SamzaSQL

Scalable Fast Data Management with *Streaming SQL*

Milinda Pathirage (IU), Julian Hyde (Hortonworks), Yi Pan (LinkedIn), Beth Plale (IU)

School of Informatics and Computing
INDIANA UNIVERSITY

Fast Data

Data has to be process as it arrives, so that we can react to changing conditions fast.

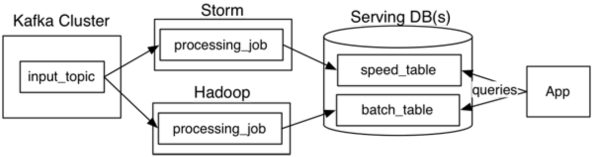
Big data isn't just big; it's also fast.

Big data is often created by data that is generated at incredible speeds, such as click-stream data, financial ticker data, log aggregation, or sensor data.

*John Hugg, **"Fast data: The next step after big data"***

Lambda Architecture (LA)

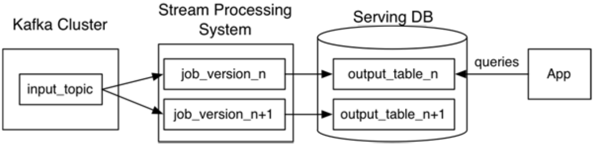
Technology agnostic data processing architecture that attempts to balance latency, accuracy, throughput and fault-tolerance by providing a unified serving layer on top of batch and stream processing sub-systems.



From: `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`

Kappa Architecture (KA)

Simplification of *Lambda Architecture* that uses append-only immutable log as the canonical data store and batch processing is replaced by stream replay.



From: `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`

Motivation

Programming APIs for LA and KA

Summingbird is a well known abstraction for writing *LA* style applications while *KA* style applications were mainly written in **stateful stream processing APIs** provided by frameworks like Apache Samza.

Limitations

- Need to maintain two complex distributed systems
- Users need to understand complex programming abstractions
- Long turnaround times

Summingbird

Word Count

```
def wordCount[P <: Platform[P]]
    (source: Producer[P, String], store: P#Store[String, Long]) =
      source.flatMap { sentence =>
        toWords(sentence).map(_ -> 1L)
      }.sumByKey(store)
```

More examples at `https://github.com/twitter/summingbird`

Samza API

Window Aggregation

```java
public class WikipediaStatsStreamTask implements StreamTask, InitableTask, WindowableTask {
  ...
  private KeyValueStore<String, Integer> store;
  public void init(Config config, TaskContext context) {
    this.store = (KeyValueStore<String, Integer>) context.getStore("wikipedia-stats");
  }
  @Override
  public void process(IncomingMessageEnvelope envelope, MessageCollector collector,
                      TaskCoordinator coordinator) {
    Map<String, Object> edit = (Map<String, Object>) envelope.getMessage();
    ...
  }
  @Override
  public void window(MessageCollector collector, TaskCoordinator coordinator) {
    ...
    collector.send(new OutgoingMessageEnvelope(new SystemStream("kafka", "wikipedia-stats"), counts));
    ...
  }
```

There are sevaral well known SQL-on-Hadoop solutions and most organisations that use Hadoop use one or more SQL-on-Hadoop solutions.

- Apache Hive
- Presto
- Apache Drill
- Apache Impala
- Apache Kylin
- Apache Tajo
- Apache Pheonix

Motivating Research Question

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)?

Can this be done with minimal and well-founded extensions to SQL?

SamzaSQL

Streaming SQL - Data Model

- **Stream:** A stream S is a possibly indefinite par- titioned sequence of temporally-defined elements where an element is a tuple belonging to the schema of S.
- **Partition:** A partition is a time-ordered, immutable sequence of elements existing within a single stream.
- **Relation:** Analogous to a relation/table in relational databases, a relation R is a bag of tuples belonging to the schema of R.

Streaming SQL - Continuous Queries

SamzaSQL

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

CQL

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

Streaming SQL - Window Aggregations

SamzaSQL

```
SELECT STREAM TUMBLE_END(rowtime, INTERVAL '1' HOUR) AS rowtime,
  productId,
  COUNT(*) AS c,
  SUM(units) AS units
FROM Orders
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), productId
```

CQL

```
SELECT ISTREAM ... AS rowtime, productId, COUNT(*) AS c,
  SUM(units) AS units
FROM Orders[Range '1' HOUR, Slide '1' HOUR]
GROUP BY  productId;
```

Streaming SQL - Sliding Windows

SamzaSQL

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

CQL

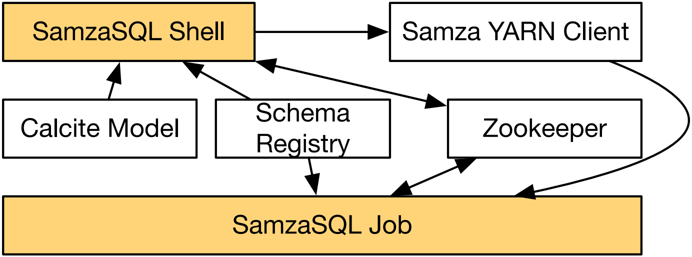
```
SELECT ISTREAM rowtime, productId, units,
  SUM(units) AS unitsLastHour
FROM Orders[Range '1' HOUR]
GROUP BY  productId;
```

Streaming SQL - Window Joins

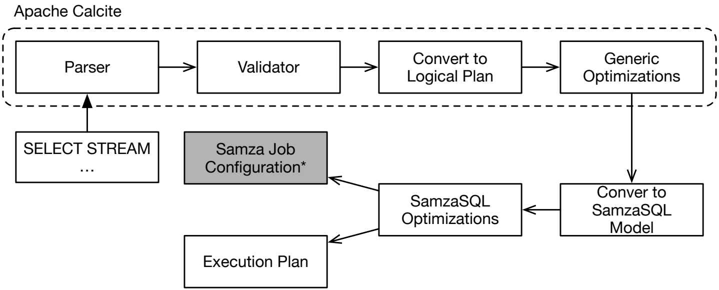
SamzaSQL

```
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
```

SamzaSQL - Architecture



SamzaSQL - Query Planner

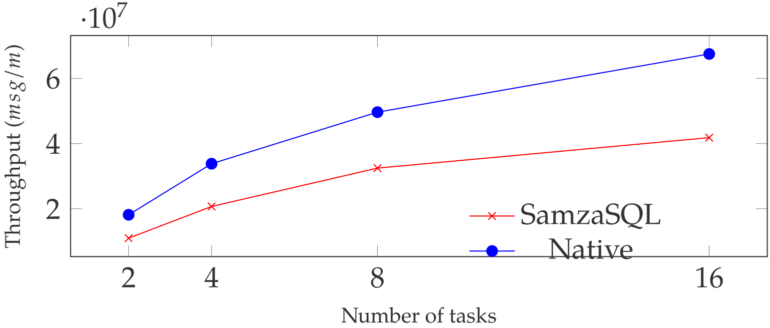


Evaluation

Evaluation - Environment

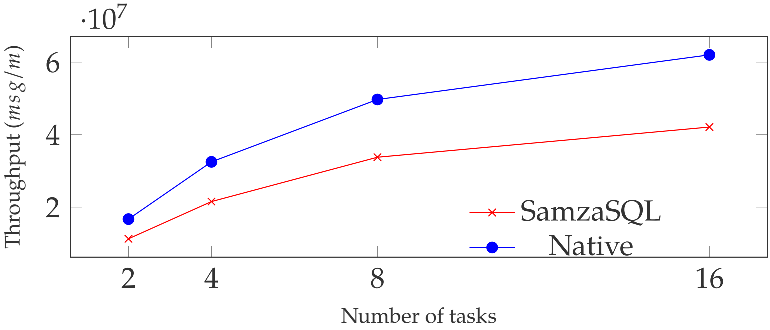
- 100 byte messages (based on previous Kafka benchmarks)
- 3 node (EC2 r3.2xlarge) Kafka cluster
- 3 node (EC2 r3.2xlarge) YARN cluster
- Each r3.2xlarge instance has 8 vCPUs, 61GB of RAM and 160 GB SSD backed storage
- Data model
  - Stream - `Orders (rowtime, productId, orderId, units)`
  - Table - `Products (productId, name, supplierId)`

Evaluation - Filter Throughput



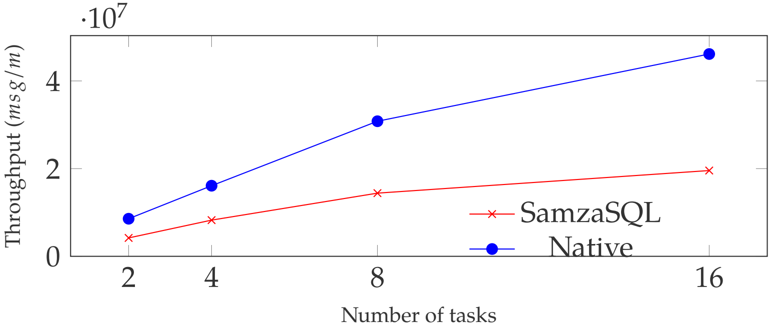

```
SELECT STREAM * FROM Orders WHERE units > 50
```

Evaluation - Project Throughput



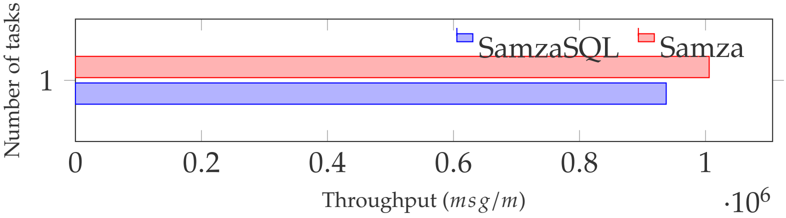

```
SELECT STREAM rowtime, productId, units FROM Orders
```

Evaluation - Stream-to-Relation Join Throughput




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

Evaluation - Sliding Window Throughput




```
SELECT STREAM rowtime, productId, units, SUM(units) OVER (PARTITION BY
productId ORDER BY rowtime RANGE INTERVAL '5' MINUTE PRECEDING)
unitsLastFiveMinutes FROM Orders
```

Sliding window query throughput was measured in a iMac due to limitations in EC2 IO rates.

Evaluation

- SamzaSQL underperform 30-40% compared to native Samza applications mainly due to message format transformations required for streaming SQL runtime
- SamzaSQL join underperform mainly due to local store message serialization/deserialization overheads
- Local storage effects the throughputs directly

Future Work and Conclusion

Future Work

- Ordering guarantees in the presence of stream repartitioning
- Code generation to bring SamzaSQL generated physical plans closer to Samza Java API based queries
- Stream-to-relation queries
- Streaming query optimizations for fast data management systems

Conclusion