AStream: Ad-hoc Shared Stream Processing

Jeyhun Karimov
DFKI GmbH
jeyhun.karimov@dfki.de

Tilmann Rabl
DFKI GmbH, TU Berlin
rabl@tu-berlin.de

Volker Markl
DFKI GmbH, TU Berlin
volker.markl@tu-berlin.de

ABSTRACT

In the last decade, many distributed stream processing engines (SPEs) were developed to perform continuous queries on massive online data. The central design principle of these engines is to handle queries that potentially run forever on data streams with a query-at-a-time model, i.e., each query is optimized and executed separately. In many real applications, streams are not only processed with long-running queries, but also thousands of short-running ad-hoc queries. To support this efficiently, it is essential to share resources and computation for stream ad-hoc queries in a multi-user environment.

The goal of this paper is to bridge the gap between stream processing and ad-hoc queries in SPEs by sharing computation and resources. We define three main requirements for ad-hoc shared stream processing: (1) *Integration*: Ad-hoc query processing should be a composable layer which can extend stream operators, such as join, aggregation, and window operators; (2) *Consistency*: Ad-hoc query creation and deletion must be performed in a consistent manner and ensure exactly-once semantics and correctness; (3) *Performance*: In contrast to state-of-the-art SPEs, ad-hoc SPE should not only maximize data throughput but also query throughout via incremental computation and resource sharing.

Based on these requirements, we have developed AStream, an ad-hoc, shared computation stream processing framework. To the best of our knowledge, AStream is the first system that supports distributed ad-hoc stream processing. AStream is built on top of Apache Flink. Our experiments show that AStream shows comparable results to Flink for single query deployments and outperforms it in orders of magnitude with multiple queries.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319884

CCS CONCEPTS

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

KEYWORDS

Data streams, ad-hoc query processing, shared query processing

ACM Reference Format:
Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-hoc Shared Stream Processing. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3319884

1 INTRODUCTION

Several open source distributed SPEs, such as Apache Spark Streaming [54], Apache Storm [46], Apache Flink [11], and Apache Apex [1], were developed to cope with high-speed data streams from IoT, social media, and Web applications. Large companies with hundreds of developers use SPEs in their production environment. Developers in the production environment create long-running stream queries for continuous monitoring or reporting and short-living stream queries for testing queries on live streams. The best practice today is to fork the input stream using a message bus like Apache Kafka [51] while adding additional resources for performing new queries [40]. Hundreds of developers, creating thousands of ad-hoc queries, make this a challenging and inefficient setup.

1.1 Motivating Example

A typical example for stream processing setups are online services such as games. Online gaming today is often cloud-based to satisfy varying user demands. Gaming companies have to provide a smooth gaming experience to ensure customer satisfaction for millions of concurrent users. According to Tencent [2], the company which owns the most played online game - PUBG, more than half of the company's employees, around 23 thousand, work in research departments: such as marketing, business risk, continuity risk, investment, development, and testing. These researchers create many ad-hoc stream queries to analyze the most relevant streams in the company.

Figure 1 shows a sample use-case of ad-hoc stream queries. In this example, there are two input streams: 1) a stream of

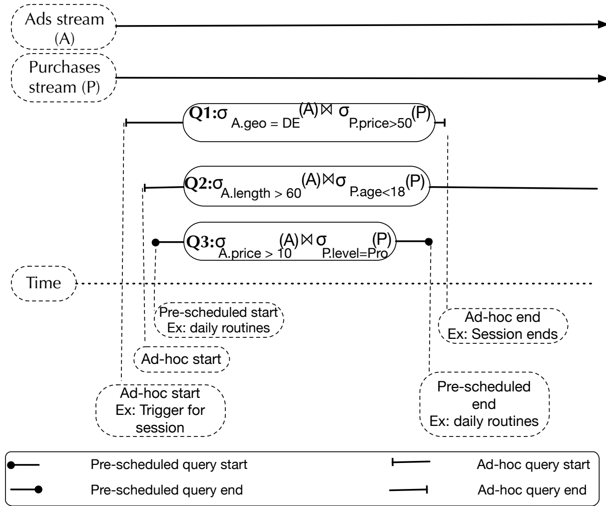


Figure 1: Ad-hoc streaming queries usecase for online gaming scenarios

advertisements, presented to players during the game, and 2) a purchases stream, which contains purchases of game packs. There are three queries in the figure, the marketing team in Europe submits a short-living query, Q1, and after getting enough information, the query is shut down. The psychology team initiates a long-living query Q2 to monitor the behavior of users under 18. Query Q3 is a session-based query created and deleted by the system to monitor the loyalty of the pro-level users. It is common to hire pro-level players to tester position, as they can reveal bugs in a game more easily.

1.2 Ad-hoc Stream Requirements

We identify three main requirements for ad-hoc stream query processing.

1. Integration: SPEs should integrate ad-hoc query support by extending stateful operators, such as window operators with different types and configurations, aggregation, join, and stateless operators, such as filters. This enables users to issue ad-hoc queries while profiting from built-in features of SPEs, such as out-of-order stream processing, event-time processing, and fault tolerance.

2. Consistency: An ad-hoc SPE executes multiple queries and serves multiple users or tenants. When removing existing queries and adding new queries to the system workload, an ad-hoc SPE must handle old and new queries in a consistent way, ensuring exactly-once semantics and the correctness of the results.

3. Performance: State-of-the-art distributed SPEs focus on maximizing the data throughput and minimizing the latency. Several well-known stream benchmarks, such as the Yahoo streaming benchmark [13], StreamBench [30], and Nexmark [48] test systems based on these metrics. Ad-hoc SPEs, in addition to the performance metrics above, need to sustain a high query throughput. The performance of such systems is boosted not only by incremental computation and resource sharing, but also by avoiding redundant computation.

1.3 AStream

We propose AStream, an ad-hoc shared-computation stream processing framework, which can handle hundreds of ad-hoc stream queries. We design AStream based on the requirements mentioned above: (1) AStream extends a wide set of components of an existing SPE, Apache Flink, but it is not tightly coupled with it. AStream supports a wide set of use-cases, windowed joins, windowed aggregations, selections, with ad-hoc query support. (2) AStream provides consistent query deletion and creation, and ensures the correctness for all running queries in the presence of ad-hoc queries; (3) Our experiments show that AStream achieves a throughput in order of hundreds of query creations per second and is able to execute in the order of thousands of concurrently running queries. AStream achieves this level of performance through a set of incremental computations and optimizations.

1.4 Sharing Limitations in State-of-the-Art Data Processing Systems

Workload sharing is a well-studied topic in the context of batch data processing systems. SharedDB [16] is one representative example for such systems. SharedDB batches user queries, creates a global query plan, and shares computation across them. We adopt some ideas from SharedDB, such as tagging tuples with query IDs to identify different subsets of (possibly computed) relations. If all stream queries are created when the system is deployed and run infinitely, meaning no ad-hocness, then this approach perfectly fits for streaming scenarios. In the presence of ad-hoc queries, however, query sharing happens among queries running on fundamentally different subsets of the data sets, determined by the creation and deletion times of each query.

Also, AStream is able to handle out-of-order stream data and to exploit and share windows of different types and configuration. AStream extends ideas from window panes [27], dynamically divides segments of time into discrete partitions at runtime, and shares overlapping parts among different queries.

We also take into consideration that aggressive work sharing among concurrent queries does not always lead to performance improvements [23]. Therefore, we compute overlapping parts of a window via dynamic programming and share if possible. Lastly, AStream is fault tolerant, all changes to query sets are deterministically replayable, which requires

that metadata modifications are deterministically woven into the streams.

1.5 Contributions and Paper Organization

The main contributions of our paper are as follows: (1) We present AStream, the first distributed ad-hoc stream processing framework. AStream is fully functional and supports a wide range of ad-hoc stream queries on shared data streams; (2) We provide exactly-once semantics and consistent query creation and deletion for ad-hoc queries; (3) We conduct an extensive experimental analysis. AStream shows comparable results to Flink in a single-query deployment and outperforms Flink by orders of magnitude in multi-query deployments.

The rest of the paper is organized as follows. Section 2 describes the system overview. We introduce implementation details in Section 3. Section 4 shows experimental evaluation. In Section 5 we discuss possible integration of AStream components to other SPEs. We discuss related work in Section 6 and conclude in Section 7.

2 SYSTEM OVERVIEW

In this section, we describe the architecture of AStream and elaborate on our data models. Figure 2 shows the general architecture of AStream. There are four main components. The *shared session* accepts queries from users and submits them to the job manager. The *shared selection*, *aggregation*, and *join operators* process queries in a shared manner. Finally, the *router* sends tuples to their associated query sinks. Our solution supports query sharing for *i*) selection, *ii*) windowed join, *iii*) windowed aggregation, and *iv*) their combination. The main assumption in this work is that operators can be shared as long as they have common upstream operators and common partitioning keys.

AStream is a framework, which can be integrated to existing SPEs as a separate layer. AStream exploits all the necessary components from an underlying SPE, such as optimizer, scheduler, network layer, and code generators.

2.1 Data Model

In the following subsection, we describe the data model of AStream.

2.1.1 *Query-set.*

AStream extends SharedDB's data model. To each tuple, we add the set of query IDs, that are potentially interested in a tuple, as an additional column. We call this column **query-set**. We assume a total numbering of queries in a query-set. We encode queries in a query-set with a bitset data structure. For example, a query-set 0010 means that the tuple is relevant only for the query with index 3 (Q3).

We compute the intersection of two query-sets through a bitwise AND operation. For any two tuples, we perform a join or aggregation if the tuples share at least one query. This way, we avoid redundant computation. Consider tuples t1, t2, t3, and t4 in Figure 3a. The bitwise AND of the query-sets of t2 and t3 returns zero, i.e., they do not share any query. However, t4 shares Q1 with t2 and t1, and Q2 with t3.

2.1.2 *Changelog.*

The above data model works well if stream queries are defined at compile time and run forever. However, for ad-hoc scenarios, this data model is not enough. For example, when the workload change occurs at time T2 in Figure 3, we observe that queries and query-sets before T2 and after T2 are different. In order to perform bitwise operations, we need a consistent query index in all query-sets so that any bitwise operations of tuples, created at different times, is correct. One way to fulfill this requirement is to assign a new index to each new query. We demonstrate this append-only approach in Figure 3b. Because Q2 is deleted, its position is permanently zero. So, the new position, 3rd position in the query-set, is assigned to the new query, Q3. The problem of this approach is that it leads to big and sparse query-sets.

AStream reuses bits of deleted queries for newly created queries in order to keep the changelog-set as compact as possible. If there is no deleted query, we allocate a new position for a new query. We use a **changelog**, a special data structure consisting of *i*) query deletion and creations meta-data and *ii*) a **changelog-set**, a bitset encoding the associated query deletions and creations. A bit in a changelog-set is set if a query in the respective position remains unchanged. A bit in the changelog-set is unset if a query is deleted or a new query is placed in the respective position. For example, in Figure 3c Q2 is deleted and Q3 is created. Because the index of Q2 is empty, Q3 is placed in this position. The associated changelog-set, 10, indicates that the first position in the query-set remains unchanged, but the second position is replaced with another query. Also, Figure 4b demonstrates the changelog-sets of the workload shown in Figure 4a. At time T5 in Figure 4a, there are two new queries (Q6 and Q7) and one deleted query (Q3). AStream allocates the index of the deleted query to Q6 and provide a new position for Q7.

By default, we use a changelog-set to indicate query changelogs between two adjacent time slots. For example, changelog-set 100 at time T2 in Figure 4b indicates the query changelog with respect to time slot T1. However, for some operations, we need to perform computations between non-adjacent time slots, such as T3 and T1.

Changelog-set in Ti with respect to Tj is a bitwise AND of all changelog-sets generated between Ti and Tj. For example, in Figure 4c for each time slot we compute their changelog-sets with respect to all previous time slots.

Equation 1 shows the dynamic programming technique to calculate CL-set[i][j], the changelog-set of time slot i, with respect to time slot j. If i is equal to j (same slot from two

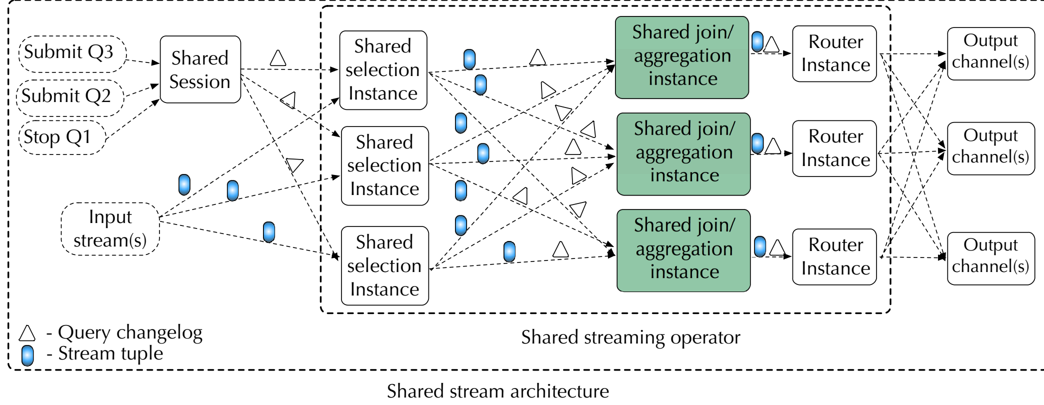
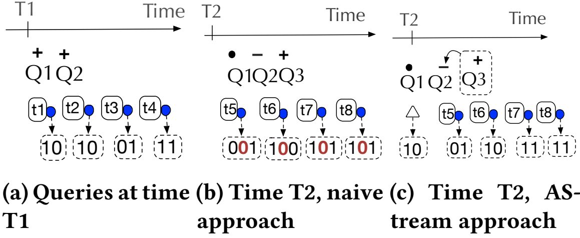


Figure 2: AStream architecture



(a) Queries at time T1 **(b) Time T2, naive approach** **(c) Time T2, AStream approach**

Figure 3: AStream and naive data model. At time T1 two new queries are submitted (Q1+, Q2+). At time T2, Q1 remains running (Q1 ·), Q2 is deleted (Q2-), Q3 is created (Q3+), and related changelog (Δ) is generated.

different streams), then there is no changelog, or 1. If i is greater than j, changelog-set is the bitwise AND of all slices from i to j.

$$\text{CL-set}[i][j] = \begin{cases} 1 & \text{if } i == j \\ \text{CL-set}[i-1][j] \,\&\, \text{CL-set}[i] & \text{if } i > j \\ \text{CL-set}[j][i] & \text{else} \end{cases} \quad (1)$$

We use changelog-sets to ensure consistency and correctness in any operation among tuples. Also, we avoid redundant computation by finding only overlapping queries among different time slots. If two time slots share queries, meaning changelog-set is non-zero, then we filter tuples by performing bitwise AND between tuples' query-sets and the changelog-set. For example, assume that we perform a join operation between the tuples created before T2 and after T2, as shown in Figure 3a. t5 is filtered, because the bitwise AND of the t5 query-set, 01, and the changelog-set, 10, is zero. As another example, joining t7 and t4 would result in the tuple with query-set 10 (10&11&11), meaning the resulting tuple matches Q1.

3 IMPLEMENTATION DETAILS

In this section, we first explain the implementation of ad-hoc operators (Section 3.1) and optimization techniques adopted by AStream (Section 3.2). We elaborate on fault tolerance in Section 3.3 and quality of service (QoS) features in Section 3.4.

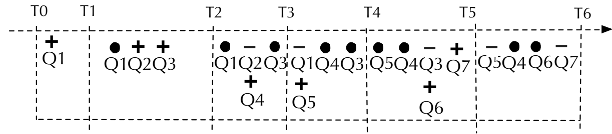
3.1 Ad-hoc Operators

Each operator in AStream keeps a list of active queries. Once active queries are updated via changelog, operators change their computation logic accordingly.

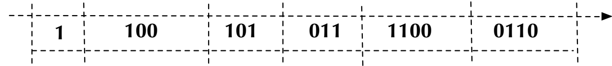
3.1.1 Shared session. The *shared session* is a client module of AStream. The shared session batches user query requests and generates a changelog. A changelog is generated for every batch-size (number of user requests) or once the maximum timeout is reached. If there is no user request, no changelog is generated.

3.1.2 Shared selection. The shared selection operator computes the query-set for each tuple and appends the resulting query-set to the tuple as a separate column. The shared selection maintains the set of active queries. It updates the set once it receives a changelog.

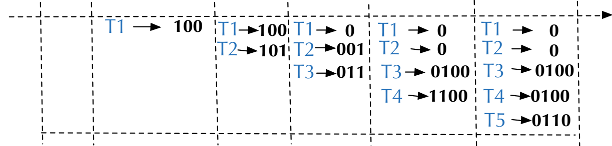
3.1.3 Window slicing. AStream supports time- and session-based windows with different characteristics (e.g., length, slide, gap). For queries involving window operators, such as windowed aggregation and windowed join, AStream divides overlapping windows into disjoint **slices**. It performs operations among overlapping slices once and reuses the result for multiple queries. The lengths of slices in Figure 4e are determined at runtime based on the created and deleted



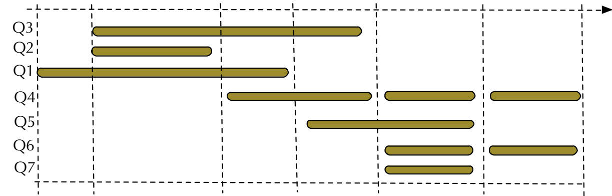
(a) Queries that remain unchanged (·), new created (+), and deleted (-) over time



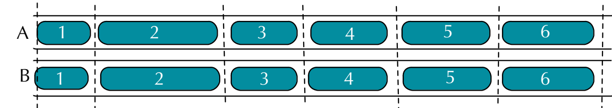
(b) Changelog-sets of the respective time slots



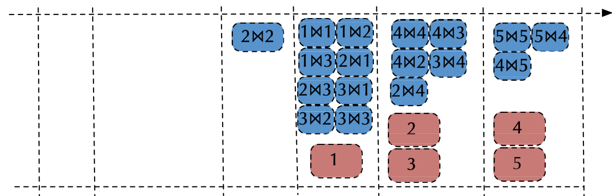
(c) Changelog-sets with respect to previous time slots



(d) Query windows



(e) Dynamically created stream (A and B) slices inside a join operator



(f) Actions taken in different time slots. Blue boxes indicate join operation between two slices and red boxes show deleted slices.

Figure 4: End-to-end ad-hoc query example. Ad-hoc queries (Figure 4a) with various window configurations (Figure 4d) are submitted. Their related changelog-sets are generated in Figure 4b. The join operator generates changelog-set table (Figure 4c) and performs join operation (Figure 4f) with dynamically generated slices (Figure 4e). All figures share the same x-axis.

queries shown in Figure 4d. Once a query changelog arrives, its changelog-set is assigned to the corresponding window slice. Also, the set of running queries inside the shared join operator gets updated.

3.1.4 Shared join. AStream executes join operations incrementally by joining slices and combining intermediate results. It joins overlapping slices once and reuses the intermediate results. For each slice, AStream keeps a computation history. Based on this information, it avoids unnecessary computation among slices and performs delta query processing. Consider the join operation in Figure 4f. At time T3, Q2 triggers and join results are emitted. At time T4, Q1 is evaluated. Note that AStream avoids joining already joined slices (slice-2 ⋈ slice-2). Also, the first slice is deleted, as it is no longer needed. Similarly, at time T5, AStream joins slices once and reuses them for multiple overlapping query windows (Q4, Q5, Q6, Q7).

We join two slices as follows. We group tuples in each slice by their query-sets. First, we check the query-set groups, e.g., G1 in slice 1 and G2 in slice 2. We join tuples residing in G1 and G2, if group IDs, which are query-sets, share at least one query. For example, if G1=010 and G2=∗0∗, then tuples residing in these groups are never joined.

Grouping tuples inside slices enables sharing tuples on-the-fly. The disadvantage of this method is that the number of possible tuple groups increases exponentially with the number of queries. In early experiments, we noticed that for more than ten concurrent queries, storing tuples as a list is more efficient than storing them inside groups.

For switching between a group and a list data structure, we use the following heuristic. As the number of queries increases, we monitor the average size of tuple groups inside slices. If the average is less than two, meaning most of the tuple groups contain only a single tuple, then we switch to a list data structure.

3.1.5 Shared aggregation. The shared aggregation works similar to the shared join. One difference is that the shared join is a binary stream operator (has two input streams), but the shared aggregation is a unary stream operator. Another difference is that the output of the shared join operator can be shared with other downstream join operators (shared n-ary joins); however, for shared aggregations, it is not possible to share the output with downstream operators.

In a shared aggregation, each window slice keeps intermediate aggregation results for all active queries. Instead of materializing input tuples, we update the query intermediate aggregation results for each new tuple. Then, we discard the tuple. For example, a tuple with the query code 101 is aggregated with the Q1 and Q3 intermediate aggregation results and discarded afterward. Aggregation between two different slices is also performed in a similar way.

3.1.6 *Router.*

The router is another component of AStream. The routing information for each tuple is encoded in its query-set. The router sends each tuple to either query output channels or to downstream operators.

3.2 Optimizations

AStream uses several optimizations to speed up query processing.

3.2.1 *Incremental query processing.*

Incremental query processing is a core feature of AStream. As shown in Sections 3.1.4 and 3.1.5, AStream computes both ad-hoc stream aggregations and joins in an incremental manner.

3.2.2 *Data copy and shuffling.*

AStream avoids data copy in all its components except in the router. A router avoids data copy if the downstream operator is a shared join or aggregation operator. It performs data copy only if the downstream operator is a sink operator, in which the router has to ship results to different query channels.

AStream also avoids redundant data shuffling by encoding a query-set for each tuple. When running a single query, this has some performance overhead, but for multiple queries, the overhead is outweighed by the performance improvements. The shared aggregation and join operators avoid data copy inside slices. Each tuple is saved only once inside a slice.

3.2.3 *Memory efficient dynamic slice data structure.*

The shared join operator adapts the data structure based on the workload. If the number of active queries exceeds a threshold, the shared session sends a marker to downstream operators. Once the marker is received, the shared join operator changes the data structure of all slices and resumes its computation.

3.3 Exactly-Once Semantics

Exactly-once semantics for SPEs ensure that every input tuple is only processed once, even under failures. Operators in AStream are exactly-once, as long as the underlying distributed streaming architecture supports exactly-once semantics, as systems like Kafka-streams [42], Spark Structured Streaming [5], and Apache Flink [11] do. AStream requires that both tuples and changelog markers and the state of shared operators are deterministically reproducible by logging the input stream and checkpointing [10].

AStream is deterministic because all its distributed components are deterministic and they are based on event-time semantics. Event-time is the time at which an event was produced; e.g., the time an ad is clicked (for tuples) or the time a query is deleted (for changelogs). Event-time semantics ensure correctness on out-of-order events or during replays of data because the notion of time depends on the data, not on the system clock. In event-time stream processing, tuples are assigned to windows based on their event-time [4, 28]. In the case of a failure, a replayed event is assigned to the same window ID, as the window ID computation is also deterministic [4, 28]. Our slicing technique (Figure 4e) is also deterministic. The length of slices depends on changelogs. The changelogs also use event-time, which is the time at which query changes were performed by users.

3.4 QoS

Controlling the performance impact of a new query on existing queries is essential to ensure the quality of service in a multi-query environment. In ad-hoc stream workloads, QoS should be ensured in many ways, such as individual query throughput, overall query throughput, data throughput, data latency, and query deployment latency. For example, for data latency, we extend the latency metric implementation of Flink [3]. To be more specific, in the sink operator of every query, we periodically select a random tuple and measure the end-to-end latency. The latency results are collected in the job manager. Also, we show in our experiments (Section 4.8) the impact of newly created or deleted queries on existing queries. AStream is capable of providing the abovementioned metrics to an external component. If measurements for a particular metric are beyond acceptable boundaries, new resources can be added; however, elastic scaling is out of the scope of this paper.

4 EXPERIMENTS

4.1 Experimental Design

To evaluate AStream, we simulate a multi-tenant environment with ad-hoc queries. We use a parallel and distributed driver and conduct experiments with two systems under test (SUT): Apache Flink [11] (v 1.5.2) and AStream, which we implement on top of Flink.

As shown in Figure 5, our driver maintains two FIFO queues. One queue stores user requests, i.e., query creation or query deletions. Periodically, the driver pops user requests from the FIFO queue, sends them to a SUT, and waits for the acknowledge message (ACK) from the SUT. The driver submits the next set of user requests to the SUT if the SUT ACKs the previous batch. This way, we implement a backpressure mechanism for user query requests. The longer the user request stays in the queue, the higher is its deployment latency.

Another queue stores input tuples. Data generators generate tuples and put them into the queue. The driver pulls tuples from the queue and sends them to the SUT. The longer the tuple stays in the queue, the higher is its event-time latency. Event-time latency is the duration between the tuple's event-time and the tuple's emission time from SUT.

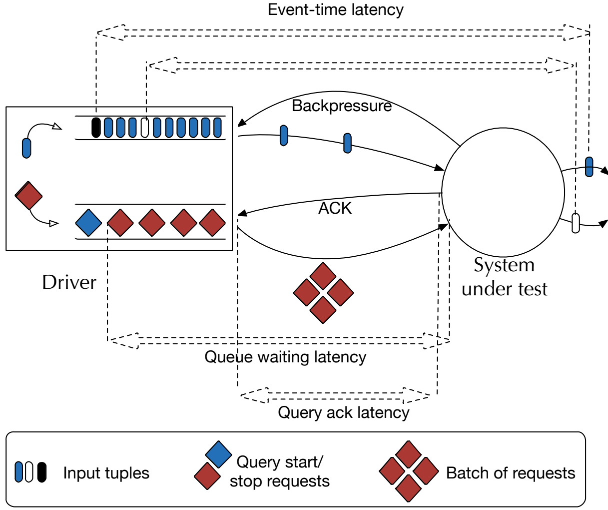
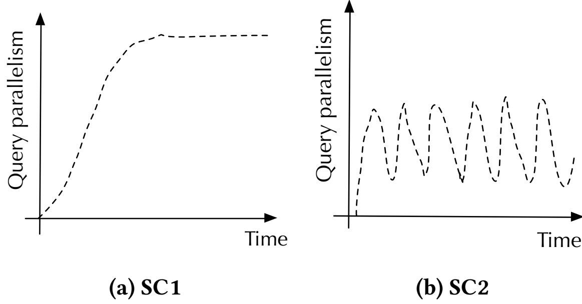


Figure 5: Design of the driver for the experimental analysis



(a) SC1 **(b) SC2**

Figure 6: Two scenarios for ad-hoc query processing environments

4.2 Generators

4.2.1 Data generation. Each generated input tuple has 6 fields: a *key* field and an array of size 5, named *fields*. Each subsequent tuple is generated with key in the form $key \leftarrow key{+}{+}\ \%\ key_{max}$. This way, we balance the data distribution among different partitions. The other fields are generated in a random manner, $fields[i] \leftarrow random(0, fields_{max})$.

4.2.2 Selection predicate generation. To generate a selection predicate, we select a random *field* of a tuple ($field[i]$), generate a random number (VAL), select a random binary operator: <, >, ==, ≤, or ≥ (o_i), and combine them to a selection predicate ($o_i(field[i], VAL)$).

4.2.3 Join and aggregation query generation. The join and aggregation query generation consists of two parts: selection predicate generation (see above) and window generation.

```
1 SELECT *
2 FROM A, B [RANGE [VAL1]] [SLICE [VAL2]]
3 WHERE A.KEY = B.KEY AND
4       A.[VAL5] [=|>|<|>=|<=] [VAL3] AND
5       B.[VAL6] [=|>|<|>=|<=] [VAL4]
```

Figure 7: Join query template. VALn is a random number, VAL5 and VAL6 are less than $|fields|$=5

```
1 SELECT SUM(A.FIELD1)
2 FROM A [RANGE [VAL1]] [SLICE [VAL2]]
3 WHERE A.[VAL4] [=|>|<|>=|<=] [VAL3]
4 GROUPBY A.KEY
```

Figure 8: Aggregation query template. VALn is a random number, VAL4 is less than $|fields|$=5

We generate window length as $random(1, \text{window}_{max})$ and slide as $random(1, \text{length})$. For session windows, window length and slide are not needed. Figures 7 and 8 show the query templates for join and aggregation queries. Line 4-5 in Figure 7 and Line 3 in Figure 8 show selection predicates. For join queries, both input streams have different selection predicates.

4.3 Metrics

Basic metrics to evaluate SPEs are event-time latency and sustainable throughput [24]. In addition to these, we propose several metrics for ad-hoc streaming environments. **Query deployment latency** is the time duration between a user request to create or delete a query and the actual query start time. For data throughput evaluations, there are two main metrics to consider. **Slowest data throughput** is the minimum sustainable throughput among active streaming queries in an ad-hoc environment. This metric is useful for a service or cloud owner, to ensure minimum QoS requirements. **Overall data throughput** is the sum of throughputs of all active queries. **Query throughput** is the highest load of query traffic (query deletion and creation) a system can handle with sustainable query deployment latency and input throughput.

4.4 Setup

We conduct experiments in 4- and 8-node cluster configurations. Each node has 16-core Intel Xeon CPU (E5620 2.40GHz) and 48 GB main memory. The data generator generates data with 1000 distinct keys with a uniform distribution. If a SUT throws an exception or error while stopping or starting a streaming job or processing submitted queries in an ad-hoc

manner (possibly with high frequency), then we consider this as a failure, meaning the SUT cannot sustain the given workload. We repeat our experiments three times and let it run for thousand seconds. For a changelog generation, we tried several combinations of batch-size and maximum timeout configurations. We configure the batch-size to be one hundred and maximum timeout to be one second, as these configurations are the most suitable for our workloads.

4.4.1 Workloads. In Figure 6, we show two workload scenarios to evaluate AStream. The main characteristics of the first workload scenario (SC1) are *i*) many users, which leads to many parallel queries, *ii*) few queries that are stopped or changed, resulting in mostly long-running streaming jobs, and *iii*) no new ad-hoc queries after some time. The main characteristics of the second workload scenario (SC2) are *i*) high query throughput, i.e., many queries are created or deleted *ii*) low query parallelism, and *iii*) short-running queries.

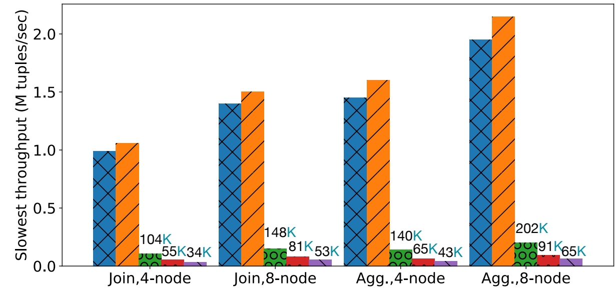
4.5 Workload Scenario 1

Figure 9 shows data throughput for SC1, 4- and 8-node cluster configurations. n q/s m qp indicates n queries per second until m active queries. For a single-query deployment in Figure 9a, Flink outperforms AStream. Although query-set generation and bitset operations come with a cost, AStream single-query deployment still has a comparable performance to Flink. Flink cannot sustain ad-hoc workloads in Figure 9. In each run, it either throws an exception or exhibits very high latency.

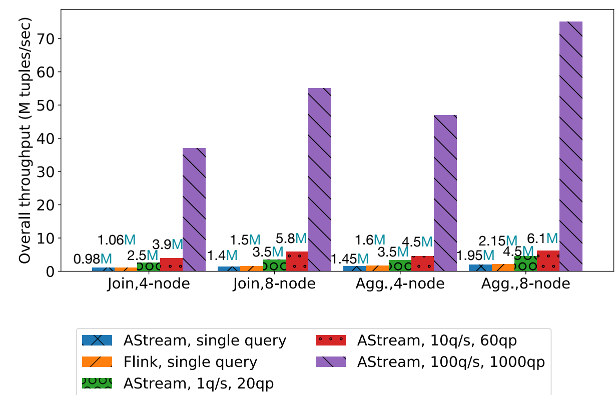
In Figure 9b there is a sharp increase in the overall throughput of served queries. AStream achieves a better throughput with more ad-hoc queries. However, this performance increase comes with a cost. In Figure 9a we see that there is a decrease in the slowest throughput because the number of served queries increase from one query to thousand queries.

After the initial decrease in Figure 9a, we observe that the throughput decrease remains steady. The main reason is that, as the number of queries increases, the probability of sharing a tuple among different queries also increases. As a result, the slowest data throughput decreases less with more queries.

We observe several differences between join and aggregation query performances in Figure 9. First, data throughput for join queries is less than for aggregation queries, because joins are computationally more expensive than aggregation in our setup. Second, the performance gap between Flink and AStream is larger for aggregation queries than for join queries. The main reason is that Flink has a built-in support for on-the-fly and incremental aggregation. In contrast, windowed join queries in Flink lack those features.

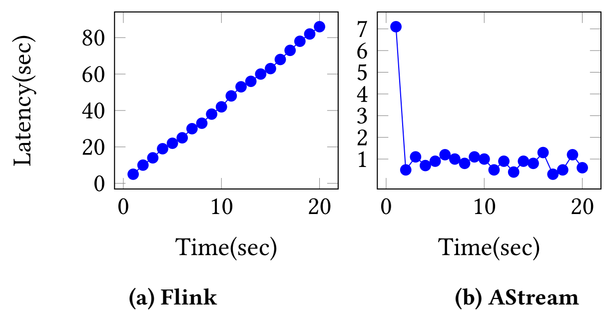


(a) Slowest data throughput



(b) Overall data throughput

Figure 9: Slowest and overall data throughputs for SC1, 4- and 8-node cluster configurations. n q/s m qp **indicates** n **queries per second until** m **query parallelism**



(a) Flink **(b) AStream**

Figure 10: Query deployment latency, one query per second, up to 20 queries

Figure 11 shows the query deployment latency for SC1. Changelog batch-size also has a contribution to the overall latency. For example, 1 q/s, 20 qp has more query deployment

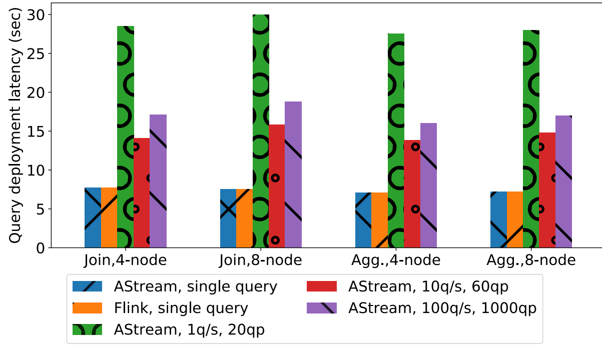


Figure 11: Ad-hoc query deployment latencies for SC1

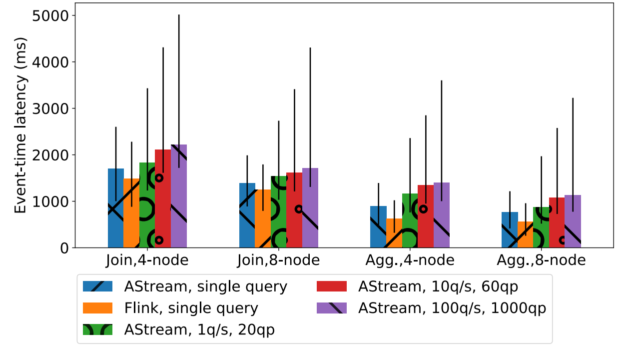


Figure 12: Average event-time latency for SC1

latency than 100q/s, 1000qp, as the former has 20 ($\frac{20}{1}$) different query changelog generations, while the latter contains 10 ($\frac{1000}{100}$) different query changelog generations.

Figure 10 shows query deployment latency for SC1 (1 q/s, 20 qp). Because Flink cannot sustain this workload, query deployment latency keeps increasing, which is why we do not show this case in Figure 9. The longer the query stays in the queue waiting for ACK, the higher is its deployment latency. For example, the sum of all query deployment latencies for Flink is 910 seconds. In general, query deployment latency is already high and will be a bottleneck in a multi-tenant environment.

In Figure 10 AStream initially exhibits high query deployment latency, because the first query deployment also involves the physical deployment of operators to the cluster nodes, which is time-consuming. Even for batch ad-hoc data processing systems with a dedicated scheduler and optimizer, such as DataPath [7], the first deployment of physical operators is time-consuming. AStream avoids deploying a new streaming topology for each query. Instead, it creates and deletes user queries on-the-fly without affecting the running topology.

Figure 12 presents the average event-time latency for streaming tuples. We note that event-time latency for shared aggregation queries is lower than shared join queries because joins are computationally more expensive than aggregations. Throughout our experiments, we observed Flink's event-time latency for ad-hoc workloads to be higher than eight seconds. As experiments continued, the latency kept increasing, which means the system cannot sustain the given workload.

In Figure 12, we notice that event-time latency increases for higher query parallelism for AStream. However, the given latency measurements for AStream are sustainable. Also, the measurements do not exhibit continuous backpressure.

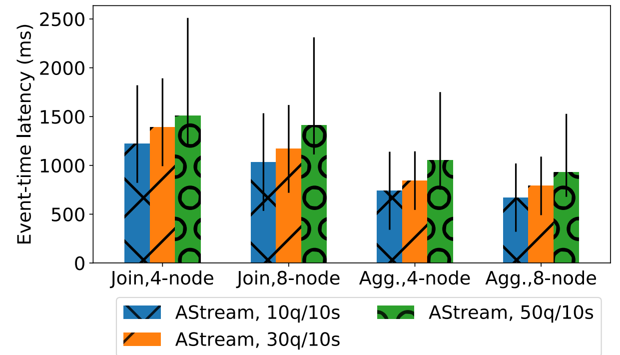


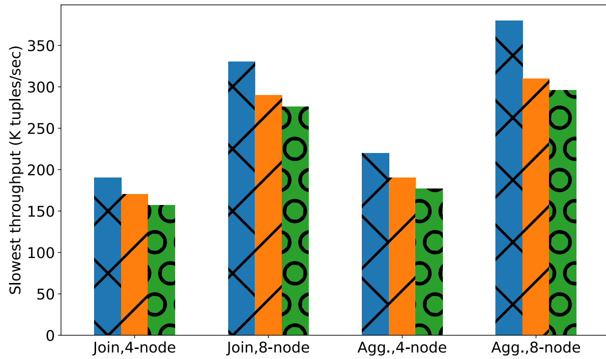
Figure 13: Average event-time latency for SC2. nq/ms means n queries are submitted and stopped every m seconds.

4.6 Workload Scenario 2

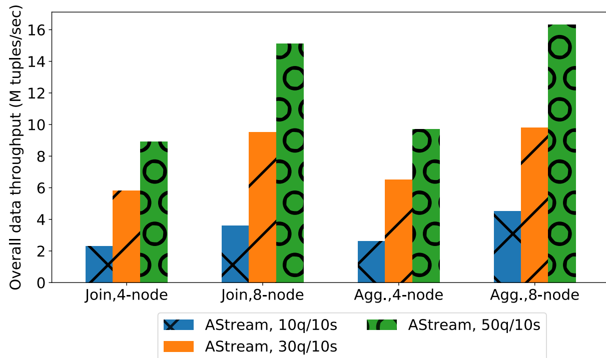
As explained above, SC2 features a more fluctuating workload than SC1. In this case, a robust data processing is needed to sustain possible churn in the workload.

Figure 13 shows the average event-time latency for SC2. We notice that event-time latency in SC2 is lower than SC1 (Figure 12). The reason is that in SC2 the query workload is highly changing, but does not increase continuously. So, the majority of the queries running in SC2 are short-running queries.

Figure 14 shows data throughput for SC2. nq/ms indicates n queries are submitted and stopped every m seconds. Although SC2 exhibits high query fluctuations, the slowest data throughput in SC2 is higher than the one in SC1 (Figure 9), which means that AStream works better in more fluctuating workloads. The main reason is that the workload in SC2 is more fluctuating, queries are short-running, and constantly changing; as a result, i) the overall number of active queries



(a) Data throughput of the slowest query



(b) Overall data throughput

Figure 14: Input data throughput for SC2

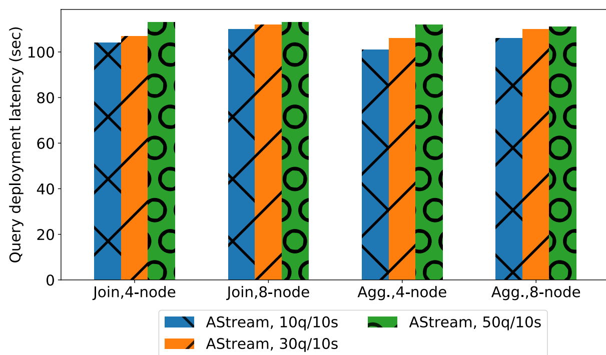


Figure 15: Ad-hoc query deployment latency for SC2

is less than in SC1 and *ii*) bitset size is less than in SC1. In our experiments, we observe that Flink cannot sustain ad-hoc workloads. For example, for the setup 10q/10s, the input data throughput of AStream was at least 10× higher than Flink's, before we stopped the experiment.

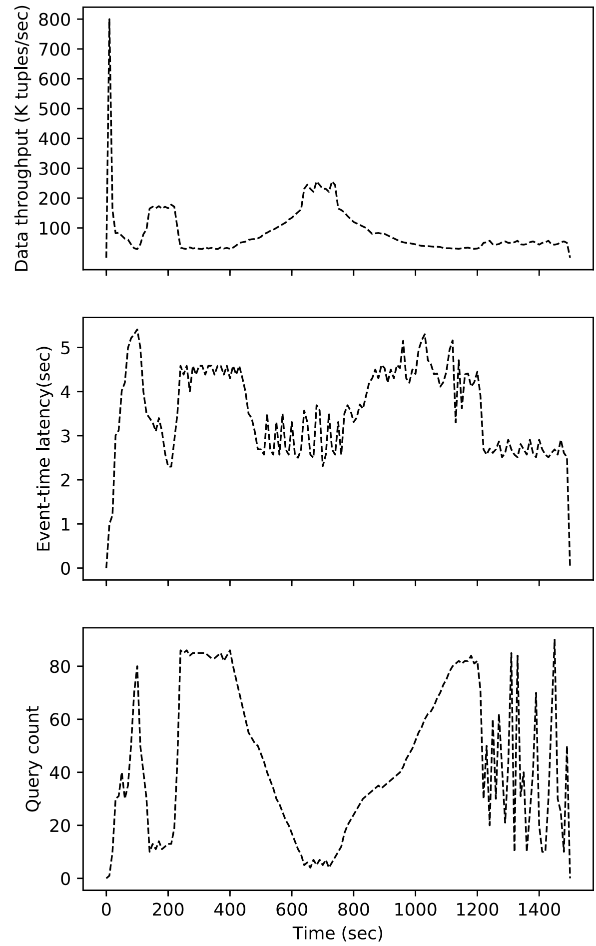


Figure 16: Slowest data throughput (upper), event-time latency (middle), and query count graphs (bottom) for complex ad-hoc queries, with the same x axis values

Figure 15 shows the ad-hoc query deployment latency for SC2. We run this experiment for thousand seconds. When we compare the query deployment latency of SC1 and SC2, the latter is significantly higher. The reason is that in SC2, we continuously create and delete queries, while in SC1 we create queries up to predefined query parallelism. Continuously creating and deleting queries results in continuous query changelog generation.

4.7 Complex Queries

In this section, we conduct experiments with complex queries, consisting of multiple joins and an aggregation. We generate

complex queries by randomly pipelining a selection predicate, n-ary windowed joins, where $1 \leq n \leq 5$, and a windowed aggregation operator. Any complex query involves at least one selection predicate, one windowed join query, and one windowed aggregation query.

Figure 16 shows the input data throughput (upper), input latency (middle), and query count graphs (bottom) for complex concurrent queries. We test three cases in this experiment. First, we perform a sharp query throughput increase at timestamps 50 and 200. Second, we gradually decrease query throughput and gradually increase, from time 410 to 1140. Third, we fluctuate query throughput after time 1200.

When we increase query throughput sharply, we notice that the input data latency stays relatively stable. The reason is that we adopt shared streaming operators and do not change the query execution plan, which would cause high latencies. The slowest data throughput drops as we increase query throughput. Also, we notice that in case of fluctuations in query throughput, both slowest data throughput and event-time latency remains stable.

4.8 Sharing overhead

Figure 17 shows slowest data throughput for different query parallelism. Similar to Figure 9, we note that slowest throughput decreases as query parallelism increases. As the number of queries increases, sharing a tuple among different queries is more probable; as a result, the slope of the figure decreases slowly with increasing query parallelism.

Adding ad-hoc support to a SPE incurs an overhead. We measure this overhead by comparing AStream with Flink. In our experiments, we see that Flink cannot sustain ad-hoc workloads. Conducting ad-hoc experiments with Flink resulted either in an exception or in ever-increasing latency. The main reason is that Flink is not designed for ad-hoc workloads. Therefore, we can only see the overhead of sharing between AStream and original Flink in a single query setup. As shown in Figure 9 AStream throughput is 9 % less than Flink's throughput in the worst case (from 2.15M/sec to 1.95M/sec, windowed aggregation, 8 nodes) because of the sharing overhead.

We also measure the individual cost of AStream's components. The cost mainly involves generating query-sets, bitset operations, and data copy in the router to ship resulting tuples to different query channels. Figure 18a shows an overhead proportion of AStream components in SC1. With low query-parallelism, the proportion is roughly equal. As the number of concurrent queries increases, data copy becomes a dominant overhead. Data copy in the router operator is inevitable as AStream has to send resulting tuples to physically different query channels. Figure 18b shows the overhead of AStream (sum of its components). We can see that with more

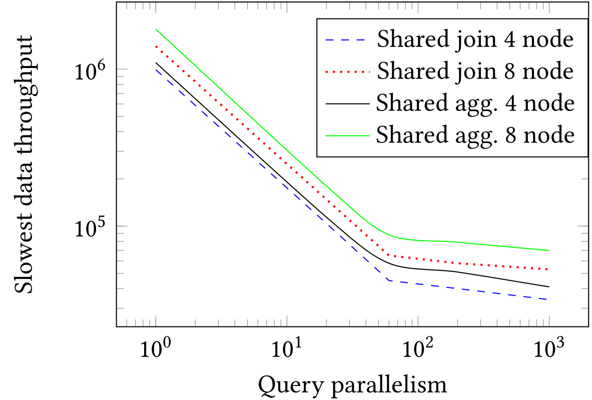


Figure 17: Input data throughput for different query parallelism in SC1

queries, the overhead of AStream is below 2%. The main reason is that with more queries the probability of sharing increases.

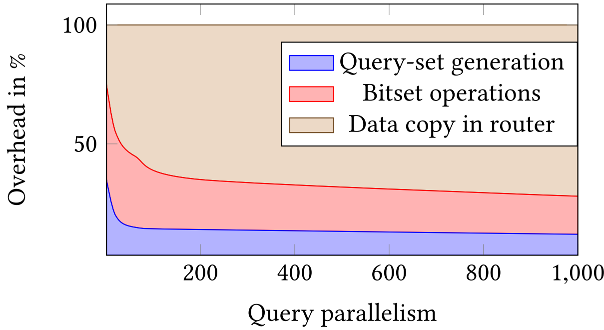
Figure 19 shows the effect of executing of ad-hoc join queries to the performance of existing ones. We perform experiments in a 4-node cluster. We observe that with many running queries, adding ad-hoc queries does not affect their performance much in both scenarios (SC1 and SC2). Also, with a small number of running queries, SC1 is more susceptible to performance decrease than SC2. The main reason is that in SC1 long-running queries are created. In SC2, on the other hand, queries are created and deleted periodically. As a result, the overall number of queries and the size of query-sets is less in SC2.

Figure 20 shows the scalability of AStream queries with different cluster configurations. In this experiment, we keep the data throughput constant for all cluster configurations. We can see that the number of ad-hoc queries scales with more nodes. We also observe that SC2 scales better than SC1. The main reason is, as mentioned above, in SC2 queries are periodically created and removed, which results in less number of active queries and less bitwise operations.

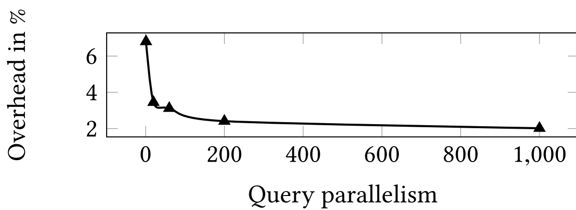
4.9 Discussion

AStream supports high data and query throughput within regular event-time and deployment latency boundaries. With thousand concurrent queries, AStream achieves more than 70 millions tuples per second data throughput (Figure 9). Our baseline, Apache Flink is not able to run twenty concurrent queries.

AStream also supports high query throughput. In SC1 AStream is able to start hundred queries in a single changelog and in SC2 it is able to start 50 queries and delete 50 queries in a single changelog.



(a) Overhead proportion of AStream componnets



(b) Overhead proportion of AStream

Figure 18: Overhead proportion of AStream and its main components in SC1, 4-node cluster

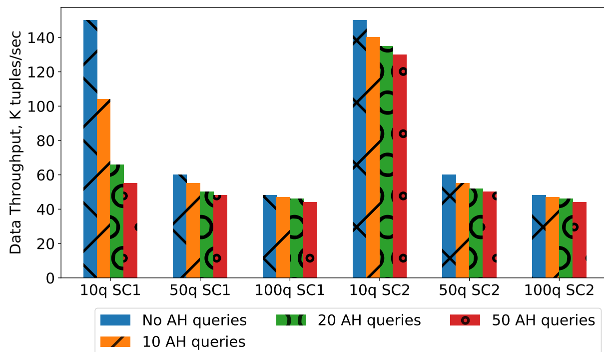


Figure 19: Effect of new ad-hoc join queries on existing long-running queries. x-axis shows the number of long-running queries and the workload scenario

AStream processes 70 millions tuples per second (Figure 9, 100q/s 1000qp) with 1.2 second average event-time latency (Figure 12). For SC2, it handles fluctuating ad-hoc queries (creating and deleting 50 queries per 10 seconds) in less than one second event-time latency.

In our experiments, we see that the deployment latency is a major bottleneck for Flink (Figure 11). AStream, however, has a very low deployment latency, in the order of milliseconds per query.

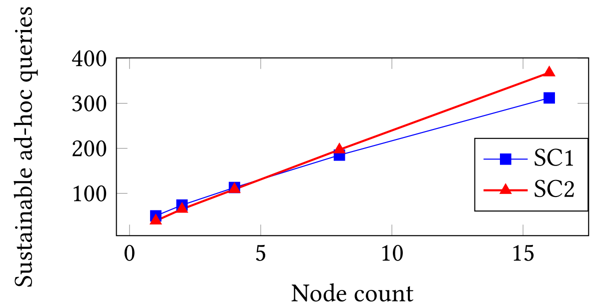


Figure 20: Scalability with the number of queries

Integrating AStream has some overhead, which is already outweighed by the efficiency improvement with two concurrent queries. The overhead for a single query is in the order of 10% in the worst case.

5 INTEGRATION

All AStream components can be integrated into an SPE by extending its components. The shared session is an extension of a remote client module, which accepts user requests and executes the requests in a remote cluster. We modify this module to generate the changelog data structure for handling user requests. The shared selection and router are mapper operators with a state. The state is updated for every changelog. We implement window operators by customizing triggers, evictors, and window functions [15] to be dynamic and updatable at runtime. Any SPE providing low-level window APIs can integrate shared windowed join and aggregation operators.

We build our prototype of AStream on top of Apache Flink; however, our design is not tightly coupled with the underlying SPE. Below, we briefly sketch possible integrations with Trill [12] and Spark Structured Streaming [5]. Unlike previous work, such as DataPath [7], which is a complete system with dedicated query optimizer and scheduler, AStream is a framework that can be used with different SPEs to enable ad-hoc query processing.

Because AStream is a pluggable component of an underlying SPE, it also supports optimizations for data representation and code generation. For example, Trill uses streaming batched-columnar data representation with a new dynamic compilation-based system architecture [12]. To support hybrid-columnar processing with AStream, one can integrate query-set field as a separate column in Trill's hybrid columnar data representation or merge the field with the key field into the single field. Trill's stateful operators can integrate AStream. AStream ensures consistency and correctness; however, the underlying processing semantics, e.g., grouping or join algorithms, can be performed by another

component of an underlying SPE. To integrate changelogs, Trill's punctuation data structure can be extended to include additional query meta-data.

Spark Structured Streaming [5] is an example of a distributed SPE with highly optimized code generation using the catalyst optimizer [6]. In the continuous processing mode, one can adopt a very similar implementation that we used in Flink. In the mini-batch processing mode, Spark Structured Streaming adopts bulk-synchronous-processing semantics, handling batch-size dynamically at runtime. In this setup, all workers can be informed about changelogs at the synchronization phase to use AStream components.

6 RELATED WORK

Below, we explore several research directions in database management and stream processing related to our work and discuss the similarities and differences.

6.1 Query-at-a-Time Processing

Query-at-a-time SPEs feature mature and widely accepted optimizers [6, 20]. These systems inherit methods adopted in traditional relational query optimization [36, 43]. However, traditional query optimizers lack optimizing ad-hoc queries submitted in real time, as the solution space is non-convex and the complexity of query optimization in many cases is exponential [17]. AStream adopts shared operators which can handle multiple user queries and share them if necessary. Thus, we avoid the optimization and deployment cost of queries created and deleted in runtime.

6.2 Stream Multi-Query Optimization

Multi-query optimization is one of the fundamental methods to share computation among queries [44]. One drawback of this method is its worst-case complexity (NP-hard) [14]. As the number of stream queries increases, finding shared parts among queries becomes costly. Another drawback of traditional multi-query optimization is that all queries should be known at compile time. Also, multi-query optimization has limited ability to share queries with blackbox selection predicates, such as with user-defined functions.

Seshadri et al. show the potential limitations of streaming multi-query optimization in a distributed streaming environment [45]. Hong et al. propose rule-based streaming multi-query optimization [21]. Dobra et al. adopt sketch-based techniques to find approximate results for streaming multi-query optimization [14].

The above work assumes prior knowledge (at compile time) of streaming queries and adopts optimizers that are not able to react to ad-hoc queries at runtime. AStream, on the other hand, has no prior knowledge about a workload and can react to ad-hoc queries.

6.3 Adaptive Query Optimization

Adaptive query optimization is another method to handle efficient execution of multiple streaming queries. Although this methodology might work for a small number of input queries, with high concurrent workloads re-optimization is a limiting factor.

Madden et al. develop an adaptive stream query sharing system [31]. The system works on a single node and is built on top of Eddies [8]. Ives et al. propose an adaptive query processing framework which adjusts processing based on I/O delays and data flow rates, and shares the data across multiple queries [22]. Raman et al. propose STeM, a shared materialization point for join queries [38].

Unlike the work above, AStream is not limited to binary joins but also supports n-way joins, aggregation, selection predicates, and their combination. Moreover, AStream is designed to be executed in a distributed environment.

Drizzle is a distributed, fast, and adaptable stream processing framework [50]. It adopts the bulk-synchronous processing model. Chi is flexible stream processing framework built for tuple-at-a-time systems [32]. By design, these systems assume that workload and cluster properties change rarely, as changing the query execution plan is costly. AStream, however, supports highly fluctuating workloads and performs query creation and deletion on-the-fly, without stopping the topology.

6.4 Batch ad-hoc Query Processing Systems

SharedDB [16] proposes query sharing for OLTP, OLAP, and mixed workloads via shared operators. QPipe, adopts on-demand simultaneous pipelining, dynamically sharing an operator's output simultaneously to parent nodes [19]. AStream is conceptually similar to SharedDB, as it also uses shared operators. Psaroudakis et al. compare the two main query sharing approaches: simultaneous pipelining, such as QPipe, and global query plan, such as SharedDB [37].

CJoin [9] and DataPath [7] propose a join operator that supports concurrent queries in data warehouses. MQJoin efficiently uses main memory bandwidth and multi-core architectures and minimizes redundant work across concurrent join queries [33, 34]. Tell has a shared-data architecture, which decouples transactional query processing and data storage into two layers to enable elasticity and workload flexibility [29].

To support multiple queries, scan sharing is a common technique. For example, Blink [39] and Crescando [49] share disk and memory bandwidth. Similarly, CoScan performs cooperative scan sharing in the cloud merging pig programs from multiple users at runtime [53]. Also, MonetDB [55]

and DB2 UDB [26] perform cooperative scans and maximize buffer-pool utilization across queries.

BatchDB adopts similar batching ideas with Crescando and SharedDB [35]. However, BatchDB isolates batching of OLAP queries from the updates propagated by the primary OLTP replica. OLTPShare specializes in sharing concurrent OLTP workloads [41].

Although there are some similarities between AStream and batch ad-hoc query processing systems, such as bitsets (CJoin, DataPath, SharedDB, MQJoin), common upstream operators and common partitioning key (SharedDB and DataPath), query batching (SharedDB, BatchDB, OLTPShare), redundant computation filtering (MQJoin), scan sharing (Crescando, Blink, MonetDB, DB2 UDB), supporting complex queries with high consistency (Tell), and high throughput concurrent query processing (MQJoin), there are also conceptual differences. One difference between ad-hoc streaming and ad-hoc batch query processing is that the former features windows with different configurations. Also, ad-hoc batch data processing systems feature only ad-hoc query creation. AStream supports ad-hoc query creation and deletion in a consistent manner. Finally, AStream also adopts techniques to avoid redundant computation among queries.

6.5 Streaming query sharing

Wang et. al propose sharing windowed join operators for CPU intensive and memory-intensive workloads [52]. The approach assumes that all input queries are known at compile time. Our approach, on the other hand, supports query creation and deletion in an ad-hoc manner. Hammad et al. propose shared join operator for multiple streaming queries [18]. Similar to the previous work, in this work, the main assumption is that input queries are known at compile time. Another limitation is that this work adopts selection pull-up approach, which might result in *i*) high bookkeeping cost of resulting joined tuples and *ii*) intensive consumption of CPU and memory.

Krishnamurthy et al. propose on-the-fly query sharing technique for windowed aggregation queries [25]. The authors partition tuples into fragments and perform incremental aggregation. Traub et.al propose general stream aggregation computation which automatically adapts to workload characteristics [47]. Although we also adopt a similar techniques to compute results incrementally, our solution is not limited to windowed aggregations. AStream supports windowed queries consisting of selection, aggregation, join, and their combinations.

Li et. al propose window ID representation of events and panes [28], and sharing computation among panes [27]. The core difference between panes and AStream window sharing technique is that the former computes overlapping parts of a window in compile-time, while the latter computes them in runtime based on ad-hoc queries and their corresponding windows.

7 CONCLUSION

In this paper, we present AStream, the first distributed SPE for ad-hoc stream workloads. We show that current state-of-the-art SPEs are not able to process ad-hoc stream workloads. We observe in our experiments that not only data latency and throughput, but also query deployment latency and throughput are bottlenecks.

AStream is a layer on top of Flink, which extends existing SPE components and supports the majority of streaming use-cases. AStream ensures easy integration, correctness, consistency, and high performance (query and data throughput) in ad-hoc query workloads.

In future work, we plan to extend AStream with a cost-based optimizer and adaptive query processing techniques. Based on sharing statistics among queries collected at runtime, a more optimal query plan can be generated by grouping similar queries.

Acknowledgements: This work has been supported through a grant by the German Ministry for Education and Research as Berlin Big Data Center BBDC 2 (funding mark 01IS18025A).

REFERENCES

[1] 2018. Apache Apex. https://apex.apache.org/. [Online; accessed 19-July-2018].

[2] 2018. Tencent Multinational conglomerate company. https://www.tencent.com/. [Online; accessed 19-July-2018].

[3] 2019. Apache Flink Latency. https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/metrics.html/. [Online; accessed 4-Feb-2019].

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 601–613.

[6] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[7] Subi Arumugam, Alin Dobra, Christopher M Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 519–530.

[8] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, Vol. 29. ACM, 261–272.
[9] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment* 2, 1 (2009), 277–288.
[10] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[12] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.
[13] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1789–1792.
[14] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2004. Sketch-based multi-query processing over data streams. In *International Conference on Extending Database Technology*. Springer, 551–568.
[15] Buğra Gedik. 2014. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience* 44, 9 (2014), 1105–1128.
[16] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment* 5, 6 (2012), 526–537.
[17] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared workload optimization. *Proceedings of the VLDB Endowment* 7, 6 (2014), 429–440.
[18] Moustafa A Hammad, Michael J Franklin, Walid G Aref, and Ahmed K Elmagarmid. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 297–308.
[19] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 383–394.
[20] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 46.
[21] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 120–131.
[22] Zachary G Ives, Alon Y Levy, Daniel S Weld, Daniela Florescu, and Marc Friedman. 2000. Adaptive query processing for internet applications. (2000).
[23] Ryan Johnson, Stavros Harizopoulos, Nikos Hardavellas, Kivanc Sabirli, Ippokratis Pandis, Anastasia Ailamaki, Naju G Mancheril, and Babak Falsafi. 2007. To share or not to share?. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 351–362.
[24] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Processing Engines. In *Data Engineering (ICDE), 2018 IEEE 34th International Conference on*. IEEE.
[25] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 623–634.
[26] Christian A Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, Sriram Padmanabhan, and Kwai Wong. 2007. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 1136–1145.
[27] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *Acm Sigmod Record* 34, 1 (2005), 39–44.
[28] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 311–322.
[29] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 663–676.
[30] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 69–78.
[31] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 49–60.
[32] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.
[33] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: efficient shared execution of main-memory joins. *Proceedings of the VLDB Endowment* 9, 6 (2016), 480–491.
[34] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2018. Many-query join: efficient shared execution of relational joins on modern hardware. *The VLDB Journal—The International Journal on Very Large Data Bases* 27, 5 (2018), 669–692.
[35] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 37–50.
[36] F. Palermo. 1974. A Database Search Problem. In *Proc. ofthe 4th Symposium on Computer and Information Science*. ACM, 67–101.
[37] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment* 6, 9 (2013), 637–648.
[38] Vijayshankar Raman, Amol Deshpande, and Joseph M Hellerstein. 2003. *Using state modules for adaptive query processing*. IEEE.
[39] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. 2008. Constant-time query processing. (2008).
[40] Rajiv Ranjan. 2014. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing* 1, 1 (2014), 78–83.
[41] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. 2018. OLTPshare: the case for sharing in OLTP workloads. *Proceedings of the VLDB Endowment* 11, 12 (2018),

1769–1780.

[42] Matthias J Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. ACM, 1.

[43] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 23–34.

[44] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[45] Sangeetha Seshadri, Vibhore Kumar, and Brian F Cooper. 2006. Optimizing multiple queries in distributed data stream systems. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE, 25–25.

[46] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[47] Jonas Traub, Philipp Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *22th International Conference on Extending Database Technology (EDBT)*.

[48] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers.

[49] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment* 2, 1 (2009), 706–717.

[50] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 374–389.

[51] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1654–1655.

[52] Song Wang, Elke Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. 2006. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 619–630.

[53] Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. 2011. CoScan: cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 11.

[54] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *HotCloud* 12 (2012), 10–10.

[55] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 723–734.