LMU

Universität Augsburg University

TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a D-86135 Augsburg

Analytics on Fast Data Using Modern Stream Processing Systems

Jan Böttcher

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Analytics on Fast Data Using Modern Stream Processing Systems

Matrikelnummer:	1356131
Beginn der Arbeit:	26. Februar 2016
Abgabe der Arbeit:	28. April 2016
Erstgutachter:	Prof. Alfons Kemper, Ph.D.
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	Andreas Kipf

Software Engineering
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 28. April 2016

ABSTRACT

The increasing number of connected devices such as cars, mobile phones, or household appliances produces a mass of events. Streaming applications try to extract as much valuable information out of these events as possible. In many cases, the information has to be extracted in real-time to be valuable. This, and the high data rates make strong demands on the systems.

Various streaming engines were built in the past years to address the requirements of the different streaming applications. They distinguish in many aspects, e. g., some use tuple-at-a-time and others micro-batching as computational model. There exist also different approaches to ensure fault-tolerance and message processing. This work analyzes some representative systems and evaluates their capabilities and performance. In particular, it should determine whether the additional programming effort required for using dedicated streaming engines pays off compared to almost "ready-to-use" main-memory database systems.

Therefore, we designed and conducted multiple benchmarks that cover typical streaming tasks. For instance, we evaluated the performance of serialization, message throughput, and aggregations on streams. Finally, we evaluated the analytic capabilities of a streaming engine by comparing it to state-of-the-art main-memory database systems. We used a mixed streaming workload that allows evaluating both the OLTP and the OLAP capabilities of a system. By varying parameters and measuring parts of this workload isolated, we could cover multiple aspects of streaming applications. Additionally, we used it to measure the scalability of the systems within a single node in term of threads.

The streaming engines scaled gracefully for both OLAP and OLTP, whereas database systems were limited in their OLTP scalability as they provide strong consistency. Thereby, streaming engines enable superior write throughput that facilitates applications with higher data rates. On the contrary, they lack OLAP capabilities, as they often do not support a queryable state or declarative APIs like SQL. Whereas, databases enable ad-hoc SQL queries using powerful query processing optimizations such as code generation and vectorization.

Both systems have their own strengths and benefits over the other category of systems. However, there are many opportunities to close the gap between them. For instance, one could lower the durability guarantees of databases according to the demands of the streaming application to achieve a higher throughput. Further, all systems should support a flexible API like SQL that supports both ad-hoc queries and streaming semantics, e. g., windowing. Until then, it depends on the use case which system one should take, but main-memory database systems already proofed great potential for streaming.

ACKNOWLEDGMENTS

This work would not have been possible without the support of many people. First of all, I have to thank my adviser Andreas Kipf. He spent a lot of time and effort on my work and was always eager to discuss new ideas and give advices.

Apart from him, I want to thank the other members of the database chair that showed interest into my work and supported me with inspiring tips and questions. In particular, Varun Pandey made important contributions by measuring the performance of related systems.

Another big thanks goes to Lucas Braun – a member of the Systems Group at ETH Zürich – who motivated the whole work by proposing and implementing a demanding streaming workload. He immediately showed interest in our work and helped us measuring and configuring the systems.

Gyula Fóra – a key contributor of Apache Flink – gave us valuable tips and remarks on our Flink implementation.

Janine Rosenbaum and again Andreas Kipf took an important role in proof-reading my thesis and gave valuable feedback.

Finally, I want to thank my supervisors Prof. Alfons Kemper and Prof. Bernhard Bauer for giving me the opportunity to write this thesis.

Contents

List of Figures

List of Tables

1 Introduction

The demand for streaming applications is driven by the increasing impact of the *Internet of Things*. Currently, there are already more than 6 billion connected devices that constantly produce events [8]. To extract valuable information out of the sheer mass of events, one needs powerful streaming applications. Typically, the information have to be extracted and evaluated in real-time and come at high data rates. Applications range from automotive applications such as real-time traffic information, over social media, to financial applications like high frequency trading.

The various streaming applications make high and versatile demands on systems. A system has to cover aspects such as fault-tolerance, high throughput, low latency, and scalability. Large distributed file and batch processing systems like Hadoop [10] are no longer sufficient as their latency is too high. For this reason, many new streaming engines aroused addressing these points, in particular the aspect of real-time processing. Some open source representatives are Flink [1], Storm [5], and Spark Streaming [4].

A key contribution of this work is to provide an overview regarding usability and performance aspects over these state-of-the-art streaming engines. Therefore, we analyze and compare how the individual systems work and perform. We conduct several benchmarks that cover typical streaming tasks such as event ingestion, serialization, and aggregations on streams. Further, we evaluate the performance of a streaming engine in a mixed OLTP and OLAP workload. We compare it to a highly specialized hand optimized solution and modern main memory databases (MMDBs). Neither the databases nor the streaming engines supported the whole workload out-of-the-box and thus had to be extended. For instance, we had to add a queryable state to the streaming engine.

In this work, we want to find the strengths and weaknesses of streaming engines compared to MMDBs. The goal is to identify the sweet spots of these systems and check how they can be transferred into other systems. Our vision is a general purpose streaming system with a convenience similar to traditional databases and a performance competitive with specialized, hand optimized solutions. Until now, many streaming applications are either fully handwritten or with the help of a streaming framework such as Spark Streaming. These specialized solutions aim for good performance, but require a careful design. In particular, for complex streaming applications the design is a time-consuming and error-prone task. The system should take off the task of development from the user by providing a flexible and declarative API similar to SQL.

The remaining thesis is structured as follows. First, we cover important aspects of streaming applications such as window semantics or fault-tolerance in Section 2. In the following section, we introduce different streaming systems and show how they address the different aspects of streaming. In Section 4, we take a look at different (historical) trends in stream processing and compare them to state-of-the-art solutions. Section 5 evaluates the performance of current streaming engines within typical streaming tasks such as simple aggregations or function evaluations. Section 6 introduces a more complex mixed workload and Section 7 shows how one can implement such workloads with a streaming engine. We use this workload to compare the performance of a streaming engine to those of other systems such as main-memory database systems. The results are shown in Section 8. Finally, the insights of the evaluation along with derived conceptual ideas are presented in Section 9.

2 Background

2.1 Semantics

Streaming engines make different guarantees according how messages, i. e., events are processed. There exist different kinds of messaging guarantees.

At-Most-Once In an at-most-once setup, messages do not necessarily be processed. A system can make use of this low guarantee by dropping messages intensionally to reduce the current load, i. e., perform load shedding. This was mainly done in earlier streaming engines like STREAM or Aurora, where computational power was more restricted than today [22, 24].

Many streaming applications cannot accept data loss and hence need a stronger guarantee.

At-Least-Once If data loss is not acceptable, but data duplication is acceptable, at-least-once semantics is a suitable guarantee. An example for such an application system would be a fire alarm system. It is not a problem if a fire alert is delivered multiple times as long as it is received at-least-once.

A common approach to implement at-least-once semantics are acknowledgments. The processing of every message has to be acknowledged in a determined time. If the sender receives no acknowledgment in this time, it re-sends the message. However, it is possible that the acknowledgment got lost on the way or the processing took longer as the timeout. In these cases, the message is processed more than once.

Exactly Once If a message should only be processed once, some special treatments are required [40]. True exactly-once delivery of messages is not possible in distributed settings, since messages or acknowledgments can always get lost: one can never be sure if the message did not arrive or only the acknowledgment did not come back.

For this reason, exactly-once processing has to deal with duplicated messages. In some cases, the processing of the message itself is an idempotent operation, e. g., set the state to the absolute value x. If the messages arrive at a fixed order, duplicate messages can be re-processed without changing the semantics.

However, this is often not possible. A more general approach is to introduce with unique message identifiers. Duplicate messages can thus be identified and dropped.

Another challenge of exactly-once semantics is fault-tolerance. If a system has to maintain this processing guarantees even in case of failures, a durable data source is required. A data source is considered durable if it can replay any message or set of messages given the necessary selection criteria. In case of a failure, an exactly-once system has to replay and reprocess the messages in a way that ensures the same state as if there was no failure. This is usually done by storing the offset of the latest message that is part of the backup checkpoint.

There exist different implementations for these guarantees.

Transactional Updates Transactions are well-known from databases. Google Cloud Dataflow [9] uses them to guarantee exactly-once processing in streaming applications. The processing of every record and its updates to the state are logged atomically, from where it can be repeated in case of failures.

Distributed Snapshots Flink [1] uses its distributed snapshotting algorithms to guarantee exactly-once processing [35]. During a snapshot, it determines the current state of the streaming computation including in-flight records, and persist it to durable storage. This allows the system to keep track which records have already been processed, i. e., are already part of the latest snapshot state, and which have to be replayed in case of failures.

Acknowledgements Working with acknowledgments, the source of the streaming pipeline backups a tuple until it receives an acknowledgment that it is fully processed and thus can safely be discarded. Storm [5] keeps track of the acknowledgments using an optimized mechanism that only requires few bytes storage per tuple. In case of a failure, the record is replayed. Note that a failure is either a processing failure or just a lost acknowledgment, thus replaying only guarantees at-least-once semantics.

Storm-Trident can guarantee exactly-once semantics by introducing micro-batches. Each micro-batch has a unique batch-id that allows the system to identify and avoid duplicate message processing. However, if the processing of only one tuple fails, the entire batch has to be replayed.

2.2 Windows Semantics

In streaming applications, particularly in the processing of real-time events, one want to perform set-based computations, e. g., aggregations. The challenge is that data is continuous stream and not fixed set.

One can overcome this problem by using windows that can be seen as finite sets of elements on an unbounded stream. These sets can be assigned according to different categories, such as time and element counts. It is also possible to combine counts and time, or assign elements to windows using a custom logic.

2.2.1 Time in Windows

Usually windows are based on time intervals. In these cases, one has to distinguish different types of window-times.

Processing time As the name suggests, processing time is the time when an event is processed by the system. Thus, processing time depends on the local wall clock time of the machine. This works fine for local environments. However, in distributed and asynchronous environments processing time does not provide determinism.

Event time Event time is a time that is externally defined in the events themselves. Typically, events come with timestamps that can be extracted and used by the system. Event time allows the system to handle out-of-order events properly, i. e., an event with a lower timestamp can arrive after an event with a higher timestamp, without distorting the correctness of the result.

Another benefit of event-time is, that the time of processing an event does not change the semantics of the system. This enables the system to process historical data at maximum speed, while it reads fresh continuous data. In particular in case of failures this is essential, as events can be replayed without leading to inconsistent states.

Event time processing provides predictable results, but incurs more latency, as out-of-order events need to be buffered. Another problem of out-of-order-arrival is, that the system does not know whether the last element of a window has already arrived and the window can be evaluated. In practice, one has to ingest special timestamp-events, so called *watermarks*, that tells the system that no event with a lower timestamp will arrive by now.

Ingestion time Ingestion time is the time assigned to an event when it enters the streaming system. After the ingestion time has been assigned, the system can continue to processing like with event time semantics. Thus, ingestion time can be seen as hybrid of processing and event time that strikes a balance between both approaches. Ingestion time is more predictable than processing time, and gives lower latencies than event time as the latency does not depend on external systems.

2.2.2 Basic Window Policies

Every window has its own trigger policy that states when its content is evaluated. Additionally, a window has an evicting strategy that defines how if or how many elements are kept inside the window after it was triggered.

The most basic window policy is represented by *tumbling windows*. A tumbling window triggers whenever it is full, i. e., the time is over, or the count of elements is reached. All elements are evicted after its evaluation. This leads to non-overlapping windows. Examples for tumbling windows are time windows, such as the current day or week, or count windows, such as the last 1000 elements. Figure 2.1 shows a tumbling count window, that calculates the sum of the last four elements. The window is evaluated whenever the window is full, i. e., four new elements have arrived.

Tumbling windows are not overlapping, every element belongs to exactly one window. Thus, tumbling windows can often be implemented without maintaining previous elements. For instance, for a sum-aggregation, it is enough to maintain a rolling aggregate of the sum. As soon as an event is not required any more, e. g., it is part of the rolling aggregate, it can be discarded.

Table 2.1 shows how common aggregates can be calculated incrementally without keeping previous elements. Simple aggregations such as count, min, max, or sum only require to maintain the current value of the aggregate. Other aggregates require additional helper variables out of which the final aggregates can be calculated. For average, two rolling aggregates have to be maintained to calculate the final average: the element count and a rolling sum. For standard deviation and variance one has to additionally keep the squared sum. Obviously, one could share helper variables throughout different aggregates. For instance, if one has already a count and a sum aggregate on the stream, one could calculate the average without any extra variables.

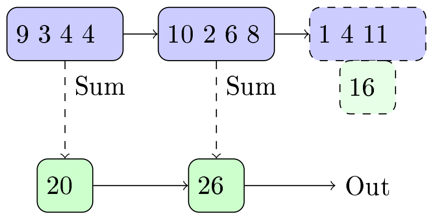


Figure 2.1: Tumbling count window of size four

Additionally, to a window size, sliding windows define a sliding range. For instance, window size of 24 hours and sliding range of one hour. In contrast to a tumbling window of the same size, this window contains the last 24 hours instead of the current day. The sliding range has to be lower or

	Incrementation	**Evaluation**
AVG	$sum{+}{=}x;$ $n{+}{+};$	sum/n
COUNT	$n{+}{+};$	n
MAX	$m{=}\ max(m, x);$	m
MIN	$m{=}\ min(m, x);$	m
STDEV	$S_1\ {+}{=}\ x;$ $S_1\ {+}{=}\ x^2;$ $n{+}{+};$	$\mu = S_1/n$ $\sigma = \sqrt{S_2/n - \mu^2}$
VAR	$S_1\ {+}{=}\ x;$ $S_1\ {+}{=}\ x^2;$ $n{+}{+};$	$\mu = S_1/n$ $\sigma^2 = S_2/n - \mu^2$

Table 2.1: Incremental calculation of rolling aggregates for a new element x

equal to the window size. Tumbling windows are a special case of sliding windows, where the sliding range equals the window size.

In sliding windows are overlapping and thus elements can belong to more than one window, i. e., *window size/slide range* windows. Hence, if the sliding range becomes smaller, more elements have to be maintained at once, as they cannot be discarded until the last corresponding window is evaluated.

In the extreme case, the sliding range is one, i. e., the windows are continuously sliding. Here, the number of elements that have to be maintained for one window can equal the size of this window depending on the window aggregation function.

For instance, maximum or minimum aggregations need to maintain all elements, as the old maximum or minimum can slide out every time. As scanning of the whole window contents is inefficient for big window sizes, there exist approaches. One could store the top-N elements in a fast accessible storage and only fall back to element archives when all top-N elements at once have fallen out of window range [33]. Another approach is to maintain histograms of the data stream to estimate the value instead of storing all elements and calculate the exact value [39].

In general, one can always reduce the amount of required space and scanning time by calculating and storing only an aggregated value for all elements within the same window slice. For instance, store the maximum store the maximum value for each of the last 24 hours. Whenever the window is calculated, one only have to determine the overall maximum among the 24 hour slides.

Having an aggregate like sum, one can do additional optimization by maintaining the rolling sum of the whole window. Whenever a window slice falls out of the window, the rolling window sum can be adjusted by subtracting the value of the outdated window slice.

2.2.3 Generic Window Design

Windowing is composed of different components: a window assigner, a trigger policy, an eviction policy, and an evaluation function. For instance, a tumbling-daily-sum-window would trigger the calculation of the sum every day evicting all its elements.

Some applications need more fine-grained control over windowing semantics. This section describes the different elements of windowing and Figure 2.2 shows how they can interact.

Window Assigner The Window-Assigner is a function that assigns elements to windows. An element can be assigned to multiple windows at once. Hence, multiple windows can exist at the same time.

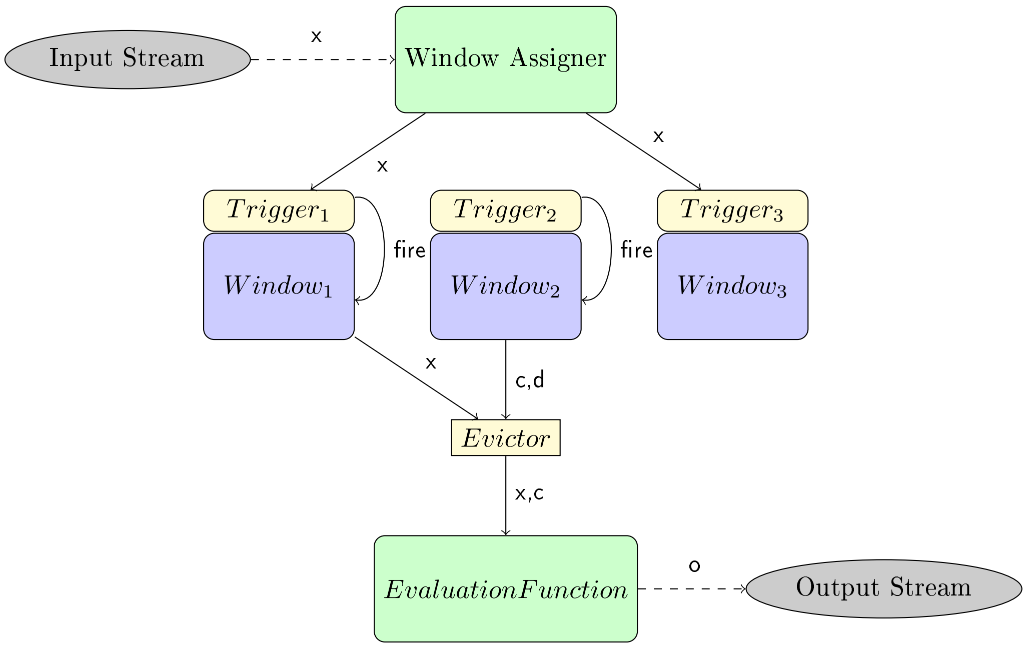


Figure 2.2: Dataflow in an exemplary window

Window A Window stores the corresponding elements and usually provides additional meta information, such as begin and end time in case of a time-based window, or number of elements in case of a count-based window.

Trigger Each window has a trigger policy that decides when it is evaluated and if its elements are purged. A trigger may be based on a time interval, the element count, or a delta. For example, it could trigger every five minutes or every ten elements. A delta-trigger fires when the change of data has reached a specified threshold.

Whenever a trigger is evaluated, it can either decide to fire (i. e., evaluate), purge (remove the window and discard its content), or fire and then purge the window. As long as a window is not purged, the elements are kept and they can be evaluated again when the trigger fires next.

Evictor An evictor can remove elements from a window after a trigger fires. Thus, it can control which elements are kept inside the window. Evictors can be used to implement sliding windows. If the window size is 1000 and the sliding factor is 100, the evictor would ensure that only 1000 elements are kept inside the window, i. e., remove the first 100 elements that slide out of the window range.

Evaluation Function The evaluation function receives the elements of a window and computes one or more resulting elements for the window. The results are added to the output stream. Possible evaluation functions range from simple aggregations such as *min()*, *max()* or *sum()* to more complex higher order logic functions such as handwritten reduce functions.

2.3 Flow Control

In general, streaming data do not have constant velocities or fixed data-rates. In many applications, there exist peeks of data, e. g., an increased number of tweets after a big event. Streaming systems should be able to handle these peeks gracefully. As long as the system has enough free resources to process the increased rate of events, there is no problem. However, if the event processing rate falls behind the event receiving rate, the system has to deal with a phenomenon called backpressure.

A naive approach to handle backpressure would be to drop data, for example, by taking a random sample of it. However, the loss of data is not acceptable in applications featuring at-least-once or higher message delivery guarantees.

A general solution to prevent data loss, is to buffer the additional data in a durably fashion. This can be done by throttling the whole system up to the data source – assuming that the source is durable. A common example for such a durable source is Apache Kafka [2].

Depending on the stream processing model, i. e., tuple-at-a-time or micro-batching, the concrete implementations vary. Hence, we will take a close look on Flink and Spark, representing the tuple-at-a-time and the micro-batching approach.

2.3.1 Tuple-at-a-time: Flink

Flink can handle backpressure simply by the design of its data shipping [37]. In Flink, each consecutive pair of operator is connected by distributed blocking queues.

The blocking queues are implemented via managed buffer pools of fixed-size. An operator takes a buffer from the pool, fills it up with outgoing data and sends it to the next consecutive operator. After the consecutive operator has consumed the data, the buffer is put back into the pool, where it can be reused. If an operator is faster than its consecutive operator, its outgoing buffer becomes eventually full. When the buffer is full, the operator has to block until all elements in the buffer are consumed and the buffer is back in the buffer pool ready to recycled. Vice versa, if the consuming operator is faster, the buffer will get empty and the faster operator has to wait until the previous operator adds new elements.

This buffer architecture ensures that tasks never produce data faster than it can be consumed. Together with a durable source like Kafka, Flink handles backpressure without data loss.

2.3.2 Micro-Batching: Spark

In micro-batching systems such as Spark Streaming [4], the calculation of a batch of records is done in specific batch intervals. Whenever the processing time of a batch lasts longer than the batch interval itself, the next batch cannot be evaluated in time, and backpressure occurs. Thus, backpressure in micro-batching can be defined by the simple inequality: $batch\ proccessing\ time > batch\ interval$.

In contrast to tuple-at-a-time engines like Flink, backpressure is not supported naturally by its design. In Spark Streaming, backpressure exhausts the resources leading to unstable states and memory overflows.[1]

For this reason, Spark Streaming added the option to support for backpressure in version 1.5. If enabled, the rate of data is controlled by measuring the processing throughput. This measurement is used to approximate a maximum number of elements per batch that can be handled by the system. The receiving operators are throttled to this value, if they are too faster. This ensures feasible data rates for the processing operators.

[1] `https://issues.apache.org/jira/browse/SPARK-7398` (04/21/2016)

2.4 Fault-Tolerance

Another important feature of streaming engines is fault-tolerance in case of crashes. In relational databases, fault-tolerance is ensured by backing up the state using redo-logs or snapshots.

However, for streams of continuous data this is not enough since messages can get lost during the downtime or recovery time of the system. Hence, full fault-tolerance requires a durable data source that can replay data. For exactly or at-least-once semantics, it is important that, in spite of replaying messages, every message is only processed once. Otherwise, the message processing guarantees would be violated.

A common data source that realizes this is Kafka [2]. It has only a minor impact on latency but ensures full durability due to persistent logs. In case of a failure, the messages are replayed from the latest offsets.

Not all data sources are durable by default. However, every data source can be wrapped by a durable data source to achieve fault-tolerance.

In summary, a streaming engine has to back up the system's state and ensure that every lost message, i. e., every message that is not part of the state backup, is re-processed.

Fault-tolerance should be implemented without causing much computational overhead. A lightweight approach doing this are distributed, asynchronous snapshots [35]. Periodic checkpoints trigger asynchronous snapshots of the system's state. This algorithm is used in Flink [1]. In case of failures, Flink restores the latest state and replays the messages that arrived since the last checkpoint. Thus, all computations within a checkpoint interval either succeed or fail atomically.

In the micro-batching approach used by Storm Trident, or Spark Streaming, one processes batches as atomically units similar to the checkpoint units in Flink. However, there is an important difference between these approaches in favor of Flink's asynchronous snapshots. In micro-batching the checkpoint pauses the stream processing before it can schedule the next micro batch, whereas it is never paused in Flink's asynchronous approach. The computation and stream flow keep running.

2.5 Continuous Query Languages

In 2005, Stonebreaker et al. proposed eight rules for streaming engines [56]. One of them was, that the system should be queryable using a streaming SQL. Some reasons for this are, that SQL is a well-known common spread language. Its declarative nature enables fast and flexible development of queries, while the query optimization is done by the streaming engine and not the user.

However, standard SQL-query semantics is not sufficient and unclear for continuous data, e. g., there are no window constructs or aggregations on continuous data. For this reason some SQL streaming dialects, such as CQL and StreamSQL were developed.

CQL

Continuous Query Language (CQL) is a streaming extension of SQL developed for the streaming engine STREAM [24, 25]. It introduces a new streaming semantics to SQL. CQL distinguishes between streams and relations. A stream is an unbounded bag of tuples, while a relation is a time-varying bag containing tuples. In contrast to traditional databases, every tuple has an attached logical timestamp, i. e., the ingestion time of the tuple. Every operator transforms a stream or relation into either an output stream or a relation. The stream-to-relation operators available in CQL transform streams into relations using sliding windows.

The idea of CQL is to rely on the well understood relation-to-relation operators, such as projection, selection, aggregations, joining, grouping, etc. For this reason, every stream is transformed into a relation before the real processing happens. Eventually, they can be transformed back into streams using relation-to-stream operators. There is no direct stream-to-stream operator: everything is converted and processed as relations and converted back afterwards.

If one want to implement a simple filter-operation on a stream it could look as follows:

Listing 2.1: Filter non-local calls

```
SELECT RSTREAM(*)
FROM event_stream [Now]
WHERE isLongDistance = true;
```

In this case, we filter all non-local call events that arrive from now on. This query is composed out of a stream-to-relation window operator, followed by a relation-to-relation operator that performs the filtering. Finally, the RSTREAM operator produces an output stream out of the resulting relation.

Note, that the general design of windows in CQL is *stream* [*Range T*]. For instance, a daily window on the call events can be defined as *event_stream* [*Range* 1 *day*]. One can also define a count-based window of size n as *stream* [*Rows n*].

StreamSQL

StreamSQL [19] goes another way. Like CQL, it distinguishes between relations and streams, but additionally it offers operators that directly work on streams. This is done by extending standard SQL by rich-window constructs and stream-specific operations [56]. StreamSQL is able to process time and count-based windows [34, 43]. In contrast to CQL, it is possible to vary the slide parameter of windows [56]. This enables tumbling windows, e. g., the current day, by setting the slide parameter to the same value as the window size.

Another difference is, that StreamSQL uses a tuple-driven instead of a time-driven model ordering, as it is done in CQL [56]. Thus, tuples are not only ordered on their timestamps, but also on their order of arrival leading to slightly different semantics [42]. StreamSQL is also extensible via expression logic or custom Java or C++ operators and functions that can be added as custom aggregates [34]. There exists also a graphical interface called *EventFlow* to define operators. It has the same expressiveness as the textual StreamSQL [43].

As StreamSQL supports stream-to-stream operators, the filter query can be defined more straightforward than in CQL:

Listing 2.2: Filter non-local calls

```
SELECT *
FROM event_stream
WHERE isLongDistance = true;
```

In contrast to CQL, it is also possible to define tumbling windows on streams. In the following exemplary query we aggregate a stream using a daily tumbling window.

Listing 2.3: Simple aggregation using a daily tumbling window

```
SELECT COUNT(*) AS count_calls,
SUM(duration) AS total_duration,
AVG(duration) AS avg_duration
FROM event_stream
GROUP BY FLOOR(event_stream.ROWTIME TO DAY);
```

This query uses a group-by statement to implement the window. We use the FLOOR function to round the time of each record down to the nearest day. The result is a stream of records – one per day. Each record contains the count, sum and average duration of calls recorded in that day.

StreamSQL also allows to define sliding windows. If one would replace the tumbling window of the previous query with a sliding window, it would look as follows:

Listing 2.4: Simple aggregation using a sliding window

```
SELECT
COUNT(*) AS count_calls,
SUM(duration) AS total_duration,
AVG(duration) AS avg_duration
FROM event_stream
WINDOW W1 AS ( RANGE INTERVAL '24' HOURS PRECEDING );
```

Both queries look similar, however, their semantics are very different. In the sliding query, an output record is generated for every newly arriving input record. Further, the aggregations goes over the preceding 24 hours instead of the last calendar day.

Summed up, one can say that StreamSQL is more extensive than CQL as it has additional operators, in particular stream-to-stream operators allowing filtering, joining, pattern matching, etc. on streams.

Today's situation

More than a whole decade passed since the first proposal of these languages. However, there is no standardized SQL version for streaming. In fact, many new streaming engines have arrived, such as Spark Streaming, Flink or Storm, that bring their own non-SQL API.

A problem is, that every system distinguishes itself in its features and abilities, thus it is hard to define a standard that can be supported by all. Jain et al. [42] conclude that

> Until fundamental model differences are settled, there is little chance of producing a good standard.

However, there are some trends to re-use the old languages or to integrate them into today's systems.

For instance, Oracle CQL is based on the original CQL [42]. The academic event streaming processor Odysseus [15] also provides an API to CQL.

StreamSQL is more broadly used, maybe due to its more extensive features. Today's, main representative is StreamBase [18], a commercialization of the original Aurora [22] project in which StreamSQL was initially developed. Another commercial system using StreamSQL is s-Server [17].

But also open-source streaming systems like Samza [3], Spark Streaming [4], or Storm [5] are currently integrating StreamSQL into their system. However, Samza remarked that they only took

StreamSQL as a base and modified it to suite SAMZA's distributed environment.[2]

There are also new SQL dialects like a model featuring continuous views used and introduced by PipelineDB [16].

[2]https://issues.apache.org/jira/browse/SAMZA-390 (04/21/2016)

3 Overview Streaming Systems

3.1 AIM

AIM stands for Analytics in Motion and is a research prototype built by the ETH Zürich Systems Group in cooperation with Huawei [32]. It is designed for the needs of the telecommunication industry, in particular the AIM workload (see Section 6).

AIM is an in-memory system that can be scaled across multiple nodes. Its three tier architecture supports hybrid workloads of stream processing and aggregations in combination with analytic queries. The three tiers are: (1) an Event Stream Processing (ESP) component, (2) a distributed key-value store, and (3) real-time analytics (RTA) processors. The architecture is shown in Figure 3.1.

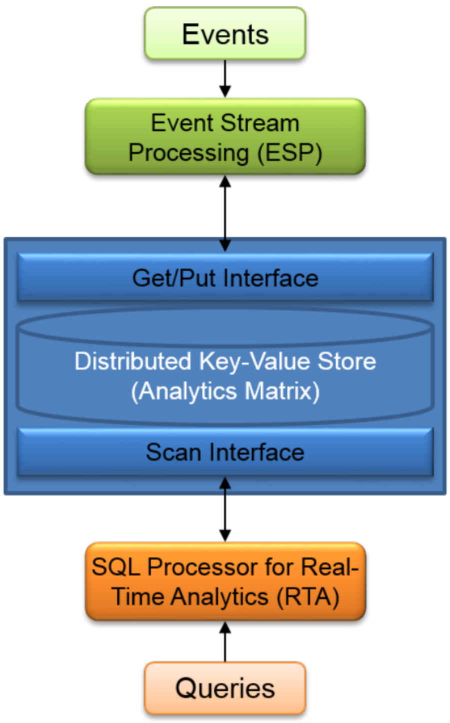

Events
Event Stream Processing (ESP)
Get/Put Interface
Distributed Key-Value Store (Analytics Matrix)
Scan Interface
SQL Processor for Real-Time Analytics (RTA)
Queries

Figure 3.1: Layers of AIM [32]

ESP The ESP nodes process the incoming event stream. For each event the corresponding record has to be updated by sending a get and put request to the storage nodes. Additionally, ESP evaluates alert triggers for every event.

Key-Value Store The key-value store is distributed across the storage nodes. Each node holds a horizontal partition of the data. Partitioning allows AIM to process the data in parallel. AIM uses its own hybrid data layout called ColumnMaps to store the data. ColumnMaps is a cache-optimized hybrid between column and row store based on Partition Attributes Across (PAX) designed for mixed OLTP/OLAP workloads.

It manages the record data in buckets. Every bucket contains multiple records that are stored column-wise. The idea is to chose the number of records per bucket so high that the whole bucket still fits into the L3 cache. However, one can also set the number of records per bucket to one to get a row store, or to the number of all records to simulate a column store.

RTA processors The processing of the analytic queries is coordinated by RTA nodes. They push the queries down to the storage nodes, where the real processing happens. Each storage node produces a partial result that is merged by the RTA node. The merged result is delivered to the client.

AIM implements shared scans for better query throughput. However, it supports only hard-coded queries, but it could be extended by a SQL processor.

AIM enables concurrent state access using differential updates. For better throughput, it maintains two deltas at once. Thus, there is always one delta available for new updates, while the other one is merged into the state.

3.2 Open Source Technologies

3.2.1 Flink

Flink [1] is a streaming engine that supports both batch processing and data streaming programs. Its runtime is designed for high throughput and low latency using a tuple-at-a-time processing model. It has an extensive and flexible support for windowing: one can chose out of different time semantics: event time, processing time and ingestion time; window types and triggers, or define own windows using the generic window API. Flink's API supports both Java and Scala to define a streaming application. A streaming application consists out of a tree of operations that is compiled into a directed acyclic graph (DAG) by Flink. The DAG is optimized by a built-in optimizer. The main optimization strategies include reusing of partitioning and sort orders, planing of pipeline and selecting of join strategies, i. e., hash-join vs sort-merge join. A core feature of Flink's design is its custom memory management that avoids the garbage collection overhead of JVMs and improves the performance using efficient data serialization.

Flink comes with fault-tolerance and exactly-once processing semantics using an asynchronous and lightweight snapshotting algorithm [35]. However, it does not support a global, queryable state. States can only be defined and used in the operators themselves. They are not exposed to the outside, which makes it hard to run external queries on them. The queries have to be inserted through the stream pipeline, which restricts their flexibility.

3.2.2 Storm

Storm [5] is an open-source distributed real time computation system. Initially designed by Nathan Marz, it was soon acquired by Twitter in 2011 and became part of the Apache Foundation in 2013.[1] Storm is designed to process unbounded streams of data in real time, for instance to identify current

[1] `http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html` (04/21/2016)

trends by analyzing tweets. Storm features a direct data flow by using Netty for message passing through the operators. Thus, it is a tuple-at-a-time system. Storm can detect failures and crashed nodes. In case of failures, it automatically reassigns involved messages in real-time.

By default, Storm guarantees at-least-once message passing and processing using acknowledgments. However, there also exists a micro-batching API called Trident providing exactly-once processing semantics. Trident is a higher level abstraction for Storm enabling stateful stream processing and distributed querying.

Storm supports any programming language since its API is based on Thrift, a software framework for cross-language development.

3.2.3 Spark Streaming

Spark Streaming extends the batch processing system Spark to a streaming engine by discretizing the data stream into micro-batches [4]. The micro-batch processing model allows high throughput, however, it comes with the latency of the batch interval. The batches are also used to enable exactly-once semantics and fault-tolerance.

Spark Streaming can use many features of Spark such as joining against against historical data, or running ad-hoc queries. Spark Streaming also supports windowing, i. e., the definition of MapReduce jobs on sliding windows [45]. In case of multiple windows, it caches the batches as long as the windows require them, i. e., until every window has benefit the intermediate result.

3.2.4 Comparison

Each of the introduced streaming engines has its own characteristics. Table 3.1 provides an overview and a comparison of the systems regarding their implementation and the accompanied properties.

The main difference of the systems are their computational model in combination with their fault-tolerance mechanism. The micro batching systems Spark Streaming and Storm Trident achieve high throughput at the cost of high latency, i. e., the batch intervals.

Both Flink and Storm use a tuple-at-a-time approach. However, Storm lacks behind Flink in some points because it uses acknowledgments for every tuple to ensure at-least-once semantics. Acknowledgments work with a timeout. If the timeout expires before a record is fully processed it is replayed even if there is no error. This, in particular the falsely replaying of records, leads to low throughput and undesirable behavior during backpressure.

Backpressure in the micro-batch systems yields a similar problem, as they use time-based batch intervals. If a micro-batch cannot be processed within a batch interval, more and more batches are queuing up.

Only Flink with its tuple-at-a-time approach in combination with distributed snapshots can handle backpressure naturally.

Another design aspect is the parallelism, i. e., how they distribute the work. All systems except of Spark Streaming use task parallelism. Each thread does a part of the workflow and passes its result to the next one. In contrast to this, Spark Streaming distributes the data among the worker threads. Thus, all worker perform the same task but on different chunks of the data.

All systems are written in JVM based languages. Thus, the problem of garbage collection has to be faced or at least experienced. Spark Streaming and Flink try to overcome or at least reduce this problem by implementing their own memory management. The idea is to introduce internal data structures that can be re-used by multiple records without being garbage collected. Flink also outsources some memory to a special off-heap memory that is slightly slower but is not garbage collected.

The different types of windows are described in detail in 2.2. Flink has a very powerful window API supporting all kinds of described windows and all kinds of time semantics, e. g., event time or processing time. The other systems lack behind in this point. They only support basic windows such as tumbling and sliding windows and are often restricted in their time semantics. In particular, the micro batching systems have a problem with windows as the window sizes must fit to the batch interval sizes, i. e., the batch interval size has to be a proper divisor of the window size. This can be problematic for multiple different window sizes, or in cases when the windows are so small that a suitable batch interval would lead to backpressure.

	Flink	**Storm**	**Storm-Trident**	**Spark Streaming**
Implementation				
Computational Model	Tuple-at-a-time	Tuple-at-a-time	Micro Batch	Micro Batch
Fault-Tolerance	Distributed Snapshots	Acknowledgments	Atomic batches	Atomic batches
Parallelism	Task parallel	Task parallel	Task parallel	Data parallel
Written in	Java	Clojure, Java	Clojure, Java	Scala
Memory Management	Yes	No	No	Yes
Properties				
Latency	Low	Low	High	High
Throughput	High	Low (Acks)	High	High
Flow Control	Natural	Problematic	Problematic	Problematic
State Management	Yes	Roll your own	Yes	Yes
Window Support	Powerful	Basic	Basic	Basic
Semantics	Exactly once	At least once	Exactly once	Exactly once
Language Support	Java, Scala	Any[1]	Any[1]	Java, Scala, Python

[1] through Apache Thrift

Table 3.1: Comparison of streaming engines

3.2.5 Kafka

Kafka [2] is a distributed messaging system that is commonly used as data source in streaming applications. It is designed for high-throughput and scalability. Kafka was originally developed by LinkedIn and became part of the Apache Foundation in 2012. Nowadays it is used by many other companies.[2] Kafka is commonly used by streaming engines for data ingestion. This has several reasons:

1. Durability
2. Scalability
3. Throughput

Kafka's messages are organized in topics. *Producers* write messages to these topics and *Consumers* can read them. To enable scalability, the topics are partitioned and replicated across multiple nodes. Each topic partition is a log, i. e., an ordered set of messages, whereas every message has a unique offset in it. These offsets can be used by the consumers to keep track of their location in the log, i. e., the messages they have already processed. This design allows multiple independent consumers relieving Kafka of the duty of individual message management. The log is persisted to disk to ensure

[2]`https://cwiki.apache.org/confluence/display/KAFKA/Powered+By` (04/21/2016)

persistence. It can also be replicated among multiple Kafka nodes to prevent data loss in case of crashes.

If a streaming engine crashes, it can restart from its latest offset in Kafka. This ensures that no message will get lost, even during the downtime of the streaming engine. Thereby, the streaming engine becomes fault-tolerant and is enabled to process every message exactly-once.

All described streaming engines integrate Kafka by providing specific producer and consumer connectors.

3.3 Database Systems

3.3.1 HyPer

HyPer [44] is a main-memory database system developed at TU München.[3] It is a relational database designed for mixed OLTP and OLAP workloads. HyPer translates every queries into efficient centric machine code using LLVM [53]. Concurrent state access is controlled using MVCC [55]. HyPer has a powerful query optimizer which is able to unnest arbitrary queries [54].

HyPer allows choosing between column and row store formats, whereas column store is the default. It supports only standard SQL semantics and does not implement any streaming extensions. However, one can add streaming semantics such as windows on streams using stored procedures and triggers. A more detailed description of this approach, in particular its drawbacks compared to native window support, can be found in Section 10.

3.3.2 MemSQL

MemSQL [13] is an in-memory database designed for streaming, transactions and analytics. MemSQL scales automatically by distributing relational data among multiple servers by partitioning the tables row-wise into smaller chunks. It follows the shared-nothing paradigm, where no resource is shared between the distributed machines. Within a single machine, concurrent state access is controlled using a lock-free implementation of MVCC.

Distributed queries are coordinated by Aggregator Nodes, that aggregate the partial results and send them back to the client.

MemSQL generates C++ code for every query plan that is compiled to machine code. To avoid the high compiling time of C++ among repeated queries, the query plans are cached.

MemSQL ensures durability using logging. In a distributed setting, it optionally does full snapshots of nodes to recreate the state of a node if it went down.

MemSQL does only support standard SQL and thus there is no window support. In contrast to HyPer, there is no support for stored procedures, which makes it impossible to directly add window semantics into a MemSQL application. One has to implement the window semantics on the client side, which adds round-trips and latency to the streaming application.

Another way to implement streaming into MemSQL is Streamliner [14]. Streamliner is an open-source extension to MemSQL that is able to build real-time data pipelines in combination with Spark Streaming. It aims to organize the whole ETL-workload at one point and without writing additional code. The data is *extracted* from real-time data sources like Kafka, *transformed* by Spark Streaming, and then *load* into MemSQL. Loading it into MemSQL persists and ensures its durability, thus parts of the Spark Streaming checkpointing can be by-passed. In general, such streaming applications can be designed using the graphical user interface of Streamliner. There is no need to write additional code, but if needed, it can still be plugged into Spark.

[3]Throughout this work we refer to the research prototype of HyPer and not its commercialized spin-off

3.3.3 TELL

Tell [20] is an open-source database project of the Systems Group at ETH Zürich. Its design is highly modular: all components can be used and developed isolated.

The main component of Tell is the main-memory storage TellStore. It stores the data using either a row store or ColumnMap, i. e., the hybrid storage developed for AIM. ColumnMap facilitates mixed workloads as it supports fast scans and fast record updates at the same time. TellStore implements its data structures lock-free and partitions the data among distributed nodes. The storage nodes can be accessed by decoupled processing nodes. These processing nodes are almost stateless and thus can be scaled easily.

Tell and AIM have multiple design elements in common. Both can batch and process multiple queries at once using shared scans. They allow concurrent state accesses using differential updates [46]. Finally, as already mentioned, both use ColumnMap.

Tell provides two interfaces to access the data. On the one side, a C++ interface (TellDB) for transactions and on the other side, a Java interface (TellJava) for read-only analytics. The Java interface is also used to integrate Apache Spark and Facebook's Presto.

4 Related Work

4.1 History of Event and Stream Processing

In the early 1990s, there have been two parallel trends: event stream processing (ESP) and complex event processing (CEP) [50].

4.1.1 Complex Event Processing

The initial motivation of CEP was to analyze event-driven simulations of distributed systems architectures [49]. One wanted to analyze independence and causal relationships between events. For example, event A is caused by B, or A happened independently of B, or boolean combinations like A and B. Therefore, the events are clustered into "event clouds" with a timing and a relational dimension. CEP aims to extract information out of such clouds [49].

Some early representatives of CEP are:

Rapide Luckham et al. developed Rapide at Stanford – an event driven language for modeling and simulating event activity in distributed systems [51]. They developed multiple techniques to analyze events that are produced during the simulation of a distributed system. They used Rapide to identify errors at the different layers of a distributed system's architecture.

Infospheres Infospheres [38] was a research project at Cambridge led by K. Mani Chandy. Part of the work was the design of a distributed system for CEP. Its key feature is the integration of the Chandy-Lamport algorithm for taking global snapshots of the system. This enables the system to recover in case of a failure by resurrecting the system from the snapshot and replaying the events from the snapshot onward [38].

Apama Apama was initially founded and developed at Cambridge University by John Bates and Giles Fraser [6]. In 2013 it was acquired by Software AG.[1] It was built to identify and analyze pre-defined events in streams. Its main application is to track market patterns in capital markets.

4.1.2 Event Stream Processing

The motivation for ESP is to perform data analysis in real-time on streams. A stream is only a sequence of events ordered by time, whereas the event clouds used in CEP are compositions of multiple streams from different sources.

ESP discretizes the data of continuous streams in slices using windows. In general, queries in ESP require less memory as those in CEP. This has two main reasons: firstly, they usually do some sorts of aggregations on the streams and thereby only have to maintain the aggregated state. Secondly, even if there are multiple operations on the stream, the operators usually only calculate a result and pass it to the next operators. Contrary to CEP, the events can usually be discarded after they have been processed.

[1] `http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/`

4.1.3 Conclusion

Nowadays, there are systems supporting both CEP and SEP. However, CEP is often only an extension of the SEP core. For instance, the streaming engine Flink offers an additional CEP library. However, these extensional libraries are often restricted to features like pattern detection or temporal relations within a single stream. Other systems focus more on the event processing part and extend it by stream processing functionality.

Esper [7] is a software supporting both CEP and ESP. It was created in 2006 and is developed under open source license. Streams can be queried in Esper using its *Event Processing Language* (EPL). EPL is an extension of SQL-92 that supports stream operations such as continuous queries, aggregation, joins, causality and missing events. But Esper also supports event processing features such as pattern matching, temporal causalities, filtering, aggregations, event windowing and joining.

4.2 Stanford Stream Project

The Stanfordstreamdatamanager (STREAM) project was a research project at Stanford from 2003 to 2005 [24]. The motivation was to build a system that is able to run queries over continuous, unbounded streams of data. They developed the general-purpose prototype STREAM to investigate data management and query processing on streams.

As stream applications are long-running and the characteristics of the streams can change over time, StreaMon – an adaptive query processing and optimizing engine was integrated into STREAM [30]. StreaMon adds two new components to the query executor: (1) a profiler, which collects and maintains statistics of the stream and plan characteristics, and (2) a re-optimizer that adapts the plans and memory usage to the current input characteristics.

They identified many problems in traditional databases regarding stream processing and provided solutions to some of them in their STEAM implementation.

Semantic of Continuous Queries Standard SQL-query semantics is not sufficient and unclear for continuous data, e. g., there are no window constructs or aggregations on continuous data. They introduced Continuous Query Language (CQL) with a new semantics to address these problems [25]. Their semantics distinguishes between streams and relations. Streams are *unbounded* bags of tuples, while relations are *time-varying* bag of tuples. In contrast to traditional databases, every tuple has an additional logical timestamp, that is attached to the tuple during its ingestion. Every operator transforms a stream or relation into either an output stream or a new relation. The transformation of a stream into a relation is done using the concept of sliding windows. For more details about CQL, see Section 2.5.

Adaptive Operator Scheduling A continuous query in STREAM is a tree of operators belonging to the three kind of classes: relation-to-relation, stream-to-relation and relation-to-stream. Each operator class transforms data according to its name, e. g., a stream-to-relation takes a stream as input and produces a relation as output. The operators are connected by queues: they consume from input queues and they produce to output queues. The goal of STREAM's query optimizer is to reduce the required amount of memory used by these queues. Therefore, it uses a strategy called chain scheduling. The basic idea in chain scheduling is to break up query plans into disjoint chains of consecutive operators. STREAM favors operators that consume numerous tuples per time unit and produce few tuples as output.

Adaptive Joins Joins on streams differ from relational joins in two points: (1) input arrives at highly variable and unpredictable rates and (2) it is more important to get a prefix of the output fast than to complete the whole computation as soon as possible to maintain the streaming data flow [60]. STREAM uses MJoins (multiple join) operators instead of a single join operator to produce output in a streaming fashion and at higher rate [30].

Multiple input streams can be joined at the same time by maintaining own HashTables of tuples for every stream. Whenever a new tuple arrives, it is inserted into the corresponding HashTable of its stream and probed with the HashTables of the other streams until the probe fails or it matches for all HashTables. In the latter case the joined tuple is emitted. The probing order should be optimized according to the selectivity of their join predicates to make the probing sequences as short as possible [60].

Babu et al. developed a greedy algorithm that does this ordering of joins and filters adaptively by sampling the data to profile the selectivity and the processing time of the predicates [27].

It turned out that the sampling technique also works well for the determination of the join order in traditional relational databases, in particular better than cardinality estimation [47]. However, it is still not spread in current streaming engines. Flink, is the only of the presented streaming engines, that has an optimizer, however, they do not use adaptive join ordering due to “the high risk in the absence of good estimates about the data characteristics”.[2]

Adaptive Memory Optimization The goal is to reduce the amount of memory required for continuous queries by exploiting stream data and arrival patterns [29]. A useful pattern within a stream is the ordering of its elements, e. g., all elements arrive in an ascending order according to their timestamps. If you have multiple streams, there could also be a pattern like: all elements of stream A arrive before their join partners arrive in stream B. Such properties are useful to reduce the size of the synopsis by order of magnitudes because one can discard unnecessary tuples, i. e., those who cannot be part of the join anymore [29]. For instance, if a tuple arrives in Stream B and there is not yet a join partner in stream A, the tuple can be discarded because there will never be one due to the stream arrival properties. The challenge for the query processor is to notice such stream properties. StreaMon’s profiler monitors the streams for such properties. However, there is still an amount of uncertainty, as stream patterns might occur only randomly or change over time.

Current streaming systems like Flink [1], work with watermarks that can be inserted into the stream to signal that no older timestamp will arrive from now on.

Adaptive Caching Subresults of stream joins can be cached to avoid recomputing of intermediate results [24]. STREAM implements this, by adaptively adding and removing of subresult caches using A-Caching [28]. A-Caching manages caches and their sizes dynamically by estimating cache benefits and costs by monitoring the data characteristics.

Some of the current systems are also providing intelligent caching, however it is not adaptively in general. For instance, Flink makes use of existing partitioning, sort orders or caching throughout their operators, but it is only done at compile-time. However, one has to notice that the memory in today’s system is far higher than it was during the time of STREAM (2002-2005). For this reason, the replacement of caches is often not required and desired any more since it is possible and faster to keep everything in memory.

Backpressure The problem of backpressure (see section 2.3) due to unpredictable load peeks was already recognized during the STREAM project. They distinguish the cases when the CPU or the main-memory is the limiting resource. However, in both cases they only provide inaccurate approximation as solution due to load-shedding, i. e., dropping elements. They justified this by saying that it would be acceptable for most streaming engines to degrade accuracy during

[2]`https://cwiki.apache.org/confluence/display/FLINK/Optimizer+Internals`

load spikes. In today's streaming applications, data loss is in general not accepted anymore, and thus current systems implement other ways to deal with backpressure as described in Section 2.3.

Crash Recovery They asked for durability guarantees as in traditional databases. However, they see some challenges, for instance that streaming data is continuous, which is problematic during the downtime of a system. This problem is addressed by current systems as described in Section 2.4.

Many of their ideas and suggestions are fully integrated or at least considered in today's streaming engines, e. g., windowing, operator scheduling, crash recovery, or the treatment of backpressure. However, there are still some open points, or points that can be improved. For instance, most streaming engines, like Storm or Flink, sees streaming more from the stream flow perspective than from the database perspective, i. e., transform a stream into a queryable data-set, as STREAM does. This makes it hard to query a stream in modern streaming engines like it can be done in traditional databases and as it was suggested by STREAM.

Further, most streaming engine lack a declarative query language. This might be the explanation, that these systems do not come with a query optimizer, as the query plans are explicit given, i. e., programmed by the data engineers. If there is a kind of optimizer, it is often restricted and does not do adaptive optimizations as it was done in STREAM.

4.3 Aurora/Borealis

The Aurora project was initiated with the goal to build a single infrastructure that can efficiently handle and analyze huge volumes of continuous data streams in real-time [22]. The data analysis should be able to analyze continuous streaming data and historical data at the same time. Aurora was a collaboration between Brandeis University, Brown University, and MIT that was superseded by the Borealis project in 2005 – a distributed version of Aurora [21].

The key features of Aurora/Borealis are:

Boxes and Arrows In contrast, to the declarative language CQL in STREAM, Aurora provides a procedural "boxes and arrows" approach as the primary interface for the application developer [26]. The boxes and arrows paradigm, leads to a directed graph without loops, i. e., a DAG of processing operations [22].

Profiling/Monitoring Their goal is to perform run-time optimization of the DAG without big disruption like going offline. During execution Aurora gathers runtime statistics such as the average cost of operator execution and selectivity. It is used to apply three kinds of optimizations: (1) project out all unneeded attributes using operator signatures, (2) combine non-pipeline breaking operators, e. g., two filters into one; pipeline breaking operators like windows cannot be combined; and (3) reorder boxes: e. g., push down filters [22].

Train Scheduling The operators in Aurora are scheduled using train scheduling heuristics. The idea is, to generate long tuple trains in front of every operator by queuing as many tuples as possible without processing them. Each operator should process a tuple train completely at once to avoid context switches between operators. The resulting tuples should be directly passed to the next operator to avoid the overhead of transferring the output tuples into the next queue.

Load Shedding Aurora detects resource overload and performs load shedding based on application-specific measures of quality of service [26]. In case of an overload of tuples, Aurora implements two approaches for load shedding. Its first approach is to drop randomly selected tuples at strategic points in the network [22]. Its second approach is semantically, according to the importance, i. e., the utility of the tuple [58]. Aurora calculates the expected utility by building frequency histograms over streams and calculating the respective loss/gain ratios, i. e., the quality cost of dropping a tuple compared to the saved processing time. It filters tuples with the lowest utility by dynamically inserting and removing of drop operators into the operator graph [58]

Distribution Borealis extended Aurora's feature by the ability to distribute operation across a cluster of commodity machines [61]. It introduces a greedy algorithm to avoid overload and to minimize end-to-end latency. Borealis does this on two levels: first it optimizes the distributed load statically using a global algorithm and then during run-time it is optimized dynamically using a pair-wise algorithm [23]. The dynamic load distributor minimizes the average end-to-end processing latency even in fluctuating and bursty workloads [61].

Delta-Checkpointing Borealis uses delta-checkpointing of operators to provide a parallel fine-grained backup and recovery [41]. This works as follows: Borealis identifies subgraphs of the execution that have to be checkpointed automatically to avoid inconsistent backups. These atomic units do not have any interdependency and thus can be checkpointed independently. The individual checkpoints can be spread out over time or processed in parallel [41].

During checkpointing, every operator sends a snapshot of its state to a backup server. Since snapshotting the whole state is often too expensive, the operators only send deltas of their states [41]. For instance, windowed aggregation operators use dirty bits to mark windows and group by values as dirty, i. e., created or modified after the last checkpoint.

To reconcile its state, a node re-initializes operator and queue states from the checkpoint and reprocesses all buffered input tuples [31]. This approach guarantees no data loss, as long as the input tuples since the last checkpoint can be replayed [41].

Borealis also features an intelligent choosing of when to trigger a checkpoint: it selects to checkpoint the operators that will most reduce the load of recovery [41].

Dynamic query modification Borealis also introduced a Query Processor (QP) into Aurora, that enables the user to modify various data and query attributes at run time [23].

Aurora has been commercialized into StreamBase [18] and is still available today.

Some design ideas are also used in new streaming engines. For instance, Flink also achieves fault-tolerance by using a checkpointing algorithm [35]. Their algorithm is based on the asynchronous state snapshot algorithm proposed by Chandy and Lamport. Due to its asynchronous design, the algorithm can work without stopping the current processing. However, in contrast to the Aurora/Borealis checkpointing, Flink also lacks behind in some points. Flink does not use delta-checkpointing and thus it is always backing up the whole state. This can be crucial in particular for large states. Further, Flink always snapshots the whole state of the system at once. It can not select individual parts of the system and snapshot them independently. Dynamic snapshotting according to the current system load like in Borealis is also not supported. Flink's snapshotting is only done cyclic at fixed-size checkpoint intervals.

4.4 PipelineDB

PipelineDB is an open-source streaming database, built into the PostgresSQL core [16]. The goal of PipelineDB is to provide a simple product for streaming and analytics applications development that requires just SQL like a conventional database. This is done by introducing continuous views into their SQL-API to support streams and continuous operations on them such as aggregations, window queries and joining on tables. Continuous views can be seen as real-time materialized views, optimized for high-throughput and incrementally updates. These continuous views are maintained over streams that can be fed with data using SQL-insert statements like on standard SQL-tables. PipelineDB collects these stream input tuples in a stream buffer, where they are concurrently processed and aggregated in micro-batches by workers. The resulting partial aggregations are sent to a combiner process that continuously merge them into the continuous view. Only this last continuous view is actually stored in the database.

Listing 4.1: Maintain a continuous view on a stream in PipelineDB

```sql
CREATE CONTINUOUS VIEW firstExample AS
SELECT entity_id::integer, COUNT(*), SUM(a::integer), AVG(a::integer)
FROM eventStream
GROUP BY entity_id;

--Feed stream
INSERT INTO eventStream (entity_id, a) VALUES(1323, 21);
INSERT INTO eventStream (entity_id, a) VALUES(392, 42);
INSERT INTO eventStream (entity_id, a) VALUES(1226, 17);
```

The continuous views in PipelineDB can be queried like tables in traditional database systems.

Listing 4.2: SQL query on continuous view

```sql
--Query a continuous view like a normal table or view
SELECT entity_id
FROM firstExample
WHERE sum > 1000 AND count < 30;
```

5 Simplified and Micro-Benchmarks

During building the AIM workload, we conducted various smaller benchmarks and micro-benchmarks, to decide which system or implementation to chose. These benchmarks were also used to identify expensive operations, i. e., the bottle-necks, in order to further optimize these crucial parts.

Many of the faced problems are of general nature such as serialization or data ingestion. Thus, the insights of these benchmarks can easily be adopted to other streaming workloads.

5.1 Systems and Setup

System	**Version**
Storm	0.10.0
Spark Streaming	1.6.1
Kafka	0.9
Flink	1.1-Snapshot

Table 5.1: Specification of evaluated systems

Table 5.1 shows the versions of the evaluated systems.

All experiments were run on an Ubuntu 15.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM. The machines were connected using 40 GBit/s InfiniBand QDR links. Unless specified otherwise, we measure the single-thread performance in all experiments of this section.

We do not only measure streaming engines, in some cases, we conduct micro-benchmarks with plain Java code. All benchmarks that use plain Java code were conducted using JMH [11] – a framework for benchmarking JVM based languages. JMH helps to avoid dead-code elimination and other distorting JIT optimizations such as constant folding. We use 20 warm-up iterations and 20 measured iterations throughout all micro-benchmarks.

5.2 Serialization

In distributed applications, data has to be serialized before it can be transferred. In general, one can identify two main requirements regarding serialization: (1) it should be fast, i. e., low processing costs for de-/serialization, and (2) the data format should be compact.

Another goal, that is also desirable but in our case not that important compared to the two performance requirements, is maintainability and error-proneness. If the data types change, one want to be able adapt the serialization without big effort.

We decided to analyze and compare two approaches of serialization: (1) a handwritten version, using only bit-operations and no compactification, and (2) a representative of serialization frameworks. We chose Kryo [12] as framework, as it is commonly used with streaming engines. Sometimes it is even directly integrated into the systems.

Our experiments should answer the question, if the effort of a handwritten serialization solution pays off.

For the following benchmarks we use the same event structure as in the AIM workload. There, an event consists out of five fields: one long, two integers, one float, and one boolean.

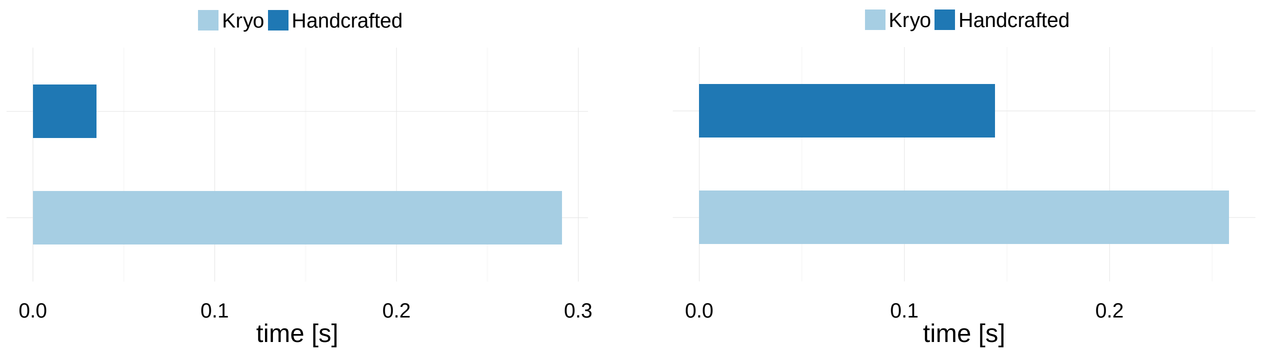
If one encodes these event-fields directly into a byte array one needs 21 bytes. Kryo compresses the events further. The compressed events of Kryo require only 14 to 19 bytes depending on the data. Thereby, one saves up to a third of memory.

The custom implementation (see Listing 5.1) has to implement serialization procedures for every of the four different event data types. The serialization procedure of integers can be re-used for the float value because they both have 32 bits. Integer and Longs are serialized using bit-shifting operations, and the boolean value is directly encoded as a byte. In spite of using loops and re-using much code, the implementation is already quite long and inconvenient to maintain. The code of de-serialization is not even considered yet. In contrast, the code for Kryo version (see Listing 5.2) is pretty tight and straightforward.

Listing 5.1: Serialize event using bit operation

```java
public byte[] serialize(Event event) {
  byte[] eventAsBytes = new byte[21];
  int curInd = 0;

  insertIntoByteArray(eventAsBytes, curInd, event.entity_id);
  curInd += Integer.BYTES;

  insertIntoByteArray(eventAsBytes, curInd, Float.floatToIntBits(event.cost));
  curInd += Float.BYTES;

  insertIntoByteArray(eventAsBytes, curInd, event.duration);
  curInd += Integer.BYTES;

  insertIntoByteArray(eventAsBytes, curInd, event.timestamp);
  curInd += Long.BYTES;

  byteArray[eventAsBytes] = event.isLongDistance ? Byte.MAX_VALUE : Byte.MIN_VALUE;
  curInd++;

  return eventAsBytes;
}

private static void serializeInteger(byte[] byteArray, int curInd, int value) {
  for (int i = 0; i < Integer.BYTES; i++) {
    byteArray[curInd + i] = (byte) (value >>> ((Integer.BYTES - i - 1) * Byte.SIZE));
  }
}
private static void serializeLong(byte[] byteArray, int curInd, long value) {
  for (int i = 0; i < Long.BYTES; i++) {
    byteArray[curInd + i] = (byte) (value >>> ((Long.BYTES - i - 1) * Byte.SIZE));
  }
}
```

Listing 5.2: Serialize event using Kryo



(a) Serialize 1 mio events

(b) Deserialize 1 mio events

Figure 5.1: Handcrafted vs. Kryo serialization

```
//Setup Kryo using a ByteArray as output
private final Kryo kryo = new Kryo();
private ByteArrayOutputStream stream = new ByteArrayOutputStream();
private Output output = new Output(stream);

public byte[] serialize(Event event) {
  kryo.writeObject(output, event);
  output.flush();
  return output.getBuffer();
}
```

Figure 5.1 shows the time required for the serialization (5.1(a)) and deserialization (5.1(b)) of one million random events. The handwritten solution serializes the events in 0.04 seconds, whereas Kryo needs 0.29 seconds, i. e., seven times longer. For deserialization, the difference is smaller, i. e., only a factor of 1.8. Kryo takes 0.26 seconds and the handwritten solution 0.14 seconds.

For streaming engines, the deserialization performance is more important since the serialization is done externally at the data sources. For this reason, the serialization costs should not directly hurt the performance of the streaming engine.

For deserialization, Kryo is also slower, but the difference is smaller. Kryo is still able to deserialize almost four million events per second. This is a cheap operation compared to other streaming operations such as windowing or aggregations. In practice, one has to decide if the performance overhead of Kryo is negligible compared to the remaining workload.

In the AIM workload, we have 10,000 events per second. In this case, the overall performance overhead of using Kryo would be 1 ms per second compared to the handwritten solution assuming that all incoming events are processed single-threaded. Using multi-threaded processing the overhead would decrease almost linearly. We decided, that this overhead is an acceptable trade-off compared to the benefits of Kryo such as compactification, robustness, and maintainability.

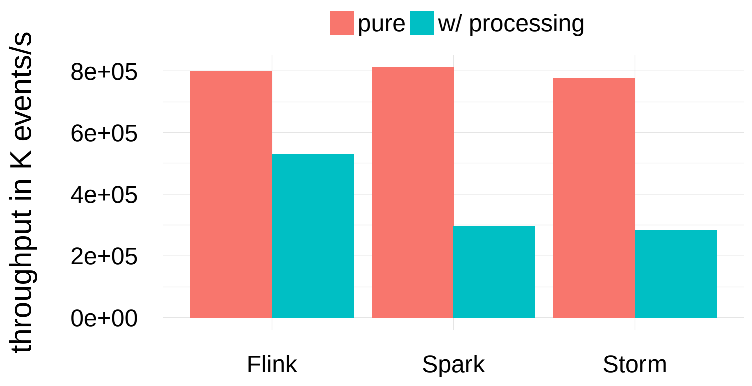


Figure 5.2: Kafka event ingestion throughput

5.3 Event Ingestion

A common way of event ingestion in streaming applications is the usage of Kafka as a data source [2]. Every streaming system described in Section 3.2 has an integrated connector to Kafka. In this section, we measure the ingestion throughput using the same types of events as in the AIM workload. In this section, we also use the AIM events, see Section 6, for our benchmarks.

First, we compare the pure event receive rate of the systems, i. e., without performing any further processing such as deserialization. Thereby, we measure the pure event receiving rate without any further processing by the streaming engine.

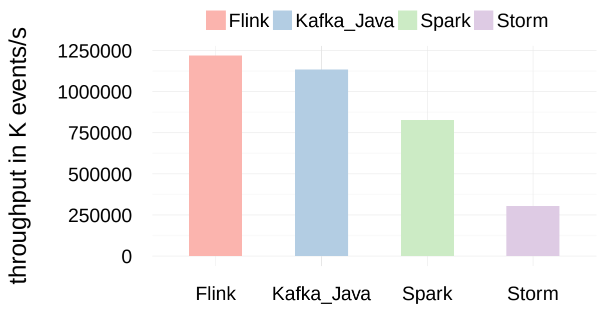
In a next step, we want to measure the “real” ingestion performance and not only the receiving performance. Therefore, we deserialize the events and pass them through a dummy operator. The dummy operator simply iterates over all incoming events and measures the throughput. This way, we measure the throughput that is usable for the first real operator.

In every system, one has to specify a function that deserializes the incoming Kafka messages. In Flink and in Storm, one deserializes the Kafka message to an object of the desired type, in our case an event. Storm’s deserialization API is different. The deserialize-function does not return an object but a list of objects. Each element of this list represents a field of the event object. Storm does this because it internally treats all records as ArrayLists. The disadvantage of this approach is that converting an event into a list costs additional object creation calls, i. e., one for each field.

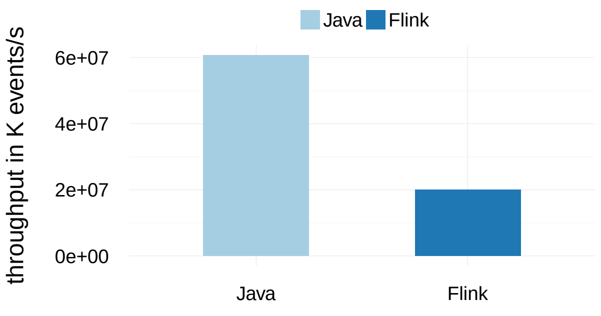
Figure 5.2 shows the event throughput for the different systems. The “pure” throughput is in all systems almost the same, i. e., about 800,000 event per second. Probably, their Kafka consumer implementation do not vary much.

The throughput of ingestion *with* processing is more interesting, as it shows how fast the events can be processed. Flink is the system with the fewest slowdown. It achieves about 530,000 events per second. Spark Streaming and Storm are significantly slower with 295,000 and 280,000 events per second. Spark Streaming and Storm might suffer from the batch processing in this case, as they have to buffer all incoming events into batches, before they can be further processed. Storm has an additional overhead compared to Spark Streaming since it has multiple object creations per event due to its conversion in lists.

In general, if one has to maintain a higher event rate, one can increase the throughput by adding more partitions to Kafka. This enables consuming the data from Kafka in parallel. The streaming engines can automatically map the available partitions to their worker threads. If the number of



(a) External event-generation w/ Kafka



(b) Plain Java vs. Flink with internal event-generation

Figure 5.3: Total call cost function throughput

partitions is greater than the number of available streaming engine threads, the streaming engine distributes the partitions as equally as possible among the threads. If it is the other way round, only some threads can read from Kafka because a partition can only be read by one consuming thread at a time. In this case, the load of data has to be partitioned in a subsequent step by the receiving threads. Ideally, the number of partitions equals the number of threads. This allows a one-to-one mapping of partitions to threads.

5.4 Function Evaluation

The goal of this experiment is to evaluate, how well the streaming systems are able to handle stored functions. In databases, some systems such as HyPer [44], compile the functions before evaluation and others such as PostgreSQL [57] only interpret them at run-time.

Inspired by the telecommunicative AIM workload, we designed a function for this experiment that calculates the overall sum of call-costs. The costs for a single call is calculated as:

$$costs = basePrice + duration * price * dayTimeWeight * areaWeight$$

The values of *basePrice, price, dayTimeWeight*, and *areaWeight* are stored in the system for each customer. The duration of each call can be extracted out of the incoming call events. To avoid the overhead of maintaining a state for each customer, we use the same constant values for each customer in this experiment:

$$\text{basePrice}=0.15,\ \text{price}=0.10,\ \text{dayTimeWeight}=0.9,\ \text{areaWeight}=0.8$$

The values of the call durations are uniformly distributed between 0 and 30. For each incoming call, we calculate its costs and add them to the overall sum of costs.

All systems are able to generate data internally, however, the way it is done distinguishes. The tuple-at-a-time system Flink, profits from internal event-generation, while the micro-batching systems Storm-Trident and Spark Streaming are faster if the events are ingested externally. To make the results of the system more comparable, we use Kafka for event ingestion in all systems. To have a baseline, we implemented a Kafka consumer in Java doing the same workload. We chose to use a Java implementation, as the function itself is specified via the Java API in all systems.

The results of this benchmark are shown in Figure 5.3(a). Flink is the fastest of all systems with 1.2 million function calls per second and event slightly bet the plain Java implementation.

Probably Flink's Kafka connector is more optimized than the simple consumer we used in our Java implementation. Spark Streaming managed to do 827,000 function calls per second, while Storm only achieves 302,000.

To eliminate potential overheads in Kafka, we also implemented versions without it using the internal event-generation of the engines. The internal event-generation is compared to a version using plain Java for both call generation and processing.

Figure 5.3(b) shows function calls per second using internal event-generation. It only shows Flink and the plain Java implementation, as the other system were not bound by Kafka. Without the overhead of Kafka, Flink was able to achieve 20 million function calls per second. This is only three times slower compared to the plain Java implementation of the benchmark. The overhead of Flink probably comes from Flink's source function. Flink uses an asynchronous loop to produce new call events, whereas in the plain Java code both the data generation and processing happens in one tight loop. Thus, there are no context switches necessary.

In summary, it seems like Flink makes fully use of the just-in-time compilation of the JVM. This makes sense, as Flink is written in Java. Storm and Spark Streaming also are written in JVM-based languages (Scala and Clojure) and thus should also be compiled at runtime. However, the streaming engines themselves seem to have too much overhead to achieve the same performance as Flink.

5.5 Simple Aggregation Benchmark

This experiment compares the throughput of the streaming engines for rolling aggregations on streams.

It can be seen as a simplification of the write-part of the AIM workload. The event stream consists only out of an entity_id and an integer value, i. e., the duration of the call. For each entity record, we maintain three types of aggregates: (1) count of calls, (2) the sum, and (3) the average duration. Note, that all aggregations are rolling, i. e., there is no windowing. The state holds 10,000 entity-records.

We measure the throughput with internal event-generation and external ingestion with Kafka.

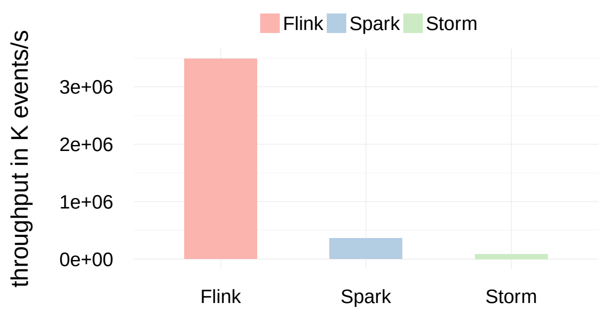
The benchmark measures the time for processing 10 million randomly generated call-tuples.

Every tested system has built-in aggregation operators. However, none of them allows maintaining a state of multiple aggregation on a single stream. For this reason, we have to use custom aggregation functions in all systems. The main differences between the systems is the way we can access the aggregated states.

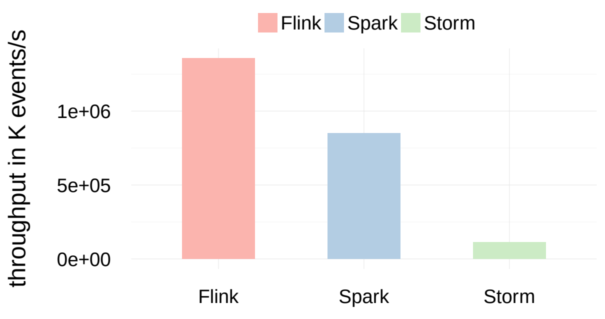
Spark In Spark Streaming we transform the input stream into a paired key-value stream. Spark Streaming can maintain and continuously update an arbitrary state using an *UpdateState-ByKey* operation. Therefore, one has to specify an update function that takes the previous state and the new input value and generates a new state value. In our case, we use a function that does the desired aggregations. Before calling the function, Spark Streaming groups the values of each batch according to their keys. Subsequently, the function processes all elements of the same key at once in a batch-fashion.

According to its documentation, Spark Streaming applies this state update function for all existing keys, regardless of whether the keys are contained in the batch.[1] This might be useful in cases, when you need to update the state regardless of the input data. For example, if you have a windowed state, and need to reset it because a new window has started. However, in our case, where the values are only updated according to the input values, it seems inefficient.

[1] `http://spark.apache.org/docs/latest/streaming-programming-guide.html#updatestatebykey-operation` (04/21/2016)



(a) Internal event-generation



(b) External event-generation w/ Kafka

Figure 5.4: Event processing throughput

This might get even worse in cases where you have many different keys in the state and only want to update a small fraction each batch. Unfortunately, Spark Streaming does not allow getting rid of this behavior.

Spark Streaming fully encapsulates the state management. Each call of the update function runs isolated on the corresponding state of the key. It cannot see or modify the state values of the other keys.

Storm In Storm, the state management is also encapsulated by Trident. We use a custom *ReducerAggregator* that “reduces” the input streams to their aggregates and updates the state. The reduce function gets a tuple and the state value of the corresponding key, and returns the updated state value.

Like in Spark, the single key values are isolated within the function calls. However, the processing in Storm is different. The function gets only one tuple at a time and not all tuples of the batch with the same key. Thus, it is only called for keys that are part of the input stream.

Flink In contrast to the other systems, the state in Flink is kept and managed inside of the operator itself. We use a custom *FlatMap* function that is called for every tuple.

Flink leaves the state management to the user. This offers great flexibility: in our case we use a standard HashMap, but one could also implement a more specialized state, that is tailored to the use case. For instance, one can store the data into an array, if the data set is fixed; or use a column store instead of a row store. Another benefit is, that the state is fully exposed to the user. When a tuple is processed, one can access all other state entries, too. This can be used for instance in window applications to reset all windowed aggregates at once, but also at analytic applications that need to query more than a single key value.

Figure 5.4(a) and 5.4(b) show the tuples per second for internal event-generation and external event ingestion via Kafka. Flink's tuple-at-a-time approach using a custom state outperforms the other systems in both cases. Using internal event-generation, Flink processes almost 3.5 million events per second, while Spark Streaming and Storm fall behind at about $363,000$ or $88,000$.

The external event ingestion throttles Flink's pipeline to about 1.35 million events per second. However, it is still faster than Spark Streaming and Storm with $850,000$ and $112,000$ events per second.

Spark Streaming and Storm benefit from external event-generation, as their internal generations does not facilitate batch processing. There are some reasons, why Flink is still faster. For one thing, it does not have the overhead of an external state management. In Flink, the state is kept directly in the operator itself which enables fast access to it. Additionally, Flink has got an integrated optimizer. It recognizes data pipelines and compiles a data-centric query plan that facilitates fast processing of single tuples. The micro-batching systems Storm and Spark Streaming, do not have such optimizations.

6 AIM Workload

The AIM workload is proposed by Braun et al. in [32]. It has its origin in the telecommunication industry and was designed in cooperation with Huawei. It performs stateful stream processing and analytics on a stream of call events.

The core components of the workload are (1) windowed aggregations over a stream of call events, (2) alert evaluation, and (3) analytic queries over the whole state. Both events and analytic queries are ingested via clients. Similar to the original paper, we refer to the event ingesting client as SEP-client and to the analytic client as RTA-client.

6.1 Motivation

The workload has many characteristics that makes it a representative streaming workload. It is a mixed streaming workload combining both OLTP and OLAP at the same time. Further, it comes with the demand for realtime alerts. This workload is also good for benchmarking as it consist out of many facets that can be easily measured isolated. Thus, one can cover a broad range of different streaming applications. Additionally, there are many variable parameters such as number of aggregates, number of entity records, i. e., the size of the state, the update or query rate, and many others. By varying these parameters one can test the scalability of systems, or adapt it to individual use cases. Finally, this workload is driven by the industry, i. e., Huawei, and thus is a representative real-world scenario.

6.2 Aggregations

The input stream consists out of call events. Figure 6.1 shows the fields of each event. The events contain an entity id, i. e., the customer who did the call, the call duration and costs, and a boolean flag signalizing whether it is a long distance call or not.

Call-Event	
integer	entity_id
integer	duration
float	cost
long	timestamp
boolean	isLongDistance

Figure 6.1: Fields of a call event

The AIM workload maintains various aggregations over these call events and stores them in an analytics matrix, i. e., a materialized view. An exemplary analytics matrix is shown in Table 6.1.

Each aggregate is composed out of following components:

1. Event property: call cost or duration

2. Filter: no filter, local-call filter, non-local-call filter
3. Aggregation type: count, sum, min, max
4. Window
   - Sliding: e. g., this day, week, or month
   - Tumbling: e. g., last 24h, last seven days, or last 30 days
   - Event-based: e. g., last 10 events, last 100 events, or last 1000 events

In our experiments we use 42 distinguish aggregates. They are composed as follows:

$$\textit{event properties}\ (2) \times \textit{filters}\ (3) \times \textit{aggregation type}\ (4) \times \textit{windows}\ (2)$$

The windows are restricted to tumbling windows, i. e., daily and weekly, like in the original AIM implementation [32]. Note, that the count aggregate type is only combined with the filters and the windows, to avoid redundant aggregates.

In real world applications of the telecommunication industry the number of required aggregates can be up to one to two thousand. For this reason, Braun et al. duplicated the aggregates and ended up by a total of 546 aggregates [32]. Unless stated otherwise, we will use the same number of aggregates throughout our experiments.

subscriberID	no calls today	total duration today	total cost today	...
230212	5	302s	$3.25	...
912321	0	0s	$0.00	...
321201	11	621s	$12.00	...

Table 6.1: Example Analytics Matrix

6.3 Alert Evaluation

The second functionality of the AIM workload is the evaluation of alerts. The alerts in the AIM workload are used as business rules.

Business rules exist for different purposes, e. g., marketing campaigns (e. g., rule 1) or for customer alerts (e. g., rule 2).

Each business rule:

1. *takes* the current event and the corresponding entity record
2. *evaluates* rule on the event and/or on the entity record
3. *triggers* action when condition is satisfied

Typical actions would be to send a message to the customer or create a new event.

RULE 1 Inform subscriber that the next 10 phone minutes today will be free
WHEN: *number-of-calls-today* > 20 and *total-cost-today* > $100.00 and *event.duration* > 300s

RULE 2 Advise subscriber to activate the screen lock as it appears that his smart phone is making calls on its own
WHEN: *number-of-calls-today* > 30 and *total-duration-today* / *number-of-calls-today* < 10 s

6.4 Analytic Queries

The third part of the AIM workload are analytic queries that could be used for decision support. The queries should run on fresh data, i. e., not older than one second. The AIM workload suggests seven queries to represent such queries. Listing 6.1 shows these queries.

Their characteristics are:

- filtering and aggregation over many entity records
- unpredictable: random parameters and scheduling
- sometimes joins with small dimension tables

Listing 6.1: RTA-queries of the AIM workload with random parameters

```
--Query 1
SELECT AVG (total_duration_this_week)
FROM AnalyticsMatrix
WHERE number_of_local_calls_this_week > α
```

```
--Query 2
SELECT MAX (most_expensive_call_this_week)
FROM AnalyticsMatrix
WHERE total_number_of_calls_this_week > β
```

```
--Query 3
SELECT SUM (total_cost_this_week) / SUM (total_duration_this_week) as cost ratio
FROM AnalyticsMatrix
GROUP BY number_of_calls_this_week
LIMIT 100;
```

```
--Query 4
SELECT city, AVG(number_of_local_calls_this_week),
    SUM(total_duration_of_local_calls_this_week)
FROM AnalyticsMatrix, RegionInfo
WHERE number_of_local_calls_this_week > γ
  AND total_duration_of_local_calls_this_week > δ
  AND AnalyticsMatrix.zip = RegionInfo.zip
GROUP BY city
```

```
--Query 5
SELECT region,
  SUM (total_cost_of_local_calls_this_week) as local_costs,
  SUM (total_cost_of_long_distance_calls_this_week) as long_distance
FROM AnalyticsMatrix a, SubscriptionType t,
  Category c, RegionInfo r
WHERE t.type = t AND c.category = cat,
  AND a.subscription type = t.id AND a.category = c.id,
  AND a.zip = r.zip
GROUP BY region
```

```
--Query 6
report the entity-ids of the records with the longest call this day and
this week for local and long distance calls for a specific country cty
```

```
--Query 7
SELECT SUM (total_cost_this_week) / SUM (total_duration_this_week)
FROM AnalyticsMatrix
WHERE CellValueType = v
```

7 Implementation of the AIM Workload

In all benchmarks described in Section 5, Flink was superior over the other systems. Flink allows natural streaming by processing tuples-at-a-time instead of batching them. Further, it is possible to maintain an arbitrary state which is a key requirement of the AIM workload. For these reasons, we chose Flink as representative of streaming engines in our further evaluations. We implemented a version of the AIM workload within Flink to compare its performance to database systems.

7.1 Overview

Flink provides many built-in operators that seem suitable for our workload, such as windowed streams supporting event time semantics and aggregation operators such as *Min*, *Max*, and *Sum*. For our workload, we tried to make use of all these pre-implemented functionalities. However, in the current version of Flink, multiple aggregates on a single stream are not supported yet. For this reason, we had to implement a custom aggregation function doing all the aggregations at once.

The aggregations should be maintained on a windowed basis, i. e., daily and weekly. Therefore, we introduced a daily and a weekly tumbling window using Flink's built-in *window-operator*. For only one different type of window this would work fine, however having two or more different window types this will result in multiple output streams. One would have to merge the different streams into one state.

Flink provides a *union-operator* that enables one to union the elements of different streams into one stream. In our case we united the aggregations of the daily and weekly aggregations to send them to the same state. However, this opened up a new pitfall: one has to ensure that the different aggregations for a given event are processed atomically to avoid inconsistent states, i. e., the state is only updated by the daily aggregates of a given event while the weekly ones are still pending.

We ended up by merging the whole window logic into a single custom aggregation function, where we take care of the windows ourselves. This finally gave us a consistent state. The events can be directly processed in the state without the overhead of additional window states and operators. This enables a performant, straightforward updating logic.

Now, having a consistent state, we had to consider how to run the analytic queries on it. Flink does not provide a global accessible state that can be used for such use cases. All states are kept in each instance of an operator and cannot be accessed from outside. We overcame this problem by connecting the event stream with the analytic query stream to process both in the same *CoFlatMap-operator* as shown in Figure 7.1. Both streams are processed using their own *FlatMap*-function, while they share the same state as they are processed in the same operator. The connected stream can be conceptually viewed as a union stream of an *Either* type, that holds either the first stream's type or the second stream's type, i. e., Flink processes both streams interleaved. This ensures consistent views of the state because there are no concurrent state accesses.

Another powerful feature of Flink is its scalability via partitioning. Flink automatically partitions elements of a stream by their key to distribute among to the parallel instances of the single operators. Every parallel instance of our *CoFlatMap* operator only receives the events of its partition and thereby only hold a partition of the total state. The analytic queries, on the other hand, should run on the whole state. Thus, we broadcast them to every parallel instance of the *CoFlatMap* and run

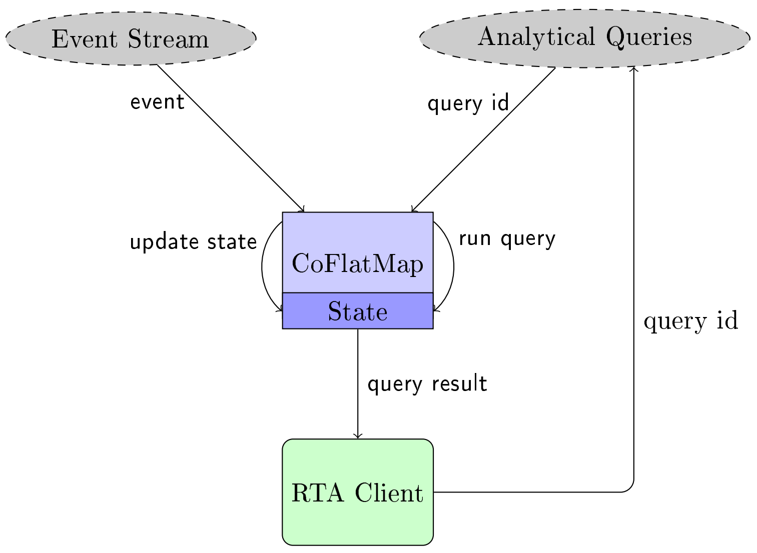


Figure 7.1: Flink interleaves the event processing and the analytic part using a CoFlatMap-operator

them on the respective partition. The resulting partial results are merged in a subsequent operator (not explicitly shown in the provided execution graph).

The *RTA-Client* in this system is implemented via Kafka. Since we use a custom state, all queries have to be hardcoded in Flink.

7.2 Data Layout

Flink does not provide a queryable operator state out-of-the-box. For this reason, we had to implement a suitable state with a handwritten data layout ourselves.

For the AIM workload, the state has to maintain multiple aggregates, that can be grouped as follows:

Window There are two different types of windows: daily and weekly windows. Both are tumbling windows that have to be reset every day or week.

Filter The AIM Workload distinguishes between three types of filters: no filter (1), local filter (2), and non-local filter (3). The local and the non-local filter allow distinguishing between local and non-local calls, whereas the no-filter aggregates provide overall call statistics.

Aggregation types For each call, we want to aggregate over its *duration* and its *cost*. We maintain their minima, maxima and their sums. Additionally, we count the number of calls. The aggregates over the costs, should be treated as numerics with a precision of two decimals, according to the original AIM workload. The remaining aggregates should be stored in 32-bit integers. Java does not support numerics as primitive types. For this reason, we convert the event costs from dollars into cents and store them into 64-bit longs. This gives us the required precision and further enables using primitive data types instead of objects.

Each aggregate in the AIM workload is a combination of a window, a filter and an aggregation type. An example would be the maximum duration of local calls this week. In total, there are 42 different combinations (2 ($Windows$) $\times$ 3 ($Filters$) $\times$ 7 ($AggregationTypes$)). As telecommunicate applications often require up to 500 over 1000 aggregates, the number of described aggregates are duplicated by a factor of 13, leading to 546 overall aggregates [33].

The AIM workload comes with event rates of 10,000 events/s. For each event, all aggregates of the related entity record have to be updated properly. Thus, the updating logic is a hot part of the system and should be designed efficiently. There are different aspects that should be taken into account during implementation.

Storage Layout In general, there are two different storage layouts for database systems: row-store and column store. Both have different strengths: the row store is preferred in OLTP-workloads, while the column store supports efficient data scans and thus is preferred in OLAP workloads. Having a mixed workload of event-updates and analytic queries, we implemented both storage layouts to figure out which one fits best.

Datatypes There are only two different kinds of data types: 32-bit integers, and 64-bit longs. For reasons of cache efficiency and alignment, one should store them as dense as possible. Therefore, we group the different aggregates according to their data types. In our row store implementation, each entity record contains an integer and a long array for the different aggregate types. In the column store, the whole data of an aggregate is stored in a single array, and thus the grouping by data types is implicitly given in this case. We use an array of long and an array of integer arrays to access the different aggregates efficiently.

Windows Our idea is to encode the window information of an aggregate directly into the storage layout. Thus, the window of an aggregate is implicitly given by its position and we do not have to check it separately.

Additionally, we store the aggregates of the same window together and in descending order, i. e., weekly windows before daily windows. Whenever a new window starts, one can take the offset of the window that has to be reset and simply iterate until the end resetting every aggregate. This works due to the implicit ordering of the aggregates.

Imagine a new day starts and all daily aggregates should be reset. If you take the daily offset you can reset all following aggregates without further checking their window type, as it is implicitly given by the ordering. If a new week starts, one would also have to reset the daily aggregates. This approach does this automatically due to the descending ordering. It does not have to check the individual window of an aggregate.

In general, this approach is based on the assumptions that all window sizes stand in direct relation to each other. For instance, a week consists out of 7 days, and a day consists out of 24 hours. Whenever a new window starts, all subsetting windows will also re-start in these cases. If the window sizes are independent from each other, one can also traverse the aggregates of a window efficiently, but has to stop at the end of the window.

Filters When a new event comes in, one always has to update aggregates with no filter condition. Depending on whether it is a local or a non-local call, one has either to update the local aggregates or the non-local ones. For this reason, we group the aggregates by their filters. Thereby, we only have to check whether to use the local or the non-local group of aggregates. This reduces branch misspredictions compared to implementations where the filters of each aggregate are not implicitly given. In these cases, each filter-condition has to be evaluated individually for every aggregate.

Usage of constant relative offsets After grouping the aggregates by their data types, windows and filters the resulting data layout looks like in Table 7.1. This data layout allows us to access the desired aggregates via relative offsets. Each basic aggregate (*min, max, sum*, and *count*) is always stored in the same order and thus their relative offsets are constant. Then, an aggregate can be accessed using the suitable window (*weekly* and *daily*) and the filter offset (*no-filter*, *local-filter*, and *non-local-filter*). For instance, if you want to access the maximum duration of local calls this week, one can get the absolute offset in the dataset by adding all corresponding relative offsets: $WEEKLY\_OFFSET + LOCAL\_FILTER\_OFFSET + MAX\_DURATION$.

WEEK			**DAY**		
No Filter	*Local Filter*	*Non Local Filter*	*No Filter*	...	...
Aggregates	Aggregates	Aggregates	Aggregates	...	...

Table 7.1: Structure of the data in the custom state implementation

7.3 Event Processing

Every time a new event arrives, it is processed in the following three steps:

1. Check windows
2. Update Aggregates
3. Evaluate business rules

7.3.1 Check Windows

First, the event time is taken and it is checked whether a new window has started. Listing 7.1 shows an algorithm that implements this. The idea of the algorithm is to use domain information, such as that a day is a subset of a week. Every time a new week starts, a new day has started, too. For this reason, both can be reset without further checking for a new day. Here, the algorithm makes use of the data layout. Since all aggregates are grouped by their windows in descending order, one can reset all aggregates after the corresponding window offset. Thus, the reset function runs sequentially over the aggregates without any branches, as it is not necessary to check the window of the current aggregate anymore.

Note, that the aggregates of the whole state are reset and not only those of the current entity record. Otherwise, the whole state would be inconsistent, as there would be entity records holding the aggregates of outdated windows.

Listing 7.1: Check for the beginning of a new window and reset the aggregates if necessary

```java
//Holds the timestamp of the last event
private long oldTimestamp;

private void updateWindows(long curTimestamp) {
  long winSize = WEEKLY_SIZE;
  long winStart = oldTimestamp / winSize;
  winStart = winStart * winSize;
  if (curTimestamp > winStart + winSize) {
    for (int i = 0; i < state.length; i++) {
      state[i].resetAggregates(WEEKLY_OFFSET);
    }
    //Can return here because all subsetting windows are already resetted
    return;
  }

  winSize = DAILY_SIZE;
  winStart = oldTimestamp / winSize;
  winStart = winStart * winSize;
  if (curTimestamp > winStart + winSize) {
    //Reset Window
    for (int i = 0; i < state.length; i++) {
      state[i].resetAggregates(DAILY_OFFSET);
    }
  }
}
```

7.3.2 Update Aggregates

Listing 7.2 shows how the updating logic is implemented in the row store version of our implementation. For column store, the only difference is that the update function works on the different attribute columns instead of a row, i. e., an entity record object. This approach provides cache efficiency by updating the aggregates sequentially. The relative offsets facilitate the readability of the code and do not have any impact on performance as they are constant and thus can easily be optimized away by the compiler.

First, the no-filter aggregates of the integer and the cost aggregates are maintained. Then, depending on the local-flag of the event, the local or the non-local filter aggregates are updated. Note, that we need two separate update procedures: one for the integer and one for the long aggregates.

Listing 7.2: Update aggregates using constant relative offsets

```java
public void update(Event event) {
  //convert event costs into cents to achieve a precision of 2
  long cost = Math.round(event.cost * 100);
  updateAggregates(event.duration, NO_FILTER);
  updateAggregates(cost, NO_FILTER);
  if (event.isLongDistance) {
    updateAggregates(event.duration, LOCAL_INT_OFFSET);
    updateAggregates(cost, LOCAL_COST_OFFSET);
  } else {
    updateAggregates(event.duration, NON_LOCAL_INT_OFFSET);
    updateAggregates(cost, NON_LOCAL_COST_OFFSET);
  }
}

//Update integer aggregates of the provided filter
private void updateAggregates(int duration, int filterOffset) {
  for (int i = filterOffset; i < filterOffset + NUM_INT_AGGS_PER_FILTER; i += 4) {
    intAggs[i + DURATION_SUM] += duration;
    intAggs[i + DURATION_MAX] = Math.max(intAggs[i + DURATION_MAX], duration);
    intAggs[i + DURATION_MIN] = Math.min(intAggs[i + DURATION_MIN], duration);
    intAggs[i + CALLS]++;
  }
}

//Update cost aggregates of the provided filter
private void updateAggregates(long cost, int filterOffset) {
  for (int i = filterOffset; i < filterOffset + NUM_LONG_AGGS_PER_FILTER; i += 3) {
    costAggs[i + COST_SUM] += cost;
    costAggs[i + COST_MAX] = Math.max(costAggs[i + DURATION_MAX], cost);
    costAggs[i + COST_MIN] = Math.min(costAggs[i + DURATION_MIN], cost);
  }
}
```

7.3.3 Evaluate Business Rules

In this final step, the system has to consider if any campaigns have to be triggered, i. e., any business rules evaluate to true. A business rule considers the current event and the current state of the related entity record. It can be considered as a boolean formula. We have implemented a version of business rule evaluation by encoding them in conjunctive normal form (CNF).

However, we decided to drop this part of the AIM workload in our benchmarks, as the efficient evaluation of boolean formula is out of the scope of this work. The implementation should mainly serve as proof of concept.

7.4 Analytical Queries

Flink started to support SQL on streams, but it is still a restricted experimental feature, for instance, aggregations on streams are not supported yet. For this reason the analytic part has to be completely handwritten.

We hard-coded all analytic queries into our Flink implementation and made them accessible by calling their corresponding query id. Depending on the data layout of our state, i. e., row or column store, different versions of queries had to be implemented.

Listing 7.3 shows an original SQL query of the AIM workload and how we implemented it in Flink. The handwritten solution can also use of the relative offsets to access the desired aggregates more conveniently. However, the handwritten solution still lacks behind in readability and complexity compared to the original SQL query. Using a column store layout, the complexity of the handwritten queries increases further due to the usage of bitmaps and other vectorized optimizations.

During the query implementation, one has to consider, that Flink scales by partitioning the streams. Each operator state only keeps a partition of the whole state. A query is broadcast to all parallel instances of these operators and is processed intra-parallel by running in parallel on the different partitions of the state. The partial results of all sub-queries have to merged in a subsequent operator. Obviously, merging of handwritten queries is also a handwritten task. For some queries, merging is very straightforward, i. e., find the overall maximum as in query 2. In other cases, it can also involve more complex merging logic, for instance, merging of grouped states. The next section provides a more detailed description of Flink's scalability and intra-parallelism.

Listing 7.3: SQL and handwritten implementation of query 5

```sql
--Query 5
SELECT
  region,
  SUM (total_cost_of_local_calls_this_week) as localCostSum,
  SUM (total_cost_of_long_distance_calls_this_week) as nonLocalCostSum
FROM AnalyticsMatrix a, SubscriptionType t, Category c, RegionInfo r
WHERE t.type = t AND c.category = cat
AND a.subscription type = t.id AND a.category = c.id AND a.zip = r.zip
GROUP BY region;
```

```java
public Object execute(RowStoreAggregates[] state) {
  Map<Integer, RegionStats> groupedRegioStats = new HashMap<>(Parameters.NUM_REGIONS);
  for (RowStoreAggregates entityRecord : state) {
    if (type == entityRecord.type && entityRecord.category == category) {
    RegionStats regionStats = groupedRegioStats.get(entityRecord.region);
    //Create new region stats for region, if region is not already in the HashMap
    if (regionStats == null) {
      regionStats = new RegionStats();
      groupedRegioStats.put(entityRecord.region, regionStats);
    }
    //Update stats for region
    regionStats.localCostSum +=
      entityRecord.costAggs[LOCAL_COST_OFFSET + WEEKLY_OFFSET + COST_SUM];
    regionStats.nonLocalCostSum +=
      entityRecord.costAggs[NON_LOCAL_COST_OFFSET + WEEKLY_OFFSET + COST_SUM];
    }
  }
  return groupedRegioStats;
}
```

7.5 Scalability

A key feature of Flink is its distributed design that facilitates scalability. Flink automatically creates parallel instances of operators for each available core and partitions the streams accordingly. Each instance of an operator processes only a chunk of a stream.

By default, the stream is partitioned by keys of the stream elements using a modulo operation. Thereby, elements of the same key are always routed to the same partition. This scale mechanism works also in distributed settings with multiple nodes.

The event stream is partitioned by entity ids as these are the keys of our analytics matrix. Thereby, all events are routed to the operator that contains the matching partition of the analytics matrix. The RTA-stream must not be partitioned, as the queries should run over the whole state respectively all partitions. Thus, they are broadcast to all parallel instances of the CoFlatMap operator.

Running the queries on partial states only, requires an additional merging operator. This merging operator is implemented using Flink's built-in *Count-Window*. The Count-Window collects all partial results of a specific query id. If the count of partial results for a specific query id equals the number of parallel CoFlatMap instances, one can determine and emit the overall query result. The merging can either be started after all partial results have arrived or whenever a new partial result is done that can be merged in. We decided to chose the latter approach, as micro-benchmarks have shown that it is slightly faster.

Listing 7.4 shows how we coordinate the data flow of the AIM workload using Flink's fluent API. The resulting DAG for a parallelism of three is shown in Figure 7.2. Note, that the count window in front of the merging is not explicitly shown.

Listing 7.4: Coordinating the data flow via Flink's fluent API

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();

DataStream<Event> eventStream =
  env
  .addSource(new EventSource())
  .keyBy("entity_id"); //partition event stream by entity_id

DataStream<Integer> queryStream =
  env
  .addSource(new QuerySource())
  .broadcast(); //broadcast queries to all partitions

DataStream queryResult =
  eventStream
  .connect(queryStream)
  .flatMap(new CoFlatMap())
  .keyBy(0)
  .countWindow(env.getParallelism()) //collect all partial results of the query
  .apply(new MergePartialResults());

queryResult.addSink(new SendQueryResult()); //send query results back to the client
```

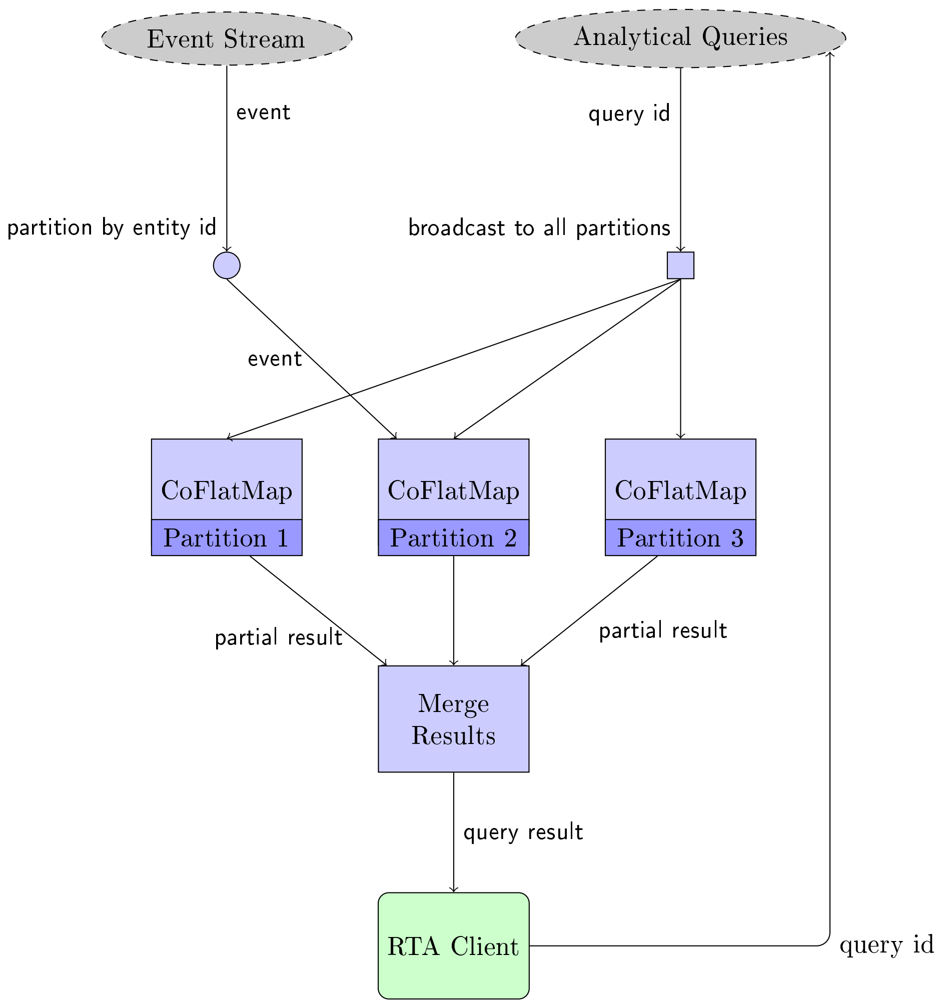


Figure 7.2: Flink's dataflow having three parallel threads

7.6 Event Ingestion

Flink supports various sources of data streams including external and internal event ingestion. The most simple option for our workload is to write an internal source function, that constantly produces new random events. The advantage of this solution is that you do not have any external impacts, such as serialization costs, network communication, or the external data source itself. Further, this approach facilitates debugging before going into production.

If the system is tested, one can easily replace the internal data source with an "real-world", external source. In our case we use Kafka as external source for event generation. This ensures fault-tolerance, as Kafka messages are durable and can be replayed in case of crashes. Kafka sends the messages encoded as byte-arrays. Thus, the events have to be serialized before and deserialized after their transport. One has to specify the serialization in Kafka. For some basic types, such as Strings or Integers, there are already pre-built serializers in Kafka. For custom types, such as the event-class in our case, one has to provide an own serializer.

We tried two different approaches: (1) a handwritten serialization function working on bit-operations only and (2) the serialization framework Kryo [12]. The first approach takes all attributes of an event and stores them byte-wise into the byte-array using native bit-shifting operations. This approach does not have any overhead such as object creation, however it is error-prone and inflexible to use this approach. For this reason, we tried Kryo, a serialization framework. Kryo does the de-/serialization automatically and additionally provides some compression. In section 5.2, we compared both approaches in micro-benchmarks. It turned out, that Kryo is slower than the raw bit-shifting implementation, however both can handle at least 5 millions of events/second. Having only 10,000 event/second, the additional performance boost of the handwritten implementation would be negligible small and thus can be refrained in return for the better readability and maintainability provided by Kryo. For this reason, we chose to use Kryo for serialization.

7.7 RTA-Client

Flink is designed to work and process streams of data. Thus, the analytic queries have to be introduced as a stream, too. Figure 7.1 illustrates this. As all queries are hardcoded into the system, we cannot provide a SQL API. Therefore, we access them via their query id. To match a query to its client, we encode them as *query id + client offset*. Each client offset is unique and a multiple of 7, i. e., the number of queries. Thereby, the desired query can be determined by a simple modulo operation. The result of the query can be sent back to the right client using the client offset.

For instance, a client with id 7 wants to get a result for query 4. It sends 11 (=7+4) to Flink. Our Flink implementation determines via a modulo seven operation that query 4 is required and executes it. By subtracting 4 from 11 it can determine to whom the query belong and send it back to client 7.

In Flink, various data sources would be feasible as RTA-client. First, we used a Flink internal feedback stream to simulate the behavior of blocking queries: every time a query was fully processed, the next query was inserted in a round-robin fashion into the RTA-stream. This was feasible for testing and micro-benchmarking, but it offers almost no flexible querying and is thereby almost not usable in practice. Alternatives are to use sockets, Kafka topics or completely custom-built solutions as data sources for the RTA-stream.

We chose Kafka because it ensures exactly-once message delivery and fault-tolerance out-of-the-box. The communication works via two Kafka topics: RTA-query and RTA-result. The client queries Flink's state by producing the desired query id into RTA-query. Flink reads these query ids as a stream (see Analytical Queries in figure), executes them, and sends the result back to Kafka using the RTA-result topic. The Kafka client waits and blocks until it receives this result.

7.8 Configuration

Flink does not require much configuration. However, there are some points to mention. First, we disabled checkpointing for our benchmarks as Flink only supports to write the whole state to disk during a checkpoint. In the AIM workload, the state holds about 50 GB and thus, the costs of writing it to disk would have a huge impact. In practice, one would set the checkpoint interval to a suitable granularity, e. g., a day, and meanwhile rely on the durability of Kafka. The comparison to the other systems is still fair, as they do not implement or enable durability either.

Another, notable configuration we use is the buffer timeout interval, i. e., the time when Flink flushes its buffers. By default, it is set to 100 ms. This works well for continuous streams of data. However, in our case, we also encode the analytic queries as stream. In contrast to normal streams, our analytic queries have blocking semantics, i. e., no new query is inserted into the stream as long as the old one is not fully processed. In our case, it turned out, that the default flushing behavior of 100 ms throttled the query throughput to about 10 queries per second. By setting the flushing interval of the query stream to zero, Flink flushes its buffer for every new incoming query and does not wait 100 ms. Thereby, we solved this problem. The buffer timeout for the event stream can still be left to its default value.

8 AIM Workload Benchmarks

In this chapter, we evaluate the performance of our Flink implementation by comparing it to main-memory databases, i. e., HyPer and Tell, and AIM – the original C++ implementation of the workload. The AIM workload covers a broad range of streaming characteristics and applications. It is a mixed workload, but one can also measure the update or query performance isolated. Thus, it can be used to unveil individual strengths and weaknesses of the systems.

Note, that the benchmarking of the related systems (HyPer, Tell and AIM) was not part of this thesis. We still include the numbers to put them into relation to Flink's results.

8.1 Systems and Configuration

Table 8.1 shows the versions of the evaluated systems. Like in the previous benchmarks, we run all experiments on an Ubuntu machine with an Intel Xeon E5-2660 v2 CPU and 256 GB DDR3 RAM. The detailed setup is described in Section 5.1.

System	**Version**
HyPer	0.6
Tell	Commit 5bb7b1f
AIM	Same version as used in [33]
Flink	1.1-Snapshot

Table 8.1: Specification of evaluated systems

Throughout most experiments, we measure the scalability of the systems by increasing the number of threads from one to ten.

8.2 Flink Storage Layout

Flink allows implementing arbitrary states. We decided to implement a row and a column store solution.

In a row store, each entity record is stored dense in memory. Thus, it has beneficial cache locality compared to the column store, whenever a whole entity record is updated. If only specific columns of data are read, the column store can read the desired data sequentially and thus is more cache friendly in this case. In this benchmark, we compare both approaches.

We measured the read and write performance isolated to see the individual strength of each approach. Finally, we run the full AIM workload on both to see which approach outweighs the other for our specific workload. Every benchmark was conducted with one to ten threads to show the scalability. Additionally, we measured the performance using all available cores, i. e., using 40 threads, to check if the scalability is sustainable.

Figure 8.1 shows that the row store implementation outperforms the column store in a pure write workload. The row store scales from 450,000 to 2.8 million events per second. The column store is an

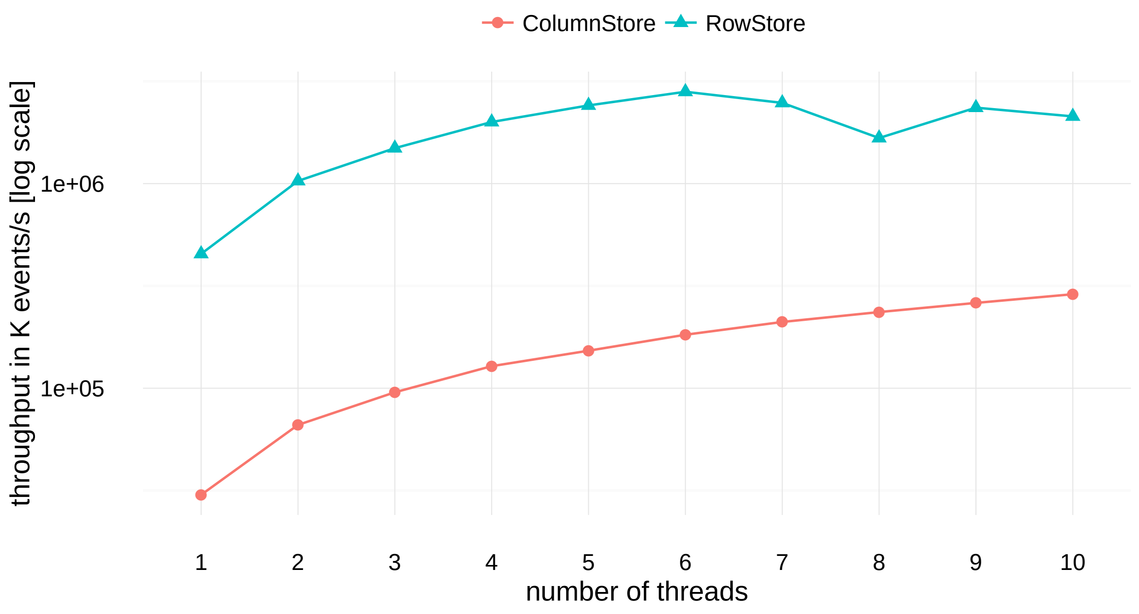


Figure 8.1: Event processing throughput with an increasing number of event processing threads

order of magnitude slower: it achieved 30,000 events per second single-threaded and 292,000 events per second using ten threads. Using all available cores of the machine, i. e., 40 threads, the column store continued its scaling up to 550,000 events per second, whereas the row store implementation did not become faster anymore. After about six threads, the partitioning of the input stream, that is used to distribute the workload among the threads, seems to become a bottleneck in the row store implementation. In the column store implementation we were not able to saturate this bottleneck: However, even using 40 threads, the event throughput of the column store was still slower than those of the row store with only two threads.

The problem of the column store is that an entity record with its 550 fields is distributed among different columns. Thus, there are many cache misses since no sequential memory access is possible during an update.

Every analytic query accesses up to five columns of the whole state. This time the column wise storage pays off for some reasons. On the one hand, the desired columns can be read sequentially and on the other hand further query optimizations such as bit vectors are possible. Thus, the required amount of data that has to be read or loaded into cache is significantly smaller compared to a row store, where every entity record has to be considered.

Figure 8.2 shows the performance differences of these two approaches. The row store scales from 1.7 (one thread) to 13.2 (ten threads), while the column store scales from 13.3 (one thread) to 69.3 queries per second (ten threads). The numbers for 40 threads are 184.3 queries per second for the column store and 24.1 queries per second for the row store. In particular, for the column store the benefit of additional threads waned.

In this benchmark, we have only used 1 client. Thus, there is only one query at-a-time in the system, i. e., no query runs inter-parallel. The queries are only running intra-parallel on the different partitions. This leads to partial results for every partition/thread that have to be merged to an

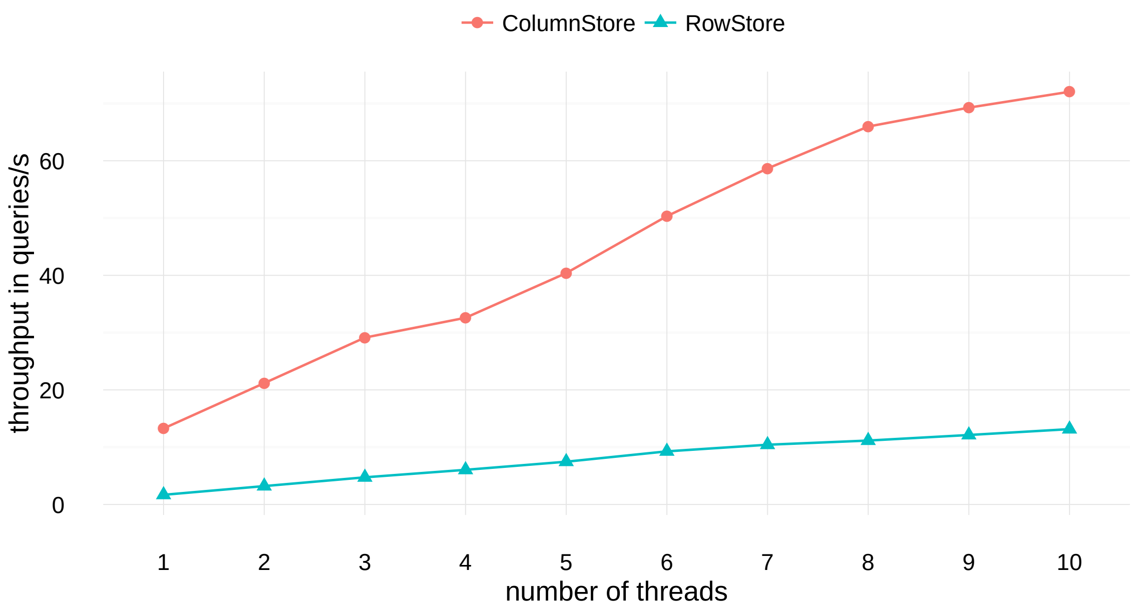


Figure 8.2: Analytical query throughput for 10M subscribers

overall result. The merging happens single-threaded and thus tends to be a bottleneck for multiple threads. For multiple clients, multiple queries can be merged in parallel. The effects of this are shown in a later benchmark, see Section 8.6.

The results of the combined workload are shown in Figure 8.3. We measured the read performance for a fixed number of events per second, i. e., 10,000 events per second.

As expected, the query throughput of both approaches becomes slower compared to the read only benchmark. This makes sense, as updating and reading are simply interleaved in Flink. However, the updating logic of both column and row store is so fast, that the 10,000 events per second required by the AIM workload do not have a huge impact. Using five threads, each core would have to process 2000 events per second. Flink achieves an overall update rate of 150,000 or 2,400,000 events per second for column or row store. Hence, it spends every second only 66 ms respectively 4 ms of the time for updating. The rest of the time can be used to run the queries. This explains why both approaches only suffer slightly from the additional writing.

The row store scales from 1.6 (one thread) to 12.1 (ten threads). Like in the previous read only benchmark, the query throughput increases gracefully until 25.2 queries per second for all cores (40 treads). The column store scales from 7.3 (one thread) to 69.6 (ten threads). For all cores it achieves 177.2 queries per second.

To get an impression of the magnitudes between our row and our column store, we compare it to those of HyPer. HyPer uses column store layout as default but one can manually switch to a row store. For our experiments we measured both storage layouts with all cores, i. e., 40 threads. HyPer achieved 17,700 events per second with its column and 81,700 event/s with its row store. Thus, HyPer's row store was faster by a factor of five. In Flink, the difference for 40 threads is similar, i. e., a factor of four. In the pure analytic part, HyPer achieved 131.5 queries per second using a column and 16.7 queries per second using a row store. This equals a magnitude of eight in favor of

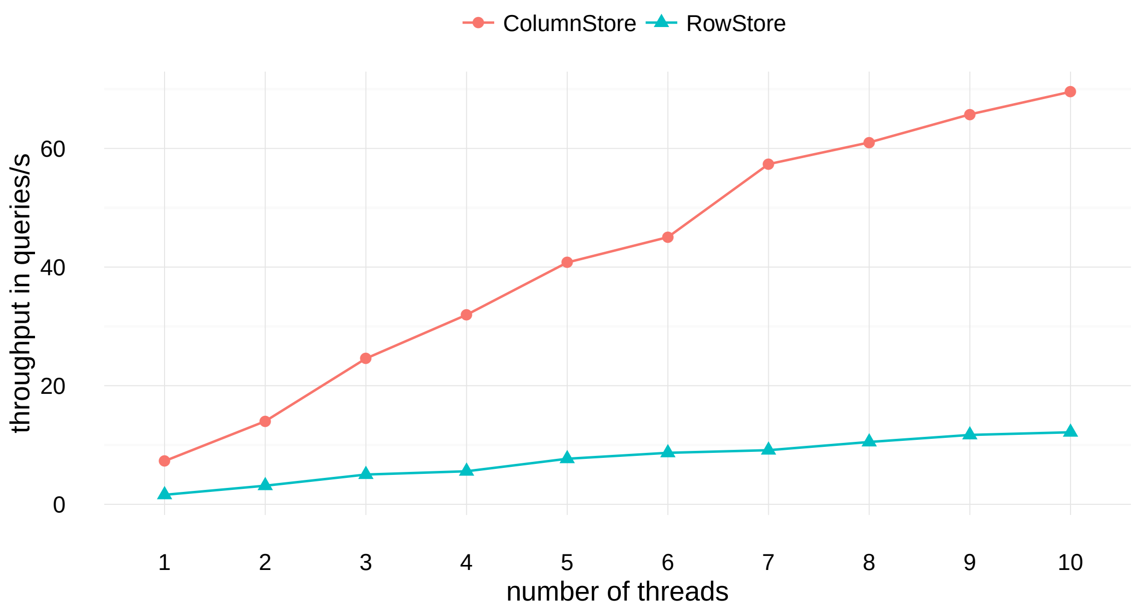


Figure 8.3: Analytical query throughput for 10M subscribers at 10,000 events per second

the column store, which is equal to those of Flink.

Independent of the used system, a column store seems more suitable for read-intense workloads like the AIM workload. For this reason, we use the column store implementation of Flink and HyPer in all further benchmarks.

8.3 Write-Only

In a first step, we compare Flink's write performance to the other systems. We measure the pure event processing throughput for an increasing number of event processing threads without any analytic query. The database systems HyPer and Tell are not able to benefit from multiple threads for writing and thus their throughput did not scale. Tell would only profit from more threads, if there were multiple event generating clients, i. e., SEP clients. AIM is able to scale by partitioning the state similar to Flink. The results are shown in Figure 8.4.

Although, Flink is only using its column store implementation it outperforms the other systems. One can see its almost linear scalability from 30,000 event/s (one thread) to 292,000 events per second (ten threads).

Even though AIM is also partitioning strategy, it falls behind Flink. Its throughput starts at 17,500 events per second (one thread) and scales until 117,000 events per second (nine threads). It does not scale as good as Flink due to problems with over provisioning of threads. AIM's throughput decreases with ten threads, which again shows the problem with over provisioning of threads.

The databases achieved a throughput of 17,700 (HyPer) and 9,000 (Tell) events per second.

Flink is already single-threaded almost two to three times faster than the other systems. This has several reasons.

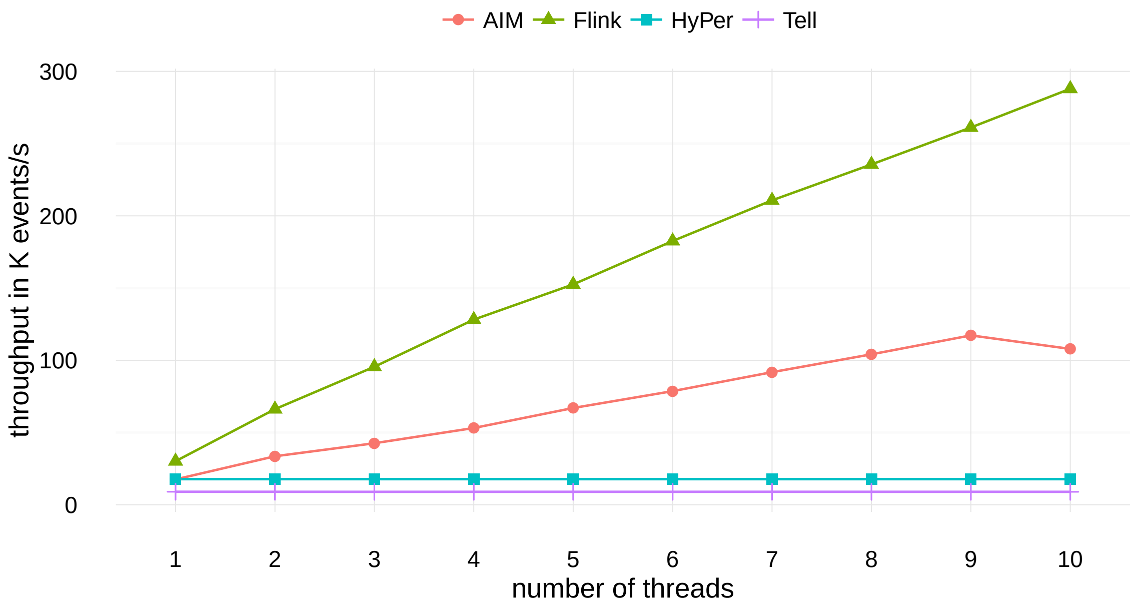


Figure 8.4: Pure event processing throughput with an increasing number of event processing threads

For one thing, it does not come with any writing overhead, such as differential updates, snapshotting mechanism or durability constraints. Flink directly accesses and updates the arrays representing the state. This works as the durability is already provided by Kafka.

For another thing, the updating logic works very efficient as the filtered and windowed aggregates are grouped together. Thereby, we have only one window and one filter branch during the whole update operation. In AIM and Tell the aggregates are not grouped together: they have to check the filter and window condition for every single aggregate. For more details on the Flink implementation see Section 7.3.2.

8.4 Read-Only

Flink is designed as a streaming engine. In this benchmark we want to evaluate how well it performs in a typical database task – processing of queries. Therefore, we compared Flink's pure read performance to native database systems and the handwritten solution AIM. We used the isolated read workload of the AIM workload for this experiment, i. e., the query throughput without any writing. The benchmarks were conducted using only one client to produce the query load. Thereby, we could measure the query throughput without the effects of any inter-query parallelism. We analyze the effects of multiple clients in Section 8.6.

Figure 8.5 shows the query throughput for one up to ten threads. In contrast to the write workload, every system is able to use and benefit from multiple threads now. Note, the numbers for Tell start at two threads as Tell requires at least one thread for its RTA client and one for scanning the data.

Flink was not able to achieve the query performance of a highly optimized database system like HyPer. Initially, both HyPer and Flink are close together: HyPer achieved 19.20 and Flink 13.27 queries per second. However, HyPer scales better for more threads and thus the difference between

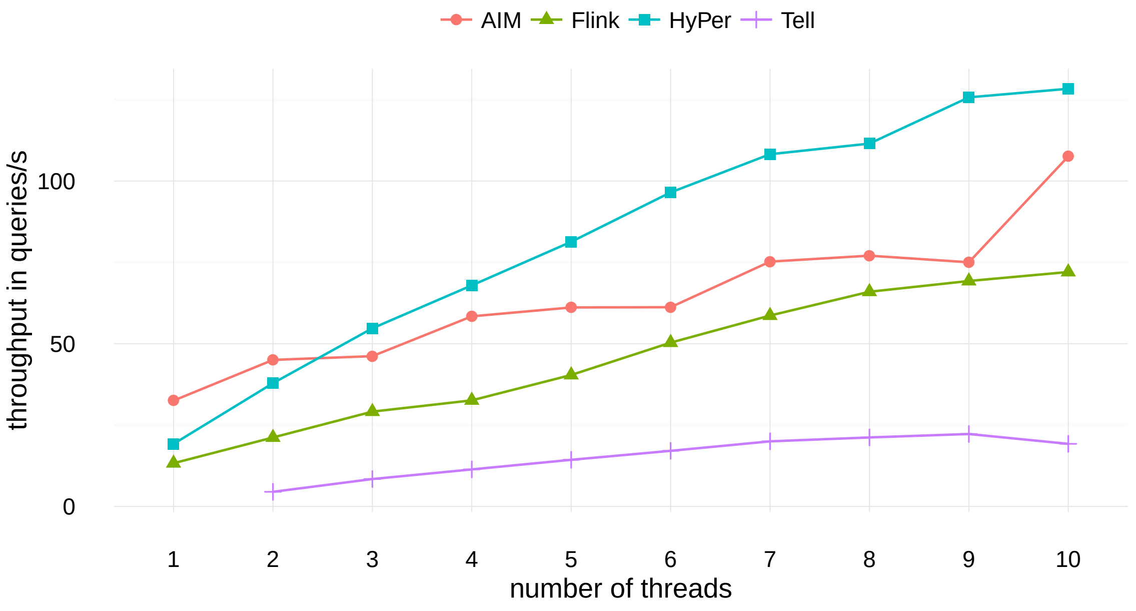


Figure 8.5: Pure analytic query throughput

both systems increases. For ten threads, HyPer achieved 128.35 queries per second and is almost twice as fast as Flink with only 72.04 queries per second.

AIM follows a similar approach to Flink in processing the queries: every thread runs scans on a partition of the data in parallel. In the end, all partial results are merged in the RTA node. Both approaches scale almost with the same speed. However, AIM's single-threaded performance of 32.52 queries per second is superior to those of Flink and since both systems scale almost equally quickly it is able to maintain the lead. For ten threads AIM achieved 107.57 queries per second.

Tell was the slowest in this benchmark. It scaled from 4.48 (two threads) to 22.25 (nine threads), where it drops again to 19.23 queries per second due to provisioning.

8.5 Combined Workload

After measuring the reading and writing isolated, we want to see how the systems handle the full workload. Thereby, we can evaluate how well Flink handles mixed workloads with concurrent state access.

In Flink there is no true concurrency: the updates and the queries are only interleaved by the CoFlatMap operator. HyPer uses MVCC to control concurrent state accesses. However, all 10,000 events are processed as a single batch and block all analytic queries meanwhile. Only AIM and Tell enable true concurrent state accesses using differential updates.

For this experiment, we use the original queries of the AIM workload as described in Section 6.4 and execute them uniformly using a single client per system like in the read-only benchmark. To avoid network impacts, the client is on the same machine as the server, i. e., the system. Note, that AIM and Tell both require at least one ESP thread for the event processing and one RTA thread for the analytic queries. Additionally, Tell needs at least one scanning thread. For these reasons, the

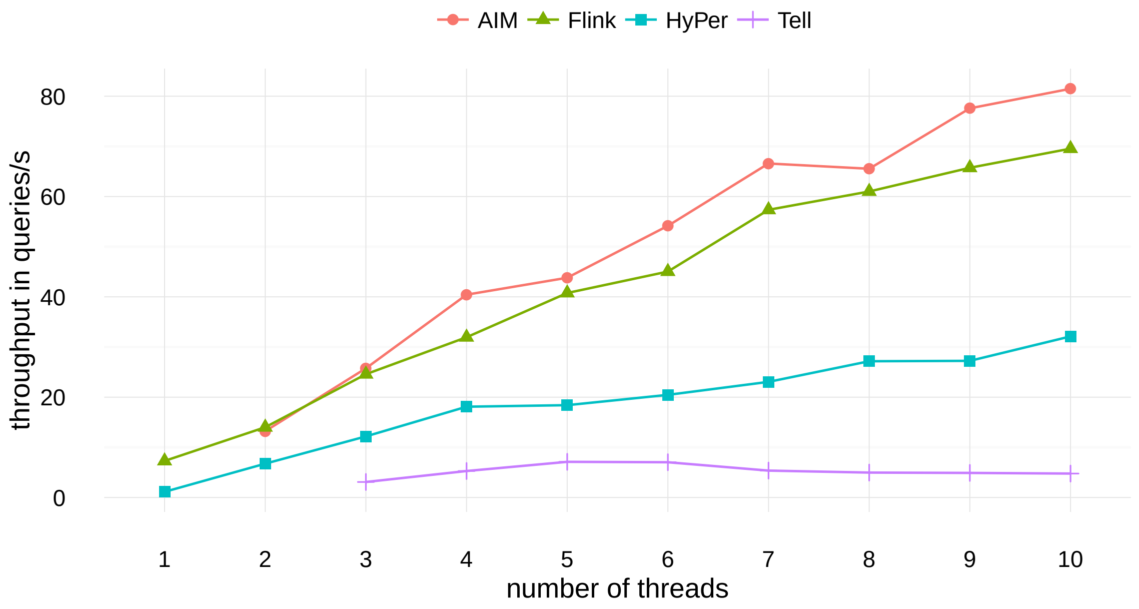


Figure 8.6: Analytical query throughput at 10,000 events per second

measurements of AIM and Tell start with two and three threads, respectively. In contrast to AIM and Tell, Flink: does not use its threads exclusively for updating or reading. Each thread handles both the reading and writing of its partition.

Figure 8.6 shows the query throughput for the full workload at 10,000 events per second. Flink profits in this benchmark from its low slowdown caused by the additional writes. It is almost as fast as in the benchmark before. Flink scales from 7.29 (one thread) to 69.55 (ten threads) queries per second. It is almost able to catch up with AIM, which scales from 13.23 (two threads) to 81.48 queries per second (ten threads).

HyPer suffers most from the additional writes. It slows down to 1.14 (one thread) respectively 32.12 queries per second (ten threads). The main reason for this behavior is the interleaved state access. Every batch of input events takes almost 600 ms, assuming an event processing rate of 17,700 events per second as measured in Section 8.3. Thus, only 400 ms are left for the query processing. Since the event processing rate does not increase with the number of threads, the “usable” amount of time does not increase either.

Flink is also interleaving the processing. However, it has two advantages over HyPer: its write throughput (i) scales for increasing number of threads and (ii) is significantly faster, i. e., the overhead of writing is smaller.

Tell has only a small slowdown due to the additional write load. However, its pure query performance is already too slow compared to the other systems and thus it is not able to catch up in this benchmark. Tell’s highest query throughput is 7.11 queries per second and was achieved using five server-side threads.

We have already noticed, that the slowdown of the systems with additional writes differs among the systems. To get a better impression of this behavior, we directly compare the slow down factors of the systems for a fixed number of threads. We chose to measure the query response times of all

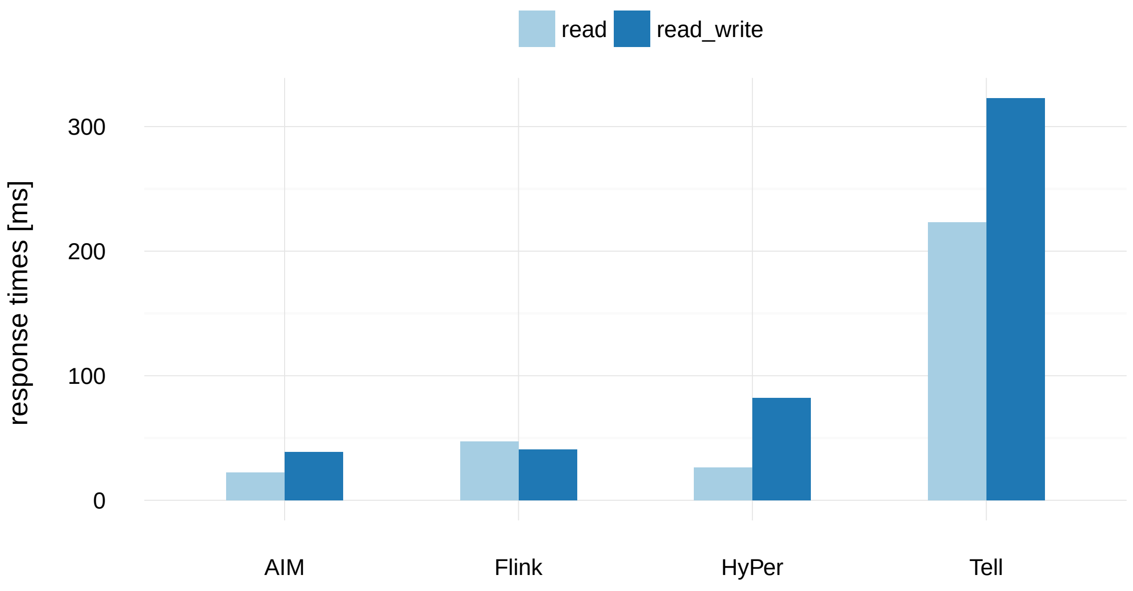


Figure 8.7: Average query response times for read only (2 threads) and combined (3 threads) at 10,000 events per second

systems with two threads for the reading and three threads for the combined workload since these are the minimal numbers of threads in Tell.

Figure 8.7 shows the average response times for queries in each system: the first bar shows the response time for pure reading and the second one those for the combined workload.

The read only query response times are relatively close together for all systems except of Tell. AIM is the best with 22 ms, followed by HyPer with 26 ms, and Flink with 40 ms.

The combined workload unveils more differences. In particular, HyPer suffers from the additional writes as they block the queries. HyPer's query response times almost quadrupled from 26 ms to 82 ms. AIM had also a significant slowdown factor of two. Tell only slowed down by 45 percent (223 ms ↔ 322 ms). Only Flink did not slow down – it became even faster.

The reason for this behavior is, that Flink's write performance is so fast that it is almost negligible at event rates of only 10,000 events per second. The additional third thread used for the combined workload outweighs the increased write load.

8.6 Impact Number of Clients

Flink's architecture enables not only intra-query but also some inter-query parallelism. In this benchmark, we want to evaluate how well it works. In particular, a comparison to AIM and Tell is interesting as both implement inter-query parallelism using shared scans. Another aspect of this benchmark is, that we minimize external effects such as network roundtrip times to the query client. For a sufficient amount of clients, there should always be a query waiting in the pipeline.

Figure 8.8 shows the query throughput for the systems using ten threads and no writes. We increase the number of clients from one to ten.

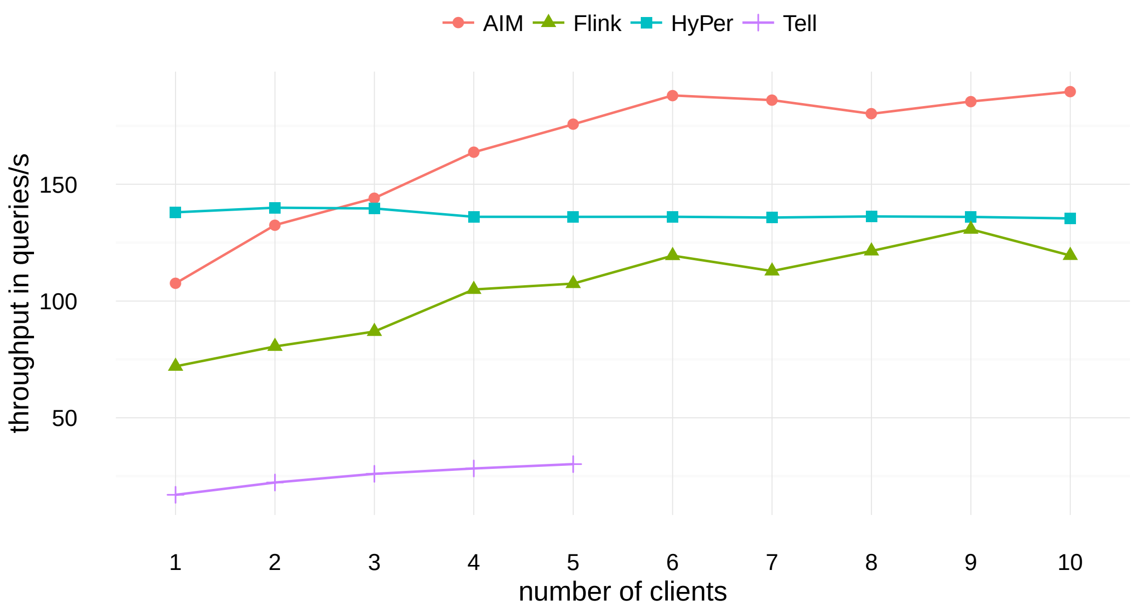


Figure 8.8: Analytical query throughput with an increasing number of clients

Flink can increase its query throughput from 72.04 to 130.72 queries per second. The main reason of the positive impact of multiple clients in Flink is its inter-query parallelism. Until now, the bottleneck of a query was the merging of the partial results as it is performed single threaded. Having multiple queries at the same time in the system, the other threads can already start processing a new query, while the partial results of the old ones are still merged. The benefit of additional clients wanes as it approaches to ten, as there are only ten server-side threads. Thus, the number of idling threads waiting for new queries goes to zero for ten parallel clients, i. e., queries.

HyPer does not come with any inter query parallelism. Thus, the only noticeable change due to multiple clients is the elimination of the roundtrip times to the client. This effect, leads to a constant maximum performance after using at least three clients. However, HyPer's one-query-at-a-time performance, i. e., 136.92 queries per second, is enough to stay in lead of Flink even for ten clients.

AIM and Tell batches the queries from multiple clients together and perform shared scans. This leads to gradually increase of throughput in both systems. For one client, AIM still lacks behind HyPer. But already for 3 clients it takes on the lead. For ten clients, AIM achieves 189.60 queries per second.

Tell repeatedly crashed in our experiments after using more than five clients. Until then it reaches a throughput of 30.17 queries per second.

9 Insights and Learnings

Database systems and streaming engines follow two different architectures. During working with them and evaluating their performance, we encountered their individual strengths and problems. This chapter gives an overview of the sweet spots we have identified and make proposals how they can be transferred to other systems. In general, we make suggestions on how the systems can learn from each other. The goal is to end up with a system combining the best of the two worlds.

9.1 From Streaming Engines to Database Systems

First, we analyze which characteristics of streaming engines would be desirable for DBMS. We have identified three main points that are important for streaming applications but are not implemented in most DBMS. These points include: scalability aspects, fault-tolerance, and missing streaming semantics such as windowing.

9.1.1 Scalability

Scalability is an important aspect in streaming applications, as the workloads tend to be very large and demanding. For one thing, a system should scale within a node in terms of threads, and for another thing, it should be able to scale across of a cluster of nodes. Further, it should be able to scale both the data processing (OLTP) and the data analytic part (OLAP). All tested systems in Section 8 are able to scale in the OLAP part, but only Flink and AIM scaled also in the OLTP part. In particular, the scalability of the streaming engine Flink in the writing benchmark, see Section 8.3, is outstanding. Flink would be able to deal with much higher event rates, while the tested DBMS are bound to their single-core throughput. Most streaming engines support this kind of scalability naturally as they are designed as distributed systems from scratch. DBMS in the contrary are in general designed for the usage on single machines.

The advantage of streaming engines is that their whole design features data flow. The data flow is organized by a DAG of operators sending the data from one operator to the next operator. Each operator does some sort of data processing such as flattening the input data, filtering, or aggregation. The result of an operator is sent to the subsequent operator or a data sink. Each pair of subsequent operators is connected by buffers, that can either be local or network buffers allowing a distributed setup.

This design facilitates scalability. A streaming engine can simply scale by distributing the different operators across the cores or nodes. For instance, the first node does the flattening of the data, the second filters the flattened data, and the last one aggregates it. Additionally, it is possible to have multiple parallel instances of a single operator. This can be done by partitioning the stream into chunks. Each parallel instance of an operator receives and processes only chunks of the data.

There are different ways to distribute data among the parallel instances of an operator. For stateful operations like aggregations on entity records, one has to partition the data by the entity id. In other cases, it is also possible to shuffle the data randomly to achieve a uniform load across the nodes.

There exist also approaches to scale out DBMS, e. g., ScyPer [52] – a distributed version of HyPer. ScyPer replicates the whole data across its nodes by multicasting the redo log entries. ScyPer is based on the assumption, that the OLTP throughput of a single machine is sufficiently fast and only the OLAP throughput has to be improved. In streaming workloads, this assumption does not necessarily hold any more. For instance, HyPer would fully saturate its throughput limit in the AIM workload, if the event rate would increase from 10,000 to 17,000.

To overcome this problem, one could implement parallel writes into HyPer. Therefore, one could partition the state in chunks and assign individual worker threads to them that write in parallel. We showed the great scalability of this approach during our write-only benchmark.

If the throughput or the available memory of a single machine is saturated, one could also scale across multiple nodes. Therefore, one could extend ScyPer's architecture by introducing additional event processing nodes. Each event processing node would maintain and write to a partition of the state. To distribute the data according to the partitions, one has to add a distribution layer on top of the architecture. The query layer would also have to be extended to coordinate the access to the different partitions, i. e., retrieve and merge partial results. Note, that each node can further partition its part of the state and use multiple write threads like in the single node case.

Streaming engines like Flink facilitate changing between local and non-local work distribution. Flink uses buffers as abstraction to communicate between operators. A buffer can either be implemented as a local or as a network buffer – the API stays the same in both cases.

9.1.2 Fault-Tolerance in Streaming

DBMS implement efficient fault-tolerance strategies using WAL. In case of failures, the log entries can be replayed to restore the consistent state before the crash. This works fine for traditional relational applications, however, for continuous input data it is problematic. It cannot ensure that the elements that arrive during the downtime of the system are processed. They are simply missed.

Streaming engines face this problem by using a durable data source such as Kafka. All messages, in particular the messages missed during downtime are preserved by the durable data source. If there is a crash, the latest state is restored and the processing can be resumed from the latest message checkpoint.

There are already databases that implement a Kafka integration. PipelineDB, for instance, makes use of various database features to integrate Kafka. They use PostgreSQL's copy infrastructure to transform Kafka messages into database-compliant rows. Kafka parameters such as the latest message offsets are stored and maintained durably in a relational database table. In case of a restart, e. g., due to a failure, PipelineDB is able to resume consuming from the last stop point. The latest stop point can be recovered during the default database recovery process as it is stored in a normal database table.

9.1.3 Native Window Support

In contrary to the finite data sets in relational databases, streaming data is continuous and unbounded. To overcome this, one can group streams into windows. The most important types of windows are tumbling and sliding windows. See Section 2.2 for a detailed description of windowing.

Streaming engines introduce window semantics using window operators. In particular, Flink offers an extensive set of window operations that allows designing complex windows.

Databases, i. e., standard SQL, do lack this functionality. There is a window keyword in SQL, but it does not correlate to streaming windows. It only defines ranges in queries on historical data [48]. For this reason, we had to implement the windowed aggregates of the AIM workload with stored procedures in HyPer.

There exist some historic approaches to add such window semantics to SQL. Prominent representatives are StreamSQL and CQL, see Section 2.5. Both languages allow defining windowed queries on streams that produce continuous outputs. However, each query produces another output stream and not a queryable state. Without such a queryable state, it is not possible to run ad-hoc SQL queries on the windows, like it is for instance required in the AIM workload.

A more suitable SQL API for stateful, analytic workloads is implemented by PipelineDB. It extends the PostgreSQL syntax by streams and continuous views. Continuous views can be considered as materialized views that are continuously and incrementally updated over time. Streams can be fed with SQL into-statements similar to relational tables.

Each record is merged into the continuous view, whereas PipelineDB tries to persist as less data as possible. In the optimal case, PipelineDB only has to persist the values of the continuous view and can discard the records immediately after processing them. This can be done if the aggregates are rolling, i. e., there are no windows, or when the windows are tumbling. These windows are non-overlapping and thus every record can be discarded after the view is updated with its information. If the windows are sliding, PipelineDB cannot directly discard a record as its information is used in multiple overlapping windows. PipelineDB has to maintain the information of the record as long as it falls out of all windows. However, PipelineDB pre-aggregates the records as far as possible to reduce the required amount of memory. For sliding windows this means, that the pre-aggregations have a granularity equal to the sliding range of the window.

Continuous views support windows and different time semantics. An example continuous view showing the number of calls per user throughout the last 24 hours would look as follows:

Listing 9.1: Continuous view in PipelineDB over the last 24 hours

```sql
CREATE CONTINUOUS VIEW calls WITH (max_age = '24 hours') AS
SELECT entity_id::integer, COUNT(*) FROM event_stream
GROUP BY entity_id::integer;
```

Listing 9.2: Advanced window constructs in PipelineDB

```sql
CREATE CONTINUOUS VIEW calls AS
SELECT entity_id::integer, COUNT(*) FROM event_stream
WHERE (arrival_timestamp > clock_timestamp() - interval '24 hours');
```

The above query (Listing 9.1) is syntactic sugar for the advanced representation shown in Listing 9.2. PipelineDB receives the current timestamp using PostgreSQL's *clock_timestamp*() and *interval* function, and compares it to the *arrival_timestamp* of the call. If the call itself comes with a timestamp field, one could also extract it and use it instead of PipelineDB's *arrival_timestamp*, i. e., use event time semantics.

Another useful feature of PipelineDB's SQL API is to define multiple windows on the same continuous view. This can be done by defining additional views for each window on top of the continuous view. Listing 9.3 shows a continuous view that counts the number of calls of the last day and the last week.

Listing 9.3: Multiple windows on own stream

```sql
CREATE CONTINUOUS VIEW calls AS SELECT COUNT(*) FROM event_stream;
CREATE VIEW daily WITH (max_age = '1 day') AS SELECT * FROM calls;
CREATE VIEW weekly WITH (max_age = '7 day') AS SELECT * FROM calls;
```

Multiple windows: not possible in Flink create regular views on top of the continuous view: internally only a single continuous view is updated; saves resources compared to updating a continuous view for every window and facilitates the querying and consistency since all queries can run on only one view

Internally, PipelineDB only updates a single materialized view for all windows to save resources. Within a view, PipelineDB tries to pre-aggregate the inputs as much as possible. When the window is evaluated, all pre-aggregates of a window are combined and outdated pre-aggregates are discarded from the view. The challenge is to choose the granularity of the pre-aggregations as high as it is possible without loosing information. Thus, the maximum granularity is the smallest sliding step range of all windows.

PipelineDB's manages to introduce window support into a SQL database with only slightly extending the standard SQL syntax. Thus, it almost feels like using pure SQL. However, it still has very powerful semantics. For instance, it allows multiple windows on one stream. In many dedicated streaming engines this is not, or only with additional synchronization possible.

9.1.4 Processing Guarantees

Databases make strong processing guarantees using transactions, i. e., exactly-once semantics. However, transactions come with an overhead and thus throttle the write throughput. Not all streaming applications require this kind of guarantee but instead come with a high demand on throughput or low latency. Imagine for instance the scenario of real time traffic data analysis. On the one hand, there is a high data rate produced by millions of vehicles, but on the other hand it can be enough to identify a traffic congestion without processing every message exactly-once. In these cases it would be good to be able to soften the processing guarantees, e. g., use only at-least-once semantics. In cases when the data rate is too high to be processed, one could also perform load shedding and switch to at-most-once semantics.

Most streaming engines allow doing this by turning off checkpointing or acknowledgments, while relational databases stick to their ACID criteria. ACID makes strong constraints on atomicity, consistency, isolation and durability. Some applications do not require all of these constraints but come with high demands on throughput. For this reason, NoSQL databases were created that give up some ACID constraints to achieve higher performance and scalability [36]. Most of them scale at the expense of consistency, i. e., they provide only eventual instead of strict consistency. Eventual consistency means, that the data of some nodes might be outdated, but updates are guaranteed to be propagated to all nodes eventually [36]. This makes data replication across multiple nodes easier and faster.

Other systems, e. g., STREAM, explicitly use load shedding, i. e., the drop of data, to handle high throughput [59]. This leads to at-most-once semantics.

In general, databases cannot be configured to the individual processing demands of an application. If one could switch between processing guarantees like in streaming engines, one could cover a broader set of applications with one system.

9.2 From Database Systems to Streaming Engines

Our experiments showed that DBMS are in particular superior in analytic workloads. We want to analyze the reasons and discuss how streaming engines can improve in this part. Further, we discuss how streaming engines can provide fault-tolerance more efficiently.

9.2.1 SQL Query API

The core feature of databases is their accessibility via SQL. This has several benefits compared to a Java API used in streaming engines.

SQL is a well-known language that facilitates fast development compared to the low-level open source platforms. Since it is only declarative, it enables the system to do powerful optimizations. The most fundamental advantage of a SQL API is the support arbitrary ad-hoc queries — one do not have to hard-code all required queries into the system.

There are many trends to add a SQL API to streaming systems, such as SparkSQL, Flink's Table API, or SQLstream for Storm. However, they are often very restricted. For instance, Flink's API only works on a fixed dataset and streams are not yet supported.[1] The APIs often do only cover parts of the SQL semantics and are not backed up by query optimization engines as powerful as in native databases. Finally, they are often not able to solve the problem of ad-hoc queries. The queries can be written in SQL but they are only a hardcoded part of the existing Java API, i. e., as string, and thus do not allow ad-hoc changes.

Our custom implementation of the AIM workload in Flink allows ad-hoc queries by ingestion the desired query id via an additional stream. However, the queries are still all hardcoded in the system. One would have to add a SQL parser and SQL operators to be able to run flexible SQL queries. This could be done, however, it would lead to a simplified database within the streaming engine. Obviously, this would go beyond the scope of our AIM workload, but it could be taken as a proof of concept.

9.2.2 Insights of Query Processing

Databases are highly specialized on processing of queries. They implement various techniques to increase the efficiency of queries.

The insights could or at least should also be used in streaming engines. In particular, in analytic workloads like the AIM workload, it would be suitable to rely on database technologies. However, since database-like states are not supported in Flink, we had to implement our own database like state and the required queries. It would have been much more convenient and robust, if there was a ready-to-use database solution doing all the state management, queries and optimizations itself. Obviously, this would be a big change for streaming engines, as they would have to implement almost a full database inside of them. A proper integration of an existing database seems more suitable.

There are other database features, that are more easily applicable to streaming engines. Modern main-memory database systems, for instance, compile queries into efficient machine code [53]. Thereby, they avoid expensive function calls and are able to generate data-centric and branch-optimized code.

Streaming engines could also make use of code generation by translating their whole DAG of operators into efficient machine code. The effort for the code generation and compilation would pay off more than it does for database queries, as streaming engine workloads are typically long running and not ad-hoc as queries. Once the code is compiled, the code can be used for every input record. Flink goes into this direction by compiling its query plans into DAGs. These DAGs feature efficient data pipelines and are can be just-in time compiled by the JVM.

A key feature of code generation is, that it can optimize for data locality and efficient branches. In windowed aggregation workloads like the AIM workload, one could use code generation to automatically generate an efficient update function and data storage. One would only have to specify the window, the filter and the aggregation type and the code generator could automatically group the aggregates efficiently. Section 7.3.2 shows a possible grouping and storage of windowed aggregates

[1] `https://flink.apache.org/news/2015/12/18/a-year-in-review.html` (04/21/2016)

that reduces the update function to one branch per event for all aggregates. With code generation, one could automatically generate such a grouping for the required aggregates.

This concept of operator grouping could also be used by the databases itself by extending their code generation capabilities.

9.2.3 Adaptive Operator Ordering

Another powerful optimization of query engines is the ordering of joins and operators. In databases, this is done by the query optimizer. Streaming engines often lack such an optimizer or stick to the ordering given by the user. Flink states in its internal developer documentation, that operator reordering would be a high opportunity optimization, but they do not implement it yet because it comes "with high risk in the absence of good estimates about the data characteristics".[2]

This problem can be overcome by monitoring the streaming characteristics. A suitable approach would be to sample the data in order to determine the selectivities and data rates of the individual operators and streams. This approach can be used for both static relational data and streaming data [30]. It is in general superior to estimation based solutions [47].

Monitoring of data in real-time enables the streaming engine to re-order its operators adaptively. An adaptive ordering would be in particular powerful for streaming applications with changing data characteristics. However, the monitoring and re-ordering of data and operators adds an overhead to the system. This overhead can be controlled, by choosing the frequency of monitoring and a threshold for re-ordering. For instance, if one expects non-changing data characteristics, it would be enough to perform the data operator order optimization only in the beginning.

9.2.4 Efficient Fault-Tolerance

Fault-tolerance is an important aspect in streaming applications and should be implemented efficiently. On the one hand, the backup mechanisms should not have a bad impact on the stream processing. And on the other hand, the recovery of the state after a failure should happen fast. In all cases, the message guarantee should be fulfilled. For instance, process every message exactly-once.

Flink implements an efficient fault-tolerance mechanism, i. e., distributed lightweight snapshots. It is very lightweight compared to the micro-batching used in Spark Streaming and Storm Trident. A key feature is that stream processing does not have to be paused doing a checkpoint. For a more detailed description see Section 2.4.

Flink's checkpointing algorithm back ups the whole state of each operator to a durable storage such as discs or HDFS. This works well for small operator states, but leads to problems with large states as writing the whole state to disk can be very expensive. In practice, companies try to avoid this problem by simply increasing the checkpoint interval to days instead of seconds. This relies on replay capability of the data source and the assumption that crashes are rare. However, if there is a crash, many messages have to be replayed from the data source to restore the state. This becomes problematic, when the checkpoint interval and/or the data rate is so high that the streaming engine cannot catch up in a sufficient time.

An alternative would be to use delta checkpointing like it was done in 2005 in the Aurora system [41]. The idea is to back up only the changes to the state, i. e., the delta, on each checkpoint. This can reduce the write-load dramatically and thus make it possible to use shorter checkpoint intervals. Aurora recognized changed data by marking it with dirty bits during a change. For a more detailed description see Section 4.3.

[2] `https://cwiki.apache.org/confluence/display/FLINK/Optimizer+Internals` (04/21/2016)

In the AIM workload one could mark every entity record as dirty after it was changed. If multiple changes happen during a single checkpoint interval, only the final version of the entity record has to be considered. This would be an advantage compared to the logical redo-logs of database systems, where intermediate step are stored and thus all log entries of each entity record have to be redone.

9.3 Design Proposal

Summed up, the results of the experiments in 8 show that: streaming engines dominate in OLTP and database engines in OLAP heavy workloads. However, streaming engines are able to catch up in OLAP if they implement efficient queries and states. In theory, a streaming engine should reach the same level of query throughput as a database, if it implements the same level of query optimizations as the database. In other words, one would end up with a "little" database inside the streaming engine.

Implementing a whole database is a non-trivial task. So let us revisit the reasons that makes the OLTP part of a streaming engine fast compared to databases. A key difference is, that they are able to weaken their consistency and their durability. In applications, where data loss is acceptable, streaming engines can turn off durability completely to achieve better throughput. In other cases, they mainly "outsource" the task of durability to the data source itself. For instance, Kafka is a data source that keeps the messages durable and allows the streaming engines to replay it when they need it.

The second approach, however, ensures durability and at the same time has a great benefit compared to databases' write-ahead logging. When the data source itself handles the durability, the system can go down without missing any data. Whereas, databases that do not rely or integrate durable data sources would simply miss the data. Further, one gets rid of the overhead of writing the log synchronously to disc. This leads to better throughput and latency at the same time.

Database could implement a similar fault-tolerance like some streaming engines, i. e., delta checkpointing. We propose to combine Flink's snapshot algorithm, STREAM's delta checkpointing and PipelineDB's Kafka integration into a fault-tolerance mechanism applicable for a database.

The approach relies on the replay capability and durability of Kafka. The database reads and processes the messages without any additional logging. In case of a failure, the state can be restored by replaying the messages from Kafka.

Over time, this can lead to a high amount of data that has to be replayed. For this reason, the database should take snapshots of the state from time to time and persist it to a durable storage. After a checkpoint is performed, the database has to store the latest Kafka message offset to mark which messages are part of the snapshot. Thereby, it knows where it has to resume after a recovery. PipelineDB manages the message offsets and other Kafka parameters in an own SQL table.

This approach works well for small states. However, for big states, the cost of writing the whole state to disk during each checkpoints are too high. We identified this problem using Flink in the AIM workload. As the costs of writing the whole analytics matrix, i. e., about 50 GB, to disk where to high, we had to disable Flink's checkpointing completely. STREAM goes a more efficient way by backing up only the changed part of the state. It identifies the changed data by marking changed data records with dirty bits.

For databases, we think it would be more suitable to maintain a set of TIDs that identifies the changed records. Thus, we do not have to scan all records for dirty bits, but can iterate only over the changed tuples using the stored TIDs. Another possibility would be to change the behavior of the already existing redo log. Instead of forcing it to disc, one could keep it in-memory until the next checkpoint.

It is sufficient to write out only the latest state of an entity record. For this reason, one can compactify the contents of the redo log by discarding all previous versions, i. e., log entries, of an entity record. One has to evaluate, if it is better to compactify the log periodically or only before writing it to disk.

In windowed workloads such as the AIM workload, we can further improve the performance by taking window semantics into account. For instance, the start of a new week resets all aggregates of the AIM workload. The events of the previous week and the state becomes outdated and thus do not have to be considered any more. The delta state can be discarded and the Kafka offset can be set to the first message of the new week.

The restoring itself can be very efficient. Instead of redoing the whole log, it could simply load the whole data into memory. Afterwards the data processing is resumed from the latest checkpoint in Kafka.

Even though, we only want to back up the delta, every checkpoint still comes with an overhead. In Flink this checkpointing overhead occurs in a fixed checkpointing interval and thus does not allow much flexibility. We want to make this overhead controllable by making the checkpointing trigger configurable. The user should be able to choose the trade off between fast recovery and throughput itself. We want to provide following checkpointing semantics:

Tuple-at-a-time The highest possible checkpoint semantics would be to write out the delta of the state after each update. This allows fast recovery from faults, as almost no tuples have to be replayed. However, the high frequency of checkpoints would throttle the throughput. This approach has the same frequency as WAL, but it should have a better latency than WAL as it does not have to write synchronously, i. e., ahead of the update. If the system crashes before the checkpoint is written, it can be recovered by replaying the lost message(s) from Kafka.

Periodic Checkpointing Flink uses periodically checkpointing over time. This approach probably works best for almost constant data rates. If the data rates are varying, some checkpoints would be significantly more expensive than others. We think, a checkpoint that is additionally based on the count of changed tuple would be more suitable in such cases. If one could define a minimum threshold of changed tuples, one could avoid the overhead of checkpointing during times, where almost no change occurred. Thus, we would suggest adding an optional minimum threshold to the periodic checkpointing. In cases, where fast recovery is desired or the storage capabilities of the data source are limited, one might also want to add a maximum count threshold.

Adaptive Checkpointing The idea is, to perform checkpoints in situations where the system's load allows it, i. e., the overhead does not hurt. STREAM does this by monitoring the load of the system. It tries to perform its checkpointing whenever the current load is low. This approach should be combined with some kind of time scheduling to control the frequency of checkpoints. Otherwise, the system might not make a checkpoint at all during times of high load or would make an unnecessary number of checkpoints just because the current load is low. For this reason, one should be able define a lower and an upper time interval for checkpointing. For instance do a checkpoint at least every hour, but at most every 10 minutes.

No Checkpointing Finally, one could also turn off the checkpointing completely. This allows for very high throughput in applications that do not require any fault-tolerance.

The power of this checkpointing approach is, that it is able to cover a broad set of applications by simply changing the triggers. The checkpointing logic itself is not affected by the trigger.

We think, that this approach would enable higher data throughputs, competitive to those of native streaming engines. Further, it can ensure that no data is lost anymore during the downtime of the system.

10 Future Work

In future work, it would be beneficial to address several points.

We plan to evaluate PipelineDB in more details. Its approach, in particular its API sounds promising. We think that the usage of continuous views would be a feasible approach, as it adds powerful streaming semantics by extending the SQL syntax only slightly. We want to evaluate whether this approach is adaptable to HyPer as both are based on PostgreSQL semantics. Further, we can use PipelineDB in future work to compare its performance to own streaming extensions to HyPer.

As a first step, we plan to integrate event ingestion via Kafka into HyPer. This can be used to implement the fault-tolerance mechanism and durability guarantees as described in Section 9.3. The outsourcing or weakening of durability requirements would enable HyPer to deal with applications that come at higher data rates.

Our idea is to make the level of fault-tolerance configurable. Thereby, one could cover a broad set of applications with a single, ready-to-use database system. In particular, the last aspect would distinguish HyPer from other streaming engines, whose APIs require much initial programming effort and do not directly support ad-hoc queries or changes.

Another important aspect would be to integrate windowing and aggregations on streams into HyPer. Until now, the only way to support such streaming application is to add the desired logic with stored procedures and triggers. The whole update logic of the AIM workload is implemented this way. Obviously, handwriting the whole update logic is not as conveniently as using a SQL API, but we experienced also other drawbacks of this approach. During initial experiments with a smaller number of aggregates, i. e., only 42 instead of 546[1], we experienced that HyPer performs significantly better. The write throughput increased by a factor of 144, while the number of written aggregates only decreased by factor 13. Just for comparison – Flink's throughput only increased by a factor of 25. Thus, HyPer's throughput boost cannot only be explained by the decreased number of random write accesses. We guess, that the main reason lies in the stored procedures. For every aggregate, we have a separate update statement. Thereby, the produced code fragment that has to be loaded by HyPer for every batch of events gets relatively long. For 546 aggregates, the loading of the code becomes particularly expensive as its size exceeds the L1 cache. This issue does not occur in Flink since the whole updating logic is implemented using tight loops. The code only differs in the number of loop iterations. A more detailed description of Flink's update logic can be found in Section 7.3.2. The advantage of using loops is, that the resulting code is very tight and thus can be loaded and processed more efficiently. Loops are possible in this case, as the same aggregates are used with different combinations of windows and filters. We think, that the write throughput of HyPer could be increased by implementing a similar tight updating logic.

Finally, it would also be interesting to evaluate not only the single machine but also the distributed performance of streaming systems. The results could be used to extend ScyPer's distributed design based on the demands of streaming applications.

[1]We received this number by removing all duplicate aggregates

11 Conclusion

It turned out that Flink's tuple-at-a-time approach in combination with its lightweight fault-tolerance mechanism is superior to the micro-batching solutions. It achieved the best performance of the tested streaming engines. Flink's design has several benefits over the other systems. Its tuple-at-a-time approach enables fine granular windowing on streams, whereas the micro-batching systems are restricted to the granularity of their batch intervals. Further, Flink handles peaks in data rates naturally, while the micro-batching systems come in trouble, when the time for processing a batch of tuples exceeds the provided batch processing interval.

Flink also proofed its strengths in the AIM workload. In contrast to the databases, it shows outstanding write throughput and scales for both OLTP and OLAP. Flink's key benefits are its automatic partitioning of streams that allows parallel processing, i. e., reads and writes. Further, it has lower overhead due to durability requirements as it can rely on the durability of the data source, e. g., Kafka.

However, Flink has some issues with analytic workloads. We had to implement a custom data storage to maintain the analytics matrix of the AIM workload. All analytic queries are also handwritten and hard-coded into the system as there is no SQL API. Thus, Flink does not support ad-hoc queries. In the contrary, databases perform very well in analytic workloads. In particular, HyPer shows an outstanding query performance. HyPer benefits from query compilation and micro optimizations such as vectorization using SIMD instructions. Tell and AIM implement additional optimizations for multiple clients, i. e., shared scans. Integrating all these optimizations in a custom data storage is probably out of scope for many applications. Further, it still does not provide the capability of ad-hoc queries, as this would require a SQL parser. One would end up with a “little database” inside the streaming engine.

We think, it would be easier to rely on the query processing capabilities of state-of-the-art main-memory databases and improve their streaming capabilities. Therefore, one has to extend SQL with streaming semantics, in particular windowing. Additionally, one has to optimize the databases for the high event rates of streaming applications. This could be done by lowering the durability guarantees according to the needs of the application. For example, a traffic analyze application has to deal with very high rates of car data in real-time, while exactly-once processing or durability are secondary requirements. If durability and fault-tolerance are important, one could implement a kind of the fault-tolerance mechanism that makes use of the durability of the data source, as we proposed in Section 9.3.

Bibliography

[1] Apache Flink. http://flink.apache.org/.

[2] Apache Kafka. http://kafka.apache.org/.

[3] Apache Samza. http://samza.apache.org/.

[4] Apache Spark Streaming. http://spark.apache.org/streaming/.

[5] Apache Storm. http://storm.apache.org/.

[6] Apama. http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/.

[7] Esper. http://www.espertech.com/products/esper.php.

[8] Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. http://www.gartner.com/newsroom/id/3165317.

[9] Google Cloud Dataflow. https://cloud.google.com/dataflow/.

[10] Hadoop. http://hadoop.apache.org/.

[11] JMH. http://openjdk.java.net/projects/code-tools/jmh/.

[12] Kryo. https://github.com/EsotericSoftware/kryo.

[13] MemSQL. http://www.memsql.com/.

[14] MemSQL Streamliner. http://blog.memsql.com/spark-streamliner/.

[15] Odysseus. http://odysseus.informatik.uni-oldenburg.de/.

[16] PipelineDB. https://www.pipelinedb.com/.

[17] SQLstream. http://www.sqlstream.com.

[18] StreamBase. http://www.streambase.com/.

[19] StreamSQL. http://www.streambase.com/products/streambasecep/streamsql/.

[20] Tell - a distributed shared-data database. http://www.tell-project.org/.

[21] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[22] Daniel J Abadi, Don Carney, Ugur Çetintemel, 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—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[23] Yanif Ahmad, Bradley Berg, Ugur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, et al. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884. ACM, 2005.

[24] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: the stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[25] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[26] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[27] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2004.

[28] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 118–129. IEEE, 2005.

[29] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[30] Shivnath Babu and Jennifer Widom. Streamon: an adaptive engine for stream query processing. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 931–932. ACM, 2004.

[31] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[32] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 251–264, New York, NY, USA, 2015. ACM.

[33] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 251–264. ACM, 2015.

[34] Ferdinando Campanile, Alessandro Cilardo, Luigi Coppolino, and Luigi Romano. Adaptable parsing of real-time data streams. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, pages 412–418, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[36] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[37] Ufuk Celebi, Kostas Tzoumas, and Stephan Ewen. How Flink handles backpressure. `http://data-artisans.com/how-flink-handles-backpressure/`, 2015.

[38] K Mani Chandy, Joseph Kiniry, Adam Rifkin, Daniel Zimmerman, Wesley Tanaka, and Luke Weisman. Infospheres distributed object system, May 2005. US Patent 6,898,791.

[39] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[40] Yongqiang Huang and Hector Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 3–12. IEEE, 2001.

[41] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[42] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, August 2008.

[43] Pythagoras Karampiperis, Giannis Mouchakis, Georgios Paliouras, and Vangelis Karkaletsis. Er designer toolkit: a graphical event definition authoring tool. *Universal Access in the Information Society*, 13(1):115–123, 2014.

[44] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, April 2011.

[45] Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and distributed stream processing: Systems classification and specific issues. 2015.

[46] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[47] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[48] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. Efficient processing of window functions in analytical sql queries. *Proc. VLDB Endow.*, 8(10):1058–1069, June 2015.

[49] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[50] David Luckham. What's the difference between esp and cep. *Online Article, August*, 2006.

[51] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336–354, 1995.

[52] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics. In *BTW*, 2013.

[53] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[54] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. In *BTW*, pages 383–402, 2015.

[55] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, SIGMOD '15, pages 677–689, New York, NY, USA, 2015. ACM.

[56] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[57] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[58] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[59] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases*, pages 799–810. VLDB Endowment, 2006.

[60] Stratis D Viglas, Jeffrey F Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 285–296. VLDB Endowment, 2003.

[61] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802. IEEE, 2005.