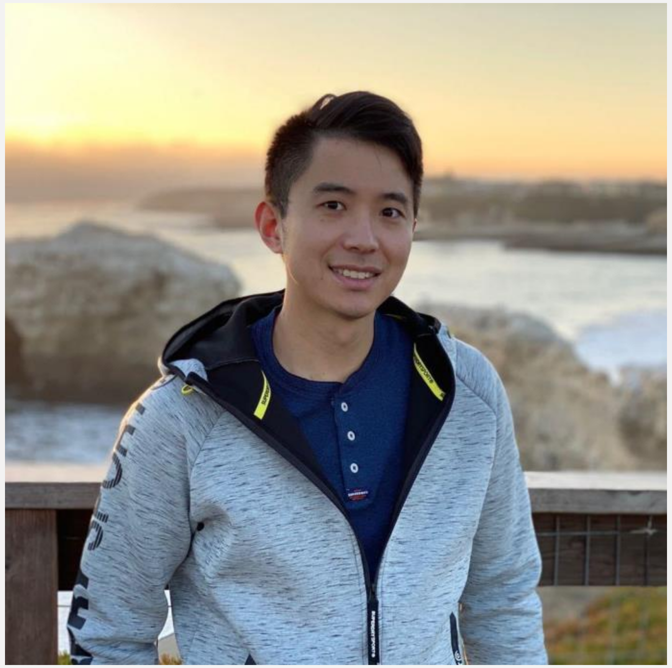
An Introduction to Stream Processing and Streaming Databases

Yingjun Wu
RisingWave Labs

risingwave.com/slack

Who Am I?

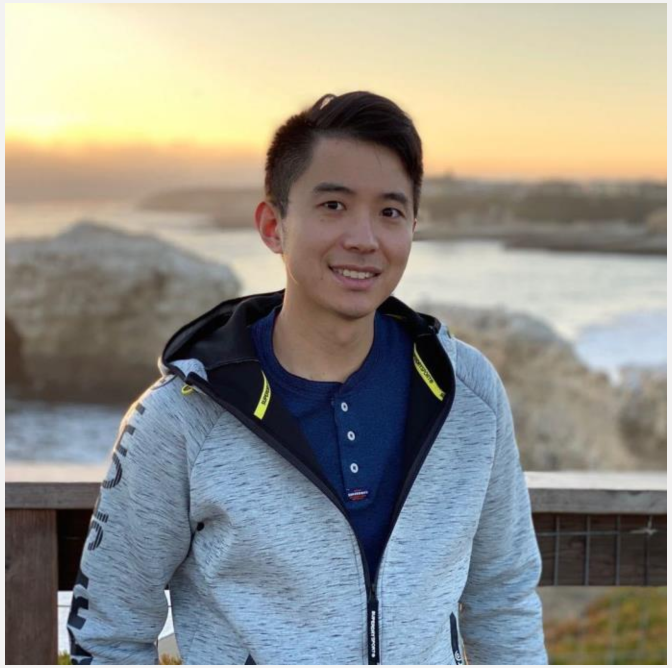
- Yingjun Wu (he/him/his)
  - Founder @RisingWave Labs
  - Ex-AWS Redshift
  - Ex-IBM Research Almaden
  - PhD’17, SoC, NUS



RisingWaveLabs

Who Am I?

- Yingjun Wu (he/him/his)
  - Founder @RisingWave Labs
  - Ex-AWS Redshift
  - Ex-IBM Research Almaden
  - PhD’17, SoC, NUS



Why Stream Processing?

Stock Trading Example

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Stock Trading Example

Calculate the total traded volume for each symbol in the dataset.

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

RisingWaveLabs

Stock Trading Example

Calculate the total traded volume for each symbol in the dataset.

- H ?

```sql
SELECT Symbol, SUM(Volume) AS TotalVolume
FROM Trades
GROUP BY Symbol;
```

Ti			Volume
20			300
20			200
20			500
20			100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Stock Trading Example

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Calculate the total traded volume for each symbol in the dataset.

What is the average trade price for TSLA between 09:30:04 and 09:30:09?

RisingWaveLabs

Stock Trading Example

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Calculate the total traded volume for each symbol in the dataset.

What is the average trade price for TSLA between 09:30:04 and 09:30:09?

Which trade(s) had the largest single-trade volume in the dataset?

Stock Trading Example

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Calculate the total traded volume for each symbol in the dataset.

What is the average trade price for TSLA between 09:30:04 and 09:30:09?

Which trade(s) had the largest single-trade volume in the dataset?

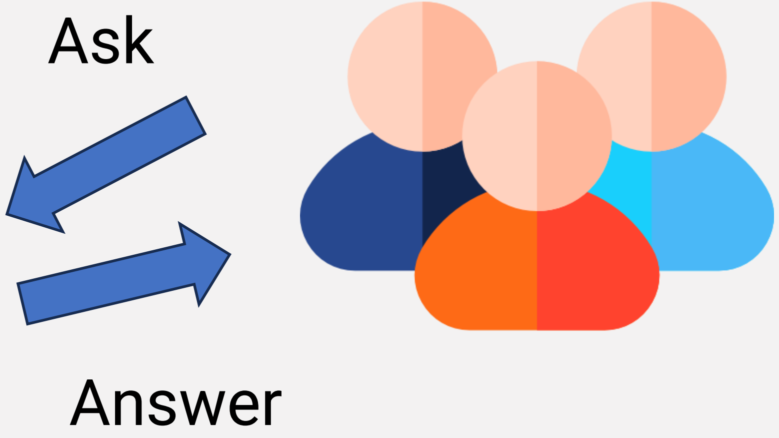
Ad-hoc queries, or “exploratory queries”

RisingWaveLabs

Stock Trading Example

- How traditional databases work?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200



Real-time stock market data

Ad-hoc queries, or “exploratory queries”

Stock Trading Example

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Real-time stock market data

Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
…30:07	GOOG	2750.10	400
…30:08	GOOG	2751.50	300
…30:09	TSLA	781.00	600
…30:10	AAPL	149.95	200

Real-time stock market data

Stock Trading Example

- How about doing monitoring?

Every second, calculate the average trade price for each symbol in the last 10 seconds.

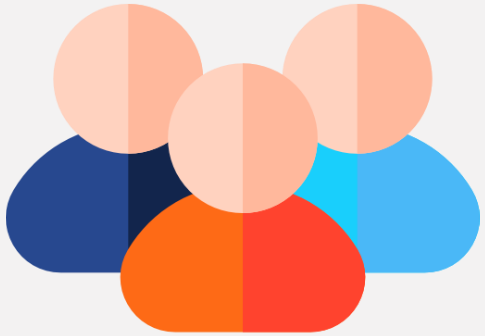
Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```sql
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Real-time stock market data

Ask

Answer



Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```sql
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Update Time	AAPL Avg Price	TSLA Avg Price	GOOG Avg Price

Real-time stock market data

Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```sql
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Trigger query!

Update Time	AAPL Avg Price	TSLA Avg Price	GOOG Avg Price
09:30:07	150.16	780.625	2750.10

Real-time stock market data

Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Trigger query!

Trigger query!

Update Time	AAPL Avg Price	TSLA Avg Price	GOOG Avg Price
09:30:07	150.16	780.625	2750.10
09:30:08	150.16	780.625	2750.80

Real-time stock market data

Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```sql
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Real-time stock market data

Trigger query!

Trigger query!

Trigger query!

Update Time	AAPL Avg Price	TSLA Avg Price	GOOG Avg Price
09:30:07	150.16	780.625	2750.10
09:30:08	150.16	780.625	2750.80
09:30:09	150.16	780.75	2750.80

Stock Trading Example

Every second, calculate the average trade price for each symbol in the last 10 seconds.

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01			
2025-02-08 09:30:02			
2025-02-08 09:30:03			
2025-02-08 09:30:04			
2025-02-08 09:30:05			
2025-02-08 09:30:06			
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

```sql
SELECT symbol, AVG(price) AS avg_price
FROM trades
WHERE event_time >= NOW() - INTERVAL 10 SECOND
GROUP BY symbol;
```

Trigger query!

Trigger query!

Trigger query!

Trigger query!

Update Time	AAPL Avg Price	TSLA Avg Price	GOOG Avg Price
09:30:07	150.16	780.625	2750.10
09:30:08	150.16	780.625	2750.80
09:30:09	150.16	780.75	2750.80
09:30:10	150.12	780.75	2750.80

Real-time stock market data

Stock Trading Exa

- How about doing monit

Timestamp	Symbol
2025-02-08 09:30:01	
2025-02-08 09:30:02	
2025-02-08 09:30:03	
2025-02-08 09:30:04	
2025-02-08 09:30:05	
2025-02-08 09:30:06	
30:07	GOOG
30:08	GOOG
30:09	TSLA
30:10	AAPL

```sql
SELECT symbol, AVG(price)
FROM trades
WHERE event_time >= NOW()
GROUP BY symbol;
```

Real-time stock mark

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class StreamQueryExample {

    // Replace these with your actual DB connection details
    private static final String JDBC_URL = "jdbc:postgresql://localhost:5432/mydatabase";
    private static final String DB_USER = "myuser";
    private static final String DB_PASSWORD = "mypassword";

    // Our SQL query to get average trade price for each symbol in the last 10 seconds
    // Note: This won't "stream" results in real time; it simply queries the current data every second.
    private static final String AVERAGE_PRICE_QUERY =
        "SELECT symbol, AVG(price) AS avg_price " +
        "FROM trades " +
        "WHERE event_time >= now() - interval '10 seconds' " +
        "GROUP BY symbol";

    public static void main(String[] args) {
        // 1. Establish a database connection (ideally once, or use a connection pool).
        try (Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER, DB_PASSWORD)) {
            System.out.println("Connected to PostgreSQL database!");

            // 2. Set up a scheduled task to run every 1 second.
            ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
            executor.scheduleAtFixedRate(() -> {
                queryAndPrintAveragePrice(connection);
            }, 0, 1, TimeUnit.SECONDS);

            // Keep the program running (for demo purposes).
            // In a real application, you'd handle shutdown logic more gracefully.
            Thread.sleep(60_000); // Run for 1 minute, then stop
            executor.shutdown();
            System.out.println("Shutting down.");

        } catch (SQLException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private static void queryAndPrintAveragePrice(Connection connection) {
        // 3. Execute the SQL query and print results.
        try (PreparedStatement pstmt = connection.prepareStatement(AVERAGE_PRICE_QUERY);
             ResultSet rs = pstmt.executeQuery()) {

            System.out.println("=== Average Price (Last 10s) ===");
            while (rs.next()) {
                String symbol = rs.getString("symbol");
                double avgPrice = rs.getDouble("avg_price");
                System.out.printf("Symbol: %s | Avg Price: %.2f%n", symbol, avgPrice);
            }
            System.out.println("--------------------------------");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

culate the average trade
nbol in the last 10

rice	TSLA Avg Price	GOOG Avg Price
	780.625	2750.10
	780.625	2750.80
	780.75	2750.80
	780.75	2750.80

RisingWaveLabs

Stock Trading Exa

- How about doing monit

Timestamp	Symbol
2025-02-08 09:30:01	
2025-02-08 09:30:02	
2025-02-08 09:30:03	
2025-02-08 09:30:04	
2025-02-08 09:30:05	
2025-02-08 09:30:06	
30:07	GOOG
30:08	GOOG
30:09	TSLA
30:10	AAPL

```
SELECT symbol, AVG(price)
FROM trades
WHERE event_time ≥ NOW()
GROUP BY symbol;
```

Real-time stock mark

Complex...

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class StreamQueryExample {

    // Replace these with your actual DB connection details
    private static final String JDBC_URL = "jdbc:postgresql://localhost:5432/mydatabase";
    private static final String DB_USER = "myuser";
    private static final String DB_PASSWORD = "mypassword";

    // Our SQL query to get average trade price for each symbol in the last 10 seconds
    // Note: This won't "stream" results in real time; it simply queries the current data every second.
    private static final String AVERAGE_PRICE_QUERY =
        "SELECT symbol, AVG(price) AS avg_price " +
        "FROM trades " +
        "WHERE event_time ≥ now() - interval '10 seconds' " +
        "GROUP BY symbol";

    public static void main(String[] args) {
        // 1. Establish a database connection (ideally once, or use a connection pool).
        try (Connection connection = DriverManager.getConnection(JDBC_URL, DB_USER, DB_PASSWORD)) {
            System.out.println("Connected to PostgreSQL database!");

            // 2. Set up a scheduled task to run every 1 second.
            ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
            executor.scheduleAtFixedRate(() → {
                queryAndPrintAveragePrice(connection);
            }, 0, 1, TimeUnit.SECONDS);

            // Keep the program running (for demo purposes).
            // In a real application, you'd handle shutdown logic more gracefully.
            Thread.sleep(60_000); // Run for 1 minute, then stop
            executor.shutdown();
            System.out.println("Shutting down.");

        } catch (SQLException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private static void queryAndPrintAveragePrice(Connection connection) {
        // 3. Execute the SQL query and print results.
        try (PreparedStatement pstmt = connection.prepareStatement(AVERAGE_PRICE_QUERY);
             ResultSet rs = pstmt.executeQuery()) {

            System.out.println("=== Average Price (Last 10s) ===");
            while (rs.next()) {
                String symbol = rs.getString("symbol");
                double avgPrice = rs.getDouble("avg_price");
                System.out.printf("Symbol: %s | Avg Price: %.2f%n", symbol, avgPrice);
            }
            System.out.println("--------------------------------");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

culate the average trade
mbol in the last 10

rice	TSLA Avg Price	GOOG Avg Price
	780.625	2750.10
	780.625	2750.80
	780.75	2750.80
	780.75	2750.80

RisingWaveLabs

Stock Trading Example

- How about doing monitoring?

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
30:07	GOOG	2750.10	400
30:08	GOOG	2751.50	300
30:09	TSLA	781.00	600
30:10	AAPL	149.95	200

Full table scan

Real-time stock market data

Complex... and inefficient!!

Why Stream Processing?

- Challenges:
  - Require humans (or programs) to repeatedly issue ad-hoc queries
  - Have to perform full computation queries

Why Stream Processing?

- Challenges:
  - Require humans (or programs) to repeatedly issue ad-hoc queries
  - Have to perform full computation queries

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250

Scan

Why Stream Processing?

- Challenges:
  - Require humans (or programs) to repeatedly issue ad-hoc queries
  - Have to perform full computation queries

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA	780.50	100
2025-02-08 09:30:05	TSLA	780.75	150
2025-02-08 09:30:06	AAPL	150.00	250
2025-02-08 09:30:07	GOOG	2750.10	400
2025-02-08 09:30:08	GOOG	2751.50	300
2025-02-08 09:30:09	TSLA	781.00	600
2025-02-08 09:30:10	AAPL	149.95	200

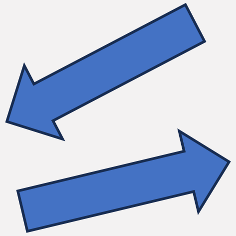
Scan

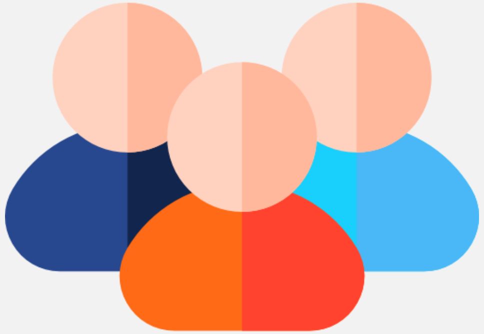
RisingWaveLabs

Why Stream Processing?

- Let’s do stream processing!
  - Instead of letting users issue queries proactively…

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA		
2025-02-08 09:30:05	TSLA		
2025-02-08 09:30:06	AAPL		
2025-02-08 09:30:07	GOOG		
2025-02-08 09:30:08	GOOG		
2025-02-08 09:30:09	TSLA		
2025-02-08 09:30:10	AAPL		



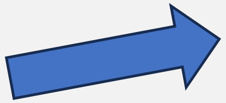


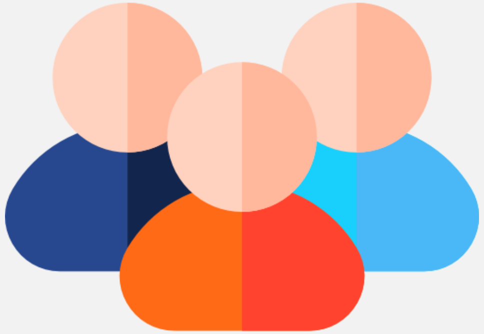
Send queries to databases

Why Stream Processing?

- Let's do stream processing!
  - Instead of letting users issue queries proactively…
  - Let databases push results to users!

Timestamp	Symbol	Price	Volume
2025-02-08 09:30:01	AAPL	150.25	300
2025-02-08 09:30:02	AAPL	150.30	200
2025-02-08 09:30:03	AAPL	150.10	500
2025-02-08 09:30:04	TSLA		
2025-02-08 09:30:05	TSLA		
2025-02-08 09:30:06	AAPL		
2025-02-08 09:30:07	GOOG		
2025-02-08 09:30:08	GOOG		
2025-02-08 09:30:09	TSLA		
2025-02-08 09:30:10	AAPL		

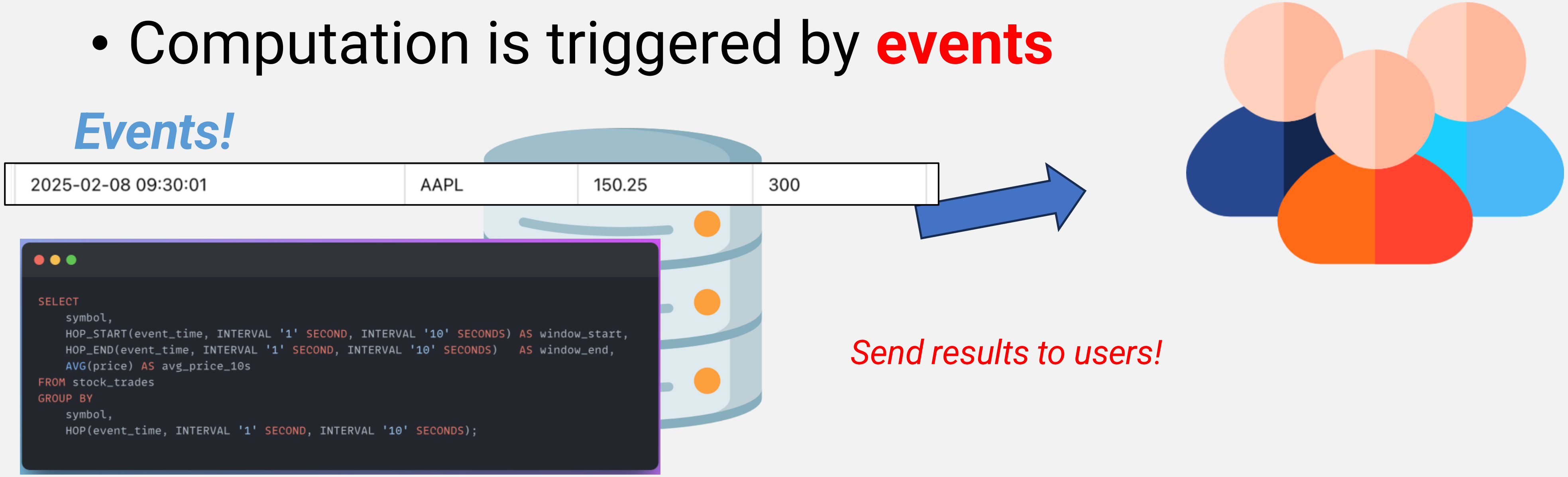




Send results to users!

Why Stream Processing?

- Let's do stream processing!
  - Instead of letting users issue queries proactively…
  - Let databases push results to users!
- Require defining queries beforehand
- Computation is triggered by **events**




```sql
SELECT
    symbol,
    HOP_START(event_time, INTERVAL '1' SECOND, INTERVAL '10' SECONDS) AS window_start,
    HOP_END(event_time, INTERVAL '1' SECOND, INTERVAL '10' SECONDS)   AS window_end,
    AVG(price) AS avg_price_10s
FROM stock_trades
GROUP BY
    symbol,
    HOP(event_time, INTERVAL '1' SECOND, INTERVAL '10' SECONDS);
```

RisingWave Labs

Why Stream Processing?

Monitoring Streams – A New Class of Data Management Applications

Don Carney, Brown University, *dpc@cs.brown.edu*
Uğur Çetintemel, Brown University, *ugur@cs.brown.edu*
Mitch Cherniack, Brandeis University, *mfc@cs.brandeis.edu*
Christian Convey, Brown University, *cjc@cs.brown.edu*
Sangdon Lee, Brown University, *sdlee@cs.brown.edu*
Greg Seidman, Brown University, *gss@cs.brown.edu*
Michael Stonebraker, M.I.T., *stonebraker@lcs.mit.edu*
Nesime Tatbul, Brown University, *tatbul@cs.brown.edu*
Stan Zdonik, Brown University, *sbz@cs.brown.edu*

Abstract

This paper introduces monitoring applications, which we will show differ substantially from conventional business data processing. The fact that a software system must process and react to continual inputs from many sources (e.g., sensors) rather than from human operators requires one to rethink the fundamental architecture of a DBMS for this application area. In this paper, we present Aurora, a new DBMS that is currently under construction at Brandeis University, Brown University, and M.I.T. We describe the basic system architecture, a stream-oriented set of operators, optimization tactics, and support for real-time operation.

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions on this repository. We call this a *Human-Active, DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrives asynchronously and answers must be computed with incomplete information. Lastly, DBMSs assume that applications require no real-time services.

There is a substantial class of applications where all five assumptions are problematic. Monitoring applications are applications that monitor continuous streams of data. This class of applications includes military applications that monitor readings from sensors worn by soldiers (e.g., blood pressure, heart rate, position), financial analysis applications that monitor streams of stock data reported from various stock exchanges, and tracking applications that monitor the locations of large numbers of objects for which they are responsible (e.g., audio-visual departments that must monitor the location of borrowed equipment). Because of the high volume of monitored data and the query requirements for these applications, monitoring applications would benefit from DBMS support. Existing DBMS systems, however, are ill suited for such applications since they target business applications.

First, monitoring applications get their data from external sources (e.g., sensors) rather than from humans issuing transactions. The role of the DBMS in this context is to alert humans when abnormal activity is detected. This is a *DBMS-Active, Human-Passive* (*DAHP*) model.

Second, monitoring applications require data management that extends over some history of values reported in a stream, and not just over the most recently reported values. Consider a monitoring application that tracks the location of items of interest, such as overhead transparency projectors and laptop computers, using electronic property stickers attached to the objects. Ceiling-mounted sensors inside a building and the GPS system in the open air generate large volumes of location data. If a reserved overhead projector is not in its proper location, then one might want to know the geographic position of the missing projector. In this case, the last value of the monitored object is required. However, an administrator might also want to know the duty cycle of the projector, thereby requiring access to the entire historical time series.

Third, most monitoring applications are trigger-oriented. If one is monitoring a chemical plant, then one wants to alert an operator if a sensor value gets too high or if another sensor value has recorded a value out of range more than twice in the last 24 hours. Every application could potentially monitor multiple streams of data, requesting alerts if complicated conditions are met. Thus, the scale of

† This work was supported by the National Science Foundation under NSF Grant number IIS00-86057 and a gift from Sun Microsystems.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

RisingWave Labs

Why Stream Processing?

Monitoring Streams – A New Class of Data M

Don Carney
Brown University
dpc@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Mitch
Brande
mfc@cs

Sangdon Lee
Brown University
sdlee@cs.brown.edu

Greg Seidman
Brown University
gss@cs.brown.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

Abstract

This paper introduces monitoring applications, which we will show differ substantially from conventional business data processing. The fact that a software system must process and react to continual inputs from many sources (e.g., sensors) rather than from human operators requires one to rethink the fundamental architecture of a DBMS for this application area. In this paper, we present Aurora, a new DBMS that is currently under construction at Brandeis University, Brown University, and M.I.T. We describe the basic system architecture, a stream-oriented set of operators, optimization tactics, and support for real-time operation.

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions on this repository. We call this a *Human-Active*, *DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrives asynchronously

† This work was supported by the National Science Foundation under NSF Grant number IIS00-86057 and a gift from Sun Microsystems.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

sensor value has recorded a value out of range more than twice in the last 24 hours. Every application could potentially monitor multiple streams of data, requesting alerts if complicated conditions are met. Thus, the scale of

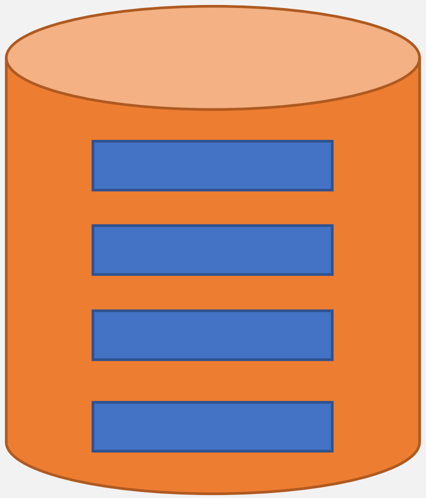
1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions on this repository. We call this a *Human-Active*, *DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrives asynchronously

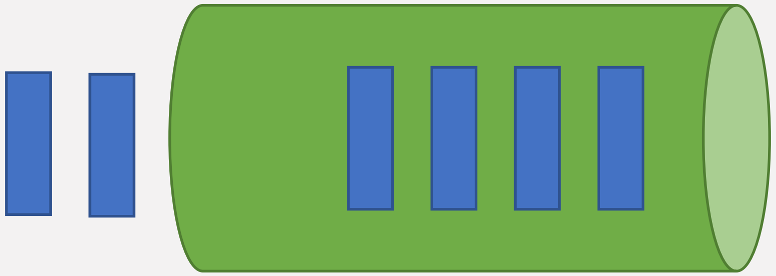
Batch Processing vs. Stream Processing

User-initiated computation
Full computation

Event-driven computation
Incremental computation



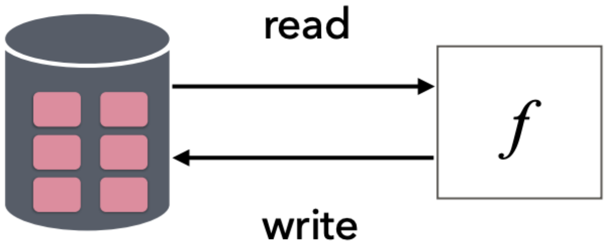
Batch processing

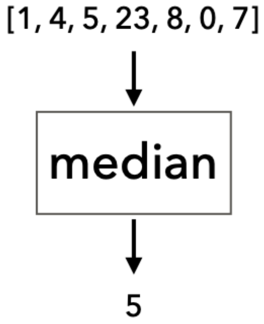


Stream processing

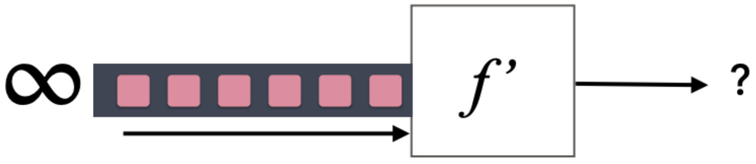
Batch Processing vs. Stream Processing

Complete data accessible in persistent storage



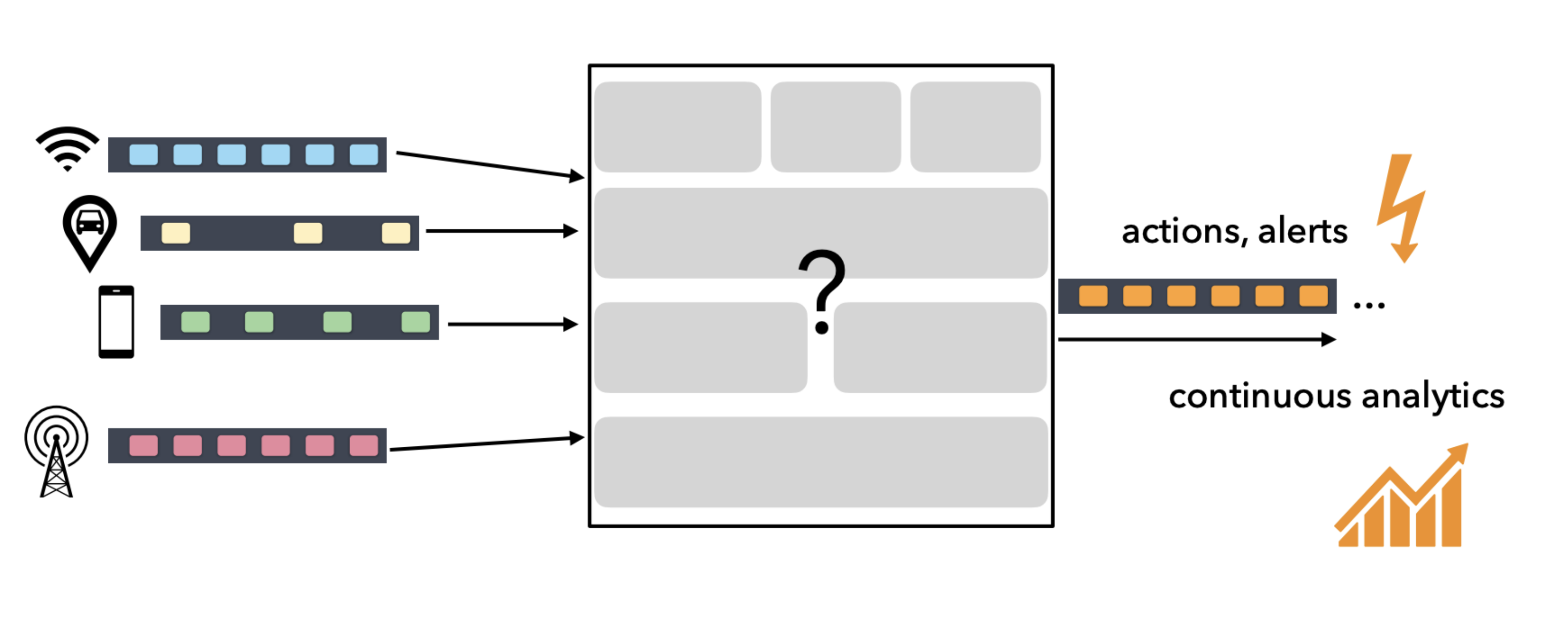


Continuously arriving, possibly unbounded data



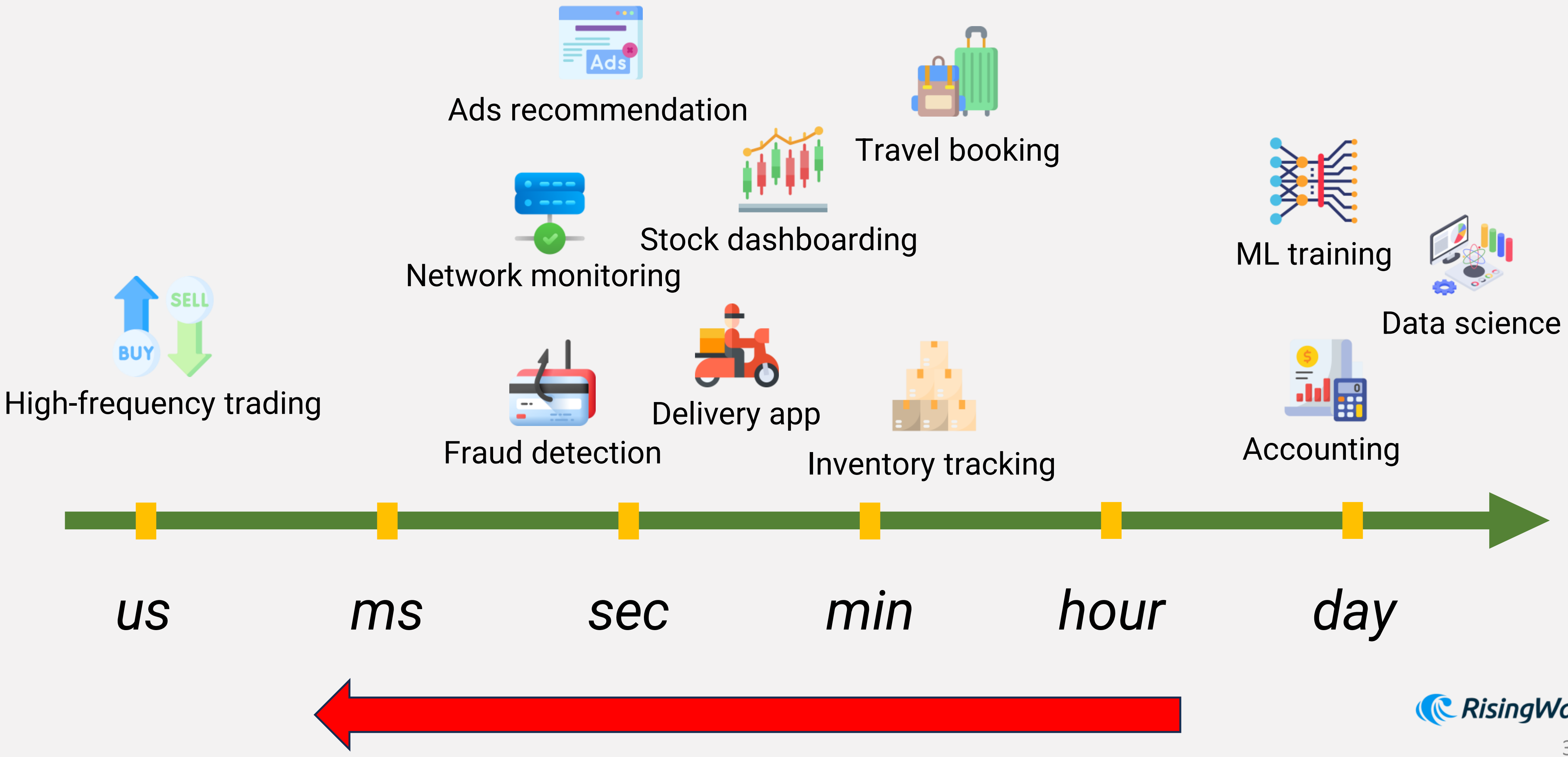
- We cannot store the entire stream
- No control over arrival rate or order

Batch Processing vs. Stream Processing



RisingWaveLabs

Stream Processing Use Cases



RisingWaveLabs

Stream Processing Use Cases

- Financial services
  - Payment services
    - Fraud detection
  - Capital markets (brokerage, hedge fund)
    - Compliance, risk control, pre-trade analytics, …

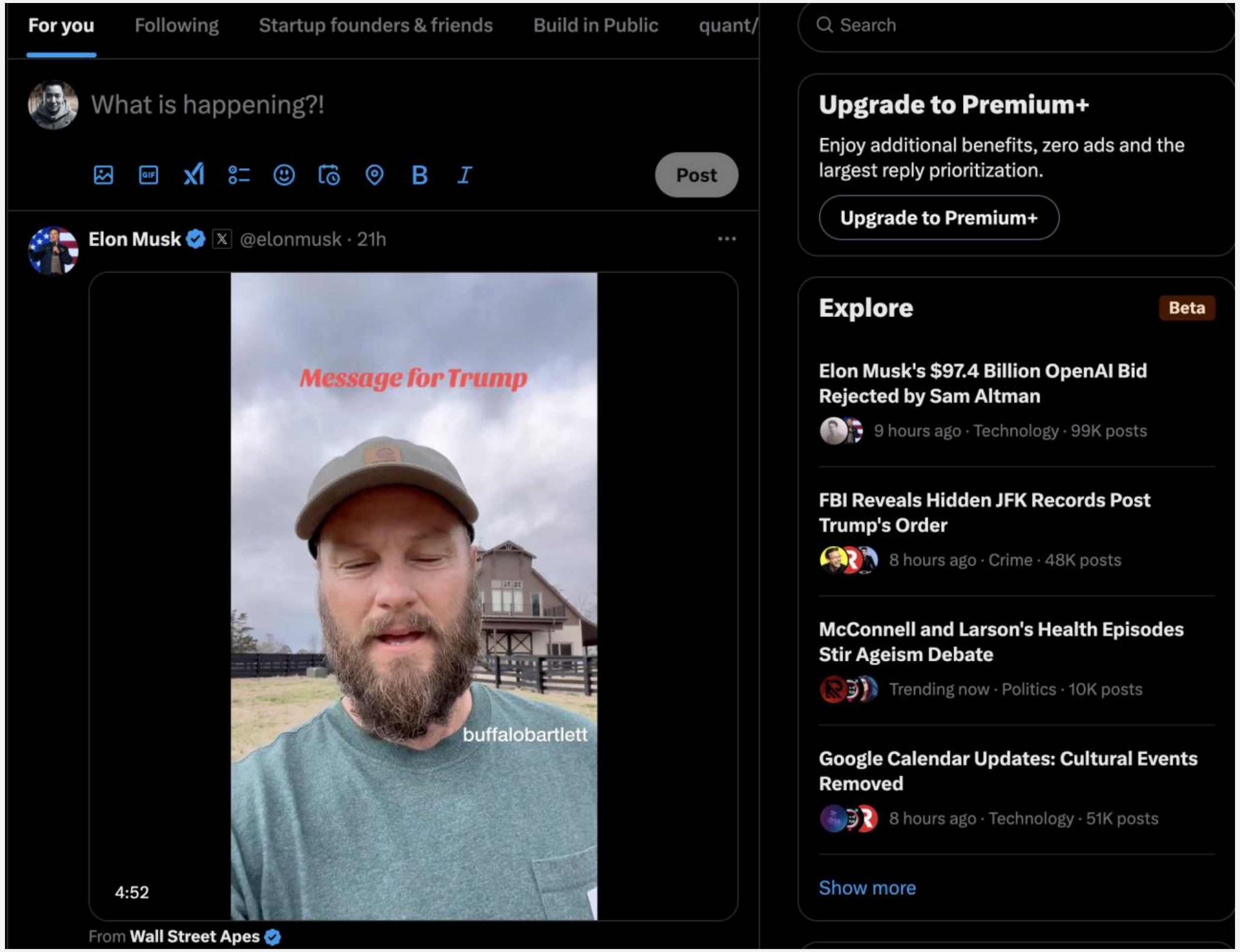
RisingWaveLabs

Stream Processing Use Cases

- Entertainment
  - Gaming
  - Sports betting
  - Publisher

Stream Processing Use Cases

- Entertainment
  - Gaming
  - Sports betting
  - Publisher



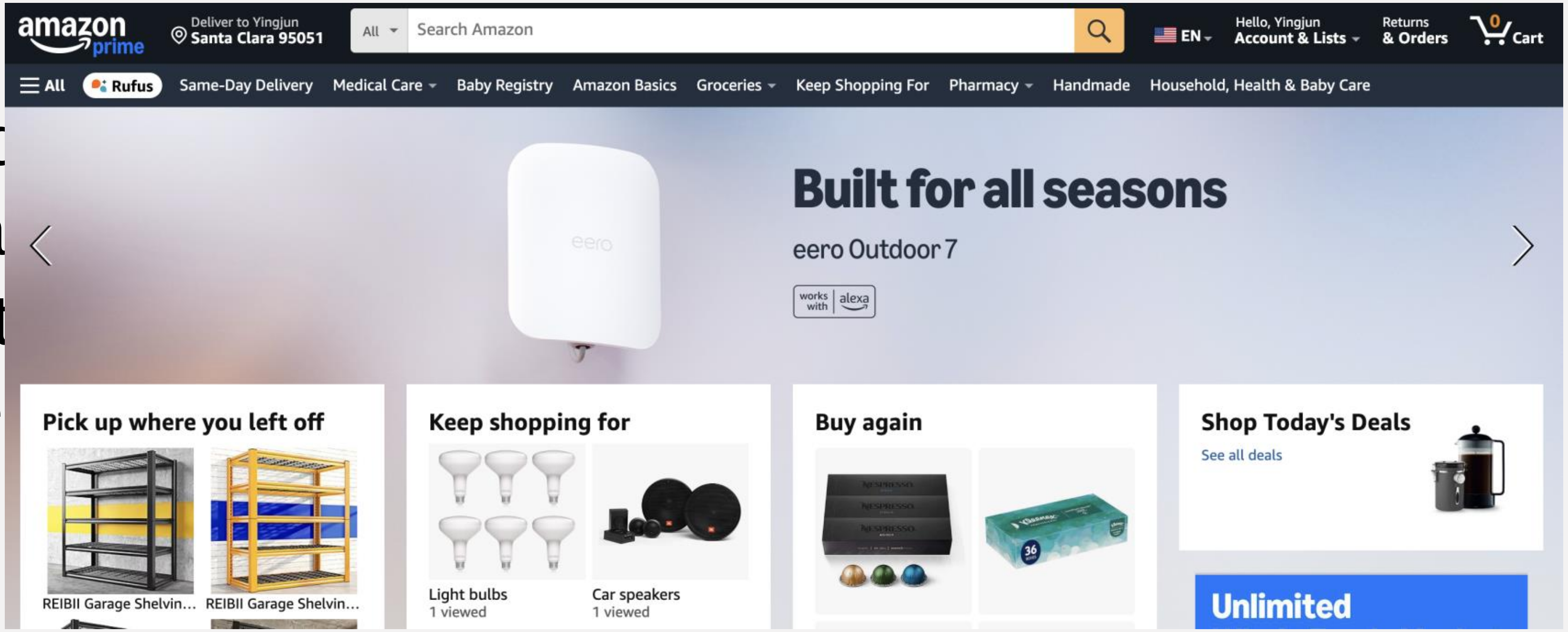
RisingWaveLabs

Stream Processing Use Cases

- E-commerce
  - Personalized recommendation
  - Price comparison
  - Fraud detection
  - Churn prevention and prediction

Stream Processing Use Cases

- E-commerce
  - Personalized
  - Price compa
  - Fraud detect
  - Churn preve



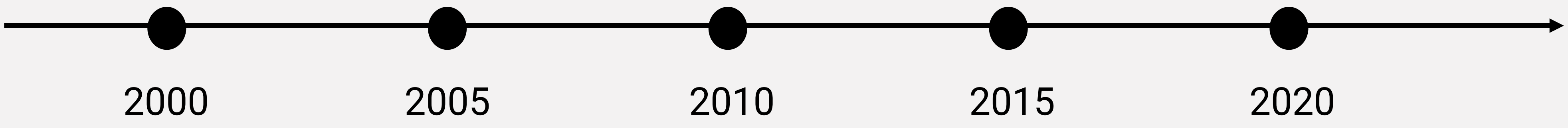
Stream Processing Use Cases

- Energy and manufacturing
- Logistics
- …

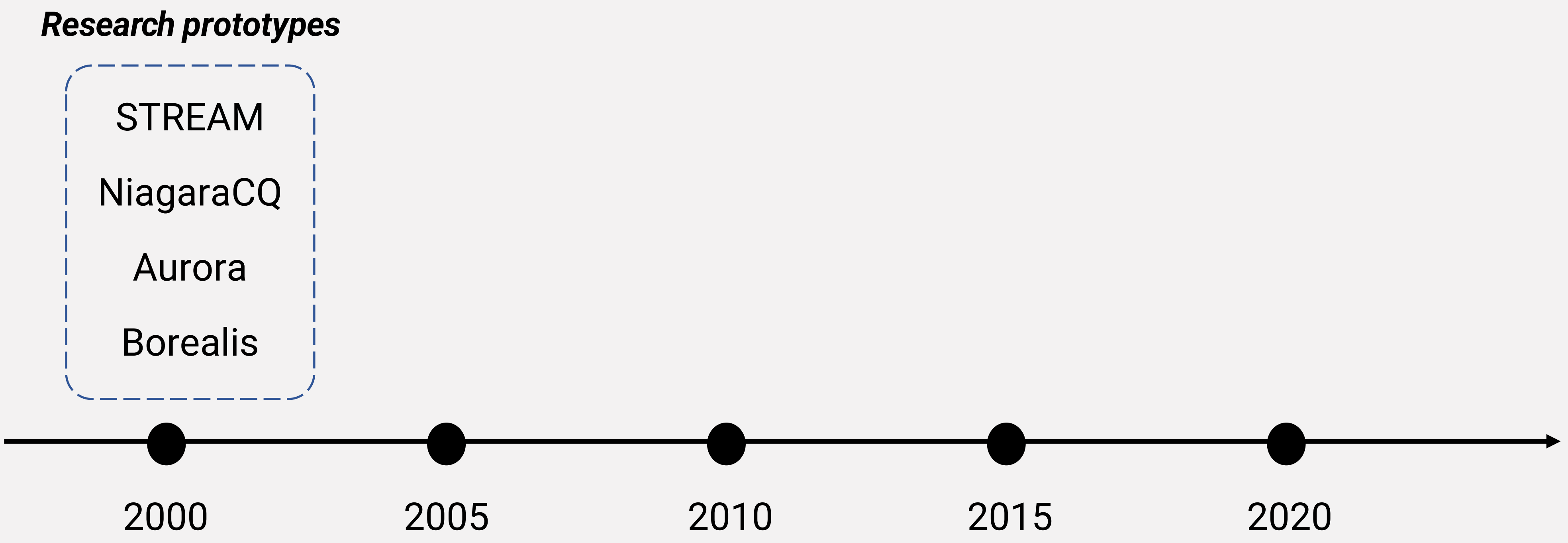
Stream Processing Use Cases

- Energy and manufacturing
- Logistics
- ...

History of Stream Processing Systems

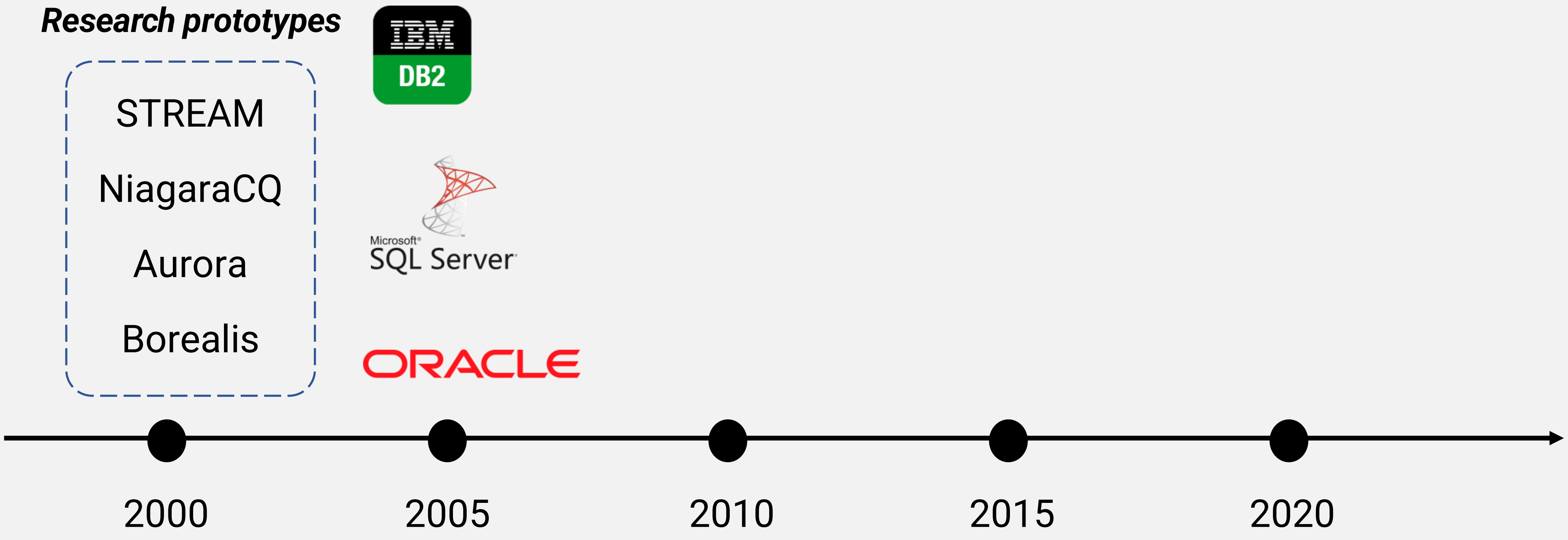


History of Stream Processing Systems

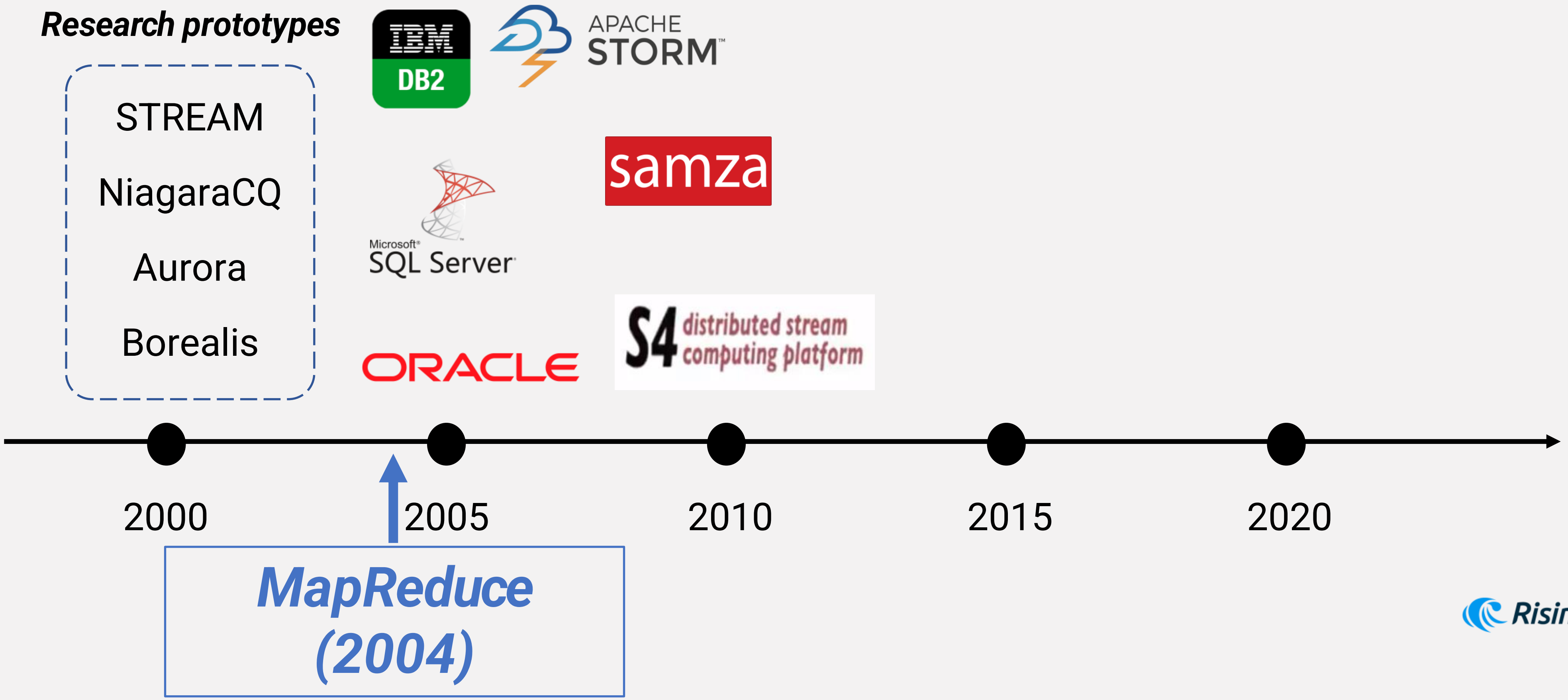


RisingWaveLabs

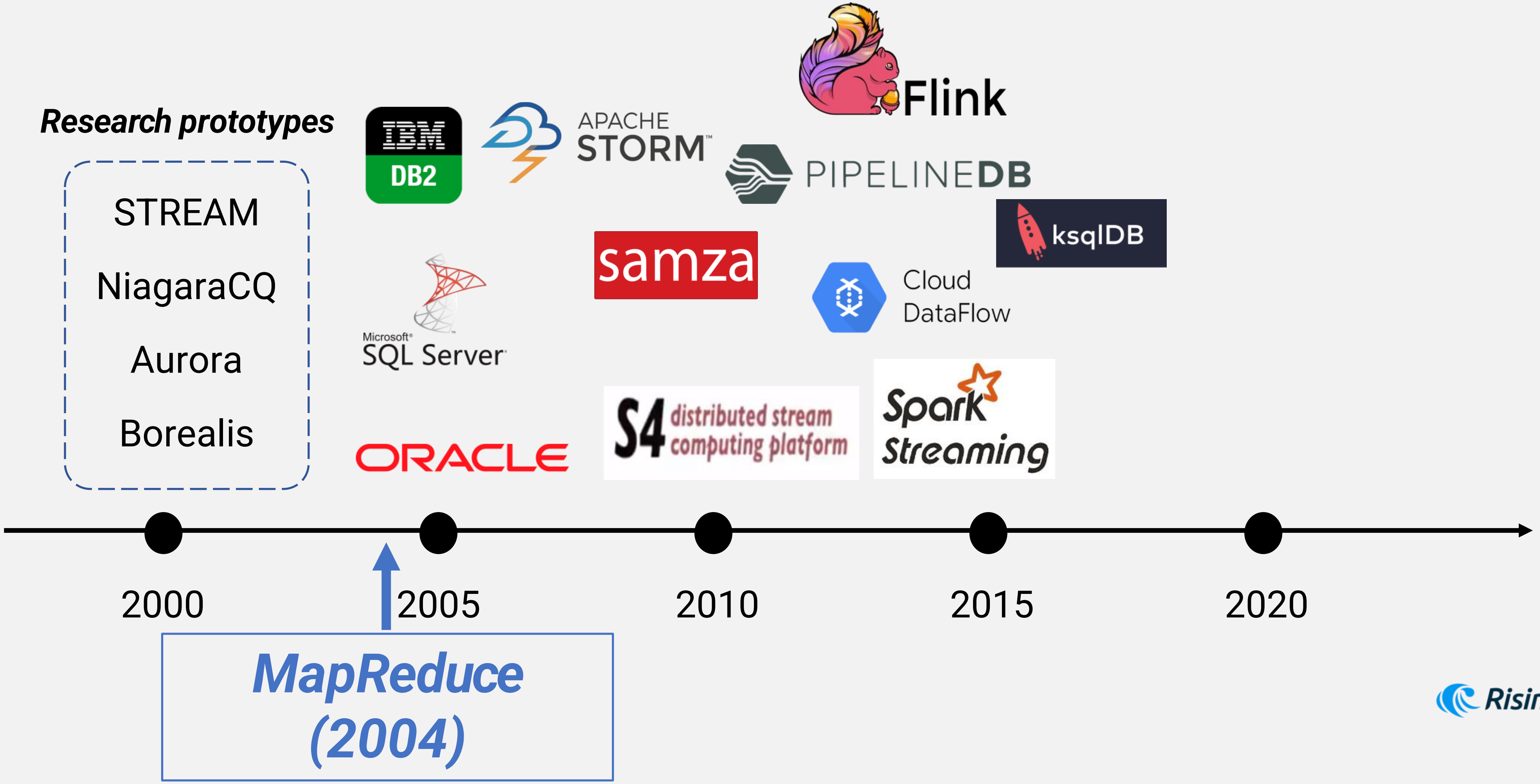
History of Stream Processing Systems



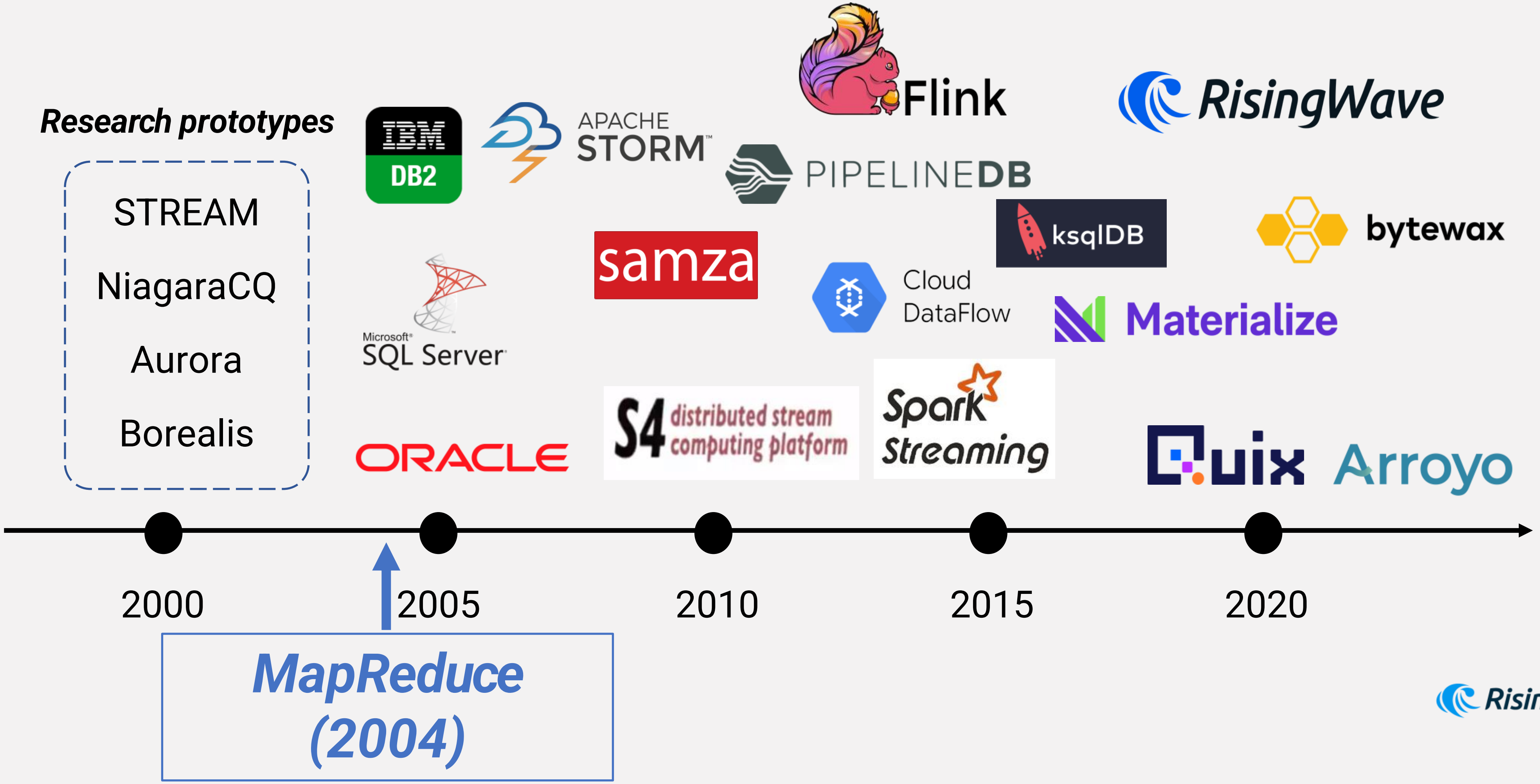
History of Stream Processing Systems



History of Stream Processing Systems



History of Stream Processing Systems



History of Stream Processing Systems

- Trend: Single node -> distributed -> cloud

Stream Processing Concepts (Boring Part!)

Stream Processing Concepts

- In traditional data processing applications, we know the entire dataset in advance, e.g. tables stored in a database.

Stream Processing Concepts

- In traditional data processing applications, we know the entire dataset in advance, e.g. tables stored in a database.
- A data stream is a data set that is produced incrementally over time, rather than being available in full before its processing begins.

Stream Processing Concepts

- In traditional data processing applications, we know the entire dataset in advance, e.g. tables stored in a database.
- A data stream is a data set that is produced incrementally over time, rather than being available in full before its processing begins.
- Data streams are high-volume, real-time data that might be unbounded
  - we cannot store the entire stream in an accessible way
  - we have to process stream elements on-the-fly using limited memory

Properties of Data Streams

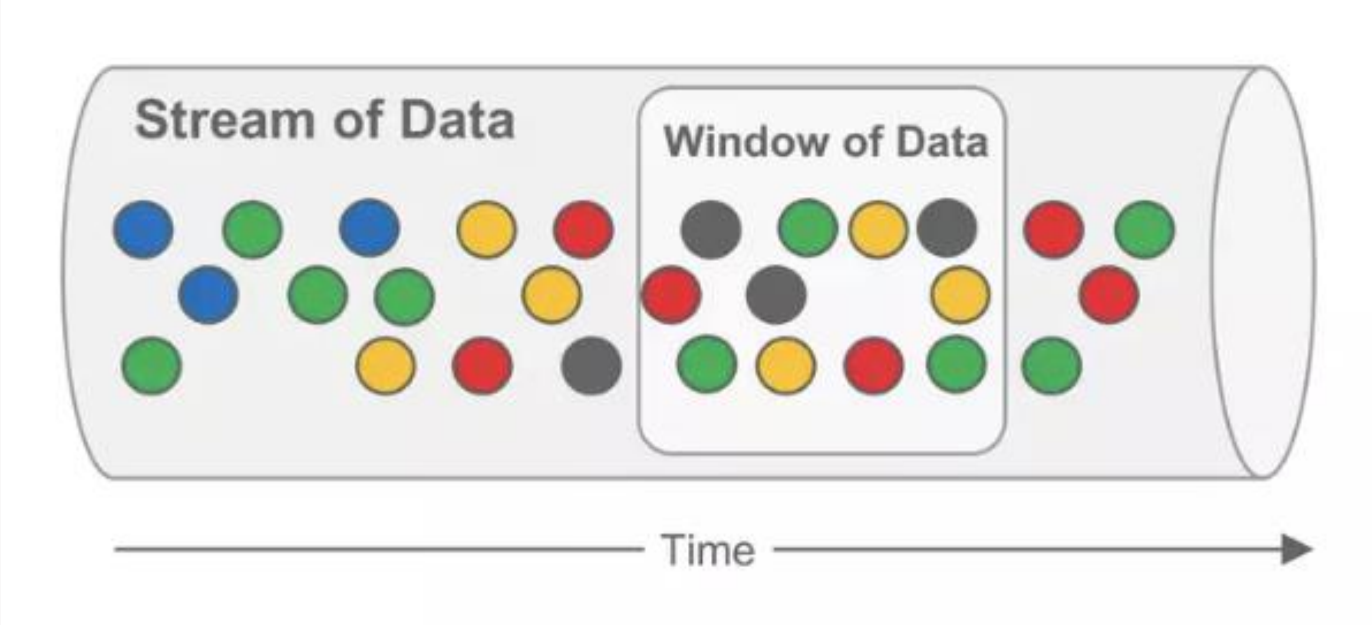
- They arrive continuously instead of being available a-priori.
- They bear an arrival and/or a generation timestamp.
- They are produced by external sources, i.e. the DSMS has no control over their arrival order or the data rate.
- They have unknown, possibly unbounded length, i.e. the DSMS does not know when the stream ends.

Two Important Concepts

- Time Windowing
  - Perform computation over a subset of data
- Watermark
  - Make sure order is guaranteed

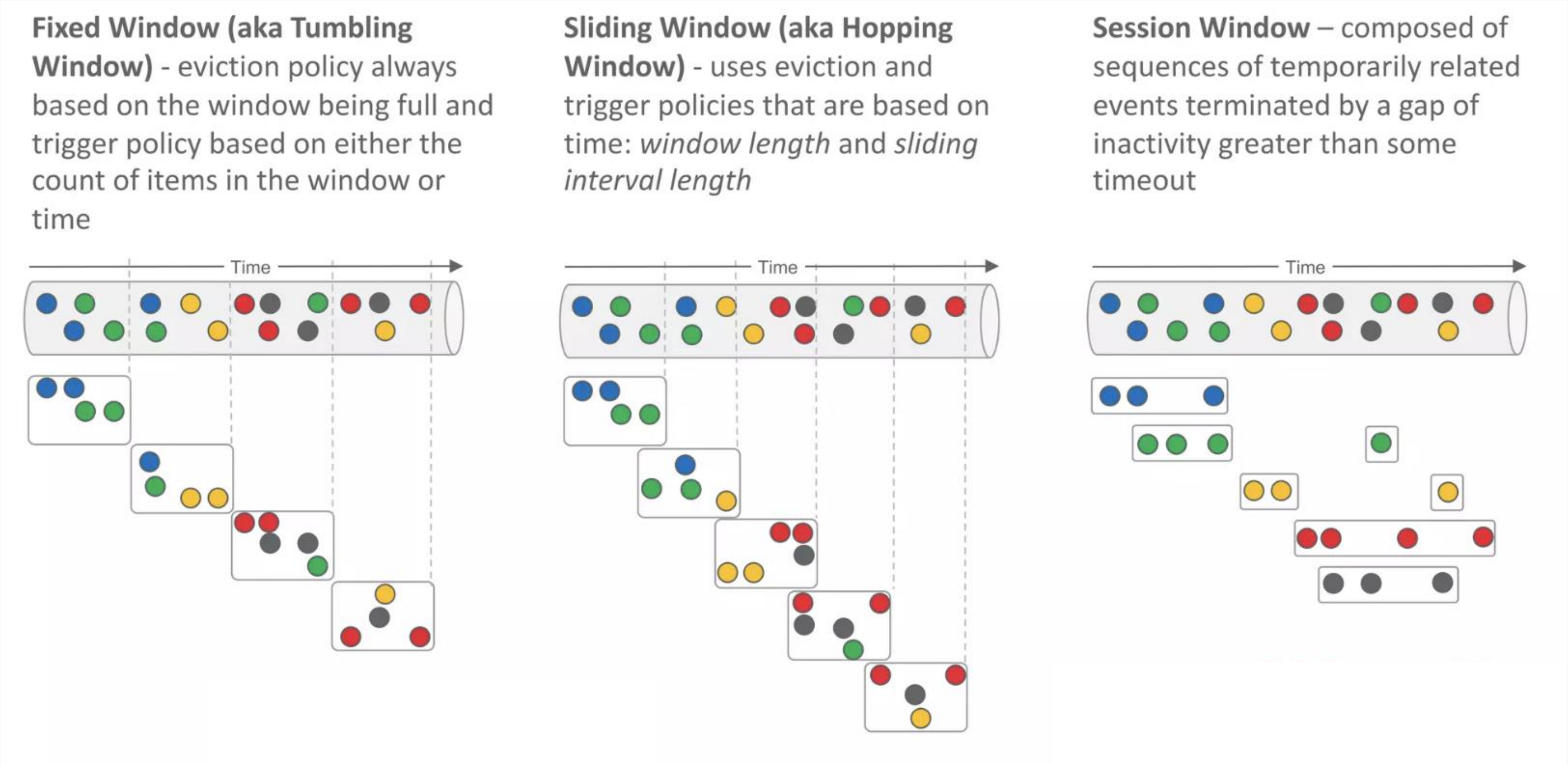
Time Windowing

- Data streams never end. We may want to compute on a subset of data.



Time Windowing

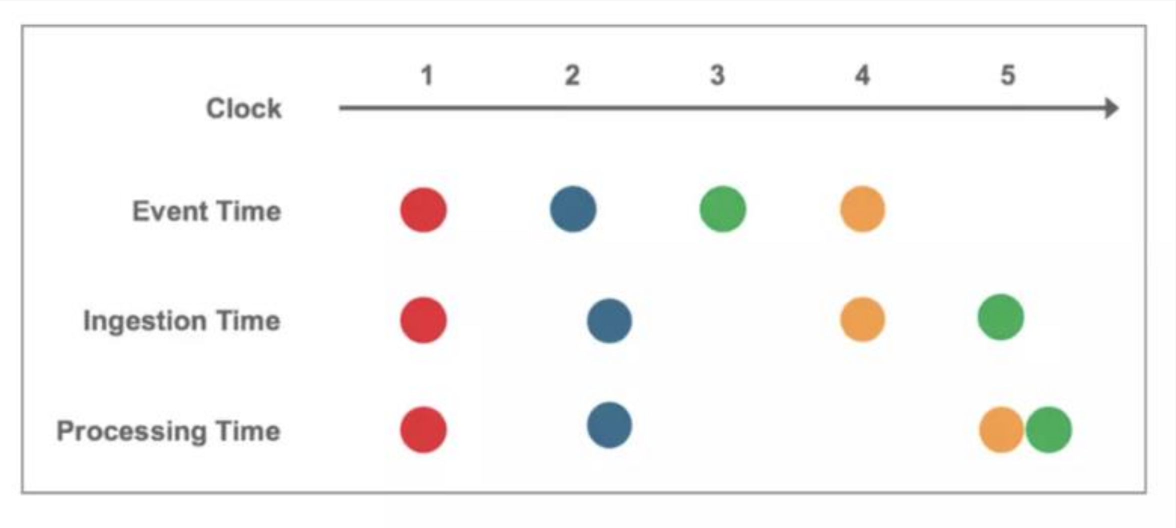
- Three types of windows



RisingWaveLabs

Watermarks

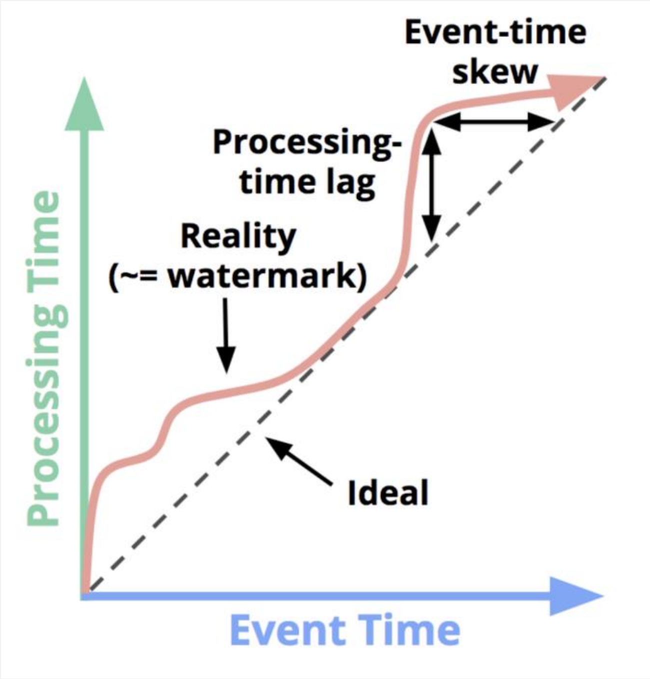
- Let’s talk about time first…
- Event time
  - the time at which events actually occurred
- Ingestion time / processing time
  - The time at which events are ingested into / processed by the system



RisingWaveLabs

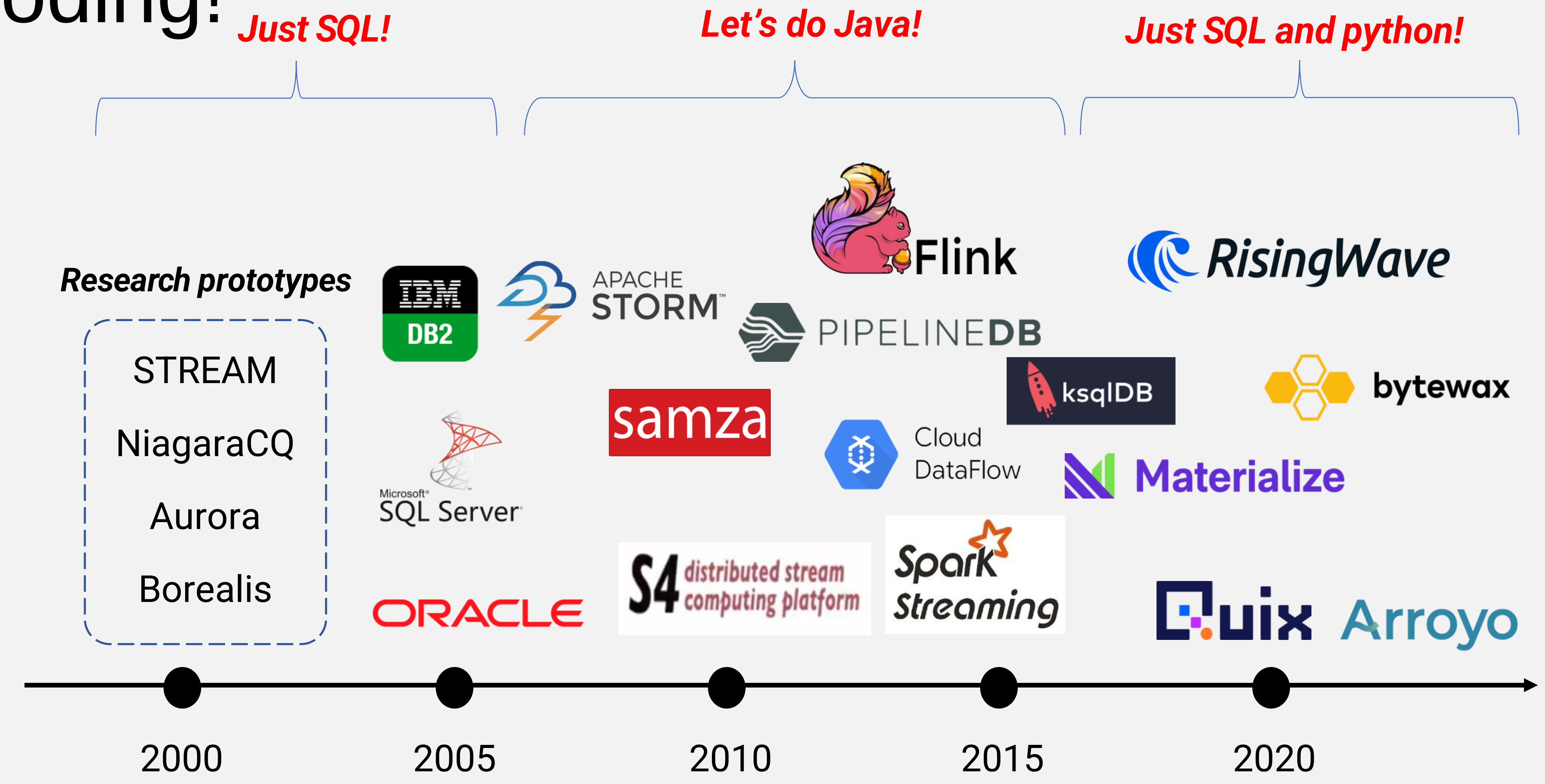
Watermarks

- It's likely that events are ingested into / processed by the system in an random order
- How to guarantee order? Well, let's use watermarks..



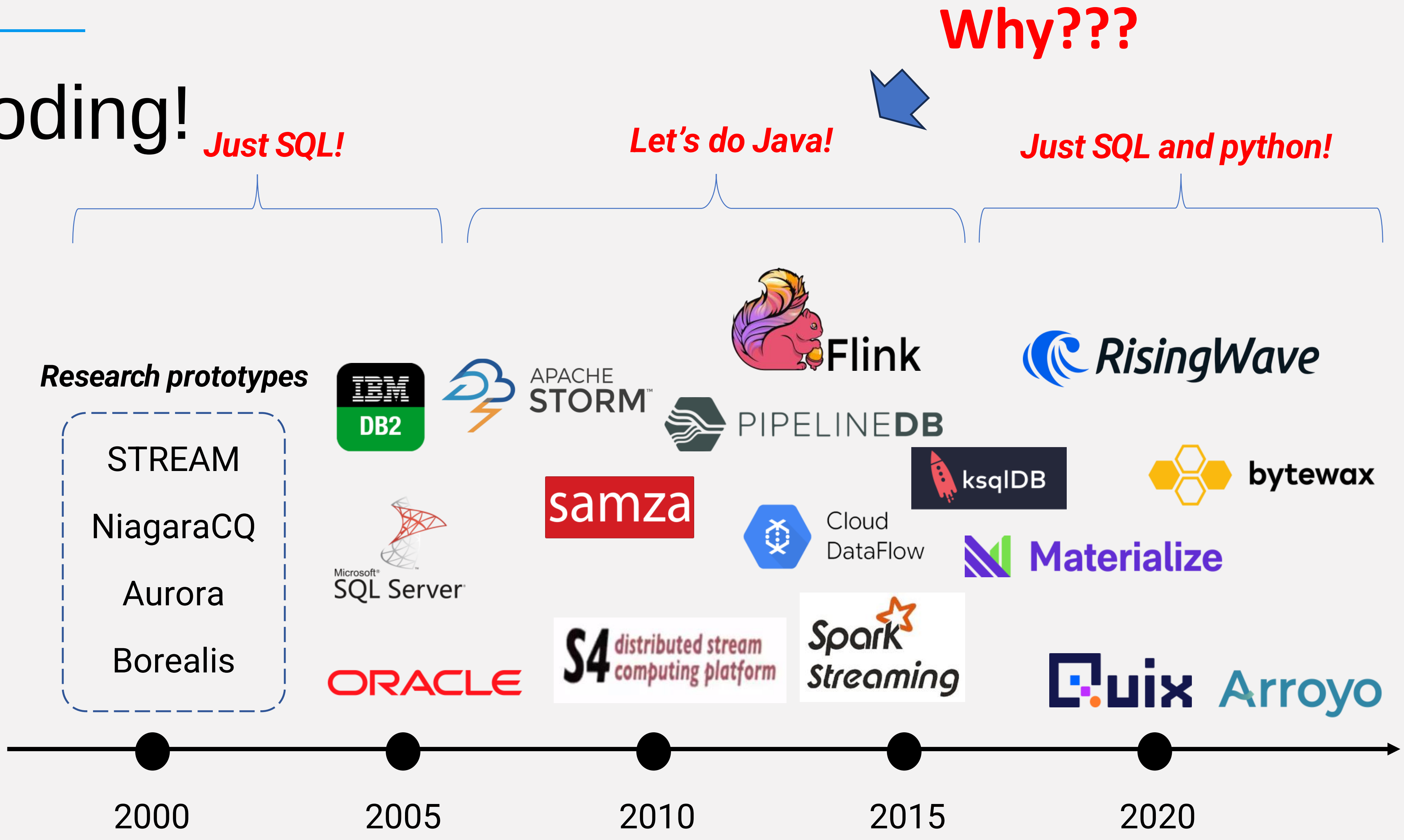
Coding!

Coding!



Just SQL!
Let's do Java!
Just SQL and python!
Research prototypes
STREAM
NiagaraCQ
Aurora
Borealis
IBM DB2
Microsoft SQL Server
ORACLE
APACHE STORM
samza
S4 distributed stream computing platform
Flink
PIPELINEDB
Cloud DataFlow
Spark Streaming
ksqlDB
Materialize
RisingWave
bytewax
Quix
Arroyo
2000
2005
2010
2015
2020

Coding!



Why???
Just SQL!
Let's do Java!
Just SQL and python!
Research prototypes
STREAM
NiagaraCQ
Aurora
Borealis
IBM DB2
Microsoft SQL Server
ORACLE
APACHE STORM
samza
S4 distributed stream computing platform
Flink
PIPELINEDB
Cloud DataFlow
Spark Streaming
ksqlDB
Materialize
RisingWave
bytewax
Quix
Arroyo
2000
2005
2010
2015
2020

MapReduce!

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

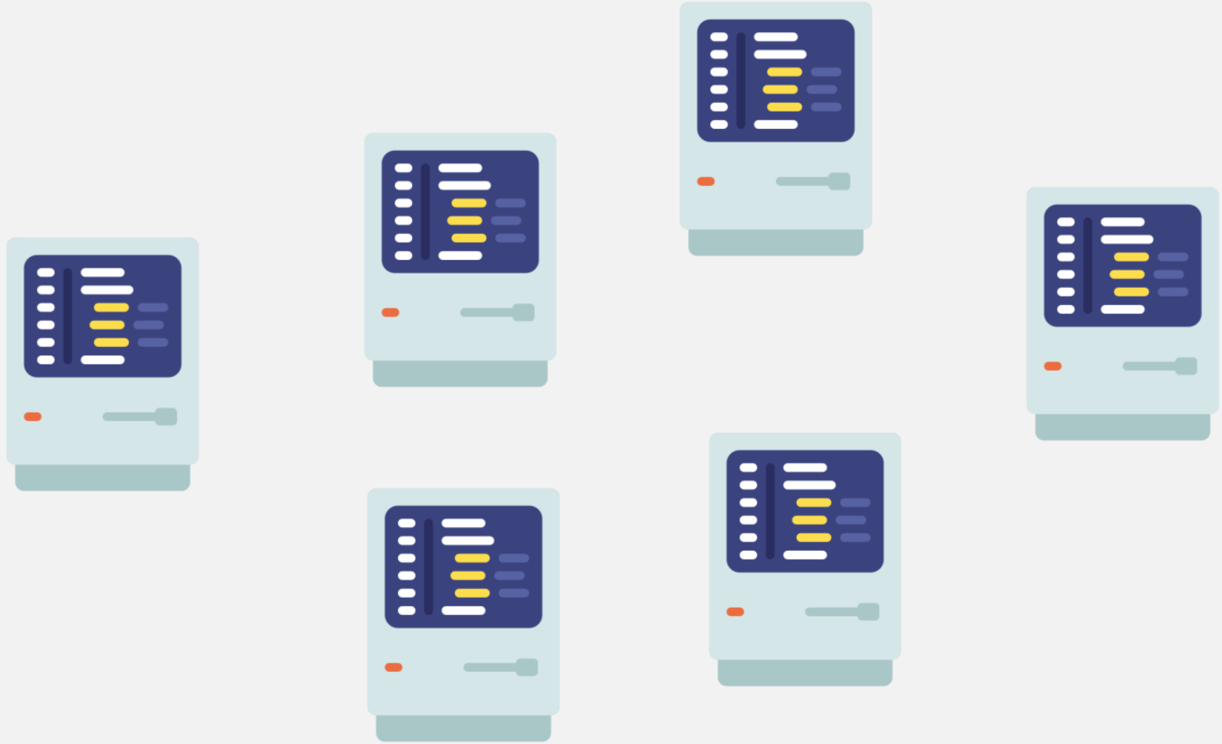
Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

RisingWaveLabs

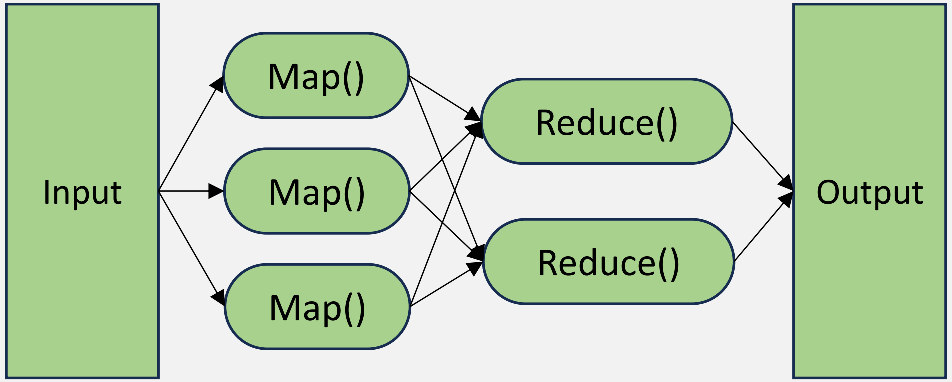
Revisiting MapReduce

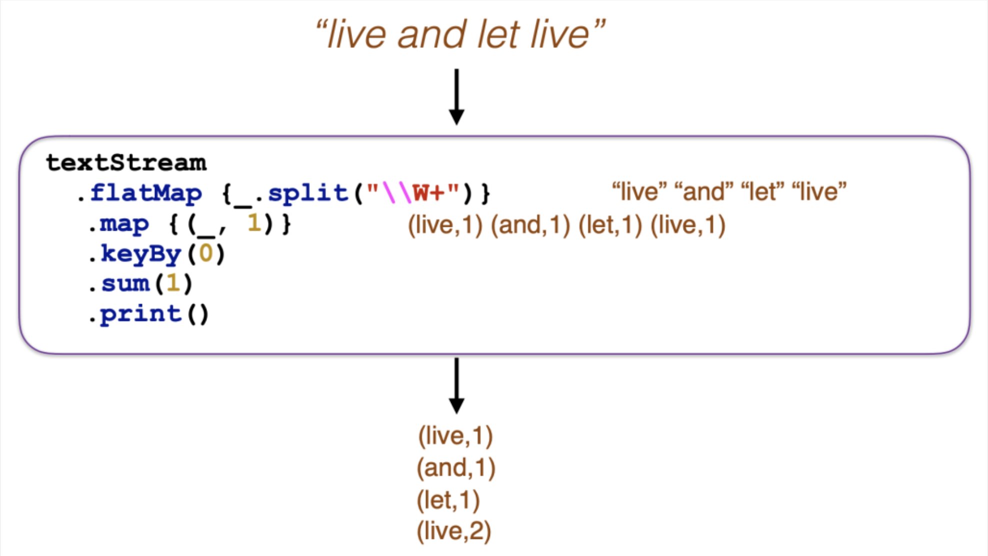
- Scale computation in commodity machines

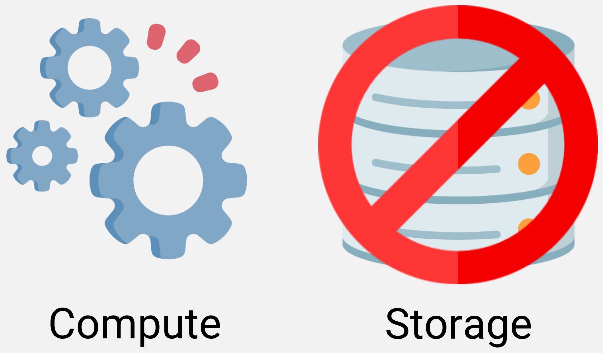


Revisiting MapReduce

- Scale computation in commodity machines
- Ideas:
  - Expose low-level APIs
  - Give up control over storage







Revisiting MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

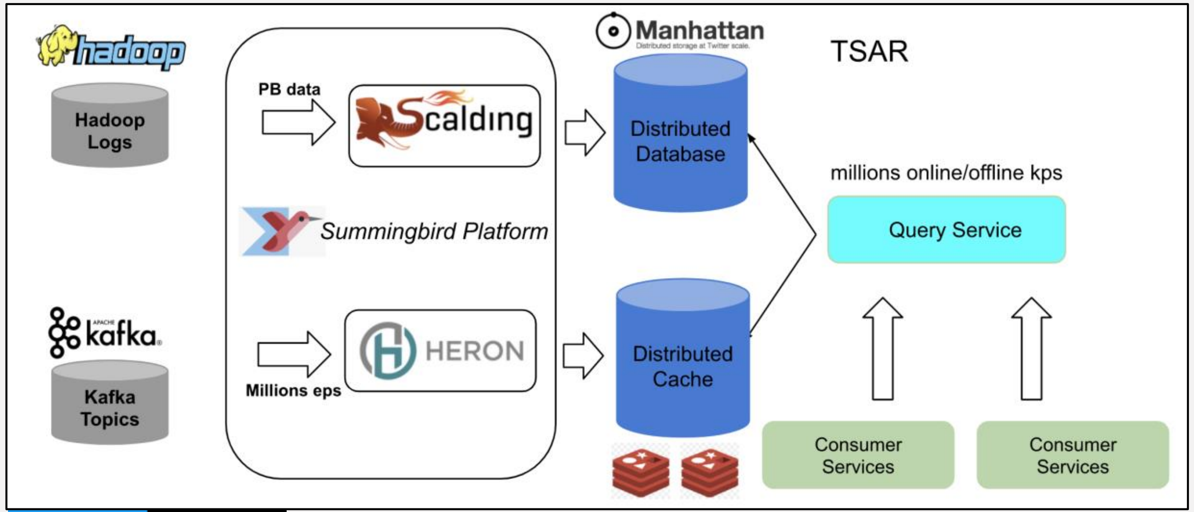
Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

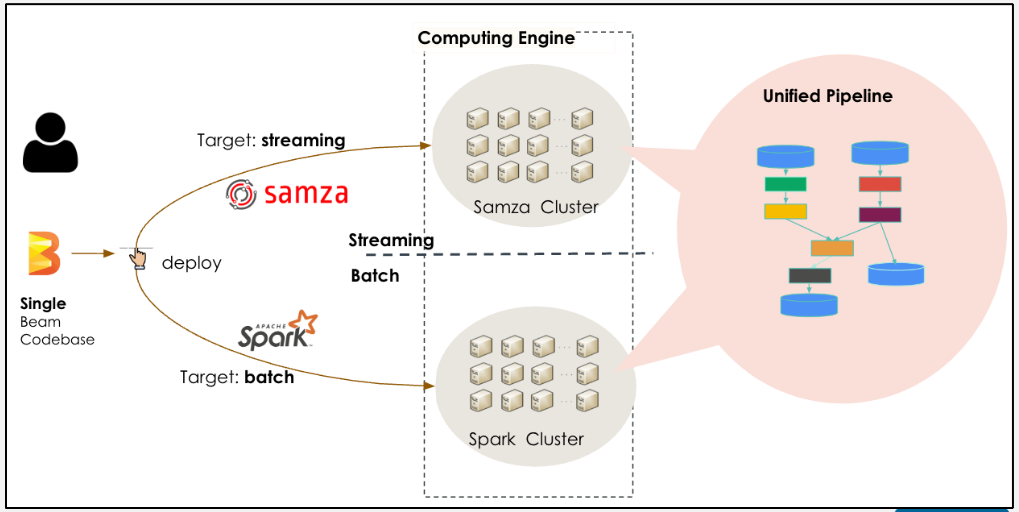
To appear in OSDI 2004

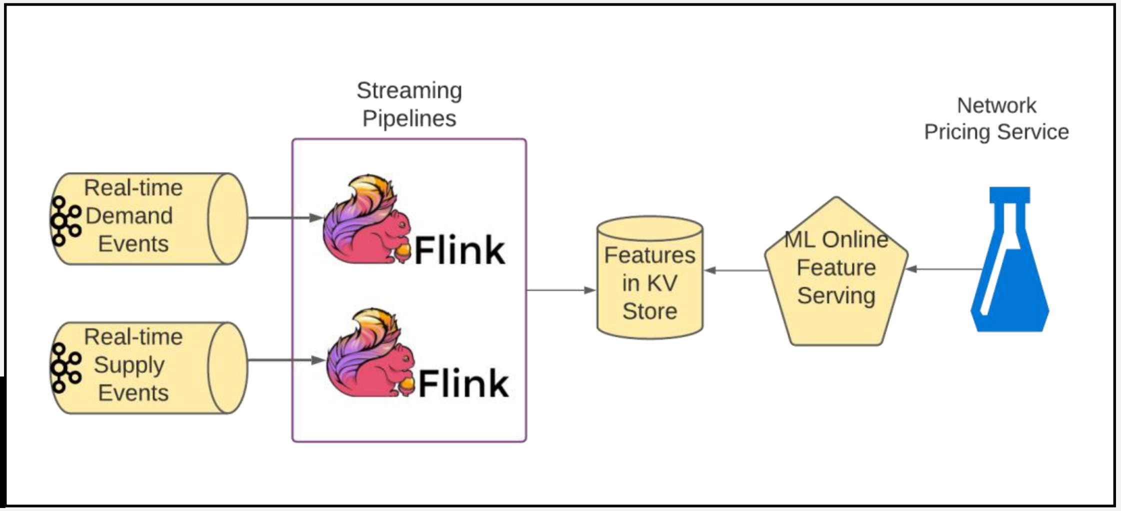
Batch

Streaming

Revisiting MapReduce





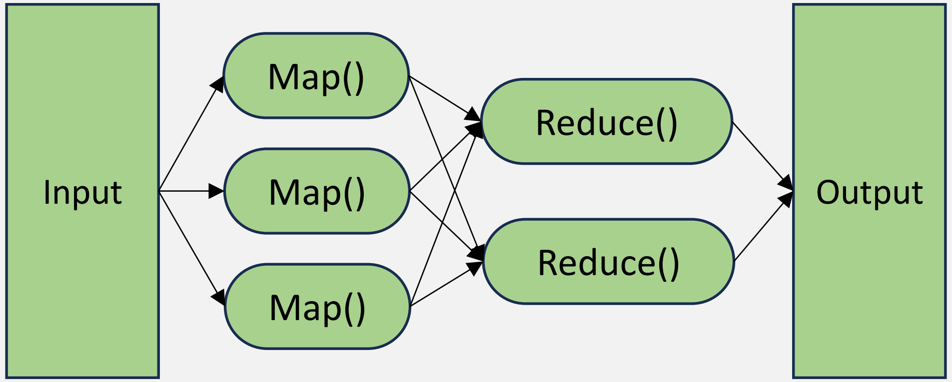


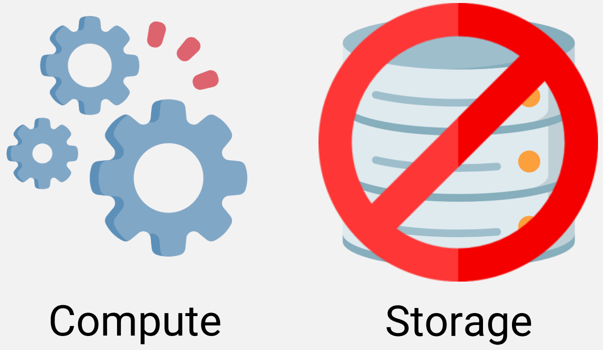
Why Switched Back to SQL Databases?

- Cost! Cost! Cost!

Revisiting MapReduce

- Scale computation in commodity machines
- Ideas:
  - Expose low-level APIs
  - Give up control over storage

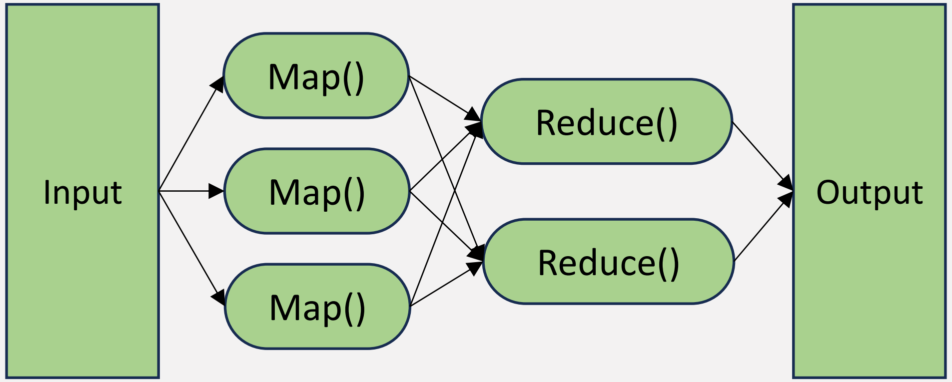


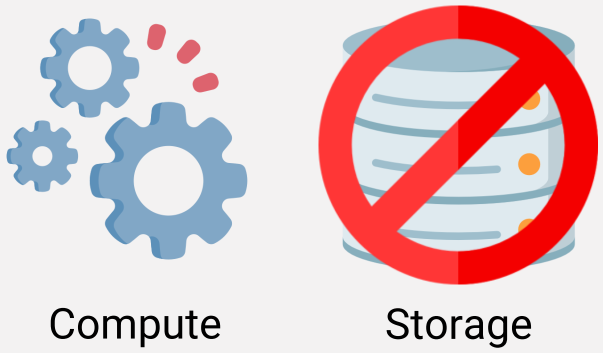


RisingWaveLabs

Revisiting MapReduce

- Scale computation in commodity machines
- Ideas:
  - Expose low-level APIs
  - Give up control over storage
- Tradeoff:
  - Learning curve
  - Efficiency
  - Development difficulty
  - Data stack complexity

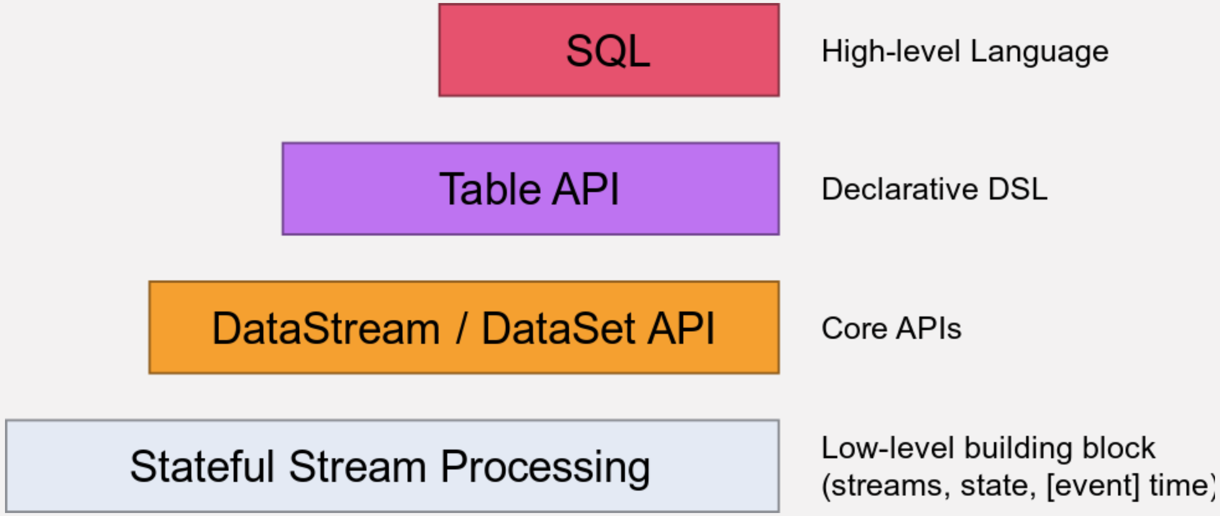




RisingWaveLabs

Limitations of Stream Processing Engines

- Learning curve
  - System-specific interfaces



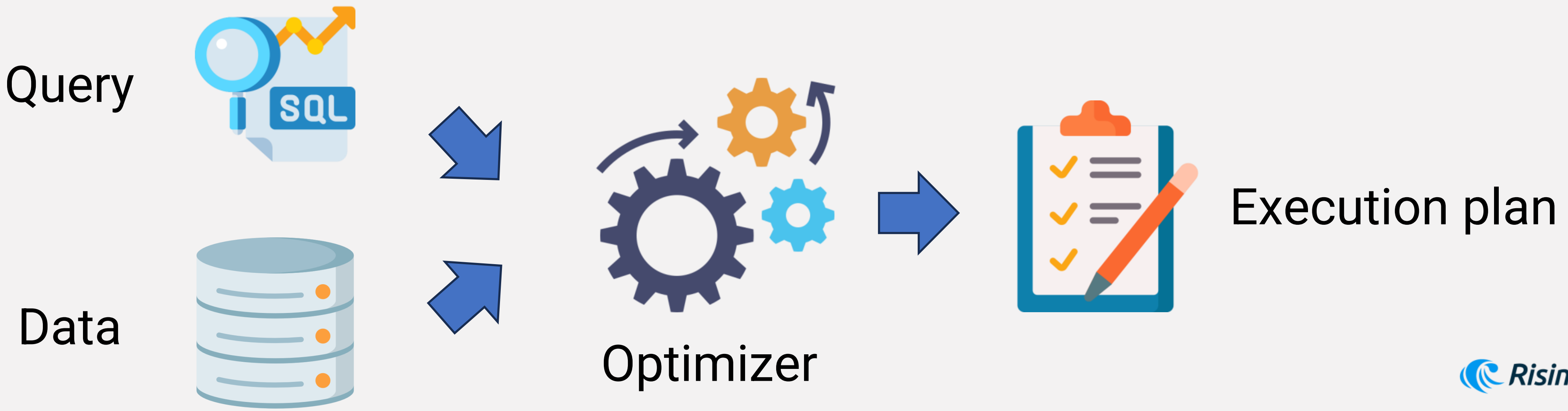
RisingWaveLabs

Limitations of Stream Processing Engines

- Learning curve
  - System-specific interfaces
- Efficiency
  - Hard to get optimal efficiency

Limitations of Stream Processing Engines

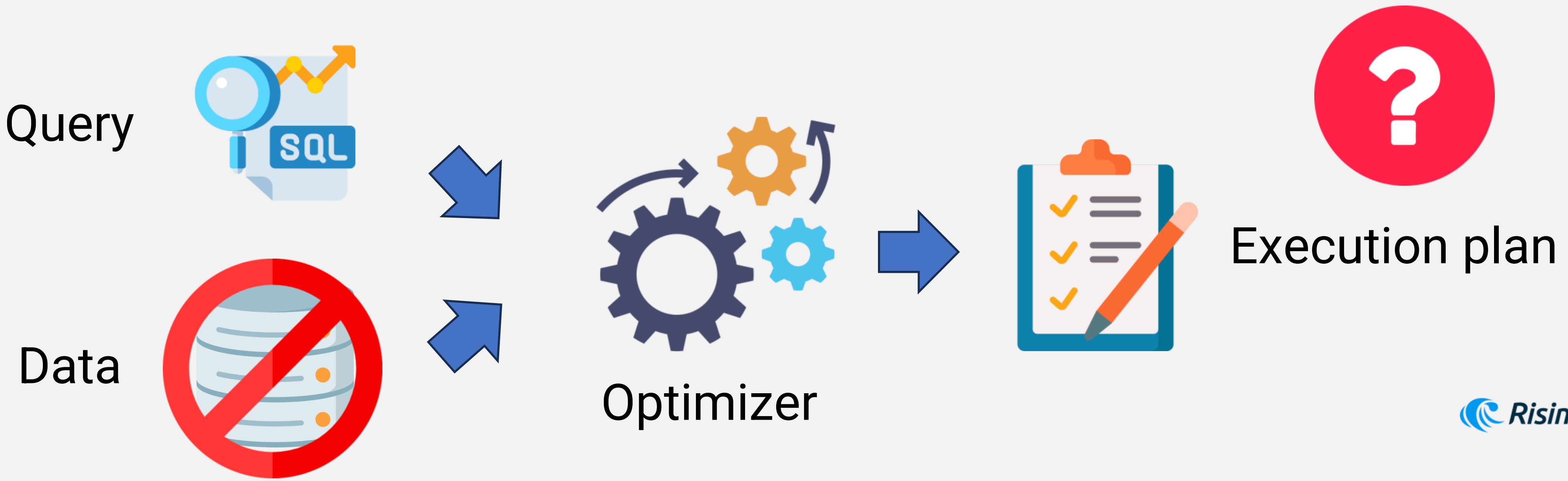
- Learning curve
  - System-specific interfaces
- Efficiency
  - Hard to get optimal efficiency



RisingWaveLabs

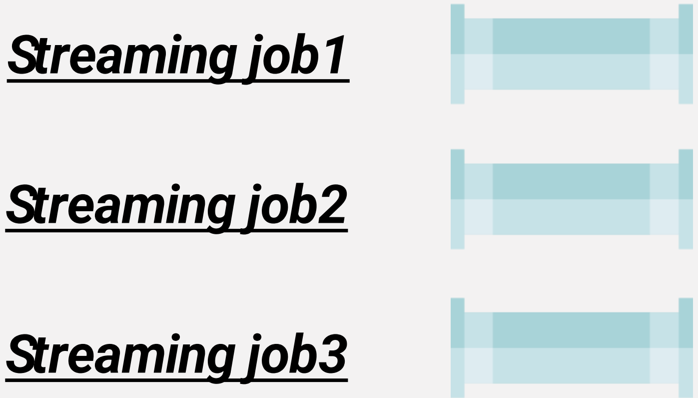
Limitations of Stream Processing Engines

- Learning curve
  - System-specific interfaces
- Efficiency
  - Hard to get optimal efficiency



Limitations of Stream Processing Engines

- Learning curve
  - System-specific interfaces
- Efficiency
  - Hard to get optimal efficiency
- Development difficulty
  - Difficult to verify correctness



Limitations of Stream Processing Engines

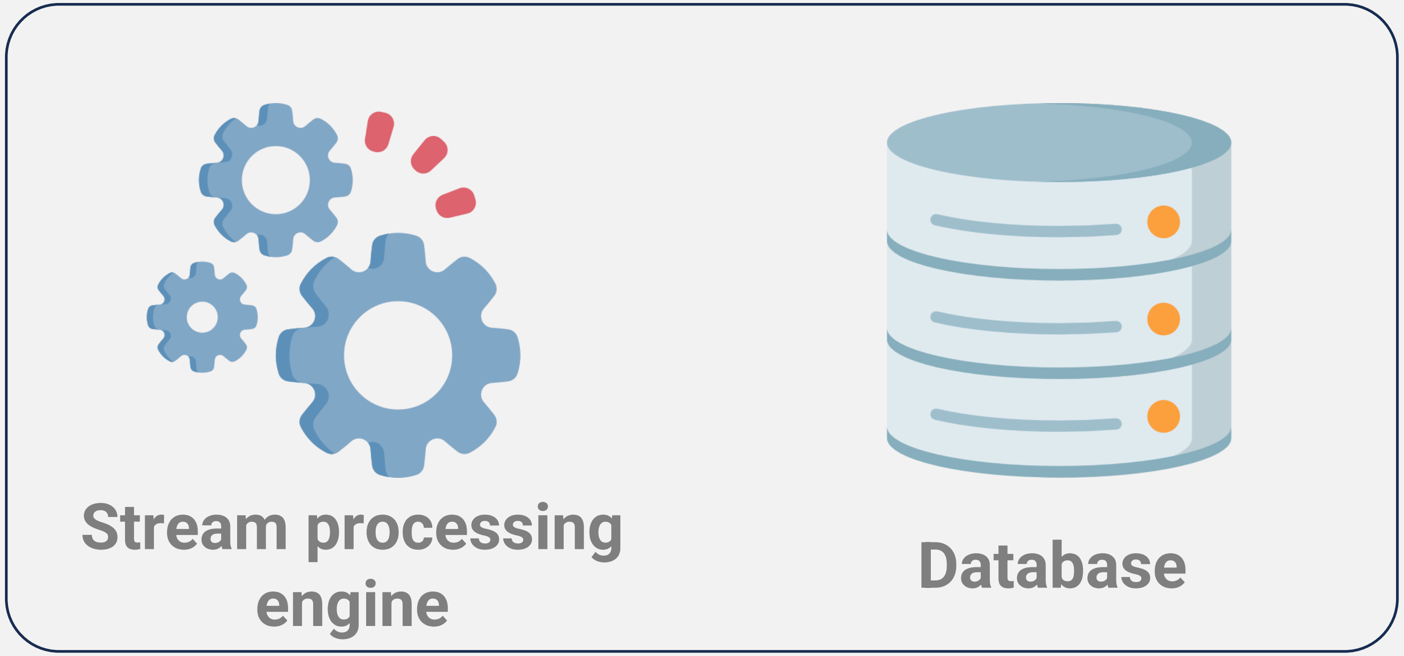
- Learning curve
  - System-specific interfaces
- Efficiency
  - Hard to get optimal efficiency
- Development difficulty
  - Difficult to verify correctness
- Data stack complexity
  - “Bring your own storage”

Limitations of Stream Processing Engines

- Streaming analytics
  - Monitoring, alerting, automation, etc…

Streaming Databases

- Get the best of ***both worlds***!



Streaming Database

Streaming Databases

- Learning curve
  - System-specific interfaces ➡ **Standard SQL!**

Streaming Databases

- Learning curve
  - System-specific interfaces ➡ **Standard SQL!**
- Efficiency
  - Hard to get optimal efficiency ➡ **Highly efficient!**

Streaming Databases

- Learning curve
  - System-specific interfaces → **Standard SQL!**
- Efficiency
  - Hard to get optimal efficiency → **Highly efficient!**
- Development difficulty
  - Difficulty to verify correctness → **Composable code!**

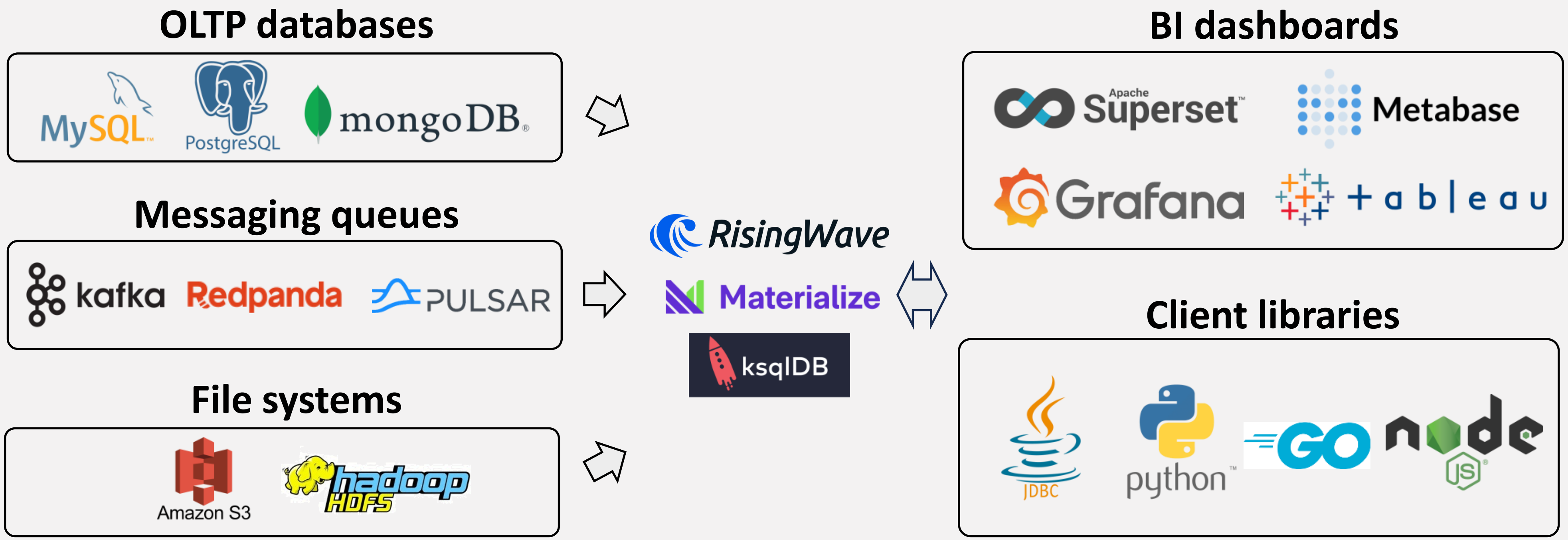
RisingWaveLabs

Streaming Databases

- Learning curve
  - System-specific interfaces → **Standard SQL!**
- Efficiency
  - Hard to get optimal efficiency → **Highly efficient!**
- Development difficulty
  - Difficulty to verify correctness → **Composable code!**
- Data stack complexity
  - "Bring your own storage" → **One single system!**

Streaming Databases in Production

- Streaming analytics
  - Monitoring, alerting, automation, etc…

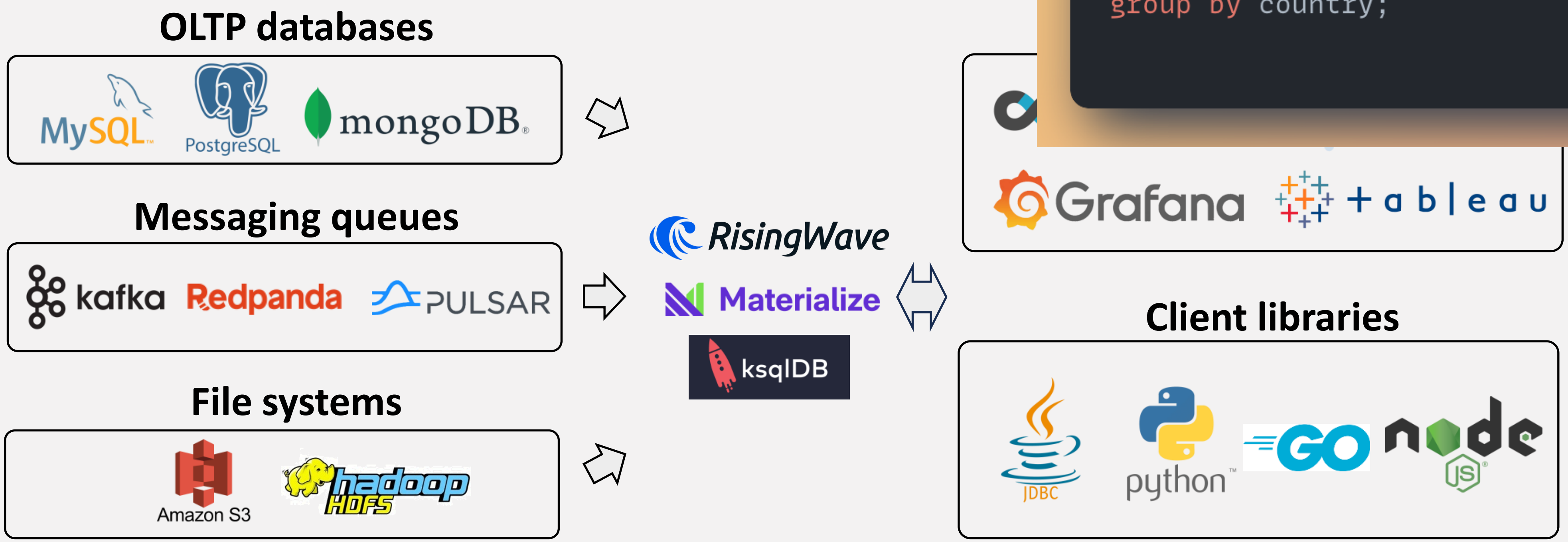


RisingWaveLabs

Streaming Databases in Production

- Streaming analytics
  - Monitoring, alerting, automation, etc

```sql
create materialized view my_mv as
select count(*) from Customers
group by country;
```



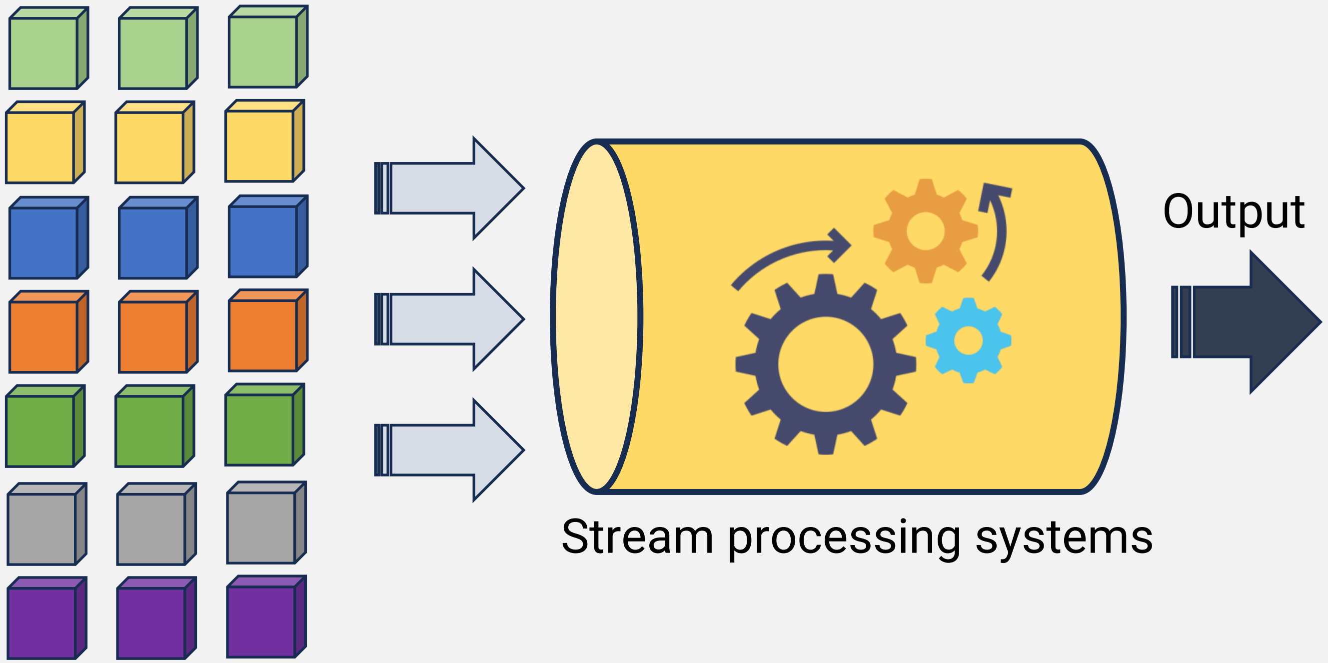
State Management (Deeply Technical)

RisingWaveLabs

State Management

State Management

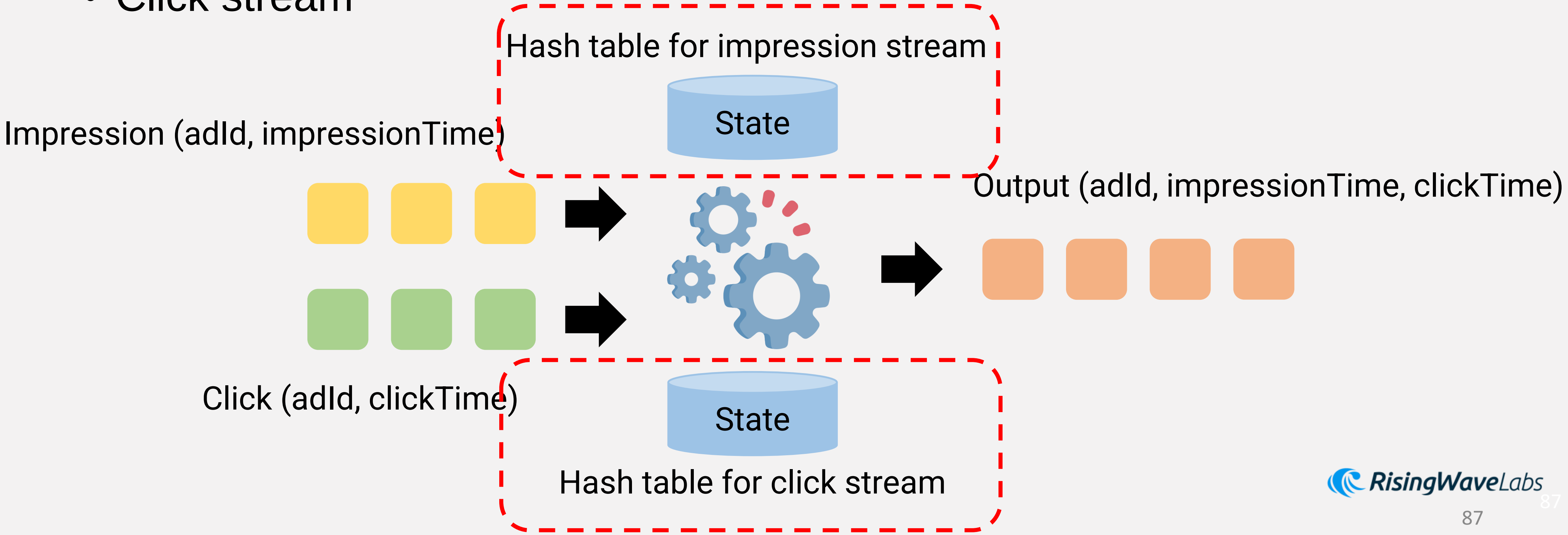
- Supporting stateful computations can be very challenging
  - Computation logics can be complicated
  - Streaming data workload may fluctuate



State Management

- Consider joining two data streams
  - Impression stream
  - Click stream

How to manage internal states?

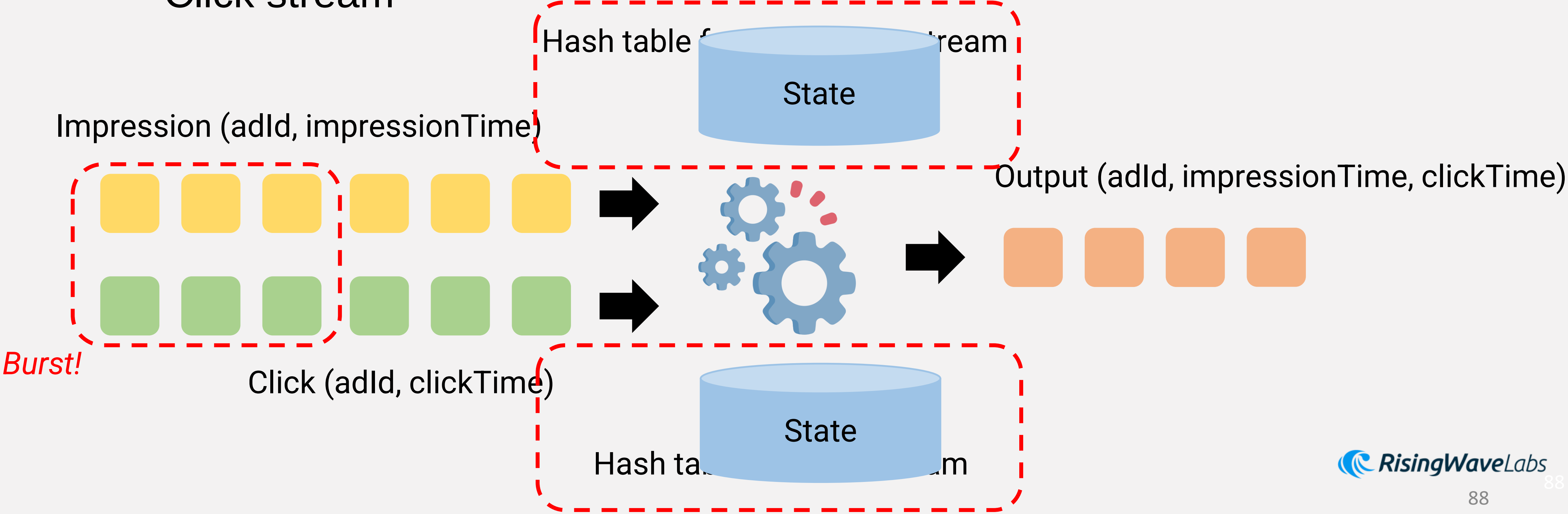


RisingWaveLabs

State Management

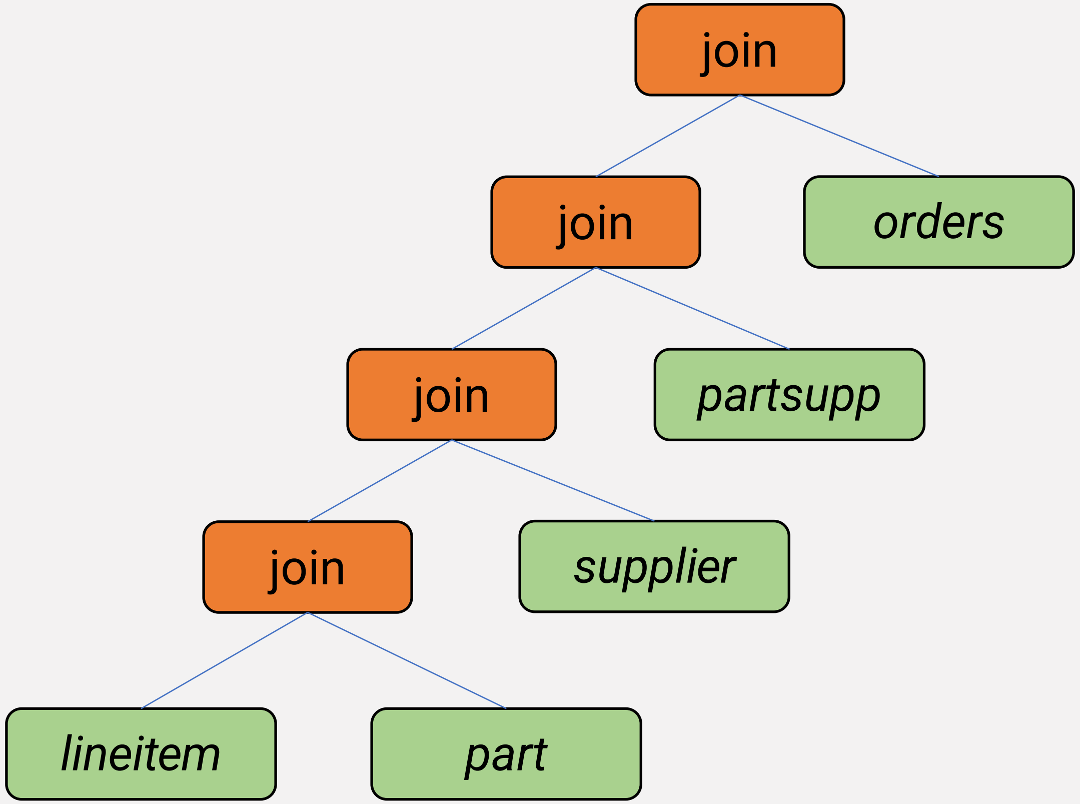
- Consider joining two data streams
  - Impression stream
  - Click stream

How to manage internal states?



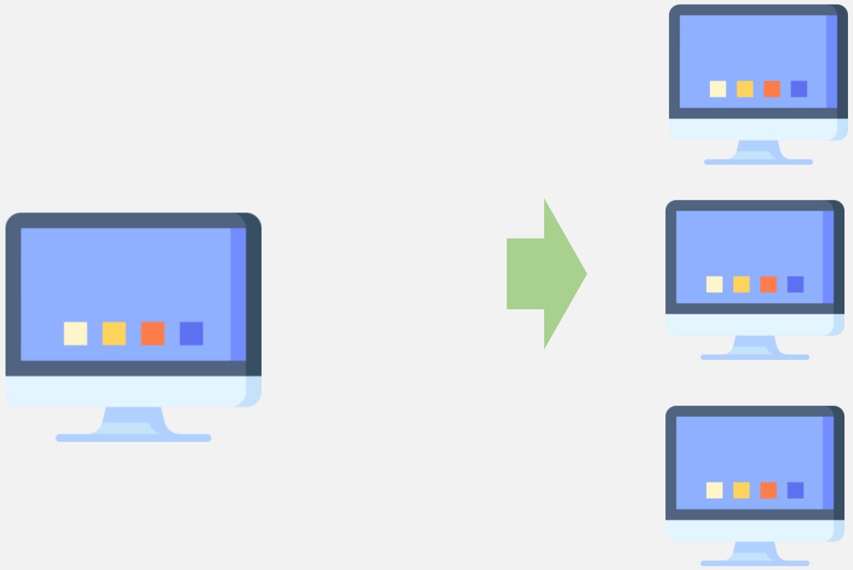
State Management

- Joining multiple data streams can be much harder than joining two data streams



State Management: MapReduce-Style

- MapReduce style
- Compute-storage coupled



Flink

Spark Streaming

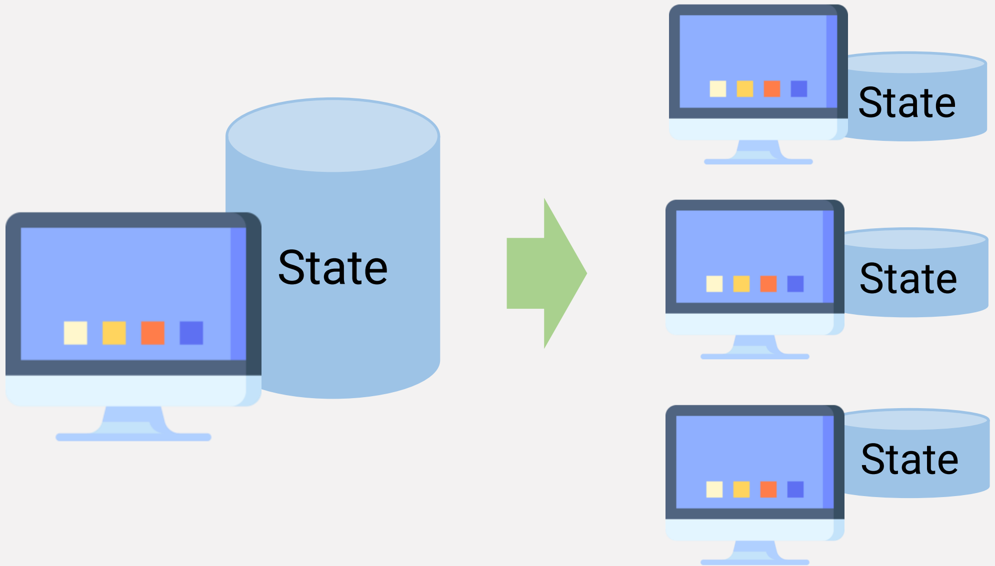
APACHE STORM

samza

RisingWaveLabs

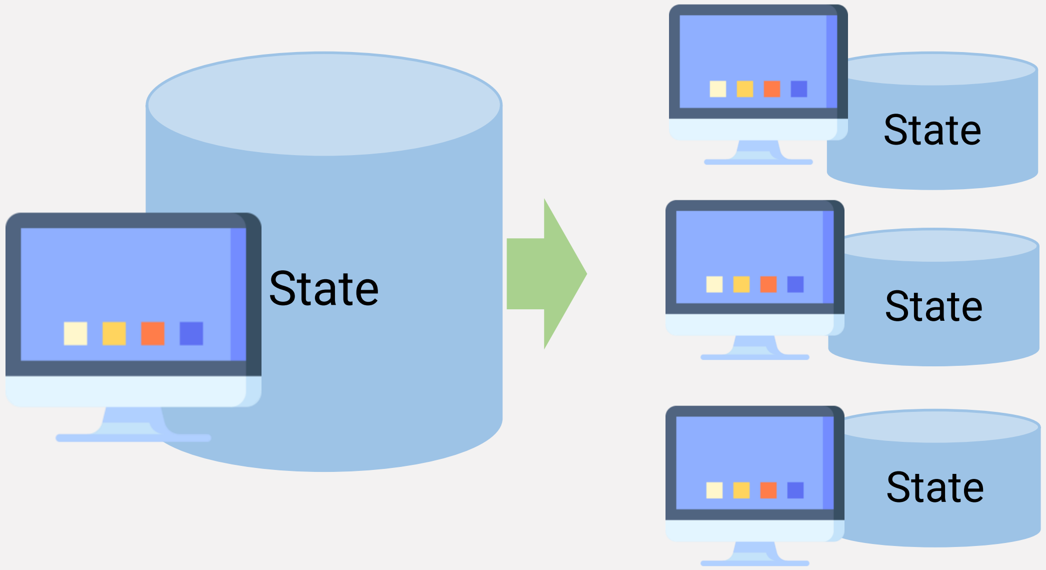
State Management: MapReduce-Style

- MapReduce style
- Compute-storage coupled



State Management: MapReduce-Style

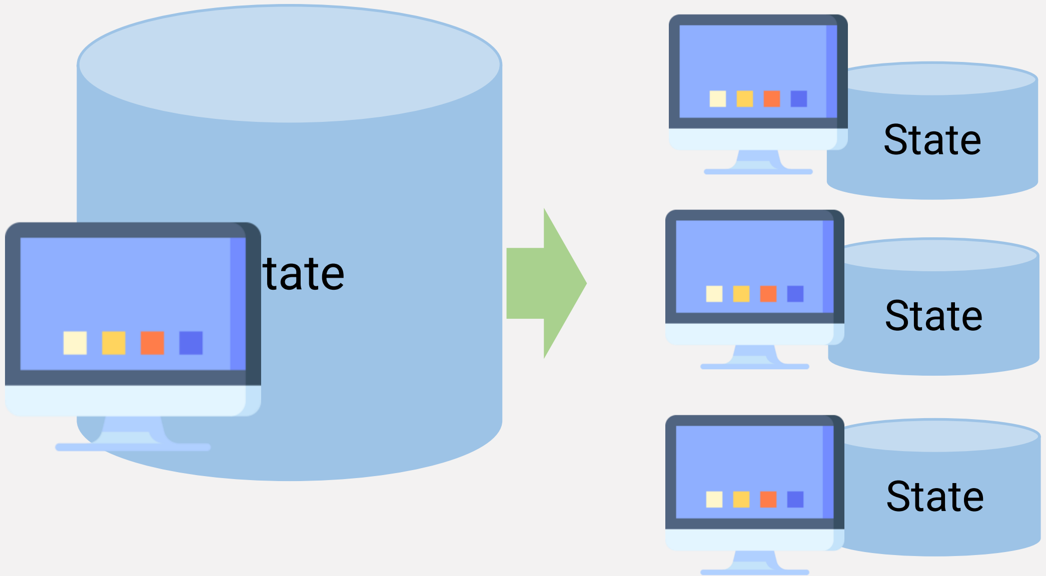
- MapReduce style
- Compute-storage coupled



Flink
Spark Streaming
APACHE STORM
samza

State Management: MapReduce-Style

- MapReduce style
- Compute-storage coupled



Flink

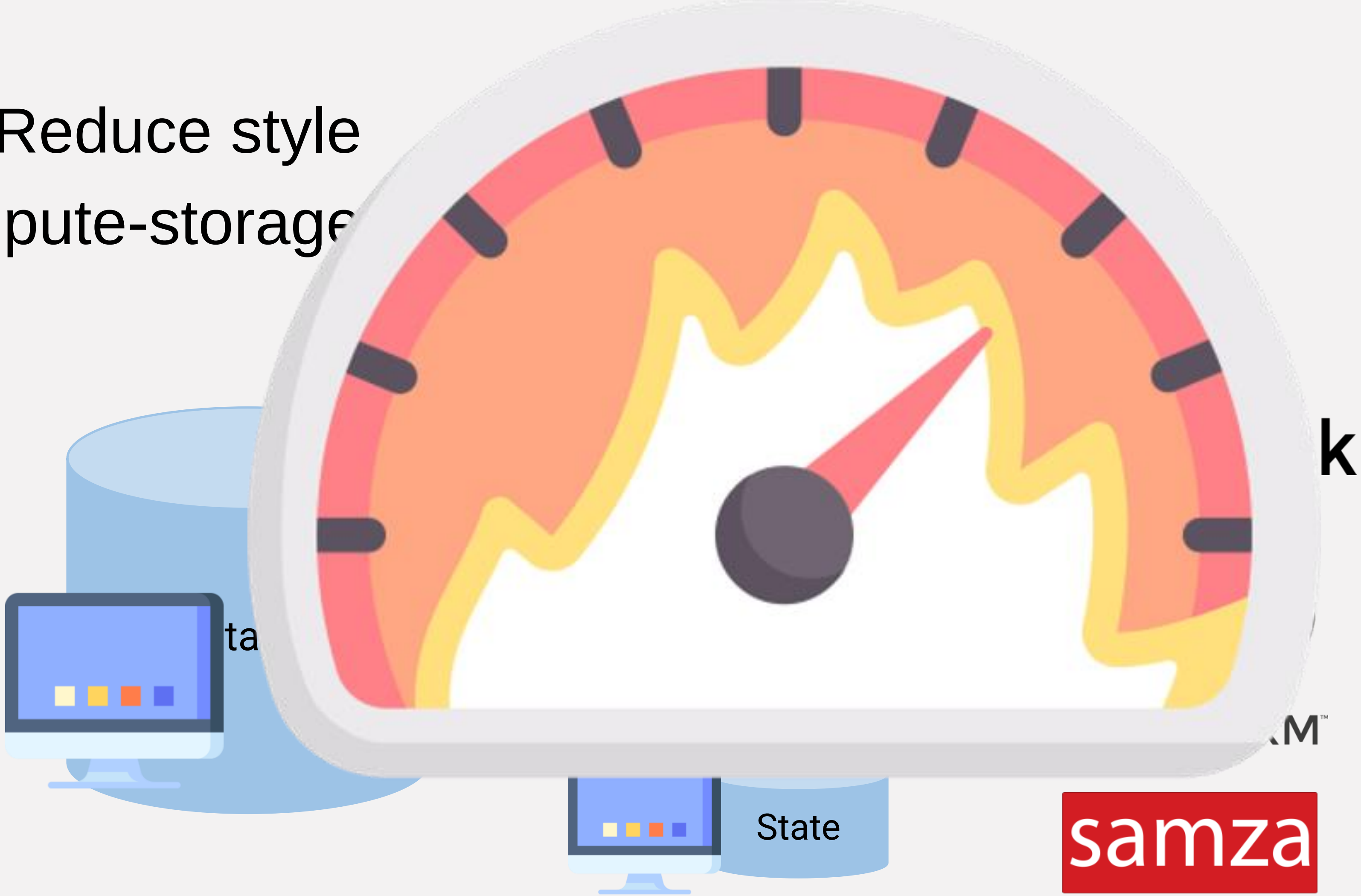
Spark Streaming

APACHE STORM

samza

State Management: MapReduce-Style

- MapReduce style
- Compute-storage



State Management in the Cloud Era

- Cloud-native style
- Compute-storage <u>*decoupled*</u>

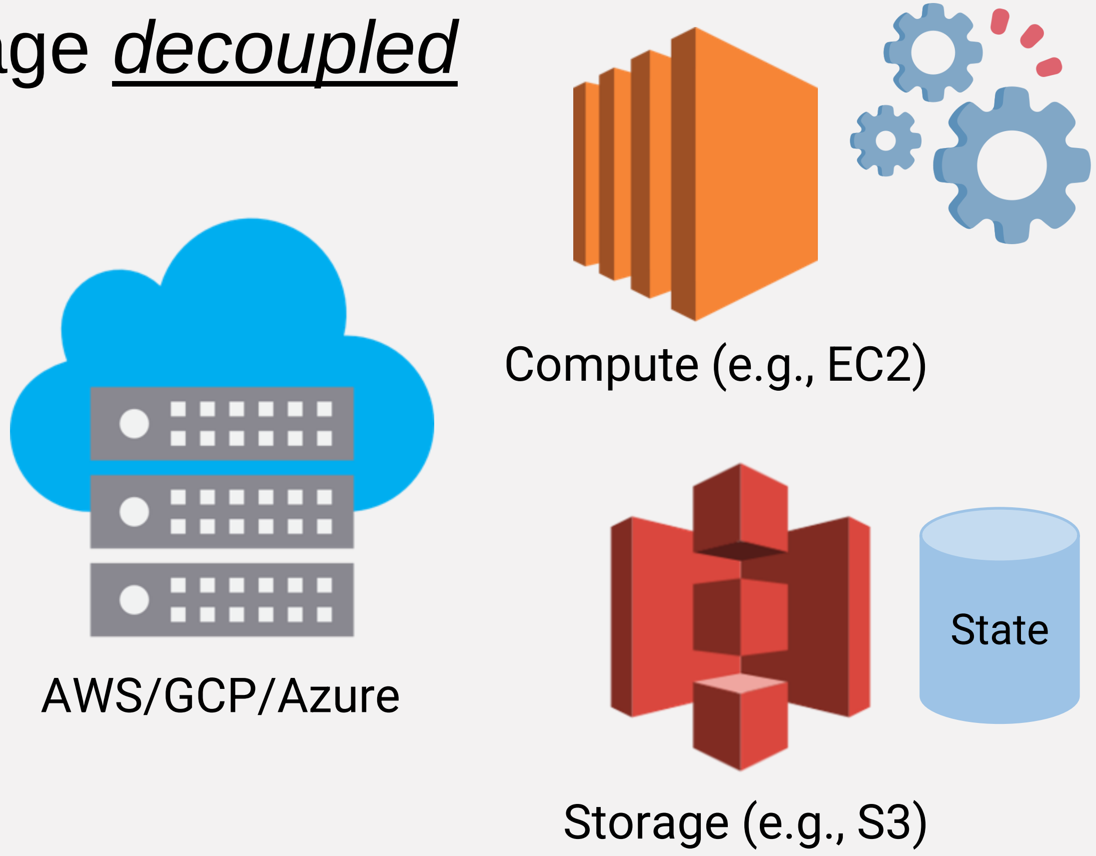
Compute (e.g., EC2)

AWS/GCP/Azure

Storage (e.g., S3)

State Management in the Cloud Era

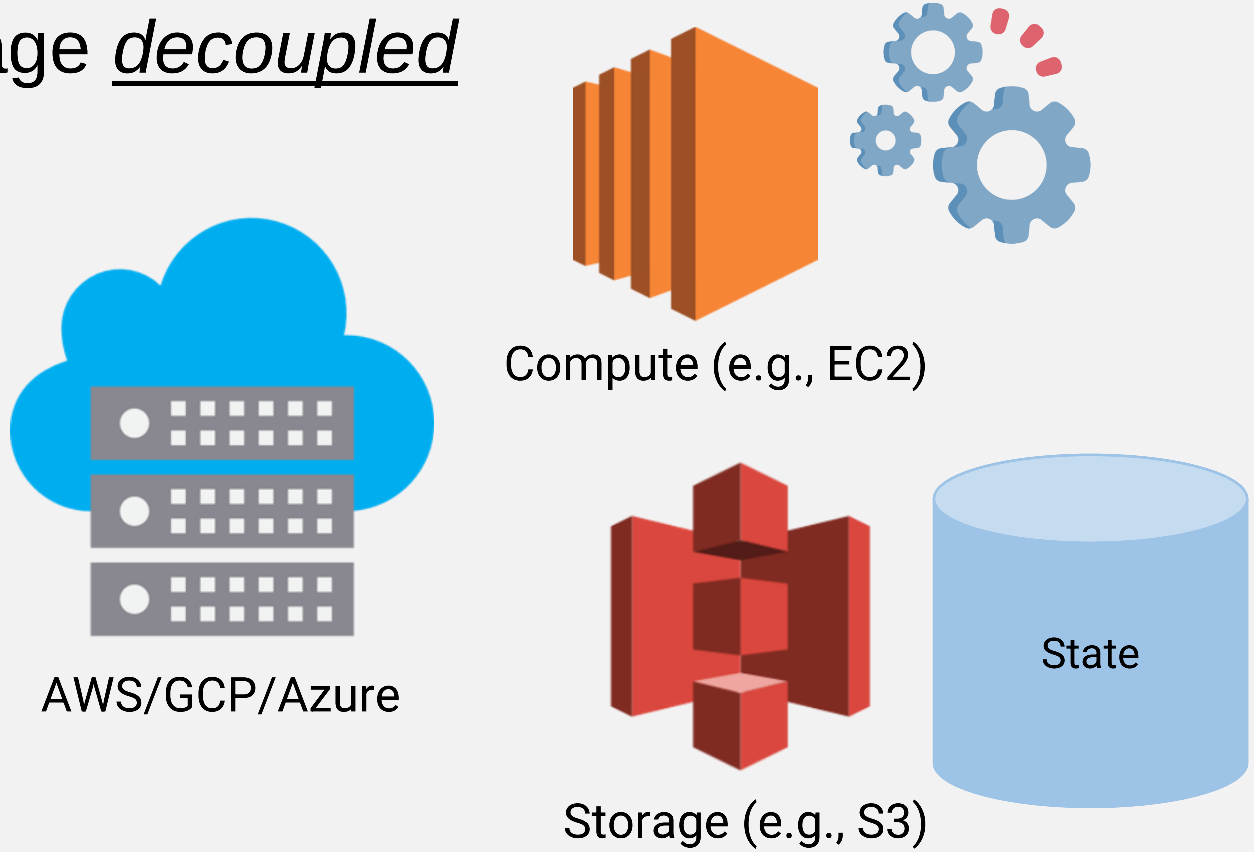
- Cloud-native style
- Compute-storage *decoupled*



RisingWaveLabs

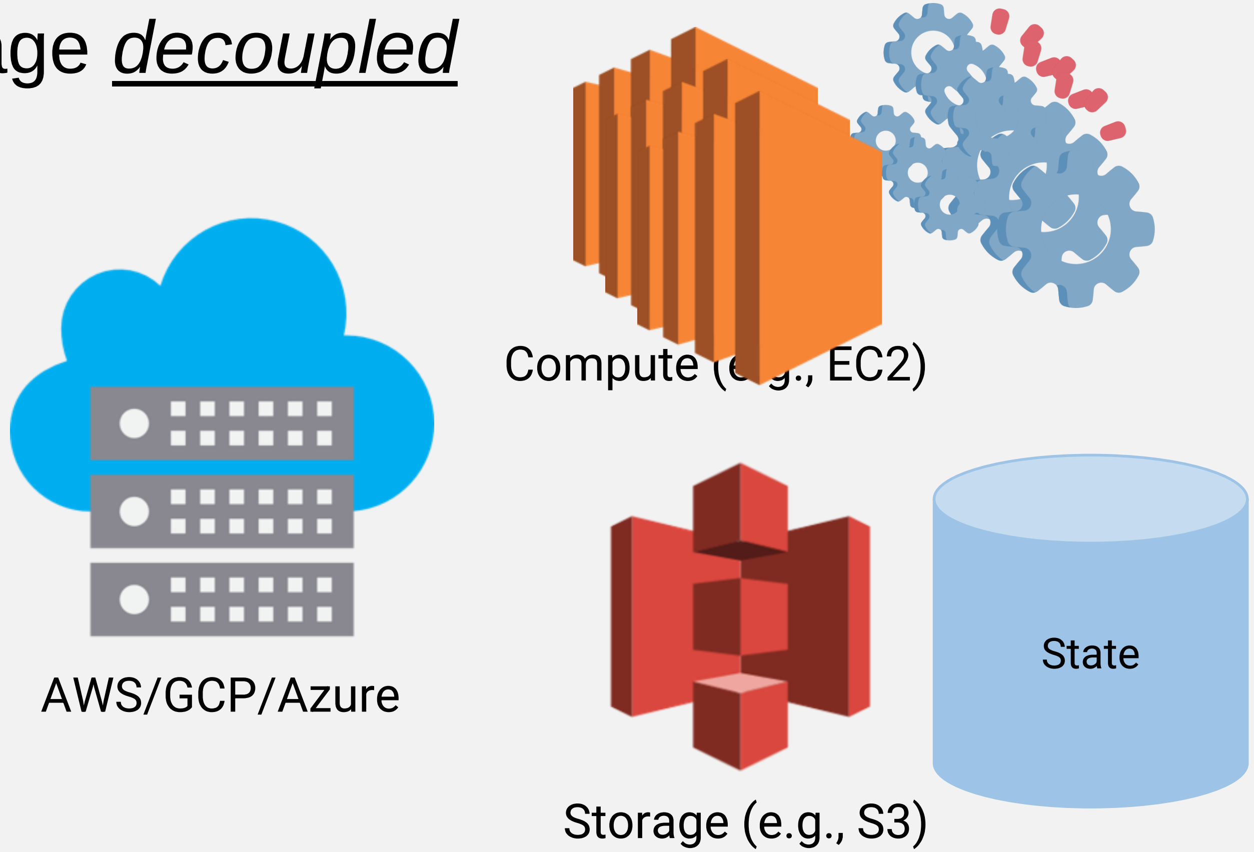
State Management in the Cloud Era

- Cloud-native style
- Compute-storage *decoupled*



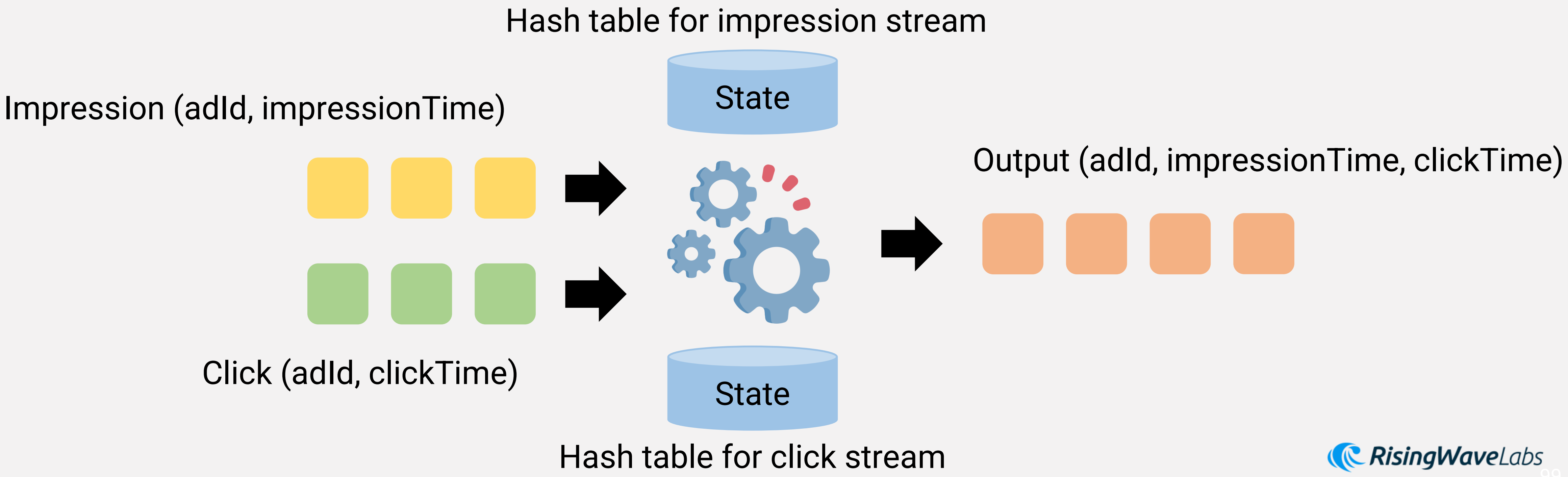
State Management in the Cloud Era

- Cloud-native style
- Compute-storage *decoupled*



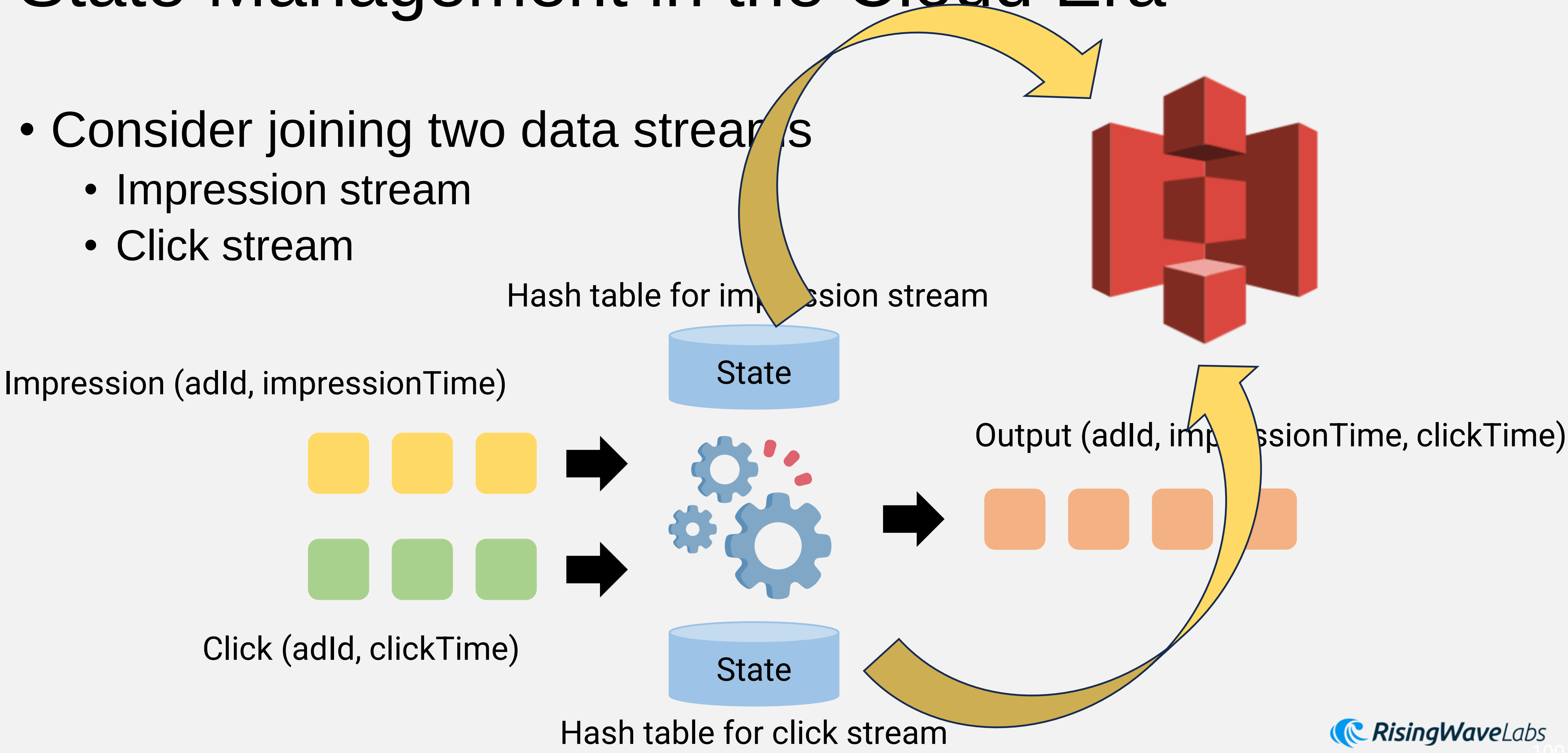
State Management in the Cloud Era

- Consider joining two data streams
  - Impression stream
  - Click stream



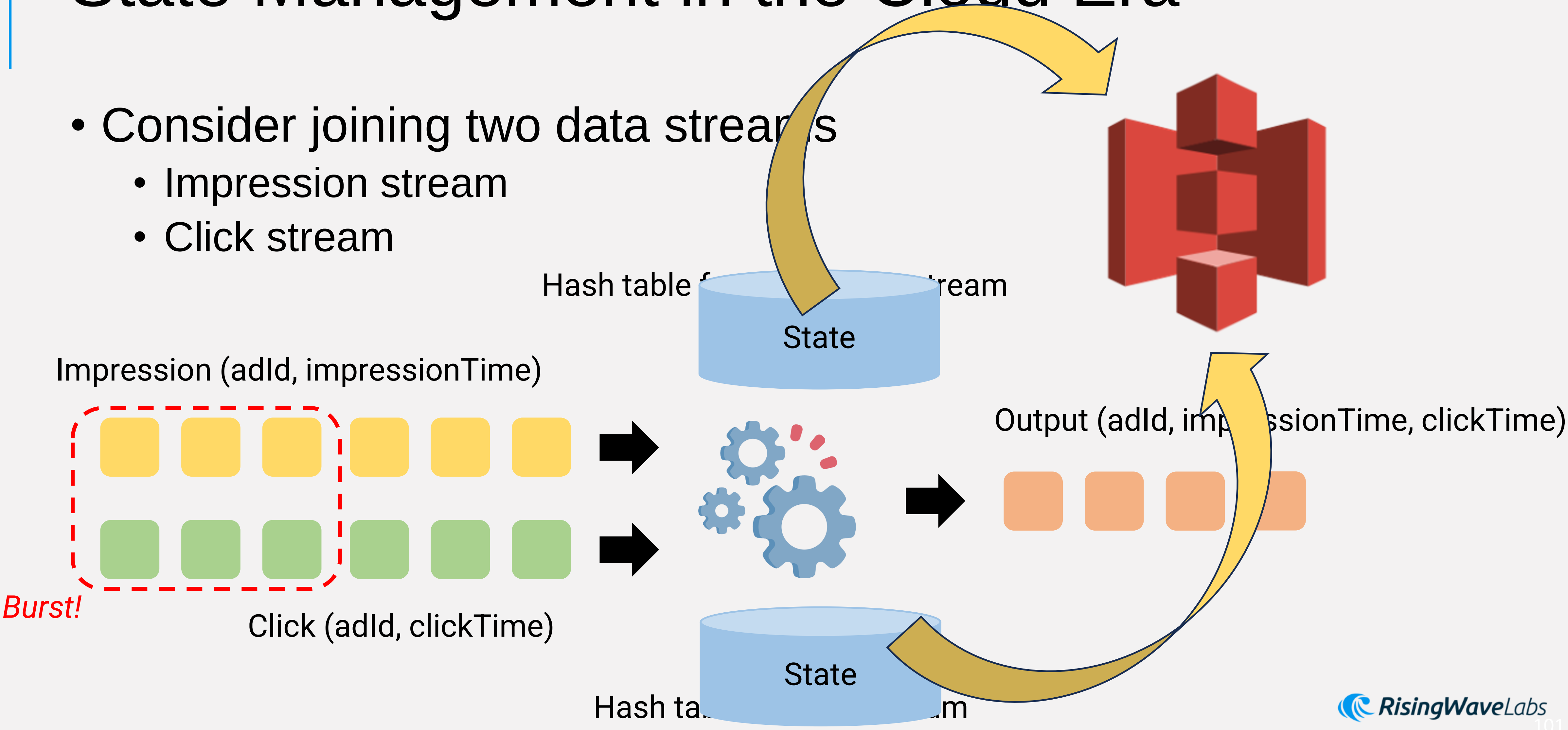
State Management in the Cloud Era

- Consider joining two data streams
  - Impression stream
  - Click stream



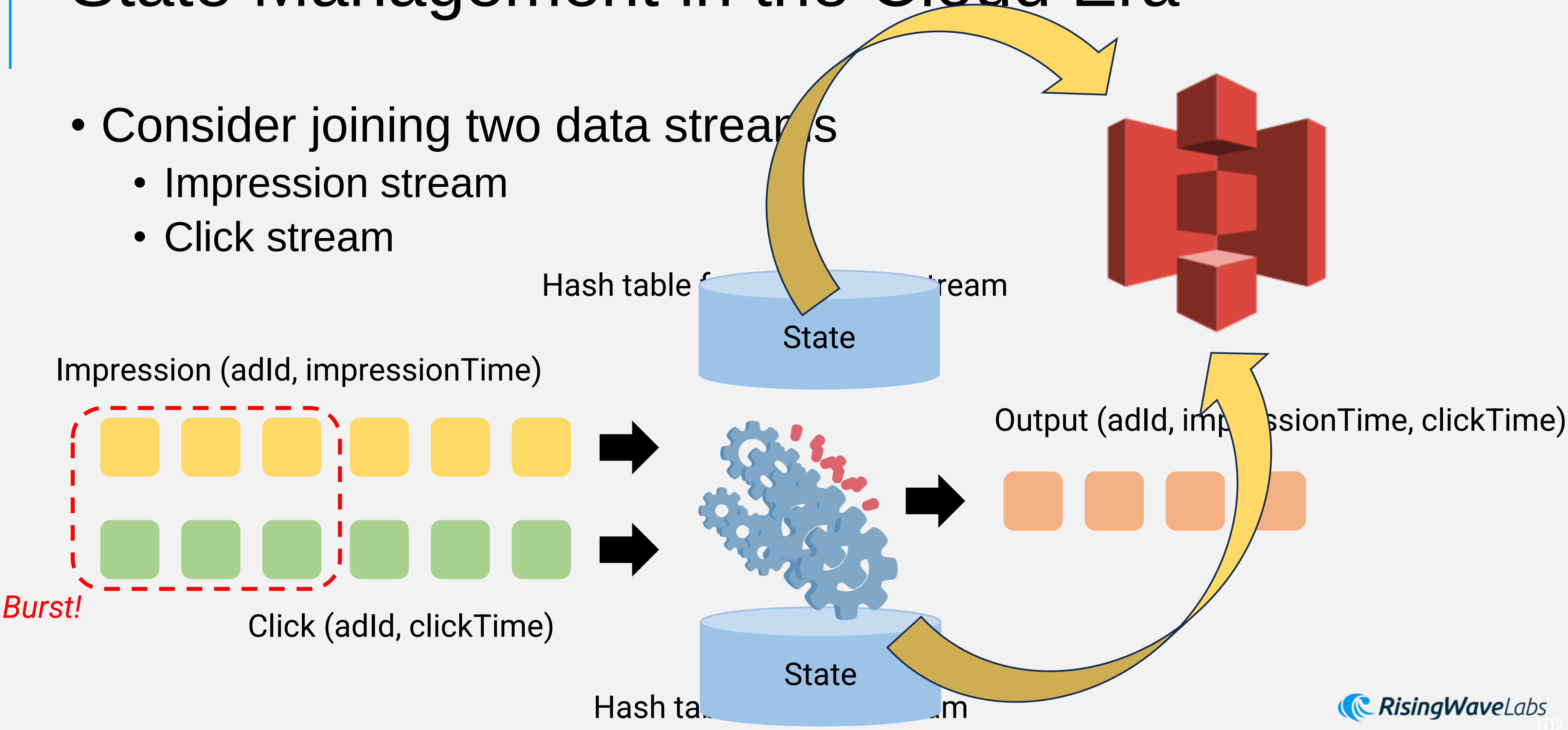
State Management in the Cloud Era

- Consider joining two data streams
  - Impression stream
  - Click stream



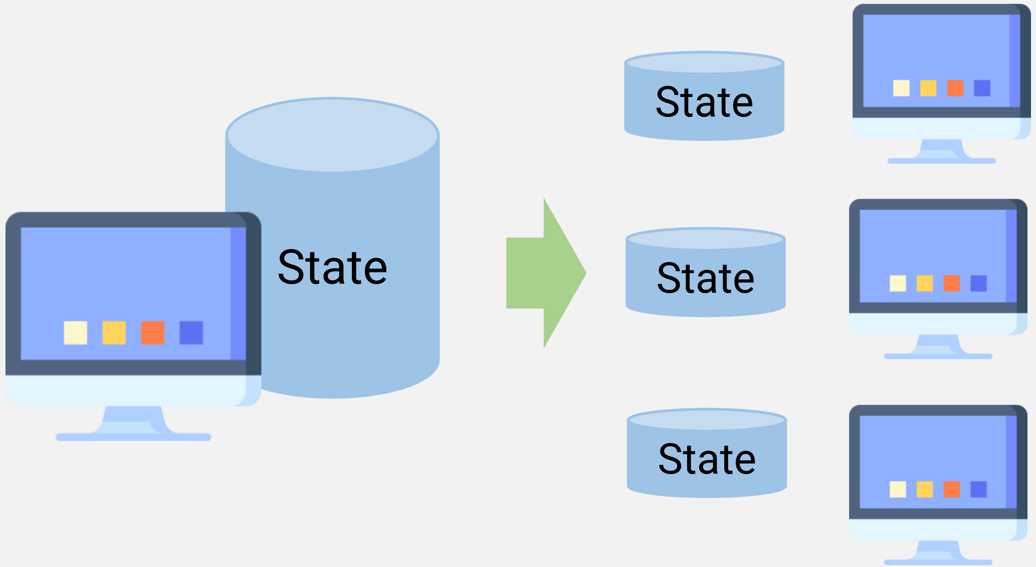
State Management in the Cloud Era

- Consider joining two data streams
  - Impression stream
  - Click stream

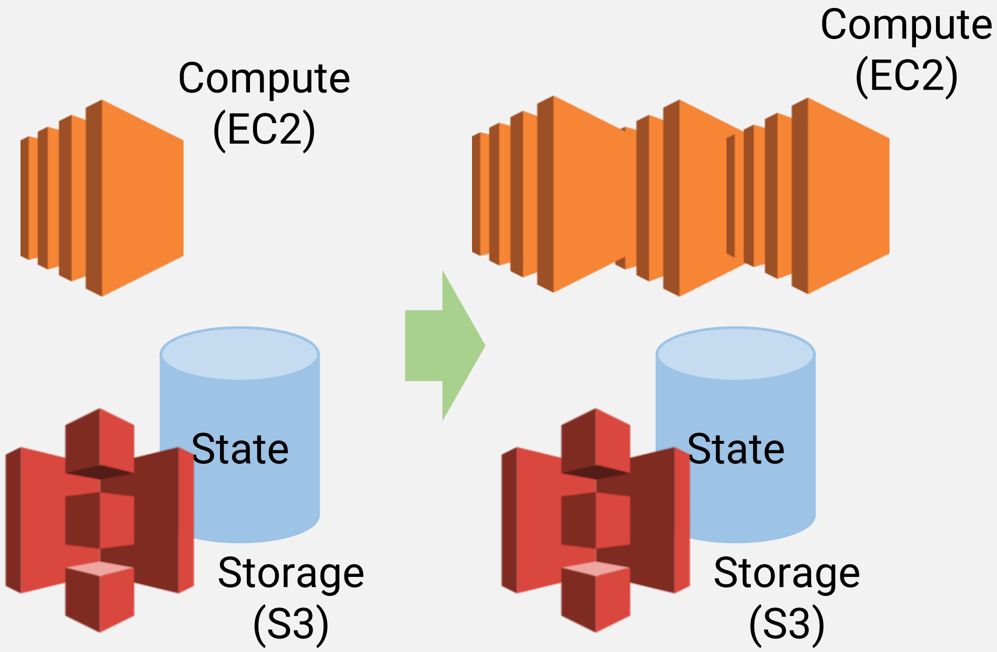


State Management: Comparison

MapReduce style, compute-storage coupled

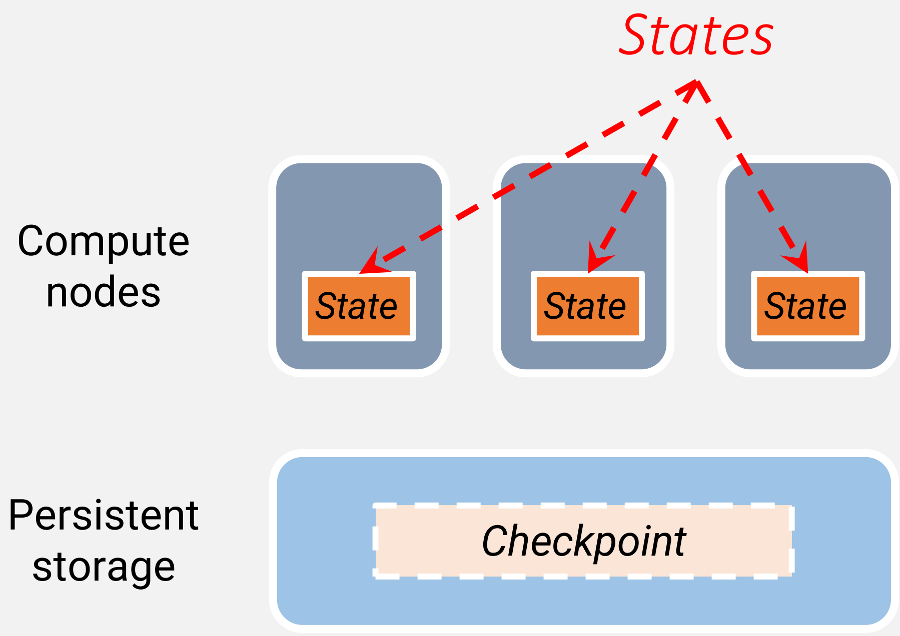


Cloud-native style, compute-storage decoupled

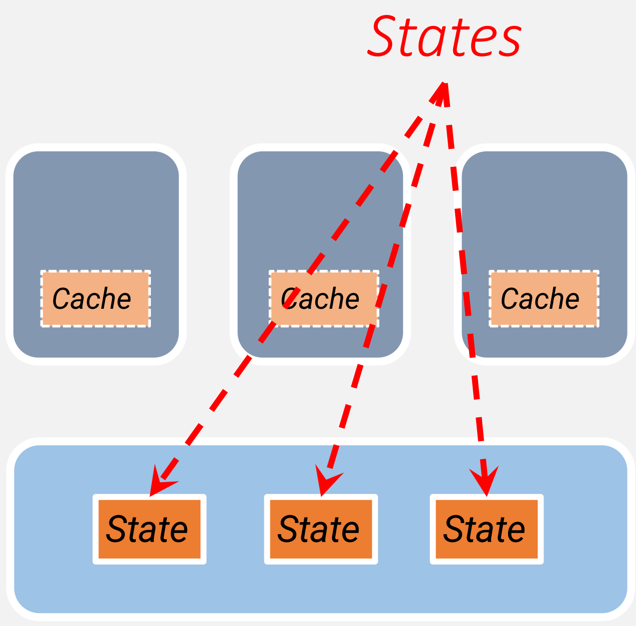


State Management: Comparison

MapReduce style, compute-storage coupled



Cloud-native style, compute-storage decoupled

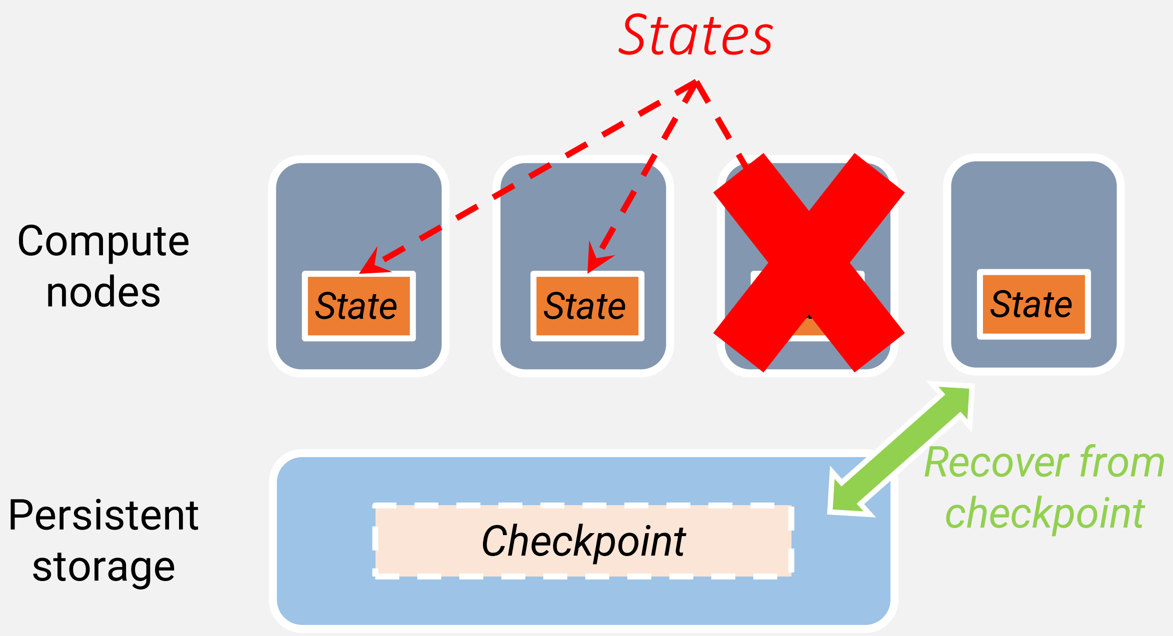


"state as checkpoint"

State Management: Failure Recovery

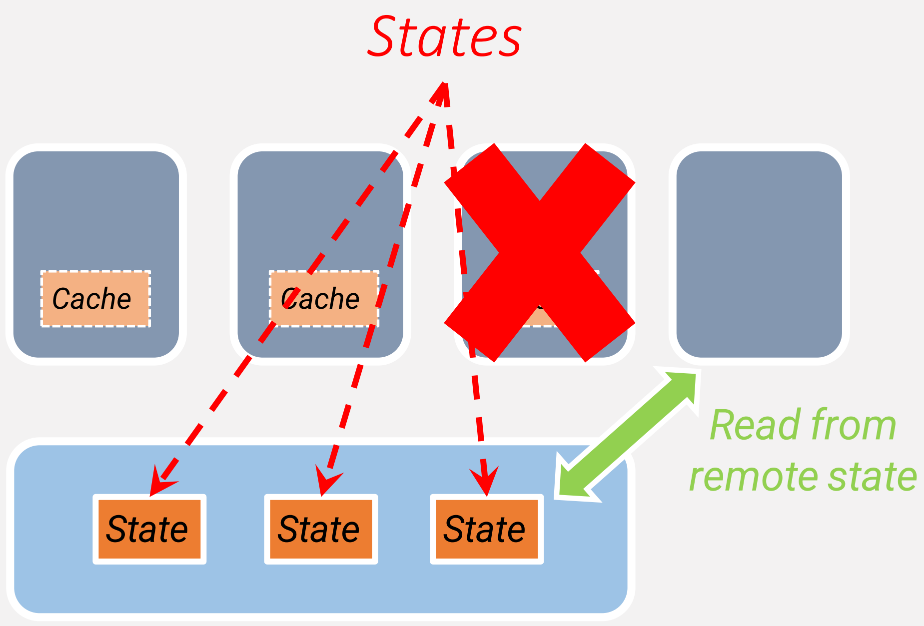
Flink Spark Streaming APACHE STORM samza

MapReduce style, compute-storage coupled



RisingWave

Cloud-native style, compute-storage decoupled

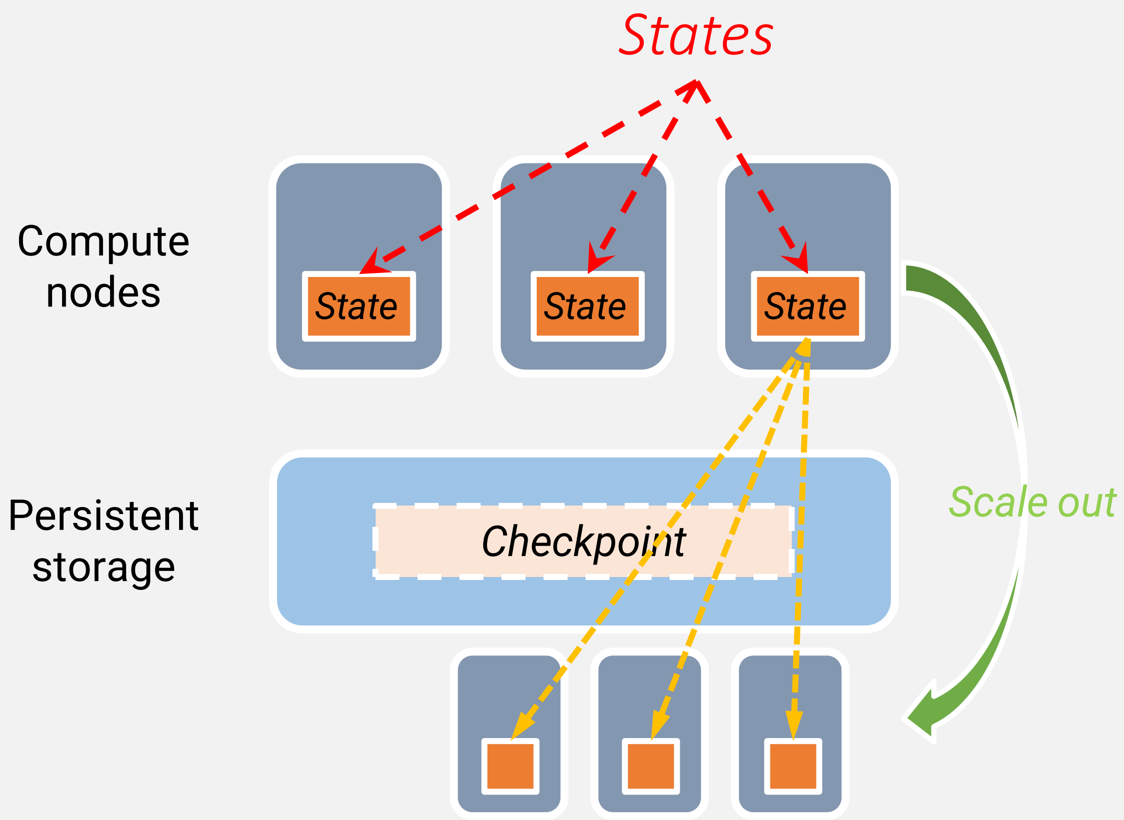


"state as checkpoint"

State Management: Elastic Scaling

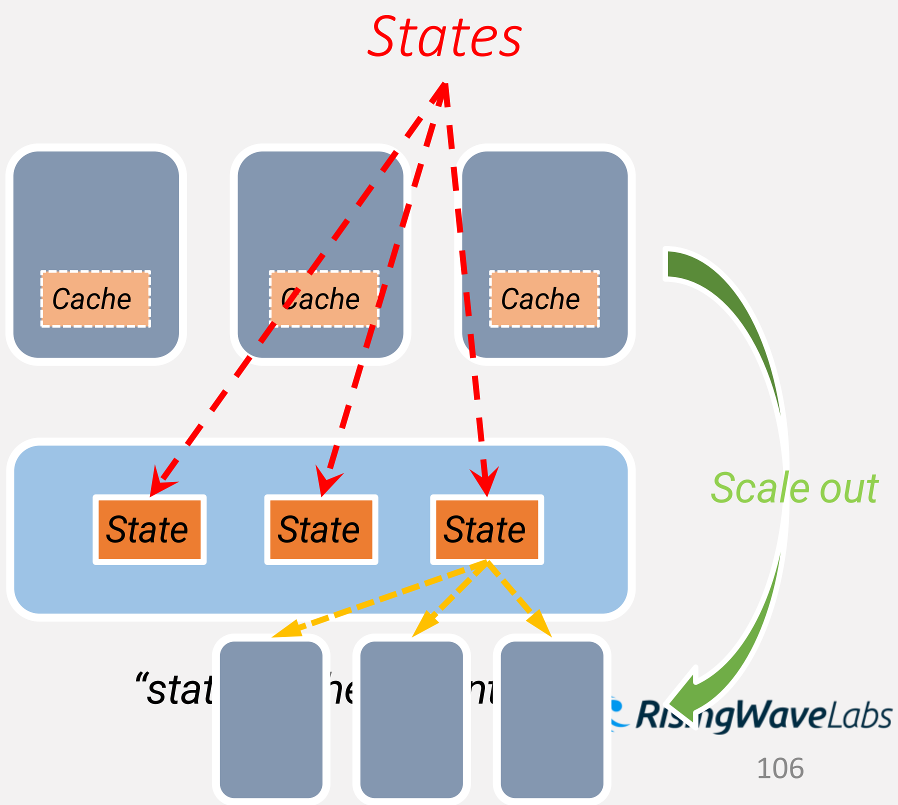
Flink | Spark Streaming | Apache Storm | samza

MapReduce style, compute-storage coupled



RisingWave

Cloud-native style, compute-storage decoupled



Summary

- Stream processing systems continuously perform incremental computations as new events arrive
- Key concepts: events, time windowing, watermark, …
- Single node -> distributed -> cloud
- State management is critical in stream processing systems!

Thank you!

Join RisingWave community today!

risingwave.com/slack