Providing Streaming Joins as a Service at Facebook

Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching,
Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz,
Anirban Banerjee, Benjamin Heintz, Shridar Iyer, Anshul Jaiswal
Facebook Inc.

ABSTRACT

Stream processing applications reduce the latency of batch data pipelines and enable engineers to quickly identify production issues. Many times, a service can log data to distinct streams, even if they relate to the same real-world event (e.g., a search on Facebook's search bar). Furthermore, the logging of related events can appear on the server side with different delay, causing one stream to be significantly behind the other in terms of logged *event times* for a given log entry. To be able to stitch this information together with *low latency*, we need to be able to *join* two different streams where each stream may have its own characteristics regarding the degree in which its data is *out-of-order*. Doing so in a streaming fashion is challenging as a join operator consumes lots of memory, especially with significant data volumes. This paper describes an end-to-end streaming join service that addresses the challenges above through a streaming join operator that uses an adaptive stream synchronization algorithm that is able to handle the different distributions we observe in real-world streams regarding their event times. This synchronization scheme paces the parsing of new data and reduces overall operator memory footprint while still providing high accuracy. We have integrated this into a streaming SQL system and have successfully reduced the latency of several batch pipelines using this approach.

PVLDB Reference Format:
G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, A. Banerjee, B. Heintz, S. Iyer, A. Jaiswal. Providing Streaming Joins as a Service at Facebook. *PVLDB*, 11 (12): 1809-1821, 2018.
DOI: : https://doi.org/10.14778/3229863.3229869

1. INTRODUCTION

Many data analysis pipelines are expressed in SQL. Although less flexible than imperative APIs, using SQL enables developers to quickly bootstrap new analytics jobs with little learning effort. SQL queries can be executed in either batch mode (e.g., running in Presto [5] or Hive [32]), or in streaming mode (running in Puma [15] or other domain-specific frameworks). When running a query in batch mode, data must first be ingested into the Data Warehouse. Once a new partition of a table lands (daily or hourly), queries that depend on the new partition can be started. In contrast, when a query runs in a *streaming* fashion, data is continuously processed as it is acquired, and the output is generated as soon as all the input data required for its computation is available. The generated results can then be immediately consumed by other downstream applications or ingested into the Data Warehouse for other use.

A common operation used in many analytic workloads is to *join* different data sources. Doing so only after ingestion into the Warehouse incurs high-latency, which causes several problems for users. One such problem is the delay of computing derived data sets, as the computation of a join can only start after a partition has been fully ingested into the Warehouse. Another disadvantage of this scheme is that the results of joins cannot be used to power real-time metrics, used for detecting and solving production issues.

We have implemented a *streaming join operator* to reduce the latency of analytics that stitch together information from different sources. The operator focuses on joining tuples according to an equality predicate (i.e., keys) and a time proximity (i.e., time window). This handles the joining of streams in which their joinable events occur close in terms of *event time*, but that might be processed by the streaming application somewhat far apart (i.e., minutes to hours). *This can happen when tuples related to the same real-world event are logged into different streams with hours of delay.* For example, in mobile applications, event logging can be delayed until a device reconnects to the network via Wi-Fi after being connected via cellular network only. Using tuple event time is a distinction from our work and other time-based streaming join operators that use the time that the tuple gets processed to establish windows [2, 22].

We have integrated the join operator into Puma - Facebook's SQL-based stream processing service, so that users can easily spawn new automatically managed applications that join matching tuples as they are processed. With Puma, users can rely on deploying application updates without loss of in-flight data, tolerance to failures, scaling, monitoring, and alarming. Implementing event-time based joins in a streaming fashion as a service should balance output latency, join accuracy, and memory footprint. It also should consider that streams have different characteristics regarding the event time distributions of their events. Prior efforts on this area range from a best-effort wall-clock time joins [22]

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: : https://doi.org/10.14778/3229863.3229869

to the persistence of metadata associated with every joinable event on a replicated distributed store to ensure that all joinable events get matched [11]. Our solution falls in the middle, as we still provide a best-effort streaming join but aim at maximizing the join accuracy by pacing the consumption of the input streams based on the event-time of incoming tuples.

To increase the accuracy of a best-effort join operator while maintaining service stability, some of the techniques we have evaluated are: (i) estimation of the stream time based on the observed tuple event times to consume each of the input streams, (ii) bound the number of tuples associated with a given key, in order to limit the in-memory state of heavily skewed data, and (iii) leverage an intermediary persistent message bus to avoid checkpointing part of the in-memory state of the join operator.

To enable users to easily deploy streaming applications using joins, we have integrated the join operator into our streaming SQL language called PQL (Puma Query Language). Users create an application by specifying a join statement with an equality attribute and a window bound. The PQL compilation service compiles the query and ensures that allowed application updates can be deployed in a backward compatible manner without loss of in-flight data. After deployment, Puma is responsible for automatically scaling the application when it needs more resources than its current reservation and setting up alarms to notify users and service maintainers when failures or SLA violations occur.

The key contributions of this paper are: (i) a streaming join operator that leverages a stream synchronization scheme based on tuple event times to pace the parsing of new data and reduce memory consumption. This operator leverages the required processing semantics of certain applications [15] to provide a more efficient fault tolerance scheme while still achieving a high join matching rate; (ii) a query planner that produces streaming join plans that support application updates, ensuring users can modify their queries without causing the join operator to lose its internal state; and (iii) a stream time estimation scheme that automatically handles the variations on the distribution of event times observed in real-world streams and that achieves high join accuracy. To the best of our knowledge, we are the first to propose a streaming join operator that paces tuple processing to reduce resource consumption and to generate streaming SQL query plans with joins that support application updates.

2. SYSTEMS OVERVIEW

The streaming join service was implemented in the context of two of Facebook's stream processing platforms: Puma and Stylus. Both systems ingest data from Scribe [15] – a persistent message bus – and can later publish data back to Scribe, Scuba [8], or Hive.

2.1 Scribe

Scribe is a persistent and distributed messaging system that allows any application within Facebook to easily log events. New data written into Scribe can be read by a different process within a few seconds. When writing or reading data from Scribe, processes specify a *category* and a *bucket*. A category contains all the logs of a system that follow the same schema. A bucket allows the data to be sharded according to a criterion (e.g., an attribute value) and is equivalent to a shard or a partition. An application can achieve data parallelism by reading different buckets of the same category. In general, Scribe keeps data available for a couple of days.

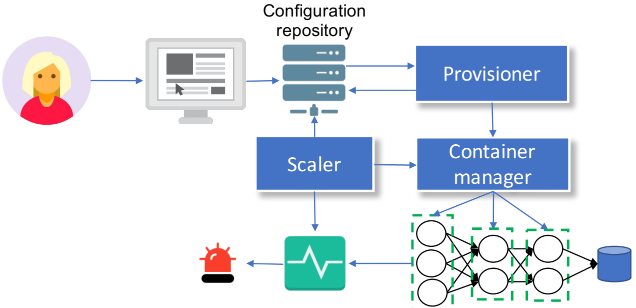


Figure 1: Puma's workflow.

2.2 Puma

Puma enables developers to write streaming applications written in PQL and easily deploy them to production. This is because Puma is offered as a service and automatically provides monitoring, alarming, fault-tolerance, and scaling.

Figure 1 shows Puma's workflow. Developers start by creating a Puma application in PQL (example in Figure 3) via the *Puma portal*. Once testing and code review have been completed, the PQL query is landed into Facebook's configuration repository [30]. Landing is allowed only if the query compiles and passes safety checks meant to ensure that it will not fail in production or hurt the performance of other apps (e.g., consuming too many resources).

The *provisioner service* monitors any application landing and constructs and deploys the application's physical plan. It first creates a directed acyclic graph (DAG) of operators to execute the query. It then identifies if it needs to create new or update existing production jobs to run the operators in the DAG. For new jobs, it creates a job configuration and contacts Facebook's container manager [28] to start it up. For existing jobs, it updates the job configuration with the new application information (e.g., version number and resource requirements) and issues an update to the container manager. The container manager is responsible for monitoring the liveness of jobs, propagating configuration updates upon a job restart, and assigning jobs to hosts according to the requested resources. The provisioner also creates any required Scribe category to execute the application's physical plan. This is because all communication between operators in a DAG happens through Scribe.

Once the application is running, it reports runtime information (e.g., tuple processing rate, backlog), which is used for monitoring and firing alarms. Depending on those runtime metrics, the *scaler* component may decide to scale up and down the jobs that compose an application. Scaling can happen both in terms of the number of tasks per job or the memory allocation per job. It does so by updating the job configuration and contacting the container manager to restart the updated job. If any of the current hosts can no longer accommodate the updated job's tasks with the new specified resource entitlement (e.g., task needs 10GB of memory, but only 5GB is available), then the container manager moves the job to a host with sufficient resources.

One characteristic of Puma is its enforcement of backward compatible application updates with respect to the internal state of stateful operators. When a user modifies an existing query, Puma ensures that the update can be performed without any loss of state. For example, when the query contains a statement for doing hourly window aggregations, user might want to add more aggregations to that same statement (e.g., count, sum, max). One simple way to carry out such an update is to drop any current aggregation value and restart the query. The disadvantage is that applications would lose the collected information from any ongoing aggregations. Puma ensures that (i) the new statement can be deployed in a backward compatible manner, and (ii) aggregations will appear to continue to be computed from the point where the application update operation started.

2.3 Stylus

Stylus [15] is a C++ framework for building stream processing operators and provides a lower level of abstraction than Puma. Stylus provides generic and flexible APIs for implementing different kinds of operators, such as stateless, stateful and monoid. The APIs and its specialized implementations for each kind of operator are also called a *Stylus engine*. A common use case for Stylus is to ingest tuples from a Scribe stream and output them to another Scribe stream or other data stores. Given Stylus is a C++ API, it is quite flexible for developers to implement various customized tuple transformations. Developers only need to focus on their business logic, while Stylus handles the common operations needed by most streaming operators, such as fault-tolerance, sharding, and scaling data processing.

Stylus allows an operator to read one or more buckets from a Scribe category. Stylus automatically splits the stream data into micro-batches and shards tuples into multiple threads for parallel processing. Stylus also provides operators the ability to replay a Scribe stream for an earlier point in time and persist any in-memory state to local (RocksDB [6]) and remote storage (HDFS). Given Stylus can both read and write to Scribe categories, operators can be easily plugged into a Puma generated DAG. The join operator is built on top of Stylus.

3. STREAMING JOIN SEMANTICS

Puma provides window-based equality joins, where the window is defined using a tuple attribute as its *event time*. Our design supports the join of two input streams, which we refer to as *left* and *right* streams. Tuples from the left stream are joined with tuples in the right stream when the specified key attribute matches and the timestamps of the tuples in the right stream fall within a *join window*, as defined below.

More specifically, *event time* is the creation time of a tuple. The event time has a delay when compared to the wall clock time that the tuple is processed by the streaming application. This delay varies for different tuple sources, and *tuples in the same stream are not usually ordered by event time*. Using tuple creation time for a streaming join is a distinction of our work when compared to other systems [2], which assign a timestamp when first processing the tuple. The *join window* is an interval on the right stream calculated from the event time of a left stream tuple. Tuples from the left stream are joined only with tuples on the right stream that fall within the calculated interval. Although the window specification is the same for every tuple, each tuple has its own window, which can be overlapping with the windows of other tuples. The *join key* is the tuple attributes that are used to do the join equality check. Left and right-side tuples only join when their join keys are the same.

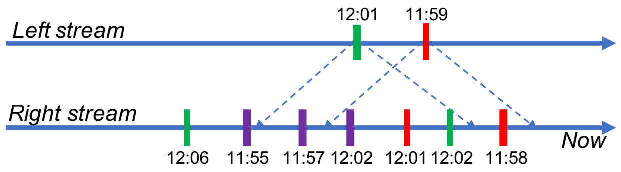


Figure 2: The join operator uses the event times of tuples on the left stream to calculate their join window on the right stream.

The *join result* is an inner join or a left outer join, and it outputs a projection of the attributes from the left and matching right tuples. In the left outer join case, right event attributes are filled with null values for failed matches. The join output can be *all matching* tuples within a window (1-to-n) or a *single* tuple within a window (1-to-1). The last case is useful when a single match is sufficient and enables reduced output latency, as the operator does not have to wait for the whole join window to be available before emitting a match.

Figure 2 shows an example of the join windows for two different events on the left stream (green and red) for a window interval of `[-3 minutes, +3 minutes]`. The color represents the *join key*, and the timestamp represents the tuple *event time*. Streams are *not ordered* by event time and the join window is computed based on the left tuple timestamp. Depending on the desired join output, the red left event (event time of 11:59) could match one or both of red tuples on the join window (event times of 12:01 and 11:58). If there is no assumption regarding the *time of a stream*, a tuple would have to *wait* forever for a match, as it is always possible to process a new tuple with a timestamp that would belong to a valid window interval. Deciding when a tuple should be emitted as a non-match or that all matches in a 1-to-n scenario can be emitted is related to how we do processing time estimation and stream synchronization. This process is described in more detail in Sections 5.2 and 5.3.

4. QUERY LANGUAGE AND PLANNING

This section describes the language-level constructs available for developers, and how Puma's planner builds a DAG of operators for streaming join queries. Left outer joins match a tuple from the left stream with all tuples from the right stream that match the join condition within the specified time window (Figure 2). We plan to support other types of streaming joins in the future.

4.1 Streaming Join Query

Users build a streaming application by writing a query in PQL, which is similar to SQL but designed to target streaming use cases. A query is a sequence of 4 kinds of statements:

1. create application - specifies a unique application name within Puma's namespace;

2. create input table - names an input stream and describes its schema. It indicates which Scribe *category* the data must be consumed from;

3. create view - specifies a stream transformation via expressions, user-defined functions, column projection, and tuple filtering. A view can specify joins between two streams;

4. create table - describes additional transformations on the data from a view, including time-based aggregations. It also includes information about where to store the results of the table's transformations (e.g., Hive). Depending on the storage chosen, users can specify a sharding expression.

A PQL query must have a single *create application* statement, but it can have an unbounded number of create input table, create view, and create table statements. A user can assemble a DAG by chaining the statements above.

Figure 3 shows a PQL query with a view containing a streaming join clause. In this example, we specify that the `left` stream has 4 attributes and consumes data from a Scribe category named *left* (lines 02-04). Similarly, the `right` stream has 4 attributes and reads data from the *right* category (lines 06-08). Users do not need to specify types when declaring a schema. Puma does type inference depending on the expressions and functions that the attribute is used in. Users can do explicit casting when necessary. The join view specification indicates the left stream (line 16), the right stream (line 17), and the equality expression (line 18). The window itself is expressed with the `BETWEEN` function and using intervals on the timestamp attributes (lines 19-21). This example shows an interval of 6 minutes. The lower and upper bounds can have different sizes and can be defined in hours or seconds. The timestamp attribute for each stream is inferred from the `BETWEEN` function called from the `ON` expression. The output of the application is published to a Scribe category named *result* (lines 23-28).

```
00: CREATE APPLICATION sample_app;
01:
02: CREATE INPUT TABLE left (
03:  eventtime, key, dim_one, metric
04: ) FROM SCRIBE("left");
05:
06: CREATE INPUT TABLE right (
07:    eventtime, key, dim_two, dim_three
08: ) FROM SCRIBE("right");
09:
10: CREATE VIEW joined_streams AS
11:   SELECT
12:     l.eventtime AS eventtime, l.key AS key,
13:     l.dim_one AS dim_one, r.dim_two AS dim_two,
14:     COALESCE(r.dim_three, "example") AS dim_three,
15:     ABS(l.metric) AS metric
16:   FROM left AS l
17:   LEFT OUTER JOIN right AS r
18:     ON (l.key = r.key) AND
19:       (r.eventtime BETWEEN
20:         l.eventtime - INTERVAL ‘3 minutes’ AND
21:         l.eventtime + INTERVAL ‘3 minutes’);
22:
23: CREATE TABLE result AS
24:   SELECT
25:     eventtime, key, dim_one, dim_two,
26:     dim_three, metric
27:   FROM joined_streams
28:   STORAGE SCRIBE (category = "result");
```

Figure 3: PQL query with left outer streaming join and a time window of 6 minutes.

4.2 Query Planning

Given a PQL query, Puma compiles it and determines its execution plan. The planner itself has two main constraints. The first is to divide the work across operators according to their capabilities. The second is to generate a plan that is backwards compatible with existing system data prior to the update, which includes state and in-transit data. The latter fulfills users' expectations regarding application updates. Even though an application is being restarted (e.g., adding a new filtering condition), Puma attempts to reduce the amount of data duplication and data loss for end users. Puma ensures that any in-memory state of stateful operators and in-transit data being used before an update are still readable after the update takes place. This is not enforced unless explicitly requested in the PQL query change, for example by removing existing views or output tables.

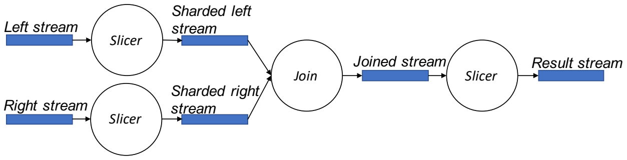


Figure 4: Logical operator graph for join query.

For joins, the Puma planner targets two operators:

1. Slicer - This is a Puma operator similar to a *mapper* in a MapReduce [17] system. Slicers can ingest data from Scribe, evaluate expressions, do tuple filtering, project columns, do stream sharding according to an expression, and write data to to Scribe, Hive, and other storage sinks.

2. Join - This is a Stylus operator and it was developed as part of this work. The join operator ingests data from two Scribe streams, maintains the join windows, executes the join logic, and outputs the result tuples into another Scribe stream. For historical reasons, Stylus operators do not share expression evaluation logic with Puma.

Example plan. Figure 4 shows the resulting plan of such a query. In short, the query is planned as (i) a *left slicer* that ingests the Scribe category (blue box) specified on the left side of the join and shards it according to the equality attribute, (ii) a *right slicer* with the same functionality as the left slicer but consuming the right-side Scribe category instead, (iii) a *join* operator that consumes the output Scribe category of both the left and right slicers and generates an output Scribe stream, and (iv) a *post-join slicer*, which does any other required stream transformation and writes it into the specified output system (e.g., Scribe). The figure illustrates the *logical plan* of the query. During execution, there are several parallel instances of each operator, where each instance reads one or more buckets of the input Scribe category. The degree of parallelism of each operator depends on the input traffic of its input Scribe category. This degree gets adjusted dynamically by the scaler component throughout the day and is independent for each operator.

PQL rewriting. Planning a streaming join query as described in Figure 3 is equivalent to re-writing it as a PQL query with the input table, view, and table statements expanded into multiple simpler statements. We can do so leveraging an extensible part of the Puma compiler and planner called the *PQL transformer*. With the transformer, we can do a sequence of PQL rewritings, which enables us to add new features without making significant changes to other parts of Puma.

For streaming joins, we have two rewritings. The first one is straightforward and it targets at eliminating the table aliasing specified in the streaming join view (lines 16-17 in Figure 3). It generates a new PQL query in which any reference to an alias is replaced by a reference to the full input table name (e.g., `l.eventtime AS eventtime` becomes `left.eventtime AS eventtime`). The second transformation is more elaborate, and it generates a PQL query that explicitly translates into a sequence of two slicers, a join operator, and a third slicer. This is equivalent to 4 groups of *create input table*, *create view*, and *create table* statements: 1 for each input stream, 1 for joining the streams, and 1 post-processing the join output stream. This ensures that each sequence is assigned to exactly one operator.

Left-side stream transformation. The objective of this transformation is to generate an equivalent PQL segment that is able to pre-compute the join equality and timestamp expressions, project and/or compute expressions on top of any left-side attribute used by the streaming join view, and shard the output stream by the equality attribute. The input table for this segment remains the same as in the original PQL query, as it just indicates the schema of the original input stream. The view statement selects first the expressions computing the timestamp and equality referred to in the `ON` clause (Figure 3 uses the raw values of `eventtime` and `key` only). It then projects any attributes, constants, or expressions that refers exclusively to attributes of the left stream. Expressions referring to multiple tables are currently not allowed. Given this is a left outer join, it is safe to evaluate expressions before performing the join. Finally, the output table statement selects all attributes from the view and writes them to a new *intermediary* Scribe category sharded by the equality attribute.

Right-side stream transformation. This transformation is similar to the left side stream with one key difference. Expressions involving attributes from the right-side stream are not evaluated until after the join operator. This is because the outcome of the join may influence the result of the expression.

Join view transformation. Given the expressions from the original join view are evaluated by the pre or post-join slicers, the transformed join view only refers to the results of evaluating the right and left side stream transformations, available in the Scribe categories they write to. The join operator writes its results to another Scribe category.

Post join transformation. The objective of this operator is to evaluate expressions involving attributes from the right side stream, as well as execute any other expressions specified in the original `CREATE TABLE` statement. The transformed `CREATE TABLE` statement also includes the specification of the storage to publish to from the original statement.

Backward compatibility. As described above, the need to make the generated plan being backward compatible comes from the expectations users have for application updates. Such updates should not cause massive data duplication or data loss. When assembling DAGs, such as the one in streaming joins, Puma uses Scribe categories as the communication substrate between stages. This choice enables operators to be less coupled and simplifies fault-tolerance, scalability, debugging, monitoring and alerting [15]. Due to the nature of Scribe, after an update, data from the previous version of an application may exist in the message bus, waiting to be processed. As a result, to enable backward compatible updates, we need to enforce that the planner creates an execution plan that extends the both the *wire* format and the *state* preserved by stateful operators in a compatible way (e.g., new columns are appended to the end stream schema). The wire format is enforced on the input and output categories of the join, and the state format is enforced for the join operator itself.

To make the plan backward compatible, we limit the changes that the user can perform in the streaming join view and forgo some possible optimizations to make application updates more flexible. Two examples of rules an update must follow are (i) preservation of the join equality expression, as its modification can cause resharding of the Scribe categories; and (ii) projection of new attributes must be specified at the end of the select list, as adding an attribute in the middle of the select list would cause the join operator to consume old attribute values as the value of a different attribute - both for tuples in the wire and tuples preserved as join operator state. This is required because the schema of Scribe categories is based on order. One example of an optimization we forgo is the possibility of projecting constants only at the final stage of the DAG. The planner ends up projecting constant expressions specified at the streaming join view to be performed by the left side slicer, as we need to maintain the wire format. Doing so enables users to later change the specification of the constant expression. Another example of optimization we do not do is to automatically remove projected attributes that do not get used by downstream operators. Automatically removing them would also cause a change in the wire format, which we must maintain for compatibility.

Update rules are enforced by the PQL compiler. Any violation is displayed to the developer coupled with alternatives for how to proceed with such an update. If the updated streaming join is significantly different, then users have the option of creating a view with a new name and deleting the old one. In such cases, the users are aware that any in-flight data will get discarded. The rules fit most of the update use cases, as often times developers just want to project a new attribute from the input streams.

5. JOIN OPERATOR

As shown in Figure 4, the join operator ingests the data computed by the left and right slicers sharded according to the specified equality attribute. The join operator processes data for both the left and right streams corresponding to the same key space (i.e., belonging to the same Scribe bucket). As a result, all the join matching decisions are performed locally within a single process.

Overall, our join operator continuously builds an in-memory hash table for the right stream, keeping all tuples belonging to the specified time window. For every tuple on the left stream, the operator performs a look up to find the events with the same key (i.e., hash join [34]) and falling into the join window as calculated from the tuple event time. Once matching tuples are identified, the operator calls a function that implements the different join scenarios, such as 1-to-1 join, or 1-to-n join, as described in Section 3.

The sections below describe in more detail how the operator is implemented on top of Stylus and how it synchronizes the two input streams, so that it can achieve a high join matching rate while limiting memory consumption.

5.1 Overview

We implemented the join operator on top of Stylus. By doing so, we inherit all its built-in features, such as scalability, fault-tolerance, and parallelism. Figure 5 shows the overall structure of the join operator. It consists of 3 components: (i) a stateful engine, used to process the left stream, (ii) a stateless engine, processing the right stream, and (iii) a coordinator, to bridge the two engines together.

Left stateful engine. This engine processes the left stream and it stores the incoming tuples in a buffer. The buffer is encapsulated into Stylus states, which are used by the framework to automatically do incremental state checkpointing. State is periodically persisted into a local RocksDB [6] instance and is replicated asynchronously to remote HDFS clusters in order to tolerate host and datacenter failures. When a tuple in the left stream is processed, the operator looks for matching tuples on the right join window. When a lookup succeeds, it calls a function to generate the join result. If there are no matches, the tuple is retained in the buffer to retry later. Once a match succeeds or permanently fails (i.e., the window has closed and there is no match), the tuple may be emitted as a non-match (in the case of a left outer join) and gets evicted from the buffer. Note that input tuple order is not preserved on the output stream. Order is not a requirement for our applications and enables us to trim the buffer more aggressively.

Right stateless engine. This engine ingests the right stream and maintains a window of tuples on the right stream that matches the specified join window for the incoming left stream tuples. The engine stores incoming tuples in an in-memory hash map on the joining attribute. The engine provides a lookup API to retrieve all matching events. The window is periodically trimmed when existing tuples fall out of the join window. This happens when the *estimated stream processing time* moves forward. Note that even though the engine maintains an in-memory state, the engine is *stateless* with respect to the Stylus framework. This is because the join window does not have to be checkpointed to local or remote storage. Here, we leverage the fact that (i) certain applications do not need exactly-once processing semantics, and (ii) that we use a persistent message bus (Scribe) for inter-operator communication. With that, we can easily replay data to re-populate the window upon an operator restart. This simplifies our implementation and the maintenance of our service, as the overhead of data backup is avoided. Based on our current deployment and applications, even if the window has several hours of data, it only takes a couple of minutes to recover a full window.

Join coordinator. The coordinator brings both engines together by providing APIs for the left engine to look up matching tuples in the right engine, and for both engines to query each other's stream time estimations. The latter is used for stream synchronization. The APIs effectively hide the implementation of the 2 engines from each other, decoupling the designs of the two engines.

The startup of a join operator occurs as follows: (i) the left engine reads the last saved checkpoint from either local or remote storage, and reconstructs its state; (ii) the coordinator pauses the left engine; (iii) the right engine is initialized and replays data from Scribe; (iv) after the right engine's current stream time is fully caught up to the left engine's, the coordinator resumes the left engine.

Pacing input of tuples for the join operator can be seen as similar to the pull-based engines that request the next tuple from its upstream operators on demand [21]. Our join operator always reads data that is already materialized in Scribe and does so by considering the estimated stream processing time.

5.2 Synchronizing Two Streams

Some streaming join algorithms are built on the assumption that tuples from opposing streams with similar event times will arrive in the system close in time. A common approach is to assign a timestamp a tuple when it is first ingested by the source operators of a streaming application [2]. However, in our scenarios, tuples with similar event times in different streams can arrive at significantly different times. One cause of this is the way that logging works in different platforms; data from mobile clients can be uploaded to the server-side hours after an event occurs. Another reason is processing delays in upstream systems. For example, systems that do sessionization emit events several minutes after ingestion (e.g., 30 minutes). Furthermore, failures in upstream systems can be a source of processing delay.

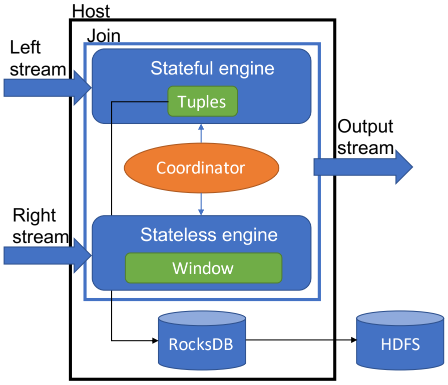


Figure 5: The join operator overview.

One simple solution to cover late data is to use a large join window. There are, however, drawbacks to this approach. First, it leads to inefficient memory utilization, as the operator may end up buffering data that does not get used for current matches. Second, using a fixed length join window falls apart when the stream delay changes over time. This is common especially when upstream systems have a backlog. Since the join semantics we offer is based on tuple event time, one natural solution is to align the *left* and *right* ingestion by the event time of their tuples. Our join operator solves these challenges by synchronizing the input stream ingestion based on a *dynamically estimated processing time*. In this way, tuples that are in memory overlap in terms of their event time, increasing the chance of tuple matches.

The estimated processing time, or PT for short, is a *low watermark* of the event times that have been processed in one stream, i.e., there is a high chance that a new incoming tuple will have an event time greater than the currently estimated PT. Details on the calculation of the processing time can be found in Section 5.3. In our operator, the join operator executes the stream synchronization. It does so by pausing the stream that has its PT too far ahead of the other. The synchronization uses the following formula: $left\_PT + window\_upper\_boundary = right\_PT$, where $left\_PT$ represents the processing time estimated for the left stream, $right\_PT$ is the processing time for the right stream, and $window\_upper\_boundary$ is the upper boundary of the window. If the window is specified as [-3 minutes, + 3 minutes], then the upper boundary is +3 minutes.

Figure 6 illustrates the formula rationale. Green boxes illustrate tuples currently maintained from both streams. All tuples have event times that are smaller than the currently estimated PT. The triangle represents newly processed tuples that have event times that are greater than the current PT. To give tuples a high chance of a match, we keep a full window on the right stream according to the estimated

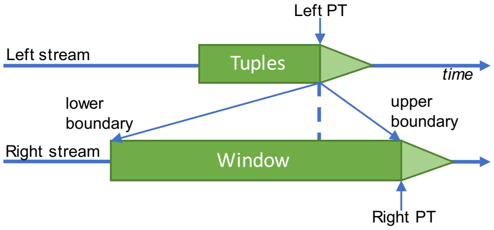


Figure 6: Buffers maintained by the synchronization algorithm to yield high join matching rates.

left PT. This means that we have nearly all tuples in the expected range to do a match. Note that the PT indicates that it is very unlikely that new events will have a timestamp lower than the estimated value. As a result, when PT moves forward, it is safe to discard tuples from the lower boundary of the window, as it is unlikely that they will yield matches.

Let's assume that we use p to compute processing time, where p is measured as a percentile of the event time distribution. For example, if the processing time is set to be the 1% lower watermark, p is equal to 1. The same p value is used for both the left and right streams. For a limited period of time, it is estimated that the buffer maintained in the right stream covers $(100-p)\%$ of the tuples with an event time that is within the left PT's join window and also smaller than the right PT (i.e., tuples in the green box in Figure 6). As a result, assuming that the PT estimation is perfect, a lookup operation using the left PT would miss at most $p\%$ of the tuples from the right stream buffer. To calculate the join success rate, let's look at the two cases that can happen when a new tuple comes in the left stream.

The new left tuple's event time can be smaller than the current left PT. Based on the PT definition, this case can occur with a probability of $p\%$ and it can have up to a 100% chance to miss matches from the right stream's buffer. The max join failure percentage for this case is $p\% * 100\%$;

The new left tuple's event time can be equal to or greater than the current left PT. The probability of this case is $(100-p)\%$. Joins can happen until the left PT becomes bigger than this tuple's event time, and the join failure possibility for tuples with an event time equal to the left PT is $p\%$. As a result, it has at most $p\%$ chance to miss matches. The max join failure percentage for this case is $(100-p)\% * p\%$.

The minimum join success percentage can be described as 100% minus the maximum failure percentages above.

$min\_join\_success\_percentage = 100\% - p\% * 100\% - (100-p)\% * p\%$

When p is equal to 1, the minimum join success rate is approximately 98%. In practice, the PT estimation is not perfect. Our experience has shown that we are able to achieve matching rates that are very close to the ideal case.

The join operator synchronizes both streams by pacing the ingestion of new data based on the PT estimates as to maintain the buffers according to the constraint described above. The operator *pauses the ingestion of data from the right stream* when the difference between the right PT and the left PT is bigger than the upper window boundary. This means that if the operator ingests more data from the right stream, it will end up evicting tuples that could still be matched with new left stream events. The operator *pauses the ingestion of data from the left stream* when the left PT plus the upper boundary of the window specification is ahead of the right PT. This ensures that the operator is not buffering tuples unnecessarily and increasing memory consumption, as these would be tuples that will likely have to be retried to join later.

In our implementation, the synchronization of the left and right streams is made when processing new data and are not strictly aligned. As a result, we end up buffering some extra minutes of data for the right stream window. This is in addition to the window size buffer. The extra buffer avoids frequent stream pauses and reduces join failure rates.

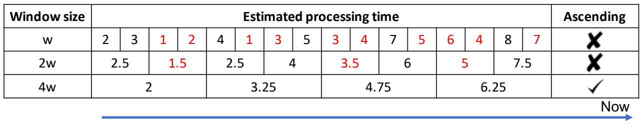
5.3 Processing Time Calculation

As described in Section 5.2, *processing time* (PT) indicates the estimated time of a stream, i.e., a time for which we estimate that there will be no new tuples whose *event time* attribute has a value that is smaller than the processing time. In Stylus, the processing time is implemented as a percentile of the processed event times, similar to other systems [9, 10]. As tuples are not ordered by their event times in a stream and we can only use event times that have been observed so far, we must calculate PT based on a statistic on top of the processed event time distribution (e.g., percentile, or average). To do so, the Stylus engine splits a stream's events into *micro-batches* with configurable size (e.g., 2 seconds or every 1000 events). It then uses the chosen statistic to calculate the PT for the batch. In other words, it makes a histogram of the raw event times for each batch. For an x percentile statistic, it then assumes that any future micro-batch will have at most $x\%$ of events with an event time that is smaller than PT.

The assumption above is based on the observation that the event times of tuples in a stream tend to increase over time. As a result, as long as we use a statistic that reflects that trend, PT should also increase. The problem though is to understand the granularity (i.e., the window size) over which to make PT calculations, so that the PT can continuously increase. If we observe events over a small window, their event times are likely to appear very disordered. However, if we observe them over a larger window, new events will tend to have event times that are more recent than previously processed tuples. The *ideal* size of the window is stream dependent, as the distribution of event times depends on how the logging happens. Event time distributions can differ greatly between streams containing *server side* events and streams containing *client side* events.

It is expected that if we have a very large window, we can observe more events and therefore have a higher confidence on the PT estimation. However, having a large window leads to increased *latency*, as one needs to wait longer for computing subsequent PTs. Furthermore, the PT itself will tend to progress slower, as the statistic will consider a larger number of older tuples. As a result, the PT calculation must balance *accuracy* and *latency*. Accuracy means how well the PT estimation can fulfill the low watermark assumption that at most $x\%$ of events processed after a given PT will have an event time smaller than it. Latency means the delta between the PT estimate and the wall-clock time, which, in the case of the join operator, manifests as the tuple output delay. If higher accuracy is preferred, the window of observation must be larger, resulting in a higher latency.

Our PT calculation algorithm aims at finding the minimum window size that still generates ascending PTs. The window size indirectly represents how out-of-order a stream is. This means that if the stream is somewhat ordered, we



Window size	Estimated processing time																Ascending
w	2	3	1	2	4	1	3	5	3	4	7	5	6	4	8	7	✘
2w	2.5		1.5		2.5		4		3.5		6		5		7.5		✘
4w	2				3.25				4.75				6.25				✓

Figure 7: PT estimation on small window sizes results in PT not being an ascending sequence. Increasing the window size enables the PT value to capture the true time trend.

can generate ascending PTs with a small window. If the stream has data that is very disordered, then we need a larger window. The steps to calculate the PT are as follows: (i) split the stream into micro-batches of events. This is provided by Stylus, which was already designed to process data from Scribe in micro-batches; (ii) for each micro-batch, calculate a PT using a statistic (e.g., percentile) over the observed event-time distribution. The value calculated for each micro-batch is used to find the ascending PT series; (iii) if the PTs of the micro-batches are not ascending, the adjacent windows are continuously merged so that we can obtain an estimation for a larger window. The window size is the main knob to balance between accuracy and latency. Our algorithm *dynamically* chooses the window size, but we have knobs in to manually tune the window parameter and customize the operator for different use cases if necessary.

Figure 7 illustrates how the window size is adjusted to find an ascending PT sequence. Each cell in the table represents the PT value computed using a statistic over the event times of the tuples processed within a given time window of the specified size (w, $2w$, or $4w$). For simplicity, the figure shows PT as a small integer and uses average as a statistic to estimate the window PT. When looking at the PT value for a small window (w), one can see it does not form an ascending PT sequence (red values). As the window size increases ($2w$), the trend of increasing PTs starts showing up, but still cannot form an ascending sequence. When the window size increases to $4w$, an ascending sequence is found. After that, $4w$ is used as the window size for PT computation and PT is set to be the result of the PT calculation over the most recent window (6.25 in this case).

In our implementation of the method above, we fix the number of windows for a PT calculation (e.g., 4 windows of observation) and increase their size by fitting more event-time values into each window as more tuples are consumed. We also use a maximum window size to limit the memory growth of the operator.

6. STREAMING JOINS AS A SERVICE

This section elaborates on some of Puma's features that enable PQL queries written by developers to run smoothly in production.

6.1 Fault Tolerance

Fault tolerance is desirable in many scenarios. Examples include application updates, Puma and Stylus software updates, planned or unplanned hardware maintenance, among others. The join operator uses both replay and checkpoint-based approaches. As the operator is designed to maintain the whole window of data in-memory for the right stream, checkpointing the operator state is expensive, as the window can grow to tens of gigabytes. To solve this, we leverage the fact that the input stream of the join operator is stored in Scribe. On a crash, the operator can easily repopulate the state by rereading the data from Scribe.

For the left stream, the join operator ingests a small chunk of data into memory each time and then does a look up. After the two streams are properly synchronized, most of the events either succeed or fail by having their join windows fully covered by the right stream's buffer. As a result, only a small percentage of events need to be kept in memory for later retry. Checkpointing such in-memory state is much more affordable when compared to the right in-memory state. The join operator checkpoints its state to local disk using RocksDB [6]. The state gets replicated to multiple HDFS clusters in different data centers. Saving state to RocksDB is synchronous, while backing up to HDFS is asynchronous (except during a graceful shutdown), as the HDFS write latency is higher than writing to local disk.

To optimize the access latency to operator state, jobs running the join operator have both host-level and region-level stickiness during scheduling. Host-level stickiness is conceptually the same as *host affinity* in Samza [29]. A join operator instance is always scheduled on the same host as it was before so that it can recover quickly by reading states locally. When this is not possible (e.g., the host needs to be taken away for maintenance), the scheduler picks a host from a data center where an HDFS snapshot can be found.

6.2 Scaling

The system must account for changes in stream traffic and data characteristics over time. The join operator is horizontally scalable. As shown in Figure 4, the output streams of the slicer operators are partitioned into disjoint substreams based on the join key. The substreams are written into different Scribe buckets, which are then processed by a join operator instance. In each substream, there is another level of sharding for higher scalability. Each tuple carries a *DB shard ID*, which is also calculated based on the join key. The DB shard ID is then used by the operator to further split its in-memory state into different RocksDB instances. This enables more efficient and scalable persistence and recovery of state. The operator also supports multi-threading for both left and right engines. Each worker thread handles tuples containing a subset of join keys. In this way, contention is avoided during lookup because worker in the left engine only needs to communicate with a specific worker in the right engine. This design also effectively partitions the join window into multiple smaller, disjoint sub-windows which can be efficiently handled by multi-core machines.

Puma's scaler is responsible for managing the resources allocated to the operators of an application throughout its lifetime. The scaler continuously monitors application metrics and takes actions as a response. For streaming joins, it must scale both *slicers* and *joins*. Since slicers are stateless, the scaler adjusts the number of parallel tasks, targeting the minimum number required to keep pace with the input data stream. In our scenarios, the join operator is mostly memory-intensive, and the scaler must be able to adapt their memory reservation during runtime. If the memory reservation is larger than necessary, then we are underutilizing our hosts. If the memory reservation is too small, the host can run out of resources, causing an out-of-memory crash, killing all streaming jobs running on the same host. The scaler periodically measures the memory the job is using, and scales the memory reservation accordingly.

6.3 Memory Optimization

The join operator is mostly memory-bound as the right engine needs to buffer a whole window of data to ensure a high match rate. We leverage compression to reduce in memory data footprint. The compression is based on Zstd [7] dictionary compression. The dictionary adapts during runtime so that it can effectively compress new data.

In our use cases, dictionary compression achieved compression ratios from 3 to 10 for different use cases and reduced the overall memory consumption in our fleet to one third. This approach generally has better performance than standard Zstd because it adapts to stream changes quickly. Although using compression incurs more CPU consumption, memory is typically the bottleneck for the join operator, so CPU resources are abundant.

6.4 Monitoring and Alerting

Puma automatically sets alarms for deployed applications. For streaming joins, Puma monitors and alerts when a backlog of tuples accumulates beyond a certain threshold (e.g., more than 30 minutes of data to process). For the join operator, it can happen that one stream is always backlogged when compared to the other stream because of stream synchronization. In that case, our alarms fire only when both streams are backlogged, as that is a certain sign that the operator is not being able to keep up with the incoming traffic. Our system also automatically exports application-level metrics, such as join success rate and operator throughput. Changes on these metrics are often triggered by changes upstream (e.g., application logging). Users can set their own alarms for those metrics for their own debugging purposes.

6.5 Join Window Configuration

Oftentimes users are unsure of how to set the join window boundary for their applications. Intuitively, the larger the join window is, the higher the join success rate is. However, there are other tradeoffs to consider. First, a larger window leads to higher memory consumption, as the operator must store all of the right stream's tuples over the entire window. Second, the recovery time after a job restart is also longer, as more data needs to be replayed to rebuild a larger in-memory state. Third, the output latency can increase with a larger window upper boundary, as, depending on the join configuration, tuples may be emitted only after a full window is available in-memory.

In practice, users find it hard to figure out a good setting for the join window, unless they have an equivalent batch pipeline to experiment with. To help users find the ideal configuration, our service measures how close the timestamps of the matched events are to the join window's lower and upper boundaries. It then exports the 99th percentile of these deltas. A small value for the percentile indicates that the matches are very close to the window boundaries, so the window can be extended to further increase the matching rate. If the value is large, then users can safely reduce the window size, leading to memory savings and reduced latency.

6.6 Data Skew

Some streams have a skew regarding the distribution of events associated to a given join equality key. Given we use hashing to partition data, skew can lead to partition imbalance. This can lead to two undesired behaviors. First, disproportionately higher memory consumption in one task causes more memory to be allocated for all parallel tasks, as the resource reservation is the same for all parallel tasks. This causes memory to be wasted, as the reservation is made but the extra memory is not really consumed by most tasks. Second, if the skew is on the left stream, then it means that the operator must checkpoint much more data to local disk, which will cause higher utilization of the I/O subsystem. To address both issues, we cap the number of tuples the join operator holds for a given key. If the cap is reached for a tuple on the left side, the operator processes the oldest tuple with the same key and emits matches immediately. For the right side, the operator just evicts the oldest tuple. This causes a decrease in matching rate, but it has shown to have negligible effect for our use cases.

6.7 Timestamp Validity

As our streaming join operator is based on tuple *event time*, the stream synchronization scheme relies on the incoming timestamps to estimate a PT value. If the timestamp is too far in the past or in the future, and it affects the PT estimation, then it can result in the operator taking a long time to find an ascending sequence of PTs. To ensure that the operator progresses smoothly in such situations we do the following. For timestamps too far back in the past, it is very likely that is already outside of the PT estimate and the window size. As a result, if those are in the left stream, they are emitted with a null match if it uses a left outer join. Tuples in the right stream are discarded. If the timestamp is too far in the future (e.g., 7 days from the current wall clock time), then we discard the tuple from the PT estimation. For the left stream, we emit the tuple immediately as a non-match. For the right stream, we discard the tuple.

7. EXPERIMENTAL EVALUATION

In this section, we describe several experiments we have conducted to evaluate the effectiveness of our techniques with respect to accuracy and performance. All our experiments were conducted on a reserved set of production hosts using a subset of existing streaming applications.

7.1 Join Accuracy

The first question we want to answer is *how accurate is our streaming join operator – with a limited join window size – compared to a batch join using daily partitions?* When the batch join operates on a window of 24 hours of data, it would seem to have a very good chance of matching tuples, as a tuple can be matched even if its matching tuple is ingested many hours later. The exception is for tuples ingested close to the 24-hour cut off, given matching tuples may end up on the other side of the cut off, in the next daily partition. The latter is a gap where a streaming solution can help, as tuples are processed as they come. When the accuracy of the streaming join is close to its batch version, we can effectively replace it and gain the benefits of a real-time solution.

To do the evaluation, we selected one existing query using streaming joins. This query aims at understanding how well our search algorithms are doing by joining client and server search sessions. When a client sends a request, the server sends a response with a unique identifier. From then on, the client uses the identifier in all subsequent search-related requests. On the PQL query, we then stitch both client-side and server-side information together. Prior to Puma, the query was running in Hive using daily partitions. With the streaming version of the query, search system developers can understand in real-time if a new deployment affects search quality and take an action as a result.

The Puma version of the query is specified with the left stream matching any tuple in the right stream with timestamps within the [-3 hours, 30 min] interval. We then have three batch versions of the query: (i) join of 24 hours of the left stream with 24 hours of the right stream, (ii) join of 24 hours of the left stream with 48 hours of the right stream, and (iii) join of 24 hours of the left stream with 24 hours of the right stream, but where the join considers only the matching interval specified for the streaming version. The join with 2 days of data shows how many more tuples we can join when ignoring the cutoff of the daily partition. We use a Hive query to join both tables. The matching rate for both cases is computed as the number of tuples on the output that led to a match over the total number of tuples on the left-side table/stream.

Figure 8 shows the comparison of the Hive queries and the real-time version over a one week period. The average matching rate for 7 days for a 1-day to 1-day join is 93.23% (batch-1d-to-1d). When joining with 2 days of data, the matching rate increases to 94.07% (batch-1d-to-2d), indicating that eliminating the 1-day partition cutoff gives an additional 0.83% matching rate. When applying the 3.5 hours constraint, the batching matching rate drops to an average of 92.49% (batch-1d-to-3.5h), which is less than 1% when compared to joining the data with a full day. This indicates that the matching window of 3.5 hours still gives this application good accuracy when compared to 1-day of data. The streaming version achieves a matching rate of 92.44% on average (realtime-ws-3.5h), which is very close to the batch join version with the interval constraint. Although the operator loses some matches for bounding the window based on the PT estimation for tuples joining during the day, it compensates it by providing smooth joins over the partition cut off time. For the search application, it is acceptable for the streaming version to have a lower matching rate than the 1-day batch version, as the latency benefits far outweighs the marginal loss in accuracy.

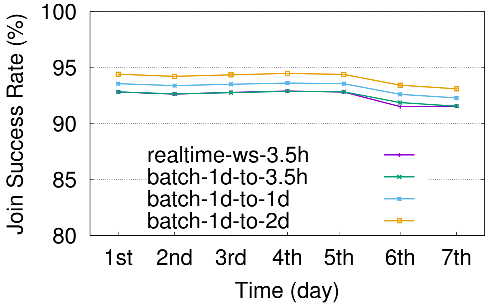


Figure 8: Streaming join accuracy is very close to the batch join accuracy.

7.2 Join Window Size vs. Join Success Rate

The second question we want to answer is *what is the gain in join accuracy when varying the join window interval?*. Intuitively, the larger the join window is, the higher the join success rate is. However, the cost of running the streaming application increases, as the join operator stores the whole window in memory. Given that, we need to evaluate how much one is actually gaining in accuracy to understand if the increase in computing cost is justifiable.

We evaluated the effect of increasing the window size on the join success rate (i.e., matched tuples / total tuples processed). We used the same production application as the one described in Section 7.1, and varied the window size from 1 hour to 6 hours. We limited the memory footprint evaluation to a single process of a parallel join.

Figures 9 and 10 show the results of running the join operator on production data for 3 days. Figure 9 shows that the memory consumption increases with the join window size, varying from about 10GB to more than 50GB at peak traffic. When looking at the accuracy (Figure 10), the improvement in success rate is not large when comparing a 1-hour window (ws-1h) to a 6 hours window (ws-6h). There is an average improvement of 1.24% on success rate observed between the 1-hour window and the 6-hour window. The small difference is due to the long tail of tuples that get processed after the PT estimation has moved forward and their matching windows have already been evicted. They would only successfully match if the operator would keep a significant larger window, similar to the Hive daily partition. The impact of window size on success rate is application-dependent and the level of loss in accuracy depends on the application at hand. When such loss in accuracy is acceptable, a streaming query becomes a computationally affordable solution with significantly lower latency when compared to a batch join.

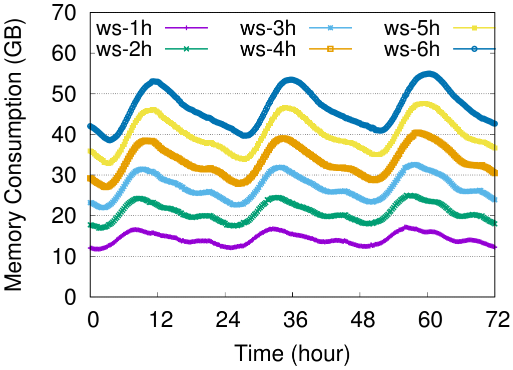


Figure 9: Memory consumption is proportional to the window size.

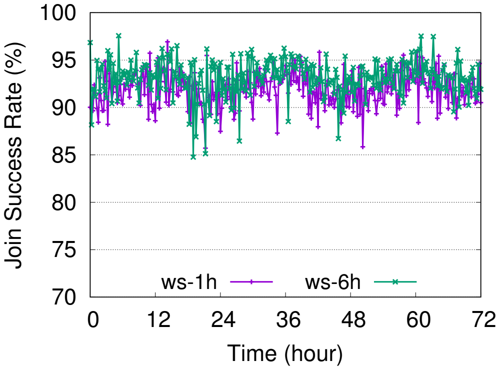


Figure 10: Join success rate with 1-hour window and 6-hour window; the results with other window sizes are in between and omitted for clarity.

7.3 Failure Recovery

With this experiment, we answer the following question: *is the approach of leveraging Scribe's persistency to replay data after a process restart feasible without causing an irrecoverable backlog?* As described in Section 6.1, we avoid persisting in-memory state of the right-side window to storage during checkpoints and fully replay hours of data from Scribe to rebuild the join window. This design helps us reduce the stress on the hosts' I/O, but also leads to increased recovery time. The longer the operator stays in recovery mode, the longer it can build a backlog of tuples and the further it will increase latency on application output.

On replay, the main cost is re-reading data from Scribe and rebuilding the key-value map in memory. Recovering the left stream state from RocksDB or HDFS is generally very fast, as the amount of data is much less compared to the right stream. With host-level sticky scheduling, the state can be recovered in just a few seconds.

To do this test, we used the same application as in Section 7.1 and with a 3-hour join window. This setup has a memory footprint of about 20GB. We then restarted the process on the same host to see how fast the join operator recovers from a failure (minute 28). Figure 11 shows that the operator can rebuild its state in about 9 minutes. Right after the operator finishes rebuilding its state (minutes 31-32), there is a peak in memory consumption, which is caused by the operator processing through its backlog (minutes 33-37). After that, the operator is fully recovered, and its memory footprint is back to pre-failure levels.

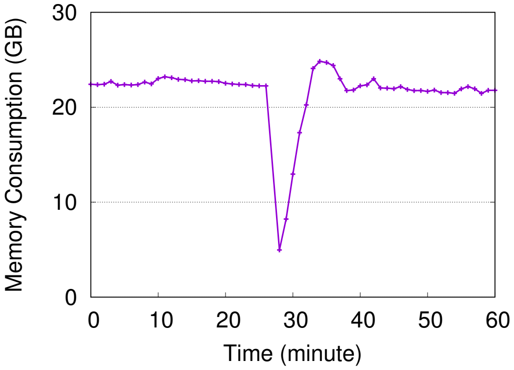


Figure 11: The join operator rebuilds 3 hours of data in about 9 minutes after a failure.

The recovery time depends on several factors. The main ones are (i) the amount of data, which depends on the right stream's throughput and the window size, (ii) the speed data can be read from Scribe, and (iii) the available CPU cycles on the host. The latter is important, as building the key-value map and compressing data are both CPU intensive. As mentioned before, the CPU is not the biggest constraint for our scenario, as we do equality joins and the number of processes co-located in the same host is limited. The host only executes processes running join operators, which are mostly memory intensive. This leads to a limited number of processes competing for CPU. This enables our operators to go through any accumulated backlog fairly fast.

7.4 Processing Time Estimation

This experiment answers the following question: *can the join operator dynamically adapt to different event time distributions in the incoming streams while preserving accuracy?* To effectively provide streaming joins as a service, the operator should be able to adapt to the different event time distributions present in different streams. This is because application developers may write a PQL query that consumes data from any Scribe stream. Furthermore, the event time distributions may vary throughout the day within the same stream. The operator should also handle such variations, otherwise we risk reducing the accuracy.

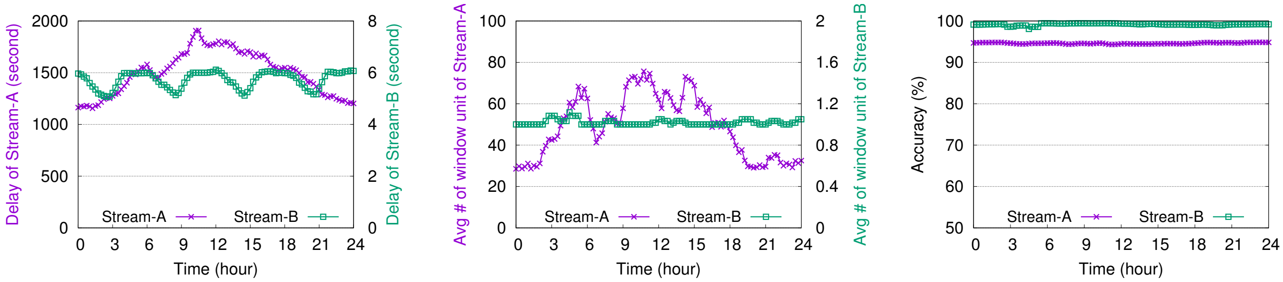
As described in Section 5.3, our PT calculation algorithm aims at achieving a stable accuracy by dynamically adjusting the window sizes according to the distribution of the observed event times. In this experiment, we validate our algorithm by observing the accuracy that the join operator achieves for 2 different streams that have different event time distributions over a period of one day starting at 12PM (Hour 0). *Stream A* contains tuples published by clients and its event times are very disordered. *Stream B* contains tuples published by servers and its event times are relatively well ordered. In this setup, we use a micro-batch size of 2 seconds. The operator is also configured to look for an ascending PT sequence of length 5. The statistic we use to estimate the PT is the 5th percentile.

Figure 12(a) shows the average stream delay throughout the day. The delay is calculated as the difference between the current wall clock time and the estimated PT. As the graph shows, the calculated PT is not constant when compared to the wall clock time. The variation depends on how disordered the stream is over time. The more disordered, the more tuples are needed to find an ascending sequence, which causes the PT to progress slower and, therefore, increasing the delay. Throughout the day, Stream A's PT delay varies from 16 minutes up to 31 minutes. Note that Stream A's stream delay has a bump at night time, as logs from client side have a larger difference from wall clock time at night, causing a more disperse event time distribution. For Stream B, the PT varies little (5-6 seconds). Although Stream B is not ordered, the event times are growing steadily as the delay between when an event is created and when its processed by the streaming application is very small.

Figure 12(b) shows the number of windows used to achieve an increasing PT sequence throughout the day. As Stream A is more disordered, it needs a significantly bigger window (up to 73) than Stream B, which is always 1. As the graph shows, the size of the window gets automatically adjusted throughout the day as the event time distributions varies. As Figure 12(c) shows, this dynamic adaptation ensures that we get good accuracy on the estimated PT. The accuracy is computed as the percentage of tuples in a batch that have their event time greater than or equal to the estimated PT. If there is a large number of tuples with their event times lower than the estimated PT, it means that the PT is moving too fast and the join matching rate may suffer.

8. RELATED WORK

Das et al. [16] is one of the first works to describe streaming joins with resource constraints in mind. The authors apply a semantic load shedding technique so that tuples are not dropped at random, which can cause a lower join matching rate. Kang et al. [24] proposes a framework to understand what join algorithm is best for processing each input stream depending on resource constraints. Gedik et al. [19] proposes selective tuple processing to shed CPU load. Our



(a) Average tuple delay

(b) Average number of windows

(c) Average PT accuracy

Figure 12: Stream A contains client-side events and is highly disordered, while Stream B contains server-side events and is fairly ordered. Our PT estimation algorithm dynamically adapts the size of the window of observation to capture the different characteristics and dynamics of real world data streams.

use cases are memory constrained. We deal with that by leveraging a persistent message bus to pace the consumption of tuples from the right and left stream according to the estimated stream time. If more memory is needed, our auto-scaler makes sure that the operator gets restarted with a higher memory reservation (up to a maximum bound).

PWJoin [18] explores punctuations to trim join states. Punctuations propagate patterns and purge the tuples in the window that match them. Punctuations are specified in terms of properties of the join attribute. Our stream time estimation scheme and its use to emit tuples can be considered a punctuation scheme where punctuations are generated by the operator itself. A difference is that it is established in terms of the timestamp attributes of a tuple. k-Mon [13] uses *k-constraints* to trim operator state. One such example is the *ordered-arrival* constraint, which establishes that out-of-order tuples are within k tuples of each other. This does not fit well in our scenario, as k can be different for different applications and even within the same stream.

Photon [11] uses a consistent ID registry to ensure that every tuple gets processed exactly once. Photon supports longer aggregation window by storing all events on persistent storage. Our solution is optimized for analytical applications that require shorter windows (e.g., several hours) by holding the events in memory, avoiding cross datacenter traffic on lookup. InfoSphere Streams [23] uses distributed snapshots to ensure exactly once semantics for stream processing graphs. Flink [14] implements fault-tolerance with a global and asynchronous snapshot of application state. AthenaX [1] is a stream processing service built upon Flink, and is similar to Puma's service model. Spark supports streaming joins [4] and guarantees no data loss when reliable receivers are used. It adopts write ahead logs and periodically checkpoints the application state to a reliable storage [12]; however, its bulk-synchronous parallel (BSP) computation model can introduce a non-trivial overhead when scheduling micro batches, which requires more advanced management techniques to achieve low latency [33]. Our scenarios have a less strict fault-tolerance requirement, which enables us to leverage a scheme with lower overhead.

Samza [29] has similarities with our system regarding fault-tolerance, as they use a persistent message bus and use local and remote storage to persist operator state. Our streaming join operator can be implemented on Samza.

Prior research uses specialized hardware to improve the throughput of streaming join operators. HELLS-Join [25] splits the join execution between CPU and GPU. Gedik et al. [20] investigates how to split sequential and parallel segments of joins onto a Cell processor. The Handshake join [31] finds join matches by processing each of the streams in different directions. This scheme shows advantages on processors with a high number of cores. These works can be helpful for processing large windows. Our case focuses on equality joins, where we can use hash tables as indexing data structures. Even though our windows can be large, the size of the window per key is generally small.

Similar to our work, Lin et al. [27] focuses on distributed stream joins that can scale and be memory efficient. Their window-based stream joins leverage timestamps assigned when the tuple is first ingested in the system. Window-Oblivious join [35] does not force users to specify a window size for the join. Unlike our work, it assumes streams are ordered or partially ordered. Our streaming join in PQL is is similar to Apache Calcite's SQL specification [3]. Calcite does not specify how to implement operators and how to generate backward compatible query plans.

An alternative approach to reduce join query latency is to incrementally maintain materialized views directly in the Data Warehouse [26]. View materialization is not currently supported by our SQL engines [5, 32].

9. CONCLUSION

This paper describes how we enable users to deploy new real-time queries with event time based streaming joins. The main challenges we had to tackle were (i) how to devise a robust stream synchronization scheme that can achieve a high matching rate while still controlling the memory consumption of the operator, and (ii) how to support application updates, so that query modifications do not lead to significant output delay, data duplication, and data loss. Our system has enabled several teams to reduce the latency of their analytics and monitor key service metrics in real-time. In the future, we plan to look into a DAG level stream time estimation, expand the types of supported joins (right outer join) and enable functors to be applied to matching tuples to decide which tuple should be emitted on a 1-to-1 matching scenario (e.g., tuple with most recent event time).

Acknowledgments

We thank the engineers who contributed to Puma and Stylus – Tim Williamson, Abhishek Maloo, Yuhan Hao, Alex Levchuck, Rajesh Nishtala, Ajoy Frank, Daniel Marinescu, and Adam Radziwonczyk-Syta. We thank David DeWitt for his detailed feedback on our manuscript, and Reynold Xin for suggestions on the final version of this paper.

10. REFERENCES

[1] Introducing AthenaX, Uber Engineerings Open Source Streaming Analytics Platform. `https://eng.uber.com/athenax/`, 2017.

[2] Amazon Kinesis Data Analytics. `https://docs.aws.amazon.com/kinesisanalytics/latest/dev`, 2018.

[3] Apache Calcite. `https://calcite.apache.org/`, 2018.

[4] Introducing Stream-Stream Joins in Apache Spark 2.3. `https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html`, 2018.

[5] Presto. `https://prestodb.io/`, 2018.

[6] RocksDB. `https://github.com/facebook/rocksdb/`, 2018.

[7] Zstd. `http://facebook.github.io/zstd/`, 2018.

[8] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at Facebook. *PVLDB*, 6(11):1057–1067, 2013.

[9] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[11] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.

[12] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*, 2018.

[13] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, Sept. 2004.

[14] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[15] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at Facebook. SIGMOD, 2016.

[16] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. SIGMOD, 2003.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[18] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. CIKM, 2004.

[19] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. CIKM, 2005.

[20] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the Cell processor. VLDB, 2007.

[21] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[22] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7, 2013.

[23] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. Consistent regions: Guaranteed tuple processing in IBM Streams. *PVLDB*, 9(13):1341–1352, 2016.

[24] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[25] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The HELLS-join: A heterogeneous stream join for extremely large windows. DaMoN, 2013.

[26] P. A. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *ICDE*, pages 56–65, April 2007.

[27] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. SIGMOD, 2015.

[28] A. Narayanan. Tupperware: Containerized deployment at Facebook. In *DockerCon*, 2014.

[29] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *PVLDB*, 10(12):1634–1645, 2017.

[30] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at Facebook. In *SOSP*, 2015.

[31] J. Teubner and R. Mueller. How soccer players would do stream joins. SIGMOD, 2011.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[33] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[34] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, 1991.

[35] J. Wu, K. L. Tan, and Y. Zhou. Window-oblivious join: A data-driven memory management scheme for stream join. In *SSDBM*, 2007.