HDES: A Dynamic Stream Processing Engine

Nico Dulduhardt
Hasso-Plattner-Institute
Potsdam, Germany
nico.duldhardt@student.hpi.de

Torben Meyer
Hasso-Plattner-Institute
Potsdam, Germany
torben.meyer2@student.hpi.de

Mavin Thiele
Hasso-Plattner-Institute
Potsdam, Germany
marvin.thiele@student.hpi.de

Anton von Weltzien
Hasso-Plattner-Institute
Potsdam, Germany
anton.weltzien@student.hpi.de

ABSTRACT

In this paper, we present HDES, a novel stream processing engine (SPE). The proposed engine focuses on enabling ad hoc queries to allow end-users to dynamically add and remove queries. While other approaches added ad-hoc functionality on top of common open-source SPEs, such as AStream and AJoin, HDES aims to make dynamicity a first-class citizen. This change in perspective allows us to optimize this use-case upfront and elevates the performance of ad hoc usage.

Moreover, we include several resource sharing optimizations into HDES to increase the performance of join-operations.

To evaluate our performance, we benchmark the proposed SPE across two distinct data-sets and compare the results against Apache Flink, a widely used state-of-the-art SPE.

1 INTRODUCTION

A stream processing engine (SPE) can provide near real-time insights from arbitrary data streams. Today, most commonly used SPEs, such as Apache Flink [1], Apache Storm [9] and Apache Spark Streaming [11], are optimized for running a fixed set of queries for a long period of time. While these tools can handle the most common workloads, they lack adaptability. When a user wants to add a query, the state-of-the-art SPEs have to halt, recompile all queries and resubmit the job. While this process is happening, no new insights can be derived from the incoming data and potentially important computations are delayed. Meanwhile, dynamic SPEs allow the user to create a query and submit a short lived query without impacting the production system. A common use case are short-lived queries. Imagine a team of data-scientist who want to quickly verify hypothesis on an incoming data streams. Nevertheless, running a large amount of queries at the same time quickly exhausts the computational resources. A large part of the overhead is cause by computation intensive join queries. To tackle this issue, we present a novel stream processing engine. HDES is a dynamic stream processing engine that enables the user to add and remove queries during runtime. It shares computation between queries that use the same data-sources and join keys which is crucial to achieve a good performance. To integrate these optimizations, we incorporated the work of AStream [7] and AJoin [6].

In the following paper, we will first introduce AStream and AJoin in Section 2. The SPE prototypes show the potential performance gains by sharing resources between queries. Utilizing these approaches, we present HDES architecture in Section 3. The architecture Section provides an overview for our engine and shows the integration of resource sharing techniques into a SPE that allows arbitrary user defined dataflows. The API to define these is presented, as well as a description of design decisions to represent dataflow graphs internally. In Section 4, we pick up the high level architecture and dive deeper into the implementation details. We start out with the transformation of queries into executable logic, followed by the implementation of time handling and windows. Then, we depict the execution of a single query in a local execution environment and multi-query handling. Next we evaluate our system in Section 6 using two different datasets, which is preceded by an overview of the benchmarking setup in Section 5. The overall resume is then summarized in Section 7 and further ideas that did not fit the limited scope of this project are described in Section 8.

2 BACKGROUND

In this section, we introduce AStream (Section 2.1) and AJoin (Section 2.2), two important concepts to understand ad-hoc and multi-query processing in HDES.

2.1 AStream

AStream [7] is a ad-hoc, shared computation stream processing framework built on top of Apache Flink. It introduces three operators that are chained. The shared selection operators are responsible for tagging each tuple with their related queries. Then, the shared aggregation or join operators perform shared computations based on these query tags. Last, the router operators send each tuple to all respective downstream processors. The tagging of queries is based on a query set represented as a bit map. A set bit indicates that a tuple is relevant for a specific query. Furthermore, to compact the query sets by reusing the bits of old queries, AStream uses another bitmap called changelog. A set bit in a changelog indicates that the query remains unchanged. Based on a dynamic programming approach, the query set k can be computed with with the query set t - k and k changelogs.
In stream processing, aggregations, as well as joins, are based on windows that group incoming data. AStream's computation model uses smaller, non-overlapping windows called slices. For each slice, it computes a result. With that, it can reuse results for queries with different windows but the same computation.

2.2 AJoin

AJoin [6] is an extension of AStream. Its goal is to improve the performance of joins in multi and ad-hoc query settings. Karimov et al. identify a skewed workload in modern SPEs. In contrast to other operators, join operators are tasked with multiple resource-intensive workloads. Consequently, there are a lot of idle resources and no potential to scale individual parts of a join operation to its needs. AJoin distributes the workload of the join operator. The AJoin's source is tasked with windowing and indexing incoming tuples. Then, AJoin's join operator builds sets of join elements that are joined by its sink operator with late materialization.
Additionally, AJoin aims at optimizing the execution plan at runtime. Joins with matching sources are deployed as one operation eagerly. An optimizer monitors statics for all joins and calculates the cost of shared and unshared join operations. Based on this, it reorders join operations. Furthermore, it allows the vertical and horizontal scaling of pipeline operators if needed.

3 ARCHITECTURE

In the following sections, we will provide an architectural overview for the stream processing engine. It consists of several abstractions depicted in Figure 1.

First of all, the user specifies each query on the stream, using our API. It allows the user to define custom dataflow graphs containing user-defined functions. Furthermore, the user specifies the event time and allowed lateness for each incoming stream event. The user-defined dataflow is transformed into a logical plan which represents the event transformations in an environment agnostic way. It can be efficiently traversed to build an environment specific execution which is executed by our engine. To support ad-hoc queries, we repeat the steps above for each query and merge the resulting logical and execution plan.

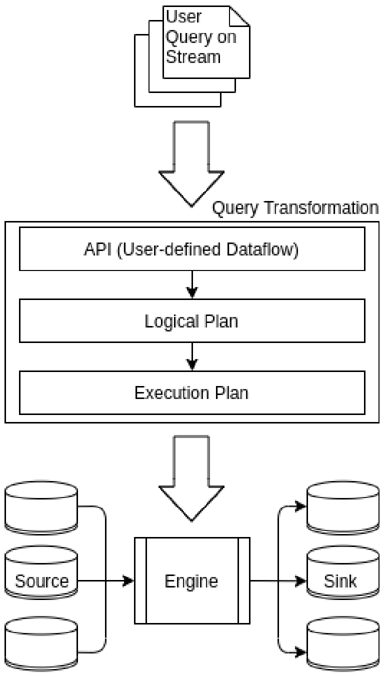

Figure 1: Architecture Overview

3.1 External View

HDES provides a Java API to define queries similar to the APIs of other well-known SPEs like Apache Flink or Storm. To explain HDES API in more detail, we consider a query based on the example of an online shop. There are two sources of streaming events. The first contains sessions created by an external service and the other purchase events. The query calculates the daily purchases of an item, which costs more than a minimal amount, within 5 minutes of a session start. In HDES, this query can be expressed as shown in Listing 1.

```
TopologyBuilder builder = TopologyBuilder.newQuery();
AStream<Session> sessionStream = builder.streamOf(sessions);
builder.streamOf(purchaseSource)
    .filter(purchase -> purchase.getAmount() > MIN_AMOUNT)
    .window(TumblingWindow.ofEventTime(Time.seconds(20)))
    .join(sessionStream,
        (purchase, session) -> purchase,
        Purchase::getUserId,
        Session::getUserId,
        Purchase::getTimestamp)
    .window(TumblingWindow.ofEventTime(Time.days(1)))
    .groupBy(purchase -> purchase.getObjectId())
    .aggregate(new CountAggregator<>())
    .to(sink);
builder.buildAsQuery();
```

Listing 1: Query definition with HDES

The entry point is a *TopologyBuilder*, which allows creating a stream from a source (lines 2 and 3). Streams in HDES are generic and therefore support any kind of input data. They can be transformed by several operations, like *map*, *window* or *join*. Those operations take one or more user-defined functions (UDF), which are defined as an interface. For example, the *filter* in line 4 has a UDF as a parameter, which takes an element that has the same type as the stream's elements and returns a boolean. In line 5, the stream is windowed by a tumbling window by 5 minutes windows of the event time. HDES supports both tumbling and sliding stream windows. Opposed to the previous operations, *window* returns a *WindowedAStream* instead of an *AStream*. HDES uses different types of streams to represent the current state of a stream, where each state supports a particular set of operations. Table 1 contains a complete list of stream types and their respective operations.
Supporting windowing, HDES requires the presence of a timestamp for each stream tuple. Therefore, the user can define a timestamp extractor in a source. Additionally, out-of-order handling with watermarks can be enabled by supplying a watermark generator. The generator is customized by the allowed lateness of events and the watermark interval. The latter determines the frequency with that HDES generates watermarks.

Stream type	Name	Description	Resulting type
AStream	flatMap	Maps an incoming elements to an iterable of any type.	AStream
	map	Maps an incoming element to a arbitrary type.	AStream
	filter	Retains only the elements, which comply with a given predicate.	AStream
	window	Windows the stream with a given window assigner.	WindowedAStream
	to	Writes the elements of the stream into a sink.	None
WindowedAStream	groupBy	Groups the stream by a key.	KeyedWindowedAstream
	aggregate	Aggregates the stream in a window.	AStream
	join	Performs a join with a different stream.	AStream
	ajoin	Performs an AJoin with a different stream.	AStream
KeyedWindowedAStream	aggregate	Aggregates the stream by a key in a window.	AStream

Table 1: HDES API for AStream, WindowedAStream and KeyedWindowedAStream

3.2 Internal Representation

3.2.1 Operators. An operator encapsulates user-defined transformation logic. For each input event, it produces zero or more output events. The output events are then passed to a collector. The collector is interchangeable and therefore, the operator is decoupled from downstream operations. The user defined functionality is encapsulated in the following four operator categories.

Source operator A source operator provides the implementation to read events from a source.

One input operator A one input operator transforms each input event into an arbitrary amount of output events.

Two input operator A two input operator encapsulates the logic to process data from two different input streams and joins them to a single stream.

Sink operator A sink operator provides an implementation to write events to some user-defined output.

3.2.2 Logical Plan. The logical plan represents the user-defined dataflow using a directed acyclic graph (DAG). Therefore, it enables us to use graph traversal algorithms for environment agnostic optimizations and other transformations. Each graph node has between zero and two inputs and an an arbitrary amount of downstream nodes. There are four types of nodes, corresponding to the four types of operators.

3.2.3 Execution Plan. The execution plan is built by traversing the logical plan. It represents the environment specific execution logic and is completely interchangeable. Currently however, HDES is limited to an execution plan that executed on a single node. The provided local execution plan is represented as a DAG and each node corresponds to a node in the logical plan. Each execution node wraps an operator, executes it and collects its output events. The events are then passed downstream as shown in Figure 2.

3.3 Dynamic Query Processing

So far, we explained the architecture of processing a single query. This section examines how HDES supports efficient scaling with dynamic addition and removal of queries during runtime.

3.3.1 Source Sharing. HDES uses source sharing to reduce costly IO operations. We consider the following example: multiple queries

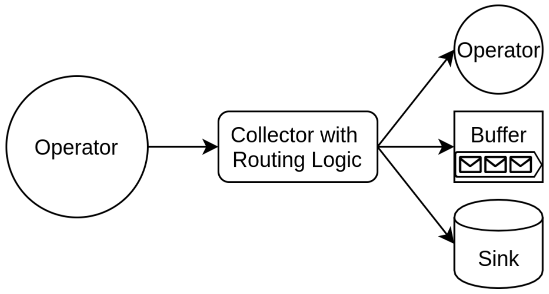

Figure 2: Routing events through the execution graph

analyzing a stream of purchases backed by an Apache Kafka topic. In a naive approach, the SPE deploys a source for each query independently. Evidently, this introduces a major overhead. HDES deploys only one source for all queries based on the same topic. Sources are identified by manually defined or automatically inferred IDs, while the latter depends on the concrete implementation of each source type. This allows HDES to read items of a source only once. Then, the source sends the read element to each downstream processor. An example of source sharing can be seen in 2 the corresponding object diagram in Listing 8. The join operator and *j* and the map operator *m* share the network source *nws1.*

3.3.2 Aggregation Sharing. HDES borrows the idea introduced by AStream to share results of aggregations between multiple queries. Tagging each tuple with its respective query, requires knowledge of currently active queries in the operator. Therefore, HDES supports a special metadata stream that operators can subscribe to. The engine's query manager informs the operators of the active queries by writing in this stream. Then, the operators can tag all relevant tuples with their query set and a custom operator performs the shared aggregation.

3.3.3 Join Sharing. HDES shares the results of joins based on AJoin's distributed join architecture. We see two issues regarding AJoins architecture and HDES. First, source sharing between AJoin and non-AJoin queries is not feasible when AJoin requires specialized sources. This is especially problematic if the source does not support multiple reads. Second, a sink operator does not allow for any further down-stream processors. With that, AJoin marks the end of a query definition. Thus, we introduce specialized one-input operators to distribute the join's workload as shown in Figure 3. This allows source sharing and further down-stream processors while maintaining the advantage of distributing the workload of the join operator.

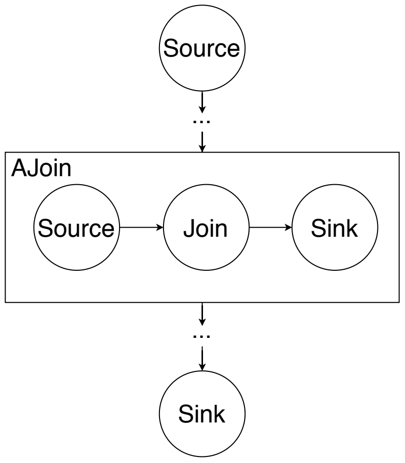

Figure 3: AJoin integration in HDES

4 IMPLEMENTATION

In this section, we first explain how HDES handles time. This includes important concepts like windows and watermarks. Next, we describe the conversion of a user-defined query into an execution plan, followed by the execution of single queries. In Sections 4.4 and 4.5, we examine HDES implementations of AStream and AJoin, respectively.

4.1 Time and Windows

4.1.1 Event time. First, we need to extract the time from the incoming events as shown in Figure 4. We attach this metadata in each source before any further transformations are applied to the event. Non-source operators may extract and utilize the event time metadata.

To extract the time from each event, we use a user provided *TimestampExtractor*. Then, we use the specified lateness and the watermark interval of the *WatermarkGenerator* to add a watermark timestamp to every *n* elements, where n is the watermark interval. The watermark timestamp is calculated as follows:

$$watermarkTimestamp = eventTime - allowedLateness$$

We only attach it every *n* times to limit the amount of data and time to transfer and process the watermarks.

After performing operations that merge events like joins or aggregation, the user has to supply a *WatermarkGenerator* and a *TimestampExtractor* again to specifiy time handling for the new events.

Currently, the watermark timestamp is used by the aggregation and join operators only, to advance their event time and close

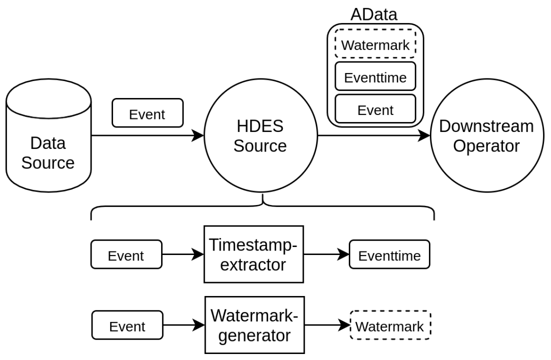

Figure 4: Wrapping incoming events with AData Metadata

windows. Join operators receive watermark timestamps from two different streams. Therefore, we decided to advance their event time to the minimum event time of the joined streams whenever they receive a timestamp.

4.1.2 Windows. Implementing windows builds the foundation for any kind of aggregation or join operator. The user specifies the window type as shown in the code example 1 creating a *WindowAssigner*.

A window assigner maps timestamps to windows which store the metadata of each window. Figure 5 illustrates a WindowAssigner for sliding windows. It receives the timestamp 18s and calculates all matching *TimeWindows*. The returned *TimeWindows* store the start and end time of each window. Each operator may use the *WindowAssigner* to buffer and transform elements depending on the *Window* they belong to.

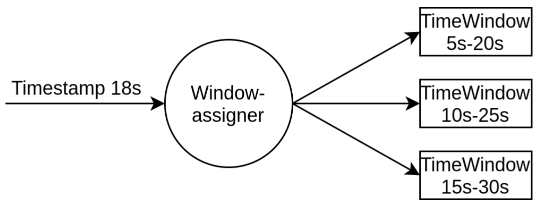

Figure 5: WindowAssigner for sliding windows with size 15s and slide 5s

Currently, we only support sliding and tumbling windows but implementing a new window only requires the user to implement a new *WindowAssigner* and a *Window* which stores the window metadata.

4.2 Conceptual Transformation

Illustrating the conceptual transformations required in HDES, we consider the example query of Section 3.1 shown in Listing 1. HDES represents the logical plan as a list of nodes, where each node stores both its children and parents. The engine differentiates between four different types: *SourceNode*, *UnaryOperationNode*, *BinaryOperationNode* and *SinkNode*. Therefore, the resulting topology as shown in Figure 6 consists of six nodes: two *SourceNode*s and *UnaryOperationNode*s as well as one *BinaryOperationNode* and *SinkNode*.

Transforming the logical plan into the execution plan, the execution

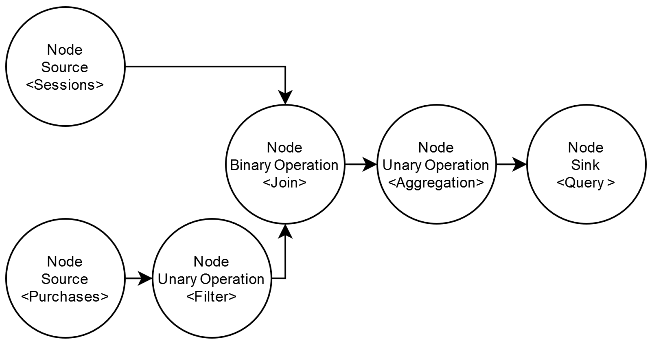

Figure 6: The logical plan of the example query

plan builder first identifies already deployed sources. As described in Section 3.3.1, this is based on IDs. Therefore, it simply uses a set intersection of the currently deployed logical plan and the query's logical plan to drop these source nodes.
We use the visitor pattern to traverse the logical plan in a type safe manner. Thus, each node accepts a *NodeVisitor*, which is an interface implemented by the *ExecutionPlanBuilder*. The builder creates a different slot for each node type visiting the nodes of the logical plan. Because of the topological ordering of the logical plan's nodes, the parents are created before their children. Hence, the builder can safely register the current slot as an output in the parent slot. As described in 3.2, this ensures the correct routing of elements in HDES.

4.3 Query Execution

HDES' execution model uses slots as its basic building blocks. A slot provides all necessary dependencies and piping for an operator. A *Slot* collects the transformed events of its operator and passes them on to the downstream operators. 7 presents a simplified model of the execution architecture. We differentiate between three different implementation of slots: *OneInputPushSlot*, *SourceSlot*, and *TwoInputPullSlot*. The two latter inheriting from *RunnableSlot*, meaning they run in their own threads.

4.3.1 *SourceSlot.*

Sources have their own thread and can read from the network or in-memory data structures. *SourceSlots* are followed by *OneInputPushSlots* or *TwoInputPullSlots* depending on the query. If a *SourceSlot* is followed by a *OneInputPushSlot*, it will directly write into that slot's *OneInputOperator* via the *SlotProcessor* abstraction. An example of this can be seen in 8, where the source slots sends the events it reads from the source to the *OneInputOperator* at the top of the diagram. In case the *SourceSlot* is followed by a *TwoInputPullSlot*, it will write into a *Buffer* also via the *SlotProcessor* abstraction, this can also be observed in 8. The *SlotProcessor* interface has only one method namely *sendDownstream*. Any *Slot* can have one or more *SlotProcessors* which receive events for further processing downstream. This mechanism also provides the functionality for adding new queries during runtime as new downstream *SlotProcessors*, i.e. operators, sinks and buffers can be added to the *Slot*'s list.

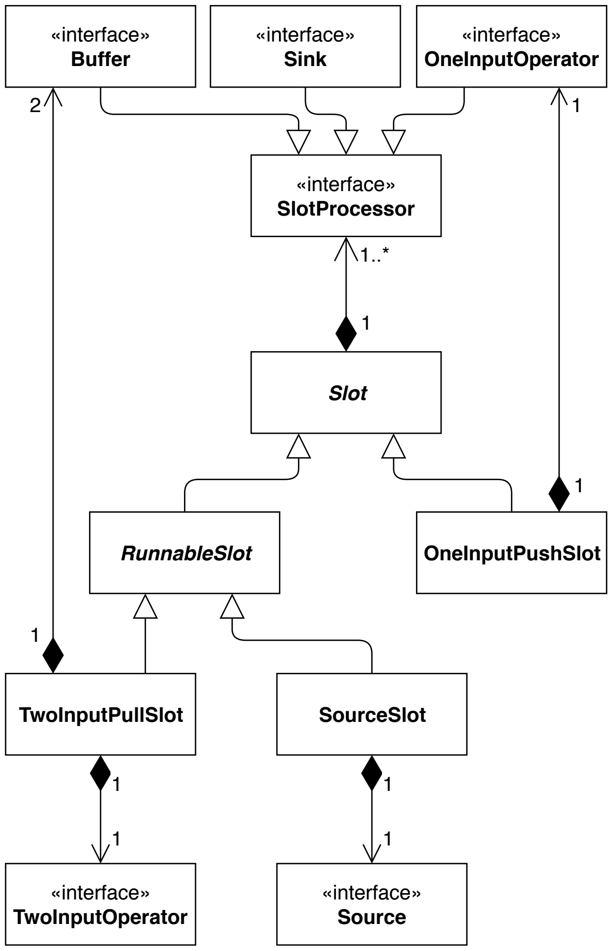

Figure 7: Class diagram illustrating the query execution structure

4.3.2 *OneInputPushSlot.*

The only slots not running in their own thread are *OneInputPushSlots*. They are chained together and executed in a thread with a *SourceSlot* or a *TwoInputPullSlot*. This is done to enable operator-chaining for unary operations. Synchronization between threads has a significant cost, thus, it is desirable to execute simple, low cost operators like map and filter together in one thread sequentially, thus avoiding synchronization between threads. An operator inside a *OneInputPushSlot* will receive its input value from the previous slot and will use its *OneInputPushSlot* to write the resulting value into its succeeding operators.

4.3.3 *TwoInputPullSlot.*

HDES uses *TwoInputPullSlots* to execute *join* or *ajoin* operators. These slots have two input *Buffers* from which they read the events that are supplied to the *TwoInputOperator*. The results of this operator are then written to the attached *SlotProcessors*. *TwoInputPullSlots* also run in their own thread, just like *SourceSlots*. This decision was made because join operations are usually the computationally most expensive operation in a stream query. By prepending joins with two buffers, that synchronize the upstream and the join thread, we can prevent upstream operators from stalling, when a join computes its window result. This helps us to keep the throughput more stable. An example can be seen in 8, the *TwoInputPullSlot* polls from two buffers and writes the events to its *TwoInputOperator*. In turn, this operator writes its join results back to the *TwoInputPullSlot*. which routes the joined events to the sink, For simplicity, the sink is omitted from the diagram.

4.3.4 Buffer. Buffers are the synchronization method between *RunnableSlots*. Therefore, they are crucial for performance. Buffers are implemented as queues. We found that the synchronizing queues of the Java standard library use a lot of locking and therefore limit performance significantly. Thus, we implemented a buffer which pre-buffers incoming elements in an *ArrayList* called chunk before writing this whole chunk into a synchronized queue. This speeds up the buffer throughput substantially. The chunk size as well as the maximum latency introduced by the buffer, and the maximum size of the buffer itself can be fine-tuned to optimize performance.

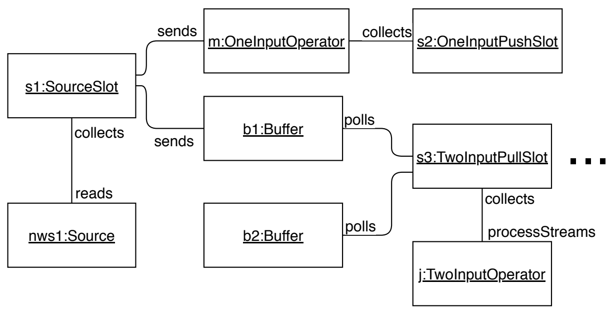

Figure 8: Object diagram of execution of the queries shown in Listing 2, the second source and the sinks are omitted

```
JobManager jobManager = new JobManager();
// Create source objects
NetworkSource nws1 = new NetworkSource(7001, ...);
NetworkSource nws2 = new NetworkSource(7002, ...);

// Create join query
TopologyBuilder builder1 = TopologyBuilder.newQuery();
AStream<Tuple2<Long,Long>> s1a = builder.streamOf(nws1);
AStream<Tuple2<Long,Long>> s2 = builder.streamOf(nws2);
s1a.window(TumblingWindow.ofEventTime(Time.seconds(5)))
    .join(s2,
        (t1, t2) -> Tuple2.of(t1,t2),
        Tuple2::v1,
        Tuple2::v1)
    .to(new FileSink("join"));
Query joinQuery = builder1.buildAsQuery();
jobManager.addQuery(joinQuery);

// Create map query
TopologyBuilder builder2 = TopologyBuilder.newQuery();
AStream<Tuple2<Long,Long>> s1b = builder.streamOf(nws1);
s1.map(t -> Tuple2.of(t.v1(), t.v2() + 1)).to(new
↪  FileSink("map"));
Query mapQuery =  builder2.buildAsQuery();
jobManager.addQuery(mapQuery);
```

Listing 2: Multi-Query definition with HDES

4.4 AStream

4.4.1 Shared session. AStream requires a central instance to inform participating operators about the current queries. HDES allows operators to subscribe to additional streams that contain current metadata such as the current query set. The shared session is a singleton that is updated by the engine. For each window slice, it writes the current query changelog in the metadata stream.

4.4.2 Selection operator. The operator *StreamSharedSelection* tags each incoming tuple with the query set. Therefore, it needs to read the metadata stream of the shared session.

4.4.3 Aggregation operator. The *StreamSharedAggregation* contains an intermediate state for each query set in a window. An incoming tuple updates the state according to the user-defined function. The processing of the intermediate state is triggered by a watermark. For each closed window, the operator iterates over the intermediate states of all query sets and combines the states for each query individually. Afterward, it computes the result for the state of each query and creates a shared value, where only the bit for the corresponding query is set.

4.4.4 Routing operator. The routing operator is required because HDES does not support dynamic routing in operators. Hence, such an operator is deployed for each query involved in the shared aggregation. Each operator is responsible for selecting the tuples of one query by selecting the respective bit of the query set. It strips the query set and forwards the element to downstream processors.

4.5 AJoin

AJoin must be explicitly enabled in user-defined queries. This can be done by using *ajoin* instead of *join*. As introduced in Section 3.3.3, HDES version of AJoin consists of three nodes. The source and sink are represented by *UnaryOperationNodes* and the join by a *BinaryOperationNode*. Therefore, AJoin requires no special handling during the conversion into the execution plan.

4.5.1 Source operator. The class *StreamASource* is an one-input operator responsible for windowing and indexing the incoming elements. For both streams of the join, an independent source operator is deployed. On processing an incoming element, the operator extracts the join index and window. It then appends the element to the list of values for this index in the window.
The windowing is based on a slicing approach, which allows to share the indexing of joins with different sized windows. The slice size of the operator's window assigner is a configurable fraction of the maximal slice size of the smallest window. When a watermarks closes a slice, the operator creates an immutable data structure called *Bucket*. A bucket contains the mapping of indices to elements for a particular window slice.

4.5.2 Join operator. The *StreamAJoin* operator processes the incoming buckets of both sources. Since AJoin leverages late materialization, the operator stores all buckets in its state until encountering a watermark. Then, it builds the set intersection for the indices of buckets in the same window. The resulting *IntersectedBucket*, which contains two sets of values with the same index and window, is forwarded to to the sink operator.

4.5.3 Sink operator. The *StreamASink* operator performs the cross-product of the two value sets. In contrast to both other operators described previously, the sink operator is deployed for each AJoin independently. Thus, joins with the same predicates but different selections can be shared efficiently.

4.5.4 AJoin identification. To avoid deploying source and join operator multiple times, HDES uses the same id-based strategy as for source sharing (Section 3.3). However, since our version of AJoin allows up-stream operators, it is not always trivial to infer an unique id. HDES offers an automatic and a manual id assignment. The automatic assignment uses the IDs of the parent operators. Therefore, this approach is feasible when there are no up-stream operators other than sources. The manual id assignment requires the user to choose an unique identifier for AJoins with the same input.

5 BENCHMARK DESIGN

To evaluate HDES' performance and put it into context, we designed a benchmarking suite that runs different benchmarks and compares HDES' performance to Apache Flink, a popular stream processing engine.

5.1 Architecture

Our benchmark setup consists of two separate elements, the data-generator and the engine itself. We decided to strictly separate the data generation and the engine to get a clear picture of the system under test. Furthermore, we execute the two programs on different machines to isolate the computation. That way, we avoid that the engine and the data-generator have to compete for resources. By running the benchmark over the local network, we are theoretically limited by the 1 Gbit/s bandwidth. However, we do not fully exhaust the network bandwidth using our specific test setup.

A benchmark starts by the data-generator listening on at least one network socket (TCP) for incoming connections. The engine then connects to the data-generator with a given IP-port combination to start the execution. The data-generator then generates events in a time-aware fashion, serializes the events, and finally sends them via the socket to the engine. The data-generator will generate data at a rate no higher than the specified events per second (EPS). However, both HDES and Flink are capable of asserting backpressure through the socket by slowing down the rate at which they read from its sockets. If the engine is overloaded, it will reduce its read rate from the socket; thus, the data-generator will reduce its event generation and send rate. This theoretically means that the engines accept events only at a sustainable throughput rate. If this is ever not the case and latencies rise unsustainably, we highlight this in our analysis. The execution of the benchmark is halted once either part is exceeding a set benchmarking time period. All benchmarks are run for five minutes.

5.2 Test Setup

To benchmark our system, we used two machines that are connected via a 1-GBit LAN. The data-generator runs on a 2.50 GHz 4-core Intel i5-7300HQ CPU with 16 GB of DDR4 memory, while the engine is utilizing a 3.40 GHz 4-core Intel i7-2600K CPU with 16 GB of DDR3 memory. Both machines are running Windows. The JVM is provided 10 GB of memory on both the data generator and the engine. We synchronized the internal clocks according to the Windows time servers before each run, although that method did not always result in exact synchronization. We tried to minimize this effect on the benchmark as much as possible. Flink runs as a local stream environment with check-pointing disabled to make the results more comparable. It has to be kept in mind that all the numbers we produce in this benchmark are tied to this setup and might look very different on a machine that has more memory and more CPU-cores.

5.3 Metrics

To evaluate the engine's performance, we employed three different metrics. While the event time t_e is defined as the time at the creation of an event by the data-generator, the processing time t_p describes the point in time when the engine first processes the event, i.e., it has been read from its socket and input buffer and reached the first operator. At last, the ejection time t_j is the time when the event leaves the engine's last operator and is written into the sink. With these three timestamps, we define the event time latency $lat_e = t_j - t_e$ and processing time latency $lat_p = t_j - t_p$ to use as our main metrics for the benchmarks. Thus, lat_e includes the time an event spends in the data-generator before being sent and the time it spends in the engine's input buffer in addition to lat_e.

As a third metric, we also measure the sustainable throughput of the system at the data-generator. This can be done as the engine pulls the events on demand according to its current load. The throughput is measured by counting the number of events sent by the data-generator as opposed to counting the events ejected by the engine. This is done because different queries can alter the number of events ejected by the engine, which would hinder comparison of query performance.

We mostly use the terminology and methodology described in [5], including how latencies are calculated for windows. However, with one notable exception: We always use a sustainable event generation rate as opposed to starting with a very high generation rate and slowly decreasing it until we reach a sustainable rate. This was mainly done not to overload our limited hardware setup and to be able to run the tests for a shorter amount of time as we do not have to spend time finding the equilibrium.

5.4 Serialization

Our benchmarking setup requires that we send the generated events over the network. This forces us to serialize the Java objects into a format that can be sent over the network. Over the development of the project, we have tried several serializers such as Kryo [8], GSON [3], Jackson [2] and Protobuff [4]. While GSON and Jackson rely on the JSON format, both Kryo and Protobuff serialize the message into a byte format. The mentioned libraries are of great help serializing objects of varying types, but this flexibility comes with a performance cost.

Since the impact of serialization and deserialization was rather large, we decided to additionally try to serialize the objects by using a custom string serialization. While not being adaptable to arbitrary objects, this approach gave us the lowest serialization overhead.

5.5 Data

The choice of the benchmarking data-sets and the associated queries for stream processing engines is not unified and varies among researchers. Especially ad-hoc stream processing has no common workload definitions [7]. Therefore, we took inspiration from existing stream processing benchmarks [5, 10] and included our own ideas.

To evaluate our engine, we used two different data-sets. On the one hand, we used a simple tuple consisting of an integer value and an event-time timestamp, further referred to as "basic benchmark" to establish a performance baseline that should be unaffected by data size. On the other hand, a reduced set of the Nexmark [10] benchmark. These two types of data-sets allow us to test the engine using small and efficient integer events and more realistic Nexmark-like events.

Our adapted version of Nexmark consists of Bids and Auctions. An auction has an auction ID, a quantity, a type and a reserve price. A bid consists of a bid ID, an auction ID, a bidder ID and a price. Several attributes from both entities are normally distributed to simulate, for example, an especially active group of bidders. In concordance with their semantic attributes, both auctions and bids contain an event-time timestamp. Each of the two objects is transported by an individual network socket.

5.6 Queries

To gain an in-depth understanding of the individual operator performance as well as the overall performance, we defined a variety of query types for our two data sets. Each query type was executed in two different categories with several different configurations. The first category measures static multi-query performance by executing a fixed number of the same queries. The second category adds and deletes a varying number of queries in a fixed interval to get an understanding of ad-hoc query performance. That means our benchmark will add X queries at once, wait a specified wait period, and then remove Y queries at once at the end of that period.

5.6.1 Basic Queries. As we use the basic data set mainly to establish a baseline, we only used two query types. A map query that simply maps the events with an identity function and a join. For the join, a key which meets the join criteria occurs every ten-thousandth element per stream. The join is performed on a tumbling window with an event-time size of 5 seconds.

5.6.2 Nexmark Queries. To demonstrate the capabilities of the engine, we decided to use simple filter and join queries in these experiments as well as two complex queries on the Nexmark dataset. This allows us to evaluate how the engine performs on queries which cover a larger range of operations in one single query. While one calculates the highest price per auction within a given time-window (HPPA), the other one aims to find the hottest category (HotCat), which is the category which received the highest amount of bids in a certain time-window.

More specifically, the HPPA query joins the bid and the auction stream on the auction id with a five-second event-time window. The resulting stream is then filtered to make sure that only bids above the reserve price are considered. After that, stream events are grouped on the auction id, using another five-second event-time window. In the last step, the highest bid for each auction is written into the sink.

Similar to the HPPA query, the HotCat query also joins both streams on the auction id within a five-second event-time window. These joined tuples are then grouped by the category, using another five-second event-time window. Finally, the category with the highest number of bids is written into the sink.

Both queries are also implemented in Flink in a very similar fashion to enable a fair comparison.

6 EVALUATION

In this section, we first have a look at static multi-query performance in subsection 6.1 where we also compare the performance of HDES with Flink, and then we look at ad-hoc query perform in subsection 6.2.

In all experiments, we measure the metrics from subsection 5.3 with the two data sets and the different query configurations. To illustrate the results, we use Box-Plots, according to their most commonly used configuration. The green line signifies the median, the box the interquartile range and the whiskers are defined as the minimum and maximum point within 1.5 times the interquartile range (IQR). For clarity, we exclude outliers outside of the mentioned 1.5 times IQR if there are any.

To make the plots more compact, we use abbreviations. The first word refers to the query type, a refers to the batch size of queries added, r refers to the batch size queries removed, w refers to the wait time in seconds between the addition and/or removal of a batch of queries, f refers to the number of fixed queries, i.e., the number of queries that run the full time of the benchmark. For brevity, we do not always include all possible comparisons but rather the ones that provide interesting insights. Often, we compare the AJoin to a standard join implementation provided by HDES. This join is not optimized for multi-query execution, i.e., does not share any data among joins. The graphs abbreviate this join with *pJoin* for plain join. If labels are prefixed with *NAR* it means the queries where performed on the Nexmark data set.

6.1 Static Multi Query Performance

In this group of experiments, we want to evaluate how many queries of one type HDES and Flink can execute at once. This way, we can get a rough understanding of how scalable a particular operation is.

6.1.1 Map Basic Data. In this experiment, we establish a first baseline for multi-query execution by mapping the events with the identity function.

As we can see in Figure 9, HDES's throughput for one fixed query is higher, while multi-query execution of the map operator performs similar to Flink. However, while Flink seems to be able to keep its latency stable (Figure 10), HDES increases the event-time

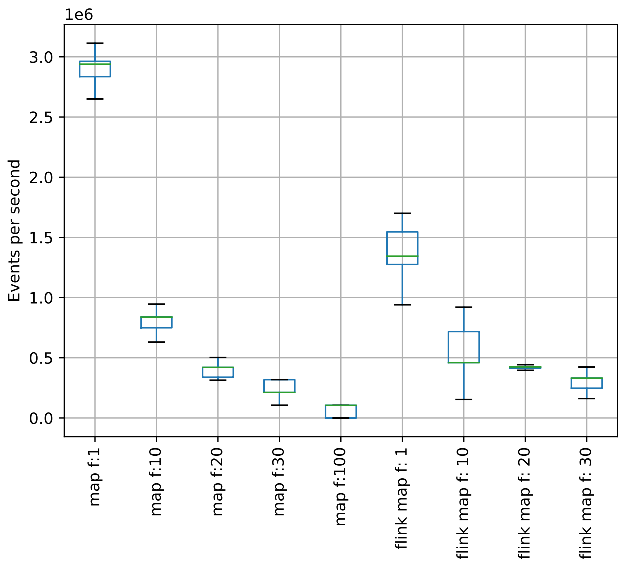

Figure 9: Map throughput comparison for basic data

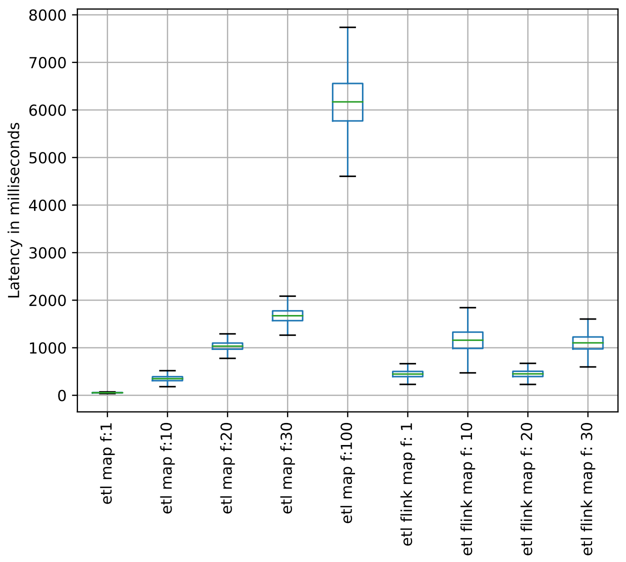

Figure 10: Map event-time latency comparison for basic data

latency with each added Map operator, as all map operators run sequentially in the *SourceSlot* whereas Flink runs map operators in multiple threads.

6.1.2 Filter Nexmark Data. To establish a baseline for Nexmark, we used a slightly more complex filter query. As we can see in Figure 11, HDES compares more favorably than in Figure 9. The latency, however, looks similar and is therefore omitted here. We believe that Flink's decreased performance is owed to the large event size of the Nexmark events, which seems to affect Flink more than HDES in this benchmark.

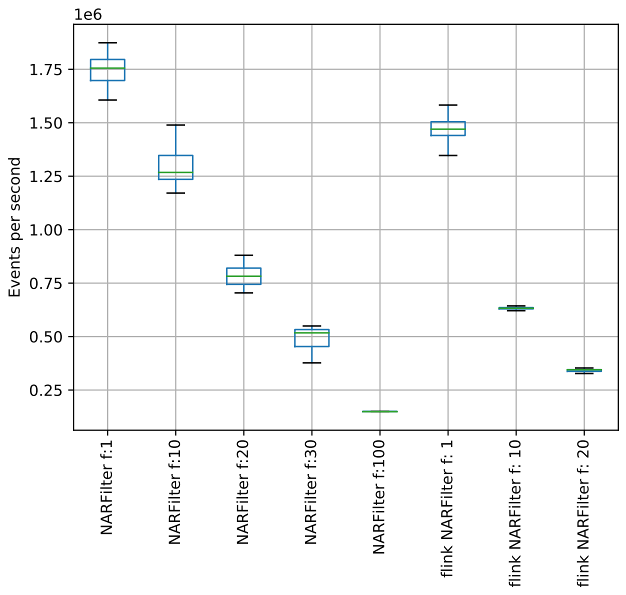

Figure 11: Filter throughput comparison for Nexmark data

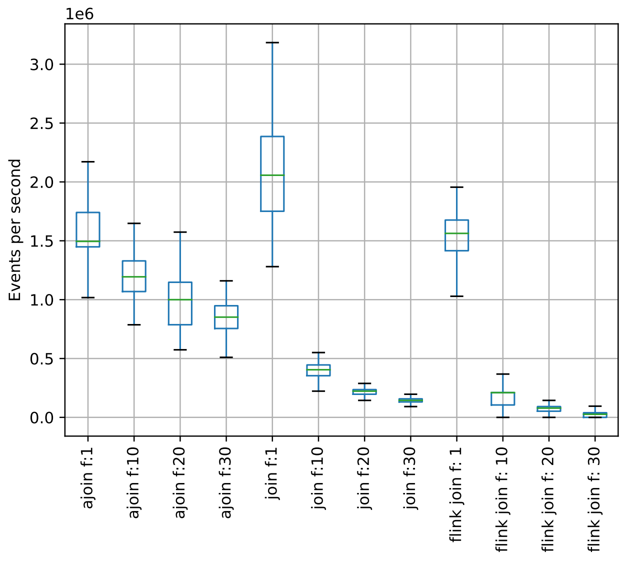

Figure 12: Join throughput comparison for basic data

6.1.3 Join Basic Data. As we can see in Figure 12 and Figure 13, the standard join implementation is superior in regards to throughput and latency when executing one join. However, as soon as the number of joins that have to be computed concurrently is increased to ten, the AJoin implementation is more performant. Additionally, we can observe that AJoin has a latency that is almost twice as high as the standard join's latency, because of its additional data structures and computations. Both implementations perform better than Flink except when only a single join query is executed. In that case, AJoin is slightly slower.

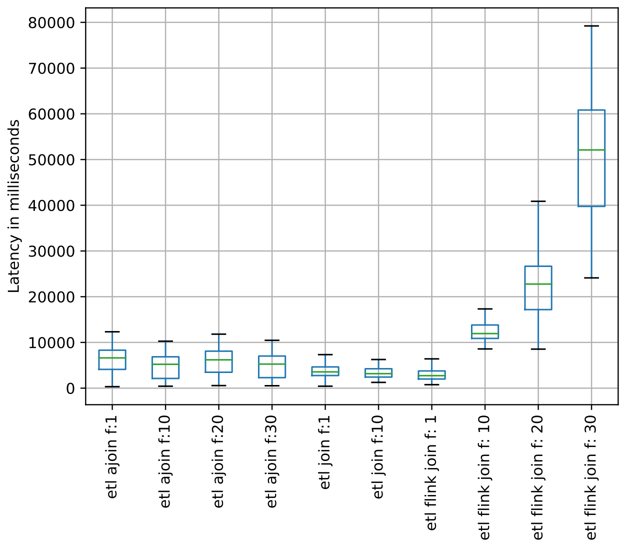

Figure 13: Join event-time latency comparison for basic data

In Figure 13 we can also see that the standard join did not provide any results for twenty and thirty queries running in parallel. This highlight one of Flink's strength quite well. While it had a very low throughput, it was stable enough to still provide results. Additionally, we can also see that the median latency for the standard join and Flink's join for one fixed query is close to the 2.5 seconds that would be ideal for a window size of 5 seconds. AJoin's optimizations for multi-query optimization, however, come with a latency penalty.

6.1.4 Join Nexmark Data. The comparison of join performances with Nexmark data is more complex. As we can see in Figure 14 and Figure 15 AJoin's performance, especially when executing just one join is worse than the one of the standard join. We can see in Figure 15 and Figure 16 that AJoin starts well, but after a few seconds the processing time latency starts to rise and after circa 150 seconds, AJoin stops reading events from the source. We investigated this issue and concluded that the problem lies with the Java garbage collection. We regularly observed more than 80% of the runtime spent with garbage collection, due to the nature of Nexmark's larger tuple size memory gets scarce faster, and the garbage collection is no longer able to free memory fast enough. AJoin is affected the most because it keeps additional data structures—especially Hashmaps—that further increase the garbage collection overhead. As the AJoin operator itself is not able to rate-limit itself, it reads a lot of events into its data structures, quickly causing large garbage collection pauses. This also explains why ten AJoins perform better than one. Because the benchmark includes a map operator before the join to set the processing time timestamp, source sharing between these map operators is enabled. As these map operators run sequentially in the same *SourceSlot*, they naturally lower the input rate. This results in fewer events in AJoin's data structures and thus less garbage collection. To limit this problem, we experimented with HDES's network input buffer sizes. As we can see in Figure 18, all tests that had a network input buffer of 10 thousand elements

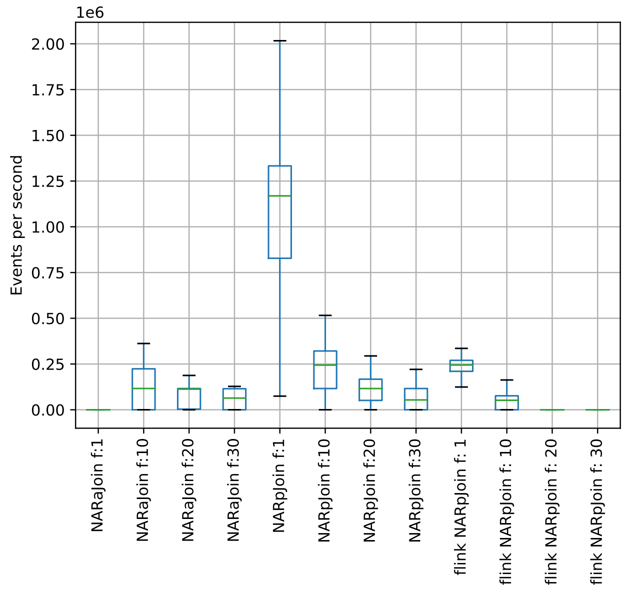

Figure 14: Join throughput comparison for Nexmark data

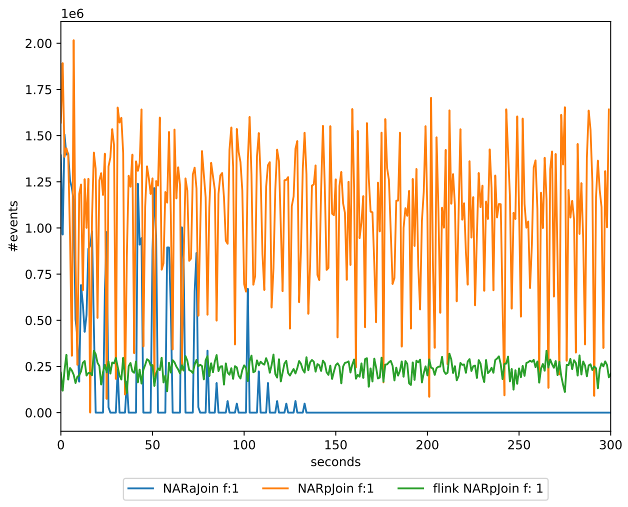

Figure 15: Join throughput comparison for Nexmark data with one join

or less, kept the latency stable and almost performed equally with a median near 200 thousand events per second. The buffer size of 17 thousand already shows increasing latencies and is therefore not sustainable.

6.1.5 Hottest Category Nexmark Data. As the hottest category query is defined on the Nexmark data-set, we had to limit the input buffer size for the experiments that used AJoin as the join operator.

We can observe in Figure 19 that the standard join implementation yields better results if only one query runs because its input buffer size is not reduced. We can also see that AJoin has slightly

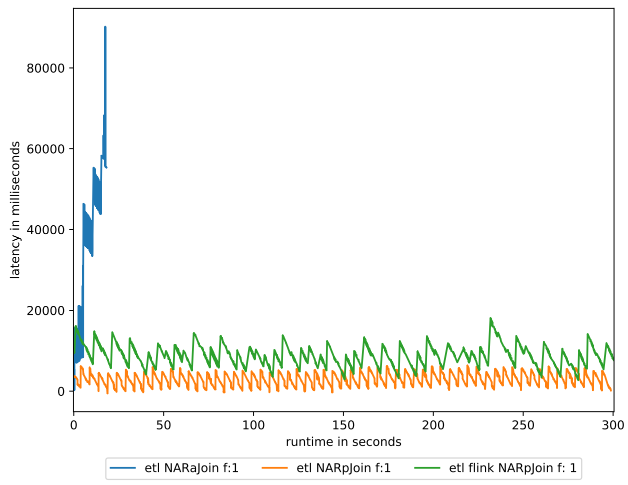

Figure 16: Join processing time latency comparison for Nexmark data with one join

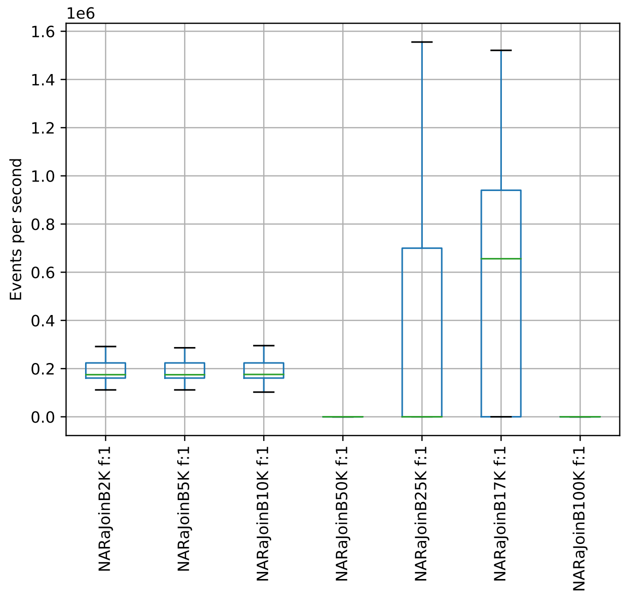

Figure 17: AJoin throughput comparison for different buffer sizes

better throughput if more than ten queries run. However, the event-time latency increases significantly if thirty queries are run at once and is not sustainable in that case, as we can see in Figure 20. Flink, on the other hand, cannot run more than ten hottest category queries at the same time.

6.1.6 Highest Price per Auction Nexmark Data. The highest price query presents a picture that is very similar to that of subsubsection 6.1.5. The queries that use AJoin again struggle with unsustainable latency if twenty or more queries run in parallel, even though we once again use a reduced input buffer size for AJoin. This can be seen well in Figure 22. If the query is performed with AJoin and 20 or 30 queries run in parallel, the latency increases consistently. This can be seen well on the red and green line. If the query is executed with the standard join (pJoin), the latency stays stable. However, the throughput is slightly worse than what the query achieves if AJoin is used as can be seen in Figure 21. However, that throughput is likely not sustainable if AJoin is used due to the rising latency. Flink performs again worse than the standard join version of HDES.

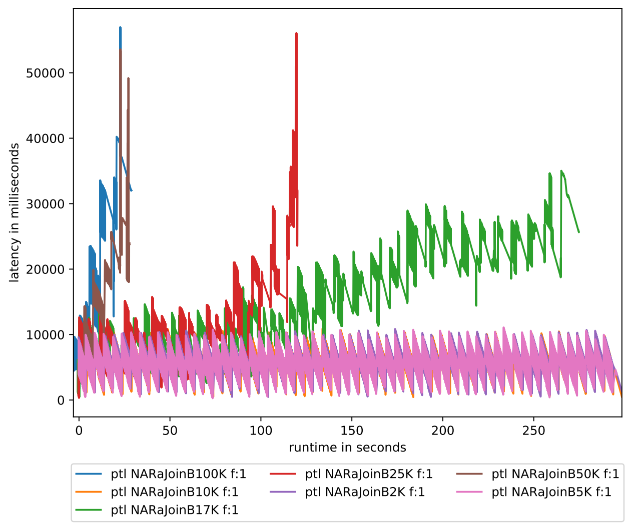

Figure 18: AJoin processing time latency comparison for different buffer sizes

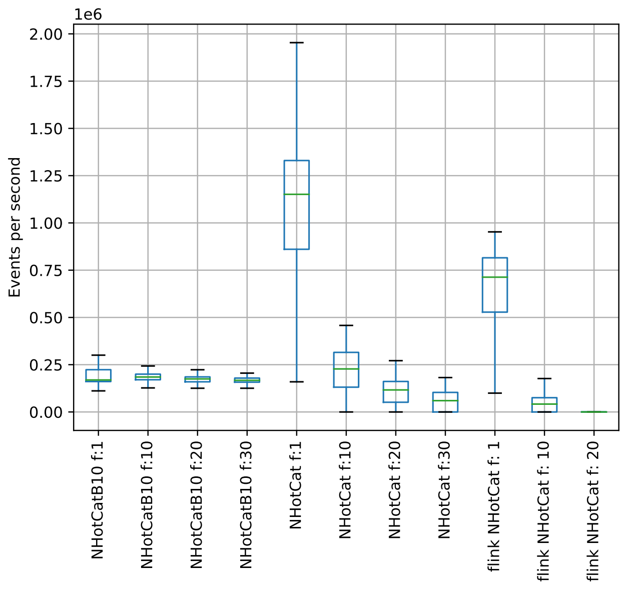

Figure 19: Throughput for Hottest Category query. The first four columns are using AJoin with a 10k buffer input size, the four middle columns use the standard join

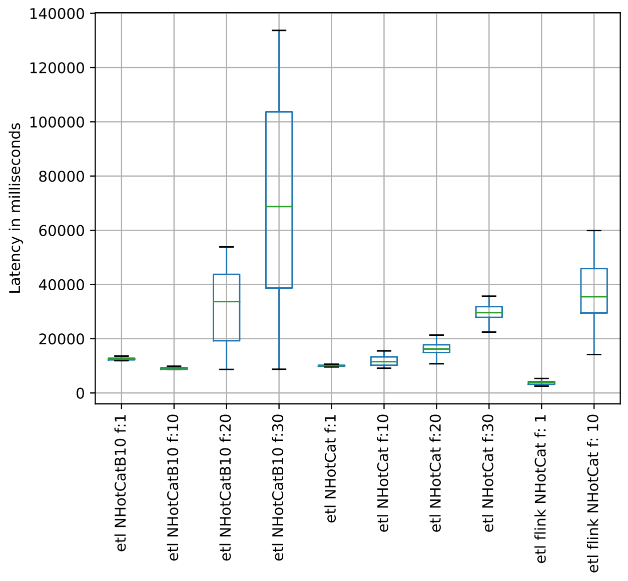

Figure 20: Latency for Hottest Category query. The first four columns are using AJoin with a 10k buffer input size, the four middle columns use the standard join

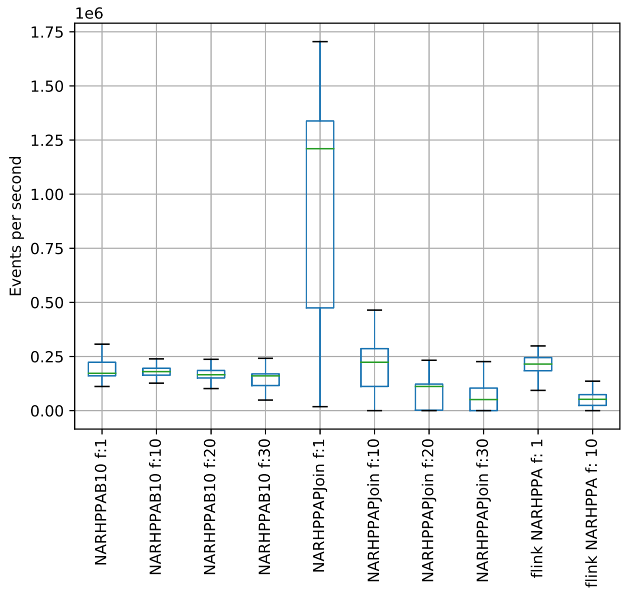

Figure 21: Throughput for highest price per auction query. The first four columns are using AJoin with a 10k buffer input size, the four middle columns use the standard join

6.2 Ad-hoc Query Performance

In this experiment, we want to evaluate the performance of HDES when adding queries in an ad-hoc fashion. To assess this capability, we add and remove X queries every Y second to the engine and monitor the performance. As Flink does not support ad-hoc query execution [7], we ran the tests only with HDES.

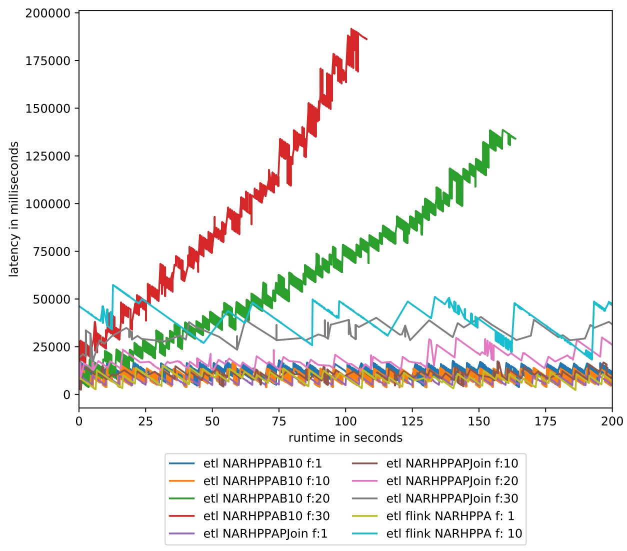

Figure 22: Excerpt of processing event-time latency for highest price per auction query

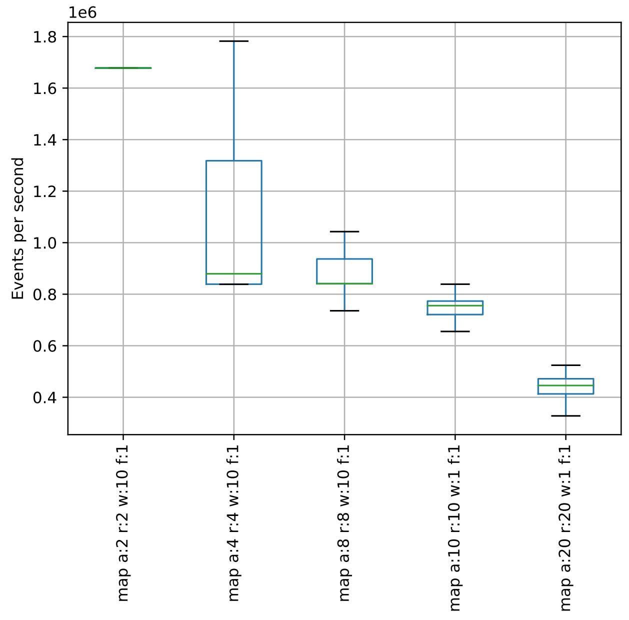

Figure 23: Throughput for map with basic data, when adding and removing queries every 10 seconds

6.2.1 Map Basic Data. As we would expect, Figure 23 shows us that the performance of the map operator decreases the more map-queries we add and remove in a ten-second interval. This is mainly owed to the fact that maps are not executed in parallel but sequentially inside a *SourceSlot*. When we compare the third and fourth

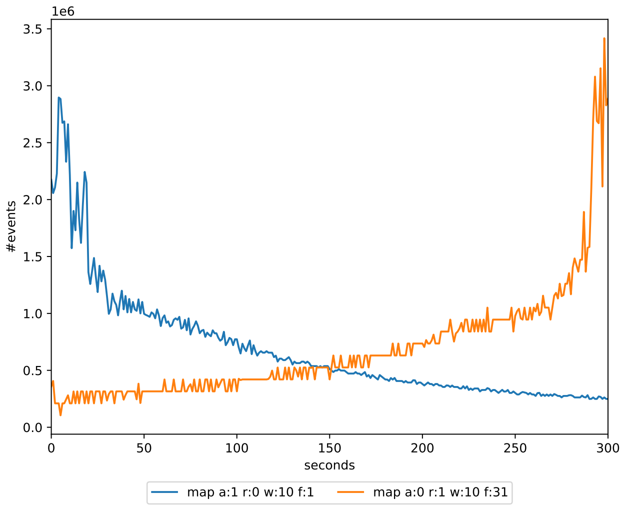

Figure 24: Throughput for map when continuously adding or removing a query per second

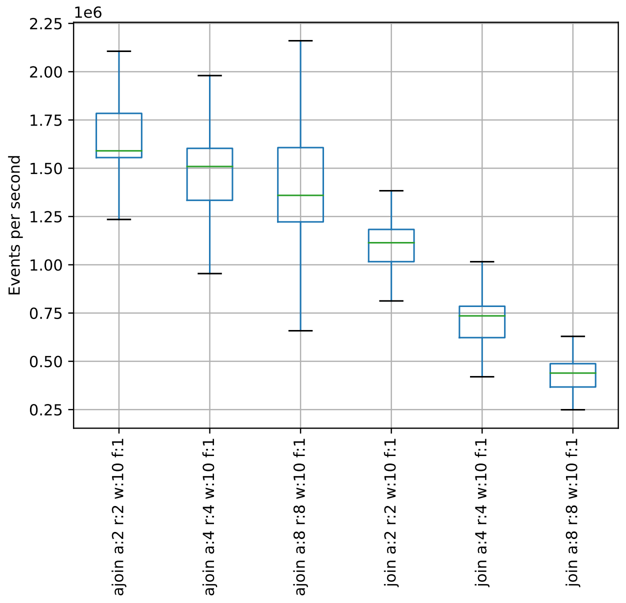

Figure 25: Throughput for join with basic data and ad-hoc queries

columns in Figure 23, we can observe that the frequency with which map queries are added and removed does not really have a large impact on the throughput. Even though we add and remove two more events per interval, the benchmark in the fourth column performs only slightly worse than the one in the third while having an interval of one second compared to ten. In Figure 24 we can see that—as anticipated—continuously adding or respectively removing a query over time exhibit mirrored throughput, i.e. the configuration that continuously adds queries starts out with a high throughput that decreases as more and more queries run in parallel and the configuration that continuously removes queries starts off with a low throughput that increases with each query that is removed. From this, we can conclude that add and remove have a similar amount of overhead and neither is more costly than the other.

6.2.2 Join Basic Data. The ad-hoc query benchmark clearly highlights AJoin's advantage. It achieves a much higher median throughput (Figure 25) in all three tests, albeit at the cost of higher processing-time latencies as Figure 27 shows. We already observed in Figure 13 that AJoin has a higher event-time latency than the standard join implementation for multi-query execution. Here we see that the same also applies to processing-time latency. This behavior is owed to the additional computation AJoin performs.

In Figure 26, we can see that AJoin also performs better for continuous query addition and removal, respectively. Even though the throughput for continuous query addition has more variance than the one of the standard join, it is significantly higher. Again, we can observe that the removal and addition lines meet at about 150 seconds, leading us to the conclusion that neither operation has more overhead than the other.

6.2.3 Join Nexmark Data. As already discussed in subsubsection 6.1.4, AJoin has a garbage collection problem when used with Nexmark data in our setup. For ad-hoc queries, this effect is further increased, as data structures that hold many objects are repeatedly removed

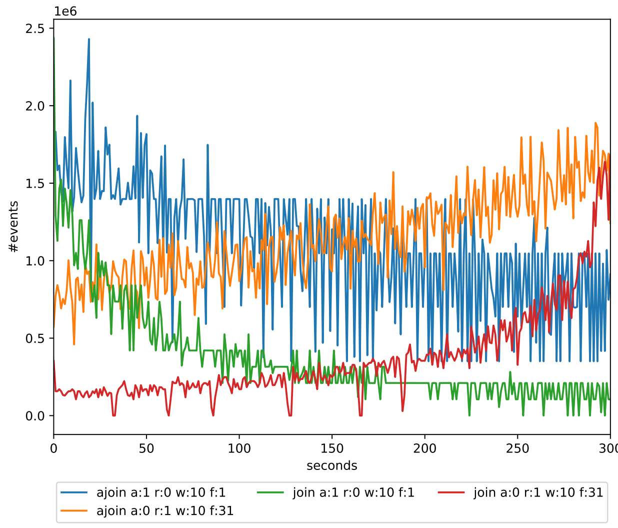

Figure 26: Throughput for join with basic data and continuous ad-hoc query addition/removal

and added, creating more references and thereby increasing the workload of the garbage collector. For ad-hoc queries, this also affects the standard join implementation. Therefore, we decided to reduce the network input buffer size to ten-thousand elements to prevent HDES from ingesting too many events. As we can see in Figure 28 the performance of the two join implementations is about equal, with AJoin having the slight edge in most cases. As

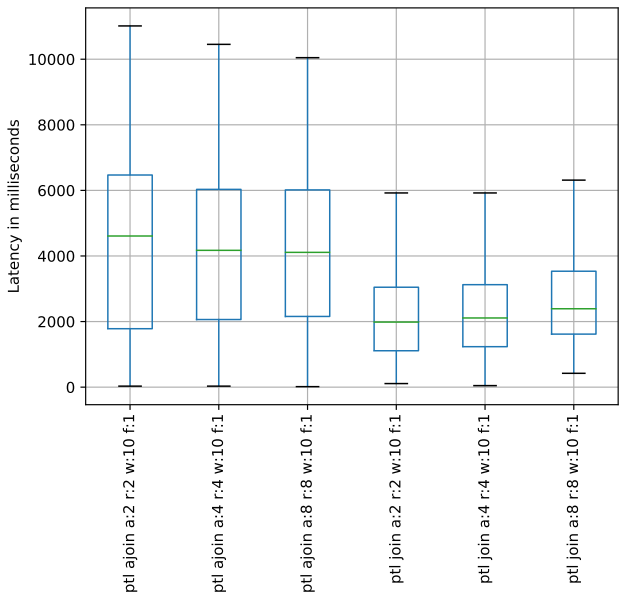

Figure 27: Processing-time latency for join with basic data

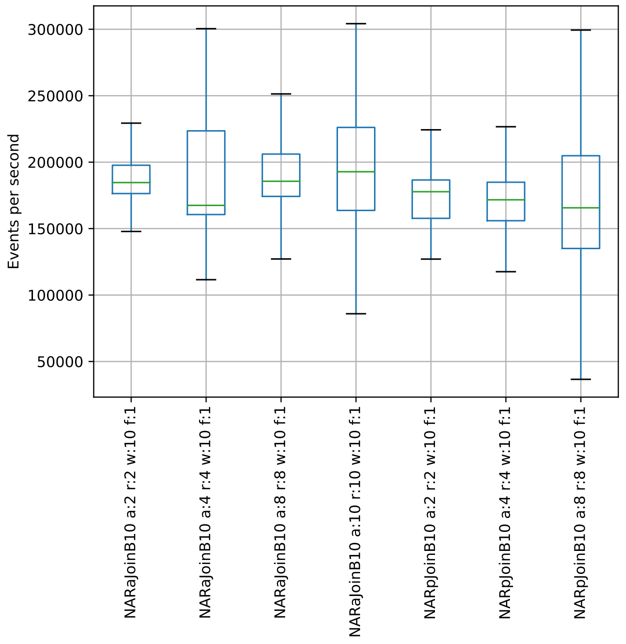

Figure 28: Throughput for join with Nexmark data and ad-hoc queries

usual, AJoins have a processing time latency which is about two seconds higher than that of the standard join.

In Figure 29, we can observe how AJoin profits from being optimized for performing multiple joins. Unlike the standard join implementation, which has the typical mirrored lines like in Figure 24 because it is not able to provide the full throughput rate if many queries are attached, AJoin can keep a more stable throughput rate with its less costly multi-query join.

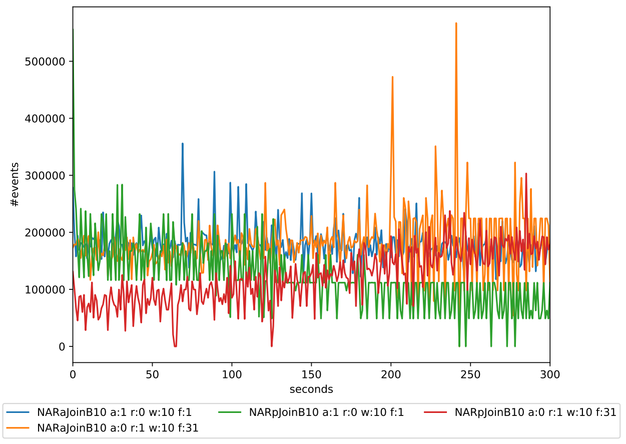

Figure 29: Throughput for join with Nexmark data and continuous ad-hoc query addition/removal

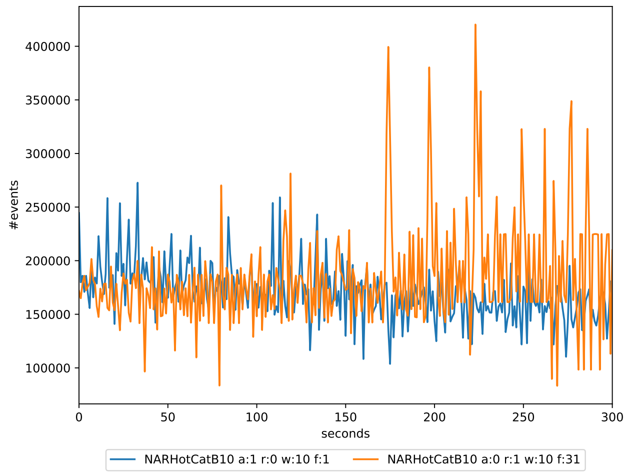

Figure 30: Throughput for the HotCat query when continuously adding or removing a query every 10 seconds

6.2.4 Hottest Category (HotCat) Nexmark Data. We executed this ad-hoc benchmark only with AJoin because the standard join queries overloaded the system in the ad-hoc scenario because they did not share any resources. For this query, a different picture compared to the simpler map and join queries emerges. In the latency diagram in Figure 31, we can see that the latency for the configuration which repeatedly removes queries starts low, even though the load at the start of the execution should be at its highest point as 31 queries run in parallel. However, it quickly rises and only begins to drop at around 150 seconds, which equates to a workload of 15 queries. That confirms what we already saw in Figure 20. Namely, 30 HotCat queries at once cannot be sustainably supported by our setup and lead to rising latencies. Conversely, we observe that the adding configuration does not increase latency in a major way until the

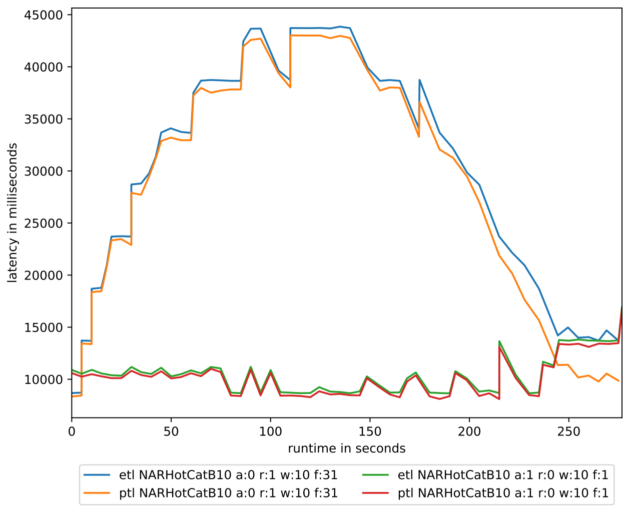

Figure 31: Latency for the HotCat query when continuously adding or removing a query every 10 seconds

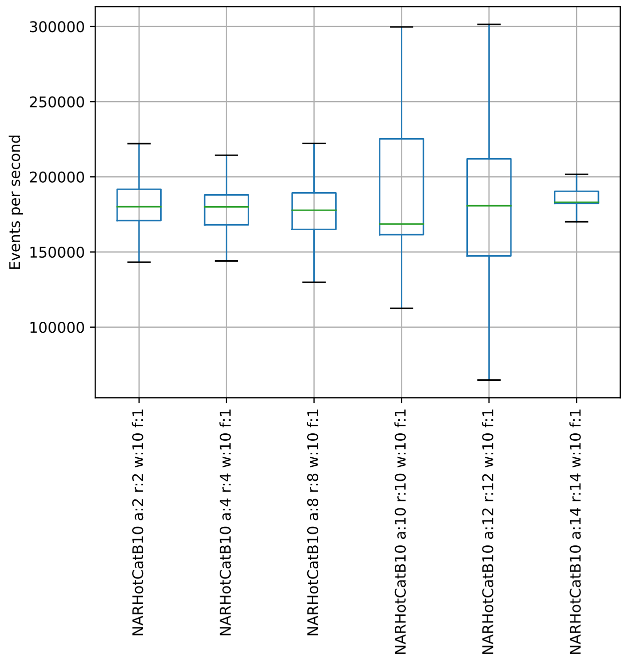

Figure 32: Throughput for the HotCat query when adding and removing a fixed number of queries every 10 seconds

210-second mark. This is striking since the removing configuration is struggling around the 90-second mark when it has to deal with the same amount of queries. One explanation might be that the latency does not increase as quickly if the load is increased gradually compared to when a high load exists from the start. Further,

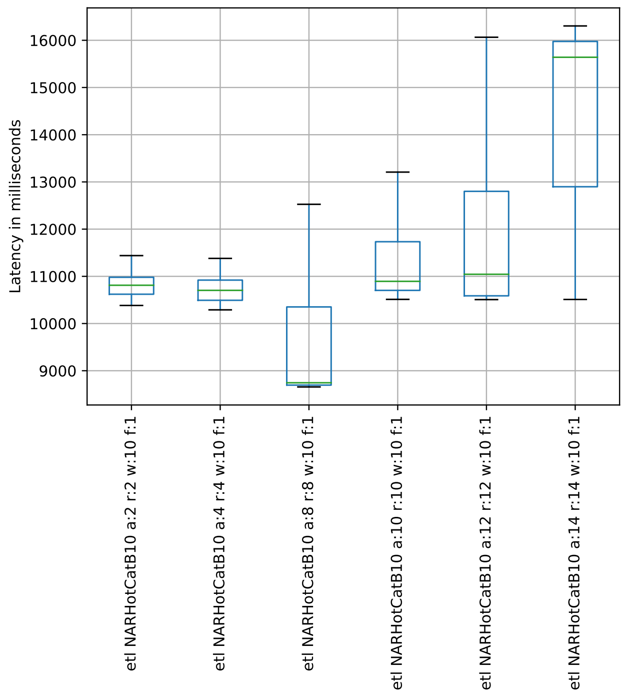

Figure 33: Event-time latency for the HotCat query when adding and removing a fixed number of queries every 10 seconds

a possible explanation is that already removed queries have a performance impact because the garbage collection has to check all objects referenced in their topologies.

When looking at the throughput diagram in Figure 30, we can see that both the adding and the removing have a roughly similar throughput at the beginning of the execution. However, as the computation advances and resources are getting sparse, the queries converge into their specific directions, meaning that the adding configuration is performing worse as more queries run simultaneously. The removing configuration is performing better as fewer queries run.

An interesting observation in this figure is that the adding configuration does not show the same peak performance in the beginning as the removing configuration shows in the end. The expectation would be that the queries would mirror their performances like show in Figure 24.

When looking at the scenario of adding and removing a fixed number of queries every 10 seconds illustrated in Figure 32 and Figure 33, we can see that AJoin is keeping the median throughput relatively stable independent of the number of queries. However, we see that the variance in both figures increases as the number of added and removed event grows larger than eight. Although we can spot some outliers, such as a:14 r:14 in the throughput box-plot and a:8 r:8 in the event time latency box-plot, the general trend remains true.

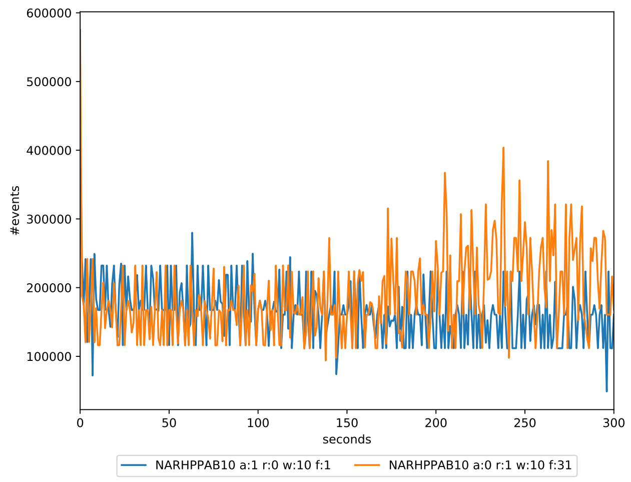

Figure 34: Throughput for the HPPA query when continuously adding or removing a query every 10 seconds

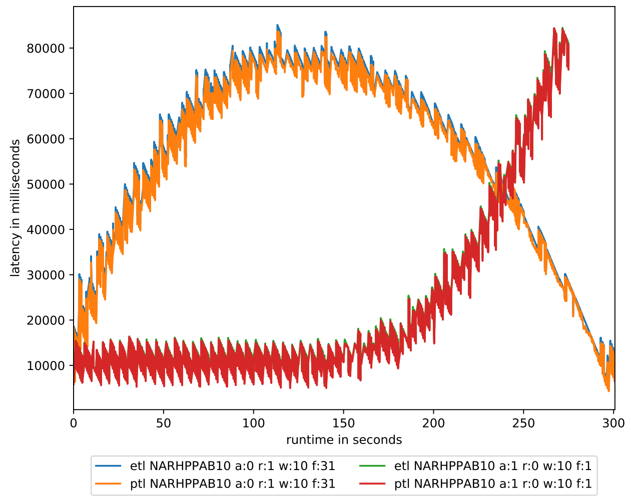

Figure 35: Latency for the HPPA query when continuously adding or removing a query every 10 seconds

6.2.5 Highest Price per Auction (HPPA) Nexmark Data. For the HPPA query, we see a picture comparable to the HotCat query. The major difference is that the adding configuration is behaving more reasonably as its latency increases as expected when the amount of queries surpasses the threshold of about 15 queries, as seen in Figure 35. However, we also see that HDES effectively stops processing for the last 20 seconds, as the workload is getting too high because too many queries have been added. Moreover, we can also observe in Figure 34 that the throughput of HPPA looks very similar to that of HotCat and also exhibits a stable throughput for the adding configuration and only a slight increase in throughput for the removing configuration.

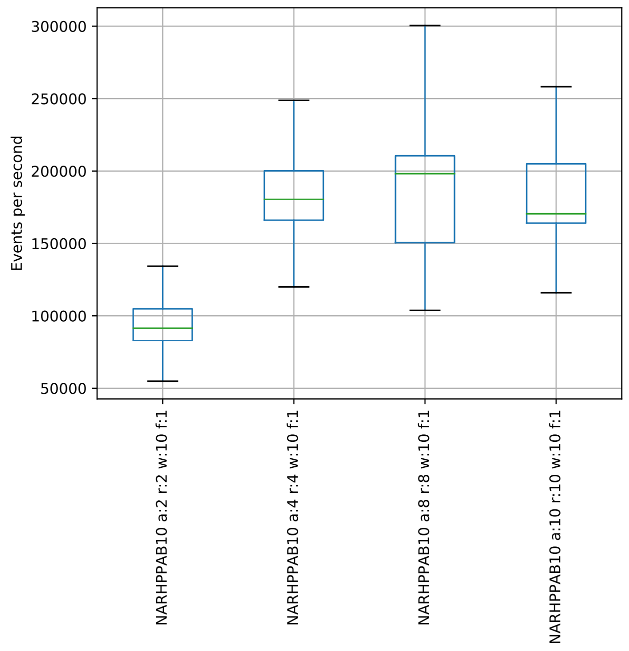

Figure 36: Throughput for the HPPA query when adding and removing a fixed number of queries every 10 seconds

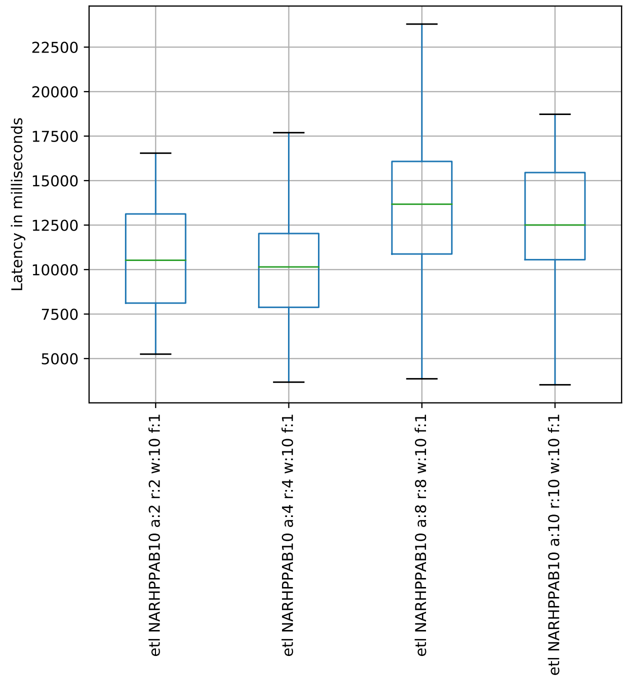

Figure 37: Event time latency for the HPPA query when adding and removing a fixed number of queries every 10 seconds

When considering the scenario of adding and removing a set of queries every 10 seconds, illustrated in Figure 36 and Figure 37, we can see that the performance increases slightly up to 8 queries and then decreases again. However, this configuration is also the configuration where the variance is at its peak. Interestingly, we see that adding and removing only two queries every ten seconds yields worse throughput than doing the same with more queries. However, this is likely an outlier. When adding and removing more than ten queries at a time, the engine does not provide satisfying results.

7 SUMMARY

The research stream processing engine HDES enables the user to define custom dataflows on processed events. Unlike other common stream processing engines, it supports the dynamic modification of the dataflow graph during runtime, without downtime. Furthermore, it implements cutting edge join sharing techniques that are not yet standard in common stream processing engines. The evaluation of queries that join two datastreams shows that join sharing significantly improves the performance for both multi-query processing and ad-hoc query processing, given similar join conditions.

The current implementation splits the transformation from a single query towards execution logic into multiple decoupled parts. This opens up the possibility to swap out parts of HDES to support future use cases.

When comparing the engine to Flink, we can see that HDES outperforms Flink in many use-cases. Due to the implemented optimizations, we were able to process query-sets that Flink cannot efficiently process. To sum up, HDES shows the performance gained by using state of the art stream processing optimizations and building a stream processing engine with ad-hoc query processing in mind. However, due to its prototype status, HDES still lacks essential features that would be desirable for production environments.

8 FUTURE WORK

Currently, HDES only supports single-node execution. As all traditional SPEs operate in a distributed setting, HDES should support distributed execution as well and thus, allowing a better comparison. In like manner, the original AJoin monitors the statistics of joins to rearrange the execution plan and dynamically scale operators horizontally and vertically. This functionality can be adopted by HDES.

Albeit HDES implementing AStream, a complete version and API support is missing. Together with a complementary benchmark, this is a feasible next step to further investigate ad-hoc query processing. Furthermore, we identified UDFs to complicate multi-query processing because their implementation is unknown. Hence, an operator containing a UDF like a filter or map has to be deployed for each query. We see two options to tackle this problem. The first one being an additional parsing step as used in SQL. The second approach can be to let the user tag each unique UDF with an ID. This could be integrated into a front-end for queries where users could submit new queries during runtime.

As shown, one major bottleneck we encountered was the garbage collection of the JVM. Therefore, we think a port to a system-level programming language like C++ or Rust would come with improved stability and performance.

REFERENCES

[1] 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.

[2] FasterXML. 2020. Jackson. https://github.com/FasterXML/jackson. (2020).

[3] Google. 2020. Gson. https://github.com/google/gson. (2020).

[4] Google. 2020. Protobuff. https://developers.google.com/protocol-buffers. (2020).

[5] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, 1507–1518.

[6] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2020. Ajoin: ad-hoc stream joins at scale. In *Proceedings of the VLDB Endowment* number 4. Volume 13.

[7] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. Astream: ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data.* ACM, 607–622.

[8] Esoteric Software. 2020. Kryo. https://github.com/EsotericSoftware/kryo. (2020).

[9] 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*, 147–156.

[10] 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.

[11] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 423–438.