SamzaSQL: Scalable Fast Data Management with Streaming SQL

Milinda Pathirage*, Julian Hyde‡, Yi Pan† and Beth Plale*
*School of Informatics and Computing, Indiana University
{mpathira, plale}@indiana.edu
‡Hortonworks
julian@hydromatic.net
†LinkedIn
yipan@linkedin.com

Abstract—**As the data-driven economy evolves, enterprises have come to realize a competitive advantage in being able to act on high volume, high velocity streams of data. Technologies such as distributed message queues and streaming processing platforms that can scale to thousands of data stream partitions on commodity hardware are a response. However, the programming API provided by these systems is often low-level, requiring substantial custom code that adds to the programmer learning curve and maintenance overhead. Additionally, these systems often lack SQL querying capabilities that have proven popular on Big Data systems like *Hive*, *Impala* or *Presto*. We define a minimal set of extensions to standard SQL for data stream querying and manipulation. These extensions are prototyped in SamzaSQL, a new tool for streaming SQL that compiles streaming SQL into physical plans that are executed on *Samza*, an open-source distributed stream processing framework. We compare the performance of streaming SQL queries against native Samza applications and discuss usability improvements. SamzaSQL is a part of the open source Apache Samza project and will be available for general use.**

1. Introduction

In today's data-driven economy, enterprises large and small are turning to data analytics to discern new knowledge from the growing number of data sources within and external to an organization. This growth is driven in part by unprecedented growth in number of mobile and Internet of Things (IoT) devices. In response, Big Data related technologies such as Hadoop have seen substantial uptake in both the enterprise and academic setting, enabling analysis of petabytes of data on commodity clusters.

While earlier work on processing real-time streams of data [1], [2], [3], [4] focus on high volume data that arrives continuously (as stream of indefinite duration), the data was often of one kind, for instance, finance data [3] or sensor data from cooperative-task robots [5] and the solutions were often centralized. With the arrival of social media, hand-held devices, smart sensors, health devices and IoT, the sources of information which can be processed in real-time have became significantly multiplied and varied.

In the more heterogeneous Big Data setting where data are analyzed from multiple real-time and non-real time sources, an architectural challenges arises in how to simultaneously handle the real-time streaming data analysis and the persisted table-type data (also called "batch processing") during analysis. Lambda Architecture (LA) platforms are a solution which brings together separate systems for each with solutions like Hadoop and Spark serving as the analysis platforms for the persisted data, and Storm, Samza and Kafka serving streaming data analysis. Summingbird [6] and Radstack [7] are two LA solutions. But LA solutions are flawed for a couple reasons. As per [8], *"The problem with the Lambda Architecture is that maintaining code that needs to produce the same result in two complex distributed systems is exactly as painful as it seems like it would be."* Too, our experience with *Summingbird*, *Apache Kafka* and *Samza* reveals that a deep understanding of complex programming abstractions is needed in order to use the APIs of stream processing frameworks effectively.

Opponents of LA argue [8] that one can achieve the same results as a LA architecture with only a stream processing system by retaining the input data unchanged (as proposed in LA) and reprocessing the data through increased parallelism and replay of historical data at a speed as fast as possible. The main advantage of this unified model of data processing is that it does not require management of two entirely different systems.

Fernandez et al. [9] discusses an improved version of this new architecture. In a proprietary system built at LinkedIn, a single-system solution is built on top of Apache Samza and Apache Kafka. At the time of writing, LinkedIn uses 90 Kafka clusters deployed across 1500 nodes to handle 150 TB of input data (including replication) distributed across 60,000 topics with 800,000 partitions daily. Further, LinkedIn is using 16 Samza clusers deployed across 700 nodes to host stream processing jobs written in either Java or Scala or both for the processing. These applications generated 630TB of output data including replication. This improved and integrated architecture is referred to as a Kappa Architecture[1], where everything is a stream. Further, it has been shown through wide adoption of frameworks

1. http://kappa-architecture.com

like *Hive* and *Drill* that declarative query language access to large-scale data works; the wide adoption is a strong indicator of the ease of use of SQL.

The authors posit a streaming SQL-based solution as a viable alternative to implementing Kappa Architecture style applications and filter, project, aggregate and join construct in streaming SQL can be used to implement the wide variety of queries found in Kappa Architecture style applications. Our motivating research question is thus: *Can the same low barrier and the clear semantics of SQL be extended to queries that execute simultaneously over data streams (in movement) and tables (at rest) for Kappa architecture applications that operate in a world of highly distributed, high volume, high heterogeneity data and fast analytics environment that characterizes today's Big Data processing? Can this be done with minimal and well-founded extensions to SQL?*

The general motivation of this work is to explore a unified framework which enables Kappa architecture style data processing pipelines based on well known standard SQL. The proposed system, SamzaSQL, supports nearline analytics tasks with latency requirements in the order of seconds and replay based high latency tasks with latencies in the order of minutes to hours.

In this paper we introduce SamzaSQL, its architecture and the novel design ideas behind it. SamzaSQL is a scalable and fault-tolerant SQL based streaming query engine implemented on top of Apache Samza with support for interaction with non-streaming data sources. SamzaSQL defines a minimal set of extensions to standard SQL to enable fast data querying and data manipulation. Queries expressed in streaming SQL are compiled into optimized procedural plans that are scalable, fault-tolerant and executed on *Samza*. SamzaSQL front-end, which does query parsing, validation and planning is built on top of *Apache Calcite* [10]. SamzaSQL is architected to support various data formats such as Avro or JSON and traditional relational and NoSQL stores using pluggable extensions. Windowing is a core and central concept of stream querying [11] because operators like *join*, min, max etc. cannot be resolved over unbounded streams. SamzaSQL provides a window operator implementation that provides timely and deterministic window output under unpredictable message delivery latencies, and deterministic window output with node failures and message re-delivery.

This paper makes the following contributions:

- SamzaSQL architecture for streaming SQL on top of a modern distributed stream processing framework
- introduction to language extensions to SQL to support streaming queries
- Baseline performance evaluation of throughput of stream extensions

The remainder of this paper is organized as follows: Section 2 gives overview of Apache Samza, Apache Calcite, and SamzaSQL in relation to these two component systems. Section 3 discusses the SamzaSQL data model and query language. Section 4 describes the SamzaSQL implementation. In Section 5, evaluate the performance and scalability of SamzaSQL. Section 6 discusses related work, and Section 7 concludes with a discussion of future work.

2. SamzaSQL Overview

Apache Samza and Apache Calcite provide core functionality in the proposed architecture so we provide a high-level view of them prior to discussing their role in SamzaSQL.

Apache Samza. Apache Samza is a distributed stream processing framework that utilizes YARN [12] for resource management, scheduling, and fault tolerance, and Apache Kafka for messaging. Samza's unit of deployment is a YARN job that can contain hundreds of streaming tasks, all executing the same processing logic on messages from one or more stream partitions belonging to one or more streams. Samza does not natively support directed acyclic graph (DAG) stream topologies as do Apache Storm and Apache S4. Samza architecture instead encourages formation of DAGs through connecting multiple Samza jobs via intermediate Kafka streams. With this approach, a failure of a downstream job will not affect an upstream job that is producing data; it further facilitates sharing across stream processing stages by allowing addition of jobs that consume an intermediate stream.

Samza provides developers with a Java API similar to Map/Reduce for implementing streaming tasks, and a message serialization and deserialization API called *Serde* API to support different message formats (textite.g., Avro, JSON or Thrift). While regular Samza jobs read and write to/from Kafka, Samza provides a separate Java API to plug in different input and output systems.

A high-level view of Samza architecture is shown in Figure 1. The features of Samza discussed here are available for SamzaSQL to utilize for executing streaming SQL queries. Samza comes with a built-in YARN client for submitting Samza jobs to a YARN cluster, and each Samza job has an application master to perform input partition assignments and task scheduling. The Samza application master also takes care of fault tolerance. A SamzaSQL query is a Samza job with SamzaSQL specific stream task implementation that performs the computation described in the query.

Samza scales its streaming tasks on Kafka's ordered, partitioned and replayable streams, where each task is assigned one or more partitions from a stream. Samza stands out from other distributed stream processing systems on the following set of features:

Fault-tolerant Local State: Each streaming task in a Samza job has managed local storage that can be used to implement stateful stream processing logic. The state is modeled as a stream and Samza manages the snapshotting and restoration by replaying the state stream in case of a task failure. This local state is built to handle many gigabytes of data per partition.

Durability: Samza utilizes Kafka's properties to guarantee messages are delivered in the order they were written to Kafka and also ensures no message is lost. In the case of a task failure, Samza automatically migrates the task to some other node using YARN resource management capabilities

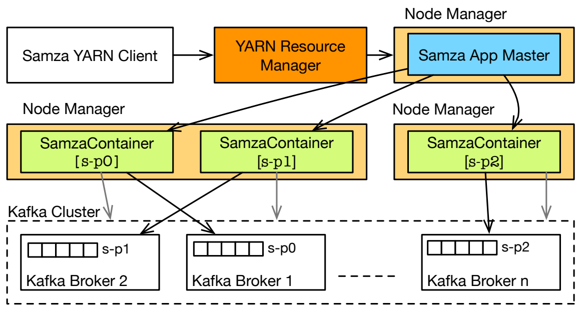


Figure 1: Samza Architecture (*Input stream is divided to 3 partitions and [s-p***x***] indicates the partition mapping. Depending on the streaming application and Samza job's configuration, Samza containers write output streams, checkpoint streams to Kafka.*)

and ensures streams will be replayed from the last known checkpointed partition offset.

Masterless Design: Samza has no master. Instead each job has a master (a YARN application master [12] to be specific) which makes scheduling and resource management decisions on behalf of its job. This decoupling makes sure glitches in one job do not affect other jobs in the system.

Bootstrap Streams: Samza supports a special type of stream called the *boostrap stream* which can be used to implement *relation-to-relation* or *stream-to-relation* joins. If an input stream to a Samza task is of type bootstrap, Samza tasks waits until all the messages from the bootstrap stream's partitions are processed before it begins processing messages from other input streams. We discuss how we utilize bootstrap stream to implement stream-to-relation join in Section 4.

Apache Calcite. Apache Calcite is a collection of software libraries and tools that can be used to parse, validate, optimize and execute standard SQL queries. Calcite is fully extensible and allows users to plug custom optimizer rules and implement custom logical and physical algebras extending in-built algebras for custom optimizations and domain specific query planning. Calcite is used by several modern big data solutions such as Apache Hive, Apache Kylin, Apache Drill and Apache Phoenix as the foundation of their query planning and optimization modules. Also, Calcite includes utility libraries that can aid the code generation phase of the query planner making it a preferred solution for the SamzaSQL stream query framework.

SamzaSQL. SamzaSQL builds on Samza and Calcite to provide an easy to use declarative programming approach based on standard SQL that compiles queries into Apache Samza jobs that run on YARN. SamzaSQL takes advantage of Samza features for fault-tolerant, scalable and isolated streaming applications. Users and developers interacting with SamzaSQL use streaming extensions to SQL that we developed. SamzaSQL still makes it possible to integrate user-defined code, different message serialization formats and messaging and storage frameworks other than Kafka. SamzaSQL natively supports Avro message format.

Samza's deployment unit consists of a job package and a property base configuration file. The configuration file specifies the streaming task implementation, input and output configurations, Serdes to use for deserializing and serializing messages, local storage configurations including the Serdes to use. This file should be created before submitting the streaming job to YARN. Due to this, SamzaSQL implements two-step query planning. In the first step, we generate the optimized physical plan and create the Samza job configuration from the physical plan at the SamzaSQL shell (Shaded gray in Figure 3) and the second step that takes care of operator and mmessage router generation 4.2 from the physical plan happens during Samza stream task initialization phase. Section 4 discuss this process in detail.

3. Data Model and Query Language

As part of SamzaSQL, we extended Calcite with stream extensions. The guiding principle of the query language selection was that the language and the semantics should be familiar to users. We want the language to support queries that combine streams and relations without semantic ambiguities. We chose standard SQL with a minimal set of extensions to support streaming queries. The SQL we support in SamzaSQL has a low barrier to entry because someone with knowledge about standard SQL can quickly learn streaming SQL. We aim to keep the semantics of streaming SQL as same as standard SQL by producing the same results on a stream as if the same data were in a table. The only deviation from standard SQL semantics happens when handling out-of-order arrivals where some tuples may get discarded due to the expiration of timeouts. There will be multiple outputs for the same window due to early results policy that send out partial results as soon as a window boundary condition is met without waiting for delayed arrivals.

A stream in SamzaSQL consists of ordered partitions. The query execution engine exists on top of this partition model. An ordered partitions model, derived from Kafka's topics and partitions model, guarantees ordering of tuples only within a single partition.

SamzaSQL expects a timestamp field in the incoming message to enable time-base window computations and to provide ordering guarantees. Users need to ensure that their queries preserve the timestamp field or send a timestamp derived from input tuples to the output. Lack of a timestamp field in a derived stream will prevent certain stream queries such as time-based window aggregations on derived streams.

3.1. Data Model

SamzaSQL's data model consists of three main components where the concept *Stream* is dervied from Samza's concept of stream as a collection of ordered partitions.

- **Stream** - A stream S is a possibly indefinite **partitioned** sequence of temporally-defined *elements* where an element is a tuple belonging to the schema of S. An *element* is uniquely identified by a *sequential ID* number (also known as *offset* in Kafka terms); the ID number is unique only within the context of a *partition*. *SamzaSQL requires the event timestamp to be included in the tuple. Streams are immutable at the element level and append only, so update and delete operations are not supported on streams.*

- **Partition** - A partition is a time-ordered, immutable sequence of elements existing within a single stream. A partition continually appends elements to a commit log. SamzaSQL guarantees the ordering of elements existing within the same partition, but not of elements existing across partitions. How a stream is partitioned is defined by the publisher at publishing time.
- **Relation** - Analogous to a relation/table in relational databases, a relation R is a bag of tuples belonging to the schema of R. A relation is not limited to a relation/table from a relational database, it can refer to a bag of tuples stored in an any kind of storage (CSV file, HBase table) given that all tuples belongs to the same schema. A relation in SamzaSQL can also be a stream where stream is a relation changelog. Relation stream mentioned in Section 3.3 is another example for an relation represented as a stream.

By default, each stream (except for some intermediate derived streams) in Samza has a corresponding topic in Kafka [13]. Often, a topic consists of multiple partitions. Messages in a Kafka topic can be in any format as long as it is wrapped in a Kafka binary format. But SamzaSQL requires all the messages in a topic to be in the same message format with the same schema to avoid unnecessary message processing failures. A topic in Kafka often retains historical data for several hours to several days depending on the requirements of the end user.

SamzaSQL supports primitive column types (integers, floating point numbers, generic strings, dates and booleans) and nestable collection types - array, map and object.

3.2. Query Language

SamzaSQL supports standard SQL with several streaming specific extensions [14] to enable queries on data streams. We aim to keep the semantics of the language clear such that we can produce same results on a stream as if the data is in a table. And the other goal of preserving SQL semantics is to make it possible to query both tables and streams in a single query.

SamzaSQL does not support data definition statements in this version and depends on both the Kafka schema registry[2] and Calcite's built-in JSON based schema descriptions to provide the query planner with the metadata necessary for query planning. SamzaSQL's streaming SQL supports all the operators and functions supported by Calcite at the language level. Basic filtering, projection, and Having queries are the same as standard SQL.

We use the following example of two tables and five streams to illustrate SamzaSQL's stream extensions to SQL. The attribute "rowtime" is the event/tuple creation time (at the event source) and "sourcetime" means packet creation time:

- `Orders (rowtime, productId, orderId, units)` - a stream
- `Products (productId, name, supplierId)` - a table
- `Suppliers (supplierId, name, location)` - a table
- `PacketsR1 (rowtime, sourcetime, packetId)` - a stream
- `PacketsR2 (rowtime, sourcetime, packetId)` - a stream
- `Asks (rowtime, askId, ticker, shares, price)` - a stream
- `Bids (rowtime, bidId, ticker, shares, price)` - a stream

2. https://github.com/confluentinc/schema-registry

3.3. STREAM Keyword

The `STREAM` keyword is the primary extension in the streaming SQL. When `STREAM` immediately follows `SELECT`, it tells the system that it should continue processing the incoming elements, not the immediate data in a database. Listing 1 outputs all columns and rows from the `Orders` stream. Like any streaming query, this query will continue to run, outputting a tuple whenever a tuple arrives in the `Orders` stream. In the absence of a `STREAM` keyword, SamzaSQL will consider the stream as a table consisting of the history of the stream up to the point of execution of the query, and work as a standard relational query. `STREAM` keyword in sub-queries or views has no effect. The query planner discards the `STREAM` keyword and figures out whether the relations reference can be converted to streams or not.

Listing 1: Select all the tuples from a stream

```
SELECT STREAM * FROM Orders
```

Applying a streaming query over one or more streams and relations combination will results an another stream and the resulting stream can be a insert stream, or a delete stream, or a relation stream (same as CQL [15] semantics). For filter and project queries output will always be a insert stream and for queries involving window operation the result will depends on what user specifies in the query (User can ask for delete or relation stream instead of default insert stream). Presence of `STREAM` keyword in query means you are asking for insert stream. SamzaSQL doesn't support other two types of outputs yet and support for relation streams will be added in future. Relation stream generally contains tuples added to or removed from the relation as a result of streaming computation and relation stream output is compatible with SamzaSQL's notion of relations as streams(Section 4.4).

3.4. Filtering and Projections

The presence of the `STREAM` keyword in a query that has a filtering or projection operation, as with Listing 2, tells the system to continue projecting or filtering the incoming tuples/rows. Presence of a `HAVING` clause will cause filtering after aggregation as with standard SQL. When doing projections, we recommend the inclusion of a timestamp field in the output tuples because having a sorted timestamp enables advance queries as mentioned in the begining of this section.

Listing 2: Select *rotime, productId, and units* columns from Orders where *units* is greater than 25

```
SELECT STREAM rowtime, productId, units
FROM Orders
WHERE units > 25;
```

3.5. Sub-queries, Views and Closure Property

The SQL closure property allows us to perform any operation possible on a *table* on a query. This makes `HAVING` obsolete and also enables view concept.

Listing 3: Calculate hourly order count (c) and units sold (*su*) per product and filter out products with less than 2 orders or less that 10 units sold

```
CREATE VIEW HourlyOrderTotals (rowtime, productId, c, su) AS
  SELECT FLOOR(rowtime TO HOUR),
    productId,
    COUNT(*),
    SUM(units)
  FROM Orders
  GROUP BY FLOOR(rowtime TO HOUR), productId;

SELECT STREAM rowtime, productId
FROM HourlyOrderTotals
WHERE c > 2 OR su > 10;
```

Same query 3 above can also be expressed using a sub-query as below:

```
SELECT STREAM rowtime, productId
FROM (
  SELECT FLOOR(rowtime TO HOUR) AS rowtime,
    productId,
    COUNT(*) AS c,
    SUM(units) AS su
  FROM Orders
  GROUP BY FLOOR(rowtime TO HOUR), productId)
WHERE c > 2 OR su > 10;
```

Views and nested queries can be useful for expressing and managing long streaming query pipelines.

3.6. Hopping and Tumbling Windows

A Hopping window is different from a sliding window in that a hopping window emits a partial/full aggregate every H time units for tuples that have arrived over the last S time units. A tumbling window is the special case of Hopping window where H and S are equal.

The proposed query language includes the following functions used in `GROUP BY` clause to express hopping and tumbling windows.

- `HOP(t, emit, retain)`
- `HOP(t, emit, retain, align)`
- `HOP(t, emit)`

Where t is the timestamp field name, $emit$ is result emmitting interval, $retain$ is window size and $align$ can be used to adjust the first emit's time. Also it's not necessary for $retain$ to be a multiple of $emit$. In addition to above functions, aggregate functions `START` and `END` was introduced to capture start and end time of a window with in aggregates.

Listing 4: Hourly orders

```
SELECT STREAM START(rowtime),
  COUNT(*)
FROM Orders
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR)
```

Listing 5: Calculate total orders within 2 hour period beginning 30 minutes past each hour and emit the order count every 90 minutes

```
SELECT STREAM START(rowtime),
  COUNT(*)
FROM Orders
GROUP BY HOP(rowtime,
    INTERVAL '1:30' HOUR TO MINUTE,
    INTERVAL '2' HOUR, TIME '0:30')
```

3.7. Sliding Windows

Analytics functions from standard SQL can be utilizied to describe sliding window aggregations in SamzaSQL. Similar to `GROUP By`, an *analytic functions* can be used to compute aggregates based on group/set of rows but without collapsing rows. One row comes out for each row goes in. Grouping of rows is done based on a window expressed over the time domain or tuple domain.

Listing 6: Calculate total units sold (*unitsLastHour*) per product in past hour

```
SELECT STREAM rowtime,
  productId,
  units,
  SUM(units) OVER (ORDER BY rowtime PARTITION BY productId RANGE
        INTERVAL '1' HOUR PRECEDING) unitsLastHour
FROM Orders;
```

You can also have multiple functions in the select clause defined over multiple windows.

3.8. Joins

Joins are used to relate information from different streams and relation combinations. SamzaSQL's query language has first class support for stream-to-stream and stream-to-relation joins. Unlike existing streaming SQL variants such as CQL we do not support `OVER` clause for specifying sliding windows over streams. Instead, joining over a window is expressed as a part of the *join condition* as shown below.

3.8.1. Stream-to-Stream Joins

List 7 illustrates stream-to-stream join scenario which calculates time taken by a packet to travel from one router (R1) to another (R2) by joining `PacketsR1` and `PacketsR2` streams over a 2 seconds sliding window. Sliding window join queries uses additional join condition on the tuple's timestamp (`rowtime`) to specify the window over the stream. SamzaSQL assumes that the tuple's timestamp monotonically increases.

Listing 7: Calculate time taken for a packet to travel from router R1 to router R2 (`GREATEST` returns the max timestamp)

```
  SELECT STREAM
    GREATEST(PacketsR1.rowtime, PacketsR2.rowtime) AS rowtime,
        PacketsR1.sourcetime,
    PacketsR1.packetId,
    PacketsR2.rowtime - PacketsR1.rowtime AS timeToTravel
  FROM PacketsR1
  JOIN PacketsR2 ON
    PacketsR1.rowtime BETWEEN
      PacketsR2.rowtime - INTERVAL '2' SECOND
      AND PacketsR2.rowtime + INTERVAL '2' SECOND
    AND  PacketsR1.packetId = PacketsR2.packetId
```

3.8.2. Stream-to-Relation Joins

Stream to relation joins are expressed similar to relation to relation joins (Listing 8) in standard SQL and presence of `STREAM` keyword tells SamzaSQL to keep processing joining records against the records in the relation.

Listing 8: Add supplier identifier (*supplierId*) to each row from Orders stream

```
SELECT STREAM
        Orders.rowtime,
  Orders.orderId,
  Orders.productId,
  Orders.units,
  Products.supplierId
FROM Orders
JOIN Products ON Orders.productId = Products.productId
```

4. Archtecture and Implementation

This section explains the SamzaSQL architecture and discusses important aspects of the operator layer implementation. We first describe how each component in our architecture diagram (Figure 2) fits together, next we discuss the SamzaSQL shell and the query planner and finally discuss implementation details of operator layer. We describe how we utilized various features of Samza discussed in Section 2 in our operator layer implementation.

4.1. SamzaSQL Architecture

Figure 2 illustrates the SamzaSQL architecture. Users interact with SamzaSQL through a special SQL shell build using SqlLine[3] library and a custom SamzaSQL specific JDBC driver implementation. SamzaSQL shell is a command line application that runs on users' desktop. SamzaSQL JDBC driver wraps the query planner that converts a streaming SQL query to a Samza job containing one or more SamzaSQL tasks. Query planner uses Samza YARN Client to submit streaming jobs to a YARN cluster. Query planner uses Calcite model files described in JSON format and Kafka Schema Registry to retrieve necessary metadata for query planning and uses Zookeeper to share metadata and configuration information between query planner and SamzaSQL streaming tasks. Information shared via Zookeeper includes streaming SQL query to use during task query planning, Schema Registry location, message schema details, etc. We are also planning to utilize Zookeeper to store SamzaSQL shell's session informaiton including running queries, and including input/output streams and the status of the query.

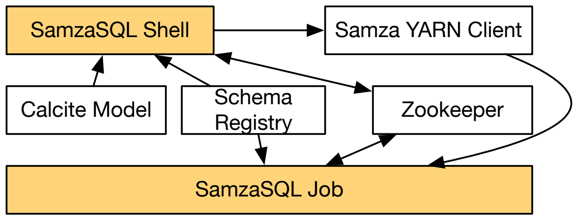


Figure 2: SamzaSQL core architectural components

4.2. Query Planner

Queries submited through SamzaSQL shell first go to the SamzaSQL JDBC driver and then to the query executor that invokes the query planner to generate the Samza job configuration. Query executor then invokes the Samza YARN client to submit the Samza job that execute the query to YARN cluster. As mentioned in Section 2 SamzaSQL uses Apache Calcite for query planning with custom planner rules and physical operator layer specific to SamzaSQL. Figure 3 shows the details of the query planning process.

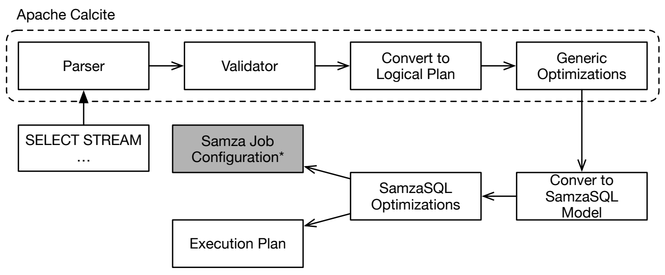


Figure 3: Query planning process

Query planning in SamzaSQL occurs in two places. It occurs at the SamzaSQL shell, where the physical plan is used to generate the Samza job configuration. It also occurs at the SamzaSQL task, where the physical plan is used to generate operator code and SamzaSQL message router. Message router is a DAG of streaming SQL opperators responsible for flowing messages through query operators. The physical plan is a tree of relational algebra operators such as scan, filter, project and join where scan operators are at the leaf nodes.

SamzaSQL uses Apache Calcite to parse, validate, convert the query to a logical plan and finally apply some generic optimizations bundled with Apache Calcite. The optimized plan from Calcite is then converted to a SamzaSQL physical plan. It is possible to apply custom optimization rules specific to SamzaSQL on top of this physical plan but the current implementation does not implement SamzaSQL specific optimizations.

During job configuration generation (highlighted in Figure 3) from physical plan, some of the metadata such as message schemas and the streaming query are stored in

3. https://github.com/julianhyde/sqlline

Zookeeper and references to those configuraitons are added to the job configuration. SamzaSQL tasks then read actual values for configurations from Zookeeper. At SamzaSQL shell query executor uses generated job configuraiton to submit a Samza job using Samza YARN client.

Physical plan generation happens again at the initialization phase of the SamzaSQL tasks and necessary metadata and configurations are loaded from Zookeeper and/or schema registry at this stage. In cases where message schemas need to be retrieved from schema registry, YARN cluster should have access to schema registry. After physical plan generation, operator code generation happens. We use Janino[4] and Calcite's Linq4j to generate Java code required for operators. We use code generation to generate filter conditions, projection expressions, window operators and join operators.

4.3. Sliding Window Operator

SamzaSQL implements window operations at two places. Hopping and tumbling windows are implemented in the streaming aggregate operator and sliding window operations have its own window operator. The currently implementation only supports time-based sliding window operations. At the time of writing streaming aggregate based window operators are partially implemented. In this subsection we explains how we have implemented sliding window operator to achieve fault-tolerance and to withstand message replays to provide exactly once guarantees.

Sliding window operator advances the window and emit aggregates upon receiving of each tuple, where hopping and tumbling windows are advanced and results are emitted in certain time intervals. Sliding window operator utilizes local state (Managed key/value storage local to streaming tasks provided by Samza) to store messages it received and to store window state. Window state consists of references to all the messages belonging to the current window and messages are purged from this state during window advancement upon receiving new tuples. We also maintain local storage backed data structures to keep the state of aggregated values and bounds of the sliding window.

Samza takes care of snapshotting the local state and restoration upon a task failure. Also takes care of checkpointing state related to consuming input stream (last read offset). So in case of a task failure, locate storage keeping the window state (aggregate state, window bounds, messages task instance has seen and references to messages in current window) will get restored with last known snapshot. After task re-scheduling and local state restoration, Samza will start replaying the messages from last read offset and window operator will continue to build the aggregate state from the restored window state. Pseudo code of the sliding window operator logic is shown in Algorithm 1.

Algorithm 1: Sliding window operator logic

```
input : tuple
if uninitialized window state then
    initialized window state;
get tuple timestamp;
update window bounds;
add a reference to the tuple into the window store;
if uninitialized partition information then
    initialize partitions;
purge messages and adjust aggregate values;
compute new aggregate values adding current tuple;
send latest aggregate values downstream;
save messages in message store;
```

4.4. Stream-to-Relation Join

SamzaSQL implements stream to relation joins through Samza's *bootstrap stream* capabilities. Assuming relation is available as a change log stream, SamzaSQL tasks that execute stream-to-relation joins are configured to read the change log stream as a bootstrap just after initializing the task. Samza task runner will not deliver other input streams to the task until partitions of bootstrap stream assigned to the task are fully read.

In the case of stream-to-relation joins, Join operator builds a cached copy of the partitions of the relation assigned to it in the local storage and join will be performed upon arrival of tuples from input stream(s). Assuming all the changes to the relation ended up in the change log stream, join operator will have latest state of the relation in its local storage. We assume that change log streams are partitioned in the same way as the other input streams so that data from relations and streams belonging to matching partitions will be ended up in the same streaming task. There are tools like Databus [16] and *bottledwater-pg*[5] PostgreSQL plugin which generates changelogs streams from relations hosted by PostgreSQL to aid in database change capture.

5. Evaluation

We experimentally evaluate performance and scalability of SamzaSQL on its throughput and scalability against Samza for filtering, projection, sliding window aggregations and stream-to-relation joins. SamzaSQL's throughput is lower than stream processing applications written in native Samza API for reasons we explain. We use a synthetic dataset for evaluation.

We also discuss the usability improvements of SamzaSQL over native Samza for several sample applications including sliding window aggregation and stream-to-relation joins. We refrain from evaluating expressiveness of the language and computability of language and runtime, and instead focus on the architecture and implementation of SamzaSQL. But we note that the Calcite query planner guarantees the computability of queries at the logical level.

5.1. Setup and Performance Evaluation

We use synthetic data generated for `Orders (rowtime, productId, orderId, units)` stream

4. http://unkrig.de/w/Janino

5. https://github.com/confluentinc/bottledwater-pg

and `Products (productId, name, supplierId)` relation from Section 3 for the performance evaluation queries. Kafka benchmark [6] shows that a 100 byte messages is good message size while providing acceptable Mbps throughput. Since the Kafka benchmark[2] shows that network saturation was reached for large messages (> 1KB) but messages per second value is too low (reduced by factor of 7) when compared with 100 byte messages, we choose 100 bytes messages for our benchmark by adding a random string to each record from Orders stream. Each Kafka topic used in our benchmark has 32 partitions and Samza's scheduler uniformly distribute partitions across tasks.

The following four queries are used to compare the throughput of SamzaSQL and native Samza API.

- **Filter** - `SELECT STREAM * FROM Orders WHERE units > 50`
- **Project** - `SELECT STREAM rowtime, productId, units FROM Orders`
- **Sliding Window** - `SELECT STREAM rowtime, productId, units, SUM(units) OVER (PARTITION BY productId ORDER BY rowtime RANGE INTERVAL '5' MINUTE PRECEDING) unitsLastFiveMinutes FROM Orders`
- **Stream-to-relation Join** - `SELECT STREAM Orders.rowtime, Orders.orderId, Orders,productId, Orders.units, Products.supplierId FROM Orders JOIN ON Orders.productId = Products.productId`

Samza v0.9.1, Kafka v0.8.2.1 and Yarn 2.4.0 are used. Experiments are carried out on Amazon EC2. YARN and Kafka clusters used for experiments consist of three **r3.2xlarge** instances each while YARN node manager and Zokeeper (3 nodes) used **r3.xlarge** instances. The **r3.xlarge** EC2 instances have 4 virtual CPU cores, 30.5 GG of memory and 80 GB SSD and **r3.2xlarge** EC2 instances have 8 virtual CPU cores, 61 GB of memory and 160 GB SSD. Due to EC2 I/O rate limits we had to carry out sliding window tests locally in a iMac with 4GHz Core i7 CPU, a SSD and 24GB of RAM. Sliding window implementation reads/writes from/to key-value store multiple times causing EC2 to throttle access to disk after a couple of minutes.

Figure 5a, Figure 5b and Figure 5c show throughputs of filter, project and join queries implemented in Samza Java API and SamzaSQL.The average throughput across containers was multiplied by the container count to get the job throughput. Results show sublinear scalability because the number of Kafka stream partitions assigned to a single task decrease with the increasing number of tasks (we keep partition count constant across tests) and lower number of partitions means lower read throughput at the streaming task.

Benchmark results show that SamzaSQL is underperforming by 30-40% compared to a streaming job written in Samza Java API for filter and project queries. CPU profiling of SamzaSQL generated streaming jobs showed that the performance overhead of the former is due primarily to message format transformations (*AvrotoArray* and *ArraytoAvro* steps in Figure 4) in SamzaSQL's generated streaming job to make it possible to use the existing code generation framework of Apache Calcite. The current prototype implementation of SamzaSQL implements SQL expressions on top of a tuple represented as an array in memory, and we convert incoming messages to an array at the scan operator and the array back to an Avro record in the stream insert operator (Figure 4). CPU profiling also showed that SamzaSQL's operator router layer also adds very little overhead when compared with message transformation overheads. The streaming job implementation in Samza Java API directly reads from incoming Avro message and writes back the message into the output stream without any modification for filter queries, and we create Avro messages directly from incoming Avro messages in project query.

6. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

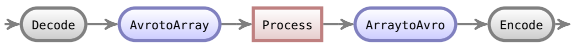


Figure 4: SamzaSQL message processing flow (***AvrotoArray*** *and* ***ArraytoAvro*** *steps are absent in native Samza processing flow.*)

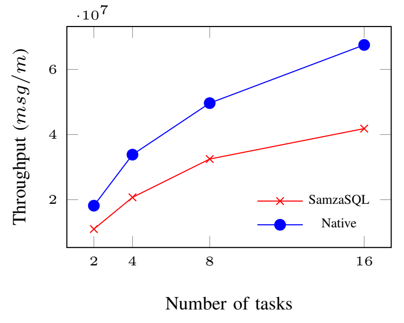
SamzaSQL's implementation of join is about 2 times slower than Samza mainly due to key-value store deserialization overhead and overheads of the operator router layer. Profiling data shows that Kryo based Java object deserialization used in SamzaSQL implementation is more than two times slower than Avro based deserialization used in Samza's Java API based implementation.

Figure 6 shows the throughput comparison for sliding window operators. Monitoring of access to key-value store (local storage) shows that throughput is dominated by access to the key-value store, and this makes the overhead of message transformations negligible. CPU profiling of sliding window jobs confirm that a considerable amount of time is spent on key-value store read/write operations in both SamzaSQL and native Samza implementations.

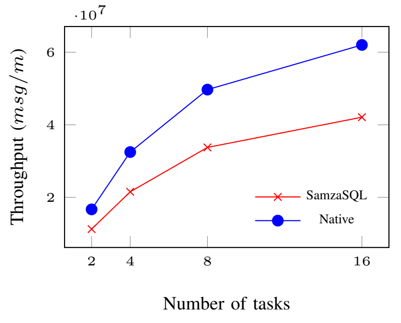
We also like to note that streaming SQL reduces development overheads by allowing users to express streaming queries declaratively using a couple of lines where as streaming jobs implemented using Samza's Java API will contain more than 100 lines for sliding window queries, more than 50 lines for simple stream-to-relation join and around 20 to 30 lines for filter and project queries. In addition to query logic and other dependencies, users needs to maintain stream job configuration for each query in case of Samza, where as SamzaSQL automatically generates query configuration.

6. Related Work

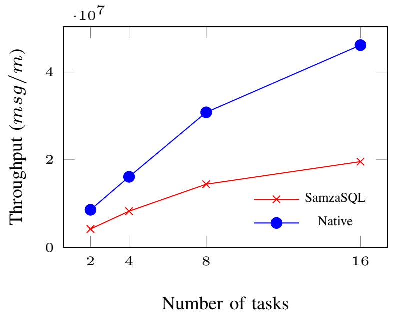
Early stream processing research worked out basic models and architectures for processing over indefinite streams including time- and rate-based windows, pattern matching as query operators, and query cost metrics. Notable stream processing engines include Aurora [17] and its distributed successor, Borealis [3] and STREAM [2] from the database community and dQUOB [1] and GATES [4] from high



(a) filter



(b) project



(c) stream-to-relation join

Figure 5: Throughput $(messages/minute)$ comparison of SamzaSQL and Samza

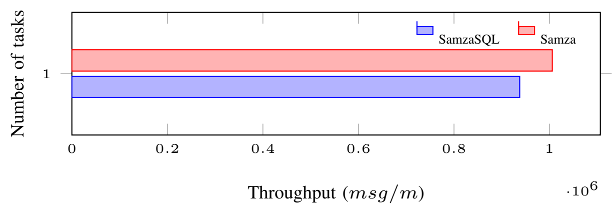


Figure 6: Sliding Window Operator Throughput comparison

performance and grid computing community.

S4 [18] and Storm [19] support a one-at-a-time processing model while Spark Streaming [20] uses a micro-batch processing model. Storm also supports micro-batching through its Trident API. Trill [21] differs in that it uses a configurable processing model where a user chooses one-at-a-time or micro-batching. The primary difference of Samza is in its stateful stream processing model where each task gets its own key/value store local to the task and Samza itself provides support for recovering local state in the case of failures. Spark Streaming and Samza both target second level latencies while Storm and S4 target millisecond latencies. On the other hand, Trill supports a wide range of latencies from real-time to batch. Apache Flink also provides both stream and batch processing capabilities.

One pass processing over data streams using a declarative language like SQL requires rethinking the language operators that can be supported, their interpretation in relational algebra, and query optimizations. TelegraphCQ [22], Tribecca [23], GSQL [24] all proposed extensions to SQL.

CQL [15] formalizes how a query can execute simultaneously over streaming data and data at rest with semantics that rest on three operations – *stream-to-relations*, *relation-to-relation* and *relation-to-stream* – and on two types of data – *streams* and *relations*. Window operations over a stream are conceived of as a *time varying relation*. Applying stream-to-relation operation on a stream results in a time varying relation. We use the time-varying relation to implement window operations. SamzaSQL support window operations by flowing the message through operators one at a time and window processing happens at the window or aggregate operators. Allowing general relations to co-exist with streams in a single query is one of the important aspects of CQL and our research to allow relations and streams in a single query are inspired by CQL. Our language does not support specifying window over a stream in the `FROM` clause. Following standard SQL, our language allows defining windows explicitly to use with SQL analytics functions (sliding window aggregations) as well as tumbling window aggregations using general data semantics as shown in Listing 4.

At the time of writing of this paper, Apache Storm and Apache Flink communities are starting to utilize Apache Calcite to implement a streaming SQL layer. Apache Storm has a prototype implementation that filters and projects messages while Apache Flink is still at the design phase. Those communities intend to utilize streaming extensions added to Apache Calcite as a part of this work.

7. Conclusion and Future Work

SamzaSQL is a novel streaming SQL implementation on top of Apache Samza which utilize standard SQL as much as possible by moving streaming related details away from the language layer to the physical execution layer. SamzaSQL demonstrates the advantages of having access to checkpointed local storage and concepts such as bootstrap streams in a stream processing framework. SamzaSQL is a part of the Apache Samza project and we are working continuously to improve the framework through new features and improved performance and user experience.

SamzaSQL is ongoing with multiple areas of future work. First, Samza achieves scalability through pre-partitioned streams. If a certain query such as join requires a different partitioning scheme (based on different set of message fields), SamzaSQL must to repartition the stream. Re-partitioning may change the original ordering of messages and this can effect order sensitive queries such as sliding window aggregates. A future work item is to explore re-partitioning to provide better guarantees on ordering between intermediate streams. Second, samzaSQL assumes the availability of a timestamp field in input messages and time-based window aggregates depends on this timestamp field. If this timestamp property is dropped during a projection, SamzaSQL loses the ability to perform time-based window aggregations on the resulting stream. The query planner should provide better warnings and error messages on such scenarios.

Third, the current implementation does not support *stream-to-relation* queries discussed in Arasu [15]. *Stream-to-relation* queries are used where outputs from window operators are rendered as relations and not as a continuous stream of updates to the window. We are working on integrating stream-to-relation queries into the query language and execution layer as a change stream to Kafka so tools outside SamzaSQL can consume this stream to generate the relation.

Fourth, the current implementation omits hopping and tumbling window aggregates as well as diverse set of aggregate functions. It further does not provide a concrete API to define user defined aggregates even though it is theoretically possible.

Fifth, we plan to bring SamzaSQL generated code closer to Samza Java API for throughput by implementing SamzaSQL specific code generation framework which avoids *AvroToArray* and *ArrayToAvro* steps in message processing flow (Figure 4) by generating expressions that directly work on SamzaSQL specific message abstration (SamzaSQL Data API) and optimizing operator router layer overhead by merging operators such as filter and project with scan operator and moving stream insert operator to other operators which come before the insert operator in the query plan.

Finally, this work lays groundwork for stream query optimizations in SamzaSQL including intra-query optimizations such as sharing of intermediate results, inter-query optimizations such as generating multiple stream jobs for a single query and stream joins related optimizations.

Acknowledgments

The authors thank Chris Riccomini, Jay Kreps, Martin Kleppman, Navina Ramesh, Guzhang Wang and the Apache Samza and Apache Calcite communities for their valuable feedback. This work supported in part through a research grant from Amazon Web Services.

References

[1] B. Plale and K. Schwan, "Dynamic querying of streaming data with the dQUOB system," *IEEE Transactions in Parallel and Distributed Systems*, vol. 14, no. 4, pp. 422 – 432, April 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," *Book chapter*, 2004.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *CIDR*, vol. 5, 2005, pp. 277–289.

[4] L. Chen, K. Reddy, and G. Agrawal, "GATES: A grid-based middleware for distributed processing of data streams," in *Proceedings Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC)*. Honolulu, Hawaii: IEEE Computer Society, June 2004.

[5] B. P. Schroeder, S. Aggarwal, and K. Schwan, "Software approach to hazard detection using on-line analysis of safety constraints," in *16th IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society Press, 1997.

[6] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, 2014.

[7] F. Yan, G. Merlino, and X. Léauté. The RADStack: Open source lambda architecture for interactive analytics.

[8] J. Kreps. (2014) Questioning the lambda architecture.

[9] R. C. Fernandez, P. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang, "Liquid: Unifying nearline and offline big data integration."

[10] ASF. Apache calcite.

[11] B. Plale, "Leveraging run time knowledge about event rates to improve memory utilization in wide area data stream filtering," in *Proceedings Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*. Edinburg, Scotland: IEEE Computer Society, August 2002, pp. 171 – 176.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[13] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011.

[14] A. Calcite. Calcite streaming.

[15] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[16] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, S. Nagaraj, D. Zhang, L. Gao, J. Westerman, P. Ganti *et al.*, "All aboard the databus!: Linkedin's scalable consistent change data capture platform," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 18.

[17] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal-The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[19] ASF. Apache storm.

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

[21] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 2014.

[22] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 668–668.

[23] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," in *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '92. New York, NY, USA: ACM, 1992, pp. 321–330.

[24] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 647–651.