Scaling up Near real-time Analytics

@ Uber and LinkedIn

UBER

LinkedIn

Who we are

Chinmay Soman

@ChinmaySoman

- Tech lead Streaming Platform team at Uber
- Worked on distributed storage and distributed filesystems in the past
- Apache Samza Committer, PMC

Yi Pan

@nickpan47

- Tech lead Samza team at LinkedIn
- Worked on NoSQL databases and messaging systems in the past
- 8 years of experience in building distributed systems
- Apache Samza Committer and PMC.

UBER

LinkedIn

Agenda

Part I

- Use cases for near real-time analytics
- Operational / Scalability challenges
- New Streaming Analytics platform

Part II

- SamzaSQL: Apache Calcite - Apache Samza Integration
- Operators
- Multi-stage DAG

UBER

LinkedIn

Why Streaming Analytics

UBER

LinkedIn

Use Cases

Real-time Price Surging

Rider eyeballs

Open car information

KAFKA

Stream Processing

SURGE MULTIPLIERS

Ad Ranking at LinkedIn

LinkedIn Ad View

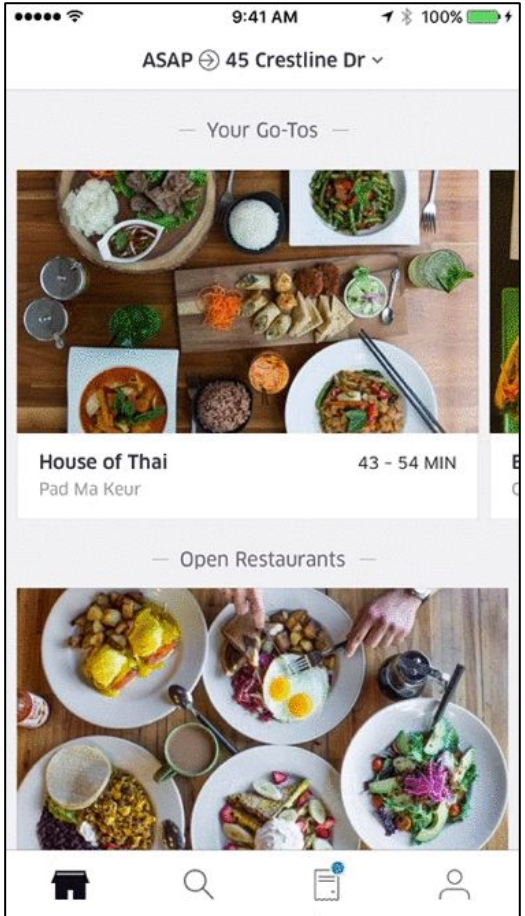
LinkedIn Ad Click

KAFKA

Stream Processing

Ads Ranked by Quality

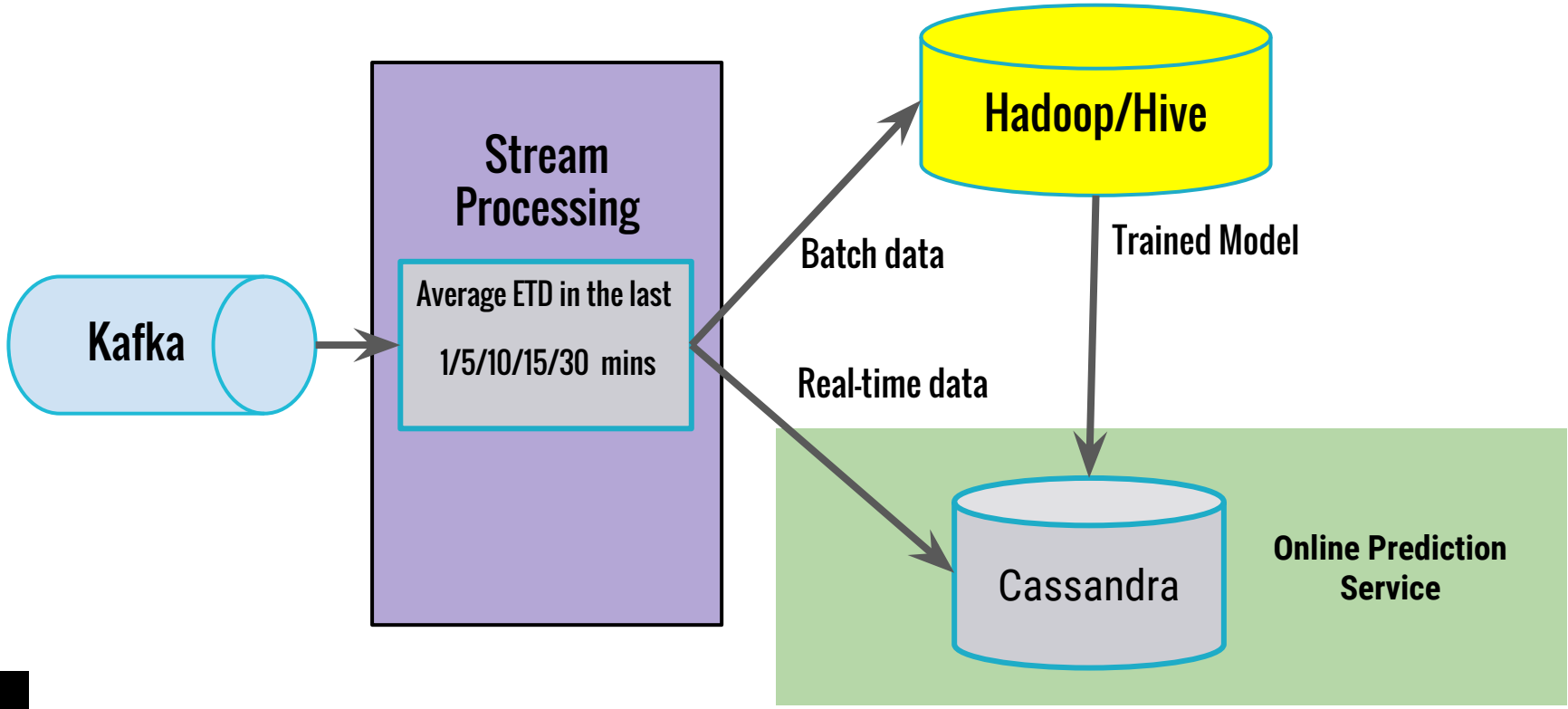
Real-time Machine Learning - UberEats



UBER

LinkedIn

Real-time Machine Learning - UberEats



UBER

LinkedIn

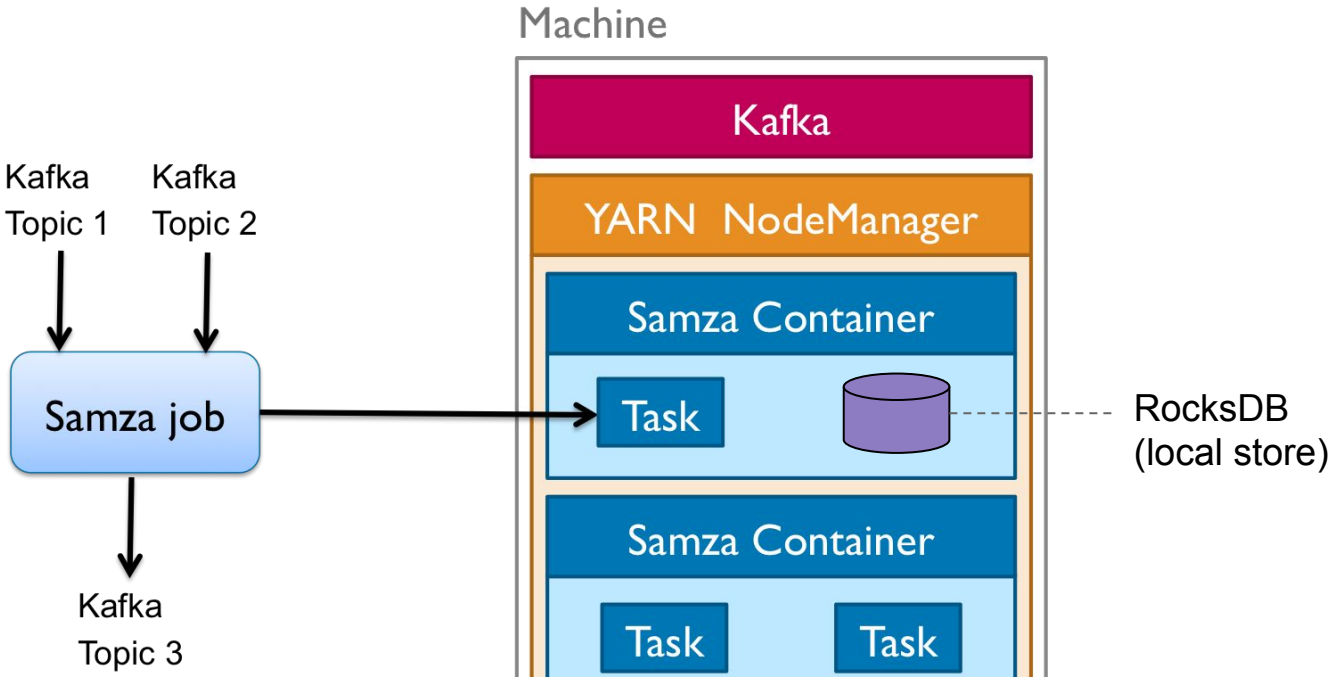
Experimentation Platform

Introduction to Apache Samza

Basic structure of a task

```
class PageKeyViewsCounterTask implements StreamTask {
  public void process(IncomingMessageEnvelope envelope,
                      MessageCollector collector,
                      TaskCoordinator coordinator) {
      GenericRecord record = ((GenericRecord) envelope.getMsg());
      String pageKey = record.get("page-key").toString();
      int newCount = pageKeyViews.get(pageKey).incrementAndGet();
      collector.send(countStream, pageKey, newCount);
  }
}
```

Samza Deployment

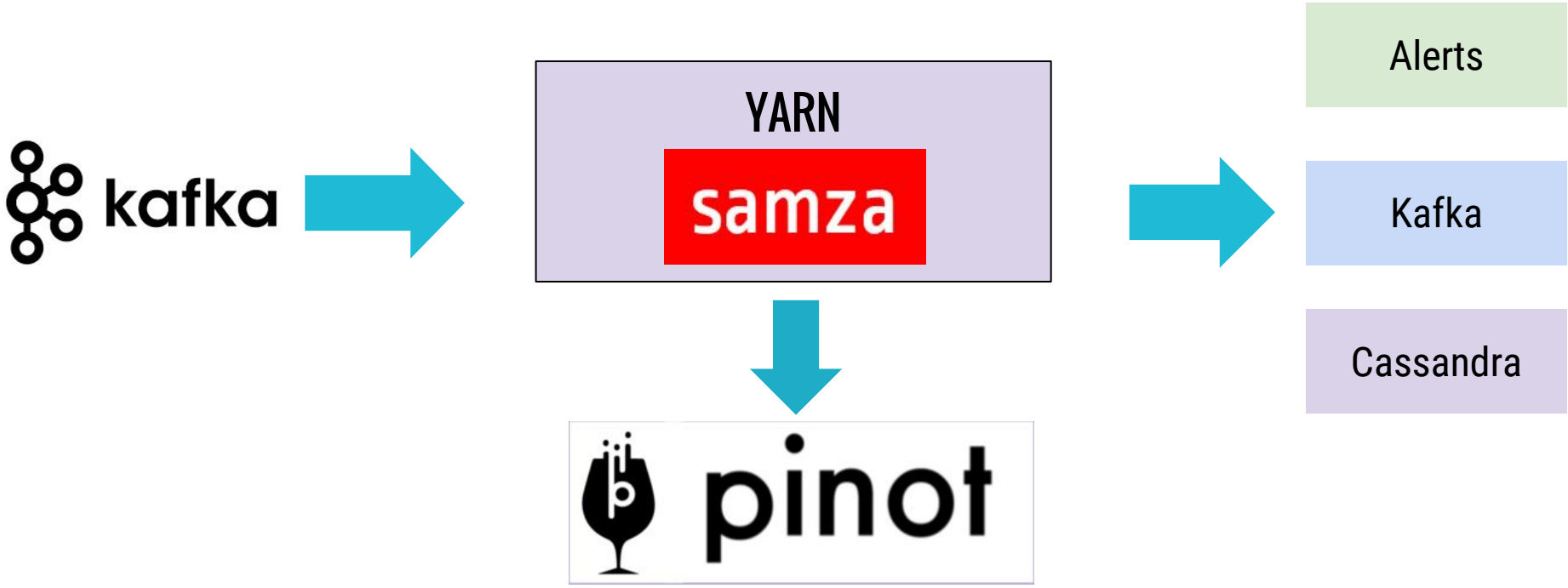


Why Samza ?

- Stability
- Predictable scalability
- Built in Local state - with changelog support
- High Throughput: 1.1 Million msgs/second on 1 SSD box (with stateful computation)
- Ease of debuggability
- Matured operationality

Athena
Stream Processing platform @ Uber

Athena Platform - Technology stack



Challenges

- Manually track an end-end data flow
- Write code
- Manual provisioning
  - Schema inference
  - Kafka topics
  - Pinot tables
- Do your own Capacity Planning
- Create your own Metrics and Alerts
- Long time to production: 1-2 weeks

Proposed Solution

SQL semantics

SURGE MULTIPLIERS

Ads Ranked by popularity

Machine Learning

JOIN

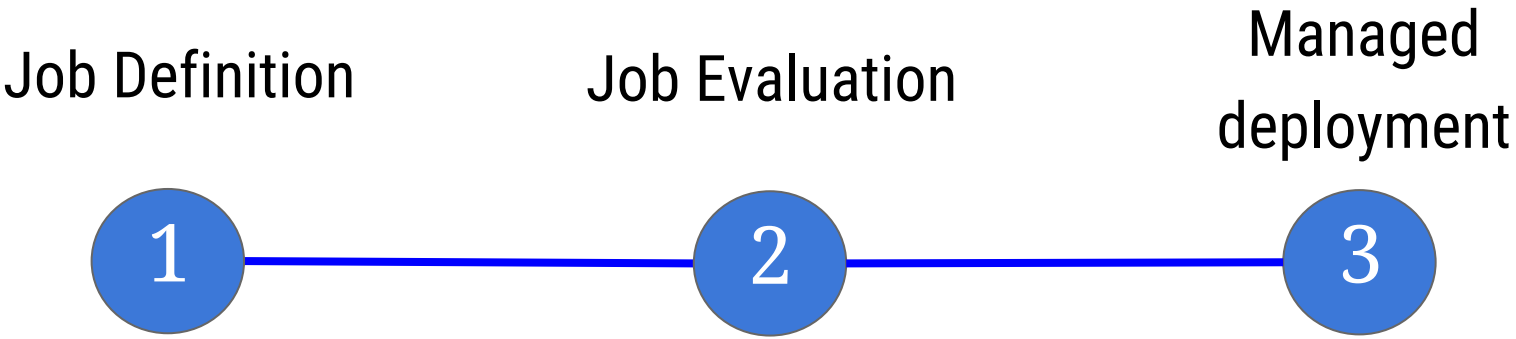
FILTERING / PROJECTION

AGGREGATION

UBER

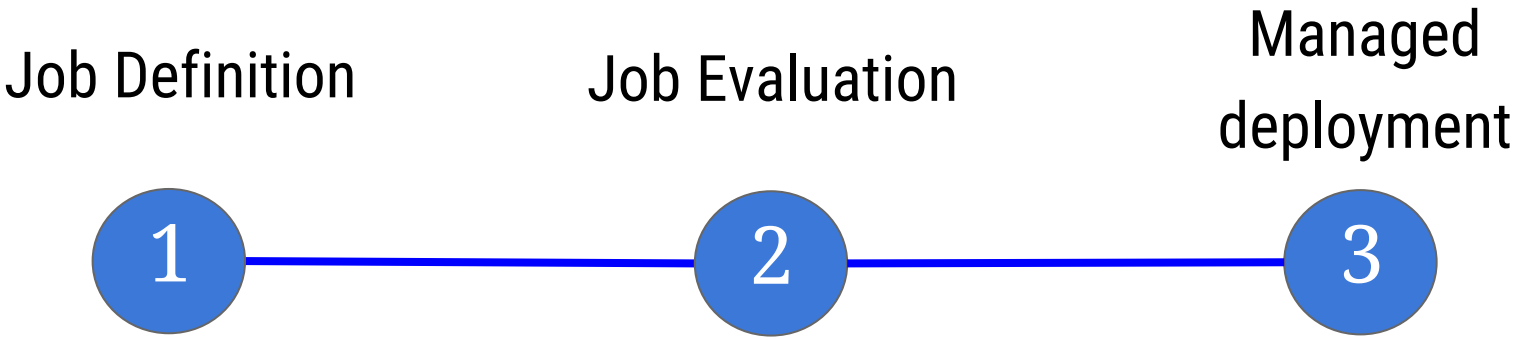
LinkedIn

New Workflow: AthenaX

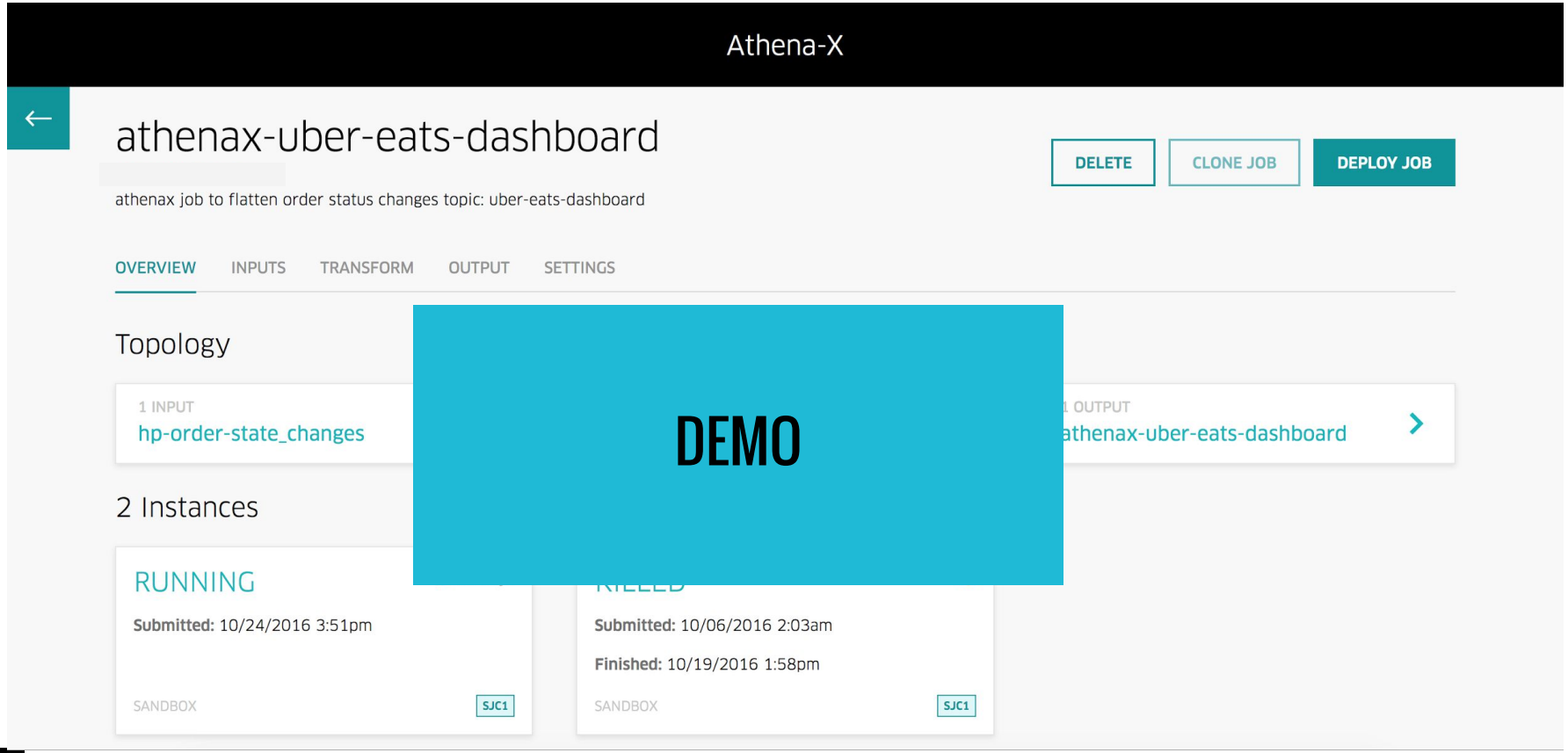


New Workflow: AthenaX

1) Select Inputs
2) Define SQL query
3) Select Outputs



Job definition in AthenaX



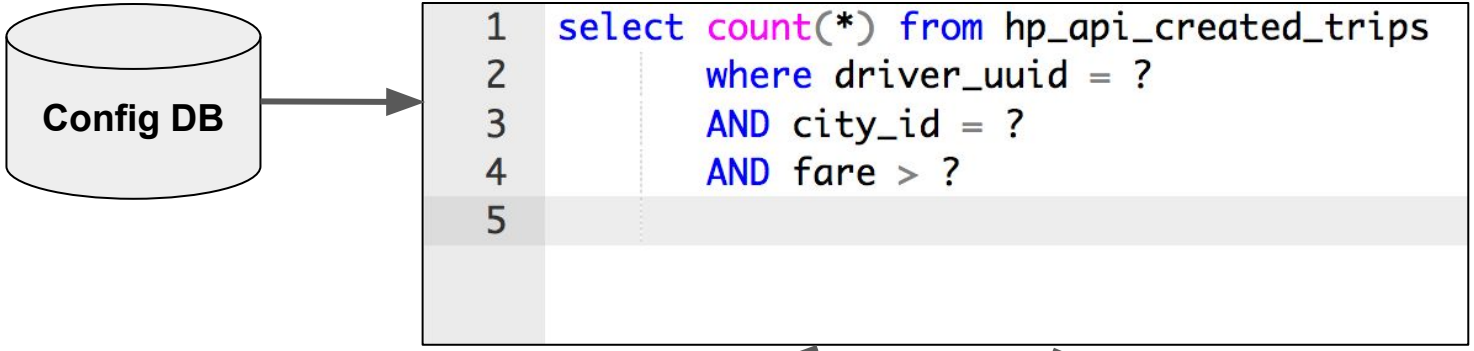
SQL Expression: Example join job

```
1  SELECT STREAM t1.trip_uuid, TUMBLE_END(event_time, INTERVAL '1' HOUR) AS event_time, count(*)
2  FROM hp.driver_log as t1
3  JOIN hp.rider_log as t2
4  ON t1.driver_uuid = t2.driver_uuid
5  GROUP BY TUMBLE(event_time, INTERVAL '1' HOUR), t1.trip_uuid;
6
```

UBER

LinkedIn

Parameterized Queries

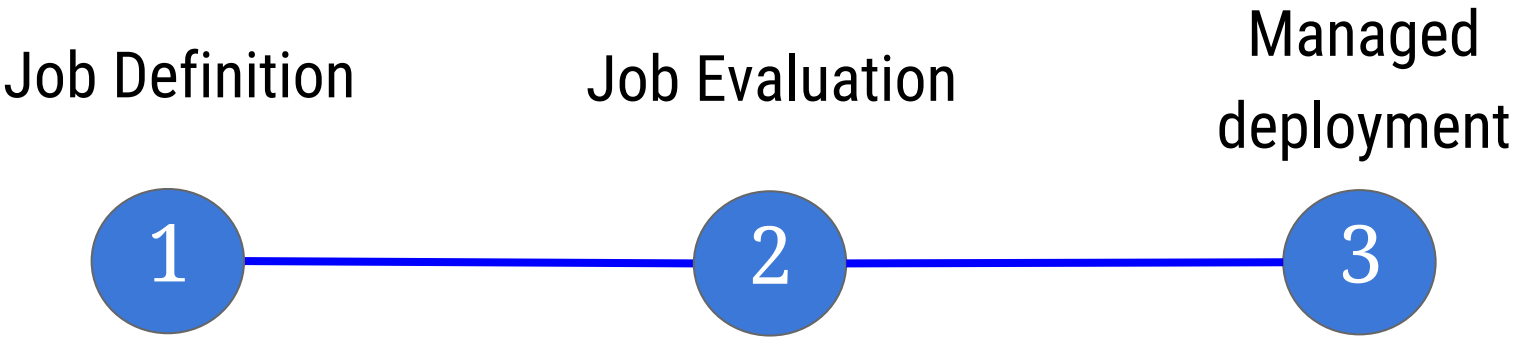



```
select count(*) from hp_api_created_trips
        where driver_uuid = f956e-ad11c-ff451-d34c2
        AND city_id = 34
        AND fare > 10
```

```
select count(*) from hp_api_created_trips
        where driver_uuid = 80ac4-11ac5-efd63-a7de9
        AND city_id = 2
        AND fare > 100
```

New Workflow: AthenaX

1) Schema inference
2) Validation
3) Capacity Estimation



Job Evaluation: Schema Inference

Job Evaluation: Capacity Estimator

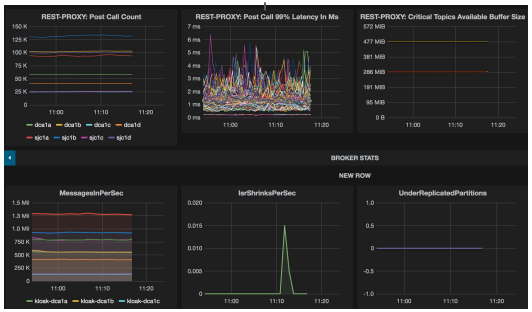
msg/s

bytes/s

Analyze Input(s)

Analyze Query

Test Deployment

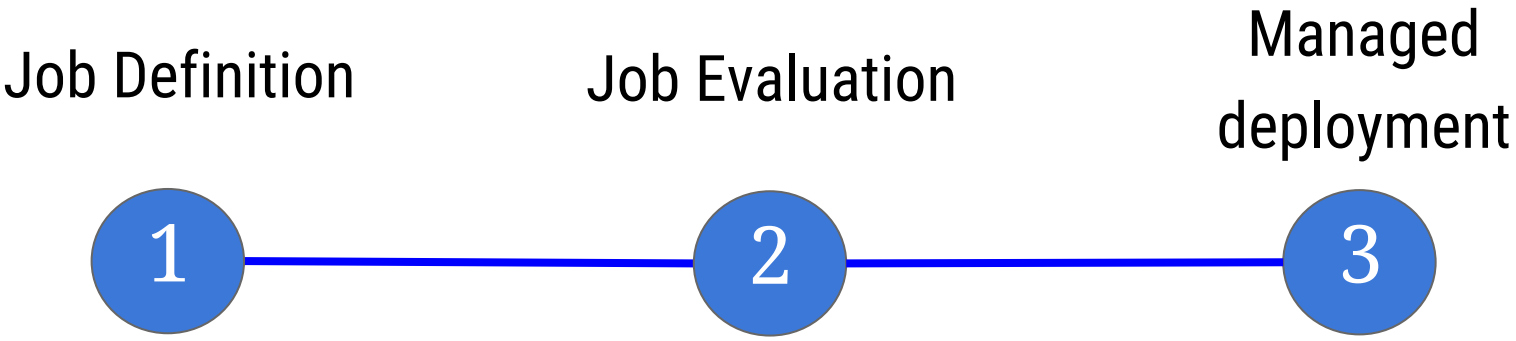


Lookup Table

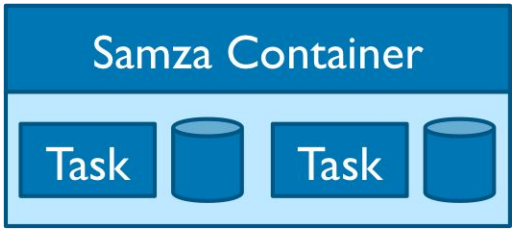
- Yarn Containers
- Heap Memory
- Yarn memory
- CPU
- ...

New Workflow: AthenaX

1) Sandbox, Staging, Production envs
2) Automated alerts
3) Job profiling

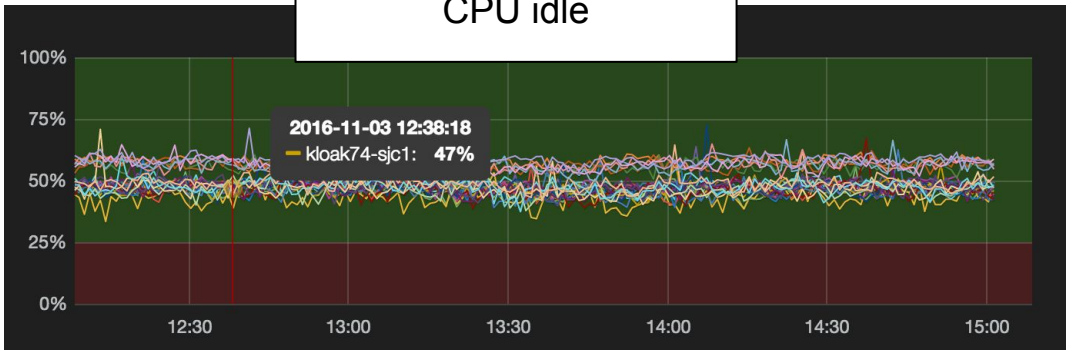


Job Profiling

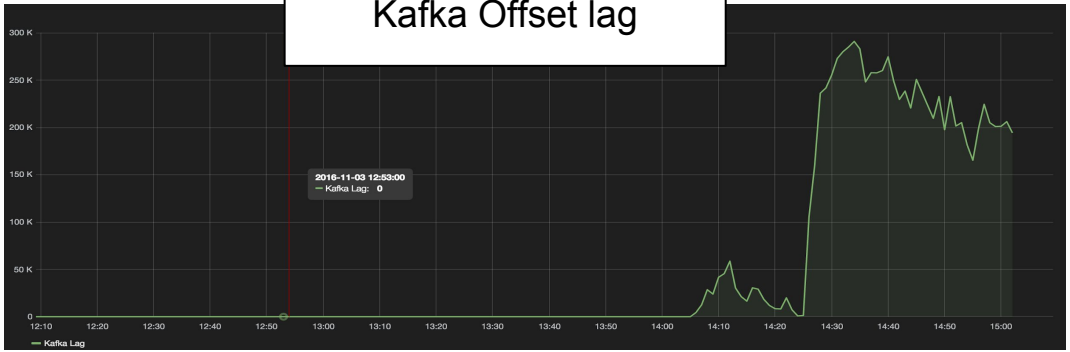


Centralized Monitoring System

CPU idle



Kafka Offset lag



UBER

LinkedIn

Managed Deployments

Sandbox

- Functional Correctness
- Play around with SQL

AthenaX

Promote

Staging

- System generated estimates
- Production like load

Production

- Well guarded
- Continuous profiling

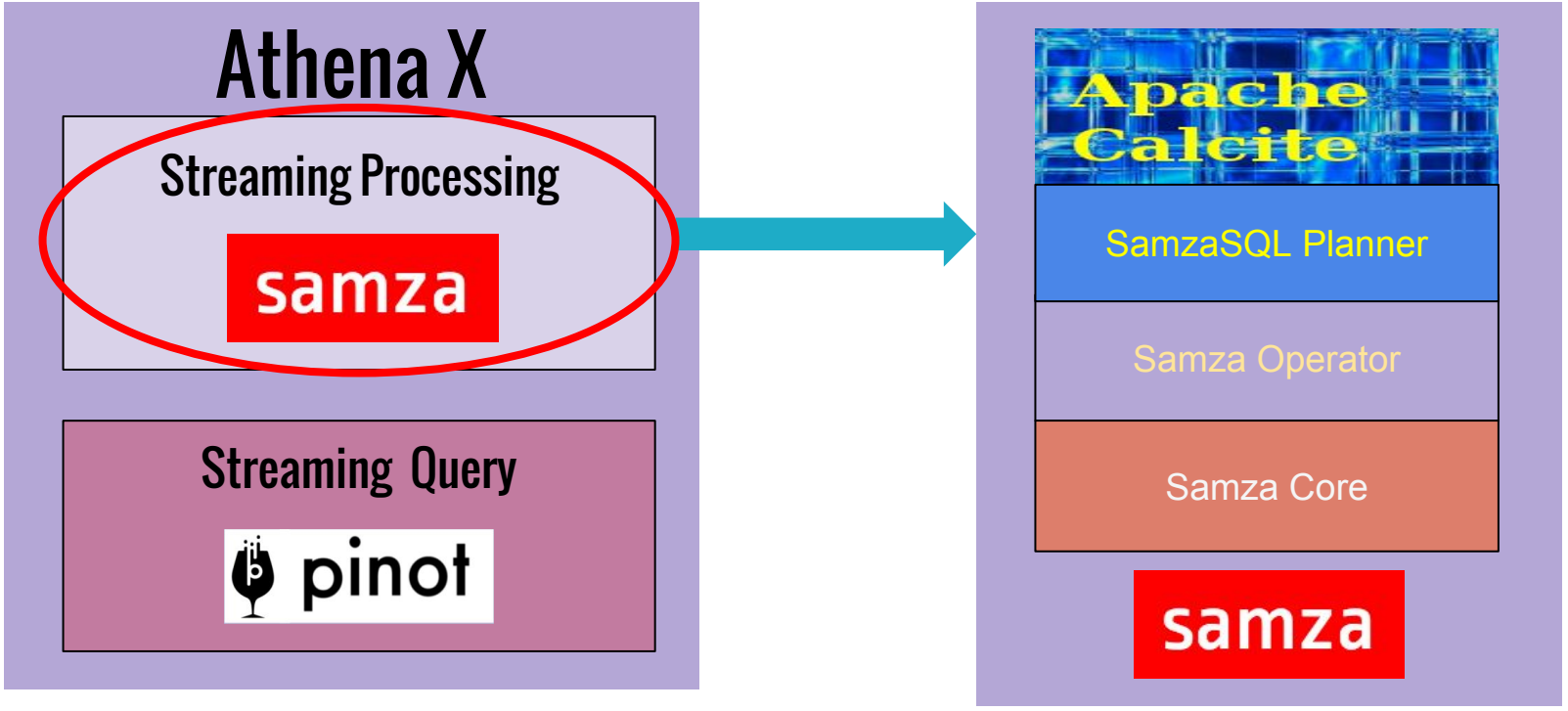
AthenaX: Wins

- Flexible SQL* abstraction
- 1 click deployment to staging and promotion to production (**within mins**)
- Centralized place to track the data flow.
- Minimal manual intervention

UBER

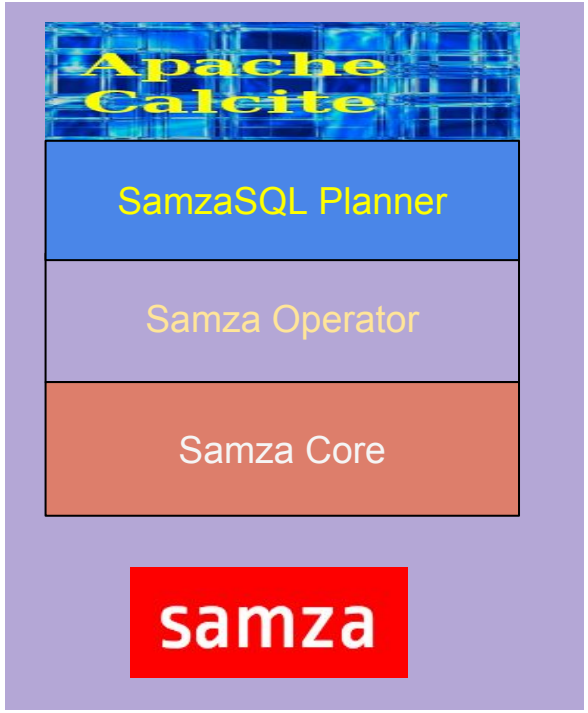
LinkedIn

SQL on Streams



Part II: Apache Calcite and Apache Samza

SQL on Samza



Calcite: A data management framework w/ SQL parser, a query optimizer, and adapters to different data sources. It allows customized logical and physical algebras as well.

SamzaSQL Planner: Implementing Samza's extension of customized logical and physical algebras to Calcite.

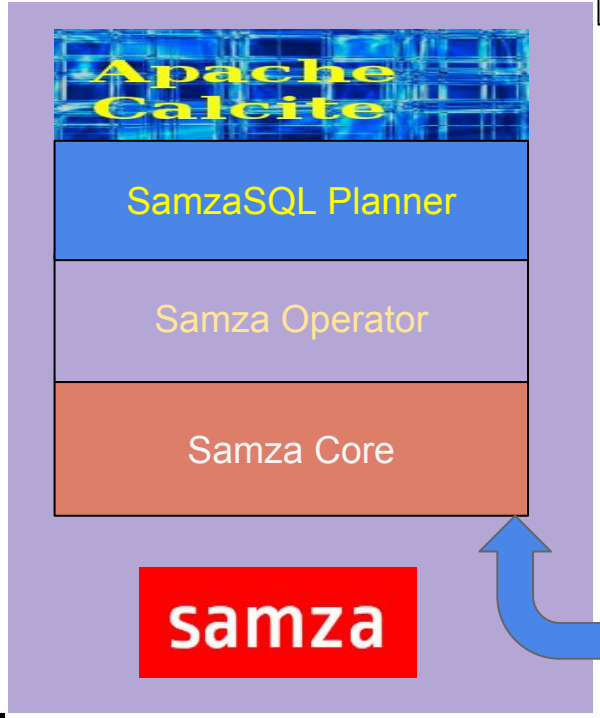
Samza Operator: Samza's physical operator APIs, used to generate physical plan of a query

Samza Core: Samza's execution engine that process the query as a Samza job

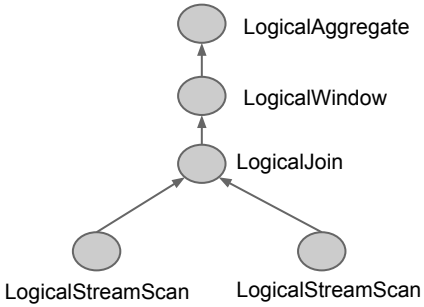
SQL on Samza: Example

SQL query

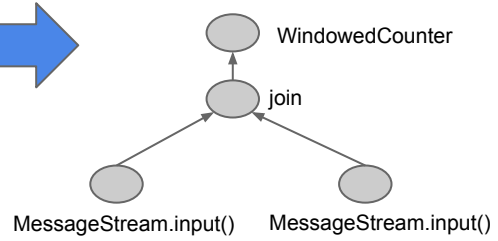
```
SELECT STREAM t1.trip_uuid, TUMBLE_END(event_time, INTERVAL '1' HOUR) AS event_time, count(*)
FROM hp.driver_log as t1
JOIN hp.rider_log as t2
ON t1.driver_uuid = t2.driver_uuid
GROUP BY TUMBLE(event_time, INTERVAL '1' HOUR), t1.trip_uuid;
```



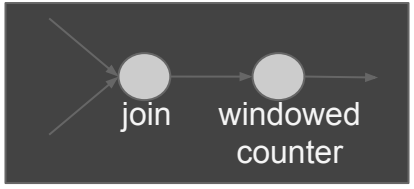
Logical plan from Calcite



Samza Physical plan



StreamOperatorTask

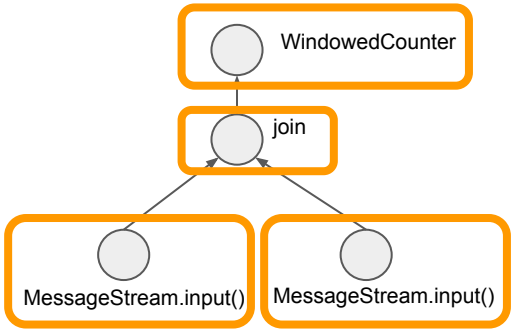


Samza Operator APIs

- Used to describe Samza operators in the physical plan in SamzaSQL
- Support general transformation methods on a stream of messages:
  - map <--> project in SQL
  - filter <--> filter in SQL
  - window <--> window/aggregation in SQL
  - join <--> join in SQL
  - flatMap

Example of Operator API

Samza Physical plan



Physical plan via operator APIs

```
MessageStream.input("hp.driver_log").
  join(MessageStream.input("hp.rider_log"), ...).
  window(Windows.intoSessionCounter(
    m -> new Key(m.get("trip_uuid"), m.get("event_time")),
    WindowType.TUMBLE, 3600))
```

Java code for task initialization

```
@Override void initOperators(Collection<SystemMessageStream> sources) {
  Iterator iter = sources.iterator();
  SystemMessageStream t1 = iter.next();
  SystemMessageStream t2 = iter.next();
  MessageStream.input(t1).join(MessageStream.input(t2).
    window(Windows.intoSessionCounter(
      m -> new Key(m.get("trip_uuid"), m.get("event_time")),
      WindowType.TUMBLE, 3600));
}
```

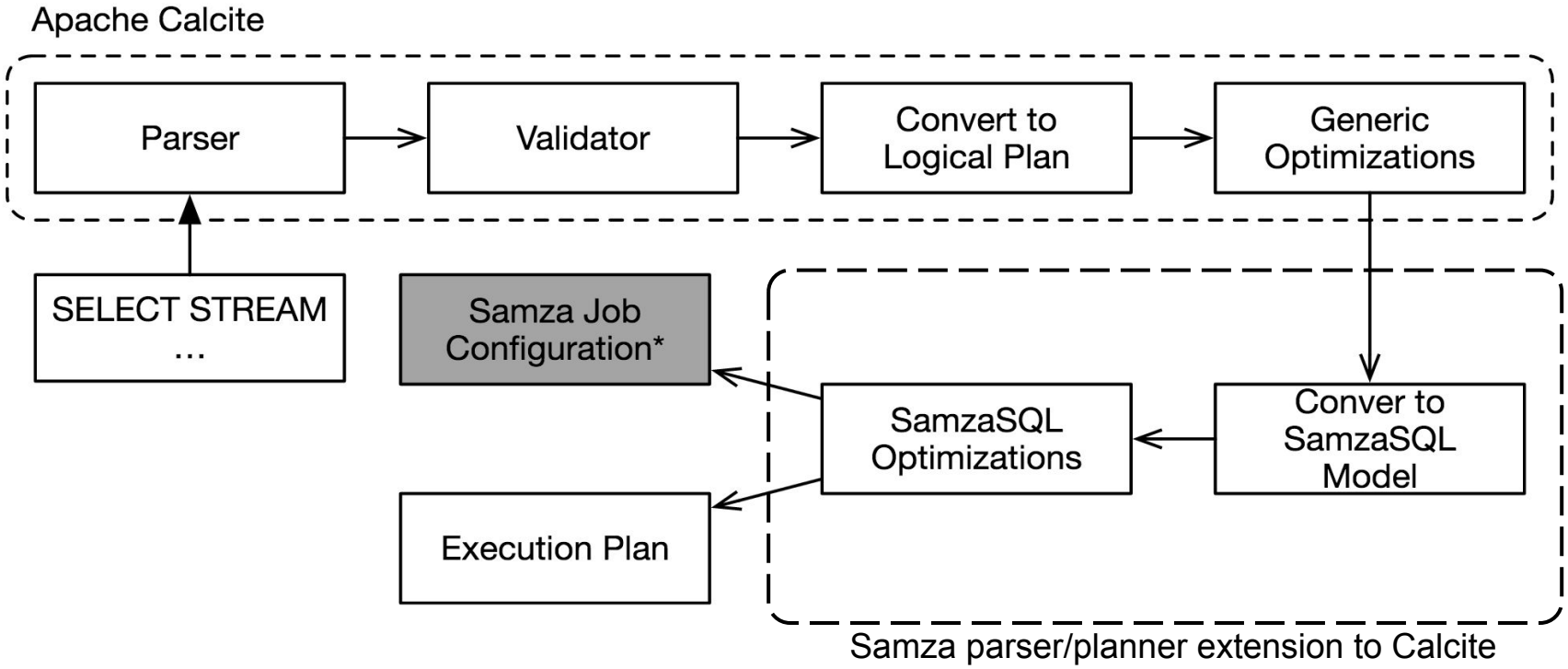
Job configuration

```
task.inputs=hp.driver_log,hp.rider_log
```

UBER

LinkedIn

SQL on Samza - Query Planner



SamzaSQL: Scalable Fast Data Management with Streaming SQL presented at IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) in May 2016

Event Time Window in Samza SQL

- How do we run the same SQL query on event time window?

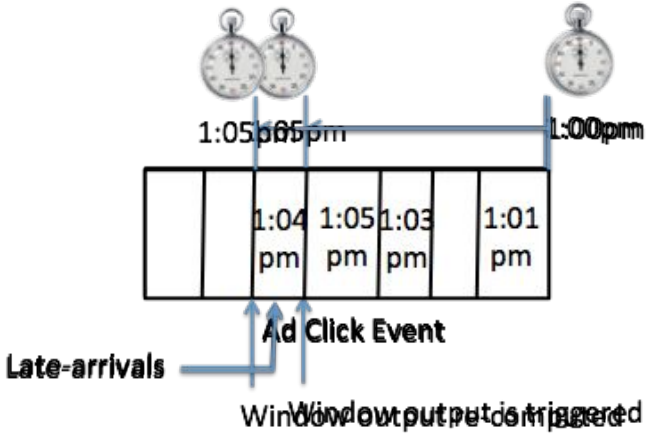
```
SELECT STREAM t1.trip_uuid, TUMBLE_END(event_time, INTERVAL '1' HOUR) AS
event_time, count(*)
FROM hp.driver_log as t1
JOIN hp.rider_log as t2
ON t1.driver_uuid = t2.driver_uuid
GROUP BY TUMBLE(event_time, INTERVAL '1' HOUR), t1.trip_uuid;
```

- Accurate event-time window output in realtime stream processing is hard
  - Uncertain latency in message arrival
  - Possible out-of-order due to re-partitioning

Samza Operator for Event Time Window

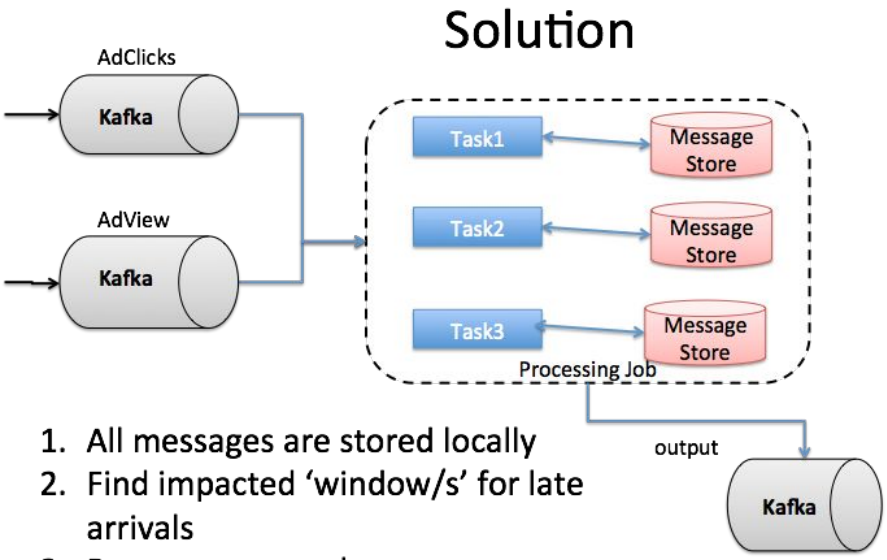
- Solution
  - Use early trigger to calculate the window output in realtime
  - Keep the window state
  - Handle late arrivals using late trigger to re-compute the corrected window output

Concept from Google MillWheel and Stream Processing 101/102



Samza Operator for Event Time Window

- Key to implement the solution:
  - Need to keep past window states
  - Need high read/write rates to update window states
- Samza's local KV-store is the perfect choice for the event time window!



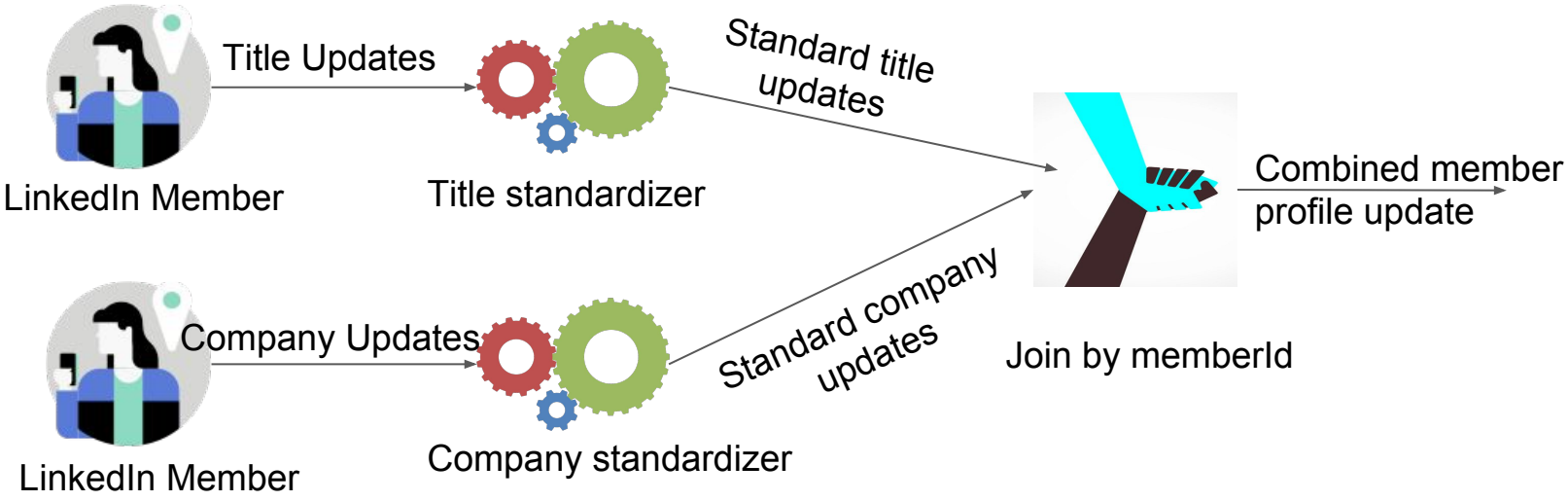
Operator API: Triggers for Event-Time Window

- Samza Operator APIs allow setting early and late triggers for window

```
inputStream.window(Windows.<JsonMessage, String>intoSessionCounter(
  keyExtractor, WindowType.TUMBLE, 3600).
 setTriggers(TriggerBuilder.
   <JsonMessage, Integer>earlyTriggerOnEventTime(m -> getEventTime(m), 3600).
   addLateTrigger((m, s) -> true). //always re-compute output for late arrivals
   addTimeoutSinceLastMessage(30)))
```

Samza SQL: Scaling out to Multiple Stages

- Supporting embedded SQL statements
  - LinkedIn standardizing pipelines



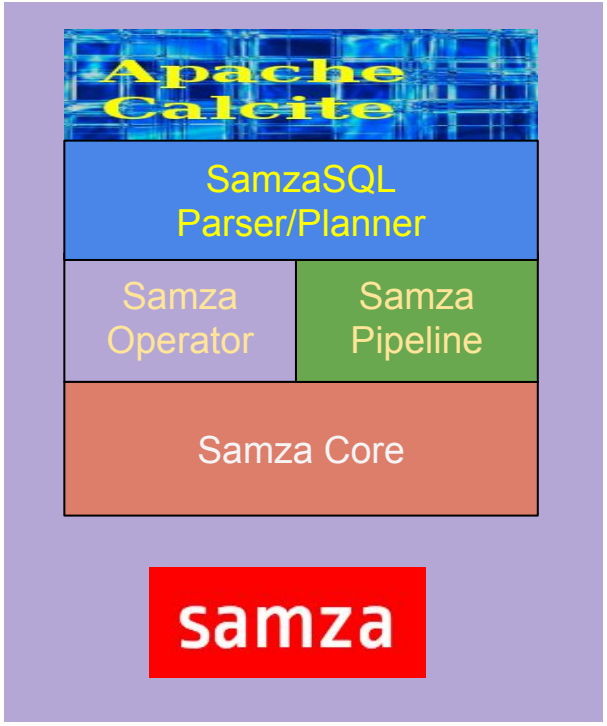
Samza SQL: Scaling out to Multiple Stages

- Supporting embedded SQL statements
  - LinkedIn standardizing pipelines in SQL statement

```
SELECT STREAM mid, title, company_info
FROM (
  SELECT STREAM mid, title_standardizer(*)
  FROM isb.member_title_updates) AS t1
OUTER JOIN (
  SELECT STREAM mid, company_standardizer(*)
  FROM isb.member_company_updates) AS t2
ON t1.mid = t2.mid;
```

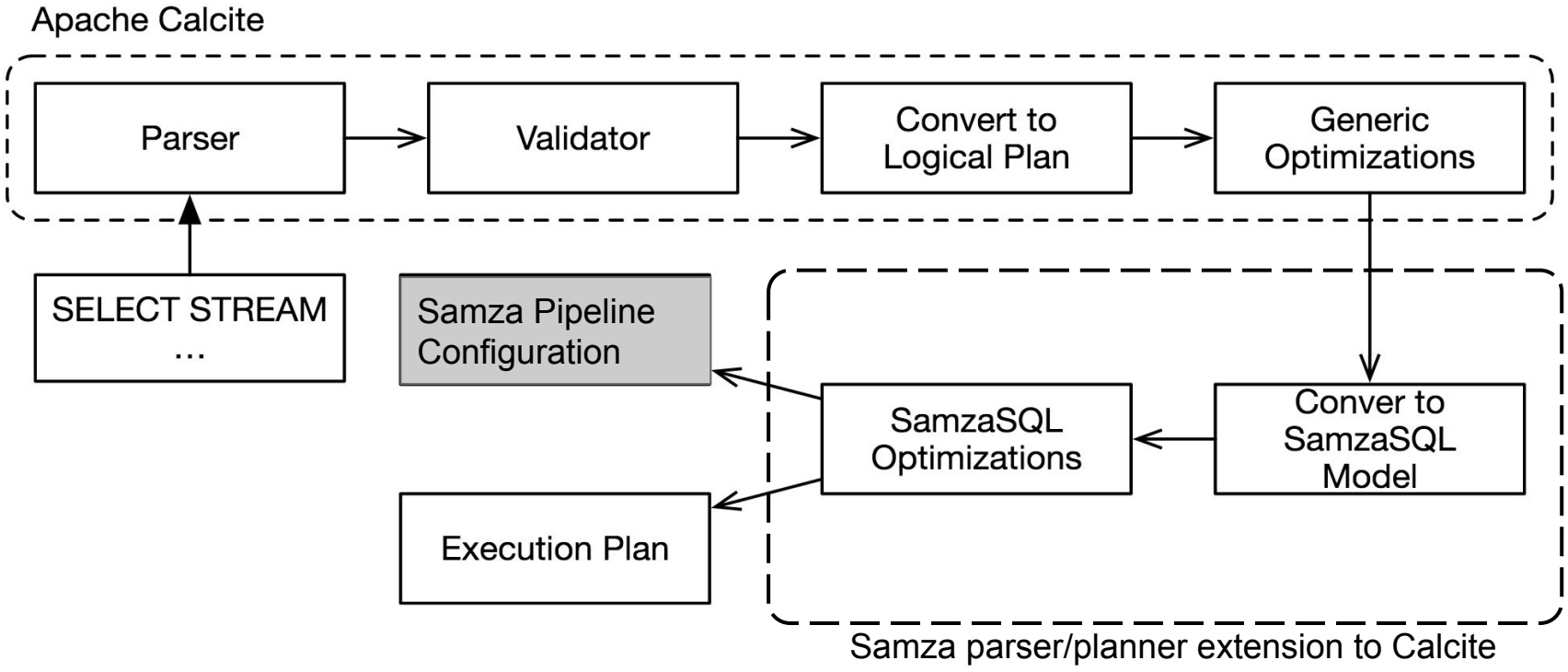
  - Motivations to move the above embedded query statements in different Samza jobs
    - Update machine learning models w/o changing join logic
    - Scaling differently for title_standardizer and company_standardizer due to
      - Different traffic volumes
      - Different resource utilization to run ML models

Samza SQL: Samza Pipeline for SQL (WIP)



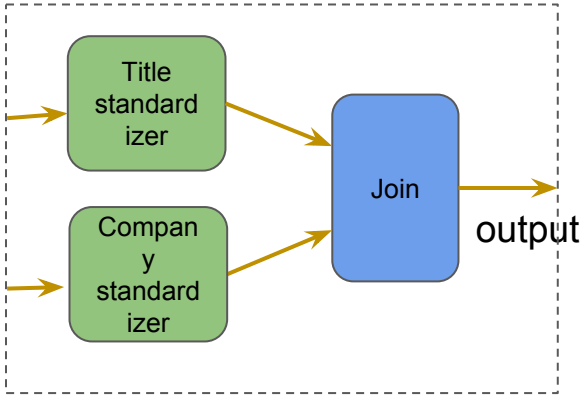
Samza Pipeline: allows a single SQL statement to be grouped into sub-queries and to be instantiated and deployed separately

SQL on Samza - Query Planner for Pipelines



SamzaSQL: Scalable Fast Data Management with Streaming SQL presented at IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) in May 2016

Pipelines for SamzaSQL (WIP)




```
public class StandardizerJoinPipeline implements PipelineFactory {

  public Pipeline create(Config config) {

    Processor title = getTitleStandardizer(config);
    Processor comp = getTitleStandardizer(config);
    Processor join = getJoin(config);

    Stream inStream1 = getStream(config, “inStream1”);
    Stream inStream2 = getStream(config, “inStream2”);
    // … omitted for brevity

    PipelineBuilder builder = new PipelineBuilder();
    return builder.addInputStreams(title, inStream1)
                  .addInputStreams(comp, inStream2)
                  .addIntermediateStreams(title, join, midStream1)
                  .addIntermediateStreams(comp, join,  midStream2)
                  .addOutputStreams(join, outStream)
                  .build();
  }
}
```

Future work

- Apache Beam integration
- Samza support for batch jobs
- Exactly once processing
- Automated scale out
- Disaster Recovery for stateful applications

References

- http://samza.apache.org/
- Milinda Pathirage, Julian Hyde, Yi Pan, Beth Plale. "SamzaSQL: Scalable Fast Data Management with Streaming SQL"
- https://calcite.apache.org/
- Samza operator API design and implementation (SAMZA-914, SAMZA-915)
- Tyler Akidau The world beyond batch: Streaming 101
- Tyler Akidau The world beyond batch: Streaming 102
- Samza window operator design and implementation (SAMZA-552)

Questions ?

csoman@uber.com
yipan@linkedin.com