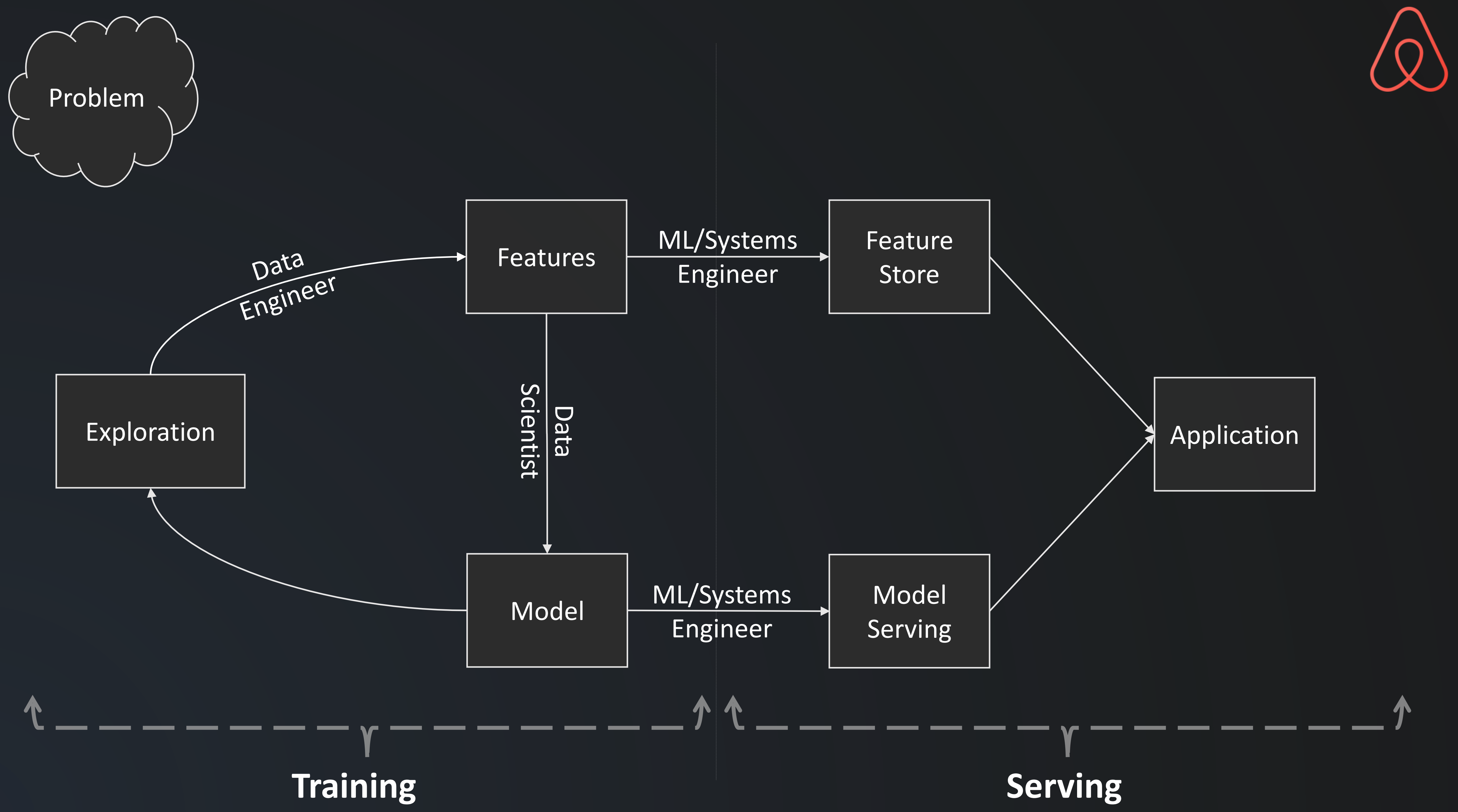
Zipline

Airbnb’s Feature Engineering Framework

Nikhil Simha
nikhil.simha@airbnb.com


Problem
Data Engineer
Features
ML/Systems Engineer
Feature Store
Exploration
Data Scientist
Application
Model
ML/Systems Engineer
Model Serving
Training
Serving

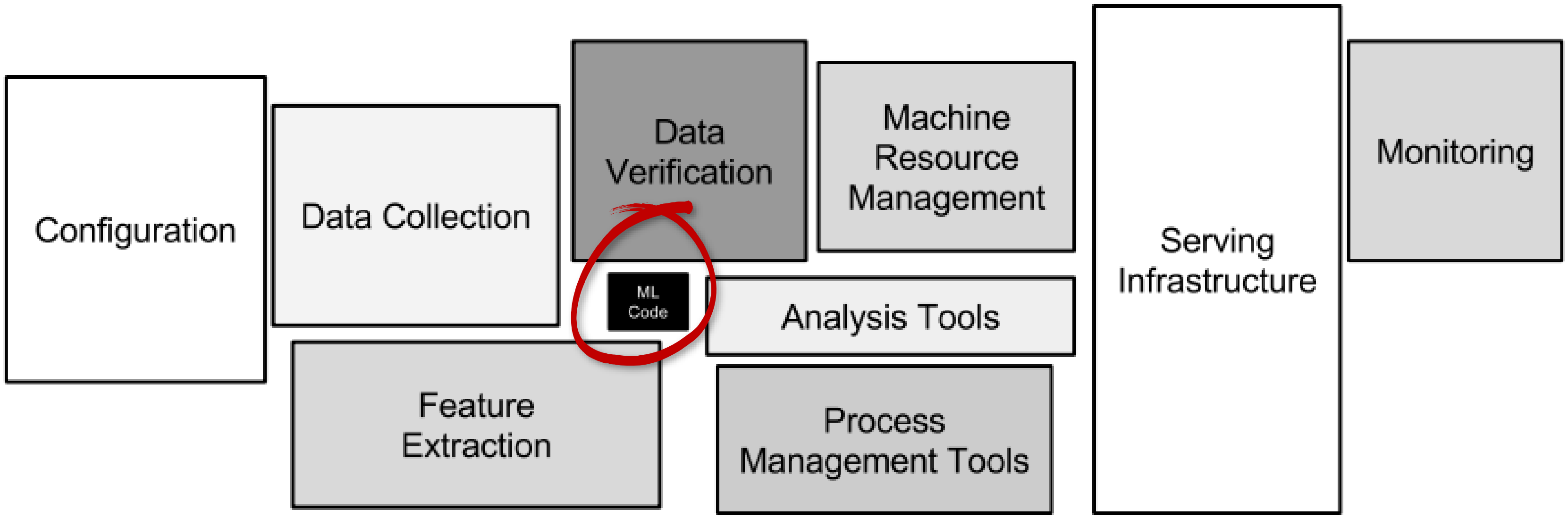


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

“We recognize that a mature system might end up being (at most)
machine learning code and (at least) 95% glue code” – Sculley, NIPS

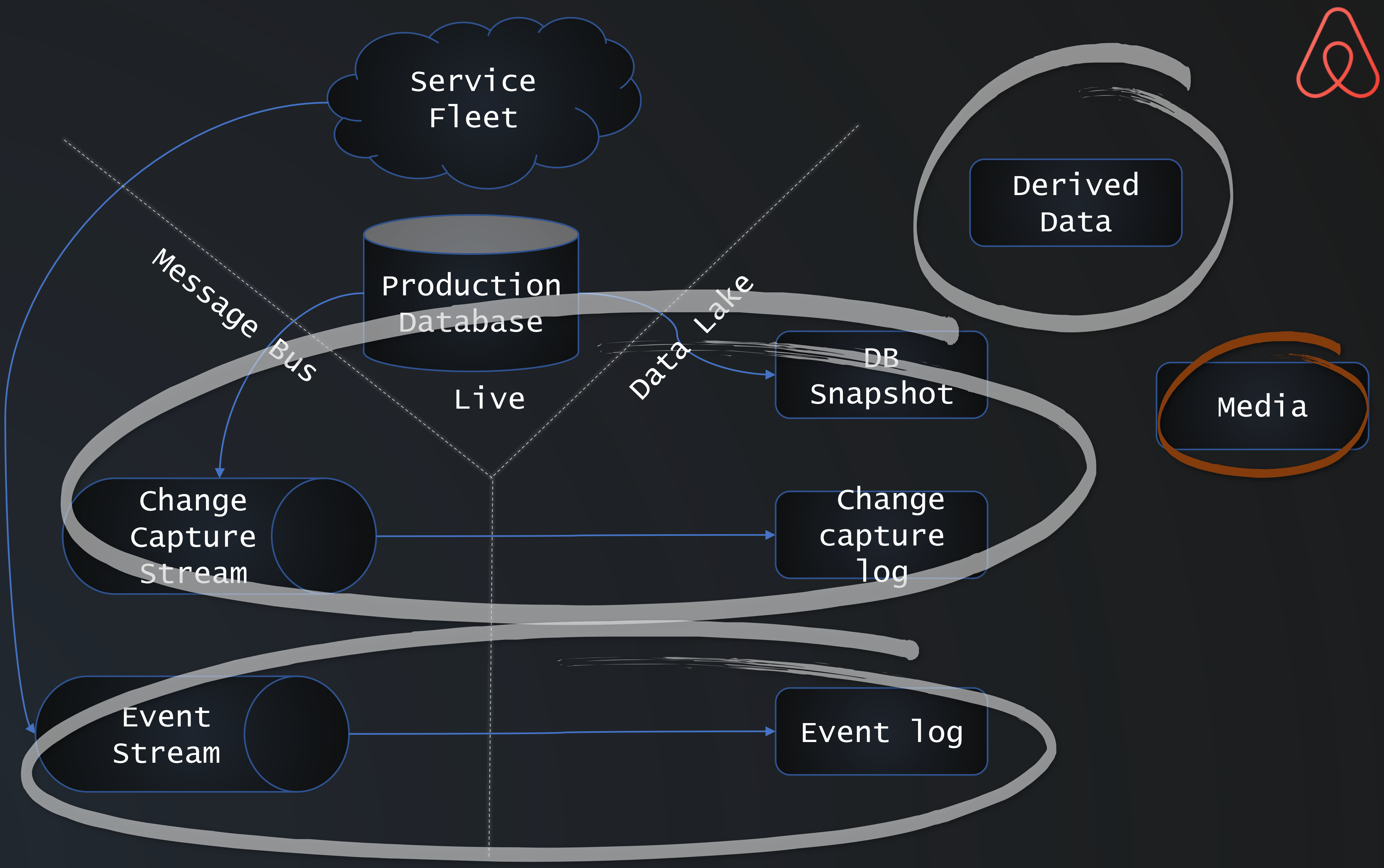
Goals

- Enable DS
- Best Practices
- Efficient
- Save time

Feature Engineering

- 60 – 70%
- Good data with okay/simple model
- Continuously arriving data

Your typical Data Warehouse



Service Fleet
Production Database
Message Bus
Live
Data Lake
Derived Data
Media
DB Snapshot
Change Capture Stream
Change capture log
Event Stream
Event log

An example

- Restaurant recommendations
  - Total visits to restaurants of the same cuisine last month
  - Average rating of the restaurant last y
  - They are all aggregations

An example

- Predict likelihood of you liking a particular Indian restaurant
    - Total visits to restaurants of the same cuisine last month
        - COUNT(visit_id) GROUP BY cuisine WINDOW 1month
        - checkin_stream(kafka) + checkin_log (hive)
    - Average rating of the restaurant
        - AVG(rating) GROUP BY restaurant_id WINDOW 1yr
        - ratings_db_snapshot(hive) +
          ratings_db_mutations(kafka/debezium)

Feature Set Example

```
ratings_features = GroupBy(
    sources=EventSource(
        event_stream="restaurant_check_in_stream",
        event_log_table="core_data.restaurant_check_ins",
        query=Query(  # <- Stateless scanning = projections + filter
            select=Select(
                restaurant="id_restaurant",
                rating="CAST(rating as DOUBLE) - 2.5",
            ),
        ),
    ),
    keys=["restaurant"],
    aggregations=Aggregations(
        avg_rating=Aggregation(
            documentation="Avg rating for this cuisine",
            operation=AVG,
            inputColumn="rating",
            windows=[Window(length=1, timeUnit=TimeUnit.YEARS)]
        )
    )
)
```

Feature Set Example

```
check_in_source = DBSource(
    snapshot_table="core_data.ratings",
    change_topic="checkin_stream",
    query=Query(
        select=Select(
            user="id_user",
            restaurant="id_restaurant",
            cuisine="restaurant_cuisine",
            rating="rating",
        ),
    ),
)

check_in_features = GroupBy(
    sources=check_in_source,
    keys=["user", "cuisine"],
    aggregations=Aggregations(
        num_checkins=Aggregation(
            documentation="Number of check-ins",
            operation=COUNT,
            windows=[Window(length=30, timeUnit=TimeUnit.DAYS)]
        )
    )
)
```

Feature Set Example

```
feature_set = LeftOuterJoin(
    left=HiveEventSource(
        table="core_data.ratings",
        query=Query(
            select=Select(
                user="id_user",
                restaurant="id_restaurant",
                cuisine="restaurant_cuisine",
                rating="rating",  # ← Label. irrelevant for backfills
            )
        ),
    ),
    right=[check_in_features, ratings_features],
)
```

Aggregations + Temporal Join

Feature Serving for inference

What is the value of these feature aggregates now?

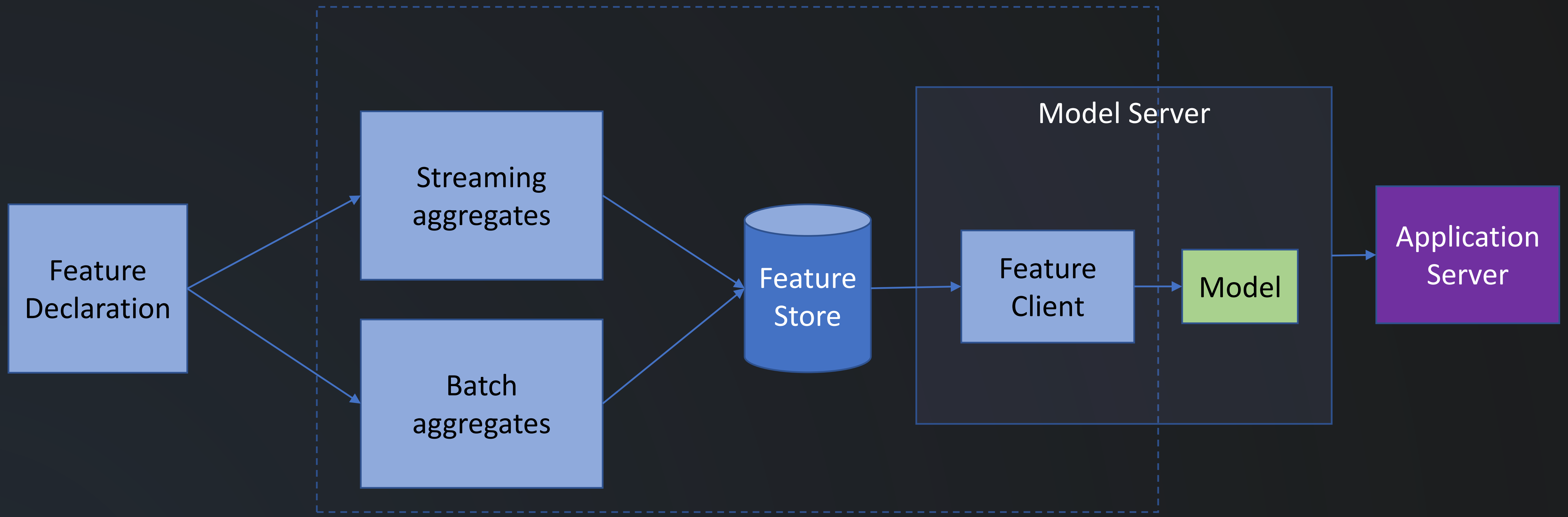
Feature Serving

- Latency
  - Optimized for point queries
- Freshness vs latency
  - Service Events and DB Mutations
- Batch correction

Real-time features

- Fact features: Fact log (hdfs) + Fact Stream (kafka)
  - Service Events
- Dim features: Dim snapshot (hdfs) + Change Stream (kafka)
  - Database snapshots

Serving Architecture



Computed Vs. Logged

- Easier to implement
- Fully consistent
- Horrible iteration time
- Computed + Consistent is our goal

Feature Computation for training

What are the exact feature values at the points-of-interest in history?

Example

Query Log			Aggregated Features	
User	restaurant	timestamp	visits last month	avg rating last year
sarah	Zeni's	2019-09-13 17:31	5	4
eve	La mar	2019-09-14 17:40	20	4
anusha	Chaat	2019-09-15 17:02	6	2

Aggregation Math

Aggregations – SUM

- Commutative: a + b = b + a
- Associative: (a + b) + c = a + (b + c)
- Reversible: (a + b) – a = b
- *Abelian Group*

Aggregations – AVG

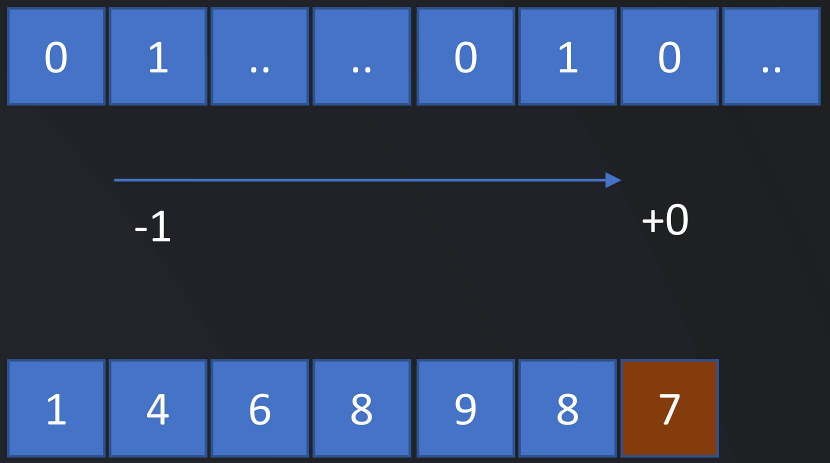
- One not-so-clever trick
  - Operate on “Intermediate Representation” / IR
  - Factors into (sum, count)
  - Finalized by a division: (sum/count)

Aggregations

- Constant memory / Bounded IR
- Two classes of aggregations
  - Sum, Avg, Count etc.,
    - Reversible / Abelian Groups
  - Min, Max, Approx Unique, most sketches etc.,
    - Non-Reversible / Commutative Monoids / Non-Groups

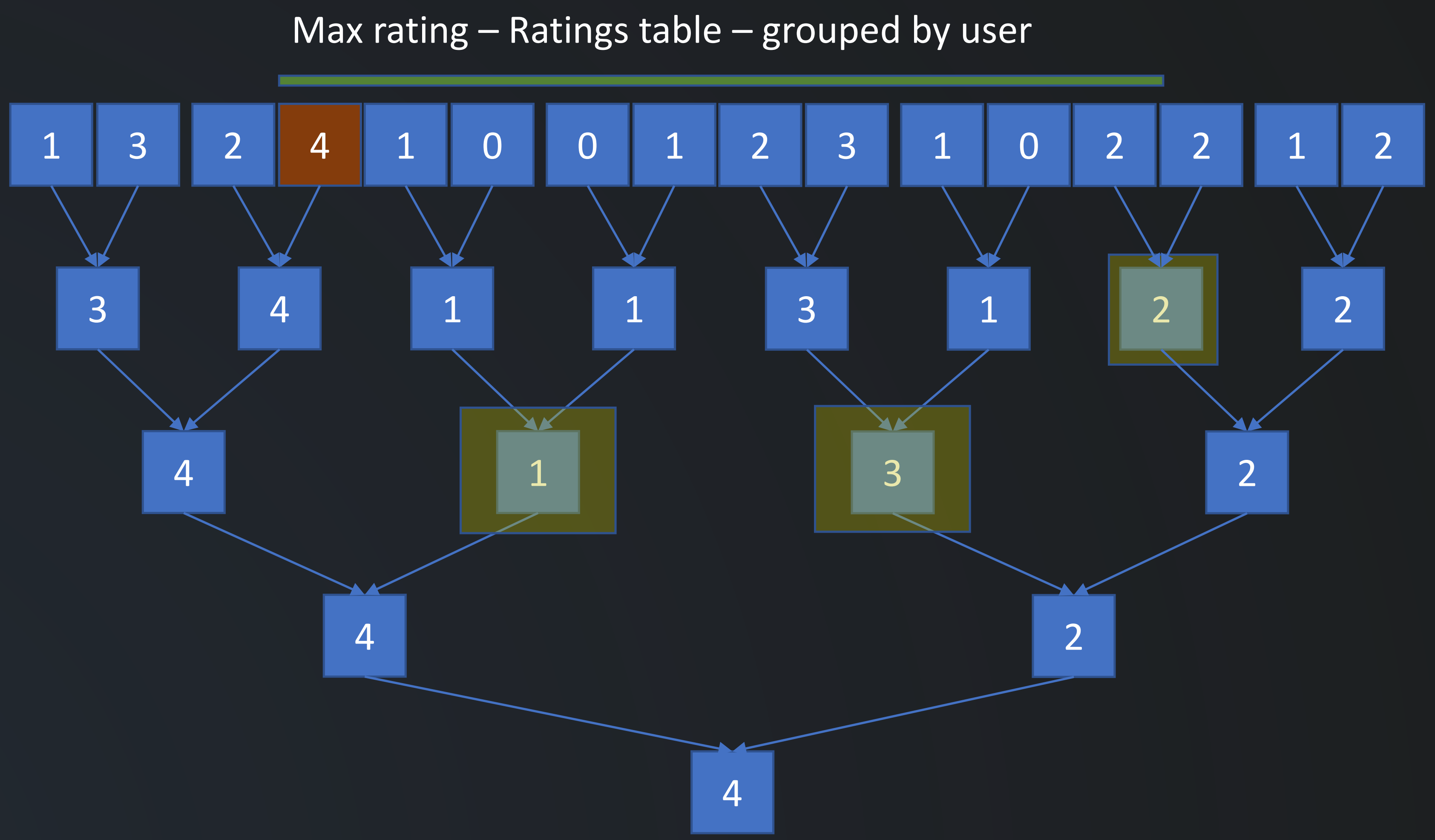
Incremental Windowing – with reversibility

Visits – check-in stream of a user



In the last year

Incremental Windowing – with reversibility



Windowing – w/o reversibility

- Time: O(N^2) vs O(NLogN)
- Space: N vs 2N memory

	Groups	Non-Groups
Un-Windowed	No-Reversal	No-Reversal
Windowed	Reversal	Tree

Reversibility – Unpacking Change data

- Deletion is a reversal
- Update is a delete followed by an insert
- Example:
  - Review is taken down

Example

Query Log			Aggregated Features	
User	restaurant	timestamp	visits last month	avg rating last year
sarah	Zeni's	2019-09-13 17:31	5	4
eve	La mar	2019-09-14 17:40	20	4
anusha	Chaat	2019-09-15 17:02	6	2

Feature Backfill

- Time-series join with aggregations
  - Left   Queries   (Key, timestamp)
  - Right   Events   (Key, timestamp, payload)
  - Output   Features   (Key, timestamp, aggregated)
- Aggregation and join is fused
- Raw data >> query log

Naïve approach

```
result = {}
for (key, query_times, events) in join_result:
    result[key] = [None] * query_times
    for (i, query_time) in enumerate(query_times):
        for event in events:
            if (query_time - window) ⩽ event.time < query_time:
                result[key][i] = update(event.payload, result[key][i])
```

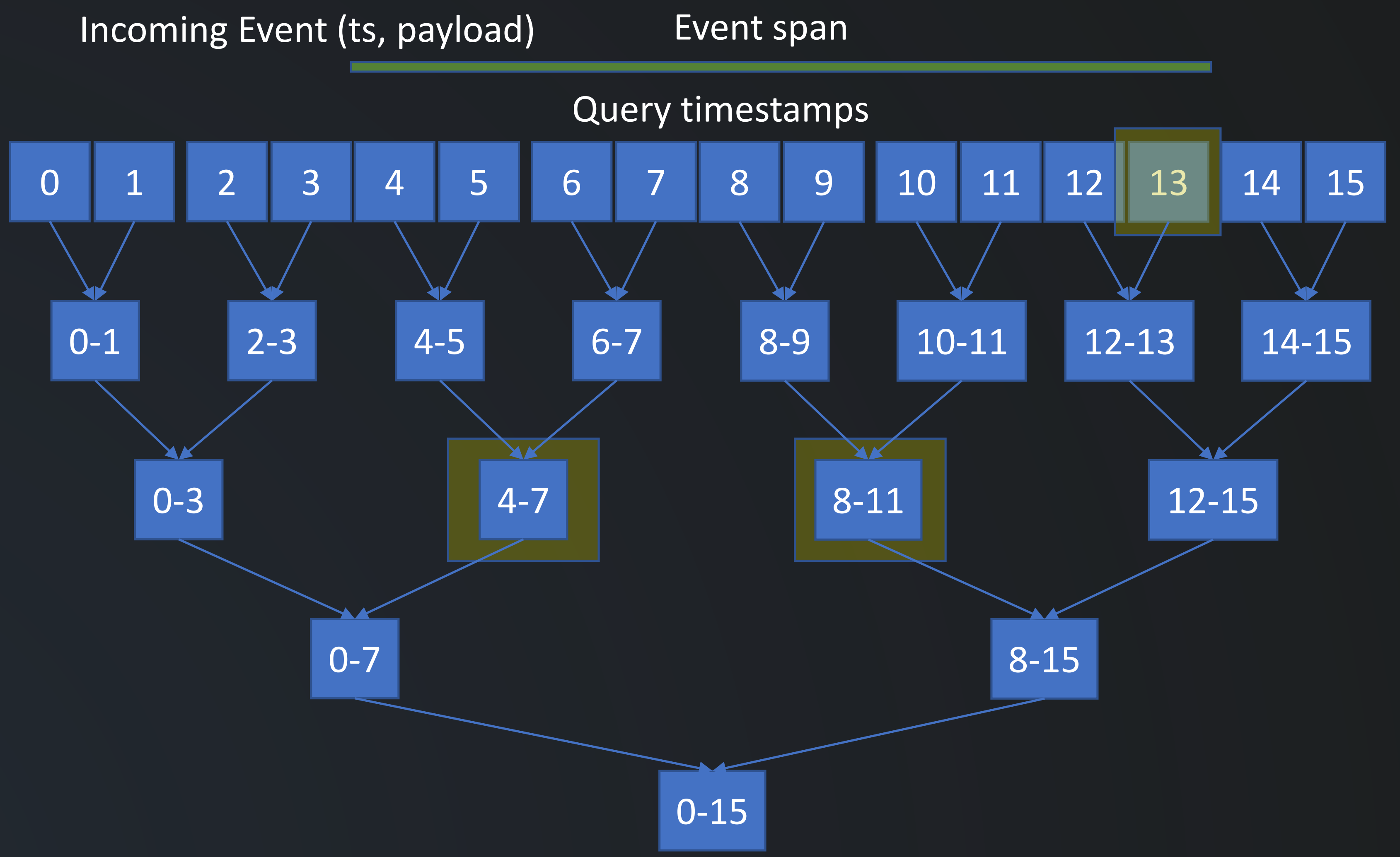
Optimization – 1: Loop ordering

```
result = {}
for (key, query_times, events) in join_result:
    result[key] = [None] * query_times
    for event in events:  # pass through events only once
        # O(queries) search + O(candidates) updates
        for (i, query_time) in enumerate(query_times):
            if (query_time - window) ⩽ event.time < query_time:
                result[key][i] = update(event.payload, result[key][i])
```

Optimization - 2 : Binary search

```
result = {}
for (key, query_times, events) in join_result:
    result[key] = [None] * query_times
    for event in events:
        # O(log-queries) search + O(candidates) updates
        for i in range(bsearch(event.time, query_times),
                       bsearch(event.time + window, query_times)):
            result[key][i] = update(event.payload, result[key][i])
```

Tree Merge



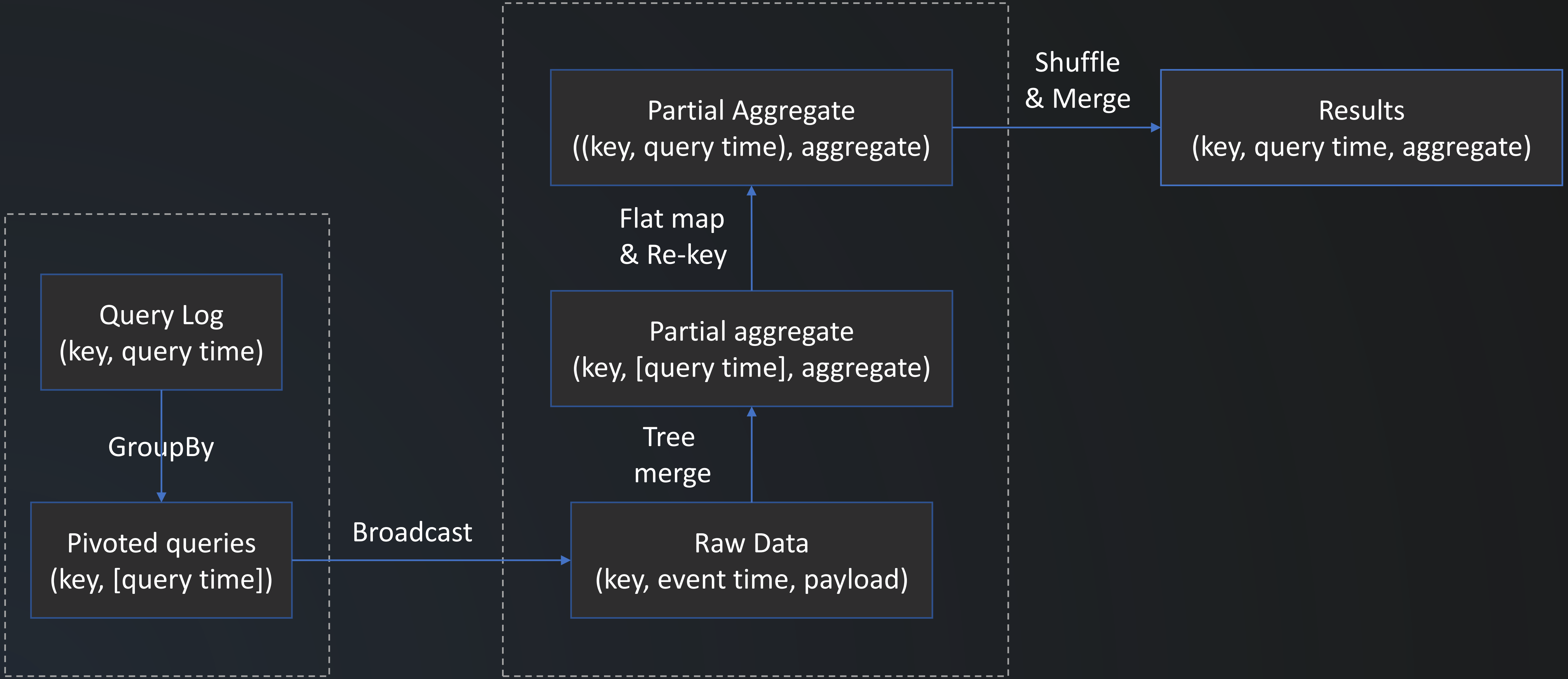
Incoming Event (ts, payload)
Event span
Query timestamps
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0-1
2-3
4-5
6-7
8-9
10-11
12-13
14-15
0-3
4-7
8-11
12-15
0-7
8-15
0-15

Optimization - 3 : Tiling

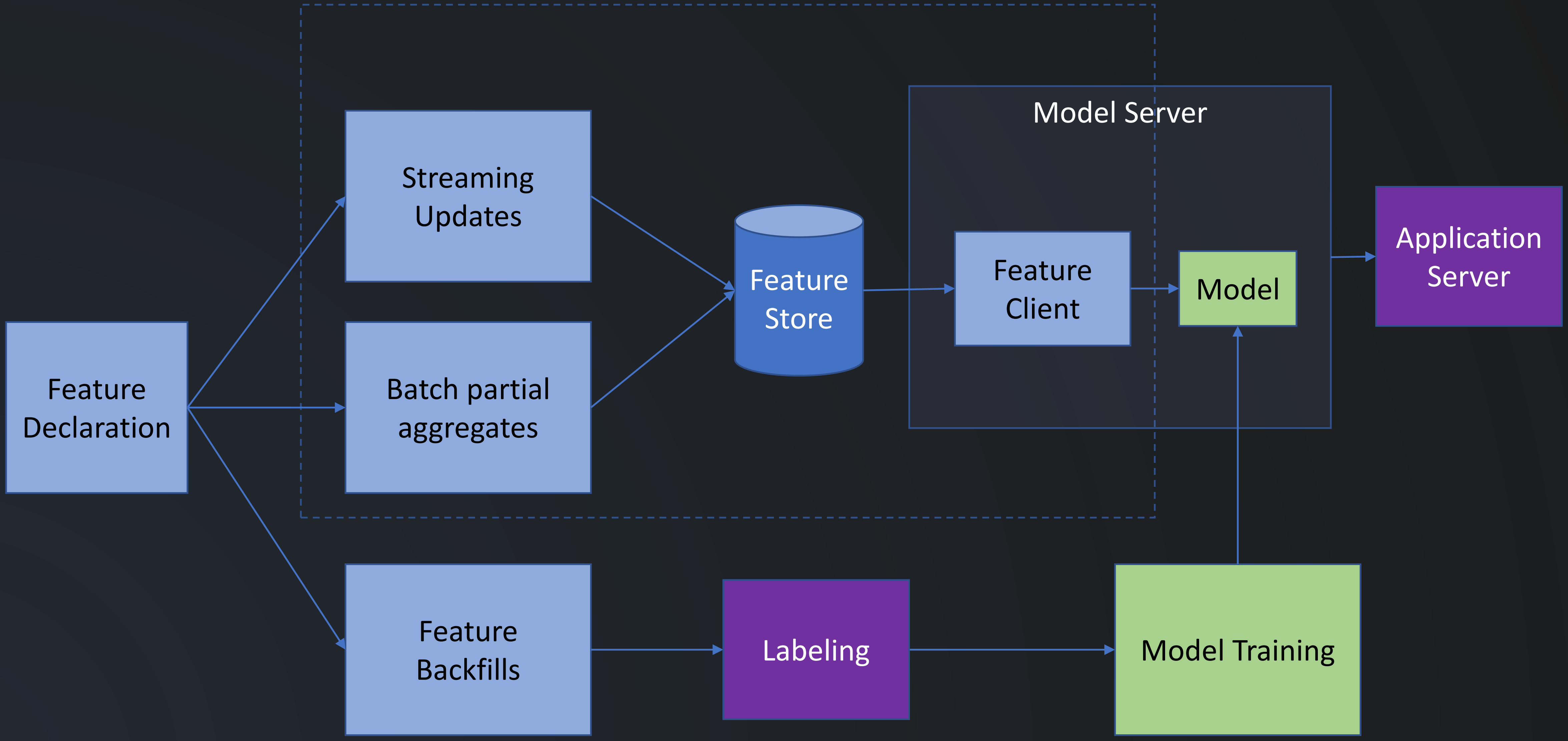
```
result = {}
for (key, query_times, events) in join_result:
    result[key] = make_tiles(query_times)
    for event in events:
        # O(log-queries) search + O(log-candidates) updates
        for tile in tile_range(event.time, event.time + window, query_times):
            result[key][tile] = update(event.payload, result[key][tile])

    # O(queries) merges
    result[key] = collapse_tiles(result[key])
```

Feature Backfill – Topology



Architecture



Questions