Building Real-Time Data Platforms with RisingWave and Clickhouse

Noel Kwan, RisingWave Labs, Nov. 2023

Adapted from slides by Samuel Hu, RisingWave Labs, Jun. 2023

About Me

- Database Kernel Engineer at RisingWave, building a cost-effective and scalable database solution designed to streamline data processing and query serving.
- Focus on Performance & Reliability, e.g. SqlSmith and Backfill
- Previously studied at NUS, focus area in Programming Languages.

Agenda

1. Stream/Batch Data Engineering
2. RisingWave Streaming Database
3. RisingWave X Clickhouse Demo

Agenda

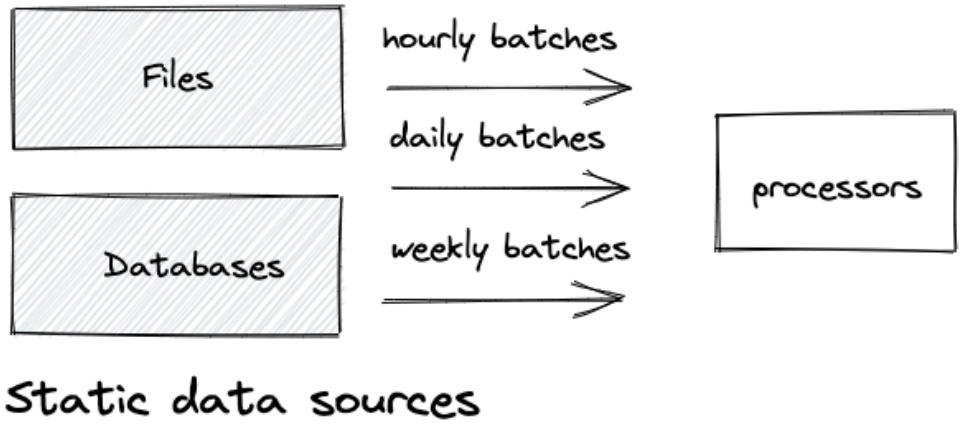
1. Stream/Batch Data Engineering

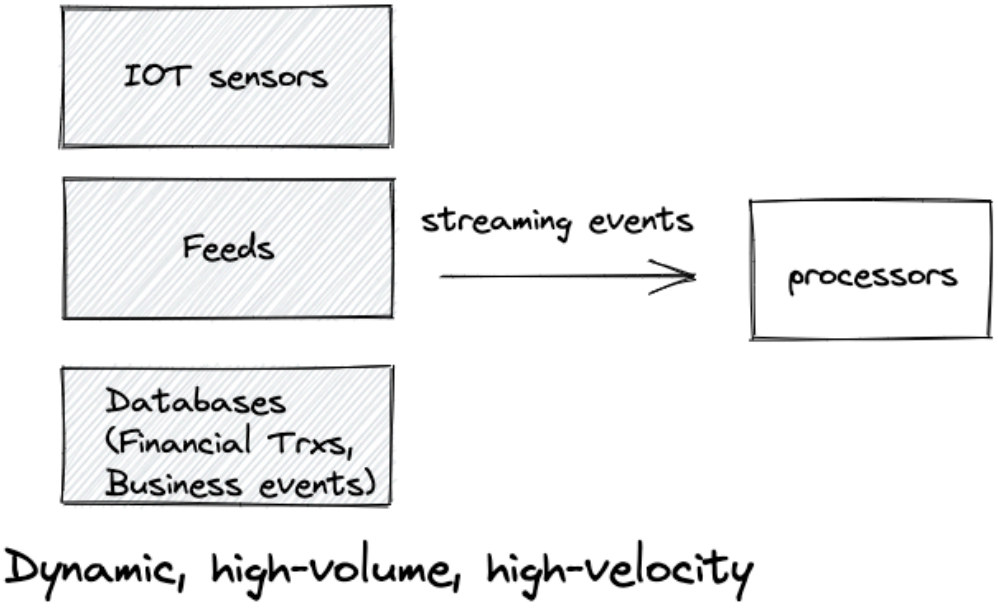
Batch Processing	teradata. PostgreSQL before year 2010 Single-Node Era	Apache Spark Year 2010 to 2020 Big Data Era	Spark SQL snowflake After year 2020 Cloud Era
Stream Processing	Microsoft SQL Server StreamInsight before year 2010 Single-Node Era	Flink Year 2010 to 2020 Big Data Era	RisingWave Flink After year 2020 Cloud Era

Batch or Stream

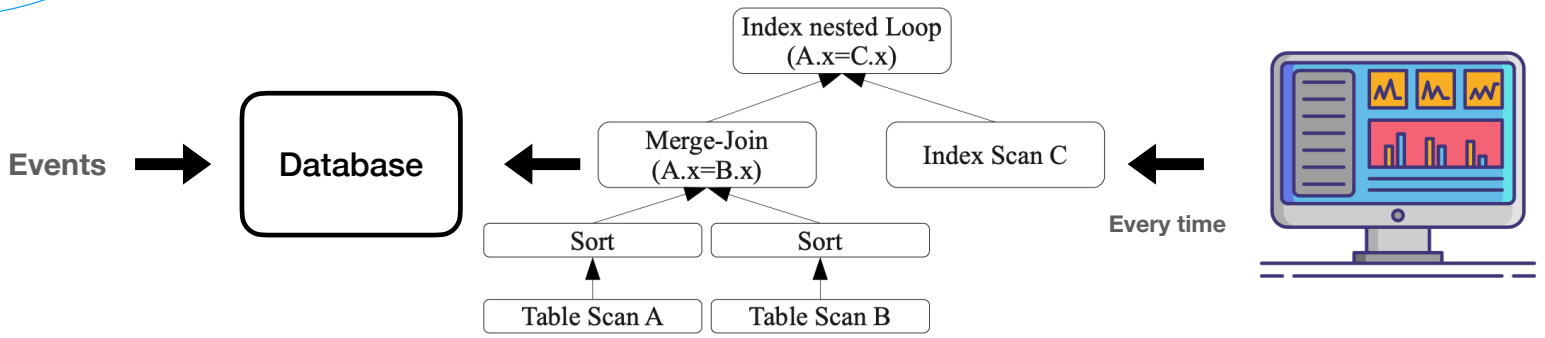
- How would you analyze social media data for sentiment analysis?
- How would you monitor the performance of a fleet of IoT devices?

Batch vs. Stream (Data Ingestion)

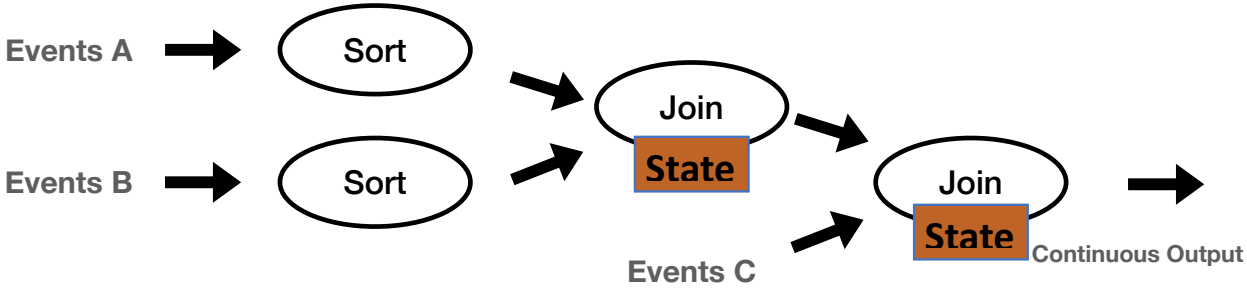




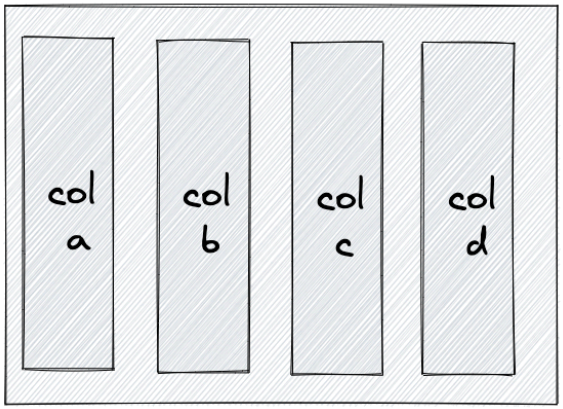
Batch vs. Stream (Data Processing Paradigm)

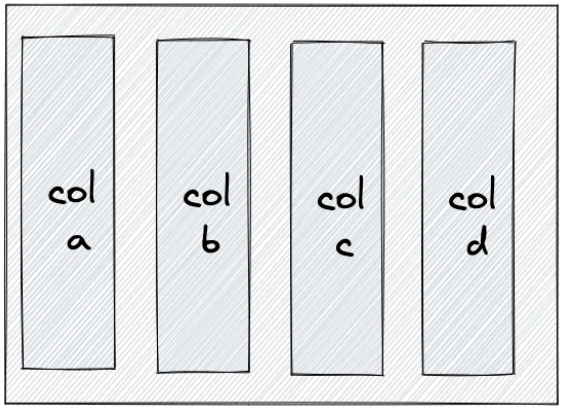


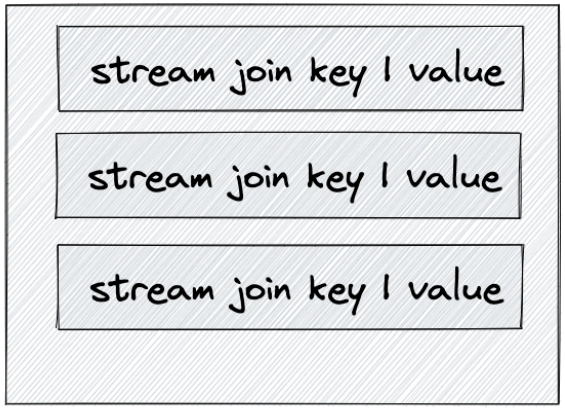

```
SELECT * FROM A JOIN B ON A.x = B.x JOIN C ON C.x = A.x;
```

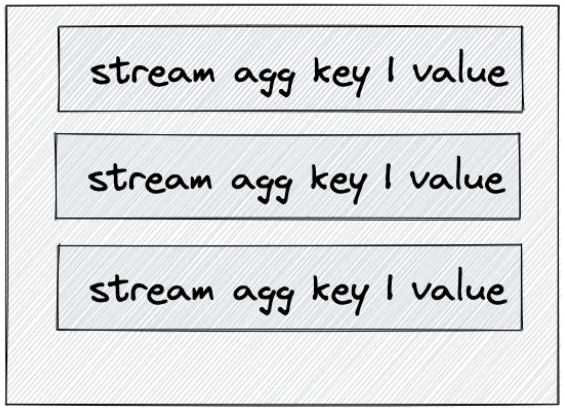


Batch vs. Stream (Storage)









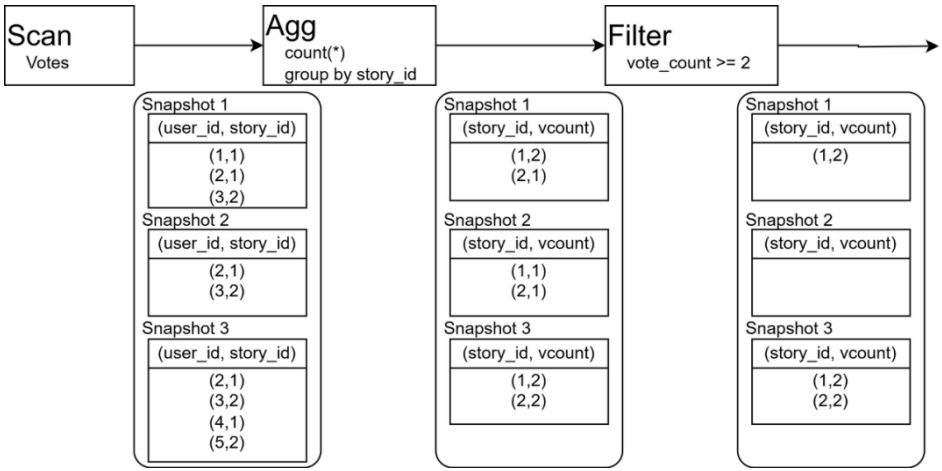
Agenda

2 RisingWave Streaming Database

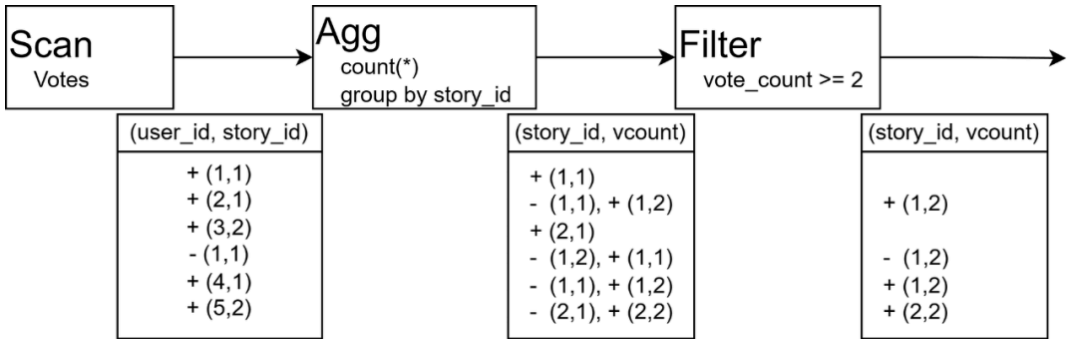
RisingWave's Computation Model

```
CREATE TABLE Votes(user_id int, story_id int);

CREATE MATERIALIZED VIEW StoriesVC AS
SELECT story_id, COUNT(*) AS vcount
FROM Votes GROUP BY story_id
HAVING vcount >= 2;
```

```
INSERT INTO Votes VALUES (1,1), (2,1), (3,2);
DELETE FROM Votes WHERE user_id = 1 AND story_id = 1;
INSERT INTO Votes VALUES (4,1), (5,2);
```

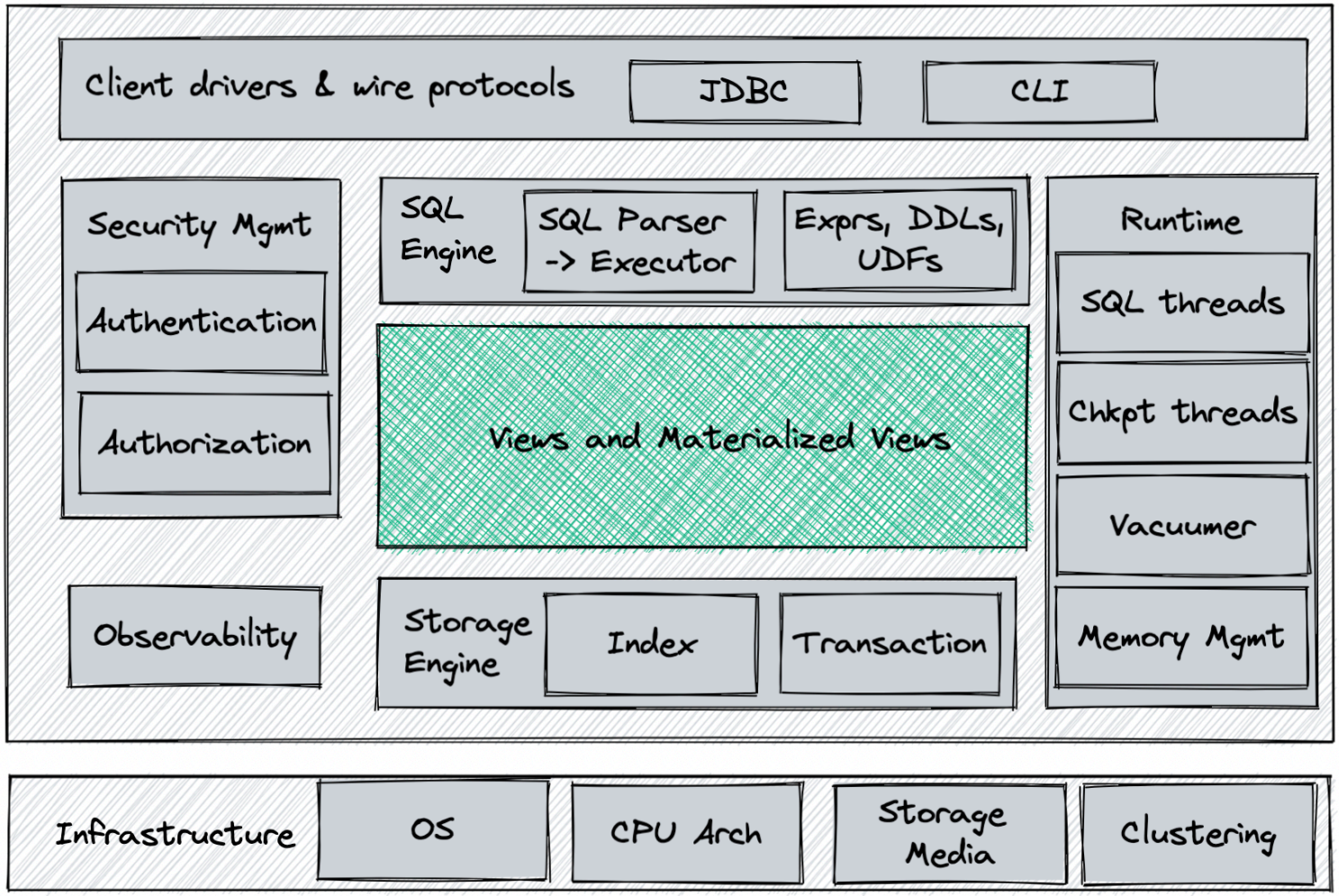


Streaming SQL built on Time-varying Relations



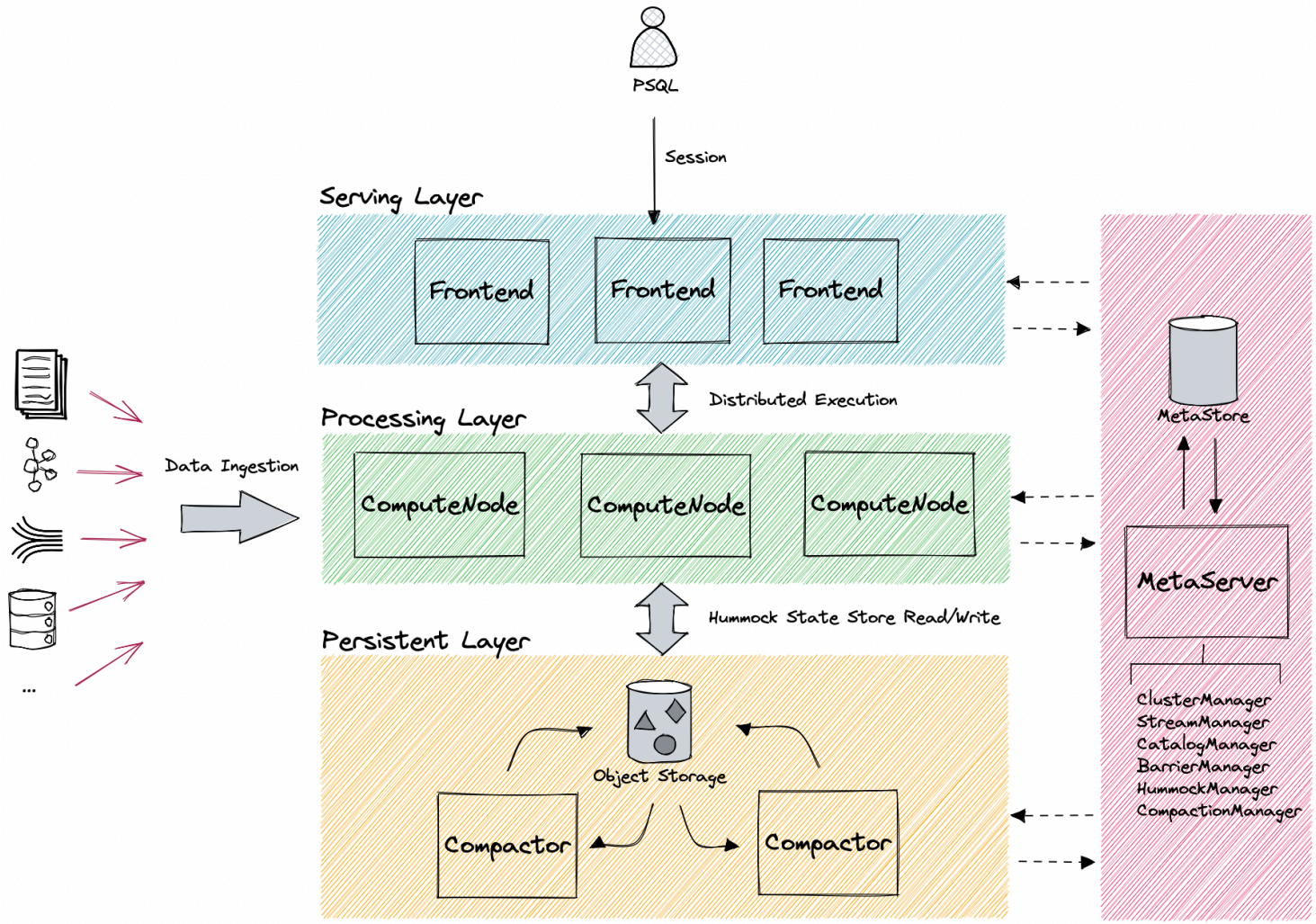
IVM with Change Propagations

Streaming Database is a Database Architecture…



- Materialized views to represent up-to-date results from changing data.
- PGWire and SQL.
- Compatible with Postgres catalogs, expressions, DDLs, DMLs, DQLs.
- Introduce streaming source syntax and treat it like a table.
- Incremental view maintenance.
- Distributed system.
- Checkpointing & recovery.

… But Purposely Built for Data Movement

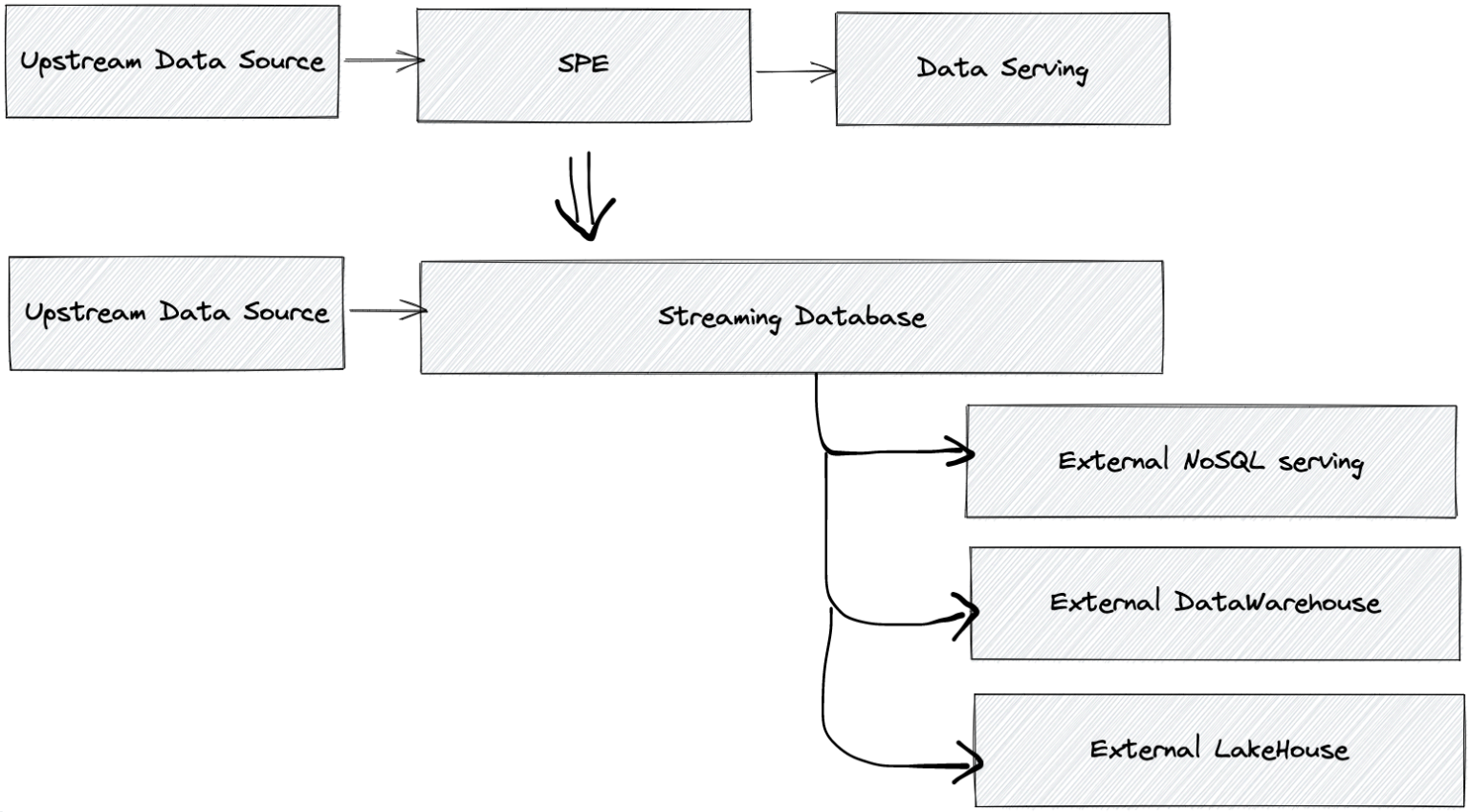


- **Frontend**: Parsing, validation, optimization, and answering the results of each individual query.
- **ComputeNode**: Executing the optimized query plan.
- **Compactor**: Executing the compaction tasks for our storage engine.
- **MetaServer**: The central metadata management service. There are multiple sub-components running in MetaServer. Key thing to highlight is that the MetaServer is responsible for the scheduling and monitoring of streaming jobs.

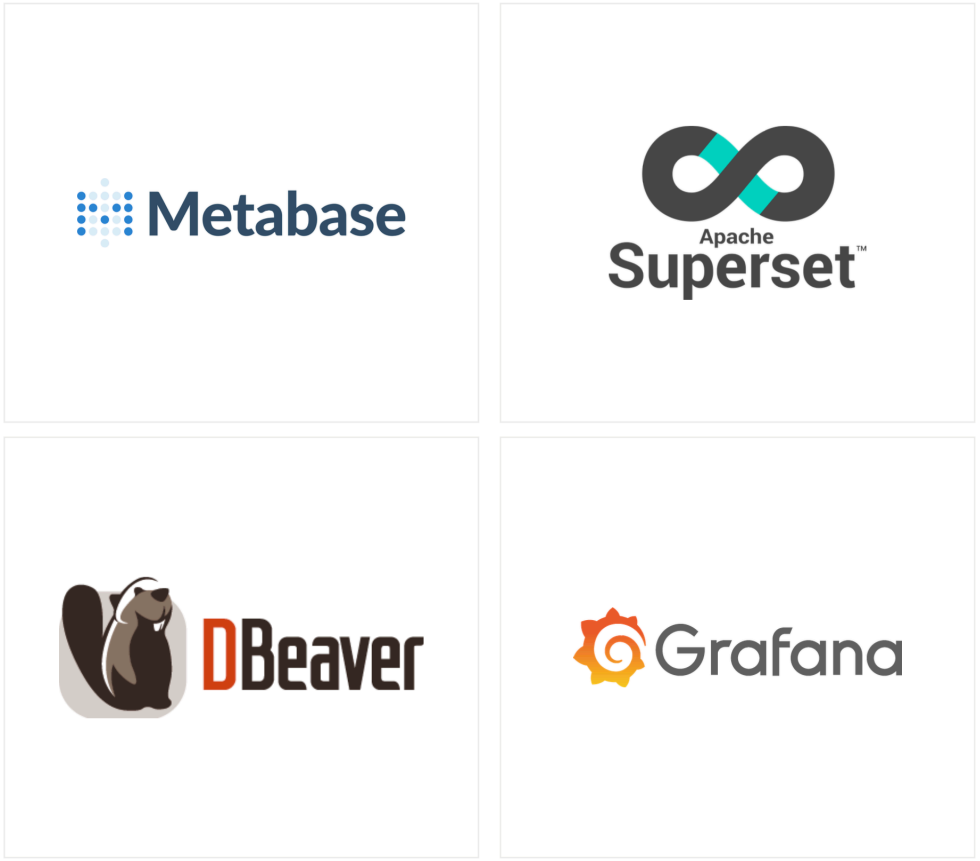
Choosing Streaming Processing Engines

Criteria	Flink	RisingWave
Processing model	Both Stream and Batch processing	Both Stream processing and Batch processing
Latency	Low (seconds - milliseconds)	Low (seconds - milliseconds)
Throughput	High	High
Fault tolerance	Exactly-once semantics with checkpoints and savepoints	Exactly-once semantics with checkpoints and snapshots
API support	Java, Scala, Python, SQL, Table API	SQL only
Storage system	External systems such as HDFS or Kafka	Internal storage system based on Hummock tiered storage.
State management	Advanced (supports stateful operators with various state backends and consistency levels)	Advanced (supports stateful queries with incremental view maintenance and materialized views)

User Journey (0/ Abstract)

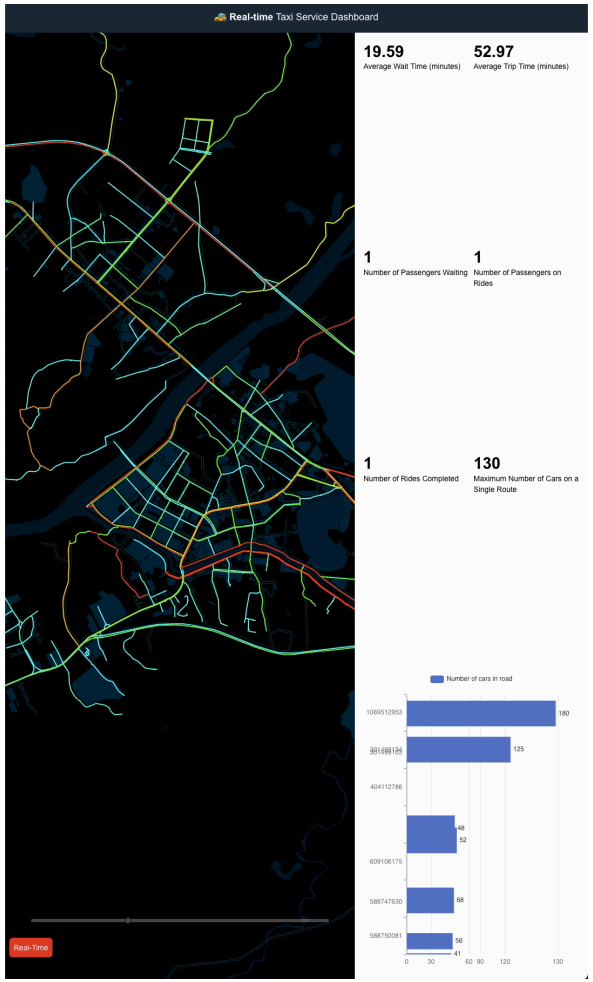


Streaming Database (Visualization)



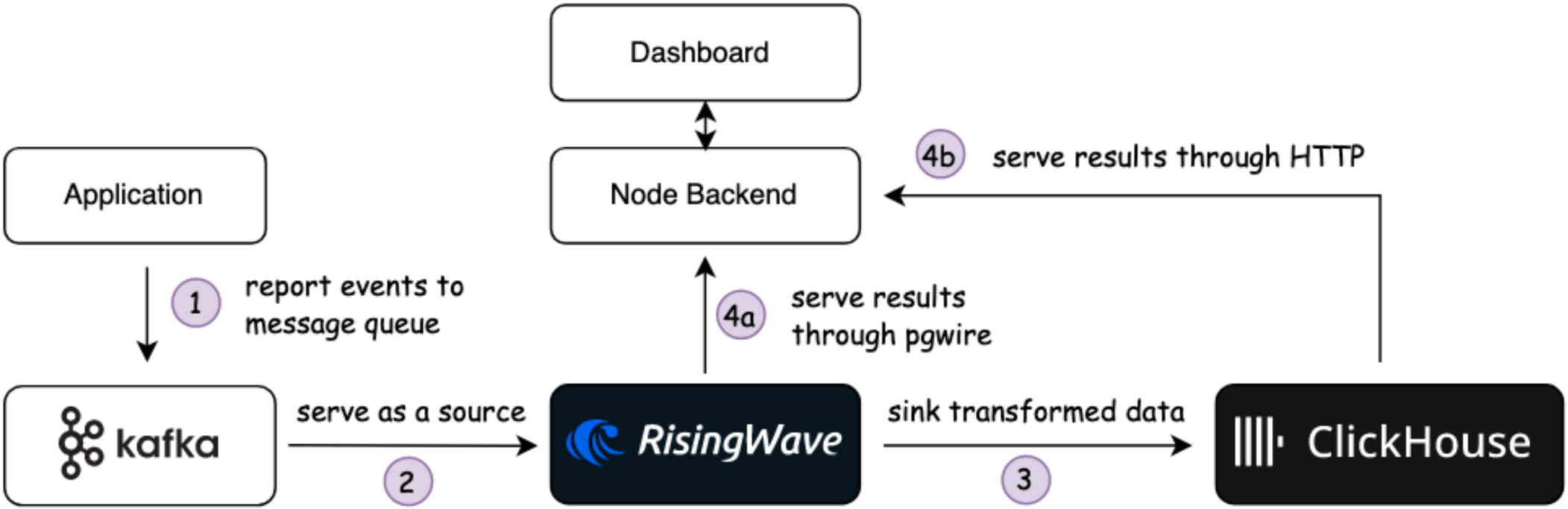
Streaming Database (Ecosystem)

Demos

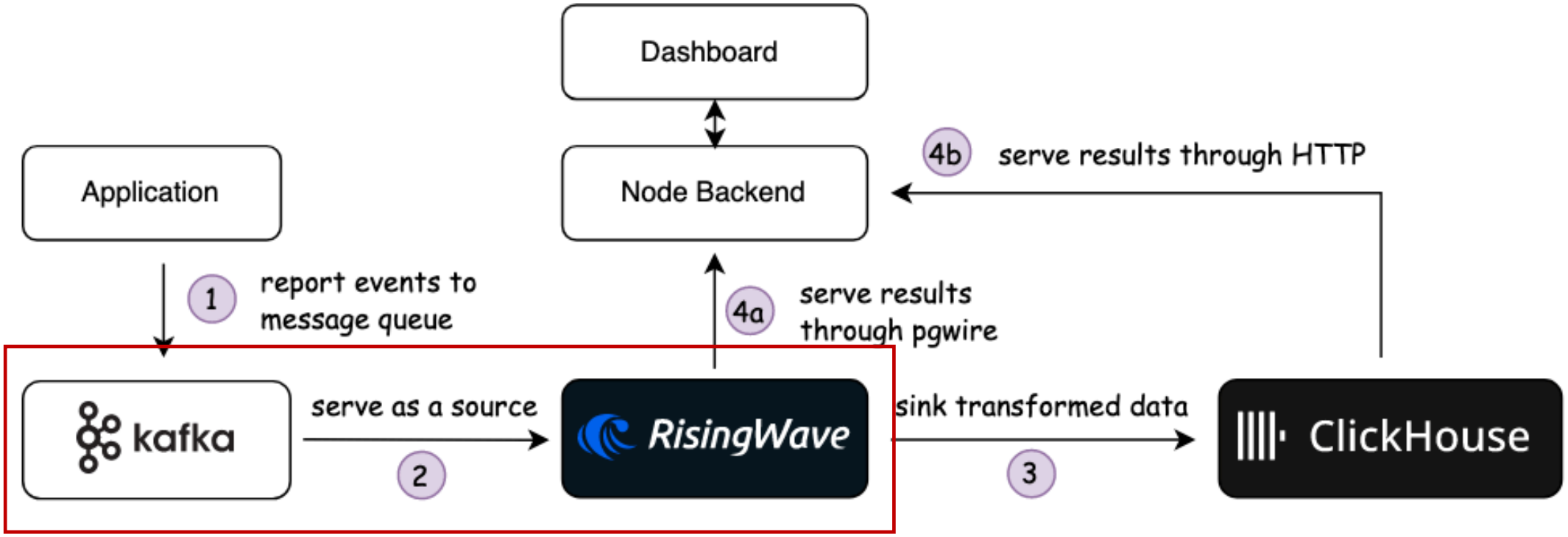


RisingWaveLabs

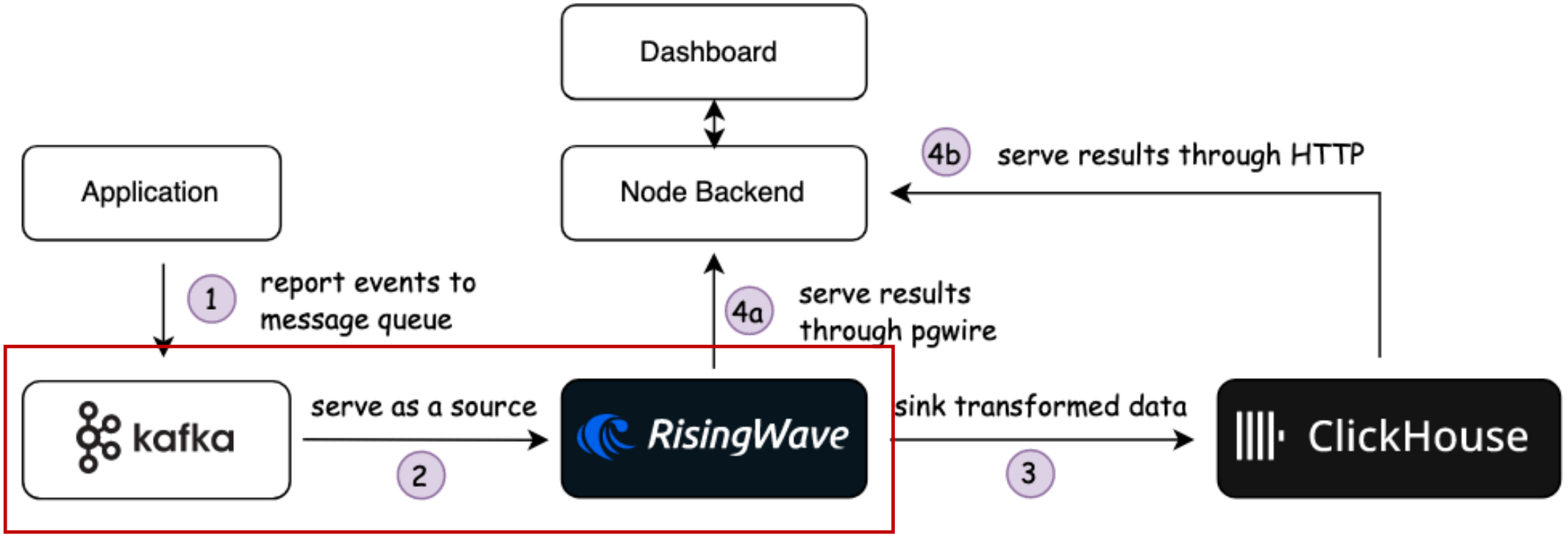
Real-time Ride Sharing Analytics Platform



Real-time Ride Sharing Analytics Platform

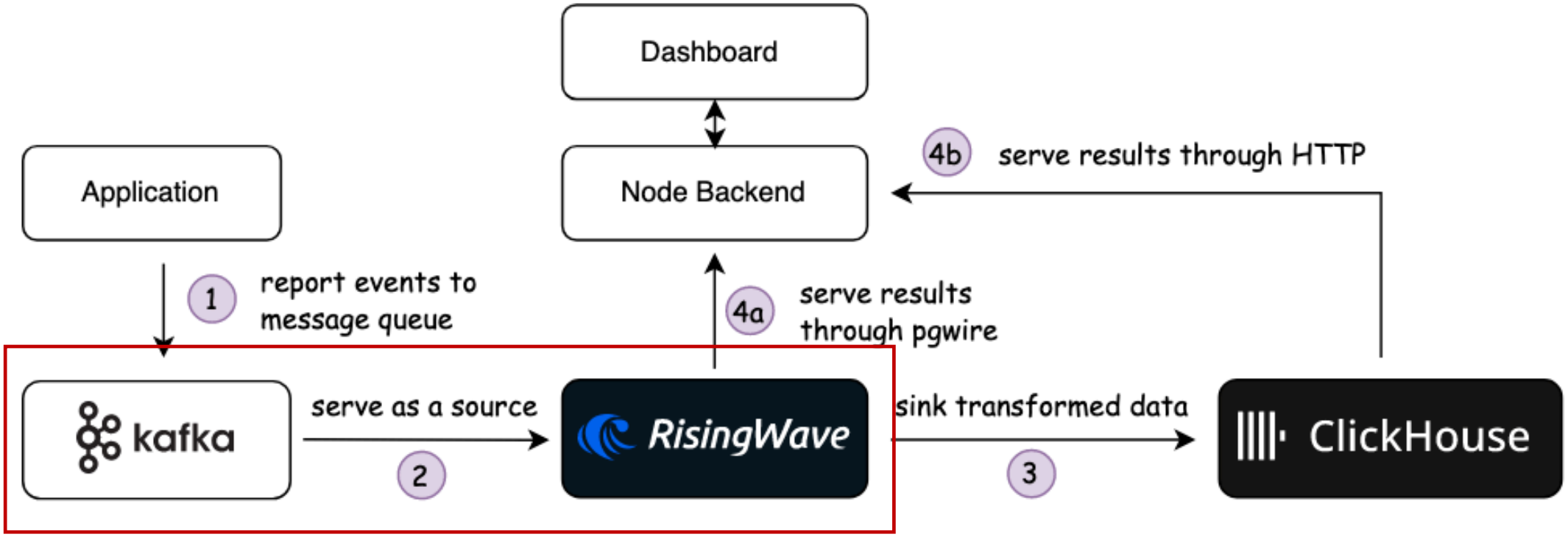


Real-time Ride Sharing Analytics Platform



1. CREATE SOURCE

Real-time Ride Sharing Analytics Platform



1. CREATE SOURCE
2. CREATE MATERIALIZED VIEW

```
CREATE SOURCE drivers (
    status STRING,
    driver_id STRING,
    location_id STRING,
    location_lon double precision,
    location_lat double precision,
    timestamp timestamptz,
    WATERMARK FOR timestamp AS timestamp - interval '5' MINUTE
)
WITH (
    connector = 'kafka',
    topic = 'drivers_pingshan',
    {kafka_info})
FORMAT PLAIN ENCODE json;
```

Create drivers source.
This ingests the drivers data into RisingWave

```
CREATE MATERIALIZED VIEW driver_status AS (
    SELECT * FROM
        drivers
    WHERE
        timestamp = (
            SELECT
                MAX(timestamp)
            FROM
                drivers));
```

Create a materialized view on the drivers source to record their latest status.

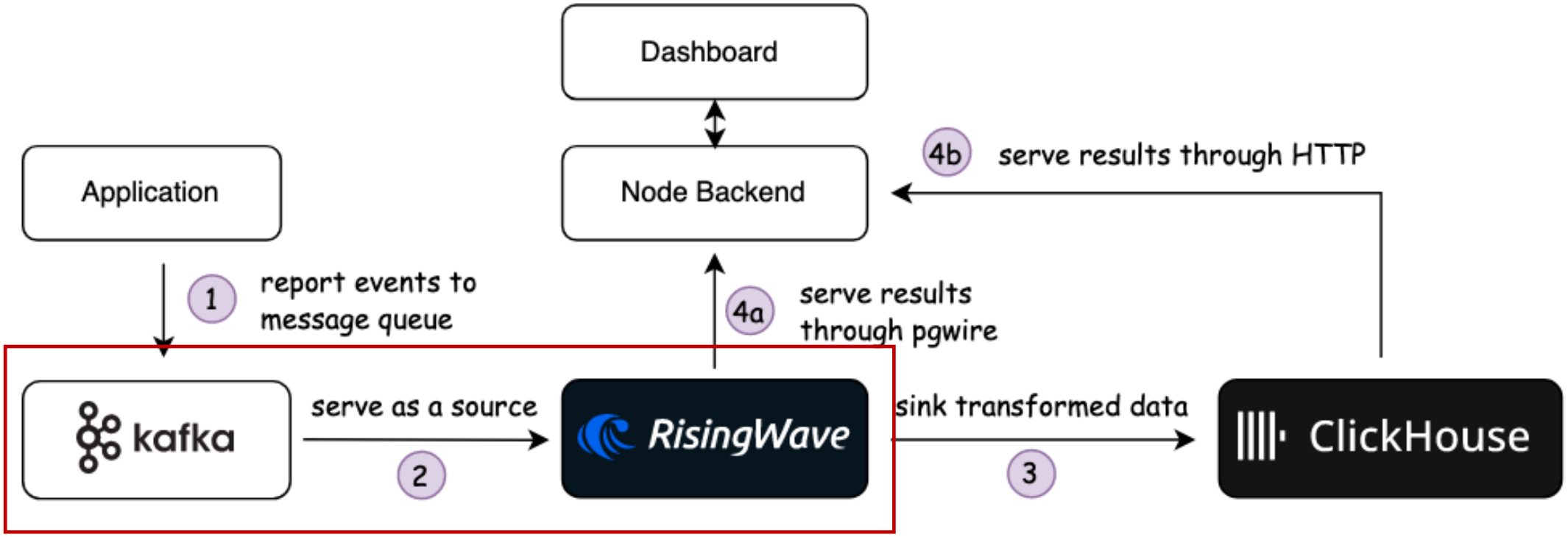
```
CREATE SOURCE rides (
    event_type STRING,
    timestamp timestamptz,
    ride_id STRING,
    driver_id STRING,
    pickup_location_id STRING,
    pickup_location_lon double precision,
    pickup_location_lat double precision,
    destination_location_id STRING,
    destination_location_lon double precision,
    destination_location_lat double precision,
    request_time timestamptz,
    handle_time timestamptz,
    pickup_time timestamptz,
    finish_time timestamptz,
    WATERMARK FOR timestamp AS timestamp - interval '10' SECOND
)
WITH (
    connector = 'kafka',
    topic = 'rides_pingshan',
    {kafka_info})
FORMAT PLAIN ENCODE json;
```

Create the rides source,
to ingest the rides data into RisingWave

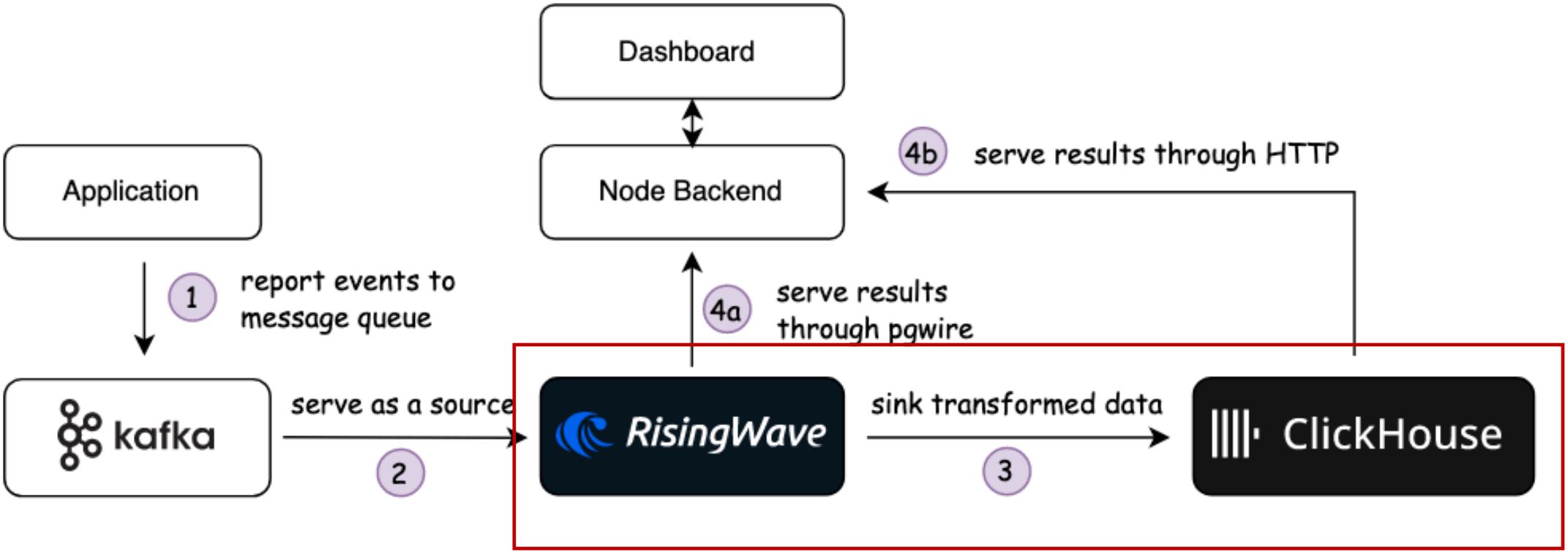
Real-time Ride Sharing Analytics Platform

```sql
CREATE MATERIALIZED VIEW ongoing_rides AS (
    SELECT * FROM rides WHERE timestamp = (SELECT MAX(timestamp) FROM rides)
);

CREATE MATERIALIZED VIEW avg_pickup_time AS (
    SELECT
        avg(extract (epoch from pickup_time) - extract (epoch from handle_time)) as avg_pickup_seconds
    FROM ongoing_rides WHERE pickup_time IS NOT NULL
);

CREATE MATERIALIZED VIEW avg_deliver_time AS (
    SELECT
        avg(extract (epoch from finish_time) - extract (epoch from pickup_time)) as avg_deliver_time
    FROM ongoing_rides WHERE finish_time IS NOT NULL
);

CREATE MATERIALIZED VIEW wait_for_handle AS (SELECT count(*), max(timestamp) FROM ongoing_rides WHERE event_type = 'ride_requested');
CREATE MATERIALIZED VIEW wait_for_pickup AS (SELECT count(*), max(timestamp) FROM ongoing_rides WHERE event_type = 'ride_handled');
CREATE MATERIALIZED VIEW people_on_road AS  (SELECT count(*), max(timestamp) FROM ongoing_rides WHERE event_type = 'ride_pickup');
CREATE MATERIALIZED VIEW cnt_ride_finish AS (SELECT count(*), max(timestamp) FROM ongoing_rides WHERE event_type = 'ride_finish');

CREATE MATERIALIZED VIEW busiest_way As (SELECT way_id, cnt, timestamp FROM way_status ORDER BY way_status.cnt DESC LIMIT 1);
```

Real-time Ride Sharing Analytics Platform



Real-time Ride Sharing Analytics Platform



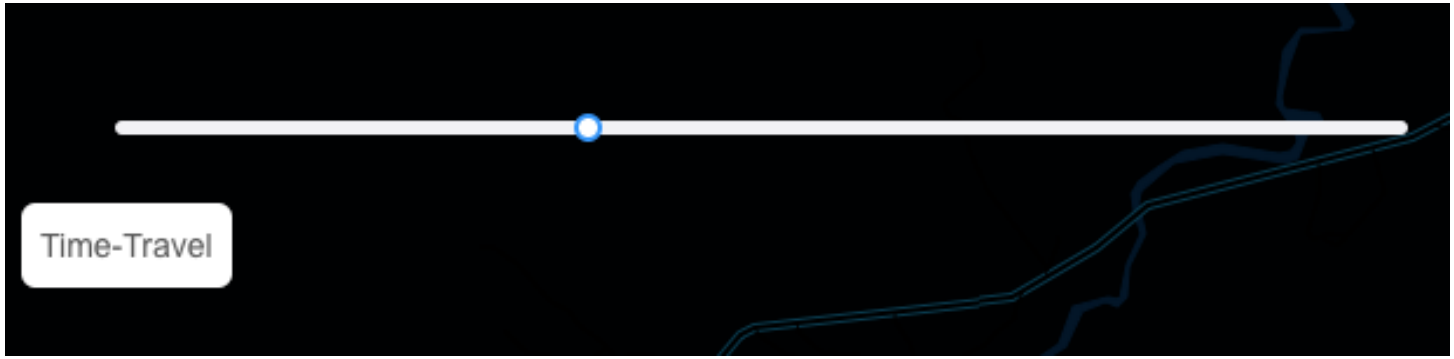
```
CREATE MATERIALIZED VIEW way_status AS (
    SELECT
      way_id,
      count(*) AS cnt,
      max(timestamp) as timestamp
    FROM driver_status
      JOIN node
      ON driver_status.location_id = node.id
    GROUP BY way_id
)
```

Create the way_status materialized view, which will get the status of each way (road).

```
CREATE SINK ridesharing_stat AS (
  SELECT
    way_id::string,
    cnt::int8,
    timestamp::timestamp
  FROM way_status
) WITH (
  connector='clickhouse',
  type = 'append-only',
  force_append_only = 'true',
  clickhouse.url = 'http://clickhouse:8123',
  clickhouse.user = 'default',
  clickhouse.password = '',
  clickhouse.database = 'default',
  clickhouse.table = 'ridesharing_stat'
);
```

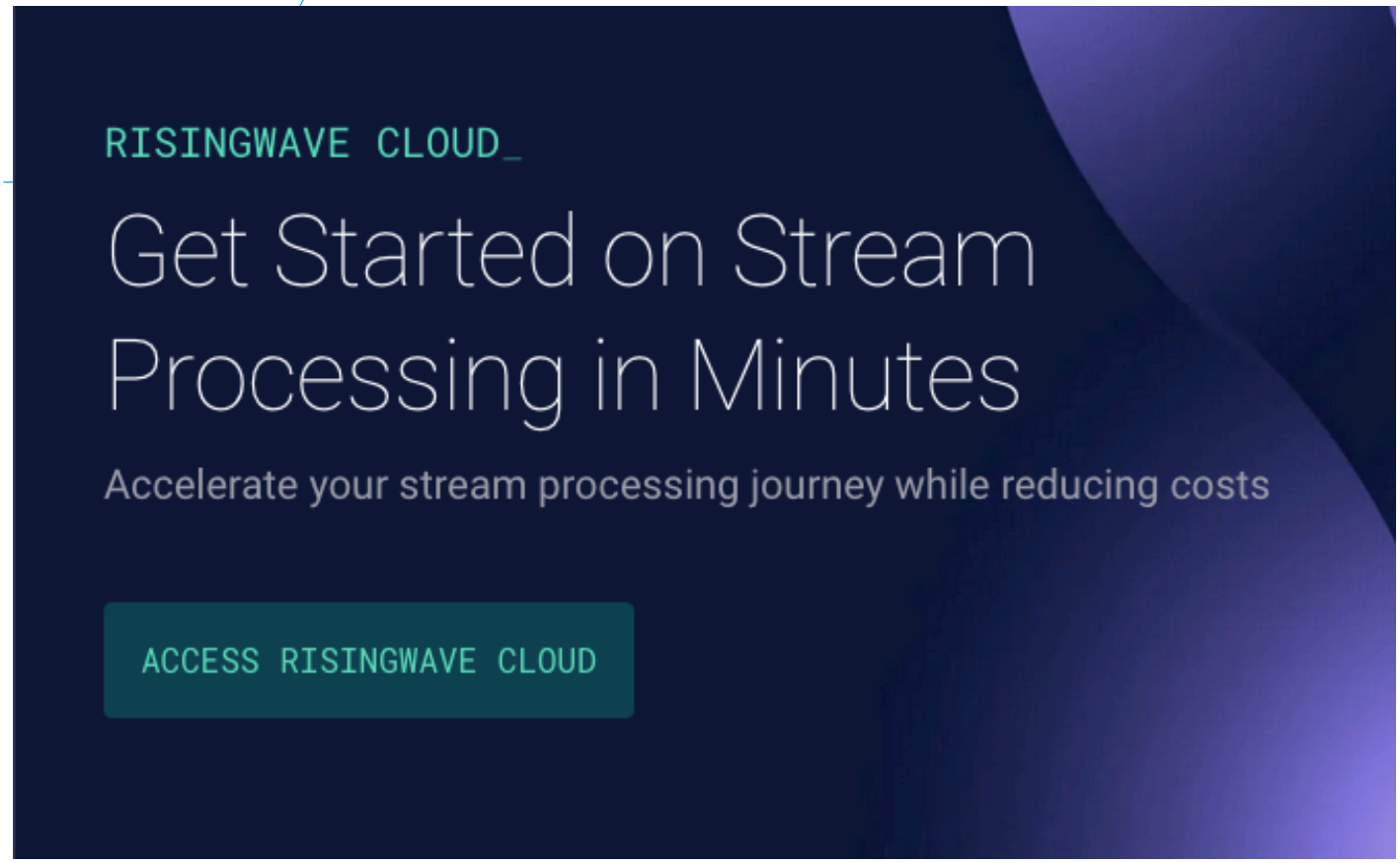
Finally sink this data out to clickhouse.

Real-time Ride Sharing Analytics Platform




```
SELECT way_id, avg(cnt)
FROM ridesharing_stat
WHERE ${time - 2.5 * 60} < timestamp
    AND timestamp < ${time + 2.5 * 60}
GROUP BY way_id
```

Thank you for listening



SCAN ME