed over those streams.

The team that developed Nexmark, started the project in the first years of the 2000s. The first releases of the benchmark are still available on [126]. However, the project has been abandoned for a long time. Meanwhile, the community has developed other Nexmark projects that are still maintained and improved. Two versions of the benchmark are popular: the Nexmark benchmark suite developed in Apache Beam [101] and the Nexmark implementation developed by Alibaba [108].

Apache Beam [107] is an open-source unified programming model to define and execute data processing pipelines, including batch and streaming processing. The Nexmark-Beam benchmark suite implements all the data streams necessary to run the benchmark as well as the queries. The benchmark can be executed using any of the supported runners, such as Apache Flink and Apache Spark. The Beam version introduced more queries than the original ones, expanding the query set from 8 to 13.

The Nexmark implementation developed by Alibaba [108], the one used in this work, is a Java implementation of the benchmark. The repository includes the data generator, the queries, and the data model. Similar to the Beam version, the Alibaba team added further queries: 8 original queries, 5 queries from the Beam version, and 9 new queries.

Scenario

The scenario adopted for the evaluation is that of the original paper, which is an online auction system (e.g. eBay). Hundreds of auctions for different items are likely happening at any given (and the same) time. New people are registering to the system, bidding on items, and selling items. Overall, the real-time nature of this domain allows the user to test a wide range of queries and operations.
The scenario presents three kinds of data streams passing through the system: *auctions*, *bids*, and *people*. Figure 3.1 shows a representation of the scenario.

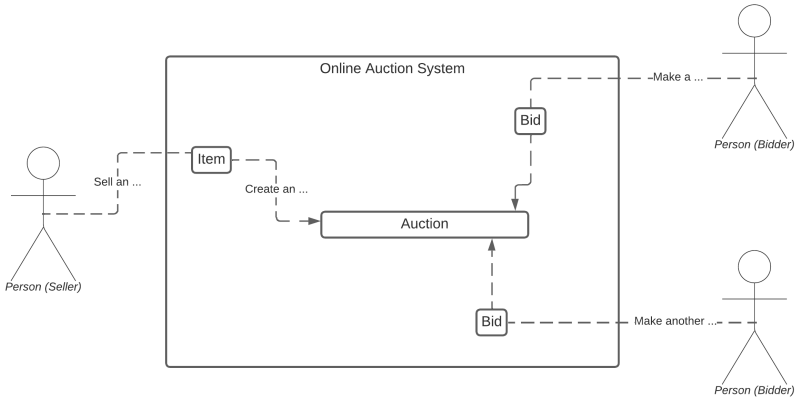

Figure 3.1: Nexmark Scenario.

The online auction system allows bidders to make bids on items put up for auction by sellers. The system has two main actors: a seller and a bidder. The flowchart shows that a seller can create an auction selling an item. Bidders can then bid on the item/auction by making a bid. The diagram also shows that multiple bidders can bid on the same item.

Metrics

In the context of benchmarking, metrics are used to perform a quantitative measure of a particular characteristic of a system. Metrics are a critical component of a

benchmark since they provide a standard way to evaluate and compare different subjects.

As we have seen before, over the years, many metrics have been proposed to evaluate many aspects of an SPS. In the case of Nexmark, the authors of the original paper [69] suggested two metrics to evaluate the performance of the SPSs under test. The first metric is the *Input Stream Rate*, which measures how fast the system can process the input data (i.e. throughput). The second metric is the *Output Matching*, which calculates a sort of processing delay introduced by the system (i.e. latency). Ideally, the system should process the data as fast as it arrives, introducing zero processing delay. This supposition is not realistic since all the systems need some time to perform operations over a stream of data. Let's consider the following query:

```
SELECT bid.price
FROM bid
WHERE bid.itemid = 5192;
```

Listing 3.1: Example query.

In a context of a continuos flow of data, the query will produce a price, that increases over time, every time a new bid is made on that specific item. The stream of price $p_1, p_2, p_3, ..., p_n$ will come into the system at time $t_1, t_2, t_3, ..., t_n$. Then, the system will introduce a delay $d_1, d_2, d_3, ..., d_n$, meaning that the system will furnish a result for the price p_i at time $t_i + d_i$. However, at the time t_i the price will be p_{i-1}, meaning that there is a difference between the ideal price and the actual price. Figure 3.2 shows a representation of the Output Matching metric. The solid line represents the ideal system, while the dashed line represents the actual system. The area between the two lines represents the Output Matching metric normalized by the time.

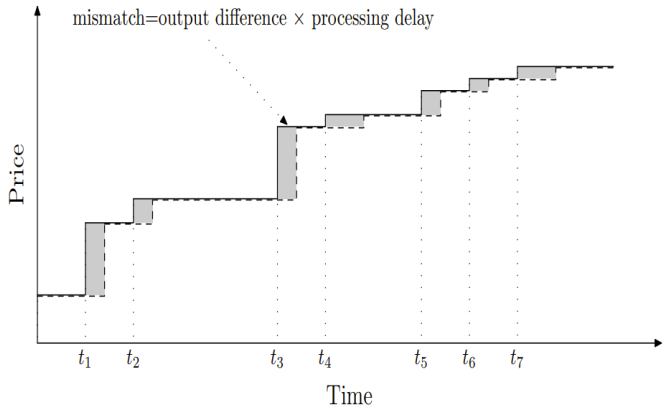

Figure 3.2: Output Matching metric for the example query.

Although the theoretical definition of these metrics is clear, their practical implementation is not. The original paper does not provide any information on how to calculate these metrics, especially the Output Matching metric. In this case, the authors insist on the fact that the metric needs an output difference function (that returns 0 if the result is expected and 1 otherwise) not better specified.
The only information furnished is that in their experiments, an approximation of the Output Matching metric has been used. This approximated metric, measures how long it takes for a relevant tuple to go through the system. While the definition of this second manipulated metric is clearer, it is still too opaque.
By contrast, Nexmark's Alibaba implementation [108] focused on the measuring of resource consumption. In particular, the authors wanted to calculate the CPU usage, during the execution of the queries, and the time needed to process the input data. To measure the first metric, the benchmark extract the CPU usage from every single worker node, consolidating this information in a reporter to provide a summary.

Queries

Nexmark authors define a new class of query called *Stream-In, Stream-Out* queries. These queries are continuous queries that take a stream as input, perform some data operations, and produce a stream as output. The constant flow of data that

characterizes these queries is perfect for testing the performance of an SPS during the execution of a materialized view.

The benchmark defines 8 queries, with the last 4 being the most interesting ones since they are the ones that use windowing. Although the project [108] proposes more queries, some are from Apache Beam [101] and others have been implemented by the authors of the project themselves, to cover more scenarios. However, in this work, we have decided to use the original queries proposed in the paper [69].

Although all the queries are defined by the authors as *Stream-In, Stream-Out* queries, we can newly classify them into two sub-categories: *monitoring* and *analytics*. The first category includes queries that monitor the system and report a piece of specific information about the current state of the system; this class includes the first three queries. Meanwhile, the analytics queries are used to extract and analyze data from the system; this class includes the last five queries.

The queries used are practically the same as the ones proposed in the original paper. The only difference is that they have been adapted to the schema used in the implementation of the benchmark. The meaning of the queries is the same. The queries are presented using the meta SQL-like language proposed in the original paper, but they have been translated into SQL for each system-specific implementation. Each query will create a materialized view that will be used to answer the query.

Query 1 - Currency Conversion

The goal of this query is to test the parse speed of the system. Indeed, what this query does is parse the input stream and convert the price of the bid from dollars to euros. DOLTOEUR is a meta function that takes a price in dollars and returns the price in euros. The query is shown in Listing 3.2.

```
SELECT
    auction,
    bidder,
    DOLTOEUR(price) AS euroPrice,
    entryTime,
    extra
FROM bids;
```

Listing 3.2: Nexmark query 1.

Query 2 - Selection

This query aims to test the selection speed of the system. The query filters all the items with a specific ID. The query is shown in Listing 3.3.

```
SELECT auction, price
FROM bids
WHERE auction = 1007 OR auction = 1020
OR auction = 2001 OR auction = 2019 OR auction = 2087;
```

Listing 3.3: Nexmark query 2.

Query 3 - Local Item Suggestion

The third query wants to test the join functionality. The motivation behind this query is to suggest to the users other items that might be of interest to them, based on a particular category of items that are on sale near them. To obtain this result, the query joins the auctions with the people and filters the auctions based on the category and the location of the person. The original query filtered only

people who lived in Oregon (OR), in this case, the query filters also people who live in Idaho (ID) and California (CA). The query is shown in Listing 3.4.

```
SELECT P.name, P.city, P.state, A.id
FROM auctions AS A
JOIN people AS P ON A.seller = P.id
WHERE A.category = 10
AND (P.state = 'OR' OR P.state = 'ID'
    OR P.state = 'CA');
```

Listing 3.4: Nexmark query 3.

Query 4 - Average Price for a Category

With this query, the authors want to test the aggregation functionality of the system. The query computes the average price of all the items sold in a specific category. The original query had an entity called "closed_auctions" that was used to store all the auctions that had already expired. Since the schema used in this work does not have this entity, the authors introduced a subquery that computes the maximum price of each auction and then computes the average price of all the auctions in a specific category. The query is shown in Listing 3.5.

```
SELECT Q.category, AVG(Q.final)
FROM (SELECT MAX(B.price) AS final, A.category
        FROM auctions A, bids B
        WHERE A.id = B.auction
        AND B.entryTime BETWEEN A.entryTime AND
        A.expirationTime
        GROUP BY A.id, A.category) AS Q
GROUP BY Q.category;
```

Listing 3.5: Nexmark query 4.

Query 5 - Hot Items

For testing a time-based window, the authors propose this query. The original query selects the item with the most bids in the past one-hour time period and shows the result every minute. To make things a bit more dynamic and easier to test, shorter windows have been proposed. Further, the query has been modified and now shows the auctions (the item is included with the auction entity) with the most bids in the last period. The query is shown in Listing 3.6.

```
SELECT A.auction
FROM auctions AS A [RANGE 10 SECONDS PRECEDING]
WHERE (SELECT COUNT(auctions.auction)
        FROM auctions [PARTITION BY auctions.auction
        RANGE 10 SECONDS PRECEDING])
>= ALL (SELECT COUNT(auctions.auction)
        FROM bid [PARTITION BY auctions.auction
        RANGE 10 SECONDS PRECEDING]);
```

Listing 3.6: Nexmark query 5.

Query 6 - Average Selling Price by Seller

Query 6 finds, for each seller, the average selling price for their last 10 auctions. The query is shown in Listing 3.7.

```
SELECT A.sellerId, AVG(B.price)
FROM auctions AS A, bids AS B
WHERE A.id = B.auction
AND B.entryTime BETWEEN A.entryTime AND
A.expirationTime
GROUP BY A.sellerId 10 ROWS PRECEDING;
```

Listing 3.7: Nexmark query 6.

Query 7 - Highest Bid

The original query 7 calculates the highest bids at the last minute. In the version employed, there will be used a shorter window (10 seconds) to help make testing easier. The term FIXEDRANGE is used to indicate that the window should be evaluated every 10 seconds instead of over a sliding 10-second window. The query is shown in Listing 3.8.

```
SELECT B.auction, B.price
FROM bids AS B
WHERE B.price = (SELECT MAX(B1.price)
                 FROM bids AS B1 [FIXEDRANGE 10 SECONDS PRECEDING]);
```

Listing 3.8: Nexmark query 7.

Query 8 - Monitor New Users

The last query selects all the users who have registered in the system and then created an auction in the last period. The original Nexmark Query8 monitors the new users in the last 12 hours, updated every 12 hours. To make things a bit more dynamic and easier to test a shorter window (10 seconds) has been proposed. The query is shown in Listing 3.9.

```
    SELECT P.id, P.name
    FROM people AS P [RANGE 10 SECONDS PRECEDING],
      auctions AS A [RANGE 10 SECONDS PRECEDING]
    WHERE P.id = A.seller;
```

Listing 3.9: Nexmark query 8.

The table 3.2 shows a summary evaluation of the queries' properties, cardinality, and typology.

Query	Type	Output Cardinality	Window
1	Monitoring	92 000 000	No
2	Monitoring	46	No
3	Monitoring	595 539	No
4	Analytics	670	No
5	Analytics	69	Yes
6	Analytics	1 369 574	Yes
7	Analytics	13	Yes
8	Analytics	830 418	Yes

Table 3.2: Nexmark queries summary.

The cardinality is the number of events that the query will output during the execution. This number is derived from an initial test of 100 million events with the configuration used throughout the tests. Must be considered that, while the number of events that the data generator will produce is exact, the query outputs will be slightly different due to a random component of the data generator and the input configuration submitted to the generator. This means that the output cardinality is not exact and can vary from run to run. However, taking into account this configuration, the cardinality is a good approximation of the number of events that the queries will produce running multiple times the generation of the events.

Chapter 4

Implementation

4.1 Technologies and Development Guidelines

In order to develop good-quality software, best practices and guidelines should be followed. Particularly, since this project is developed within the context of the Agile Lab, the guidelines proposed by them [105] as well as the tools, were followed. The software design guidelines defined by the company trace the vast majority of the best practices set out in the literature. In this section, we will describe the most important choices, in terms of tools and best practices, that were followed during the development of the project.

Code Formatting and Analysis

Code formatting and analysis are important aspects of software development. They help to improve the readability of the code and to detect possible errors. To support the development process, it is possible to use tools that automatically format the code and detect possible errors.
For the formatting and the analysis of the code, the tool used is *Checkstyle* [116], a development tool capable of automatically checking the Java code for adherence to a predefined set of rules. Most of the rules are based on [115], a generic Checkstyle configuration that covers a broad range of coding style aspects.

Whereas for the static analysis of the code, *Spotbugs* (ex- FindBugs) [118] is used. This tool leverages static analysis to look for bugs in Java code.

TDD approach

Test-driven development (TDD) is a software development process that relies on the principles of test-first development. The idea behind this approach is to write a test before writing the code that should satisfy the test. More in detail, the process is composed of three steps. During the first one phase, "red phase", you write a test that fails (it must fail because the functionality is not implemented yet); then, during the second phase, the "green phase", you write the minimum amount of code to pass the test; finally, during the third phase, the "refactor phase", you refactor the code to improve its design. This cycling process, red-green-refactor, is repeated until you are not satisfied with the code. This process increases the quality of the code and reduces the number of bugs or time spent on debugging.
The test cases are written using *JUnit* [127], a testing framework for Java and the JVM. Furthermore, since the project involves the use of containerized applications, *Testcontainers* [129] has been used to write integration tests. This library provides lightweight instances of common services or anything else that can run in a Docker container, supporting JUnit tests, and allowing to write integration tests without needing to maintain a large number of cumbersome stubs, or mock objects.

Logging

The mechanism of logging is important to monitor the execution of the application, and a good use of it can help to increase verifiability, observability, and debuggability of the system.
The stack used in this case is *SLF4J* [131] with the *Log4J* setup. The first one is an API designed for giving generic access to many logging frameworks, the second one is a logging component that does the actual logging. Particularly, SLF4J defines five levels of logging (from lowest severity to highest):

- **TRACE**: finest-grained informational events;

- **DEBUG**: fine-grained informational events that are most useful to debug an application;
- **INFO**: generic informational messages that highlight the progress of the application at a coarse-grained level;
- **WARN**: potentially harmful situations;
- **ERROR**: error events that might still allow the application to continue running.

Monitoring

With the term monitoring, we refer to the process of collecting critical metrics about an application. It is particularly important since it helps to ensure that everything is working as expected and to detect possible errors. Monitoring can be done in two different ways: *agent* and *agentless*. An agent is a piece of software that runs on the same machine as the application and collects the metrics; the metrics are then sent to a monitoring server or an external service. On the other hand, an agentless approach is based on the use of an external service that collects the metrics, without the need to install an agent on the machine where the application is running.
For this project, monitoring has covered a crucial role since it was necessary to monitor the performance of the applications. The agent approach has been chosen since it allows the collection of native metrics. The stack used is composed of *Prometheus* [139] with *Grafana* [132], and *JMX* [133] for the monitoring of ksqlDB and Materialize, whilst the monitoring of Flink has been done using its REST API. For what concerns the monitoring of Materialize and Flink, the internal dashboard of each system has also been used. Furthermore, JMX alone was tested for metrics management, but it was not used in the final version of the project, due to the lack of maintaining the historical data of ksqlDB's metrics.
Prometheus is an open-source monitoring system that can scrape target services with periodicity and store the metrics in a time-series database. It is often com-

bined with Grafana, a multi-platform open-source analytics and interactive visualization web application. Grafana allows to query, visualize, alert on, and understand metrics once they are stored.

DevOps techniques

DevOps is a set of principles and practices that aims to improve the collaboration between the development and operations teams. The main goal of DevOps is to shorten the systems development life cycle while delivering features, fixes, and updates frequently. Enhancing code quality, maintaining organizational structure, advocating for rigorous testing, and automatizing repetitive tasks are some of the benefits of DevOps.

Version Control Administration

Version control is a system that manages changes to a project's code over time. It allows to track every modification made to the codebase, and it is particularly useful when working in a team since it is a distributed system and improves the collaboration between developers.
The correct administration of a codebase was made with a distributed version control system called *Git* [138]. The hosting service of the project's repository is *GitLab* [134], a web-based DevOps lifecycle tool that provides a Git-repository manager. The workflow used is the *Feature Branch Workflow* [137], to isolate the development of new features and reduce errors. The main idea behind this workflow is to create a new branch for each new feature and to merge it into the main branch when the feature is completed. Contrary to the famous *Gitflow Workflow* [136], this workflow does not have two main branches, but only one, the `master` branch. Figure 4.1 shows the workflow employed in this project.

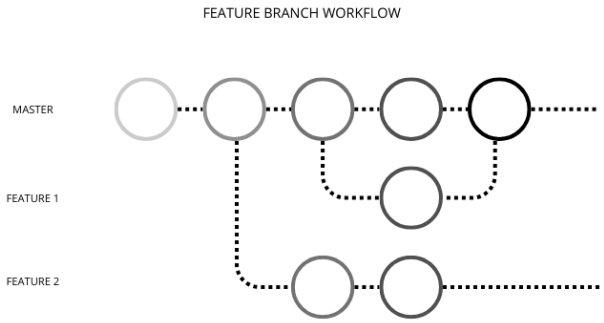

Figure 4.1: Feature Branch Workflow.

Moreover, the *Conventional Commits* [135] specification has been used to enforce a standard for commit messages. This specification defines a set of rules for creating an explicit commit history, which makes it easier to write automated tools on top of it. The commit message should be structured as follows: `<type>[optional scope]: <description>`. The `type` describes the kind of change that this commit is providing (i.e. `feat`, `fix`, `docs`), the `scope` is optional and describes the part of the codebase that has been changed, and the `description` is a short description of the changes.

Build Automation

The term *Build Automation*, refers to the process of automating the tasks required to build and distribute software. Fundamental part of the software development process because it helps to reduce the time spent on repetitive tasks and to improve the quality of the software. Along with the tasks that a build automation tool can perform, we found: dependency management, compiling source code, packaging the compiled code into a deployable format, running automated tests, and deploying the software to production.
One of the most established build automation tools is *Gradle* [142]. Gradle, which has been adopted in this work, is an open-source build automation tool extremely popular in the JVM ecosystem, even though it can be used with other languages. One of the strengths of Gradle is the management of dependency. On one hand, it

permits the management of the dependencies of the project (i.e. libraries, frameworks) simply: you only have to specify the dependency in the `build.gradle` file and Gradle will download it for you. On the other hand, can manage the dependencies of the build tool itself, thanks to the *Gradle Wrapper* [148], which is the recommended way to execute any Gradle build. The Gradle Wrapper is essentially a program that downloads the correct Gradle distribution, defined within a configuration file.

Continuous Integration

The software development practice of *Continuous Integration* (CI) involves the integration of changes in source code into a shared repository frequently. Through the CI process, it is possible to detect and address bugs quicker, improve software quality, automatically run tests, and build the software. Most of the time, this mechanism is integrated with a third-party service that automatically builds and tests, and many other tasks, the codebase every time a change is pushed to the repository.
From the beginning of the project, the CI process has been integrated with the version control system. Since the service used for the hosting of the repository is GitLab, the CI process has been implemented using *GitLab CI* [145]. To construct your CI pipeline, you must create a `.gitlab-ci.yml` file in the root directory of the repository. This file contains the configuration of your workflow (pipeline), made up of *jobs* and *stages*. A job executes a specific task, such as running tests, while a stage is a group of jobs (i.e. build, test, deploy). The *Gitlab Runner* [147] is the main component of Gitlab CI architecture and it is responsible for running the jobs in the pipeline. Runners can work either locally or in a specific container. Another important piece of Gitlab CI is the *Gitlab CI Component* [146]. These components offer the possibility to reuse a single pipeline configuration unit.

Containerization

Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. The com-

ponent that makes this possible is called *container*. The concept of the container is similar to that of a virtual machine, but it is more lightweight, portable, and, most importantly, includes all the environments needed to run a single application. There is only one dependency for the container to run on a host machine, the *container engine*. The container engine is a software capable of running and managing containers. Another relevant aspect of containerization is the *container image*. The container image is a file that includes all the dependencies and what is needed to run the application. The container engine uses the image to create a running instance of the container.
The containerization technology used in this project is *Docker* [144]. Docker is a container engine able to run containers on most common operating systems in a relatively simple way. To create a new (Docker) image (and consequently a container), you have to write a `Dockerfile`, a script that aims to automate the creation of the image, defining step-by-step what is needed to run the application. Although Docker offers the possibility to create containers starting from a single personalized image, it is not the best solution when you have to manage a large number of containers. In this case, it is better to use an orchestration tool, such as *Docker Compose* [143], which allows you to define and run multi-container Docker applications. Likely the `Dockerfile`, to (un)deploy several containers, you have to write a single declarative file, called `docker-compose.yml`, that specifies which containers should be created, how, whether they should be linked to each other, and so on.

4.2 Data modelling

The data model that we decided to use is the one of the Alibaba implementation of Nexmark [108], which is slightly different from the original one proposed in [69]. In particular, the original schema had more entities, such as an item entity and category, that could be collected from the auction entity since they do not add any new information.
The schema is composed of three entities: *Auction*, *Bid*, and *Person*. The *Auction*

entity represents a submission of a seller to sell an item. This object is likely used to store information about the seller, the item being auctioned, the opening bid, the reserve price, and auction start and end times. The *Bid* entity represents an entered bid on an auction from a buyer (person). This object has information about the details of the bid, such as the price and the URL, and also the reference to the auction and the person who made the bid. The *Person* entity represents a person who is either selling or bidding on an auction. This object has information about the person, such as name and email address. The entry time represents the time when the person registered to the system. The whole data model has been designed to be as close as possible to the original one in a Java environment.

Figure 4.2: Nexmark Data Model.

4.3 Data generator

The data generation process is a crucial part of the benchmark since it is necessary to generate a large amount of data to test the performance of the system. The data generator follows the structure of the data model, involving a generator for each entity, as shown in Figure 4.3.

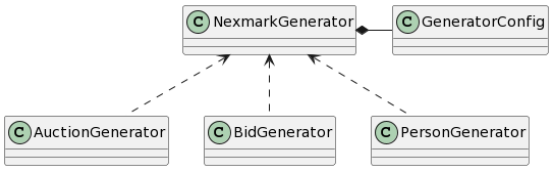

Figure 4.3: Nexmark Data Generator Main Class Diagram.

The main generator is the `NexmarkGenerator`, which is responsible for generating the whole dataset. The dataset is not predefined, but it is generated on the fly, according to the parameters passed to the generator, through the `GeneratorConfig` class. The `GeneratorConfig` class is a simple class that contains the parameters needed to generate the dataset. Some relevant parameters are the proportions of the entities in the dataset (i.e. 50% of the dataset is composed of auctions, 25% of bids, and 25% of persons), the number of events to generate, and the base time from which the events are generated (usually the system's current time). Based on the configuration imposed by the user, the generator can create a dataset with many different characteristics. For example, it is possible to define what is the possibility that a person creates an auction in a short time interval, or the number of out-of-order events.
What compose the principal generator are the generators of the entities. There are three generators, one for each entity, and each one is responsible for generating the corresponding entity (i.e. the `AuctionGenerator` generates the `Auction` entity).
The `NexmarkGenerator` allows the user to define the parallelism degree of the generator itself. What happens behind the scenes is that the generator is split into multiple sub-generators, each one responsible for generating a part of the dataset. Along with the generator, also the `GeneratorConfig` is split into multiple sub-

configurations, each one containing the parameters needed to generate a part of the dataset. This approach allows the generator to be scalable and to generate a large amount of data in a short time. However, the parallelism degree is limited by the number of cores of the machine where the generator is running or the parallelism level of the cluster where the generator is deployed.
The temporal analysis that the data generator allows for its events, spaces between four dimensions: the *event timestamp*, the *adjusted timestamp*, the *watermark*, and the *wallclock timestamp*. In other words, each event has four timestamps associated with it. The event timestamp is the timestamp of the event when it is generated and it is monotonic. The adjusted timestamp is the timestamp of the event when it is generated, but, depending on some configuration parameters may have jitter (it is used to simulate out-of-order events). Then, we have the watermark, which takes into account the jitter in the event timestamp, and the simulated wallclock timestamp.

4.4 Pipelines Stack

The architecture of the project is composed of five main components, each of them with a specific role in the end-to-end data processing and monitoring workflow. The five components are:

1. **Data source**: is responsible for generating the dataset and it is composed of the combination of the Nexmark data generator and a Flink job that streams the data to the storage.
    - **Nexmark data generator**: the data generator that creates the dataset;
    - **Flink job**: a Flink streaming job is responsible for streaming the data to the storage layer, facilitating the process of the data ingestion.
2. **Data storage**: The data storage is the component that stores the data and shares it with the other components. This layer is made up of a Kafka cluster, the well-known distributed streaming platform and message broker. Kafka's

fault-tolerant and scalable nature ensures reliability; apart from that, Kafka granting the possibility of decoupling the data source from the stream processing engine, allowing the data source to be independent from the stream processing engine. Many other similar solutions have involved the use of Kafka as the data storage layer [104] [109] [111];

3. **Data processing**: this level is the key component of the architecture since it is responsible for processing the data and generating the results. This portion of the architecture is composed of the already mentioned stream processing engines: ksqlDB, Flink, and Materialize.

4. **Monitoring**: the monitoring layer tracks the performance of the applications.

    - **Prometheus**: monitoring toolkit that collects metrics from a source and stores them in a time-series way;
    - **Grafana**: analytics and interactive visualization application; it is a big help to query, visualize, and understand metrics once gathered;
    - **JMX**: Java Management Extensions is a Java technology that supplies tools for managing and monitoring applications.
    - **Flink REST API**: the REST API of Flink is used to monitor the performance of the Flink application;

5. **State storage**: responsible for storing the state of the stateful operations within the architecture. This layer is composed of RocksDB, a persistent key-value store for fast storage. The design of this storage system (RocksDB was primarily born for OLTP applications [155] [157]), has allowed the system itself to be largely used for streaming applications.

Figure 4.4 depicts the architecture of the work done.

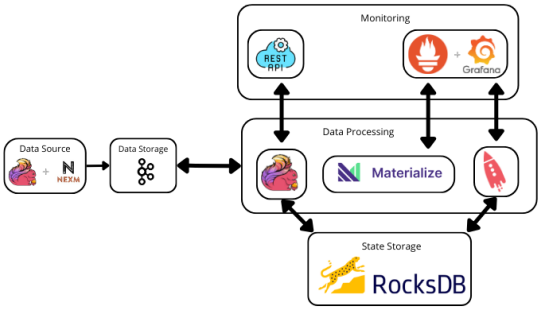

Figure 4.4: Benchmark Architecture.

Inside the architecture of the project, there are three different pipelines, one for each stream processing engine. Each pipeline is composed of all the five components described above. All the scenarios involve the same data source and the same data storage.

Flink Pipeline

The Flink-specific pipeline is composed of the Flink cluster, which executes the stream processing, RocksDB, which stores the state of the stateful operations, and the Flink REST API, which is used to monitor the performance of the Flink application. The pipeline is depicted in Figure 4.5.

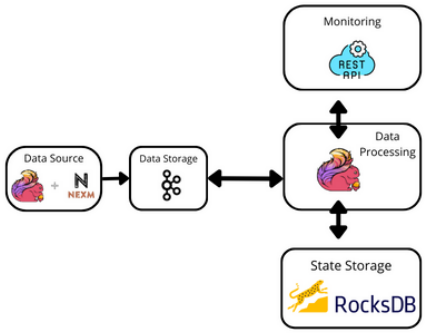

Figure 4.5: Flink pipeline Architecture.

RocksDB has been chosen as the state storage system for Flink because has a better performance during long window operations and with jobs with very large states [156]. Moreover, preliminary tests have shown that the default state storage system of Flink, which saves data as objects on the Java heap, is not suitable for the scenarios of this project [160] [161]. Vice versa, RocksDB is an off-heap state backend and uses storage.
Regarding the monitoring, the internal REST API of Flink has been used to monitor the performance. Furthermore, also the Flink web-based interface has been used to monitor the performance of the application and the state of the running job.

ksqlDB Pipeline

The end-to-end path of the ksqlDB pipeline is composed of the ksqlDB cluster, in charge of data processing, and the Prometheus and Grafana tools used to monitor the progression of the application. Likely the Flink pipeline, the state storage system used is RocksDB. The pipeline is depicted in Figure 4.6.

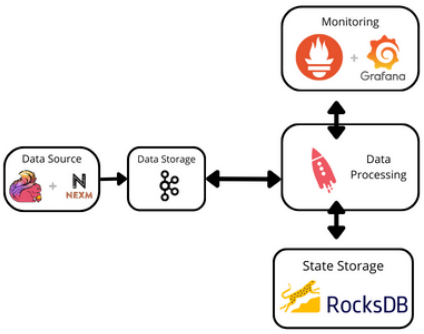

Figure 4.6: ksqlDB pipeline Architecture.

Initially, the monitoring of ksqlDB was done using the JMX metrics alone. However, this approach was not sustainable since the metrics disappeared after a certain amount of time after the end of the task. To avoid to craft a custom

solution to store the metrics, the Prometheus-and-Grafana stack has been used, with a JMX-Prometheus Java agent [175]. This approach allows the creation of dashboards on the metrics and a proper understanding of the execution of the application.
The state storage system used is RocksDB, the same used for Flink. The reason behind this choice is that RocksDB is the default state storage system of ksqlDB, and did not raise any particular problem during preliminary tests.

Materialize Pipeline

Finally, the Materialize pipeline is composed of the Materialize in-cloud solution, assuming the role of the stream processing engine, and the just-seen mix of Grafana and Prometheus, to analyze the status of the task. The pipeline is depicted in Figure 4.7.

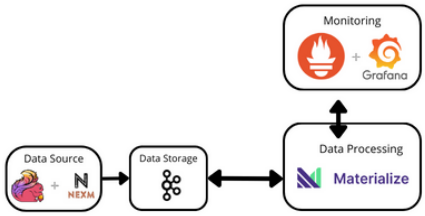

Figure 4.7: Materialize pipeline Architecture.

Materialize allows the alerting, setting threshold, and monitoring of the performance of the application. To do so, it is possible either via the internal dashboard or an external service such as *Datadog* [140] or Grafana. Eventually, both the internal solution and the combo with Grafana and Prometheus have been tested. As regard the internal solution, has been chosen since it is straightforward to use and is already integrated with Materialize letting you monitor the performance of the application without the need to install any additional software.
As stated in 2.5 Materialize uses a different approach to storing the internal state of the activities and, to sum up, this is due to its internal dataflow model. Thus, the state storage is not included in the pipeline since it is internal to Materialize.

Chapter 5

Test

This chapter describes the test system and the results of the tests, created to evaluate the performance of the SPSs under analysis. Both quantitative and qualitative results are presented.
Two main quantitative tests were performed: the first one aimed to evaluate the performance of all the SPSs under test, while the second one wanted to comprehend the behavior of the most promising system with a higher computational power.

5.1 Quantitative results

Test Architecture

The whole test system architecture is founded on *Amazon Web Services* (AWS) [162]. AWS is a comprehensive cloud computing platform provided by Amazon, that furnishes a mixture of Iaas (i.e. Amazon EC2 [163]), PaaS (i.e. Amazon Lambda [164]), and SaaS (i.e. Amazon WorkMail [165]) services. AWS provides a wide range of services, from storage to computing power with a pay-as-you-go pricing model. What concerns the test of Materialize, the already cloud service offered by the company itself was used. The test system architecture is shown in Figure 5.1.

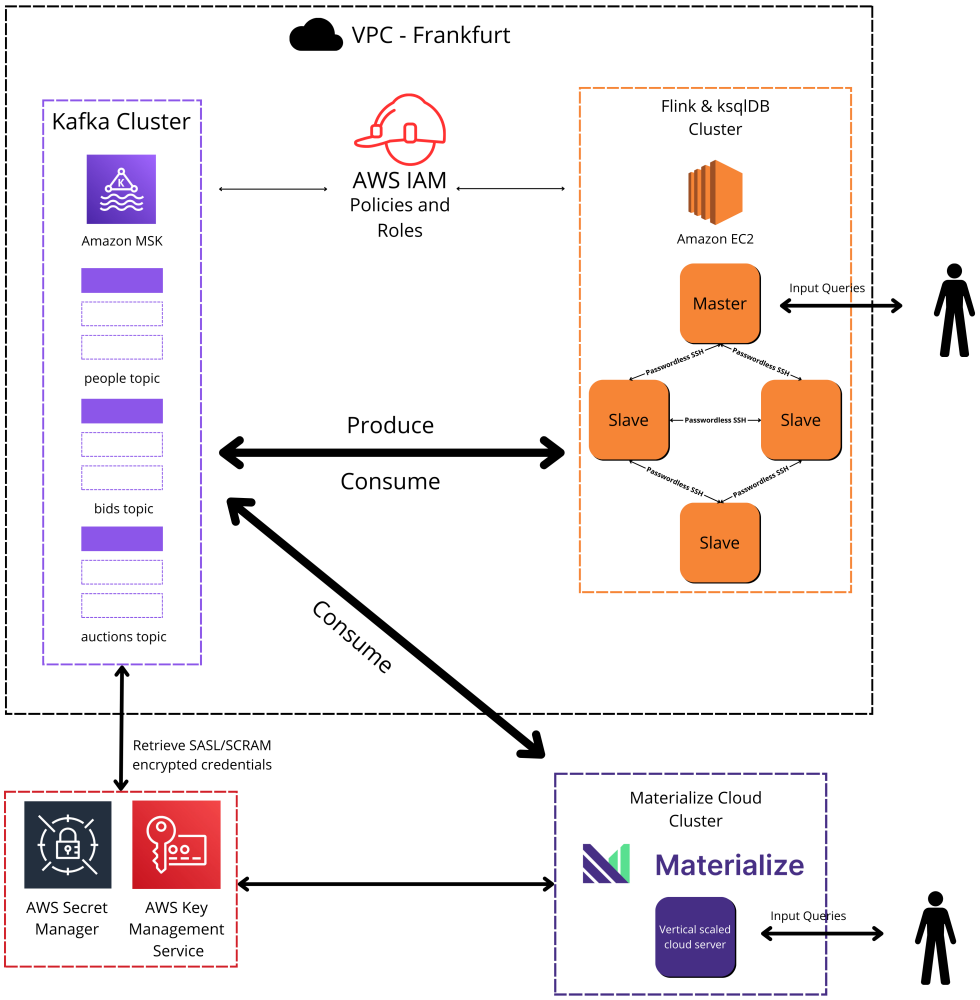

Figure 5.1: Architecture of the test system.

Amazon EC2 Cluster

Flink and ksqlDB were deployed on *EC2* instances. EC2 is a service that provides resizable computing capacity in the cloud. The ease of use and the flexibility of the service allow to quickly scale the resourcing capacity depending on the needs. Two different EC2 instances were used: for the first class of tests, 4 machines; then, only for the most promising system (considering both the qualitative and

quantitative analysis), 11 machines. For both tests, the architecture followed the same pattern: *master-slave*. Thus, in the first case, there were 3 slaves and 1 master, while in the second case, there were 10 slaves and 1 master. The master was used to run the Flink JobManager, ksqlDB Bash client and the monitoring tools. The slaves were used to run the Flink TaskManagers and the ksqlDB servers. This means that the same number of workers have been used for both testing Flink and ksqlDB. All the machines were in the same VPC (Virtual Private Cloud) [168] and could communicate freely with each other due to a passwordless SSH authentication method.

The dimensioning of these instances was made after a preliminary test with a mini cluster configuration and then scaled. This test wanted to evaluate the performance of the system with a small amount of data and to find the right dimension of the cluster. Eventually, this evaluation showed that the initial dimension was not enough (we started from a *t2.small* type) and at least a *m5a.large* type was needed, to run all the components required while having a good performance. Moreover, the size of the cluster had to consider also the dimension of the Materialize cloud service, which leverages a vertical scaling approach. Thus, the instances of the cluster used were a *m5a.2xlarge* type, which has 8 vCPUs and 32 GB of RAM. The OS mounted was *Linux Ubuntu 22.04 LTS*, which is one of the most used OS for cloud computing. Each machine was equipped with a *100 GB EBS* [166] volume, which is a block storage service that provides storage to use with EC2 instances. The system has been deployed in the *eu-central-1* region, which is the Frankfurt region. The Flink's version used was *1.17.1* and the ksqlDB's version used was *0.29.0*. Whereas the Grafana's version used was *10.1.1* and the Prometheus's version used was *2.47.0*. The table below 5.1 resumes what has been said on the EC2 instances so far.

Category	Configuration
EC2 Instance	
Type	m5a.2xlarge
# of instances	4 (3+1) / 11 (10+1)
vCPU	8
RAM	32 GB
Storage	100 GB EBS
Environment	
OS	Linux Ubuntu 22.04 LTS
Region	eu-central-1
Software Versions	
Flink	1.17.1
ksqlDB	0.29.0
Grafana	10.1.1
Prometheus	2.47.0

Table 5.1: EC2 instances configuration

Some of Flink's configuration parameters are shown in the table below 5.2.

Parameter	Value
Memory Configuration	
taskmanager.memory.process.size	28g
jobmanager.memory.process.size	24g
Task Management	
taskmanager.numberOfTaskSlots	8
parallelism.default	8
State Management	
state.backend	rocksdb
state.backend.incremental	true
state.checkpointing.interval	180000
execution.checkpointing.mode	EXACTLY_ONCE

Table 5.2: Flink configuration parameters

Flink allows the definition of most of these parameters either directly in code within your Flink job, via the `flink-conf.yaml` file, or through the command line when submitting the job to the cluster. The first method overrides the other ones. This means that if a parameter is defined in the code (or takes the parameters in input from the command line), it will be used instead of the one defined in the configuration file.
In Flink's configuration, the state volume is saved on the local disk of the machine. This is not the best solution, because if the machine fails the state is lost. In a production environment, it is better to use persistent storage like *HDFS* [174] or *Amazon S3* [167].
The parameters define that per each TaskManager 8 task slots are available (the same number of vCPUs), so the parallelism is set to 8. Overall, the cluster has 24 task slots (8 * 3) available, leading to a total parallelism of 24. This decision implies that to avoid idle tasks, the number of partitions of the input topics must be a multiple of 24. The same reasoning is in the second case, where the number of machines in the cluster is raised.
ksqlDB lets you define the ksql server's configuration parameters in the `ksql-server.properties`. Several of the parameters used are shown in the table below 5.3.

Category	**Configuration**
ksqlDB Configuration Parameters	
ksql.logging.processing.topic.auto.create	true
ksql.logging.processing.stream.auto	true
ksql.streams.replication.factor	3
ksql.streams.producer.acks	all
ksql.internal.topic.replicas	3
ksql.internal.topic.min.insync.replicas	2
ksql.streams.num.standby.replicas	2
ksql.query.pull.enable.standby.reads	true

Table 5.3: ksqlDB configuration parameters

For better fault tolerance, the replication factor of the internal topics is set to 3. This means that each partition of the internal topics is replicated 3 times, across the cluster, but the cluster must have at least 3 brokers. Some other parameters concern the behavior of the ksqlDB cluster under failure. For example, the `ksql.streams.num.standby.replicas` parameter defines the number of hot-standby replicas of the internal state to maintain. Setting this parameter to a value greater than 1 has two distinct advantages. Firstly, we will minimize the downtime of the ksqlDB cluster in case of failure. Secondly, if the parameter `ksql.query.pull.enable.standby.reads` is set to true, the standby replicas can continue to serve pull queries while the primary replicas are unavailable.
Contrary to Flink, ksqlDB lets you define the number of partitions and replicas, of the underlying topic of a stream/table, directly on the query.

Amazon MSK Cluster

The Kafka cluster was deployed on *Amazon MSK* [170] (Amazon Managed Streaming for Apache Kafka). MSK is a fully managed service for Apache Kafka that helps to provision Kafka clusters without having to manage the infrastructure.
The generated cluster was located in the same region and VPC of the EC2 cluster. The cluster was composed of 3 brokers, the default number of brokers for an MSK cluster which ensures high availability. Furthermore, each broker was located in a different availability zone (so three different subnets in the same VPC), to ensure that the cluster was resilient to the failure of a single availability zone. The broker type used was *kafka.m5.large*, which has 2 vCPUs and 8 GB of RAM. This version can handle up to 1000 partitions per broker, which is more than enough for the tests. The amount of storage used was 200 GB, which was enough to keep all the data generated during the tests. The version of Kafka applied was the *3.5.1*, which is one of the latest and most recommended versions.
Access management is one of the most important aspects of a cloud system. It is necessary to grant access to the cluster only to the components that need it. In this case, the EC2 cluster needed to access the Kafka cluster to send and read data from it. To grant access to the cluster's topics, an *IAM policy* [171] was

created. Afterward, the policy was attached to an *IAM role* [169] created for the EC2 cluster. This role was then attached to the EC2 instances. The policy was configured to allow the EC2 cluster to perform all the actions on the Kafka cluster. Lastly, the Materialize cluster needed to access the Kafka cluster too. However, the Materialize cluster was not located in the same VPC of the EC2 cluster. So, this machine accessed the Kafka cluster through a *SASL/SCRAM* [172] [173] authentication method. This method involves the use of a username and a password, and a symmetric key is used to authenticate the client. SCRAM uses a secured hashing algorithm (i.e. SHA-256) and does not share credentials in plaintext over the network (TLS encryption is used).
The table below 5.4 resumes what has been said on the MSK cluster so far, including some of the configuration parameters of the Kafka cluster inside the MSK cluster.

Category	**Configuration**
MSK Cluster Configuration	
MSK Broker type	kafka.m5.large
# of brokers	3
# of availability zones	3
Max # of partitions	1000
vCPU	2
RAM	8 GB
Storage	200 GB
Region	eu-central-1
Kafka version	3.5.1
MSK Configuration Parameters	
auto.create.topics.enable	true
default.replication.factor	3
min.insync.replicas	2
num.partitions	24

Table 5.4: MSK configuration

Materialize Cloud Service

In like manner to the dimensioning study of the EC2 and MSK clusters, a preliminary study was done to understand the dimensioning of the Materialize cloud service. Again, the study was focused on the the necessity to respect the guidelines of the resources required by each system, the willingness to test the systems in as realistic environment as possible, the company's budgeting available, and both the EC2 cluster and the Materialize one had to have similar resources. This was a challenging task because Materialize only leverages a vertical scaling approach, differently from the other systems. So, it was necessary to consider this aspect too. This last requirement was perhaps the most difficult one between the four factors. In fact, on the Materialize documentation, there are only the dimensions available (e.g. S, M, L, XL, etc.) but it is not clear how many resources each dimension corresponds to. Eventually, the resources associated with each dimension were understood and the Figure 5.2 shows the dimensioning of the Materialize cloud service.

size	processes	workers	cpu_nano_cores	memory_bytes	disk_bytes	credits_per_hour
3xsmall	1	1	500000000	4294967296	19327352832	0.25
2xsmall	1	1	1000000000	8589934592	38654705664	0.5
xsmall	1	1	2000000000	17179869184	77309411328	1
small	1	5	6000000000	51539607552	231928233984	2
medium	1	12	14000000000	120259084288	541165879296	4
large	1	27	30000000000	252329328640	1134945107968	8
xlarge	1	56	62000000000	504658657280	2270963957760	16
2xlarge	2	56	62000000000	504658657280	2270963957760	32
3xlarge	4	56	62000000000	504658657280	2270963957760	64
4xlarge	8	56	62000000000	504658657280	2270963957760	128
5xlarge	16	56	62000000000	504658657280	2270963957760	256
6xlarge	32	56	62000000000	504658657280	2270963957760	512

(12 righe)

Figure 5.2: Available resources for each dimension of the Materialize cloud service.

Here is a brief explanation of each column of the table:

- **size**: is the nominal size of the Materialize cluster;
- **processes**: the number of processes in the cluster;
- **workers**: the number of Timely Dataflow workers per process;
- **cpu_nano_cores**: the CPU allocation per process, in billionths of a vCPU core;

- **memory_bytes**: the RAM allocation per process, in bytes;
- **disk_bytes**: the disk allocation per process, in bytes;
- **credits_per_hour**: the number of compute credits consumed per hour.

The cluster consumes credits at a rate proportional to its size. To obtain the final costs, the total number of consumed credits must be multiplied by a cost factor, which varies depending on the region where the cluster is located [177]. For the tests, the region was the same as the EC2 and MSK clusters.
After carefully considering the four factors, the Materialize cluster was sized to be a M dimension. This decision was made since it was a good compromise between all the factors and preliminary tests showed that it was enough to run the queries.

Metrics and Workload Employed

Section 3.2 formally describes the metrics of the benchmark. However, the second metric described in the baseline paper [69] was too vague to be implemented. Moreover, the Output Matching metric needs to support a form of latency measurement, which is not a straightforward task. Latency measures the required time from a record entering the system to some results produced after some actions performed on the record. This may be challenging to support without modifying the original queries.
Despite concentrating the evaluation, also, on resource consumption is a good idea [108] we did not dig into this direction. The reasons are multiple, from time constraints to the willingness to not expand too much the project's scope.
Thus, the metrics utilized in this thesis are the *Throughput*, measured in *bytes/second* and *row/second*, and the *Duration* of the processing of the queries, measured in *seconds*. The throughput is determined by the number of bytes/rows consumed by the system per second. The duration is the time passed from the submission of the query to the system to the end of the processing of the query.
This decision was also made after confronting the metrics used in the state of the

art. Almost all the solutions used a throughput metric. Instead, the duration as a single metric was used only in [108], which is the starting project. Traditionally, DBMS benchmarks measure the length of time to execute a query. Ideally, we could not apply this scheme to data stream query systems, since the input is continuous and theoretically the queries never end. However, in this case, the queries have a finite input, so we can measure the duration of the query.
While these two metrics are a good compromise between the complexity of the implementation and the information provided, this choice of metrics gives us the possibility to opt for a less common set of metrics, resulting in a unique perspective.
The workload used for the tests is shown in the table below 5.5.

Property Name	**Value**
Event Generated	100M
Event Generated per second	$\sim$ 200 000
Bid proportion	92%
Auction proportion	6%
Person proportion	2%
Out of order events	No
Window size	10 seconds
Window slide	2 seconds

Table 5.5: Workload used for the tests

Local tests have shown that the generator can produce more than 1M events per second. However, the bottleneck is the Kafka cluster, the network, and the overhead of the Flink job. Due to this, the data source implemented in a Flink Job and the real environment was able to produce roughly 200,000 events per second.

Preliminary Assessment

Before starting the tests, a preliminary assessment was done to understand some aspects of the systems under test. This assessment was done for multiple reasons. First, to understand the behavior of the systems under test with a small amount of data and seek the right configuration parameters for each system. Second, to ensure the quality of the translated queries. Third, we wanted to evaluate some qualitative aspects of the systems (i.e. the ease of use, the documentation, the quality of the Java client, etc.). While the last reason will be explored in the next section 5.2, the first two reasons are explained here.

Firstly, we needed to choose the best way to handle the partitioning strategies. So far, only the number of partitions of the input topics was considered. Nevertheless, the partitioning strategy is a crucial aspect of the performance of the system. The partitioning strategy is the way the data are assigned to the partitions of the input topics. Kafka provides three main paths: the *key-based* partitioning, the *partition-number-based*, and to create a custom partitioner [178]. The first way is the most common and it is based on the key of the record. We need to specify a key for each record during the serialization phase and, after the value is hashed, a partition will be assigned to the record. The second possibility is to specify the partition number directly. If neither the key nor the partition number is specified, the record will be assigned to a random partition in a round-robin fashion. Finally, the last way is to create a custom partitioner, which is the most flexible way but also the most complex one. The last way was not considered since it was not necessary for the tests, it was totally out of the scope of the thesis: it would have required a lot of time to implement and test it. So, unless it was strictly necessary (i.e. Kafka was a huge bottleneck for the tests), the first two ways were considered. The key-based partitioning provides many advantages among which automatic load balancing but less control over the partitioning; vice-versa, the partition-number-based provides more control over the partitioning but less automatic load balancing. After analyzing the queries, it was clear that the key-based partitioning was the best choice. Indeed, the queries were based on the ID of the events, so it was natural to use the ID as the key of the record. This way,

all the events with the same ID will be assigned to the same partition, avoiding an increasing number of shuffles. In addition, the automatic load balancing was a great advantage that affected our choice.
The second aspect was the translation of the queries. The queries were translated in all the SPSs under test. This task was done manually since there was not found any tool capable of translating each query in its respective SQL dialects. During this phase, some well-known issue was found, in almost all the SPSs. The queries were tested with a small amount of data, in a local environment, to ensure the correctness of the translation.
Here below a table 5.6 shows the translation of the queries in the SPSs under test. The queries are the same as the ones described in the section 3.2, but reproduced with the SQL slang of each SPS.

Query	**Name**	**Summary**	**Flink**	**ksqlDB**	**Materialize**
q1	Currency Conversion	Convert each bid value from dollars to euros.	✓	✓	✓
q2	Selection	Find bids with specific auction ids and show their bid price.	✓	✓	✓
q3	Local Item Suggestion	Who is selling in OR, ID, or CA in category 10, and for what auction ids?	✓	✓	✓
q4	Average Price for a Category	Select the average of the winning bid prices for all auctions in each category.	✓	✓	✓
q5	Hot Items	Which auctions have seen the most bids in the last period?	✓	✗(Not supported due to [180])	✓
q6	Avg Selling Price by Seller	What is the average selling price per seller for their last 10 closed auctions.	✗(Not supported due to [179])	✓	✓
q7	Highest Bid	Select the bids with the highest bid price in the last period.	✓	✓	✓
q8	Monitor New Users	Select people who have entered the system and created auctions in the last period.	✓	✓	✓

Table 5.6: Comparison of the queries' support by Flink, ksqlDB, and Materialize.

The last preliminary test was to understand whether to create the Nexmark generator using directly the *Kafka Producer API* [178] or Flink API. Both the solutions have pros and cons. Either way, generators implement a checkpointing mechanism and have been tested in a local environment in exactly once end-to-end scenario. The producer Kafka is more flexible and can be used with any system,

but it is more complex to implement and presents more boilerplate code than the other solution. On the other hand, the Flink API is more integrated with the Flink system and is easier to implement, even though not suitable for every business case and you need to depend on Flink. The quantitative test wanted to test speed, throughput, profiling (with VisualJVM) and resource utilization (with JMX) during the generation of 10 million events. However, this was not necessary since the test showed that the Kafka producer did not create all the events, while the Flink API did, due to an error in the configuration of the Kafka producer, related to disconnection problems. No further tests were done to better understand the nature of the error. We believe that this would have been an out-of-scope task. Finally, the Flink API was chosen because of its better integration with the Flink system and its suitability for our goals.

Result Analysis

Two main tests were performed: the first one aimed to evaluate the performance of all the SPSs under test, while the second one wanted to comprehend the behavior of the most promising system with a higher computational power and how the system behaves when scaled up.
The first test was aimed to evaluate the performance of all the SPSs so far presented under the conditions described in the sections before. In each plot (figure 5.3) we have reported in the x-axis the queries' names, and in the y-axis the duration, the throughput in bytes, and the throughput in rows, respectively. Each system is represented by a different color, repeated in each plot and each query.
Since duration and throughput are inversely proportional, where the duration is lower, the throughput is higher, and vice-versa. The results show that Flink and ksqlDB have similar performance except for the 4th and 7th queries, where Flink has a better performance. Materialize, in general, has a better performance than the other two systems. While ksqlDB can push its throughput, in terms of rows, similarly to Materialize when facing monitoring queries, gives way to Materialize and Flink in analytical queries. A huge difference, in terms of duration, is shown

in the 4th, 6th, and 7th materialized views, where it is clear that ksqlDB has a worse performance, where the duration is even 10 times higher than the other two systems. Flink's result shows that is the most balanced system, with a good performance in all the queries, but the most promising system, in these terms, is Materialize.

Materialize outperforms the other two systems in almost all the queries and all the metrics. However, the qualitative results, which are shown in section 5.2, will give us a more complete view of the systems under test. Taking into account the results of these tests and the qualitative results, we decided to deep dive into the Flink system, which shows the best compromise between quantitative and qualitative results. Furthermore, since Materialize was the best-performing system in the first test with a great margin, we decided to increase Flink's resources. This was done to assess how much a well-established system must improve to surpass the performance of a new system and to gauge the scalability of the system's performance.

In the second scenario, Flink was tested with a higher computational power and compared to the first test. The plots in figure 5.3 clearly show that the goal of the second test was achieved. In almost all the queries and all the metrics, the performance of Flink has overcome the performance of Materialize or demonstrated a significantly more balanced comparison. With a particular focus on the last 4 queries (except the 6th), the analytical ones with the highest complexity, the performance of Flink is better than the performance of Materialize. It is interesting to note that to exceed the performance of Materialize, it was necessary to add more than 2 times the computational power, highlighting the resource-intensive nature of optimizing Flink's performance. Adding more computational power to the system may enhance its performance, this leads to increased operational costs and management overhead.

The results of both two tests are shown in the figure 5.3.

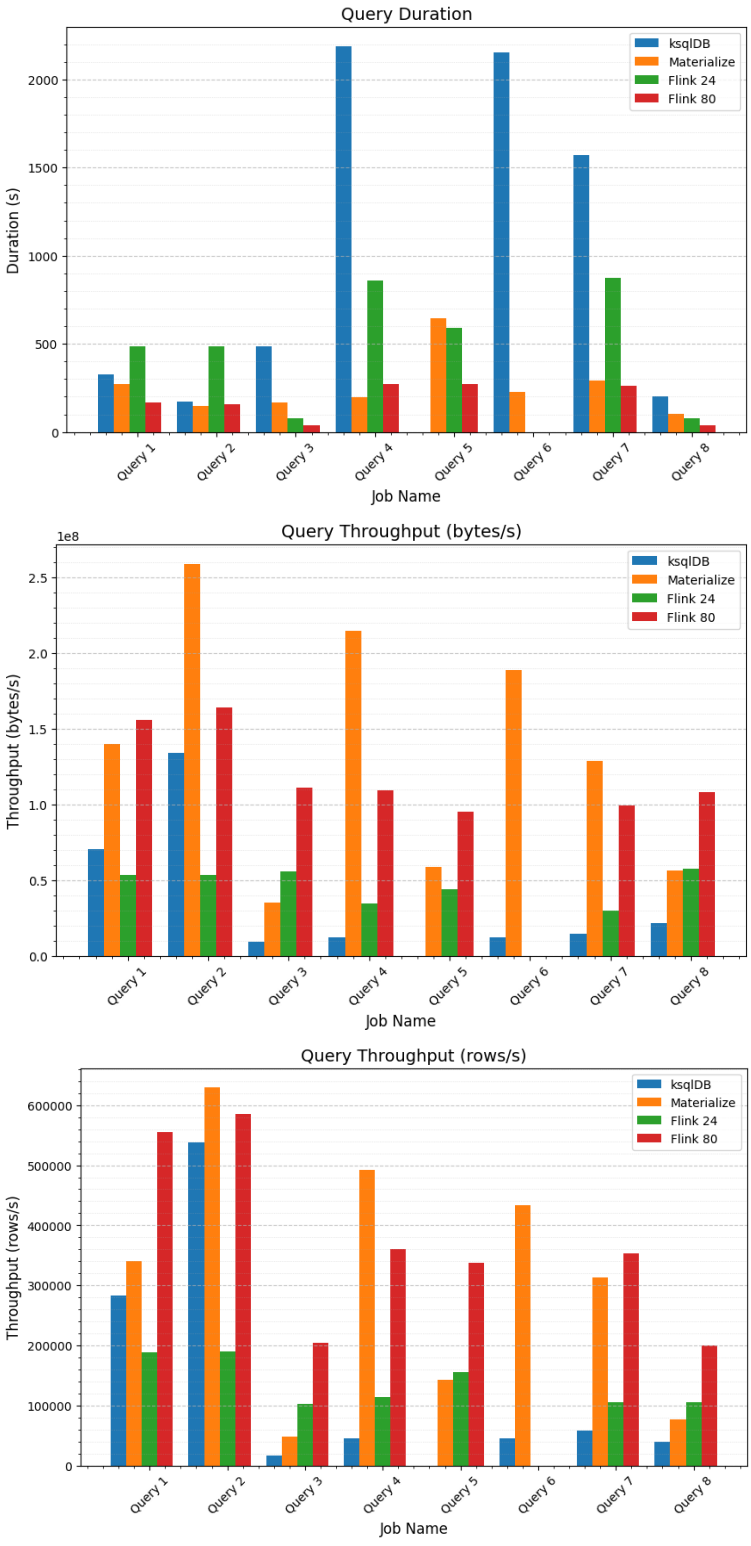

Figure 5.3: Quantitative test results. Both two different versions of Flink are depicted.

As expected, the results show that the performance of Flink has improved. We can see that the duration of the queries has decreased, and the throughput has increased, reaching peaks of improvement of more than 200%. The results show that the performance of Flink is scalable, and it is possible to improve the performance of the system by adding more computational power. However, the improvement is not strictly linear, and the performance of the system is not proportional to the computational power. This is a common behavior of distributed systems and stream processing applications, and it is typically due to the overhead of the communication between the nodes of the cluster or due to the skewed distribution of the key attributes used to group the input tuples. Figure 5.4 depicts the percentage of Flink improvement from the first to the second test.

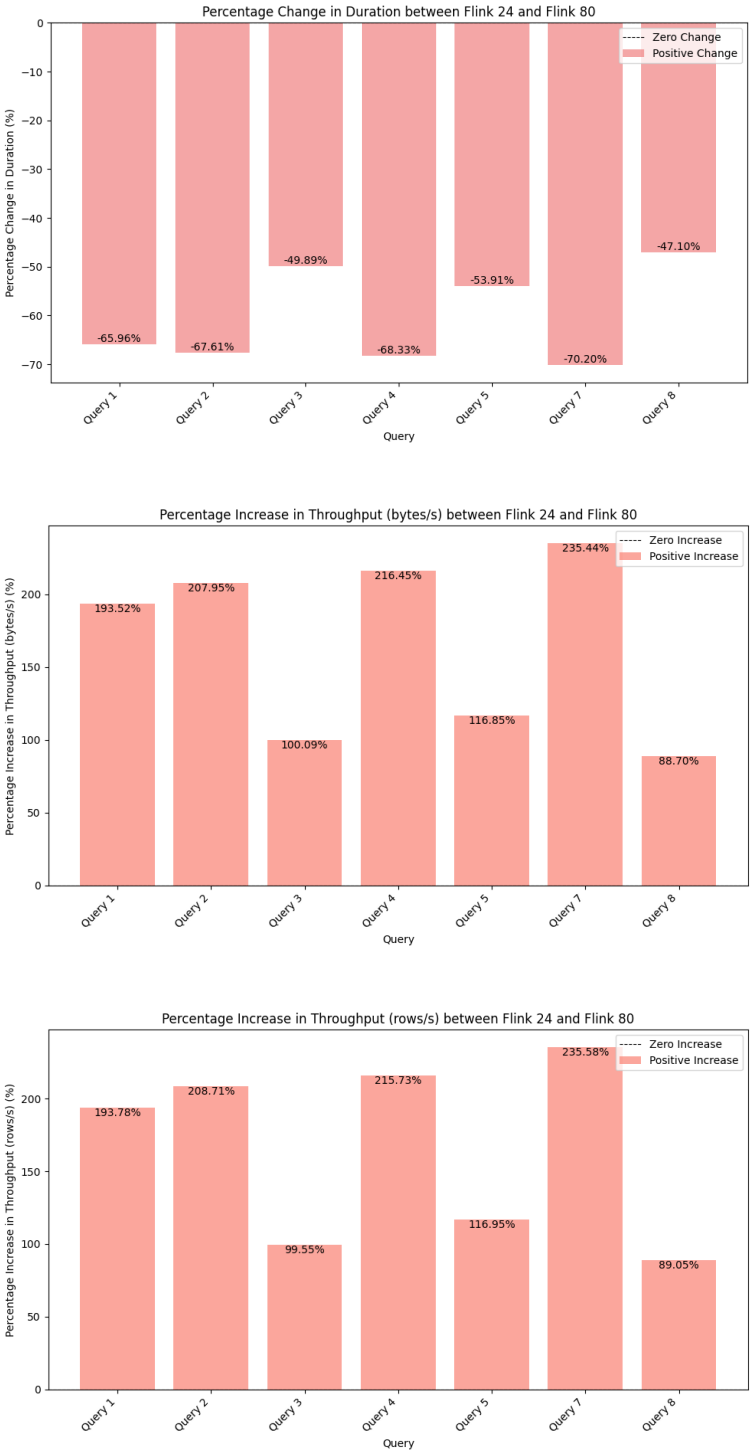

Figure 5.4: Comparison between the results of the first and second tests of Flink.

Query tuning

Query tuning is the process of improving the performance of a query by modifying its structure or the way it is executed. The goal of query tuning is to reduce the duration of the query and increase the throughput.
Since query optimization is a complex and time-consuming task, we decided to tune the queries in case of extremely high duration or low throughput. This was the case with ksqlDB. After the completion of Flink's tests, it was the turn of ksqlDB. From the result of the first query, we noticed that the duration was extremely high compared to Flink. Thus, we decided to tune the query. After some tests, we found that the biggest problem was the multiple deserialization for each query.
Data are pulled out from the Kafka topics in a serialized format (i.e. JSON), and then they are deserialized in a structured format, during the execution of the query. This process was repeated for each event in each query and produced a huge overhead. To solve this problem, we decided to deserialize all the data before introducing them into the queries. This way, the data were deserialized only once, concentrating on a single phase the overhead of the deserialization and the crafting of virtual stream/tables of ksqlDB, and the queries were able to work with the de-structured data. The query refactor is shown in the listing 5.1.

```
CREATE OR REPLACE STREAM nexmark_q1
WITH(KAFKA_TOPIC='RESULT_QUERY1', FORMAT='JSON') AS
SELECT
    actualEvent->actualEvent->auction AS auction,
    actualEvent->actualEvent->bidder AS bidder,
    actualEvent->actualEvent->price * 0.908 AS price,
    actualEvent->actualEvent->entryTime AS entryTime,
    actualEvent->actualEvent->extra AS extra
FROM bids
EMIT CHANGES;
```

Listing 5.1: Before Refactor

```
CREATE OR REPLACE STREAM nexmark_q1
WITH(KAFKA_TOPIC='RESULT_QUERY1', FORMAT='JSON') AS
SELECT
    auction AS auction,
    bidder AS bidder,
    price * 0.908 AS price,
    entryTime AS entryTime,
    extra AS extra
FROM bids
EMIT CHANGES;
```

Listing 5.2: After Refactor

Listing 5.1: Example of ksqlDB query refactor.

Before completing the refactor, we tested again the performance of the first query and the results are shown in the table below 5.7.

Metric	**Before**	**After**	**Improvement**
Duration	3632s	325s	-91%
Throughput (bytes)	10.4MB/s	70.4MB/s	577%
Throughput (rows)	25.33K/s	283K/s	1017%

Table 5.7: Results of the first query of ksqlDB before and after the refactor.

5.2 Qualitative results

The qualitative analysis was done to understand the intrinsic characteristics of the systems under test. The analysis focuses on certain aspects crucial for Agile Lab.
The table below 5.8 summarizes the results of the qualitative analysis. Then, the results are explained in the following sections.

Aspect	Flink	ksqlDB	Materialize
Exactly-Once Semantics	Supports out-of-the-box with checkpointing	Supports with configuration	Supports towards sinks by default (with Kafka/Redpanda)
Ease of Use	Smooth development process, well-documented, active community	Development process more viscous, good documentation but sometimes misleading, useful `docker-compose` for testing	Most problematic development process, documentation broad but not clear/complete, guides need improvement
SQL Dialect	Wide range of functions/operators specific to stream processing	Limited set of functions/operators, better integration with Kafka	Offers a wide range of analytical functions/operators but problematic with windowing and complex queries
Learning Curve	Steeper learning curve initially, smoother with experience	Gentler learning curve, intuitive interface	Gentlest learning curve initially, steeper with complex tasks
Java Client	Mature and complete	Good starting point, needs improvement	Leverages PostgreSQL JDBC driver no specific Java API UDFs are not supported

Table 5.8: Summary of Qualitative Analysis

Exactly-Once Semantics

The first aspect was the *exactly-once semantics*. Due to the customers of Agile Lab, it is crucial that the systems under test support exactly-once semantics. However, the current state of the test does not fully support this requirement. Indeed, to test the exactly-once semantics, we should have tested the systems under a failure scenario, which was not possible at the moment. Nevertheless, we wanted to test the support of this feature. While the theoretical part of the exactly-once semantics has already been covered, we will focus on the practical part.
The first system to be tested was Flink. Flink supports exactly-once semantics out of the box, and it is one of the most mature systems in this sense. Flink can thus guarantee the consistency of the job as long as the checkpointing mechanism is enabled, without worrying about the coding complexity of the application. So,

the only thing to do is to enable and configure the checkpointing mechanism. Even though Flink does not enable by default the checkpointing mechanism, it is possible to enable it programmatically. An example of how to enable the checkpointing mechanism is shown in the listing 5.2.

```
env.enableCheckpointing(180_000);
env.getCheckpointConfig()
   .setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig()
   .setMinPauseBetweenCheckpoints(1000);
env.getCheckpointConfig()
   .setCheckpointTimeout(180_000);
env.getCheckpointConfig()
   .setTolerableCheckpointFailureNumber(0);
env.getCheckpointConfig()
   .setMaxConcurrentCheckpoints(1);
// Must restart a job from a checkpoint to avoid duplicate events
env.getCheckpointConfig()
   .setExternalizedCheckpointCleanup(CheckpointConfig
            .ExternalizedCheckpointCleanup
            .RETAIN_ON_CANCELLATION);
```

Listing 5.2: Example of how to enable the checkpointing mechanism in Flink.

Through Flink, it has been necessary to implement a custom source that implements the possibility of recovering from a failure. In other words, the source had to implement the `CheckpointedFunction` interface. The Nexmark generator Flink source implements two methods (listing 5.3):

```
void snapshotState(FunctionSnapshotContext context) throws Exception;
void initializeState(FunctionInitializationContext context) throws Exception;
```

Listing 5.3: Methods of the `CheckpointedFunction` interface.

Whenever a checkpoint is triggered, the `snapshotState` method is called. Whereas, when the job is started or restarted, the `initializeState` method is called.
The second system to be analyzed was ksqlDB. As we have already seen, ksqlDB highly depends on Kafka and Kafka Streams. And, since Kafka Streams supports both exactly-once and at-least-once semantics, ksqlDB supports them too. It

supports it using topics and transactions as stated in 2.4. Even though at-least-once semantics are the default, it is possible to enable the exactly-once semantics. All is need to do is to set the `processing.guarantee="exactly_once_v2"` in the ksqlDB server configuration. Both the processing and sinking will be guaranteed to be exactly-once. What distinguishes ksqlDB from Flink, is that all is perfectly integrated with Kafka, so it is not necessary to implement a custom source or sink because you control the full ecosystem. Whilst in Flink you might have to integrate more systems. For example, if you are joining two streams from two different sources, you have to ensure that both sources support exactly-once semantics so Flink must ensure that this will be an exactly-once operation.

As with the other systems, Materialize supports exactly-once semantics. At the time of writing, Materialize supports exactly-once semantics towards sinks, with Kafka and Redpanda [185] by default. To achieve exactly-once semantics, Materialize uses an additional topic that is shared among the sinks connected to that specific Kafka cluster. Materialize primarily focuses on ensuring consistency in data streaming systems [186]. While it emphasizes the importance of accurate and coherent results, it explicitly mentions achieving exactly-once guarantees, only for those systems.

So far, we have seen that all three systems support exactly-once semantics. However, the exactly-one processing semantics is not the only thing to consider. To achieve a real end-to-end exactly-once semantics, it is necessary to support it throughout the entire pipeline. So not only the processing section must ensure that, but also the source and the sink. For instance, using Kafka as a message broker, it is necessary to configure it to have at least a replication factor of 3, to adapt the maximum timeout of a transaction as required from the business case, and so on. But also on the consumer side, the downstream consumer must read only committed data and the processing of the data must be idempotent.

Although the systems under test support exactly-once semantics, it is important to notice the pros and cons of each system. Flink supports most flexibility and control over the exactly-once semantics and could require little to a big effort to implement. It mostly depends on what the business case requires. Moreover, has a

lot of compatibility with other systems, so it is possible to integrate it with a lot of systems maintaining the delivery guarantees required. ksqlDB supports almost by default exactly-once semantics and is perfectly integrated with Kafka: as always this is a double-edged sword. Ultimately, Materialize supports this semantic automatically, but, likely ksqlDB, this option is strictly related to the use of Kafka.

Ease of Use

In this section, we will analyze the ease of use of the systems under test. What we take into account in this evaluation is the viscosity of the development process, the quality of the documentation/community, the opacity of the SQL dialect, and the learning curve. The analysis considers the project as a whole, from the development to the deployment, from the point of view of a beginner in the field of SPSs.

In this context, we define the *viscosity* of the development process as the resistance to the development of the system. The viscosity is high when the development process is slow, and the system is hard to develop due also to environmental issues. With Flink, the entire development process was smooth and fast. Starting from the development and the testing in local of the job, to the deployment on the cluster, everything was well documented and easy to do. Moreover, the community is very active. On the other hand, a mature system like Flink has many features and configurations that could be overwhelming for a beginner. Especially for the initial setup of the cluster, it is necessary to have a good understanding of the system and its possible configurations. Plus, using Flink with Kafka, is a must-have to understand Kafka and how to integrate it with Flink. I believe that all these drawbacks can be easily overcome with a good study of the system, and all the errors that someone may encounter, could be supported by online resources. ksqlDB in this sense is less mature than Flink, and the development process was more viscous. The documentation is good but sometimes can be misleading. However, the testing phase was helped by multiple possibilities to run ksqlDB tools. Especially the `docker-compose` file provided by

Confluent was very useful for testing the system in a local environment. Some delays were encountered during the Java client development, due to the lack of documentation and the absence of a ready-to-use Testcontainer for ksqlDB. Then, the deployment on the cluster was not as smooth as Flink, and it was less straightforward than expected. Especially during the configuration phase of the networking and monitoring, some issues were encountered. One of the main scripts of the ksqlDB server (`ksql-run-class`), does not accept correctly some environment variables related to the exposition of the metrics defined in the `ksql-server.properties` and, to workaround this issue, it was necessary to modify the script manually. A similar issue was reported in [192], so I suppose that this is a common issue related to the setup of environmental variables in ksqlDB. A GitHub issue needs to be opened to report this problem. Materialize, from this point of view, is the less mature system. The development process was the most problematic. The documentation is broad but not sufficiently clear and complete, and sometimes wrong. In particular, the connection phase with the MSK cluster was difficult due to multiple problems in the guide that the Materialize team provided. At the time of writing, the guide is a work in progress, and the team is working to improve it and make it better over time, aiming to make the development process more frustration-free. Regardless of these challenges with configuration and connectivity, both ksqlDB and Materialize offer a good trade-off between the viscosity of the development process and the quality of the documentation/community but not as good as Flink. This allows one to reach a good level of understanding of the system in a short time and establish a data pipeline for processing efficiently.

The *opacity* of the SQL dialect is the difficulty of understanding the SQL dialect of the system. The opacity of the SQL dialect is high when the SQL dialect is hard to understand and use. The differences between the SQL dialects of the systems are more evident when the queries are complex. All three dialects follow the SQL standard [193] so the base syntax is the same. However, the differences are in the extensions and the optimizations that each system offers, considering the nature of the system. Flink, for example, offers a wide range of functions

and operators that are not present in the other systems and that are specifically designed for stream processing, with a focus on windowing operations. This leads to an easy and intuitive development of the queries, even those complex. ksqlDB, instead, offers a more limited set of functions and operators, but with better integration with the Kafka ecosystem (i.e. topic and partition management) and provides greater support to pull queries. Materialize, instead, offers a wide of analytical functions and operators with a specialization on temporal filters. It is the system with the least powerful SQL dialect. The graphical comparison between the queries is omitted for brevity, but it is possible to find those in the repository of the thesis [194]. The queries in Flink are more understandable and more readable than the other systems, and easier to develop and maintain. On the other hand, the queries in ksqlDB and Materialize are less readable and more complex. This is due to the lack of some windowing operations and the lack of optimization that requires a workaround to achieve the same result. For example, in those queries (especially in ksqlDB's queries), we can find multiple definitions of stream and table in the same query, which makes the query less readable and more complex. Moreover, during the translation phase of Materialize's queries, we had to ask for help from the Materialize team to understand the best way to translate the queries. Thanks to their help, we were able to understand the best way to translate the queries otherwise we would have encountered many problems and eventually we would have had to give up the translation. The nature of the query in ksqlDB explains the minor efficiency of the system in the quantitative tests. An example of the verbosity of the queries in ksqlDB is shown in the listing 5.4.

```
CREATE
OR REPLACE TABLE subnexmark1_q7 WITH(KAFKA_TOPIC='SUB_RESULT1_QUERY7', FORMAT='JSON') AS
SELECT TIMESTAMPTOSTRING(WINDOWSTART, 'yyyy-MM-dd HH:mm:ss') AS window_start,
        TIMESTAMPTOSTRING(WINDOWEND, 'yyyy-MM-dd HH:mm:ss')   AS window_end,
        MAX(price)          AS highest_bid,
        auction
FROM bids
    WINDOW TUMBLING (SIZE 10 SECONDS)
GROUP BY auction
    EMIT CHANGES;

CREATE
OR REPLACE STREAM IF NOT EXISTS subnexmark1_q7_stream (window_start VARCHAR, window_end VARCHAR, highest_bid
     BIGINT) WITH (KAFKA_TOPIC='SUB_RESULT1_QUERY7', VALUE_FORMAT='JSON');

CREATE
OR REPLACE TABLE subnexmark2_q7 WITH(KAFKA_TOPIC='SUB_RESULT2_QUERY7', FORMAT='JSON') AS
SELECT window_start,
        window_end,
        TOPK(highest_bid, 1) AS highest_bid
FROM subnexmark1_q7_stream
GROUP BY window_start, window_end EMIT CHANGES;

CREATE
OR REPLACE STREAM IF NOT EXISTS subnexmark2_q7_stream (highest_bid ARRAY<BIGINT>) WITH (KAFKA_TOPIC='
     SUB_RESULT2_QUERY7', VALUE_FORMAT='JSON');

CREATE
OR REPLACE STREAM nexmark_q7 WITH(KAFKA_TOPIC='RESULT_QUERY7', FORMAT='JSON') AS
SELECT B.auction,
        B.price,
        B.entryTime,
        B.bidder,
        B.extra
FROM bids AS B
            INNER JOIN subnexmark2_q7_stream AS S WITHIN 1 HOURS GRACE PERIOD 15 MINUTES
ON B.price = S.highest_bid[1]
EMIT CHANGES;
```

Listing 5.4: Example of verbosity in a ksqlDB query.

The learning curve associated with Flink can initially become a significant obstacle for newcomers due to the breadth of topics one needs to know before developing a Flink application. The initial phase demands a significant investment of time and effort to comprehend its architecture, programming model, and ecosystem. However, as developers delve deeper and gain hands-on experience, this curve tends to smooth out, with proficiency gradually increasing over time and this time investment paying off in the long run especially when dealing with complex tasks. In contrast, platforms like ksqlDB and Materialize offer a gentler learning curve, particularly when you have to manage straightforward queries. Their intuitive interfaces (i.e. SQL-like syntax) and simplified workflows (i.e.

abstract the complexity of the underlying systems) make it easier for beginner developers to get started and quickly achieve basic functionality. However, as the complexity of queries grows, also the learning curve associated with these platforms increases. Thus, the learning path for each platform evolves based on the complexity of the tasks that need to be accomplished.

Java Clients comparison

The third aspect was the comparison of the Java clients of the systems under test. Even though the Java client is not the only way to interact with the systems, and much more effort was put into the interaction through the SQL dialects, it is still an important aspect to consider.
Initially, the idea was to test the Scala capabilities of these systems. However, only Flink supports this language and it is deprecated from [187]. So, we decided to switch to the Java language.
We chose Java as the main language for testing the clients for mainly two reasons. First of all, Java is one of the most used languages in the world. Secondly, Java is the only language supported by all three systems. While Flink supports also Python and Materialize supports a lot more languages (essentially if a PostgreSQL client/tool exists, can communicate with Materialize [188]), ksqlDB only supports Java.
The tests wanted to understand the quality and the depth of the Java clients. Initially one of the goals was also to test the support of the user-defined functions (UDFs), but eventually, this was not possible. This was due to a time constraint and the fact that the UDFs are not supported by all the systems. Flink and ksqlDB support UDFs, while Materialize does not [189] (and it seems that it is not in the roadmap).
Flink not only supports connectivity with Java, but it also provides all the necessary tools to develop a streaming application through it. With Java in Flink, is possible to manage the lifecycle of the application, from the deployment to the monitoring. The Java client is well documented and the community is very active.

It is important to notice that Flink supported from the beginning Java language and it is obvious that the Java client support is wider.
ksqlDB offers a lightweight Java client, that wraps the REST API and offers an alternative to using the REST API. The client enables many operations, such as creating streams and tables, running push and pull queries or even listing the available topics in the Kafka cluster. A `Client` class accepts a `ClientOptions` object, which is used to configure the client. From this class, which is the main entry point of the client, you can use the API asynchronously or synchronously. To consume records asynchronously, you can implement the *Reactive Stream* [190] subscriber interface. The interface `Subscriber<T>` is a simple interface that models what to do with the data in different situations (i.e. when the data is received, when the data is completed, when an error occurs). Even though the client is not well documented and still unripe under many aspects, it is a good starting point, and combined with the REST API and the CLI, it is a powerful tool. Below a simple example of how to use the ksqlDB Java client is shown in the listing 5.5.

```
// connect to the ksqlDB server
ClientOptions options = ClientOptions.create()
    .setHost(this.ksqlDBHost)
    .setPort(this.ksqlDBPort);
Client client = Client.create(options);
.
.
// create all the streams/tables and wait for the result
client.executeStatement(createBidType).join();
.
.
// asynchronously consume the records
CompletableFuture<StreamedQueryResult> streamQuery =
client.streamQuery("SELECT * FROM bids LIMIT 10;");
// create a subscriber
NexmarkSubscriber subscriber = new NexmarkSubscriber();
streamQuery
    .thenAccept(streamedQueryResult -> {
        streamedQueryResult.subscribe(subscriber);
    })
    .exceptionally(e -> {
        throw new RuntimeException(e);
    });
```

Listing 5.5: Essential example of how to use the ksqlDB Java client.

As said before ksqlDB, supports also the crafting of UDFs in Java; however,

this functionality was not tested.
Materialize offers a Java client that is a wrapper around the PostgreSQL JDBC driver. Materialize client leverage the *PostgreSQL JDBC Driver* [191] allowing Java programs to connect to a PostgreSQL database using standard, database-independent Java code. Materialize does not extend the PostgreSQL JDBC driver, and it offers documentation on how to use it.
Overall, only the Flink-Java functionalities are mature and complete, offering a wide range of possibilities. Notwithstanding, the ksqlDB Java client is a good choice for those who want to use ksqlDB in a Java environment, it needs to be improved and expanded. Materialize, instead, does not offer a specific Java API but only encourages to use of the appropriate driver. Again, we want to emphasize that ksqlDB and Materialize were born as SQL-first systems, so the Java client is not the main focus of the systems, but a qualitative analysis of the Java clients was necessary to enhance the picture of the systems under test.

Chapter 6

Conclusions

The work presented in this thesis has been focused on a comparison between three different stream processing systems, Apache Flink, ksqlDB and Materialize, in the context of materialized views crafted through SQL queries. The comparison has used a dual approach: a quantitative comparison, based on the performance of the systems, and a qualitative comparison, based on the features of the systems. The quantitative comparison has been carried out by means of a benchmark, which has been developed as part of this thesis. The benchmark uses a baseline benchmark as a starting point, Nexmark, and has been extended to support the three systems. In order to provide extensibility, the benchmark provides a modular generator that can be used to generate different workloads. However, the benchmark is still in its early stages and there is still a lot of work to be done to make it a complete tool. The quantitative trial has shown that Materialize is the fastest system in terms of throughput and duration of the queries, followed by Flink and ksqlDB. The qualitative test showed that, taking into account that is also the oldest system, Flink is the most complete and flexible system. Although Flink requires more effort to be set up, it can lead to full integration with other systems and can be used in a wide range of use cases. Moreover, the SQL support of Flink is more complete than the one of ksqlDB and Materialize. ksqlDB offers a SQL interface for defining stream processing in SQL instead of coding with Java or Scala (Java and SQL cannot be mixed in the same application, only defining

UDF functions in Java). It is a perfect choice when a stream processing based on Kafka need to be done, it is necessary to set up a system quickly and the work does not require complex processing. Materialize is one of the new entries in the stream processing domain, powered by new concepts. It is very fast and easy to set up, especially employing the cloud version. The integration with other systems and some other shortcomings is a bit immature but it is a very promising system and it is worth keeping an eye on it. Yet, the results are not enough to draw a conclusion about the best system, as the systems have different features and are designed for different use cases. The choice of the system depends on the specific use case and the requirements of the user, even if the performance is an important factor to consider.

6.1 Future work

Regardless of the results obtained, there is still a lot of work to be done to make the benchmark a complete tool. The following is a list of possible future work to be done on the project:

- The generator supports different workloads, and could be very useful to test the systems in different scenarios. For example, further research could be conducted to test the systems with peak and burst data.
- The benchmark could include more tools specific to the systems that can be easily integrated with other systems. For example, the benchmark could include a validator to check the correctness of the results, and a metrics exporter to export the metrics of the systems to a monitoring system.
- The project does not currently utilize UDF functions, but considering their importance in stream processing systems, extending the benchmark to include UDFs could be valuable. To achieve this, define clear objectives, choose a variety of UDFs for testing, and assess their performance in systems that support them.

- The benchmark could be extended to support more metrics, such as the latency of the system. Still, this is a tough task, as explained in the previous sections.
- Currently, the generator produces each time a different set of data, and the results of the systems are not reproducible. The benchmark could be extended to support a seed as input to the generator so that the results of the systems are reproducible.
- As we noticed during the development of the benchmark, the fluidity of a development process is very important. Indeed, during local testing of ksqlDB, we noticed that the system could include a Testcontainer to speed up the development process. Despite this is not a feature of the benchmark, it is a very important feature to be included in ksqlDB as well as to improve the Java client.
- The comparison between the Kafka producer generator and the Flink generator is stranded due to a connection issue in the Kafka producer generator. A deeper analysis should be done between these two solutions to better understand the performance of these two generators. This analysis should be based on speed, throughput, profiling, and resource utilization. This may include a deepening of the internal metrics of Flink [197].

Bibliography

[1] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience benchmarking. *Resilience assessment and evaluation of computing systems*, pages 283–301, 2012.

[2] Jóakim v. Kistowski, Jeremy A Arnold, Karl Huppler, Klaus-Dieter Lange, John L Henning, and Paul Cao. How to build a benchmark. In *Proceedings of the 6th ACM/SPEC international conference on performance engineering*, pages 333–336, 2015.

[3] Henrique C. M. Andrade, Buğra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[4] Yuke Yang, Lukasz Golab, and M. Tamer Ozsu. Viewdf: Declarative incremental view maintenance for streaming data. *Information Systems*, 71:55–67, 2017.

[5] Inc IBM. Ibm streams home page, 2023. https://www.ibm.com/products/streaming-analytics.

[6] Inc Alphabet. Google cloud dataflow home page, 2023. https://cloud.google.com/dataflow?hl=com.

[7] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, pages 1–35, 2023.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] 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. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[10] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, 2000.

[11] Daniel J Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12:120–139, 2003.

[12] Inc IBM. IBM System S White Paper. `https://public.dhe.ibm.com/software/data/sw-library/ii/whitepaper/SystemS_2008-1001.pdf`. [Accessed 15-12-2023].

[13] Overview of Oracle CEP — docs.oracle.com. `https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/overview.htm#CEPGS106`. [Accessed 15-12-2023].

[14] Apache Storm — storm.apache.org. `https://storm.apache.org/`. [Accessed 15-12-2023].

[15] Apache Flink® — Stateful Computations over Data Streams — flink.apache.org. `https://flink.apache.org/`. [Accessed 15-12-2023].

[16] Introducing Kafka Streams: Stream Processing Made Simple — Confluent — confluent.io. `https://www.confluent.io/blog/`

introducing-kafka-streams-stream-processing-made-simple/. [Accessed 15-12-2023].

[17] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and {Fault-Tolerant} model for stream processing on large clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, 2012.

[18] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[19] About — beam.apache.org. https://beam.apache.org/about/. [Accessed 15-12-2023].

[20] Apache Flink 1.2 Documentation: Dataflow Programming Model — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.2/concepts/programming-model.html. [Accessed 10-01-2024].

[21] Meet Materialize — materialize.com. https://materialize.com/about/. [Accessed 15-12-2023].

[22] Portals Project. Welcome to Portals — Portals Project Committee — portals-project.org. https://www.portals-project.org/. [Accessed 15-12-2023].

[23] About Us — Arroyo — arroyo.dev. https://www.arroyo.dev/about. [Accessed 15-12-2023].

[24] Rising Wave Labs. Risiwing Wave About Us. https://risingwave.com/about-us/. [Accessed 15-12-2023].

[25] Jim Verheijde, Vassilios Karakoidas, Marios Fragkoulis, and Asterios Katsifodimos. S-query: Opening the black box of internal stream processor state. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1314–1327. IEEE, 2022.

[26] Why Hazelcast — hazelcast.com. https://hazelcast.com/why-hazelcast/. [Accessed 15-12-2023].

[27] ksqlDB and Kafka Streams — Confluent Documentation — docs.confluent.io. https://docs.confluent.io/platform/current/streams-ksql.html#:~:text=Kafka%20Streams%20is%20a%20client,processing%20tasks%20using%20SQL%20statements. [Accessed 16-12-2023].

[28] TU Berlin University. Stratosphere Home Page. https://stratosphere.eu/. [Accessed 16-12-2023].

[29] GitHub - apache/flink: Apache Flink — github.com. https://github.com/apache/flink. [Accessed 16-12-2023].

[30] On Apache Flink. Interview with Volker Markl. — ODBMS Industry Watch — odbms.org. https://www.odbms.org/blog/2015/06/on-apache-flink-interview-with-volker-markl/. [Accessed 16-12-2023].

[31] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[32] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows, 2015.

[33] Overview — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/overview/. [Accessed 16-12-2023].

[34] Stateful Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/stateful-stream-processing/#what-is-state. [Accessed 18-12-2023].

[35] Checkpointing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/datastream/fault-tolerance/checkpointing/#checkpointing. [Accessed 18-12-2023].

[36] Savepoints — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/ops/state/savepoints/#what-is-a-savepoint. [Accessed 18-12-2023].

[37] Checkpoints vs. Savepoints — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/ops/state/checkpoints_vs_savepoints/#overview. [Accessed 18-12-2023].

[38] Stateful Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/stateful-stream-processing/#keyed-state. [Accessed 18-12-2023].

[39] Working with State — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/datastream/fault-tolerance/state/#operator-state. [Accessed 18-12-2023].

[40] A Practical Guide to Broadcast State in Apache Flink — flink.apache.org. https://flink.apache.org/2019/06/26/a-practical-guide-to-broadcast-state-in-apache-flink/. [Accessed 18-12-2023].

[41] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM Transactions on Computer Systems*, pages 63–75, February 1985.

[42] Stateful Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/stateful-stream-processing/#barriers. [Accessed 18-12-2023].

[43] Stateful Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/stateful-stream-processing/#unaligned-checkpointing. [Accessed 18-12-2023].

[44] Timely Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/time/#notions-of-time-event-time-and-processing-time. [Accessed 18-12-2023].

[45] Tyler Akidau. Streaming 101: The world beyond batch — oreilly.com. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101. [Accessed 18-12-2023].

[46] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:1–1, 10 2019.

[47] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[48] Tyler Akidau. Streaming 102: The world beyond batch — oreilly.com. https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/. [Accessed 19-12-2023].

[49] Timely Stream Processing — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/time/#event-time-and-watermarks. [Accessed 19-12-2023].

[50] Apache Flink - A Complete Introduction — developer.confluent.io. https://developer.confluent.io/courses/apache-flink/intro/?utm_source=youtube&utm_medium=video&utm_campaign=tm.devx_ch.cd-apache-flink-101_content.apache-flink. [Accessed 19-12-2023].

[51] How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet — nathanmarz.com. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html. [Accessed 19-12-2023].

[52] Jay Kreps. questioningTheLambdaArchitecture. https://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html. [Accessed 19-12-2023].

[53] Flink Architecture — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/flink-architecture/#anatomy-of-a-flink-cluster. [Accessed 19-12-2023].

[54] Glossary — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/glossary/#logical-graph. [Accessed 19-12-2023].

[55] Flink Architecture — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/flink-architecture/#task-slots-and-resources. [Accessed 19-12-2023].

[56] Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala — akka.io. https://akka.io/. [Accessed 19-12-2023].

[57] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973.

[58] Akka and Actors - Apache Flink - Apache Software Foundation — cwiki.apache.org. https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors. [Accessed 19-12-2023].

[59] Fault Tolerance — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/learn-flink/fault_tolerance/#exactly-once-guarantees. [Accessed 19-12-2023].

[60] Fault Tolerance Guarantees — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/connectors/datastream/guarantees/. [Accessed 19-12-2023].

[61] Streaming Analytics — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/learn-flink/streaming_analytics/#introduction. [Accessed 19-12-2023].

[62] GitHub - faust-streaming/faust: Python Stream Processing. A Faust fork — github.com. https://github.com/faust-streaming/faust. [Accessed 21-12-2023].

[63] Confluent. KSQL versus ksqlDB - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/operate-and-deploy/ksql-vs-ksqldb/#:~:text=For%20the%20purposes%20of%20this,over%20an%20existing%20KSQL%20deployment. [Accessed 21-12-2023].

[64] Confluent — Apache Kafka Reinvented for the Cloud — confluent.io. https://www.confluent.io/. [Accessed 21-12-2023].

[65] ksqlDB: The database purpose-built for stream processing applications. — ksqldb.io. https://ksqldb.io/overview.html. [Accessed 21-12-2023].

[66] Apache Kafka — kafka.apache.org. https://kafka.apache.org/. [Accessed 21-12-2023].

[67] Matthias J Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, pages 1–10, 2018.

[68] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322, 2005.

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

[70] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*, pages 1–19. Springer, 2003.

[71] Streams and Tables: Two Sides of the Same Coin — Confluent — confluent.io. https://www.confluent.io/blog/streams-tables-two-sides-same-coin/. [Accessed 22-12-2023].

[72] Confluent. ksqlDB Architecture - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/operate-and-deploy/how-it-works/#ksqldb-and-kafka-streams. [Accessed 22-12-2023].

[73] Confluent. Queries - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/concepts/queries/#persistent. [Accessed 22-12-2023].

[74] Confluent. Queries - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/concepts/queries/#push. [Accessed 22-12-2023].

[75] Confluent. Queries - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/concepts/queries/#pull. [Accessed 22-12-2023].

[76] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, 1987.

[77] Confluent. ksqlDB Architecture - ksqlDB Documentation — docs.ksqldb.io. https://docs.ksqldb.io/en/latest/operate-and-deploy/how-it-works/. [Accessed 28-12-2023].

[78] ANTLR — antlr.org. https://www.antlr.org/. [Accessed 28-12-2023].

[79] ksql/ksqldb-parser/src/main/antlr4/io/confluent/ksql/parser/SqlBase.g4 at master · confluentinc/ksql — github.com. https://github.com/confluentinc/ksql/blob/master/ksqldb-parser/src/main/antlr4/io/confluent/ksql/parser/SqlBase.g4. [Accessed 10-01-2024].

[80] Priya Narasimhan, Louise E Moser, and P Michael Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant corba applications. *Comput. Syst. Sci. Eng.*, 17(2):103–114, 2002.

[81] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data*, pages 2602–2613, 2021.

[82] Ali M Alakeel et al. A guide to dynamic load balancing in distributed computer systems. *International journal of computer science and information security*, 10(6):153–160, 2010.

[83] Install Materialize — Materialize Documentation — materialize.com. https://materialize.com/docs/lts/install/. [Accessed 29-12-2023].

[84] Announcing the next generation of Materialize — materialize.com. https://materialize.com/blog/next-generation/. [Accessed 29-12-2023].

[85] Introducing Materialize: the Streaming Data Warehouse — materialize.com. https://materialize.com/blog/introduction/. [Accessed 29-12-2023].

[86] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[87] GitHub - TimelyDataflow/timely-dataflow: A modular implementation of timely dataflow in Rust — github.com. https://github.com/TimelyDataflow/timely-dataflow. [Accessed 29-12-2023].

[88] GitHub - TimelyDataflow/differential-dataflow: An implementation of differential dataflow using timely dataflow on Rust. — github.com. https://github.com/TimelyDataflow/differential-dataflow. [Accessed 29-12-2023].

[89] When to use Timely Dataflow — timelydataflow.github.io. https://timelydataflow.github.io/timely-dataflow/chapter_0/chapter_0_2.html. [Accessed 29-12-2023].

[90] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, aug 1978.

[91] Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. Verified progress tracking for timely dataflow. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193, page 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[92] Architecture overview — Materialize Documentation — materialize.com. https://materialize.com/docs/lts/overview/architecture/. [Accessed 03-01-2024].

[93] Tools and integrations — Materialize Documentation — materialize.com. https://materialize.com/docs/lts/integrations/#client-libraries-and-orms. [Accessed 03-01-2024].

[94] Strict Serializability — jepsen.io. https://jepsen.io/consistency/models/strict-serializable. [Accessed 03-01-2024].

[95] Virtual Time: The Secret to Strong Consistency and Scalable Performance in Materialize — materialize.com. https://materialize.com/blog/virtual-time-consistency-scalability/. [Accessed 03-01-2024].

[96] What's inside Materialize? An architecture overview — materialize.com. https://materialize.com/blog/architecture/#the-big-picture. [Accessed 04-01-2024].

[97] INSERT — Materialize Documentation — materialize.com. https://materialize.com/docs/sql/insert/. [Accessed 05-01-2024].

[98] Arrangements — timelydataflow.github.io. https://timelydataflow.github.io/differential-dataflow/chapter_5/chapter_5.html. [Accessed 05-01-2024].

[99] TPC-H Homepage — tpc.org. https://www.tpc.org/tpch/default5.asp. [Accessed 08-01-2024].

[100] TPC-C Homepage — tpc.org. https://www.tpc.org/tpcc/. [Accessed 08-01-2024].

[101] Nexmark benchmark suite — beam.apache.org. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. [Accessed 12-01-2024].

[102] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload ch-benchmark. In *Proceedings of*

the Fourth International Workshop on Testing Database Systems, pages 1–6, 2011.

[103] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491, 2004.

[104] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 69–78. IEEE, 2014.

[105] Software Factory HonKit — handbook.agilelab.it. `https://handbook.agilelab.it/SoftwareDevelopment.html`. [Accessed 15-01-2024].

[106] TPC Who We Are — tpc.org. `https://www.tpc.org/information/who/whoweare5.asp`. [Accessed 12-01-2024].

[107] Apache Beam — beam.apache.org. `https://beam.apache.org/`. [Accessed 12-01-2024].

[108] GitHub - nexmark/nexmark: Benchmarks for queries over continuous data streams. — github.com. `https://github.com/nexmark/nexmark`. [Accessed 08-01-2024].

[109] Benchmarking Streaming Computation Engines at Yahoo! — yahooeng.tumblr.com. `https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at`. [Accessed 08-01-2024].

[110] Curious Case of the Broken Benchmark: Revisiting Flink vs Databricks — ververica.com. `https://www.ververica.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime#hs_cos_wrapper_post_body`. [Accessed 09-01-2024].

[111] Giselle Van Dongen and Dirk Van den Poel. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.

[112] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio FR Geyer, and Luiz Gustavo L Fernandes. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[113] Dakshi Agrawal, Ali Butt, Kshitij Doshi, Josep-L Larriba-Pey, Min Li, Frederick R Reiss, Francois Raab, Berni Schiefer, Toyotaro Suzumura, and Yinglong Xia. Sparkbench–a spark performance testing suite. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31–September 4, 2015. Revised Selected Papers 7*, pages 26–44. Springer, 2016.

[114] Chunhui Li and Robert Berry. Cepben: a benchmark for complex event processing systems. In *Performance Characterization and Benchmarking: 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers 5*, pages 125–142. Springer, 2014.

[115] GitHub - DanySK/boilerplate: Java boilerplate code that should help writing more compact programs — github.com. `https://github.com/DanySK/boilerplate`. [Accessed 15-01-2024].

[116] Checkstyle 10.12.7 — checkstyle.sourceforge.io. `https://checkstyle.sourceforge.io/index.html`. [Accessed 15-01-2024].

[117] Alessio Pagliari. What is NAMB? — apgl.xyz. `https://apgl.xyz/namb/`. [Accessed 09-01-2024].

[118] SpotBugs — spotbugs.github.io. `https://spotbugs.github.io/`. [Accessed 15-01-2024].

[119] Alessio Pagliari, Fabrice Huet, and Guillaume Urvoy-Keller. Namb: A quick and flexible stream processing application prototype generator. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 61–70. IEEE, 2020.

[120] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[121] Apache Spark - Unified Engine for large-scale data analytics — spark.apache.org. `https://spark.apache.org/`. [Accessed 14-02-2024].

[122] Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. Espbench: the enterprise stream processing benchmark. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 201–212, 2021.

[123] Adriano Marques Garcia, Dalvan Griebler, Claudio Schepke, and Luiz Gustavo Fernandes. Spbench: a framework for creating benchmarks of stream processing applications. *Computing*, 105(5):1077–1099, 2023.

[124] Esmail Asyabi, Yuanli Wang, John Liagouris, Vasiliki Kalavri, and Azer Bestavros. A new benchmark harness for systematic and robust evaluation of streaming state stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 559–574, 2022.

[125] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 974–985. Elsevier, 2002.

[126] NEXMark Benchmark — web.archive.org. `https://web.archive.org/web/20100620010601/http://datalab.cs.pdx.edu/niagaraST/NEXMark/`. [Accessed 11-01-2024].

[127] JUnit 5 — junit.org. https://junit.org/junit5/. [Accessed 15-01-2024].

[128] Flink Ecosystem Website — flink-packages.org. https://www.flink-packages.org/packages/nexmark-benchmark. [Accessed 09-01-2024].

[129] Testcontainers for Java — java.testcontainers.org. https://java.testcontainers.org/. [Accessed 15-01-2024].

[130] Event-driven Applications — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/event_driven/#side-outputs. [Accessed 10-01-2024].

[131] SLF4J — slf4j.org. https://www.slf4j.org/. [Accessed 15-01-2024].

[132] Grafana: The open observability platform — Grafana Labs — grafana.com. https://grafana.com/. [Accessed 16-01-2024].

[133] KAZIMIERZ Balos, Dominik Radziszowski, PAWEŁ Rzepa, KRZYSZTOF Zieliński, and SŁAWOMIR Zieliński. Monitoring grid resources: jmx in action. *TASK Quarterly. Scientific Bulletin of Academic Computer Centre in Gdansk*, 8(4):487–501, 2004.

[134] GitLab - The DevSecOps Platform — about.gitlab.com. https://about.gitlab.com/. [Accessed 16-01-2024].

[135] Conventional Commits — conventionalcommits.org. https://www.conventionalcommits.org/en/v1.0.0/. [Accessed 16-01-2024].

[136] Atlassian. Gitflow Workflow — Atlassian Git Tutorial — atlassian.com. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Accessed 16-01-2024].

[137] Atlassian. Git Feature Branch Workflow — Atlassian Git Tutorial — atlassian.com. https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow. [Accessed 16-01-2024].

[138] Diomidis Spinellis. Git. *IEEE software*, 29(3):100–101, 2012.

[139] Prometheus. Prometheus - Monitoring system & time series database — prometheus.io. https://prometheus.io/. [Accessed 16-01-2024].

[140] Datadog. Cloud Monitoring as a Service — Datadog — datadoghq.com. https://www.datadoghq.com/. [Accessed 18-01-2024].

[141] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[142] Gradle Build Tool — gradle.org. https://gradle.org/. [Accessed 16-01-2024].

[143] Docker Compose overview — docs.docker.com. https://docs.docker.com/compose/. [Accessed 17-01-2024].

[144] Home — docker.com. https://www.docker.com/. [Accessed 16-01-2024].

[145] Use CI/CD to build your application — GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/topics/build_your_application.html. [Accessed 16-01-2024].

[146] CI/CD components — GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/components/index.html. [Accessed 16-01-2024].

[147] Runner SaaS — GitLab — docs.gitlab.com. https://docs.gitlab.com/ee/ci/runners/index.html. [Accessed 16-01-2024].

[148] Gradle Wrapper Reference — docs.gradle.org. https://docs.gradle.org/current/userguide/gradle_wrapper.html. [Accessed 16-01-2024].

[149] RocksDB — A persistent key-value store — rocksdb.org. https://rocksdb.org/. [Accessed 18-01-2024].

[150] Flink Architecture — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#task-slots-and-resources. [Accessed 18-01-2024].

[151] RocksDB Compaction — github.com. https://github.com/facebook/rocksdb/wiki/Compaction. [Accessed 18-01-2024].

[152] Why not RocksDB for streaming storage? — materialize.com. https://materialize.com/blog/why-not-rocksdb/. [Accessed 18-01-2024].

[153] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *arXiv preprint arXiv:1812.02639*, 2018.

[154] Alex Woodie. The New Economics of the Separation of Compute and Storage — datanami.com. https://www.datanami.com/2018/04/20/the-new-economics-of-the-separation-of-compute-and-storage/. [Accessed 18-01-2024].

[155] Domas Mituzas. How InnoDB lost its advantage. https://dom.as/2015/04/09/how-innodb-lost-its-advantage/. [Accessed 19-01-2024].

[156] Using RocksDB State Backend in Apache Flink: When and How — flink.apache.org. https://flink.apache.org/2021/01/18/using-rocksdb-state-backend-in-apache-flink-when-and-how/. [Accessed 19-01-2024].

[157] Meta Engineering. Under the Hood: Building and open-sourcing RocksDB. https://www.facebook.com/notes/10158791582997200/. [Accessed 19-01-2024].

[158] Stateful Operations in ksqlDB: Aggregations and Materialized Views — developer.confluent.io. https://developer.confluent.io/courses/inside-ksqldb/stateful-operations/. [Accessed 19-01-2024].

[159] Compacted topics — docs.aiven.io. https://docs.aiven.io/docs/products/kafka/concepts/log-compaction. [Accessed 19-01-2024].

[160] can i use flink rocksDB state backend with local file system? — stackoverflow.com. https://stackoverflow.com/questions/58614739/

can-i-use-flink-rocksdb-state-backend-with-local-file-system. [Accessed 19-01-2024].

[161] IOException: Size of the state is larger than the maximum permitted memory-backed state — stackoverflow.com. https://stackoverflow.com/questions/50149005/ioexception-size-of-the-state-is-larger-than-the-maximum-permitted-memory-backe. [Accessed 19-01-2024].

[162] Servizi di cloud computing: Amazon Web Services (AWS) — aws.amazon.com. https://aws.amazon.com/it/. [Accessed 22-01-2024].

[163] Calcolo sicuro e ridimensionabile nel cloud - Amazon EC2 - Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/ec2/. [Accessed 22-01-2024].

[164] Calcolo serverless – AWS Lambda – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/lambda/. [Accessed 22-01-2024].

[165] Amazon WorkMail – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/workmail/. [Accessed 22-01-2024].

[166] Amazon Elastic Block Store (Amazon EBS) - Amazon Elastic Compute Cloud — docs.aws.amazon.com. https://docs.aws.amazon.com/it_it/AWSEC2/latest/UserGuide/AmazonEBS.html. [Accessed 22-01-2024].

[167] Archiviazione di oggetti cloud – Amazon S3 – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/s3/. [Accessed 23-01-2024].

[168] Virtual Private Cloud isolato logicamente - Amazon VPC - Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/vpc/. [Accessed 22-01-2024].

[169] IAM: Gestione dei ruoli — aws.amazon.com. https://aws.amazon.com/it/iam/features/manage-roles/. [Accessed 23-01-2024].

[170] Apache Kafka Completamente gestito – Amazon MSK – Amazon Web Services — aws.amazon.com. https://aws.amazon.com/it/msk/. [Accessed 22-01-2024].

[171] Policies and permissions in IAM - AWS Identity and Access Management — docs.aws.amazon.com. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html. [Accessed 23-01-2024].

[172] Sign-in credentials authentication with AWS Secrets Manager - Amazon Managed Streaming for Apache Kafka — docs.aws.amazon.com. https://docs.aws.amazon.com/msk/latest/developerguide/msk-password.html. [Accessed 23-01-2024].

[173] Abhijit Menon-Sen, Alexey Melnikov, Nicolás Williams, and Chris Newman. Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms. RFC 5802, July 2010.

[174] HDFS Architecture Guide — hadoop.apache.org. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction. [Accessed 23-01-2024].

[175] GitHub - prometheus/jmx_exporter: A process for exposing JMX Beans via HTTP for Prometheus consumption — github.com. https://github.com/prometheus/jmx_exporter. [Accessed 23-01-2024].

[176] Esper — espertech.com. https://www.espertech.com/esper/. [Accessed 24-01-2024].

[177] Pricing Plans & Options — Materialize — materialize.com. https://materialize.com/pricing/. [Accessed 26-01-2024].

[178] ProducerRecord (kafka 2.4.0 API) — kafka.apache.org. https://kafka.apache.org/24/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html. [Accessed 30-01-2024].

[179] [FLINK-19059] Support to consume retractions for OVER WINDOW operator - ASF JIRA — issues.apache.org. https://issues.apache.org/jira/browse/FLINK-19059. [Accessed 31-01-2024].

[180] Support persistent queries on windowed tables · Issue #6513 · confluentinc/ksql — github.com. https://github.com/confluentinc/ksql/issues/6513. [Accessed 31-01-2024].

[181] ActiveMQ — activemq.apache.org. https://activemq.apache.org/. [Accessed 31-01-2024].

[182] RabbitMQ: easy to use, flexible messaging and streaming; RabbitMQ — rabbitmq.com. https://rabbitmq.com/. [Accessed 31-01-2024].

[183] Welcome to Apache Flume; Apache Flume — flume.apache.org. https://flume.apache.org/. [Accessed 31-01-2024].

[184] Checkpointing under backpressure — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/checkpointing_under_backpressure/#unaligned-checkpoints. [Accessed 06-02-2024].

[185] Redpanda. Redpanda — The streaming data platform for developers — redpanda.com. https://redpanda.com/. [Accessed 08-02-2024].

[186] Consistency Guarantees in Data Streaming — Materialize — materialize.com. https://materialize.com/blog/consistency/. [Accessed 08-02-2024].

[187] FLIP-265 Deprecate and remove Scala API support - Apache Flink - Apache Software Foundation — cwiki.apache.org. https://cwiki.apache.org/

confluence/display/FLINK/FLIP-265+Deprecate+and+remove+Scala+API+support. [Accessed 08-02-2024].

[188] Java cheatsheet — Materialize Documentation — materialize.com. https://materialize.com/docs/integrations/java-jdbc/. [Accessed 08-02-2024].

[189] Support User-defined Functions · Issue #3688 · MaterializeInc/materialize — github.com. https://github.com/MaterializeInc/materialize/issues/3688. [Accessed 08-02-2024].

[190] reactive-streams.org. https://www.reactive-streams.org/. [Accessed 08-02-2024].

[191] Home — pgJDBC — jdbc.postgresql.org. https://jdbc.postgresql.org/. [Accessed 08-02-2024].

[192] Can't run ksql metrics from docker container · Issue #2716 · confluentinc/ksql — github.com. https://github.com/confluentinc/ksql/issues/2716. [Accessed 09-02-2024].

[193] Brad Kelechava. The SQL Standard - ISO/IEC 9075:2023 (ANSI X3.135) - ANSI Blog — blog.ansi.org. https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/#gref. [Accessed 09-02-2024].

[194] GitHub - AngeloParrinello/streaming-materialized-views-benchmarking: Streaming materialized views / streaming event processing / stream processing systems benchmarking, with a focus on SQL-based approaches. Thesis work by Angelo Parrinello. — github.com. https://github.com/AngeloParrinello/streaming-materialized-views-benchmarking/tree/main. [Accessed 12-02-2024].

[195] Homepage – bytewax — bytewax.io. https://bytewax.io/. [Accessed 12-02-2024].

[196] One tool to develop and run event streaming apps — Quix — quix.io. https://quix.io/. [Accessed 12-02-2024].

[197] Metrics — nightlies.apache.org. https://nightlies.apache.org/flink/flink-docs-master/docs/ops/metrics/. [Accessed 13-02-2024].