Streaming your shared ride

Berlin Buzzwords, June 17th 2019
go.lyft.com/berlin-buzzwords-2019

Thomas Weise | @thweise
Agenda

1. Streaming Architecture Overview
2. Streaming Analytics
3. Streaming Applications
4. Integrations / Connectors
5. Deployment
(Streaming) Data at Lyft
Pricing
Dynamic Pricing
Supply/Demand curve
ETA

Fraud
Behaviour Fingerprinting
Monetary Impact
Imperative to act fast

Core Experience
Top Destinations

User Delight
Notifications
Detect Delays
Coupons
Data platform users

Data Modelers  Analysts  Product Managers  General Managers  Data Scientists  Engineers  Experimenters

Analytics  Biz ops  Building apps  Experimentation

Data Platform
Data platform users

- Data Modelers
- Analysts
- Product Managers
- General Managers
- Data Scientists
- Engineers
- Experimenters

Analytics
Biz ops
Building apps
Experimentation

Data Platform
Data Platform architecture

Services (e.g. ETA, Pricing)

Operational Data stores (e.g. Dynamo)

Models + Applications (e.g. ETA, Pricing)

Amazon S3

BI/Data Viz

Apache Superset

Custom apps

Data Discovery app - Amundsen

Marketplace Operations app

Other custom apps
Data Platform architecture

Services (e.g. ETA, Pricing)

Models + Applications (e.g. ETA, Pricing)

Operational Data stores (e.g. Dynamo)

Amazon S3

BI/Data Viz

Apache Superset

Custom apps

Data Discovery app - Amundsen

Marketplace Operations app

Other custom apps
PubSub: From Kinesis to Kafka

- **Latency**
  - Kinesis exhibits high tail latency, with p99 write latency over 100ms and p999 write latency reaching a few seconds
  - Kafka: <20ms p99 write latency and <75ms p999
  - Difficult to achieve latency SLA and durability

- **Fanout limitation**
  - Each Kinesis shard can support at most five read transactions per second, with a maximum total read rate of 2MB per second (per shard)
  - Even enhanced fanout capability with Kinesis 2.0 API is still limited to 5 consumers by default

- **Scalability limitation**
  - Limit on number of shards (by default 500), we are using 1000s
  - Increase only by factor of 2
  - Resharding is manual, disruptive and time-consuming
  - Cost increases with number of shards
Streaming Compute Stack

**Source**
- Kafka

**Streaming Application**
- (SQL, Java, Python via Beam)

**Sink**
- Flink
- Amazon S3
- Elasticsearch

**Stream / Schema Registry**
- Amazon Kinesis

**Deployment Tooling**
- Amazon EC2

**Metrics & Dashboards**
- Flink
- Wavefront

**Alerts**
- Salt (Config / Orca)

**Logging**
- Docker (Development)
Flink Abstraction Levels

High-level Analytics API

Stream SQL / Tables (dynamic tables)

DataStream API (streams, windows)

Stream- & Batch Data Processing

Stateful Event-Driven Applications

Process Function (events, state, time)
## Use Case Categorization

<table>
<thead>
<tr>
<th></th>
<th>Analytics</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
<td>Imperative</td>
</tr>
<tr>
<td>Language</td>
<td>SQL, Table API</td>
<td>Transforms in Java, Python, ...</td>
</tr>
<tr>
<td>Schema</td>
<td>External (tables)</td>
<td>Expressed in programming language</td>
</tr>
<tr>
<td>Execution</td>
<td>Optimized by system</td>
<td>As programmed</td>
</tr>
<tr>
<td>State and time</td>
<td>Automatic state and triggers</td>
<td>Explicit control over state and triggers</td>
</tr>
<tr>
<td>Use cases</td>
<td>Many (data preparation, feature generation, ...)</td>
<td>Fewer with complex, use case specific logic</td>
</tr>
<tr>
<td>TTV/TCO</td>
<td>Lower (self-serve, fully managed, fast onboarding)</td>
<td>Higher (skill set, longer to implement)</td>
</tr>
</tbody>
</table>
Analytics
use case
● **Need - Consistent Feature Generation**
  ○ The value of your machine learning results is only as good as the data
  ○ Subtle changes to how a feature value is generated can significantly impact results

● **Solution - Unify feature generation**
  ○ Batch processing for bulk creation of features for training ML models
  ○ Stream processing for real-time creation of features for scoring ML models

● **How - Flink SQL within fully managed service**
  ○ Use Flink as the processing engine using streaming or bulk data
  ○ Add automation to launch and maintain feature generation programs at scale

https://www.slideshare.net/SeattleApacheFlinkMeetup/streaminglyft-greg-fee-seattle-apache-flink-meetup-104398613/#11
## Dryft Program Specification

**Configuration file**

```
{
  "source": "dryft",
  "query_file": "decl_ride_completed.sql",
  "kinesis": {
    "stream": "declridecompleted"
  },
  "features": {
    "n_total_rides": {
      "description": "All time ride count per user",
      "type": "int",
      "version": 1
    }
  }
}
```

**decl_ride_completed.sql**

```
SELECT COALESCE(user_lyft_id, passenger_lyft_id, passenger_id, -1) AS user_id, 
COUNT(ride_id) as n_total_rides
FROM event_ride_completed
GROUP BY COALESCE(user_lyft_id, passenger_lyft_id, passenger_id, -1)
```
Dryft Program Execution

- Backfill - read historic data from S3, process, sink to S3
- Real-time - read stream data from Kinesis/Kafka, process, sink to DynamoDB
Bootstrapping

- Read historic data from S3
- Transition to reading real-time data
Benefits

- Low latency computation on streaming data
- Fast onboarding
- Minimal development time
- Fully managed
- Self Service
- Reliable
Applications
use case
Dynamic Pricing

- Dynamic Pricing - price evaluated minutely per location bucket
- An Imbalanced Market is Inefficient
  - Too many available drivers: bad
  - Too few available drivers: bad
  - Solution: Price lever controls passenger request rate, which maintains healthy supply levels
- Result: increase price if demand >> supply
What is PrimeTime?

- Belief: There exists some set of optimal price multipliers per location/time bucket
- PrimeTime- Lyft product that sets a multiplier for each gh6 each minute
- Example: In ‘9q8yyv’, at 5:01pm PST, PrimeTime = 2.0
- Scale: Millions of geohashes prices every minute
Legacy architecture: A series of cron jobs

- Ingest high volume of client app events (Kinesis, KCL)
- Compute features (e.g. demand, conversation rate, supply) from events
- Run ML models on features to compute primetime for all regions (per min, per gh6)

SFO, calendar_min_1: {gh6: 1.0, gh6: 2.0, ...}
NYC: calendar_min_1: {gh6: 2.0, gh6: 1.0, ...}
Problems

1. Latency

2. Code complexity (LOC)

3. Hard to add new features involving windowing/join (i.e. arbitrary demand windows, subregional computation)

4. No data driven / smart triggers
Streaming!

Stream / Schema Registry
- Amazon EC2

Deployment Tooling
- Amazon S3

Metrics & Dashboards
- Wavefront

Alerts
- Salt (Config / Orca)

Logging
- Docker
Apache Beam

1. **End users**: who want to write pipelines in a language that’s familiar.

2. **SDK writers**: who want to make Beam concepts available in new languages. Includes **IOs**: connectors to data stores.

3. **Runner writers**: who have a distributed processing environment and want to support Beam pipelines

https://s.apache.org/apache-beam-project-overview
Multi-Language Support

- Started with Java SDK and Java Runners
- 2016: Initiate cross-language support effort
- 2017: Python SDK on Dataflow
- 2018: Go SDK (for portable runners)
- 2018: Python on Flink MVP
- Next: Cross-language pipelines, Samza and other (?) runners
Python Example

```python
p = beam.Pipeline(runner=runner, options=pipeline_options)
(p
  | ReadFromText("/path/to/text*") | Map(lambda line: ...)
  | WindowInto(FixedWindows(120)
    trigger=AfterWatermark(
      early=AfterProcessingTime(60),
      late=AfterCount(1))
    accumulation_mode=ACCUMULATING)
  | CombinePerKey(sum))
  | WriteToText("/path/to/outputs")
)
result = p.run()
```

(What, Where, When, How)
Python via Beam on Flink

```bash
python -m apache_beam.examples.wordcount
    --input=/etc/profile
    --output=/tmp/py-wordcount-direct
    --runner=PortableRunner
    --job_endpoint=localhost:8099
    --streaming
```
Pipeline (conceptual outline)

Lyft apps (phones)

- kinesis events (source)
  - ride_requested, app_open, ...

- filter events
  - valid sessions, dedupe, ...

- aggregate and window
  - unique_users_per_min, unique_requests_per_5_min, ...

- run models to generate features (culminating in PT)
  - conversion learner, eta learner, ...

internal services

redis
Benefits

- Latency: 3 minutes -> 30s
  - Latency now dominated by model execution
- Reuse of model code
- 10K => 4K LOC
- Fewer AWS instances
Integrations
Flink Connectors

- Kinesis Consumer
  - also as custom Beam source
- Kafka Consumer & Producer
- S3 Read & Write
- Elasticsearch
- DynamoDB Streams (special Kinesis Consumer)
- Checkpointing!
  - S3 for checkpoint storage
Challenges

- Production readiness
  - Observability, Configuration, Performance

- AWS integration
  - Transient service errors => retries
  - S3 hot shards with checkpointing => entropy injection

- Event time
  - Source watermarks
  - Watermark skew

- Rate controls
Watermark Skew

partition 1

partition 2

partition 3

5:10pm

5:00pm

event time

5:02 - 5:03

5:01 - 5:02

5:00 - 5:01
Solution: Source synchronization

partition 1

partition 2

partition 3

partition 4

consumer

global watermark

shared state

global watermark

shared state

global watermark
skew leading to large state size

with synchronization
Contribution to Flink

- Support for global aggregates: [FLINK-10887](#)
  - Released with Flink 1.8.0
- Synchronization in Kinesis Consumer: [FLINK-10921](#)
  - Upcoming Flink 1.9.0
- Synchronization in Kafka Consumer: [FLINK-12675](#)
- Long term: New Source Interface: [FLIP-27](#)
  - Framework developed by Flink community
  - Will include watermark alignment capability
Deployment
Flink on Kubernetes

- Goal: Improve stability, flexibility, ease of use, and speed of development
- How? By building a Kubernetes operator that manages Flink applications
- Check it out: [https://github.com/lyft/flinkk8soperator](https://github.com/lyft/flinkk8soperator)

Flink Operator - CRD

- Custom resource represents Flink application
- **Single Flink job**
  (“Flink cluster” == Flink application)
- **Docker image** contains all dependencies
- CRD modifications trigger update (includes parallelism and other Flink configuration properties)

```yaml
apiVersion: flink.k8s.io/v1alpha1
cr: FlinkApplication
metadata:
  name: flink-speeds-working-stats
  namespace: flink
  annotations:
    iam.amazonaws.com/role: 'arn:aws:iam::100:role/abc-iam'
labels:
  app: app-name
  environment: staging
spec:
  image: '100.dkr.ecr.us-east-1.amazonaws.com/abc:xyz'
  flinkJob:
    jarName: name.jar
    parallelism: 10
  deploymentMode: Single
```
(Stateful) Upgrade

Operator detects change to CRD

Running

Updating

Savepointing

New

Creates new Flink cluster, cancels existing Flink job with savepoint

Waits for savepoint to succeed, and updates savepoint location in CRD

Launches new Flink job and tries to transition to Running
Beam Summit Europe
Berlin 2019
19-20 June, 2019
Save the date
Free event
Q & A

We are hiring! lyft.com/careers
Slides: go.lyft.com/berlin-buzzwords-2019