SamzaSQL: Open Source Distributed Fast Data Management System

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

***Abstract*—In a data-driven economy, it's important act on data as it arrives to stay competitive. Specialized message queues like Kafka and specialized streaming processing platforms like Apache Samza was invented to take value out of fast data. But these system still lacks support for standard querying capabilities which is available in *Big Data* processing systems like *Hive* or *Presto*. In this paper, we describe a SQL based streaming data querying and manipulation framework implemented on top of Apache Samza – a open source stream processing framework.**

***Keywords*-Big Data; SQL; Streaming;**

I. Introduction

In today's data-driven economy, organisations – whether *big* or *small* – depends on data analytics to stay competitive. Advancements in *Big Data* related technologies have transformed how organisations interact with data resulting ever increasing data generation rates. To stay competitive in today's data-driven economy, you have to act fast. And previous generation *Big Data* platforms like *Hadoop* couldn't keep up with the decreasing latency requirements. As a result, message queues and stream processing systems like *Kafka*, *Apache S4*, *Apache Storm* and *Apache Samza* which are robust, fault tolerant and scalable were invented to take most out of the fast moving data. These systems allows us to act on data as it arrives, generating value to both organisations and users which wasn't possible half a decade ago at this scale.

While it is important to act on data as it arrives, it is also important for these fast data processing pipelines to interact with static data or output data from batch jobs for the purpose of data enrichment or near real-time analysis. Reference architecture known as Lambda Architecture was proposed as a solution to this hybrid analytics requirements and there are implementations of this architecture such as Summingbird [1] and RADStack [2] in use today. Instead of widely used declarative language like SQL, these frameworks including Summingbird provide low level APIs in languages like Java or custom query languages specific for the framework. But custom programs written for specific purposes add maitenance overhead and often hard to reuse. And custom query languages doesn't have the adavantages of extensions to standard SQL such as

- easy to learn if you know SQL
- clear semantics
- allows combining stored relaitons and streams in a single query

In this paper, we present *SamzaSQL*, an open-source SQL based streaming query engine implementation with support for interacting with non-streaming data sources on top of Apache Samza. SamzaSQL supports queries expressed in standard SQL with minor extensions to add stream producing and windowing capabilities. These queries get compiled into streaming jobs executed in Samza. Depending on the query, these streaming jobs may involve access to static data sources such as stored relations and may access outputs from recurring batch jobs. SamzaSQL front-end, which does query parsing, validation and planning is built on top of *Apache Calcite* and supports querying stream data formats such as Avro or JSON. SamzaSQL also includes support for pluggable catalogs or schema registries based on Calcite's schema factory functionality. This allows us to extend SamzaSQL to support various data sources with custom formats. SamzaSQL's streaming query execution layer is implemented based on CQL [3] continuous query execution model.

Until now, we have used custom programs or data processing pipelines built on APIs provided by streaming and batch data processing frameworks or programming models such as Summingbird [1] to build Lambda or Kappa architecture style applications. General motivation of this work is to explore how we can provide a unified framework – which enables both Lambda Architecture and Kappa architecture style data processing pipelines based on well known SQL related standards and continuous query execution model presented in CQL [3]. TODO: Revisit this because window operator design proposed by Yi may need us to make changes to query execution model.

Windowing is a core and central concept of stream querying due to the fact that operators like *join* and some of the aggregations cannot be evaluated over unbounded streams.

SamzaSQL comes with a widow operator implementation with supports for

- timely and deterministic window output under unpredictable message delevery latencies
- and, deterministic window output with node failures and message re-delivery

TODO: We are mainly targeting soft and near real-time systems. We need to describe this. Also we need to discuss about partition based parallelism supported.

II. SamzaSQL

SamzaSQL is a streaming query layer on top of Apache Samza stream processing framework and utilizes Apache Calcite [4] for query planning.

A. Data Model

- Streams - A stream is a (possibly infinite) sequence of elements. By default, each stream in Samza has a corresponding Kafka [5] topic. There can be multiple partitions for each topic in a Kafka cluster. A element of a stream/topic can be in any format (Generally, elements in the same topic is in same data format). Users can associate stream with the serialization format of the underlying data. SamzaQL provides support for several serialization formats such as Avro, JSON and users can also add new serialization formats by implementing *SerdeFactory* and *Data* interfaces. In the current version, Kafka Schema Registry based Calcite Schema implementation provides stream meta-data necessary to query compilation and planning.
- Partitions - Each partition is an ordered, immutable sequence of messages that is continually appended to a commit log. The elements in the partitions are each assigned a sequential id number called the offset that uniquely identifies each element within the partition. SamzaQL guarantees the ordering of elements within the partition, but not inbetween partitions.
- Tables - Analogous to tables in relational databases, but not limited only to relational databases. *Tables* in SamzaQL can refer to any static data source such as *HBase* table, *Espresso* table which is exposed to SamzaQL environment as relational table. *TODO: Describe semantics little bit more. May need to consider partitions of tables too.*

SamzaQL supports primitive column types (integers, floating point numbers, generic strings, dates and booleans) and nestable collection types - array, map and object. Users can also define their own types via Calcite's JavaType which allows use of a Java Class as a type.

B. Query Language

SamzaQL supports standard SQL with extensions for stream queries. Data definition (DDL) statements are not supported at this stage and SamzaQL depends on Kafka schema registry and Calcite schema models to provide metadata about streams and tables to the query compiling and planning phase. Custom serialization formats are supported through Calcite's schema models. Streams in SamzaQL is immutable and append only, so update and delete operations are not supported on streams. SamzaQL also supports user defined functions for data transformation and aggregations via a Java API.

Streaming specific extensions were added to standard SQL supported by Calcite to support *relation-to-stream* transformation, flexible window specification and stream partitioning.

- **Relation-to-stream transformation**: Following CQL [3] continuous query execution model, SamzaQL first transform the stream into a *time-varying relation* based on the windows specification in the query or using defaults. And query body is executed on this/these time-varying relation/s. So to convert the time-varying relations results from query execution to streams, we use three basic operations *ISTREAM*, *DSTREAM* and *RSTREAM* from CQL. To support this, we have extended SQL *SELECT* statement with *STREAM*, *DSTREAM* and *RSTREAM* keywords which should come right after *SELECT* keyword like below.

```
SELECT [STREAM | DSTREAM | RSTREAM] ... FROM
    ...
```

- **Window Specification**: This is sitll under discussion. But there are some special cases we need to support except well known *tumbling*, *sliding* and *hopping* windows. e.g. Consider that we want to have a count of stock trades (as a infinite stream) happened in the last hour, but only every 11min. It is easy to write the first part in sqlstream as:

```
SELECT STREAM rowtime, count(*)
            OVER (ORDER BY rowtime RANGE
                   INTERVAL '1' HOUR PROCEDING)
  FROM Trades
```

  The above will create a stream of counts that happened every hour continuously as rows are scanned
  Now here is the question:
  - How do we have the count every 11min instead of as the row comes in?
  - If there is no row in Trades between 12:00pm to 2:00pm, how do we tell the system to still generate 0 counts for the time moments: 12:11pm, 12:22pm, 12:33pm, etc.? Or, those rows are delayed in the delivery in the system and user wants to ignore late-arrival of messages after 5min timeout to close the counting window? How can we support that use case w/o breaking SQL grammar?
- **Stream Partitioning**: Not yet decided on how to integrate partitioning. But requirements are as follows:

  - On the producer-side (when inserting query results to a stream), we have no way to determine which key to use for partitioning and how many partitions an output stream should be right now. So there should be a way to specify this in query.

In addition to this, there are situations where queries which are valid in the context of relation databases are no longer valid in streaming scenarios due to non-blocking nature of streaming queries. So, we need to extend Calcite with custom validation queries. These things are still under discussion.

C. Windowing

D. Aggregates

E. Joins

References

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

[2] F. Yan, G. Merlino, and X. Leaute. The radstack: Open source lambda architecture for interactive analytics. [Online]. Available: http://static.druid.io/docs/radstack.pdf

[3] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006.

[4] ASF. Apache calcite. [Online]. Available: http://calcite.incubator.apache.org/

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