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

Cristian
Figueroa

Haozhen
Ding

Pengyu
Hou

Vamsee
Yarlagadda

Varant
Zanoyan

Sophie
Wang

Atul
Kale

Jack
Song

Haichun
Chen

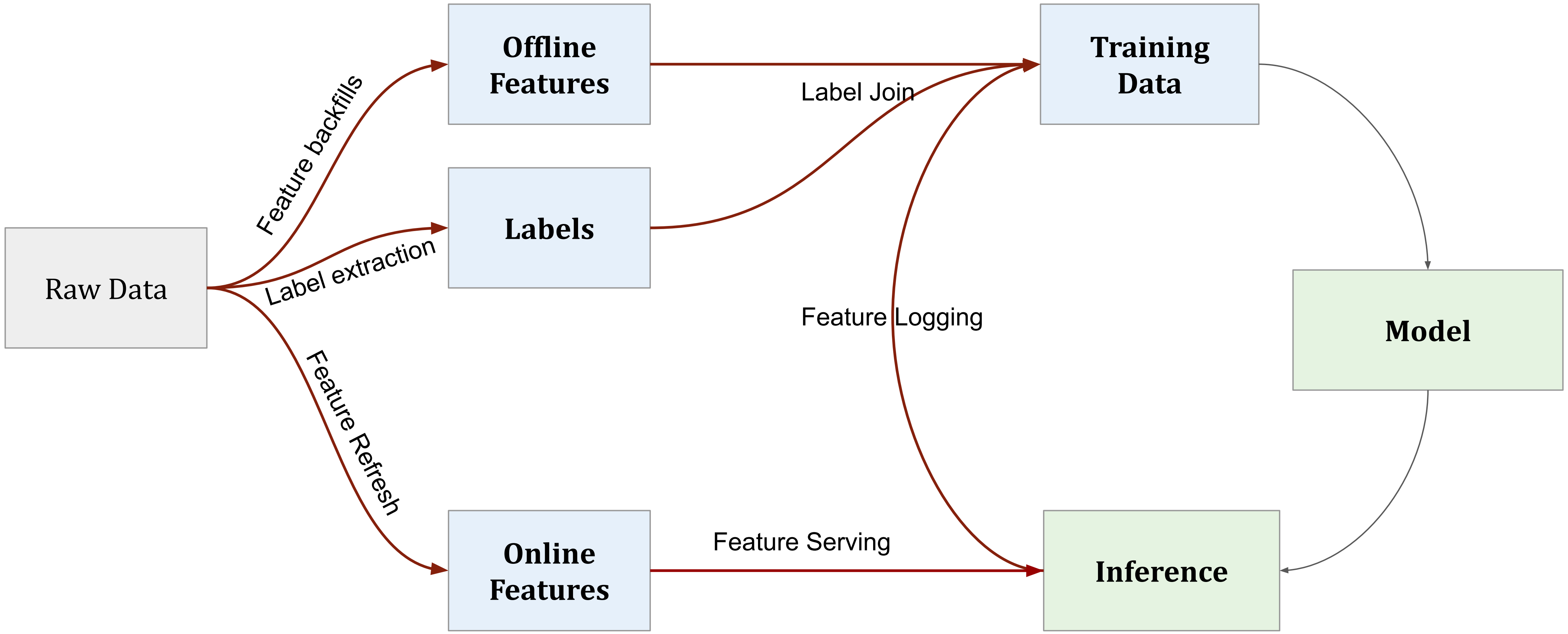
Nikhil
Simha

Agenda

What’s a feature platform?

How to use it?

Machine learning flywheel



Goals - management

Unified API

Feature Lifecycle

- Authoring & Release

Feature observability

- Training data quality
- Realtime feature drift
- Online-offline consistency

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 training data ranges -> lot of waiting
- New features need to be 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 Inference

Not for report generation use-cases b

- Spark vs Clickhouse/Druid
- Static usage is fine

Requirements

?

Kafka
Event Store

Hive (optional)
Batch-Catalog

Spark
Compute Engine

KV Store
Bring-Your-Own

Airflow
Scheduler
or B-Y-O

Offline - problem statement (item recommendation)

user_id	timestamp
alice	2021-09-30 5:24
bob	2021-10-15 9:18
carl	2021-11-21 7:44

- view_count_5h
  - From view stream
- avg_rating_90d
  - From ratings db table

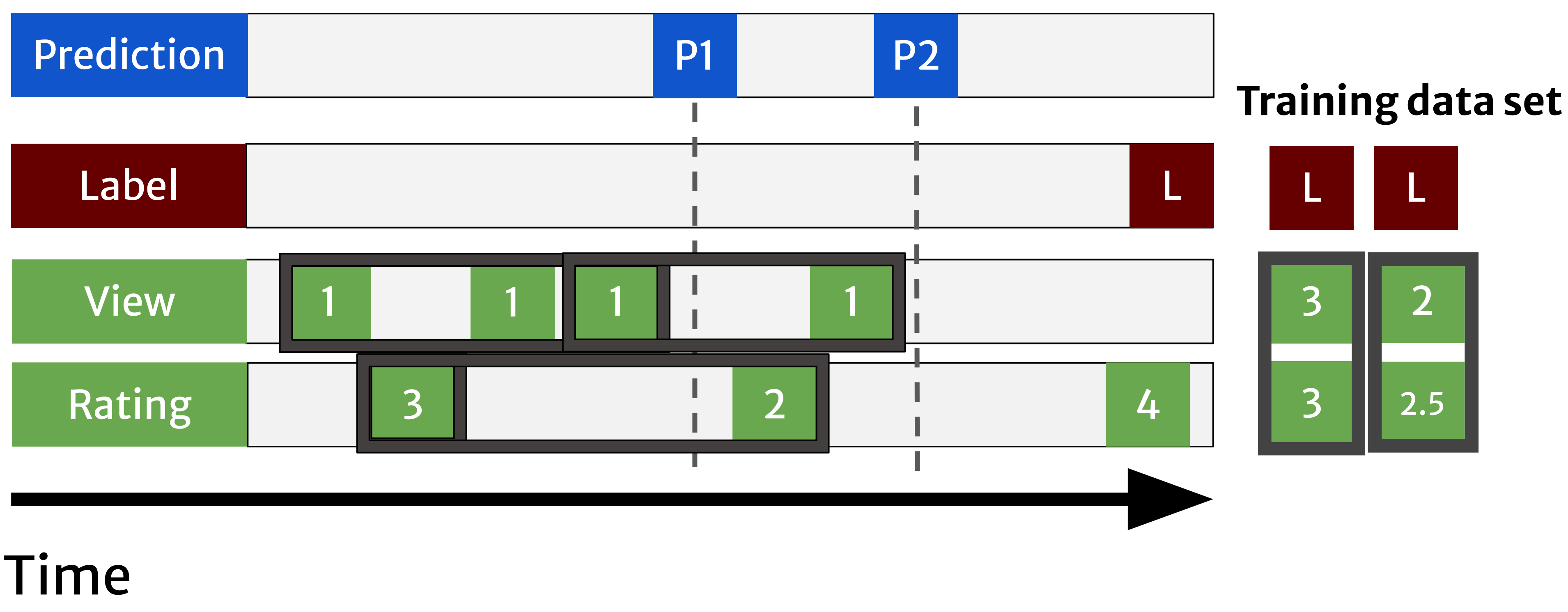
code

Offline - problem statement

user_id	timestamp	views_count_5h	avg_rating_90d
alice	2021-09-30 5:24	10	3.7
bob	2021-10-15 9:18	7	4.5
carl	2021-11-21 7:44	35	2.1

Online - problem statement

user_id	timestamp	views_count_5h	avg_rating_90d
alice	2021-09-30 5:24	10	3.7
bob	2021-10-15 9:18	7	4.5
carl	2021-11-21 7:44	35	2.1



Prediction
P1
P2
Label
L
View
1
1
1
1
Rating
3
2
4
Time
Training data set
L
L
3
3
2
2.5

Examples – E-Commerce platform

<u>Count</u> of <u>Item views</u> of a <u>user</u> in the <u>last 5 hours</u> – from a <u>item view stream</u>

<u>Average</u> <u>rating</u> of an <u>item</u> in the <u>last 90 days</u> – from a <u>ratings table</u>

Count / Average – Aggregation operations

Item Views/Rating – Aggregation Inputs

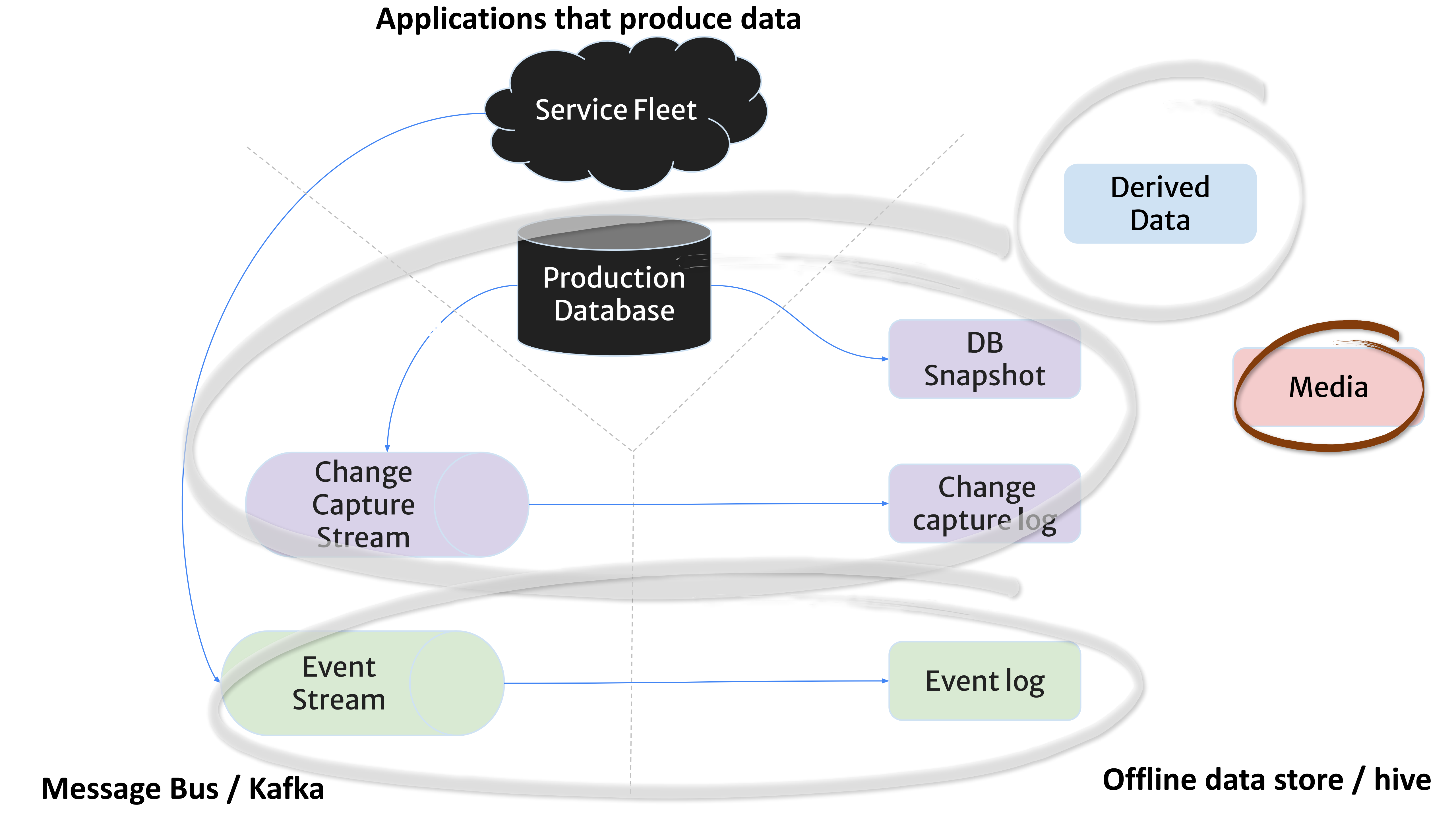
User/item – Aggregation Key

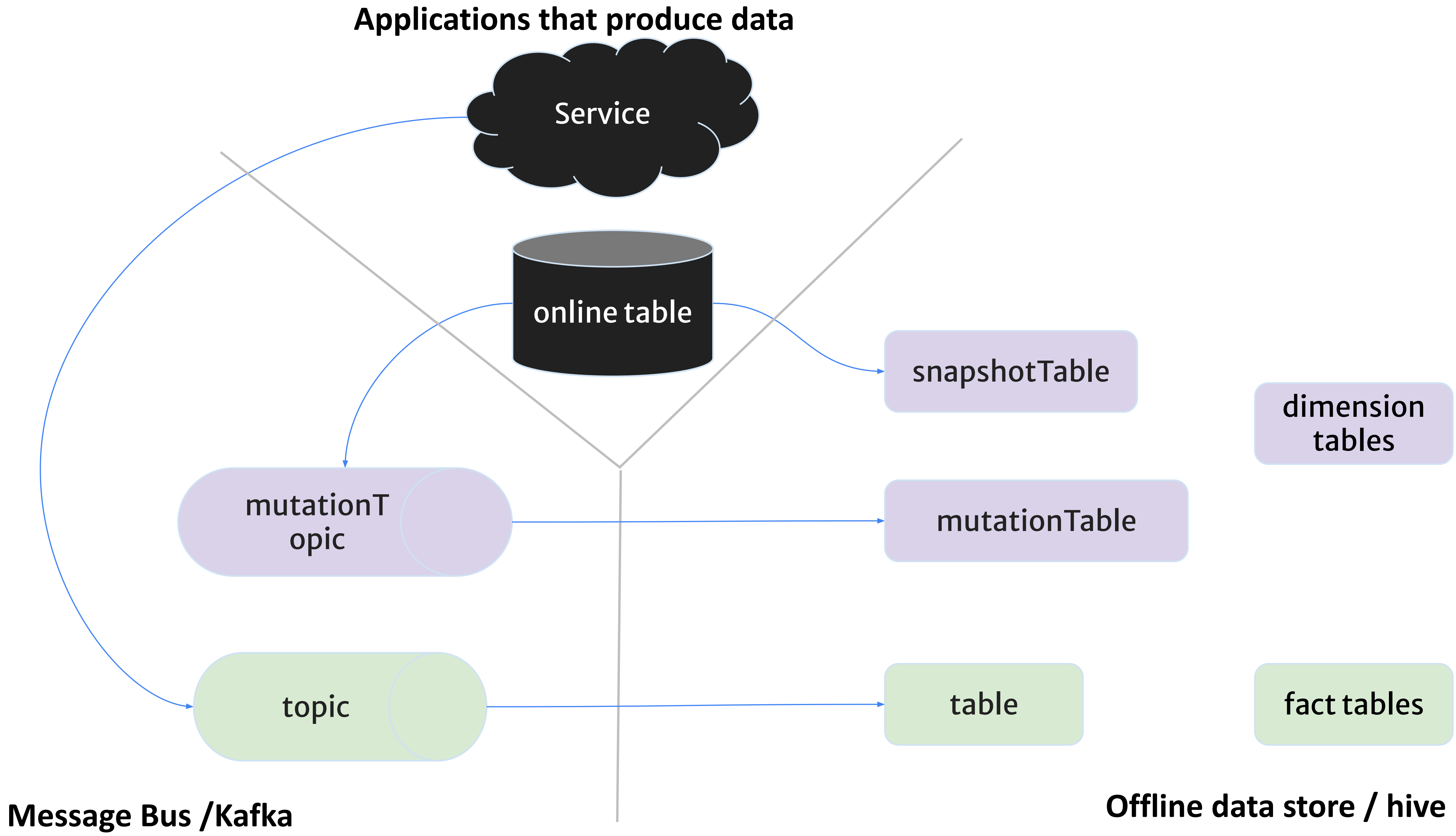
Last X days – Aggregation Window

Ratings Table/ Item View Stream – Data Source

Accuracy - Real-time or Daily

Data Sources


Applications that produce data
Service Fleet
Production Database
Derived Data
DB Snapshot
Media
Change Capture Stream
Change capture log
Event Stream
Event log
Message Bus / Kafka
Offline data store / hive


Applications that produce data
Service
online table
snapshotTable
dimension tables
mutationTopic
mutationTable
topic
table
fact tables
Message Bus /Kafka
Offline data store / hive

Sources - Events

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

Events - item views

item	user	timestamp	date
a	user_1	10:30 am	9/20
b	user_1	1:11 pm	9/20
a	user_2	3:45 pm	9/20

item	user	timestamp	date
c	user_3	9:15 am	9/21
a	user_2	5:31 pm	9/21

...

`item_views` - date Partitioned hive table

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

Entities - item reviews

item	user	review	updated_at	ds
a	user_1	good	09/10 09:03	9/20
b	user_2	bad	09/20 17:15	9/20
d	user_3	okay	09/05 13:21	9/20

item	user	review	updated_at	ds
a	user_1	good	09/10 09:03	9/21
b	user_2	**okay**	09/21 09:03	9/21
c	user_1	bad	09/21 15:31	9/21

`item_reviews` - snapshotTable (ds partitioned)

is_before	item	user	review	updated_at	mutation_ts	date
true	b	user_2	bad	09/20 17:15	09/21 09:03	9/21
false	b	user_2	okay	09/21 09:03	09/21 09:03	9/21
true	d	user_3	okay	09/05 13:21	09/21 15:55	9/21
false	c	user_1	baad	09/21 15:31	09/21 15:31	9/21

`item_reviews_mutations` - mutationsTable (ds partitioned)

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

Cumulative Events

item	user	timestamp	date
a	user_1	1/1 10:30 am	9/20
b	user_1	3/21 1:11 pm	9/20
a	user_2	9/20 3:45 pm	9/20

item	user	timestamp	date
a	user_1	1/1 10:30 am	9/20
b	user_1	3/21 1:11 pm	9/20
a	user_2	9/20 3:45 pm	9/20
c	user_3	9/21 9:15 am	9/21
a	user_2	9/21 4:21 pm	9/21

`item_views` - date Partitioned hive table

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!

Code Examples

Examples

Count of Item views of a user in the last 5 hours – from a item view stream

```
view_features = GroupBy(
        sources=[
            EventSource(
                table="user_activity.user_views_table",
                topic="user_views_stream",
                query=query.Query(
                    selects={
                        "view": "if(context['activity_type'] = 'item_view', 1 , 0)",
                    },
                    wheres=["user != null"]))
        ],
        keys=["user", "item"],
        aggregations=[
            Aggregation(
              operation=Operation.COUNT,
              windows=[Window(length=5, timeUnit=TimeUnit.HOURS)]),
        ])
```

Examples

Average rating of an item in the last 90 days – from a ratings table

```
ratings_features = GroupBy(
    sources=[
        EntitySource(
            snapshotTable="item_info.ratings_snapshots_table",
            mutationsTable="item_info.ratings_mutations_table",
            mutationsTopic="ratings_mutations_topic",
            query=query.Query(
                selects={
                    "rating": "CAST(rating as DOUBLE)",
                }))
    ],
    keys=["item"],
    aggregations=[Aggregation(
      operation=Operation.AVERAGE,
      windows=[Window(length=90, timeUnit=TimeUnit.DAYS)]),
    ])
```

Examples - Join

Putting it all together

```
item_rec_features = Join(
    left=EventSource(
        table="user_activity.view_purchases",
        query=query.Query(
            start_partition='2021-06-30'
        )
    ),
    # keys are automatically mapped from left to right_parts
    right_parts=[
        JoinPart(groupBy=view_features),
        JoinPart(groupBy=ratings_features)
    ]
)
```

API

Spark SQL Expression language

Time is first class

- Source Types
- Windows
- PITC joins

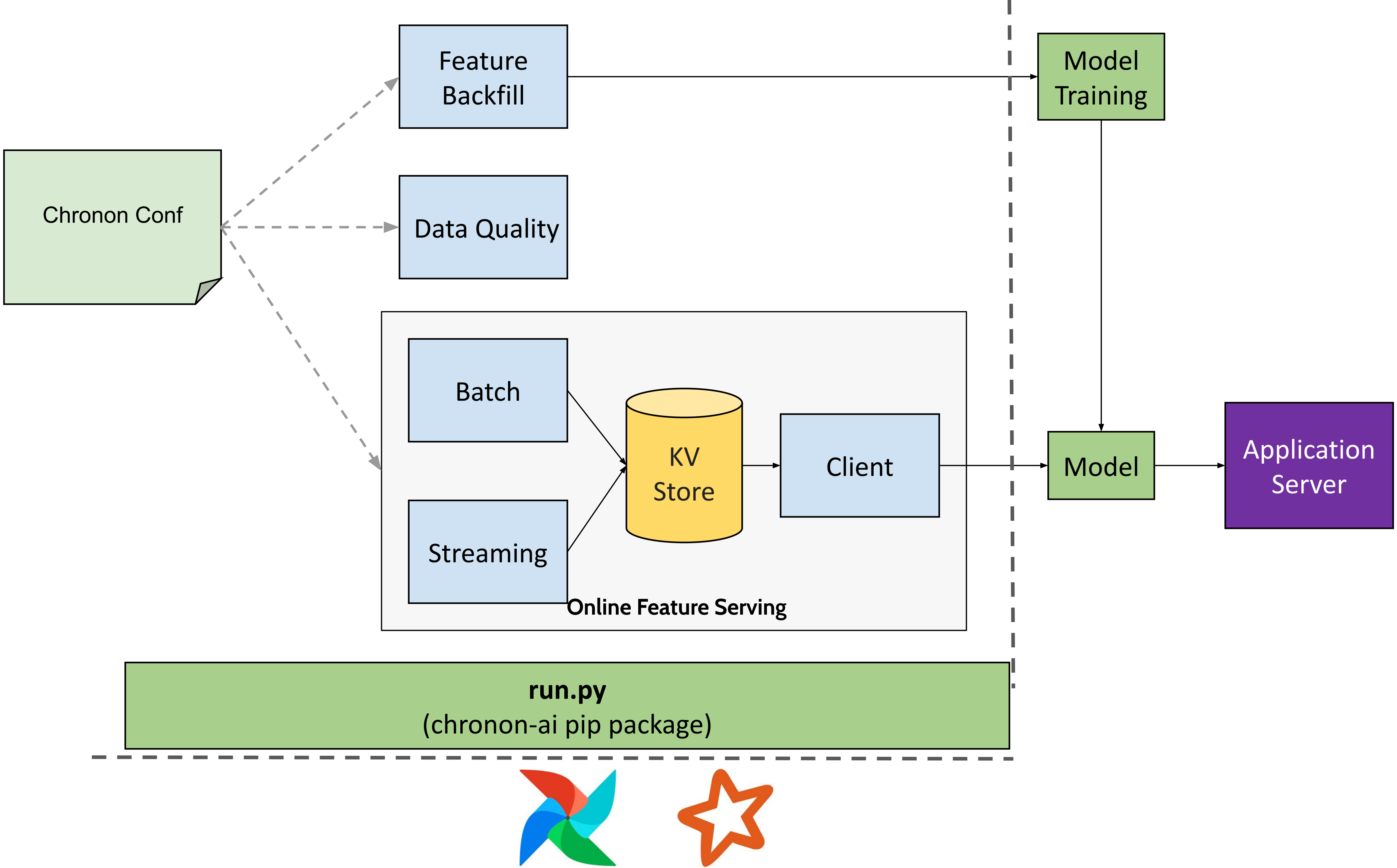
Aggregations

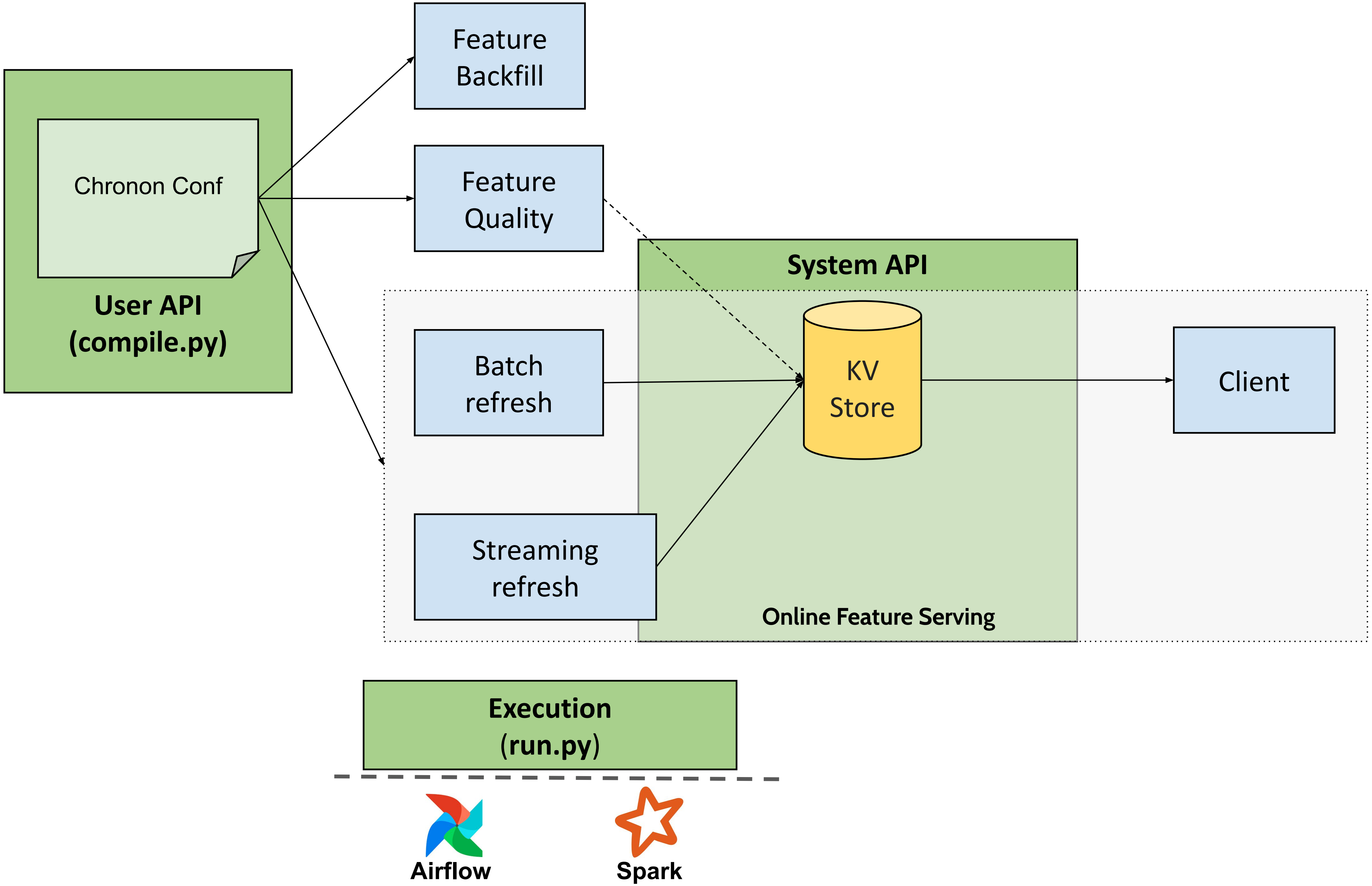
- Commutative
- Bucketing
- Auto Flattening

Composability of python

Architecture

Very High Level


Feature Backfill
Model Training
Chronon Conf
Data Quality
Batch
KV Store
Client
Model
Application Server
Streaming
Online Feature Serving
run.py
(chronon-ai pip package)


Feature Backfill
Feature Quality
Chronon Conf
User API (compile.py)
System API
Batch refresh
KV Store
Client
Streaming refresh
Online Feature Serving
Execution (run.py)
Airflow
Spark

GroupBy

Concepts - GroupBy

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

Concepts - Aggregations

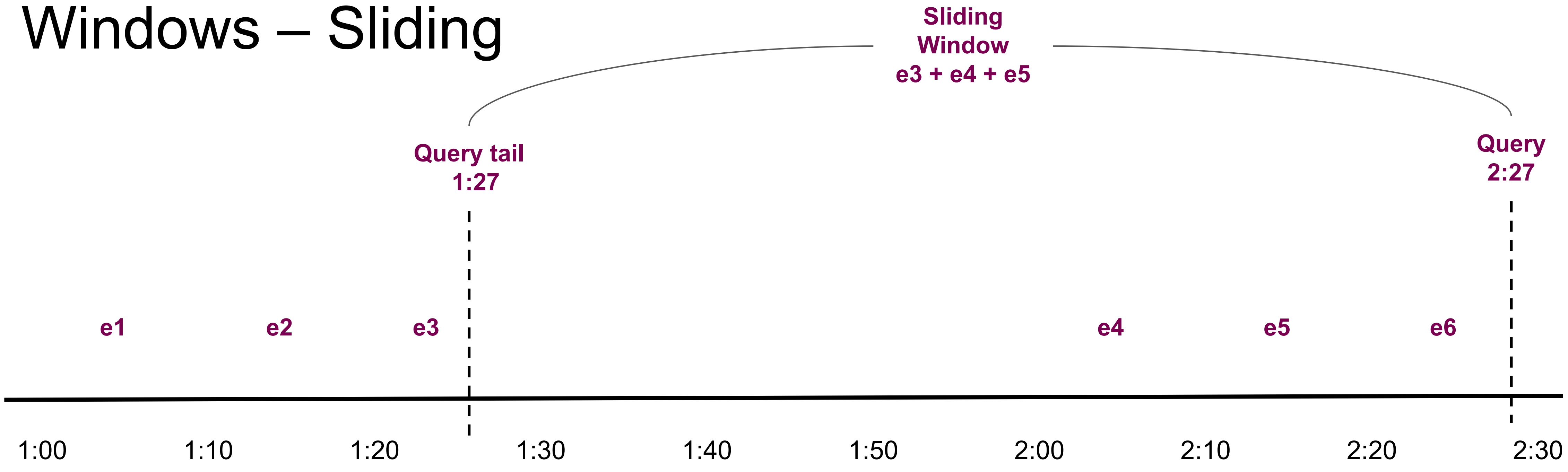
SUM, COUNT, AVG, VARIANCE, MIN, MAX, TOP_K, BOTTOM_K, FIRST, LAST, FIRST_K, LAST_K, APPROX_DISTINCT, FREQUENT_ITEMS, HISTOGRAM…, APPROX_PERCENTILES

Commutative and associative - order independent & mergeable

Sometimes reversible - CDC updates

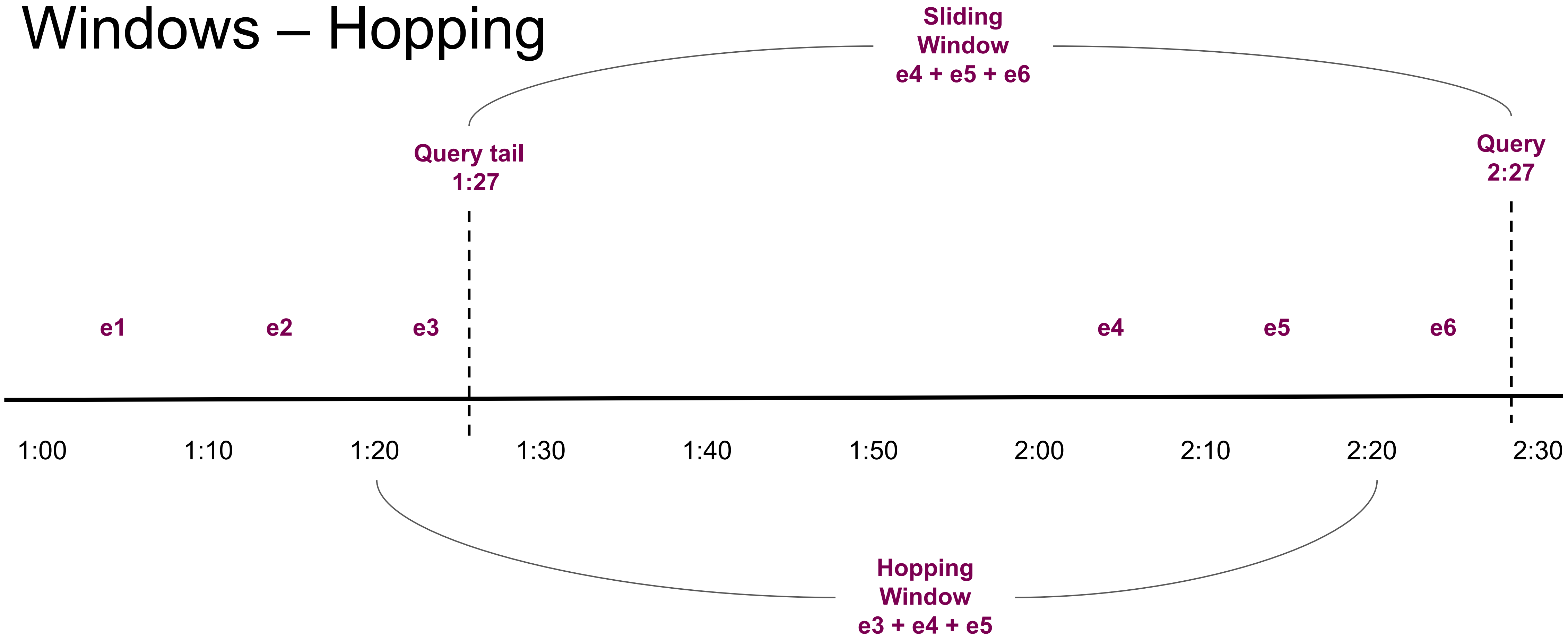
Windows

Windows – Sliding



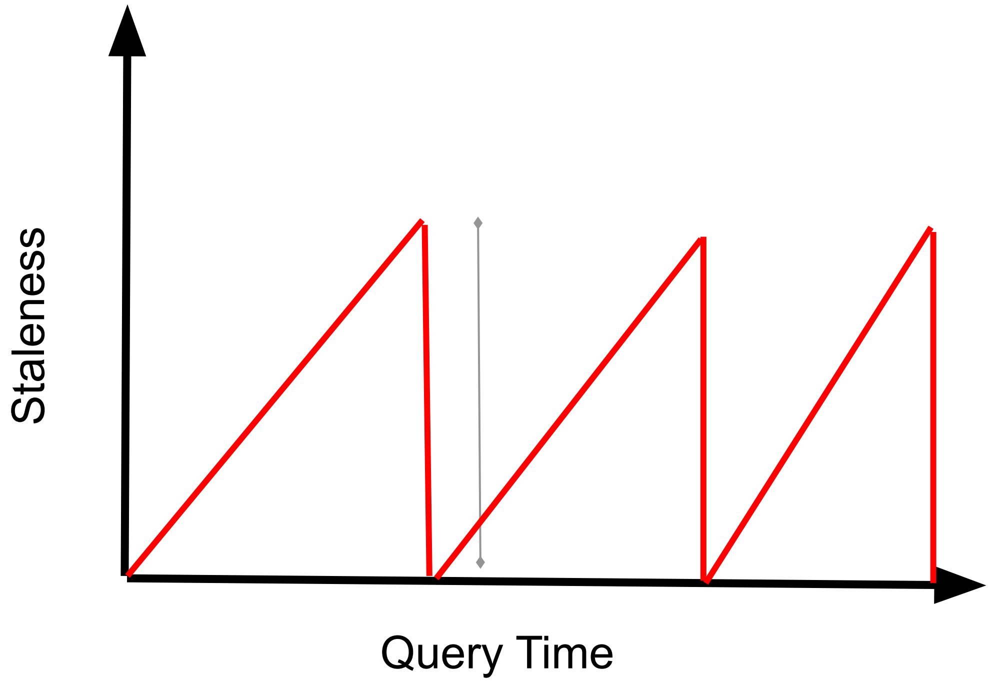
- Freshness
- Memory intensive

Windows – Hopping

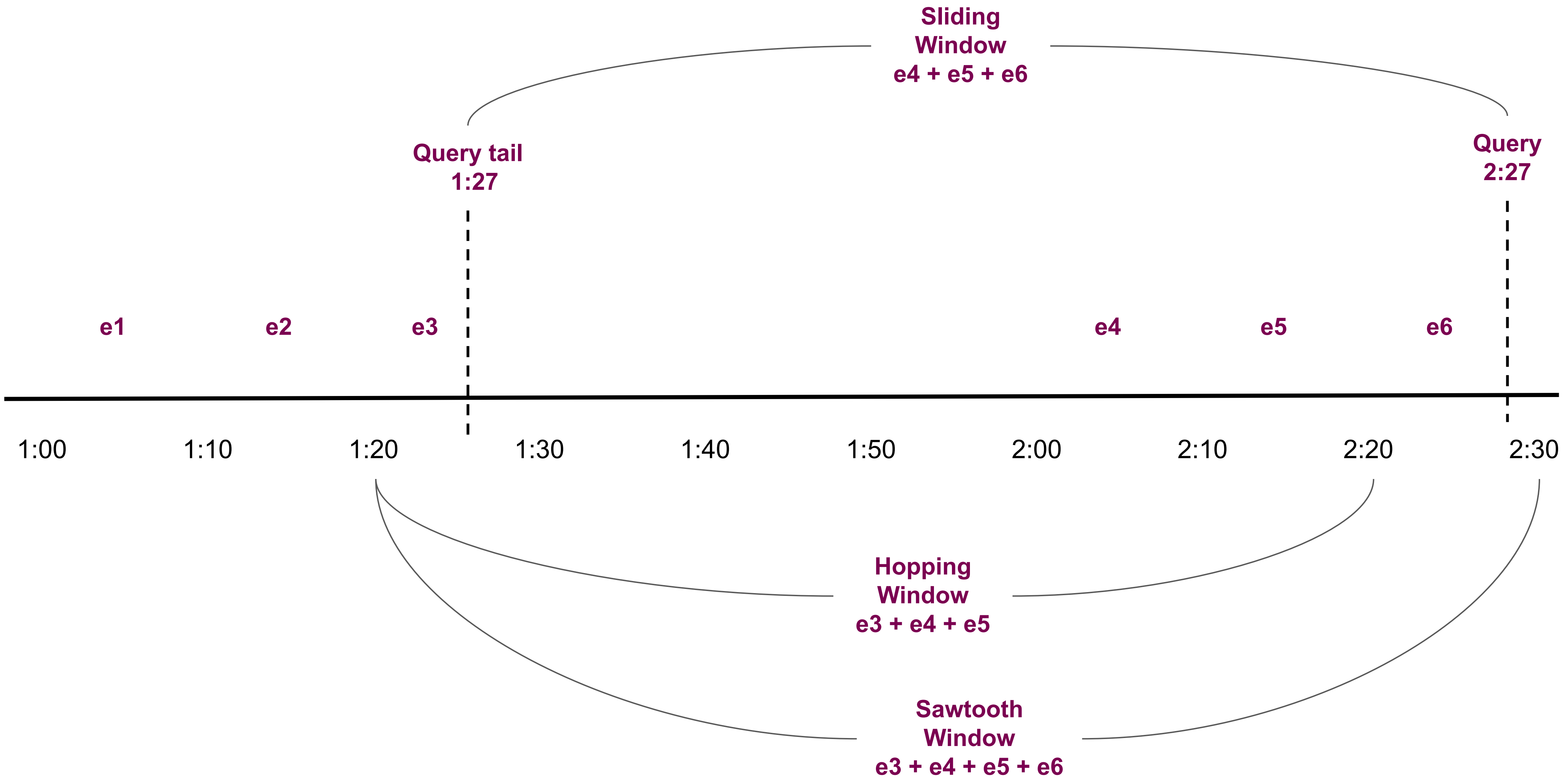


Windows – Hopping

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



Windows – Sawtooth

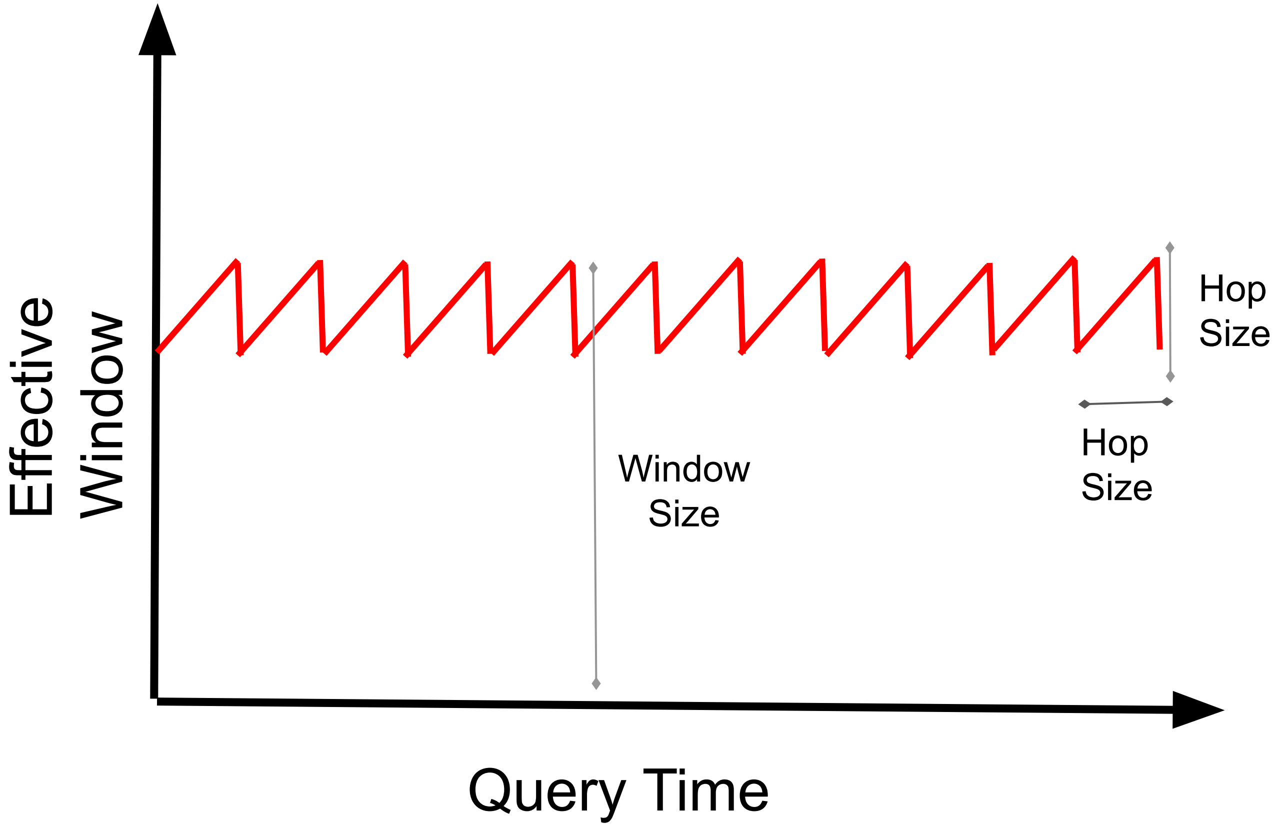


Windows – Sawtooth

- Freshness
  - Writes are taken into account immediately
- Memory
  - Partial aggregates per hop

Windows – Sawtooth

- Catch
    - sum/count vs others
- Consistency



Join

Concepts - Join

user_id	timestamp
alice	2021-09-30 5:24
bob	2021-10-15 9:18
carl	2021-11-21 7:44

- view_count_5h
  - From view stream
- avg_rating_90d
  - From ratings db table

Concepts - Join

user_id	timestamp	views_count_5h	avg_rating_90d
alice	2021-09-30 5:24	10	3.7
bob	2021-10-15 9:18	7	4.5
carl	2021-11-21 7:44	35	2.1

Concepts - Join

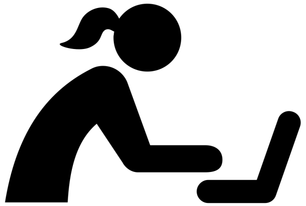
- Join multiple GroupBy-s (feature groups) together
  - Decide to show a particular user a particular item – likelihood to buy
    - X User Features groups
    - Y Item Features
    - Z (User, Item) Features– past interactions
- Gather both Online & Offline
- Left & rights
  - labelled data + timestamped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

Workflow

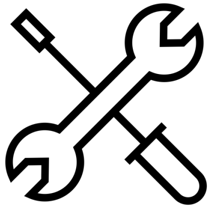
User workflow

explore.py

Lineage search



Author



compile.py

Code Validation

Release

run.py

Pipeline gen

Testing

Explore

- explore.py: Keyword lineage search
    - raw data > feature group > feature set > model
- compile.py:
    - validation & change management
- run.py:
    - data pipeline generation & testing

Compile

- Python is powerful
- Change Management
- 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

Run.py - scheduling

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

user_id	timestamp	views_count_7d
alice	2021-09-30 5:24	10
bob	2021-10-15 9:18	7
carl	2021-11-21 7:44	35

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

```
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?

```
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.
```

- sort + cursors
- Complexity? n*log(n)
- Distribute friendly?
- Use of subtraction - doesn’t work

  for max, min etc.
- Even better?

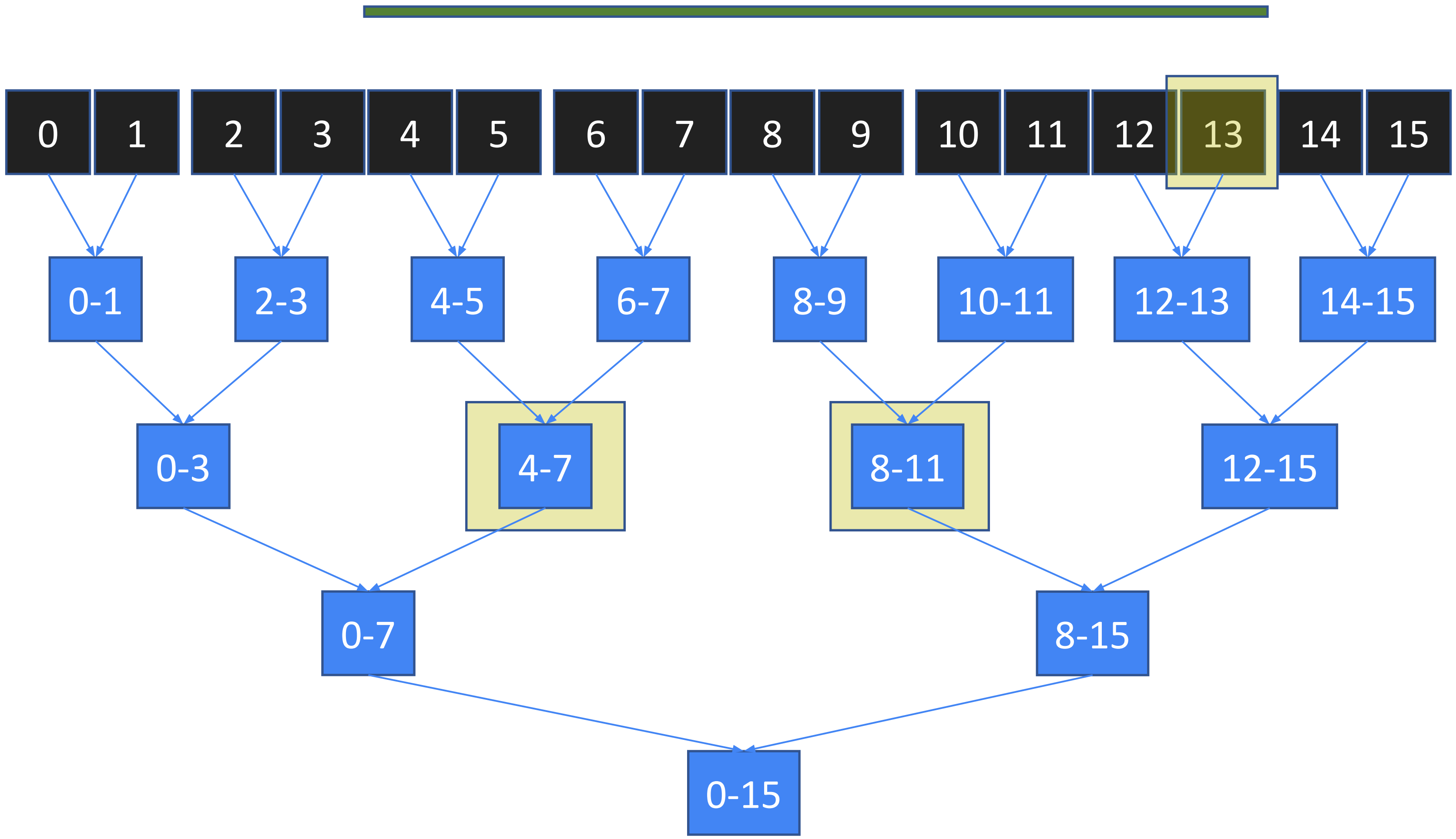
Some important observations

- 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



Window tiling

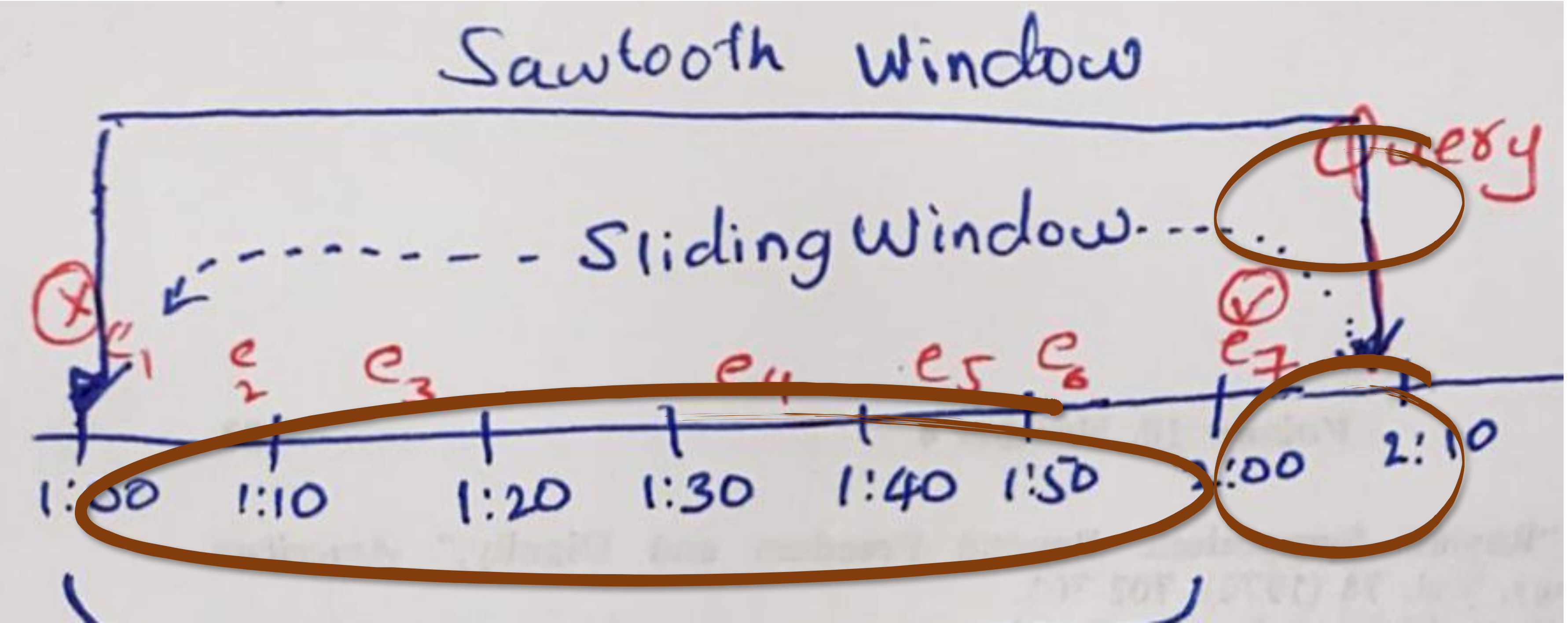
- Hopping tail is common across all queries that fall into the head!
- The idea is to compute tails and heads separately.

Window tiling

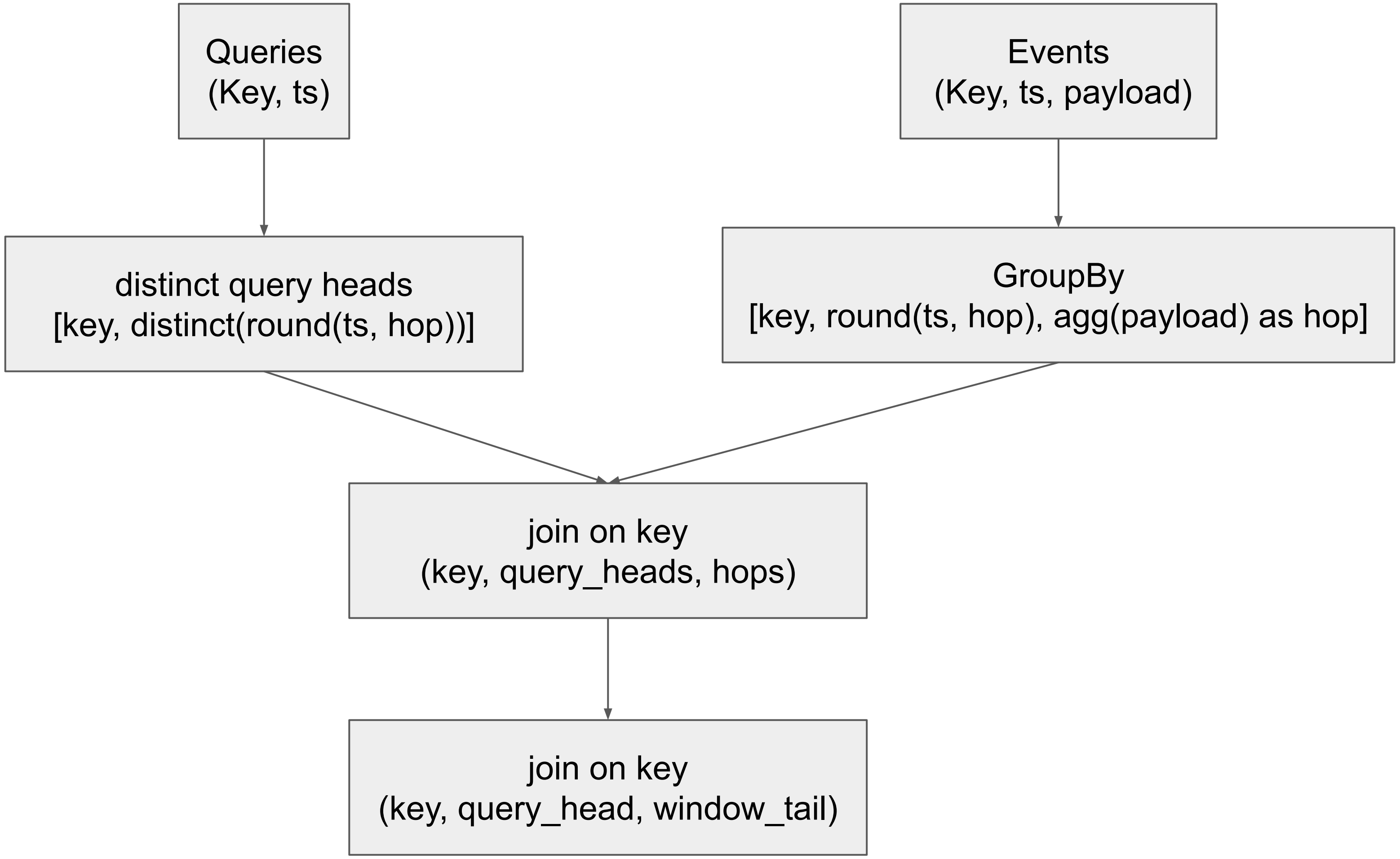
- What if queries don’t fit in memory?
    - Tiling can’t be dynamic(query dependent)
- Hops?
    - Let’s examine window semantics

Window tiling

- We need to stitch together
  - Tail value
  - Raw events in the head
  - Queries in the head

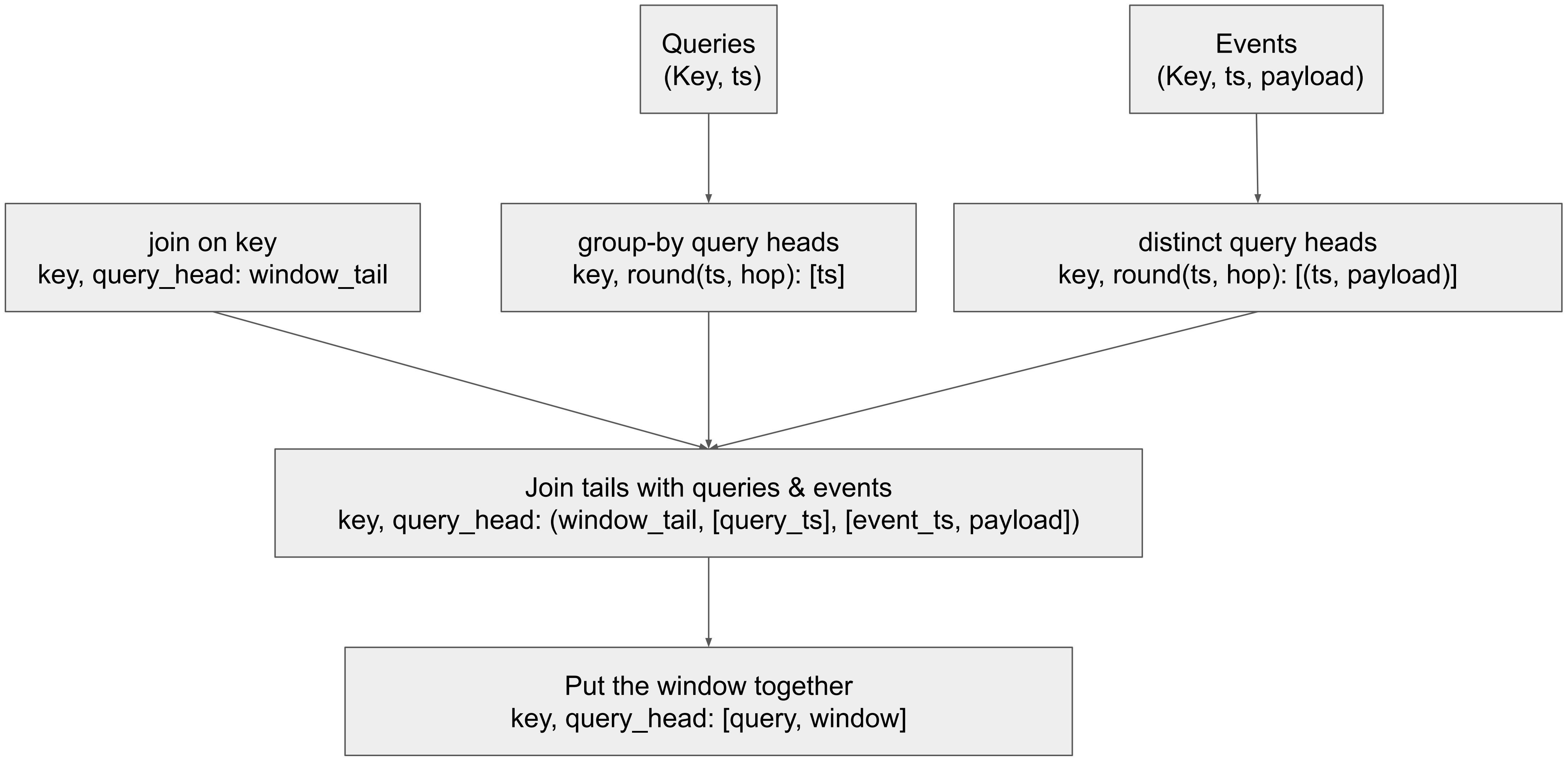


Topology 1/2

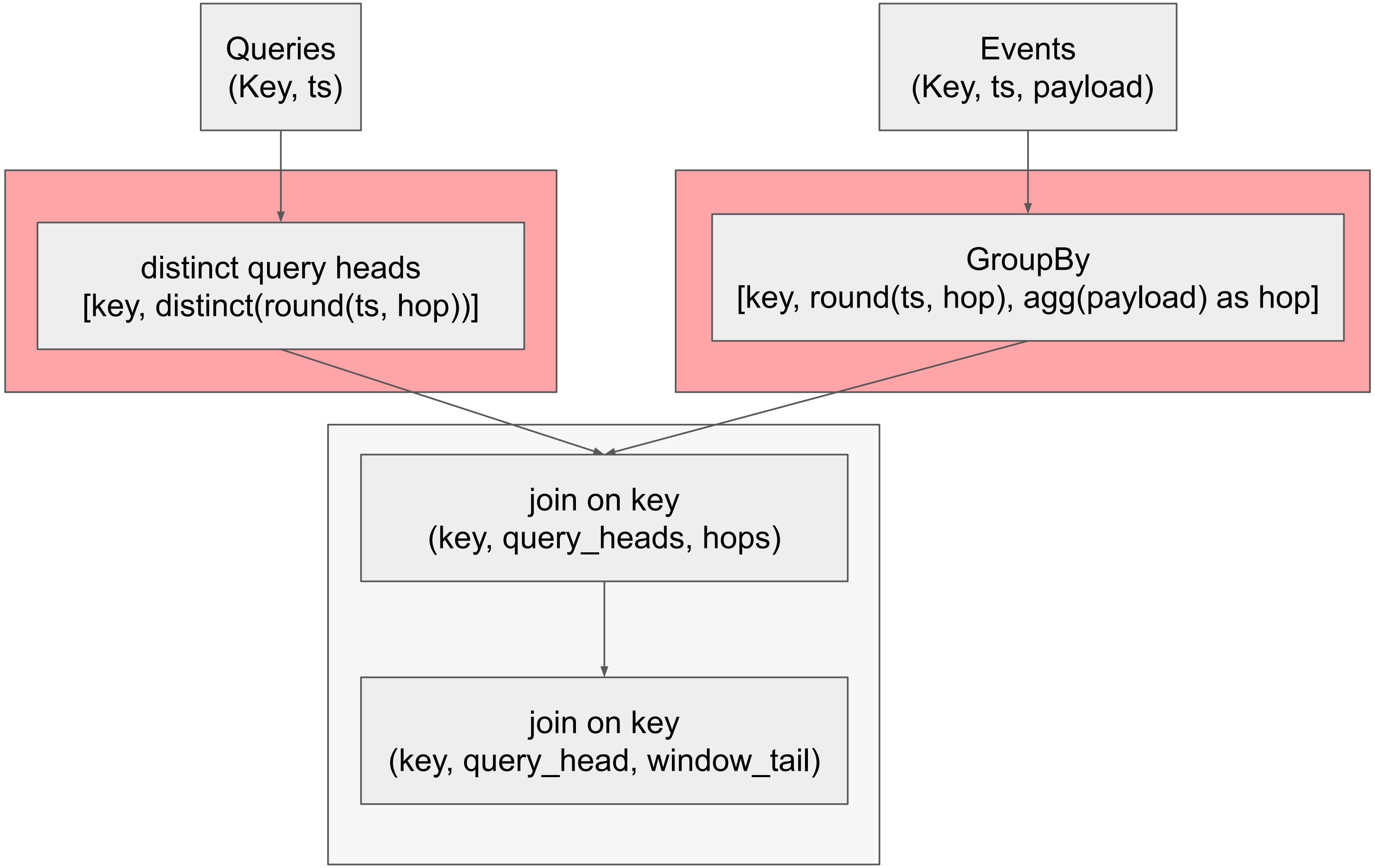


Queries
(Key, ts)
Events
(Key, ts, payload)
distinct query heads
[key, distinct(round(ts, hop))]
GroupBy
[key, round(ts, hop), agg(payload) as hop]
join on key
(key, query_heads, hops)
join on key
(key, query_head, window_tail)

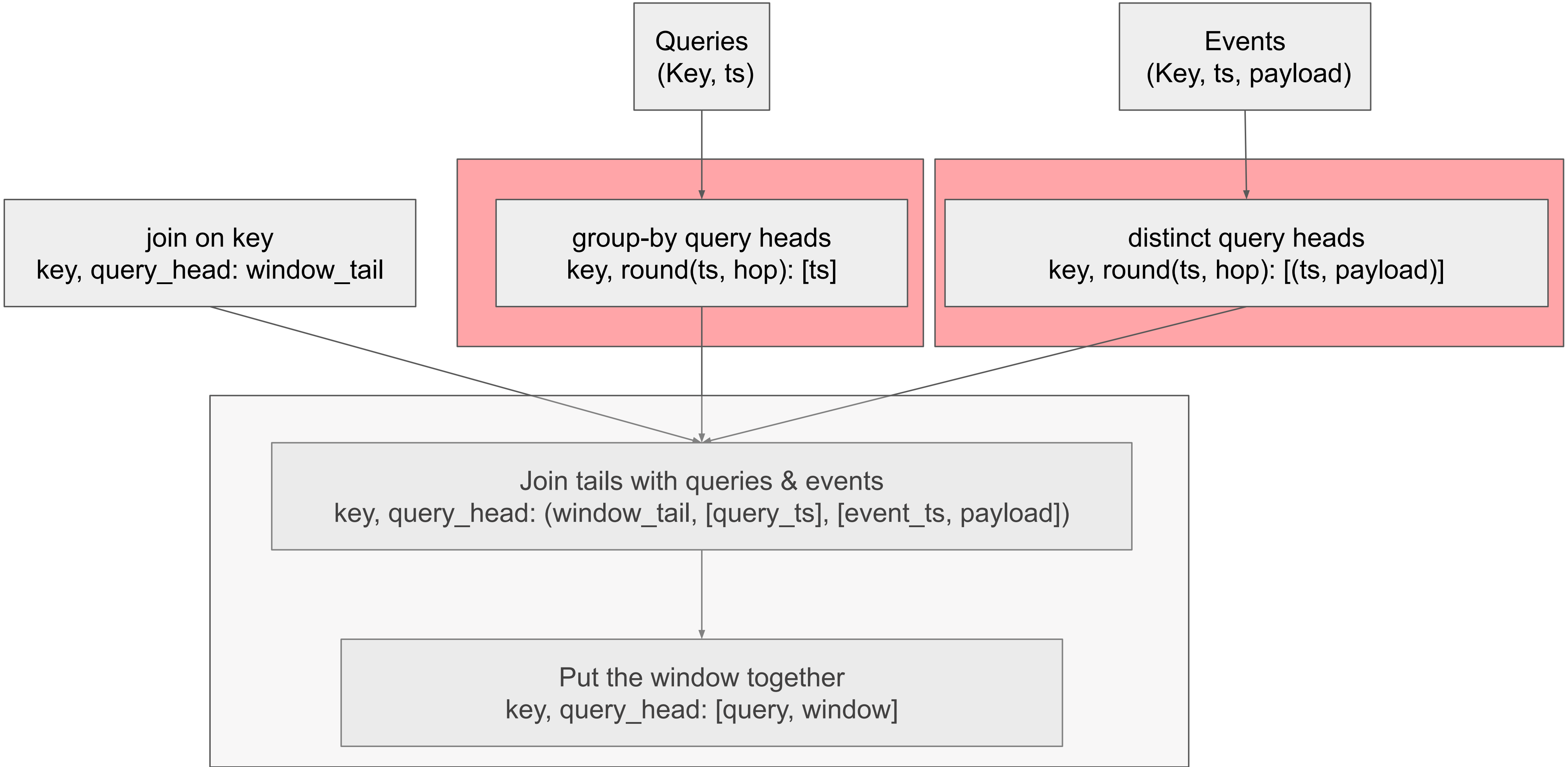
Topology 2/2


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]

Topology 1/2


Queries
(Key, ts)
Events
(Key, ts, payload)
distinct query heads
[key, distinct(round(ts, hop))]
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

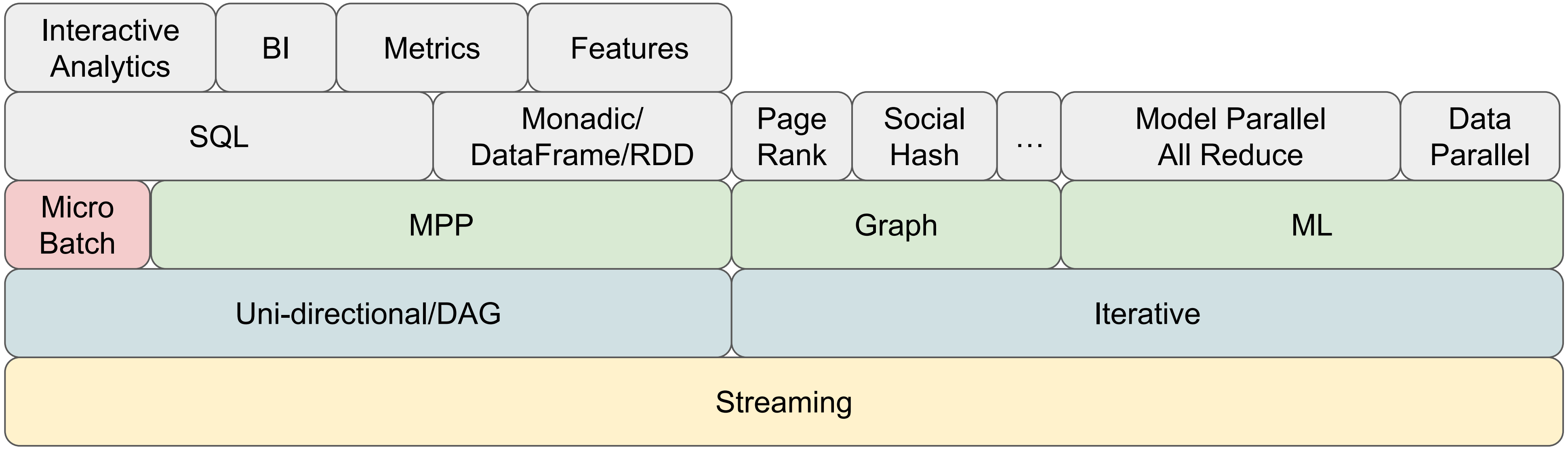
- Trade-off
  - Moving too much data
  - Evenly distributing work across machines

Resources

- Pig's perf page
- VLDB
    - anything that has "groupjoin" on it.
- sketches
    - Yahoo datasketches library
        - cardinality estimation - CPC sketch
        - frequent items
    - Bloom filters

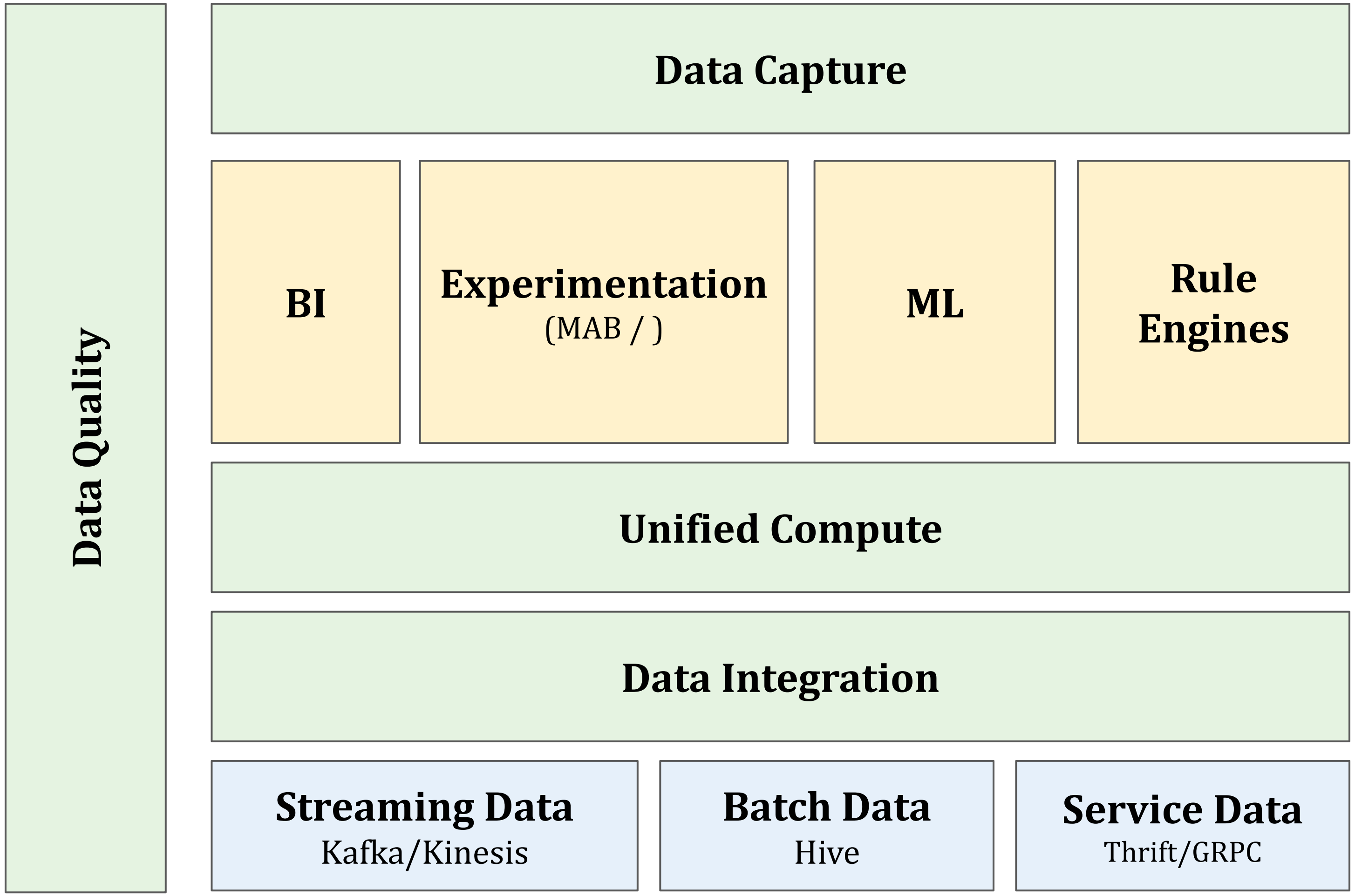
Opinions

- 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



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
 }
```



Chronon

- Unified view of data from three contexts
  - Batch / Hive / Service
  - Only possible if warehouse has conventions
  - Data integration is one of the hardest and most underrated problems.
- Unified compute
  - Scanning, Projections, Filtering, Join, Aggregations
  - **Aggregation is where big data becomes small / meaningful data.**
  - Time is global ordering. All existing OLAP systems don’t model time. Warehouses have a strong time convention.
  - Realtime compute is essential for ML. RT with regression is better than transformers.
  - Without modeling time it is very hard to make computation tractable - specially in RT case.
- Quality
  - Input data, output data, realtime actioning
    - converate, correlation, distributions
  - Compute in real-time & batch