Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems

[Experiments and Analyses]

Andreas Kipf[1] Varun Pandey[1] Jan Böttcher[1]
Lucas Braun[2] Thomas Neumann[1] Alfons Kemper[1]

[1]Technical University of Munich
{kipf, pandey, boettche, neumann, kemper}@in.tum.de

[2]ETH Zurich
braunl@inf.ethz.ch

ABSTRACT

Today's streaming applications demand increasingly high event throughput rates and are often subject to strict latency constraints. To allow for more complex workloads, such as window-based aggregations, streaming systems need to support *stateful* event processing. This introduces new challenges for streaming engines as the state needs to be maintained in a consistent and durable manner and simultaneously accessed by complex queries for real-time analytics.

Modern streaming systems, such as Apache Flink, do not allow for efficiently exposing the state to analytical queries. Thus, data engineers are forced to keep the state in external data stores, which significantly increases the latencies until events are visible to analytical queries. Proprietary solutions have been created to meet data freshness constraints. These solutions are expensive, error-prone, and difficult to maintain. Main-memory database systems, such as HyPer, achieve extremely low query response times while maintaining high update rates, which makes them well-suited for analytical streaming workloads. In this paper, we identify potential extensions to database systems to match the performance and usability of streaming systems.

CCS Concepts

•**Information systems** → **Stream management;**

Keywords

stream processing; main-memory database systems

1. INTRODUCTION

Gartner recently forecasted that there will be more than *20 billion* connected devices in 2020, a 400% increase compared to 2016[1]. The growing popularity of Internet of Things applications [11], including connected vehicles, cell phones, and health monitoring devices, enable a variety of new business use cases and applications. These applications are typically built around streaming systems that are able to ingest and aggregate enormous amounts of events from different data sources. Given the spike of interest in building such applications, it is not surprising that dedicated stream processing systems, like Apache Storm[2], Apache Spark Streaming[3], or Apache Flink[4], are receiving significant attention not only in the database but also in the data science and in the open-source community.

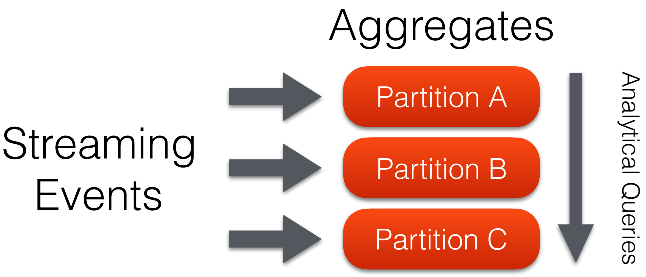

Figure 1: Analytics on fast data. Streaming events may be processed concurrently in different partitions, whereas analytical queries cross partition boundaries and require a consistent state.

To better understand the different types of workloads that these systems need to handle, we will walk through different ways of processing sensor readings (events) of connected vehicles that contain information about street conditions such as icy road segments.

First, a streaming system could warn vehicles about icy road segments based on the information of single events. In that case, the streaming system does not need to maintain state. We refer to such workloads as *stateless* streaming.

Second, a system could process the aggregated information of multiple events to decide if vehicles should be warned. Such an implementation requires the system to maintain a processing state, which introduces new challenges such as consistency and durability. We call this kind of workloads *stateful* streaming.

[1] http://www.gartner.com/newsroom/id/3165317

[2] http://storm.apache.org/

[3] http://spark.apache.org/streaming/

[4] http://flink.apache.org/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.06

Third, a streaming system allows users to perform analytical queries on the entire set of aggregates (the conditions of all road segments across the city) to find the most critical segments. We refer to such workloads as *analytics on fast data*. These workloads are particularly challenging for a streaming system since it needs to perform computations across multiple partitions in a consistent manner to answer analytical queries (cf., Figure 1). In fact, without modifications, none of the streaming systems mentioned above can handle this use case.

One idea to mitigate this problem is to make use of the fact that these systems periodically flush their state to durable storage (e.g., HDFS) to address fault tolerance. This means that the system state becomes queryable for an analytical engine like Apache Spark[5]. However, the delay that this design introduces prohibits analytical queries to run on the most recent state, which is required by use cases like the one above.

Another example is the Huawei-AIM telecommunication workload described in [2]. In this use case, events represent sales and marketing information generated by phone calls. On the one hand, the application needs to maintain a huge set of aggregates per customer in order to trigger alerts for this particular customer (a *stateful* streaming workload). On the other hand, maintenance specialists might query the overall system state to localize sources for network failures or business analysts might run analytics to gather insights and propose new offers in real time (*analytics on fast data*).

Again, using an off-the-shelf stream processor does not solve the case described in [2] because it cannot handle the real-time analytics. There are, however, state-of-the-art main-memory database systems (MMDBs) dedicated to handle mixed OLTP and OLAP workloads, such as HyPer [7] and Tell[6], which seem promising because stream processing could also be seen as a particular class of OLTP workloads. These systems feature advanced query optimizers, compile queries to native code, and can thus achieve extremely low response times for complex analytical queries. Using efficient snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates* [7, 15, 8], these systems are able to sustain high transaction throughput rates in parallel to analytical query processing making them well-suited for workloads where analytical queries need to consider recently ingested data.

Despite all these advantages, it seems that data engineers are still reluctant to use MMDBs for stream processing. They either build their own solutions on top of modern streaming systems (e.g., Apache Flink) or hand-craft systems from scratch that are specifically tuned for particular workloads (e.g., AIM [2]). One reason that MMDBs are not widely used for streaming workloads is that they lack out-of-the-box streaming functionality, such as window functions, and adding this functionality (e.g., through stored procedures or user-defined functions) results in additional engineering. If MMDBs would offer a better support for streaming workloads (e.g., streaming extensions for SQL as proposed in StreamSQL [16]), they would be preferable over hand-crafted systems, which are also costly to maintain.

In this *Experiments and Analyses* paper, we thoroughly evaluate the usability and performance of MMDBs, modern streaming systems, and AIM, a hand-crafted system, using the Huawei-AIM workload [2]. Based on the evaluation results, we answer the question how off-the-shelf MMDBs can be extended to sufficiently satisfy the requirements of *analytics on fast data*. We identify a set of modifications that, if properly applied to the off-the-shelf MMDBs, allow these systems to address the needs of *analytics on fast data*.

Our contributions include:

- A rich survey of various MMDBs, modern streaming systems, and a hand-crafted system specifically designed to address the Huawei-AIM workload
- A thorough usability and performance evaluation including at least one representative of each of these classes of systems
- A discussion of how MMDBs can be extended to match the performance and usability of modern streaming systems

The remainder of this paper is structured as follows: Section 2 summarizes a broad variety of existing systems and Section 3 revisits the Huawei-AIM workload and describes how it can be implemented with these systems. Section 4 evaluates the performance of representatives of each kind of system with respect to this workload. Section 5 enumerates ideas regarding how to bridge the performance and usability gap between MMDBs and modern streaming systems and is followed by the conclusions to our evaluation presented in Section 6.

2. APPROACHES

There are numerous systems that can be used to build stream processing pipelines, including near real-time data warehousing solutions like Mesa [6] and in-memory incremental analytical engines like Trill [4]. S-Store [12] is an approach to integrating stream processing into an OLTP engine. Since addressing all of these systems is beyond the scope of this paper, we will focus on representative MMDBs, popular streaming systems from the open-source domain, and AIM [2], a hand-crafted highly-optimized solution.

2.1 Main-Memory Database Systems

There are multiple MMDBs that can handle *analytics on fast data* or more generally hybrid transactional/analytical processing (HTAP) workloads. In HTAP, transactions are usually more complex (e.g., TPC-C transactions) than the single-row transactions studied in this work.

2.1.1 HyPer

HyPer[7] is a MMDB that achieves an outstanding performance for both OLTP and OLAP workloads, even when they operate simultaneously on the same database. HyPer uses two different snapshotting mechanisms to avoid expensive synchronization. By leveraging the *copy-on-write* feature of the MMU, the *fork* mechanism [7] efficiently creates consistent copies of the database to enable analytical queries to run without interruptions. The second snapshotting mechanism [15] is based on *multi version concurrency*

[5] http://spark.apache.org/

[6] http://www.tell-project.org/

[7] When saying HyPer, we are referring to the research version of HyPer developed at the Technical University of Munich.

control (*MVCC*) and isolates transactions by versioning individual attributes. Currently, HyPer does not yet implement physical *MVCC* meaning that transactions do not run simultaneously with analytical queries but are interleaved. HyPer further features data-centric LLVM code generation with just-in-time compilation. Finally, HyPer has an advanced dynamic programming-based optimizer including the ability to unnest arbitrary queries.

2.1.2 MemSQL

MemSQL[8] is a MMDB that uses LLVM for code generation. In-memory data is organized row wise while on-disk data is organized column wise. MemSQL currently does not support stored procedures, thus making it difficult to implement stream processing workloads requiring a complex logic for updating state. To implement such a workload in MemSQL, one needs to implement the update logic externally, leading to costly round trips between the application and the database. Another alternative to implement such workloads in MemSQL is to use MemSQL Streamliner[9], which offers a connector between Spark (Streaming) and the relational database. Streaming results from Spark Streaming can be materialized into MemSQL for further investigation. The main drawback of this solution is that the two systems remain separated causing higher than necessary latencies. Further, streams cannot be joined with regular tables residing in MemSQL without materializing and transferring them to the relational database.

2.1.3 Tell

Tell is a distributed shared-data MMDB that supports OLTP and OLAP in parallel and is developed at the Systems Group at ETH Zurich. The implementation of Tell is fundamentally different from that of other systems presented in this paper as it separates the computation from the storage layer in such a way that both layers can scale out individually [10].

The storage layer, TellStore, is a versioned key-value store with additional support for fast scans and different storage layout options, such as *RowStore* and *ColumnMap*. *ColumnMap*, the preferred layout for HTAP workloads, was created as part of Analytics in Motion (AIM) [2] (cf., Section 2.3) and is a modified *Partition Attributes Across* (PAX) [1] approach that optimizes cache locality by storing data columnwise in blocks of cache size. This optimization allows *ColumnMap* to support fast scans and, at the same time, reasonably fast record lookups and updates. TellStore employs the *shared scan* technique, which allows incoming scan requests to be batched and processed all at once by a single thread. The *shared scan* can be parallelized efficiently by partitioning the data and using a dedicated scan thread for each of these partitions in parallel [18]. Isolation is guaranteed using a combination of *differential updates* [8] and *MVCC*. Updates are put into a *delta* data structure, which gets periodically merged with the *main* data structure that serves analytical queries. This approach is also used in SAP HANA [5].

Tell's compute layer offers two processing APIs: TellDB (C++) for general-purpose transactions and TellJava (Java) for read-only analytics. TellJava can be further integrated into distributed processing frameworks, including Apache Spark and Presto.

2.2 Modern Streaming Systems

In addition to MMDBs, there are dedicated streaming systems allowing for the implementation of streaming pipelines. These systems provide out-of-the-box functionality, including a rich set of operators to help data engineers to address the specific demands of streaming use cases.

2.2.1 Apache Samza

Apache Samza[10] is a distributed framework for continuous real-time data processing that is lightweight, elastic, and fault-tolerant. Samza uses Apache Kafka[11] (a durable publish-subscribe-based message passing system that allows replaying messages) for real-time feeds and produces output feeds for Kafka to consume. For distributed scheduling, fault tolerance, and resource allocation, Samza depends on Apache YARN and on Kafka. Samza employs a checkpointing mechanism to provide *at-least-once* guarantees. It creates checkpoints at predefined time intervals and in case of a job failure, it replays messages from the last checkpoint. A drawback of Samza is that it does not support *exactly-once* semantics. A message might be processed twice after a job failure, which can lead to non-exact results. That effect can be minimized by using shorter checkpoint time intervals.

2.2.2 Apache Flink

Apache Flink [3] is a combined batch and streaming processing system that supports *exactly-once* semantics. Flink follows a *tuple-at-a-time* approach, providing low latency. Using asynchronous checkpointing, Flink is able to decouple its fault-tolerance mechanism from the tuple processing. The processing continues while Flink periodically creates snapshots of the operator states and the in-flight tuples. Flink can achieve superior throughput compared to Apache Storm (cf., Section 2.2.4). In contrast to the other streaming systems, Flink allows for event time semantics. Flink allows the extraction of the actual event timestamp (i.e., the time when the event was originally captured) when an event arrives at the streaming engine to assign it to its appropriate window. A drawback of Flink is that current versions only allow maintaining state on an operator level. However, there is a pull request for a queryable state[12] to be released with Flink 1.2.0. The idea is to maintain an operator-independent state within Flink and expose it to external queries. Internally, the state is partitioned and guarantees fault tolerance (i.e., *exactly-once* semantics). A restriction of this solution is that it is only a key-value state supporting only point lookups. More complex queries, including full table scans, are not possible.

As a workaround, one can implement a custom operator that holds both the state and the logic for the corresponding analytical queries. The drawback of this approach is that the whole state and query logic has to be implemented manually. Further, this approach does not support concurrent stream and query processing since analytical queries can only be ingested through the stream processing pipeline itself resulting in an interleaved execution.

[8] http://www.memsql.com/

[9] http://blog.memsql.com/spark-streamliner/

[10] http://samza.apache.org/

[11] https://kafka.apache.org/

[12] https://issues.apache.org/jira/browse/FLINK-3779

2.2.3 *Apache Spark Streaming*

Apache Spark Streaming [19] is the streaming extension to the cluster computing platform Apache Spark. Spark Streaming organizes incoming streaming tuples into *micro-batches* that are being processed atomically thus optimizing for throughput. This approach allows the use of the same programming model for batch and stream processing. Spark Streaming supports *exactly-once* semantics.

2.2.4 *Apache Storm*

Apache Storm [17] is a widely used stream processing system that does not guarantee state consistency and follows a *tuple-at-a-time* approach, thus favoring low latency over throughput. Storm implements *at-least-once* semantics by keeping upstream backups of data that are being replayed if no acknowledgements have been received from downstream nodes. Trident[13] extends Storm with *exactly-once* semantics and allows running queries on consistent state.

2.3 AIM

In collaboration with Huawei, researchers of the Systems Group at ETH Zurich designed the AIM system to address the specific characteristics of a telecommunications workload. AIM is a research prototype that allows efficient aggregation of high-throughput data streams. It was specifically designed to address the Huawei-AIM workload that we use for evaluation purposes in this paper (cf., Section 3). Due to its hand-optimized nature, AIM achieves an outstanding performance on that workload and therefore serves as a baseline for our experiments. AIM has a three-tier architecture consisting of storage, event stream processing (ESP), and real-time analytics (RTA) nodes (or threads if deployed in a standalone setting). RTA nodes push analytical queries down to the storage nodes, merge the partial results, and finally deliver the results to the client. ESP nodes process the incoming event stream, evaluate alert triggers, and update corresponding records by sending *Get* and *Put* requests to the storage nodes. The storage nodes store horizontally-partitioned data in a *ColumnMap* layout and employ *shared scans* as described in Section 2.1.3. AIM can also be deployed standalone, which eliminates network costs and therefore tests the pure read, write, and scan performance of the server.

2.4 Summary

A comparison of different aspects of stream processing approaches is presented in Table 1. These aspects include:

Semantics Streaming engines make different guarantees regarding how messages (i.e., events) are being processed. A streaming engine only ensures completely correct results when providing *exactly-once* guarantees. Some engines optimize for low latency and thus often cannot provide *exactly-once* guarantees as this would require them to implement transactions, which are expensive in a distributed setting. Therefore, streaming engines often fall back to *at-least-once* semantics (i.e., a message will be resent until it is processed at least once), which are good enough for many applications. Many stream processing engines require a durable data source for *exactly-once* guarantees because they only persist their processing state at certain points of time (often called checkpoints). In case of a failure, messages need to be replayed from the last checkpoint. In contrast, database systems achieve durability through the use of redo logs and thus only need to replay messages sent during the time the database system was down. The third processing guarantee is *at-most-once*. In an *at-most-once* setting, messages might get lost but are never processed twice or more often. Few systems implement this approach since loosing data is an undesirable property for most applications.

Durability Durability is closely related to the semantics offered by stream processing systems. While some systems require a durable data source to achieve durability, others provide durability out-of-the-box.

Latency Especially in real-time scenarios, low latencies are crucial to deliver valuable results. As stated above, latency often depends on the processing guarantee offered by a system. MMDBs that often run on a single machine or are optimized for low-latency networks can yield low latencies while providing *exactly-once* processing guarantees.

Computation model There are two computation models: *tuple-at-a-time* and *micro-batch*. The natural approach is to process streams continuously. However, streams can also be batched and processed as small chunks of data. Spark Streaming follows this approach allowing it to achieve high throughput rates. However, following a *tuple-at-a-time*-based approach does not necessarily lead to lower throughput since the computation model can be independent from the checkpointing interval. For instance, Flink follows a *tuple-at-a-time*-based approach combined with a batch-based checkpointing mechanism thus optimizing for both latency and throughput. MMDBs usually treat stream events as transactions, which might also be batched for better performance (e.g., Tell processes 100 events within a single transaction).

Throughput Another important aspect in stream processing is throughput. Particularly when costs matter, higher throughput helps to reduce the number of required resources. Due to the low costs to process single-row transactions (updating aggregates of single entities), throughput mainly depends on the employed fault-tolerance mechanism and whether a system batches transactions. Throughput increases with longer checkpointing intervals.

State management For mixed OLTP and OLAP workloads, the state updated by the OLTP subsystem needs to be exposed to the OLAP subsystem. Traditional streaming engines, such as Apache Storm, do not allow maintaining state. They are only designed to process and transform an input into an output data stream preventing writing *stateful* stream processing applications (e.g., aggregations over windows). Trident extends Storm with state management capabilities. Flink only maintains states on an operator basis and currently does not support global states that can be accessed by analytical queries. Database systems, on the other hand, can persist streaming results in temporary

[13] http://storm.apache.org/documentation/Trident-state

	MMDBs			Modern Streaming Systems				
Aspect	**HyPer**	**MemSQL**	**Tell**	**Samza**	**Flink**	**Spark Streaming**	**Storm**	**AIM**
Semantics	Exactly-once	Exactly-once	Exactly-once	At-least-once	Exactly-once	Exactly-once	Exactly-once	Exactly-once
Durability	Yes	Yes	No	With durable data source	With durable data source	With durable data source	With durable data source	No
Latency	Low	Low	Low	High (writes messages to disk)	Low	Medium (depends on batch size)	Low	Low
Computation model	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Micro-batch	Micro-batch	Tuple-at-a-time
Throughput	High	High	High	High	High	Medium (depends on batch size)	Low	High
State management	Yes	Yes	Yes	Yes (durable K/V store)	Yes	Yes (writes into storage)	Yes	Yes
Parallel read/write access to state	Copy on write, MVCC	No	Differential updates, MVCC	No	No	No	No	Differential updates
Implementation languages	C++, LLVM	C++, LLVM	C++, LLVM	Java, Scala	Java	Java, Scala	Java, Clojure	C++
User-facing languages	SQL	SQL	C++, Java, Scala (through Spark shell), SQL (through Presto shell)	Java, Scala	Java, Scala	Java, Scala, Python, SparkSQL	Any (through Apache Thrift)	C++
Own memory management	Yes	Yes	Yes (w/ GC)	No	Yes	Yes	No	Yes
Window support	Using stored procedures	Only manually	Only manually	Very basic	Very powerful	Basic	Basic	Using template code

Table 1: Comparison of different stream processing approaches

tables allowing OLAP queries to access them as if they were regular database tables.

Parallel read/write access to state As mentioned earlier, Trident extends Storm with state management functionalities; however, it does not allow analytical queries and updates to access state in parallel. Instead, they have to be interleaved to ensure a consistent view of the state. In contrast, modern MMDBs can efficiently expose their current state to analytical queries through the use of snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates.*

Implementation languages Most of the streaming systems are written in a JVM-based language, whereas MMDBs are usually implemented in C or C++. The trend is to compile queries to native code. HyPer, Tell, and MemSQL use LLVM as a compiler backend.

User-facing languages The Apache systems support primarily JVM-based languages while the MMDBs all support SQL and, in the case of Tell, additional languages through its Spark and Presto integration.

Own memory management Whether a system employs its own memory management or fully relies on the memory management of the JVM. Spark Streaming and Flink are based on the JVM but still employ their own memory management to have a better control over garbage collection cycles.

Window support In streaming applications, aggregations are usually computed on a window basis. Two basic window types are sliding and tumbling. Sliding windows are contiguous time or count-based intervals, such as *last 24 hours* or *last 10,000 events.* Tumbling windows are non-overlapping time or count-based intervals, such as *today* or *every 10,000 events.* All of the analyzed streaming engines support these two kinds of windows. In particular, Flink offers extensive functionality to specify windows, supporting custom window assigners, triggers, and evictors. AIM supports tumbling windows for specific time intervals and the standard aggregation functions through templated code. The window definitions are loaded at startup and cannot be changed afterwards. The analyzed MMDBs have no natural window support. However, in the case of HyPer, windows can be manually implemented using stored procedures.

3. WORKLOAD

AIM was motivated by a telecommunication workload, which we will refer to as Huawei-AIM use case [2]. We chose this workload as it is well-defined and represents the workload class of *analytics on fast data.*

3.1 Description

The Huawei-AIM use case requires events, more specifically *call records*, to be aggregated and made available to analytical queries. The system's state, which AIM calls the

<table>
<tr><td rowspan="4">subscriber ID</td><td colspan="6">international calls</td><td>...</td></tr>
<tr><td colspan="5">today</td><td>...</td><td>...</td></tr>
<tr><td rowspan="2">count</td><td colspan="3">duration</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>sum</td><td>min</td><td>max</td><td>...</td><td>...</td><td>...</td></tr>
</table>

Table 2: Schema snippet of the Analytics Matrix

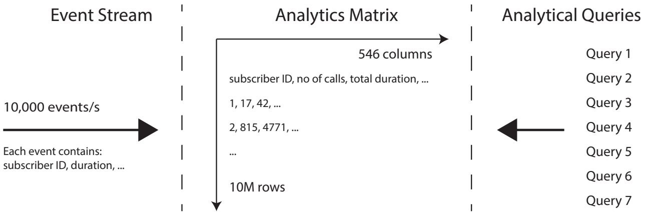

Figure 2: The AIM-Huawei workload

Analytics Matrix, is a materialized view on a large number of aggregates for each individual subscriber. There is an aggregate for each combination of aggregation function (min, max, sum), aggregation window (this day, this week, ...) and several event attributes as shown in Table 2, which shows a small part of the conceptual schema of an *Analytics Matrix*. For instance, there is an aggregate for the shortest duration of an international phone call today (attribute *min* in Table 2). The number of such aggregates (which defines the number of columns of the *Analytics Matrix*) is a workload parameter with default value 546, which we use in our experiments. The *Analytics Matrix* also contains foreign keys to dimension tables. Since these dimension tables are very small, we omit them in our experiments.

The use case requires two things to be done in real time: (a) update *Analytics Matrix* and (b) run analytical queries on the current state of the *Analytics Matrix*. (a) is referred to as Event Stream Processing (ESP) and (b) as Real-Time Analytics (RTA). When an event arrives in ESP, the corresponding record in the *Analytics Matrix* has to be atomically updated. RTA, on the other hand, is used to answer business intelligence questions. RTA queries are continuously being issued by one or multiple clients and are evaluated on a consistent state of the *Analytics Matrix*. This consistent state (or snapshot) is not allowed to be older than a certain bound t_{fresh}, which is a service level objective (SLO) of the Huawei-AIM benchmark and defaults to one second. Table 3 shows the seven queries from the original benchmark [2]. Additionally, users may issue ad-hoc queries. Since ad-hoc queries are not available upfront and can involve any number of attributes, it is impractical for a stream processing system to create specialized index structures.

Figure 2 summarizes the workload components. Events are ingested at a specific rate f_{ESP}, which will usually be 10,000 events per second in our experiments. Each event consists of a subscriber ID and call-dependent details, such as the call's duration, cost, and type (i.e., local or international). The *Analytics Matrix* is the aggregated state on the call records as described earlier and consists of 546 columns and 10 million rows, each representing the state of one subscriber. Depending on the event details, the corresponding subset of columns in the *Analytics Matrix* is updated for the particular subscriber. These updates are made available to analytical queries within t_{fresh}.

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, 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;

Table 3: RTA queries 1 to 7, $\alpha \in$ [0,2], $\beta \in$ [2,5], $\gamma \in$ [2,10], $\delta \in$ [20,150], *t* $\in$ SubscriptionTypes, *cat* $\in$ Categories, *cty* $\in$ Countries, *v* $\in$ CellValueTypes

3.2 Implementations

We implemented the workload using at least one representative of each of the three categories: MMDBs, modern streaming systems, and hand-crafted systems. We chose Flink as a representative modern streaming system since it features a continuous processing model combined with a batch-based fault-tolerance mechanism allowing for low latency under high throughput conditions. MemSQL currently does not support stored procedures[14]. Without this feature, we were not able to implement the event processing part of the workload in an efficient way and therefore decided not to further evaluate MemSQL.

3.2.1 HyPer

Our workload implementation in HyPer was based on the work of [2]. ESP is performed using a stored procedure that updates aggregates stored in the *Analytics Matrix*, which is implemented as a regular database table. RTA query processing is implemented using SQL queries on that table.

When HyPer was first evaluated using the Huawei-AIM benchmark in [2], HyPer was configured to use a *copy-on-write*-based snapshotting technique that forked a child from the main OLTP process at a specific time interval. This enables RTA queries to be executed on a consistent snapshot of the *Analytics Matrix*. Since the table representing the *Analytics Matrix* can be as large as 50 GBs, forking a child of the OLTP process (essentially a copy of its page table) may take up to a hundred milliseconds. Additionally, our workload updates the records of randomly selected subscribers at a rate of 10,000 events/s, which may impact performance as the *copy-on-write* mechanism copies updated pages to main-

[14] http://docs.memsql.com/docs/mysql-features-unsupported-in-memsql

tain consistent snapshots for RTA queries. HyPer currently does not implement physical *MVCC*[15], which would lead to better results than a *copy-on-write*-based approach. The evaluated implementation interleaves the execution of multiple analytical queries thereby hiding memory latencies and single-threaded phases (e.g., result materialization). Writes, however, are never executed at the same time than analytical queries.

HyPer implements the PostgreSQL wire protocol allowing one to use any PostgreSQL client. In our experiments, we used PostgreSQL's C++ library (pqxx) to communicate between clients and HyPer (using TCP over UNIX domain sockets). Since HyPer currently does not implement batched transactions, HyPer's event processing throughput would be purely limited by network round trips between subsequent write requests, context switches on the server to receive incoming requests, and deserialization costs. To simulate batch processing, we decided to additionally generate the events within HyPer and only process these. In other words, instead of actually transferring the batch of events from the client to the server, we send a request to generate and process a specified number of events.

3.2.2 *Tell*

For Tell, we used the Huawei-AIM benchmark implementation from the Tell GitHub project[16]. We configured TellStore to use the *ColumnMap* layout with a total of 84 GB of memory, more than twice the memory that HyPer uses. With less memory, TellStore regularly ran out of memory, especially with multiple storage threads. To minimize NUMA effects, we configured Tell to run the storage layer (TellStore) on NUMA node 0 and the compute layer (RTA and ESP server-side threads) on node 1. RTA and ESP clients were also run on node 1. With this configuration, Tell achieved significantly better numbers than with the non-NUMA-aware configuration.

It is worth mentioning that Tell, as opposed to AIM, cannot be deployed in standalone mode. Whereas in AIM, Flink, and HyPer events are generated internally, Tell needs a client that generates events and sends them to the server (using UDP over Ethernet). Additionally, the server needs to send read and write requests to the storage (using RDMA over InfiniBand). Compared to all other implementations presented in this section, this makes ESP much more expensive as the overheads of network costs, context switching, and deserialization cost are paid twice (cf., Section 3.2.1). These extra costs should be be taken into account when looking at the performance results.

As Tell is a layered system, we have to carefully allocate threads to layers. In the compute layer, we have to allocate a specific number of ESP and RTA processing threads, whereas in the storage layer, we have to allocate the right number of scan threads (responsible for analytical query processing). The storage layer also runs one thread that integrates updates into the next snapshot for analytics and one thread for garbage collection. Microbenchmarks revealed the optimal thread allocation strategy for each workload as shown in Table 4. In general, Tell has a lot of different parameters most of which relate to memory management; and fine-tuning these parameters to get the best performance was a tedious task.

	Compute		**Storage**			
Workload	**ESP**	**RTA**	**scan**	**update**	**GC**	**Total**
read/write	1	n	n	1	1	$2n+2^*$
read-only	0	n	n	0	0	$2n$
write-only	n	0	0	1	0	$n+1$

Table 4: Thread allocation strategy for different workloads

*Since GC is only running from time to time and the update thread is also mostly idling for 10,000 events per second, both threads have an average CPU usage clearly below 50%. This is why we count them as one.

3.2.3 *AIM*

Since the AIM system was specifically designed to address the AIM-Huawei workload, we assumed that it would achieve the best performance on the full workload and thus we used it as a baseline for our experiments. We used the same version used in [2] but in standalone mode where client and server communicate through shared memory. For the overall and the read-only experiments, we increased the number of RTA threads (and used one ESP thread), whereas for the write-only experiments, we increased the number of ESP threads.

3.2.4 *Flink*

Currently, Flink does not support exposing its internal state to external analytical queries. There is, however, a pull request for a partitioned key-value store that will be queryable. However, this queryable state only supports point lookups and thus cannot be used to implement the AIM workload. We implemented a custom operator that supports table scans to meet the requirements of the AIM workload. We experimented with a row and a column store layout for storing the state. Since the AIM workload is mostly analytical, we opted for the column store layout.

Similar to HyPer, we generated the events internally in Flink. We also implemented a version that uses Kafka for event ingestion, which will not be included in the results, as we found no significant difference in performance compared to the version that generates the events internally. In production, Kafka, or any other durable data source, is preferable to ensure full fault tolerance.

Since we want to make the most recent state available to analytical queries, windows need to be computed on an event basis. As Flink's built-in operators are not optimized for these continuous window computations, we chose to manually implement the window logic, which yielded better results. We did not enable Flink's checkpointing mechanism since the processing state of the Huawei-AIM workload can be as large as 50 GBs. Persisting a state of this size would lead to a significant performance penalty.

Flink provides many built-in functionalities that seem suitable for our workload including windowed streams supporting various aggregation functions (e.g., min, max, and sum). We tried to make use of the provided functionalities. However, in the studied version of Flink, combining multiple aggregation functions that produce only one single output stream is not yet supported. For this reason, we implemented a custom aggregation operator.

[15] [15] explains how versioned positions allow for fast scans.

[16] https://github.com/tellproject/aim-benchmark

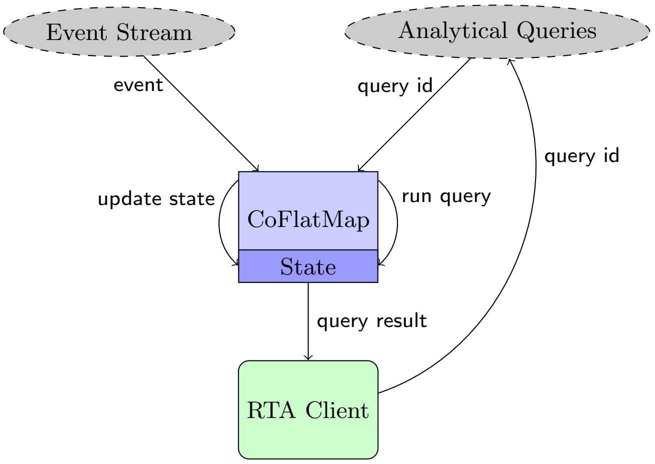

Figure 3: Hybrid processing in Flink. A CoFlatMap operator interleaves events with analytical queries.

All aggregations in the AIM workload are windowed. We could express this behavior using Flink's built-in window operators. For only one window type, this works well. However, with two or more different window types, the different windows would need to be merged into one consistent state across all windows. As this is not a straightforward operation in Flink, we decided to implement windows ourselves.

Another challenge was to run the analytical queries on the state maintained by the event processing pipeline. Flink does not provide a globally accessible state that can be used in such cases. States are only maintained at an operator level and cannot be accessed from outside. We solved this problem by processing both the event stream and the analytical queries in the same CoFlatMap operator as shown in Figure 3. Both streams are processed interleaved using two individual FlatMap functions that both work on the same shared state. This works as both functions are part of the same operator. Our implementation interleaves the two different streams on a partition basis. Since Flink follows the embarrassingly parallel paradigm, it is not designed to synchronize access across partitions. As described in [2], the AIM-Huawei workload does not require such a global synchronization since events are only ordered on an entity basis.

A powerful feature of Flink is its partitioning. Flink automatically partitions elements of a stream by their key and assigns the partitions to a parallel instance of each operator. Each instance of our CoFlatMap operator only receives the events for its partition and thereby maintains a part of the total state. The analytical queries, however, should run on the whole state. Therefore, we broadcast the queries to each CoFlatMap operator instance and run them on the individual partitions. The resulting partial results are merged in a subsequent operator.

In our experiments, we used Kafka to send queries since it integrates well with Flink and ensures that no queries are lost. It would also be possible to ingest the queries using a TCP client or other more sophisticated handwritten clients.

4. PERFORMANCE EVALUATION

We begin with a performance evaluation using the complete Huawei-AIM workload as described in Section 3. We drill down into the different aspects of the workload, including updates and real-time analytics. We then investigate the performance impact of the number of clients and the number of maintained aggregates.

System	Version
HyPer	Sep 12, 2016
Tell	0.2
AIM	Same version as used in [2]
Flink	1.1-Snapshot

Table 5: Evaluated systems

We did not evaluate how the systems scale out to multiple machines since the evaluated version of HyPer is standalone. In future work, we plan to extend HyPer with distributed event processing. However, the boundary between distributed systems and single-machine many-core systems with non-uniform memory access is blurry. In fact, as shown in the following section, even on our single two-socket machine the performance dropped when scaling beyond a single NUMA node.

4.1 Configuration

We evaluated the different systems (cf., Table 5) on an Ubuntu 15.10 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 hyperthreads) each, resulting in a total of 20 physical cores (40 hyperthreads). The sockets communicate using a high-speed QPI interconnect (16 GB/s).

We placed the clients on the same machine as the server and generated events and queries by one client thread each (except for Tell where we used eight RTA client threads). Setting the total number of threads[17] was enough to run HyPer and Flink out-of-the-box. Conversely, Tell and AIM required more tedious fine-tuning and server threads were allocated as explained in Sections 3.2.2 and 3.2.3. As one can see from these allocation schemes, some workloads require more than one thread even in the most basic setting, which is why the measurements for AIM and Tell do not typically start at one thread and may have gaps.

4.2 Overall Performance

Figure 4 illustrates the query throughput when running the full workload, which consists of 10M subscribers, 10,000 events per second, and the seven analytical queries (cf., Table 3) where each of them is executed with equal probability. Further, daily and hourly windows are maintained leading to a total of 546 aggregates. AIM achieved the best performance. With two threads, it had a throughput of 14.8 queries/s and its best throughput, with eight threads, was 145 queries/s. The reason why AIM achieves its best performance at eight (and not at ten) threads is a NUMA effect: Since AIM statically pins threads to cores and allocates memory locally whenever possible, the total number of client and threads (2 + 8 = 10) precisely fits on NUMA node 0. Hence, there is no communication to a remote memory region as it is the case for nine and ten threads. The spike at four threads probably relates to non-uniform communica-

[17] Unless otherwise noted, we are always referring to the server-side threads.

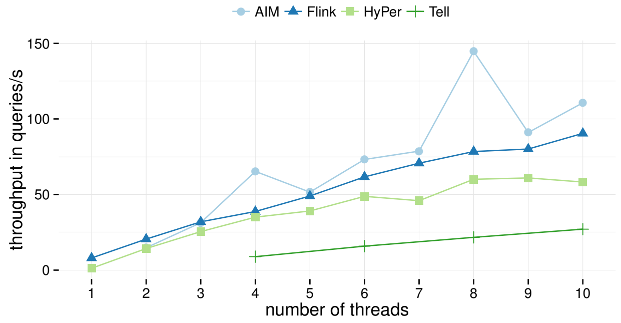

Figure 4: Analytical query throughput for 10M subscribers at 10,000 events/s

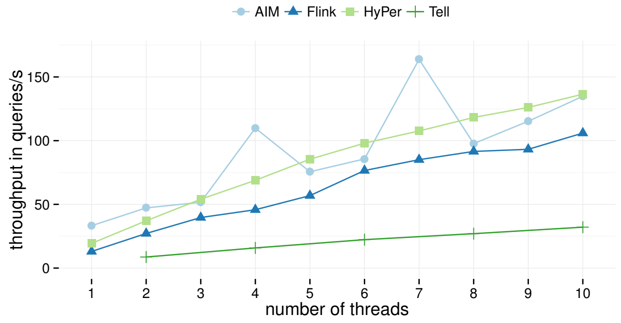

Figure 5: Analytical query throughput for 10M subscribers

tion paths between the cores on NUMA node 0. The spikes observed here are reproducable, and are, as we will see, also present in other workloads. Flink matches the performance of AIM for two threads and scales up to 90.5 queries/s using ten threads. HyPer achieved a throughput of 14.3 and 70.0 queries/s with two and nine threads, respectively. HyPer's throughput is lower than AIM's since it interleaves analytical queries with writes (i.e., writes block reads) while AIM processes them in parallel. With four threads, Tell achieved a query throughput of 8.90 queries/s and 27.1 queries/s with ten threads.

4.3 Read Performance

Figure 5 shows the analytical query throughput for the different systems with an increasing number of threads without concurrent events. With one thread, HyPer processed 19.4 queries/s while AIM sustained a throughput of 33.3 queries/s. As we increased the number of threads, HyPer sometimes outperformed AIM and its throughput increased linearly while AIM showed the same spikes as before[18]. HyPer's maximum throughput was 136 queries/s with ten threads compared to 164 queries/s for AIM with seven threads. Flink's throughput was 13.1 queries/s using one thread and gradually increased to 105.9 queries/s with ten threads. With two threads, Tell sustained a throughput of 8.68 queries/s while its maximum throughput was 32.1 queries/s using ten threads.

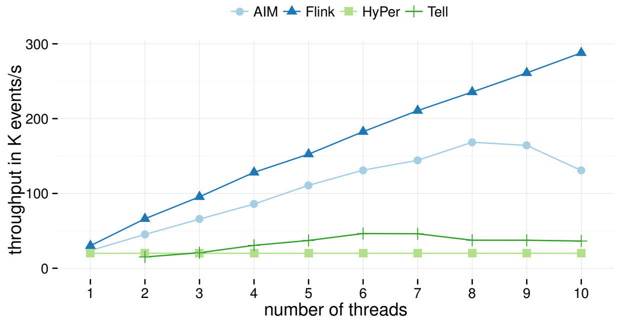

Figure 6: Event processing throughput with an increasing number of event processing threads

4.4 Write Performance

Figure 6 shows the event processing throughput of the different systems with an increasing number of event processing threads. This time, we evaluated the systems purely on the basis of their write throughput without running any analytical queries in parallel. Flink achieved the best write performance by far. Using one thread, it had a throughput of 30,100 events/s and the throughput scaled almost linearly to 288,000 events/s using ten threads. There are two reasons for this: (1) Flink partitions the state depending on the number of available processing threads. With this strategy, it scales well since there is no cross-partition synchronization involved. (2) Flink does not have any overhead introduced by snapshotting mechanisms or durability guarantees. AIM processed 23,700 events/s using one thread and achieved a maximum throughput of 168,000 events/s using eight threads, roughly 1.7x less than Flink. Again, we see the NUMA effect described earlier. AIM also partitions the state to scale its write throughput, but since its *differential update* mechanism introduces an overhead, AIM did not perform as well as Flink. Tell was able to process up to 46,600 events/s using six threads. The reason for the performance degradation after six threads is again a NUMA effect. All ESP processing threads as well as threads that handle UDP events are allocated on NUMA node 1 leading to an oversubscription of cores. HyPer sustained a throughput of 20,000 events/s in all cases since it only uses one single thread to process transactions.

4.5 Query Response Times

In this experiment, we measured the response time for each of the seven analytical queries with and without concurrent writes (10,000 events/s) using four threads. Table 6 shows the individual query response times and the overall average. HyPer's performance degraded the most when writes were added to the query processing workload. The reason is that HyPer interleaves analytical queries with writes. As shown in the previous section, HyPer's write throughput is limited to 20,000 events/s and does not scale for multiple threads. Thus, an event throughput of 10,000 events/s blocks the query processing for about 500 ms ev-

[18] AIM cannot be configured with zero ESP threads, which is why there is an additional idle ESP thread that we do not account for, but which nevertheless occupies its CPU. This is why the spike is at seven threads this time.

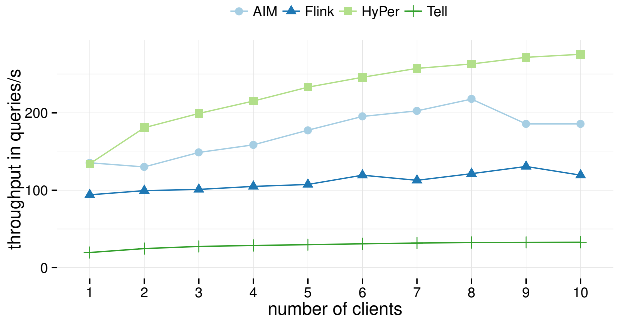

Figure 7: Analytical query throughput with an increasing number of clients

ery second. The query processing can only happen in the remaining 500 ms. AIM and Tell did not experience the same performance degradation since they perform writes and reads in parallel using the *differential updates* approach. Flink's performance does not drop much when adding 10,000 events/s as its (parallel) write throughput is so high that the analytical queries remain almost unaffected. However, we expect a higher performance degradation in Flink's analytical performance when increasing the number of events per second as Flink lacks efficient snapshotting mechanisms.

4.6 Impact of Number of Clients

Figure 7 shows the analytical query throughput with an increasing number of clients using ten server-side threads. HyPer performed the best of all systems and achieved a maximum throughput of 276 queries/s with ten client threads. HyPer's performance improves with multiple clients since it interleaves the execution of analytical queries (cf., Section 3.2.1). AIM's peak throughput was 218 queries/s with eight client threads. The gradual increase in the throughput shows the effect of the *shared scan* technique as AIM can now batch queries from multiple clients and process them all at once. The fact that the performance drops after eight threads shows that batching is only beneficial up to a certain point. Flink executes analytical queries as follows: Once a worker completed its part of the query, i.e., processed the query on its partition of the state, the worker can continue with the next query. The worker does not have to wait until the other partitions have been processed and the partial query results have been merged. For this reason, the idle time of threads decreases for more clients and the query throughput increases to 131 queries/s. Tell employs the same strategy as AIM and we can see a similar gradual increase in its throughput.

4.7 Impact of Number of Aggregates

In this experiment, we studied the impact of the number of aggregates being maintained. We measured the overall as well as the write performance of AIM, HyPer, and Flink while maintaining 42 instead of the original 546 aggregates. We did not measure Tell here since its AIM benchmark implementation was not flexible enough to accommodate schema changes.

Figure 8 shows the analytical query throughput for 10M subscribers and 42 aggregates at 10,000 events/s. Again, AIM achieved its best performance at eight threads (cf., Section 4.2) whereas Flink and HyPer did not experience such spikes. In contrast to the overall workload with 546 aggregates, HyPer achieved a higher performance than Flink throughout this experiment. The gain in HyPer's performance is expected since writes are now less expensive and thus singlethreaded phases are reduced. With ten threads, HyPer achieved a throughput of 125 queries/s (2.14x speedup over 546 aggregates) while Flink sustained 97.4 queries/s (1.08x).

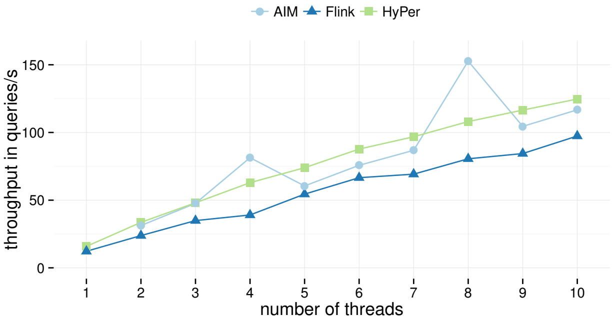

Figure 8: Analytical query throughput for 10M subscribers and 42 aggregates at 10,000 events/s

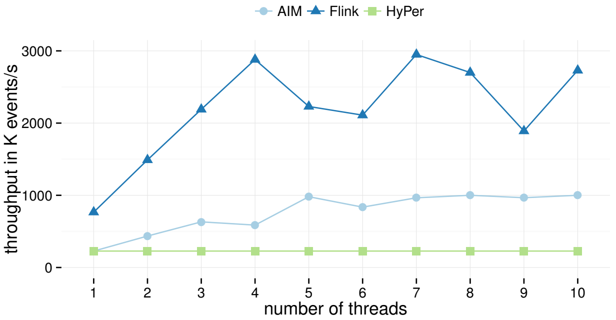

Figure 9: Event processing throughput for 42 aggregates with an increasing number of event processing threads

Figure 9 shows the event processing throughput for 42 aggregates with an increasing number of event processing threads. Note that we reduced the number of aggregates by a factor of 13. As expected, the throughputs improved significantly with less aggregates (cf. Section 4.4). With one thread, AIM and HyPer achieved a throughput of 227,000 (11.4x) and 228,000 events/s (9.62x), respectively, whereas Flink sustained 766,000 events/s (25.5x). With ten threads, AIM and Flink reached a throughput of 1,000,000 (7.69x) and 2,730,000 events/s (9.51x), respectively. HyPer's performance did not increase with more threads since it currently does not parallelize transactions.

5. CLOSING THE GAP

We have shown that general-purpose MMDBs perform fairly well on streaming workloads. Nevertheless, our experiments indicate that there is still a gap between the per-

	Read (in isolation)				Overall (w/ concurrent events)			
Query	**HyPer**	**Tell**	**AIM**	**Flink**	**HyPer**	**Tell**	**AIM**	**Flink**
Query 1	5.25	249	2.44	5.83	12.2	242	5.32	16.9
Query 2	7.41	241	3.91	5.10	14.3	253	4.94	8.03
Query 3	20.4	298	10.4	29.9	29.5	289	10.5	37.2
Query 4	4.05	269	2.98	3.14	12.1	281	4.67	6.97
Query 5	12.5	264	21.1	37.8	20.7	271	38.3	45.1
Query 6	33.8	505	13.8	24.4	84.1	492	54.4	33.6
Query 7	17.7	246	9.04	24.4	25.8	236	17.5	32.8
Average	14.4	296	9.10	18.7	28.4	295	19.3	25.8

Table 6: Query response times in milliseconds

formance and usability of MMDBs and modern streaming systems, such as Flink, particularly when it comes to scalable event processing (cf., Section 4.4).

We propose a threefold approach to improve the overall write performance of MMDBs: (a) improve single-threaded write performance, (b) use all cores on a single machine, and (c) distribute load across multiple machines. In the following, we will describe each of these aspects in more detail.

Event ingestion in modern streaming systems is usually implemented using a durable data source, such as Kafka. This allows the streaming system to neglect durability, leading to higher throughputs. Durability in these systems is usually more coarse-grained than in MMDBs. To match Flink's single-threaded write performance, MMDBs would need to offer a more coarse-grained durability level by using durable data sources instead of employing fine-grained redo log mechanisms.

AIM, Flink, and Tell are capable of processing events in parallel, whereas HyPer processes transactions in a single thread. To match their scalability, HyPer would need to be extended with parallel single-row transactions, which are less complicated to parallelize than full transactions. Such a streaming-optimized transaction isolation would only ensure that there are no conflicts on the primary key column(s). Work on allowing concurrent write transactions in HyPer is ongoing [9]. In Tell, for instance, batches of events are processed within the scope of a transaction and several transactions can be executed in parallel. The latter, however, comes at the high price of maintaining multiple versions of the data, which again reduces performance as our experiments illustrate.

MMDBs also need to be able to distribute writes across multiple machines. HyPer, for instance, could employ a similar strategy as Flink, which partitions the event input stream and distributes it across nodes. Towards this end, HyPer could employ the ScyPer architecture as suggested in [13], where transactions are processed by the primary ScyPer node, which multicasts redo logs to secondary nodes. These secondaries are dedicated to query processing thus freeing resources and leading to higher throughput rates on the primary node. To scale out writes as well as reads, these two strategies could be combined by having multiple event processing nodes (the primary node in the ScyPer architecture), each of them being responsible for a subset of events. These event processing nodes would then multicast their redo logs to query processing nodes (secondary nodes in the ScyPer architecture). We plan to investigate such an architecture in future research.

From a usability perspective, modern streaming systems offer many features that help users to set up streaming applications, such as their out-of-the-box support for sliding and tumbling windows. On the one hand, MMDBs support arbitrary SQL allowing users to customize the analytical parts of their workloads and to issue ad-hoc queries. On the other hand, adding windowed aggregation functions using stored procedures is a cumbersome task. PipelineDB[19], which is built on top of PostgreSQL, solves this usability issue by extending SQL with streaming features but still cannot match the performance of dedicated streaming systems as we found in early experiments. In addition to out-of-the-box streaming features, modern streaming systems allow users to add custom code. There has been work to allow for the same in MMDBs, such as the integration of high-level programming languages (using user-defined functions). These additions, however, still do not allow for the same flexibility as writing plain old Java code. These limitations are mainly caused by the multi-tenant nature of database systems and the security level that these systems need to fulfill. MMDBs would need to allow for optionally disabling security arrangements (e.g., enforcing access rights) in favor of better extensibility. Another mitigation path that MMDBs could follow is to simply add more streaming features to its SQL processing logic, namely, window-based semantics as proposed by PipelineDB and StreamSQL [16]. This is also a topic we plan to address in future research.

Besides extending MMDBs to better support streaming use cases, the gap between MMDBs and streaming systems could be closed from the other direction, which would mean extending streaming systems with additional storage management features and query mechanisms. There is ongoing work to make use of Apache Calcite[20] (a SQL parser and optimizer framework) to extend Flink with streaming SQL and query optimization capabilities. Cache and register locality are crucial for high query performance and they can both be addressed very efficiently by compiling query plans into native code [14]. While this was possible to achieve in systems like HyPer or Tell, which are written in C++ and LLVM, it is more difficult to implement using JVM-based systems such as the streaming systems evaluated in this paper. Moreover, implementing efficient storage management capabilities is a tedious task in JVM-based languages because to ensure data locality, custom memory management has to be implemented outside the JVM heap.

[19] https://www.pipelinedb.com/

[20] https://calcite.apache.org

6. CONCLUSIONS

In this paper, we evaluated a broad set of architectures to address *analytics on fast data*. We performed an experimental evaluation including at least one representative of each architecture. Our experiences as well as the performance results indicate that there still exists a gap between MMDBs and dedicated streaming systems. MMDBs are built for multi-tenant environments where durability and isolation guarantees are essential. Dedicated streaming systems, on the other hand, often compromise these guarantees in exchange for better performance or higher flexibility. To catch up with these systems, MMDBs need to be able to optionally lower their guarantees. In addition, new architectures are required that allow MMDBs to scale both their event and query processing performance to keep up with the demands for increasingly high event throughput rates. The question remains whether these additions as well as new architectures can enable MMDBs to address a broad set of streaming workloads. Once fully implemented, we plan to evaluate HyPer with enabled *MVCC* to investigate the impact of event processing on analytics including the effects of the garbage collection overhead *MVCC* introduces.

7. ACKNOWLEDGEMENTS

We want to thank Gyula Fóra for his help in optimizing our workload implementation in Flink. This work has been sponsored by the German Federal Ministry of Education and Research (BMBF) grant FASTDATA 01IS12057. This work is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180. ACM, 2001.

[2] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. 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.

[3] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[5] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[6] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment*, 7(12):1259–1270, 2014.

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

[8] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[9] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[10] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 663–676. ACM, 2015.

[11] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[12] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-store: Streaming meets transaction processing. *Proceedings of the VLDB Endowment*, 8(13):2134–2145, 2015.

[13] 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.

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

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

[16] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[18] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[19] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.