Chronon

Airbnb’s Feature Engineering Framework
Nikhil Simha
nikhil.simha@airbnb.com
Announcements

You are in the right place!
Renamed to “Chronon” from zipline

**Private Beta** - user / contributor

   If you are interested drop a mail to

   nikhil.simha@airbnb.com or jack.song@airbnb.com
Agenda

Goals and Requirements
API Overview
  Concepts & Examples
Dependencies Overview
Integration guide
Goals - management

Uniform API
- Python + Spark SQL
- Online & Offline
- Raw Data -> Training Data
- Raw Data -> Feature Serving

Feature Repository
- Compiled
- Team based

Feature monitoring
Goals - API

Powerful & Composable Building blocks

Source types
  Entities Events & Cumulative Events
GroupBy - Aggregation engine
Join - PITC joins
Staging Query
  Arbitrary ETL to prepare data
Goals - computation

Log & Wait vs Backfill
- Large models -> large training data ranges -> lot of waiting
- New features are mostly derived from existing raw data

Realtime Features
- Hardest systems problem in ML
- Stream processing + Batch processing + Storage + Fetching
- Backfills
Non-Goals

- No Model Training or Serving
- Not for interactive exploration
  - Spark vs Clickhouse/Druid
  - Static usage is fine
Requirements

- **Kafka**
  - Event Store
  - or BYO

- **Hive** (optional)
  - Batch-Catalog
  - or BYO

- **Spark**
  - Compute Engine

- **KV Store**
  - Bring-Your-Own

- **Airflow**
  - Scheduler
  - or BYO
Offline - problem statement (item recommendation)

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
</tr>
</tbody>
</table>

- view_count_5h
  - From view stream
- avg_rating_90d
  - From ratings db table
## Offline - problem statement

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
<th>views_count_5h</th>
<th>avg_rating_90d</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
<td>10</td>
<td>3.7</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
<td>7</td>
<td>4.5</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
<td>35</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Online - problem statement

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
<th>views_count_5h</th>
<th>avg_rating_90d</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
<td>10</td>
<td>3.7</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
<td>7</td>
<td>4.5</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
<td>35</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Examples – E-Commerce platform

Count of Item views of a user in the last 5 hours – from a item view stream
Average rating of an item in the last 90 days – from a ratings table

Count / Average – Aggregation operations
Item Views/Rating – Aggregation Inputs
User/item – Aggregation Key
Last X days – Aggregation Window
Ratings Table/ Item View Stream – Data Source
Accuracy - Real-time or Daily
Data Sources
Service Fleet

Production Database

Change Capture Stream

Event Stream

DB Snapshot

Change capture log

Event log

Derived Data

Media
Sources - Events

- Each partition contains data/events that occur in $[ds, ds + 1]$
- fct sources/dim sources
- PITC -> hive table
- materialized view -> topic
Sources - Entities

- Each partition contains data for all entities - as of ds (date_string)
- DB Table snapshots
  - Sqoop
- Mutations! (CDC)
  - Mutations Table & a Mutation Topic
  - Debezium + Kafka
- PITC -> snapshot table + mutation table
- materialized views -> snapshot table + mutation topic
Sources - Cumulative

- Insert only tables
  - Each new partition is a superset of any old partition
- Latest partition is enough to backfill features at arbitrary points in time
- No deletes/updates - mutations table not needed
- Events in db tables
Sources - Why?

Error-prone date wrangling

fct/event scan = partition_of(min_query_ts - max window)

cumulative scan = latest_partition

entity scan

    snapshot_table - partition_of(min_query_ts) - 1

    mutation_table - partition_of(min_query_ts)

Optimization hints!
GroupBy
Concepts - GroupBy

- Group of Features derived from the same/similar sources of data
  - Data Source
    - From + Where + Select - powered by spark sql
  - Keys
  - Aggregations
    - Input
    - Operation
    - Window - optional & hourly or daily
    - Bucketing - ratings by category - Map [category -> rating]
Concepts - Aggregations

SUM, COUNT, AVG, VARIANCE, MIN, MAX, TOP_K, BOTTOM_K, FIRST, LAST, FIRST_K, LAST_K, APPROX_DISTINCT, FREQUENT_ITEMS, HISTOGRAM...

Commutative and associative - order independent & mergeable

Sometimes reversible - CDC updates
Windows – Sliding

- Freshness
- Memory intensive
Windows – Hopping

Query tail 1:27
Sliding Window e3 + e4 + e5
Hopping Window e4 + e5 + e6

1:00 1:10 1:20 1:30 1:40 1:50 2:00 2:10 2:20 2:30
Windows – Hopping

- Staleness
  - As stale as the hop size
- Memory Efficient
  - One partial per hop
Windows – Sawtooth

Sliding Window: e3 + e4 + e5
Query: 2:27

Hopping Window: e4 + e5 + e6

Sawtooth Window: e3 + e4 + e5 + e6
Windows – Sawtooth

• Freshness
  • Writes are taken into account immediately

• Memory
  • Partial aggregates per hop
Windows – Sawtooth

- Catch
  - sum/count vs others
- Consistency
Join
Concepts - Join

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
</tr>
</tbody>
</table>

- **view_count_5h**
  - From view stream
- **avg_rating_90d**
  - From ratings db table
# Concepts - Join

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
<th>views_count_5h</th>
<th>avg_rating_90d</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
<td>10</td>
<td>3.7</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
<td>7</td>
<td>4.5</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
<td>35</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Concepts - Join

- Join multiple GroupBy-s (feature groups) together
  - Decide to show a particular user a particular item – likelihood to buy
    - X  User Features groups
    - Y  Item Features
    - Z  (User, Item) Features – past interactions
- Gather both Online & Offline
- Left & rights
  - labelled data + timestamped keys & feature derivations
Workflow
User workflow

explore.py  Author  compile.py  run.py
Explore

- Key word search “view” / “rating” etc.
- Search for features in a group
- Models using a feature
- Data Sources for a feature
- Features used in a model
- Feature group authors + last changed
Compile

- Py is powerful
- Complete & Final representation
- Change Reasoning
- Hand-off to scala engine
Run.py - testing

- Offline flows
  - Join – training data generation
  - StagingQuery – arbitrary ETL
  - GroupBy – midnight accuracy – metrics style

- Online flows
  - Lambda – batch + streaming
  - Fetching join & groupBy
  - Uploading metadata
Scheduling needs

- Feature Declaration
- Streaming Updates
- Batch partial aggregates

Feature Store

- Feature Client
- Model

Model Server

Application Server
Run.py - scheduling

- Airflow based - but flexible
- Joins: DAG each
- GroupBy: DAG per team
  - Lambda Serving
  - Streaming task is “heartbeat-or-restart”
- Staging Query: DAG per team
Repo structure

- staging_queries - free form etl
- group_bys – aggregation primitive
- joins – gathering multiple groupBy’s

Folder/module per “team”
  - teams.json

Compiled artifact folder
Scripts - spark batch & streaming jobs + fetch online jar
Repo structure - one time setup

- Scripts
  - spark batch job submission
  - spark streaming jobs
  - fetch online api implementation jar
Workflows – offline

- Idempotency / Auto backfill
  - Job always tries to fill in all of its unfilled range
  - Airflow convention is task instance per date
    - Re-use compute & Natural ML user-flow

- Staging Queries
  - Free form ETL
  - Spark SQL Based

- Join Backfills – already covered

- GroupBy Standalone Backfills
Workflows – Online

- Read optimized materialized views
  - Low latency ~10ms, high QPS

- Based on
  - Kafka
  - Spark Streaming
  - General KV Store API
Online Integration API

- One time integration
- **KV Store**
  - Point Read + Scan from timestamp
  - Single Write + Bulk Write
- **Streaming**
  - Decode Bytes into a Row in Chronon Schema
  - Intersection of Avro & Parquet

**Airflow Scheduling**
- We provide airflow integration template
Perf Stats

- Serving
  - Read: latency, qps, payload sizes - breakdown by groupBy
  - Streaming Write: Freshness, qps, payload size
  - Bulk write: Compute time, data sizes etc.
- Training data generation
  - Compute time – breakdowns
  - Row count
Data Stats

- Online offline consistency
  - Numerical: SMAPE
  - Categorical: Inequality percentage
  - Lists: Edit Distance
- Feature Quality
  - Coverage
  - Cardinality
  - Distribution
  - Correlation
Cases

- Online / Offline
- Backfilled / Logged
- PITC / Midnight accurate
- Events / Entities / Cumulative
- Windowed / Lifetime Aggregations
- Reversible / Non Reversible
- Single Column, Single Aggregation, Single window
Problem statement - Events PITC

<table>
<thead>
<tr>
<th>user_id</th>
<th>timestamp</th>
<th>views_count_7d</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>2021-09-30 5:24</td>
<td>10</td>
</tr>
<tr>
<td>bob</td>
<td>2021-10-15 9:18</td>
<td>7</td>
</tr>
<tr>
<td>carl</td>
<td>2021-11-21 7:44</td>
<td>35</td>
</tr>
</tbody>
</table>
Naive approach

SELECT user, query.timestamp as query_timestamp, COUNT(view_id) as view_count_7d
FROM queries JOIN views ON
  queries.user = views.user AND
  view.timestamp < queries.timestamp AND
  view.timestamp >= (queries.timestamp - 7d) -- 7 * 24 * 3600 * 1000 milliseconds
GROUP BY user, query_timestamp

Complexity?
Naive approach

```python
result = []
for query_ts in queries:
    view_count = 0
    for view_ts in views:
        if view_ts < query_ts and view_ts > query_ts - millis_7d:
            view_count += 1
        result.append((query_ts, view_ts))
# result now contains the desired data
```

Complexity?

\( N^2 \)
Can we do better?

- sort + cursors
- Complexity? $n \log(n)$
- Distribute friendly?
- Use of subtraction - doesn’t work for max, min etc.
- Even better?

```python
def view_stats(views, queries):
    result = []
    start = 0
    end = 0
    count = 1
    sorted_views = sorted(views)
    for query_ts in sorted(queries):
        query_start = query_ts - 7 * day_millis
        # scan forward the start cursor and decrement the counter
        while start < len(sorted_views) and sorted_views[start] < query_start:
            start += 1
            count -= 1
        # scan forward the end cursor and increment the counter
        while end < len(sorted_views) and sorted_views[end] < query_ts:
            end += 1
            count += 1
        result.append((query_ts, count))
    # result now contains desired data.
```
Some important observations

- Windows overlap a lot for a given key
- Label data is usually much smaller than raw data
- Fraction of keys that engage on the platform is small
  - The fraction with labels could be even smaller.
Approaches

- Windows overlap a lot for a given key
  - Break windows into reusable tiles.
- Label data is *usually* much smaller than raw data
  - Use labels/queries to determine the tiles effectively
- Fraction of keys that engage on the platform is small
  - Use a compact approximate structure to filter out “most” of unwanted keys
  - Bloom filter - false positives are okay, true negatives are not.
Tiling windows

Incoming Event (ts, payload)
Window tiling

- Hopping tail is common across all queries that fall into the head!
- The idea is to compute tails and heads separately.
Window tiling

- What if queries don’t fit in memory?
  - Tiling can't be dynamic(query dependent)
- Hops?
  - Let’s examine window semantics
Window tiling

- We need to stitch together
  - Tail value
  - Raw events in the head
  - Queries in the head
Topology 1/2

Queries
(Key, ts)

distinct query heads
[key, distinct(round(ts, hop))]

Events
(Key, ts, payload)

GroupBy
[key, round(ts, hop), agg(payload) as hop]

join on key
(key, query_heads, hops)

join on key
(key, query_head, window_tail)
Toplogy 2/2

Queries
(Key, ts)

Events
(Key, ts, payload)

join on key
key, query_head: window_tail

group-by query heads
key, round(ts, hop): [ts]

distinct query heads
key, round(ts, hop): [(ts, payload)]

Join tails with queries & events
key, query_head: (window_tail, [query_ts], [event_ts, payload])

Put the window together
key, query_head: [query, window]
Topology 1/2

Queries
(Key, ts)

distinct query heads
[key, distinct(round(ts, hop))]}

Events
(Key, ts, payload)

GroupBy
[key, round(ts, hop), agg(payload) as hop]

join on key
(key, query_heads, hops)

join on key
(key, query_head, window_tail)
Topology 2/2

**Queries**

(Key, ts)

**Events**

(Key, ts, payload)

- Join on key: `key, query_head: window_tail`
- Group-by query heads: `key, round(ts, hop): [ts]`
- Distinct query heads: `key, round(ts, hop): [(ts, payload)]`

Join tails with queries & events:

`key, query_head: (window_tail, [query_ts], [event_ts, payload])`

Put the window together:

`key, query_head: [query, window]`
Window tiling - final

- Trade-off
  - Moving too much data
  - Evenly distributing work across machines
• Pig’s perf page
• VLDB
  • anything that has “groupjoin” on it.
• sketches
  • Yahoo datasketches library
    • cardinality estimation - CPC sketch
    • frequent items
  • Bloom filters
Opinions

- MPP compute - trino, clickhouse etc., traditional OLAP
  - Don’t scale
- RDD lacks “stream one side of the join into the other WHILE aggregating”
- OLAP / MPP is actually streaming
- Not new / flink / beam / tf

<table>
<thead>
<tr>
<th>Interactive Analytics</th>
<th>BI</th>
<th>Metrics</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>Monadic/DataFrame/RDD</td>
<td>Page Rank</td>
<td>Social Hash</td>
</tr>
<tr>
<td>Micro Batch</td>
<td>MPP</td>
<td>Graph</td>
<td>ML</td>
</tr>
<tr>
<td>Uni-directional/DAG</td>
<td>Iterative</td>
<td>Streaming</td>
<td></td>
</tr>
</tbody>
</table>
Appendix - Tree Tiling

def generateTiles(left: Int, right: Int, tileConsumer: (Int, Int) => Unit): Int = {
  // find m, i such that
  // (m + 1) * (2 power i) < left <= m * (2 power i) <= right < (m + 1) * (2 power i)
  val powerOfTwo = 1 << (31 - Integer.numberOfLeadingZeros(left ^ right))
  val splitPoint = (right/powerOfTwo) * powerOfTwo
  // tiles on the left side
  var leftDistance = splitPoint - left
  var rightBoundary = splitPoint
  while(leftDistance > 0) {
    val maxPower = Integer.highestOneBit(leftDistance)
    tileConsumer(rightBoundary - maxPower, rightBoundary)
    rightBoundary -= maxPower
    leftDistance -= maxPower
  }
  // tiles on the right side
  var rightDistance = right - splitPoint
  var leftBoundary = splitPoint
  while(rightDistance > 0) {
    val maxPower = Integer.highestOneBit(rightDistance)
    tileConsumer(leftBoundary, leftBoundary + maxPower)
    leftBoundary += maxPower
    rightDistance -= maxPower
  }
  splitPoint
}