Freshet: CQL Based Implementation of Big Data Kappa Architecture on Apache Samza

Milinda Pathirage
School of Informatics and Computing
Indiana University
mpathira@indiana.edu

Beth Plale
School of Informatics and Computing
Indiana University
plale@cs.indiana.edu

ABSTRACT

Abstract goes here. Talks about Kappa Architecture and how CQL concepts are used to implement that.

1. INTRODUCTION

Introduction to Lambda Architecture and Kappa Architecture. Then CQL and how we used CQL concepts to implement KappaQL.

Rest of the paper is organized as follows. Section 2 briefly discuss about CQL which Freshet's query DSL is based on, Lambda Architecture and Kappa Architecture which inspired this work. Then in the Section 3 we discuss the underlying concepts of Freshet, its query DSL and also discuss the Freshet implementation details including its overall architecture. In Section 4, we present some preliminary benchmarks of Freshet and comparison of Freshet query DSL to other modern stream processing programming APIs. Section 5 discuss work similar to Freshet and how Freshet is different/similar from/to them. Then in the following sections 6 7 we conclude the paper and discuss some improvements we can do to Freshet to improve its useability and scalability.

2. BACKGROUND

In this section we discuss the inspirations [2.1, 2.2] behind this work and some of the early works(2.3, 2.4) this work is based upon.

2.1 Lambda Architecture

Lambda Architecture [6] is a framework usefull for designing and implementing reliable, scalable, fault-tolerant and functional big data applications. Designed by Nathan Marz, based on his experience working on distributed big data applications, this generic architecture trying to address various functional and non-functional requirements of distributed data processing systems.

- Fault-tolerance against software, hardware failure and human errors
- Support real-time, near real-time and batch workloads
- Linear scalability and scale out instead of scale up
- Extensibility and flexibility to accommodate changing requirements

Lambda architecture is composed out of three major components – *batch layer*, *speed layer* and *service layer*. Input data is dispatched to both *batch* and *speed* layers. *Batch layer* manages the master dataset while also providing pre-computed batch views. *Speed layer* fill the latency gap of *batch layer* by generating mergeable views from recent data. *Serving layer* is responsible for indexing *batch views* for low-latency querying. Queries are answered by merging views from *batch* and *speed* layers.

Lambda Architecture encourage and emphasize the importance of retaining immutable raw input data, which enables us to process data in ways that didn't originally planned and enables re-computation in case of changing requirements or algorithm error. This can be considered as one of the most important aspects of Lambda Architecture.

Twitter's *Summingbird* [2] is one of the first implementation of Lambda Architecture on top of Scalding and Storm. Summingbird uses mathematical concept *Monoid* to abstracts over both the real-time and batch processing frameworks. Druid [8] is another platform which implements a architecture similar to Lambda Architecture.

Critics [4] argue that maintaining two code bases which produce the same results will be painful and even in cases where language abstractions, need to maintain completely different and complex distributed systems will add lot of overhead. We believe that *Lambda Architecture* can be used a good reference architecture for implementing robust, scalable and flexible Big Data processing systems.

2.2 Kappa Architecture

Kappa Architecture which has the notion of – Everything Is A Stream – is proposed [4] as an alternative to Lambda Architecture. Author of [4] argues that, stream processing is a generalization of data-flow DAGs with support for checkpointing intermediate results and continuous output to the end user. And he emphasizes that we can actually use current distributed stream processing framework like Aapache Samza combine with message queue which retains ordered data like Kafka to implement use cases handled by Lambda Architecture and handle the reprocessing by replaying the stream through new versions of stream processing code or completely new algorithm.

2.3 CQL

CQL [1] - aka Continuous Query Language - is a SQL-based declarative language for expressing queries over data streams and time varying relations. CQL's abstract semantics are based on two data types - streams and relations - and three types of operations - *stream-to-relations*, *relation-to-relation* and *relation-to-stream*. CQL take advantage of well understood relational semantics and keep the language simpler and queries compact by introducing minimal changes to SQL.

- Window specification derived from SQL-99 to transform streams to relations
- Three new operators to transform time varying relations into streams.

Listing 1: CQL Rstream operator and window specification

```
SELECT Rstream(*)
 FROM PosSpeedStr [Now]
 WHERE speed > 65
```

CQL uses SQL for *relation-to-relation* transformation while relations in CQL is different from relations in SQL due to the fact that the CQL relations vary with time. Concepts like *plus-minus streams* which used in CQL prototype to encode both streams and relations in a unified way, *synopses* which is *plus-minus streams* are based on are still useful in contexts like Freshet 3.3. **Need to talk how these concepts are a good match for implementing Kappa architecture**.

CQL also comes with several syntactic shortcuts to reduce the complexity of simple queries as well as couple of equivalences which enable query optimizations.

Freshet uses semantics and concepts in CQL to implement its continuous query DSL as discussed in **??**. More information about CQL concepts and how they are mapped to modern streaming system can also be found in Section **??**.

Need to discuss why we choose CQL. Refer to papers like 'Query Languages and Data Models for Database Sequences and Data Streams' amd Temporal Stream Algebra'. Also talk about non-blocking and blocking operators of SQL in the context of CQL. Also its better to refer to StreamSQL. Best way is to list down the issues each work addressed, solutions and then compare them with CQL and our work.

2.4 Apache Samza

Apache Samza [3] is a open source distributed stream processing systems built on top of Apache Kafka [5] messaging system and Apache YARN [7] resource managment framework. Stream processing logic in Samza applications are built on top of *stream* and Samza *job* abstractions. Samza job can read multiple input streams, process/transform tuples from these streams and append resulting tuples to one or more output streams. Scalability is achieved by *stream partitioning* and dividing a job in to *multiple tasks* where each task consumes data from one partition for each of the input streams. Samza *stream* is a ordered sequence of immutable *messages* of similar type and support pluggable implementations of *stream* abstraction. Data flow graphs or stream processing topologies are composed by using one jobs output stream as other jobs input stream in the context of Samza. Jobs are independent of each other except the dependency between output and input streams of jobs.

Even though Apache Storm is the widely used open source distributed stream processing system, following properties of Samza makes it the most suitable option for implementing Kappa Architecture.

- Samza it self manage the snapshotting and restoration of stream processor's state based on ordered and replayable streams support of Kafka.
- Kafka which is the message layer used by Samza guarantee that messages are processed in order there are written and Kafka ensures that no messages are ever lost. This ordering and fault tolerance features make Samza suitable for implementing time varying relations in CQL.
- Samza utilizes Kafka's partitioning capability to implement scalability. This makes it easy to parallelize CQL operators like aggregation and group-by.
- Samza allows to keep stream processors local state in a key/value storage local to stream processor. This makes it easy to incorporate, concepts like *Synopses* found in CQL to Freshet.

3. FRESHET

- Talk about the main goal
- Use of CQL abstractions
- What type of outputs are generated, the flow input to output
- Describe the DSL using a sample
- Describe the query plan using the same sample above
- Describe how insert/delete streams are used to implement
- How views are implemented as instantaneous relations
- How to use istream, rstream to model push based web apps

Freshet is first step towards a complete implementation of Kappa Architecture based on extension to SQL [1] to support continuous queries. Freshet implements a subset(*select*, *windowing*, *aggregates*) of CQL on top of Apache Samza. Freshet implements *RStream* and *IStream relation-to-stream* operators, tuple and time based sliding windows to convert streams to relations and basic relation to relation operators for implementing business logic. Following CQL, Freshet uses insert/delete stream to model *instantaneous relations*.

Freshet is build out of five main components as shown in Figure 1.

- **Query DSL**: Implemented as a Clojure DSL and used to express CQL queries against streams. Queries expressed in Freshet DSL will get compiled in to relation algebra model and then will get converted into execution plan which consists of set of operators written as Samza stream tasks connected together as a DAG via Kafka queues.

- **Query Compiler**: Compile SQL model generated from DSL to intermediate representation based on relation algebra which can be converted in to execution plan.
- **Execution Planner**: Generate execution plans (Samza jobs connected via input, intermediate and output streams to form a DAG) based on intermediate representation and current status of the Freshet cluster.
- **Scheduler**: Does the actual scheduling of Samza Jobs.
- **Query Operators**: Samza stream tasks. Implement CQL operators like *window*, *select*, *aggregate*, and view generation operators like *rstream*, *istream*. Theses operators, connected via intermediate streams perform stream processing according to the query express in Freshet DSL.

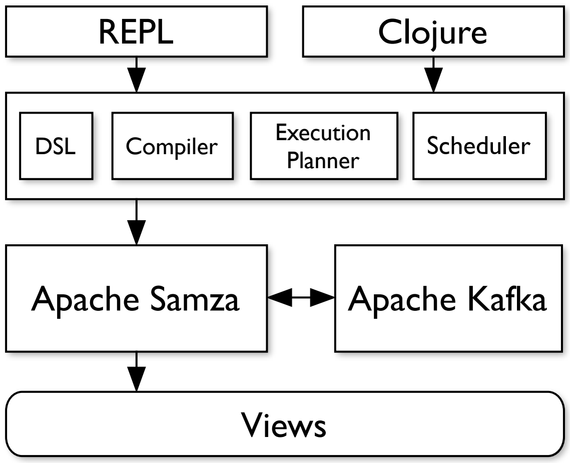

Figure 1: Freshet Architecture

3.1 Query DSL

Define subset of CQL supported and the DSL used to describe it.

3.2 Execution Model

Describe how execution concepts in CQL paper is mapped to Samza based implementation. **Need diagram explaining how queries are executed.**

3.3 Implementation

Describe about operators, parallelisation based on partitions, avro for serde, scheduling.

4. EVALUATION

Implement application on pure Samza and then using this library.

4.1 Example Application

We use several real-time statistics calculation queries on Wikipedia activity stream for demonstration and evaluation of Freshet DSL and the query execution layer. Activity in Wikipedia activity stream is a JSON message which looks like 2.

Listing 2: Wikipedia Activity

```
{
  "anonymous": False,
  "comment": "Changed statement that orbit was
      the eye to saying that the orbit was the
      eye socket for accuracy",
  "delta": 7,
  "flag": "M",
  "namespace": "article",
  "newPage": False,
  "page": "Optic nerve",
  "pageUrl": "http://en.wikipedia.org/wiki/
      Optic_nerve",
  "robot": False,
  "unpatrolled": False,
  "url": "http://en.wikipedia.org/w/index.php?
      diff=449570600&oldid=447889877",
  "user": "Moearly",
  "userUrl": "http://en.wikipedia.org/wiki/User:
      Moearly",
  "wikipedia": "#en.wikipedia",
  "wikipediaLong": "English Wikipedia",
  "wikipediaShort": "en",
  "wikipediaUrl": "http://en.wikipedia.org"
}
```

4.1.1 *The Set of Active Pages*

This query outputs a relation containing, the set of “active pages” at any time instant. Pages we saw in Wikipedia activity stream within the last 60 seconds.

Listing 3: The Set of Active Pages

```
(select wikipedia-activity
    (modifiers :distinct)
    (window (range 60))
```

4.1.2 *All Edits With* bytes-changed *Greater Than 100*

This query outputs a stream of Wikipedia edits where size of change is greater than 100.

This query can be written in three different ways, using *insert stream* operator, *relation stream* operator and using defaults.

- We tell Freshet to convert relation generated by applying window operator to a stream, by adding **:istream** modifier to the query:

Listing 4: bytes-changed > 100 (*insert stream*)

```
(select wikipedia-activity
    (modifiers :istream)
    (window (unbounded))
    (where {:delta [> 100]}))
```

- For *relation stream* we need to use **:rstream** modifier.

Listing 5: bytes-changed > 100 (*relation stream*)

```
(select wikipedia-activity
    (modifiers :rstream)
    (window (now))
    (where {:delta [> 100]}))
```

- Otherwise, we can use defaults

Listing 6: bytes-changed > 100 (*defaults*)

```
(select wikipedia-activity
    (where {:delta [> 100]}))
```

4.1.3 *Hourly Summary of Wikipedia Edits*

In this query we calculate hourly summaries for *number of edits*, *number of bytes added*, and *unique titles* seen.

Listing 7: Wikipedia Activity

```
(select wikipedia-activity
    (window (range 60))
    (aggregate (count :*) :edits)
    (aggregate (sum :delta) :bytes-added)
    (aggregate (count-distinct :pageUrl)
        :unique-titles))
```

5. RELATED WORK

StreamSQL on Spark can be consider as a related work. Also other attempts like Storm Trident, StreamSQL's Storm support. Summingbird [2] which implements Lambda Architecture is also a another related work.

6. CONCLUSION AND DISCUSSION

7. FUTURE WORK

Talks about multi-tenancy, provenance, visualizations.

8. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[2] O. Boykin, S. Ritchie, I. OâĂŹConnell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13), 2014.

[3] A. S. Foundation. Apache samza, 2014.

[4] J. Kreps. Questioning the lambda architecture, 2014.

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

[6] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. O'Reilly Media, 2013.

[7] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[8] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. A real-time analytical data store.