DDAM, 2019
CONTINUOUS APPLICATIONS: EVOLVING STREAMING

Docente: Patrizio Dazzi

FROM STREAMING TO CONTINOUS APPLICATIONS

- Frameworks for structured stream computing have become one of the most widely used distributed streaming engines
  - thanks to its high-level API and exactly-once semantics.
- Nonetheless, as these types of engines became common developers often need more than just a streaming programming model to build real-time applications.
- The need is to simplify real-time applications

MOTIVATION

- Most streaming engines focus on performing computations on a stream.
- As an example, one can map a stream to run a function on each record, reduce it to aggregate events by time, etc.
  - Unfortunately, no use case of streaming engines only involved performing computations on a stream.
  - Instead, stream processing happens as part of a larger application, which we'll call a continuous application.

CONTINOUS APPLICATIONS: MOTIVATING EXAMPLES

- **Updating data that will be served in real-time.**
- For instance, developers might want to update a summary table that users will query through a web application. In this case, much of the complexity is in the interaction between the streaming engine and the serving system:
  - for example, can you run queries on the table while the streaming engine is updating it?
  - The "complete" application is a real-time serving system, not a map or reduce on a stream.

CONTINOUS APPLICATIONS: MOTIVATING EXAMPLES

- **Extract, transform and load (ETL).**
- One common use case is continuously moving and transforming data from one storage system to another (e.g. JSON logs to an Apache Hive table).
- This requires careful interaction with both storage systems to ensure no data is duplicated or lost — much of the logic is in this coordination work.

CONTINOUS APPLICATIONS: MOTIVATING EXAMPLES

- **Creating a real-time version of an existing batch job.**
- This is hard because many streaming systems don't guarantee their result will match a batch job.
- For example, build live dashboards using a streaming engine and daily reporting using batch jobs, only to have customers complain that their daily report (or worse, their bill!) did not match the live metrics.

CONTINOUS APPLICATIONS: MOTIVATING EXAMPLES

- **Online machine learning.**
- These continuous applications often combine large static datasets, processed using batch jobs, with real-time data and live prediction serving.

CONTINOUS APPLICATIONS

- Continuous applications are an end-to-end application that reacts to data in real-time.
- In particular, developers would like to use a single programming interface to support
  - the facets of continuous applications that are currently handled in separate systems,
  - such as query serving or interaction with batch jobs.

CONTINOUS APPLICATIONS: WHY? (1)

- **Updating data that will be served in real time.** the developer would write a single application that handles both updates and serving or would use an API that automatically performs transactional updates on a serving system

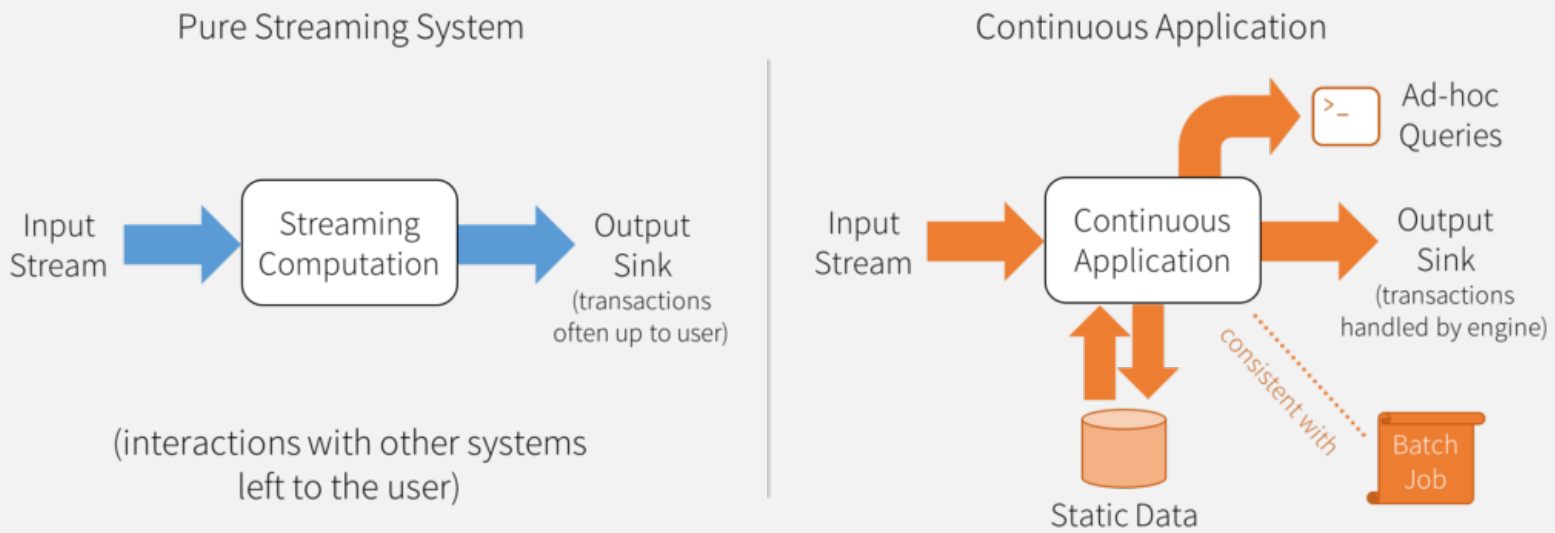
- **Extract, transform and load (ETL).** The developer would simply list the transformations required as in a batch job, and the streaming system would handle coordination with both storage systems to ensure exactly-once processing.

CONTINOUS APPLICATIONS: WHY? (2)

- **Creating a real-time version of an existing batch job.** The streaming system would guarantee results are always consistent with a batch job on the same data.

- **Online machine learning.** The machine learning library would be designed to combine real-time training, periodic batch training, and prediction serving behind the same API.

CONTINOUS APPLICATIONS: STRUCTURE



IN SHORT

- to make it easier to build end-to-end streaming applications,
  - which integrate with storage, serving systems, and batch jobs
    - in a consistent and fault-tolerant way.
    - as this is hard to do with current distributed streaming engines

BECAUSE… STREAMING IS DIFFICULT

- At first glance, building a distributed streaming engine might seem as simple as launching a set of servers and pushing data between them.
- [that was not simple at all in our class, btw!!!]
- Unfortunately, distributed stream processing runs into multiple complications that don’t affect simpler computations like batch jobs.

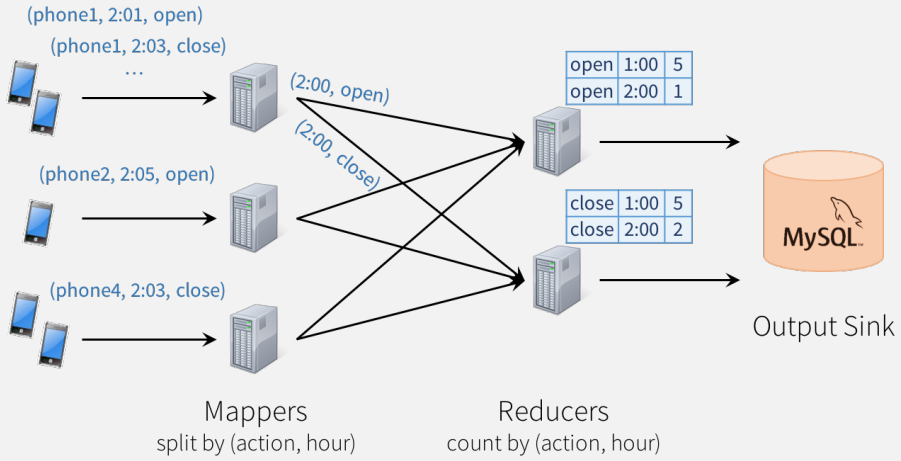
BECAUSE… STREAMING IS DIFFICULT

- To start, consider a simple application: we receive (phone_id, time, action) events from a mobile app, and want to count how many actions of each type happened each hour, then store the result in MySQL.
- If we were running this application as a batch job and had a table with all the input events, we could express it as the following SQL query:

```
SELECT action, WINDOW(time, "1 hour"), COUNT(*)
FROM events
GROUP BY action, WINDOW(time, "1 hour")
```

SAMPLE ARCHITECTURE

- In a distributed streaming engine, we might set up nodes to process the data in a "map-reduce" pattern.
- Each node in the first layer reads a partition of the input data (say, the stream from one set of phones), then hashes the events by (action, hour) to send them to a reducer node, which tracks that group's count and periodically updates MySQL.



UNFORTUNATELY, THIS TYPE OF DESIGN CAN INTRODUCE QUITE A FEW CHALLENGES

- **Consistency:** This distributed design can cause records to be processed in one part of the system before they're processed in another, leading to nonsensical results.
  - For example, suppose our app sends an "open" event when users open it, and a "close" event when closed.
  - If the reducer node responsible for "open" is slower than the one for "close", we might see a higher total count of "closes" than "opens" in MySQL, which would not make sense. The image above actually shows one such example.

UNFORTUNATELY, THIS TYPE OF DESIGN CAN INTRODUCE QUITE A FEW CHALLENGES

- **Fault tolerance**: What happens if one of the mappers or reducers fails?
- A reducer should not count an action in MySQL twice, but should somehow know how to request old data from the mappers when it comes up.
- Streaming engines go through a great deal of trouble to provide strong semantics here, at least *within* the engine.
- In many engines, however, keeping the result consistent in external storage is left to the user.

UNFORTUNATELY, THIS TYPE OF DESIGN CAN INTRODUCE QUITE A FEW CHALLENGES

- **Out-of-order data**:
- In the real world, data from different sources can come out of order: for example, a phone might upload its data hours late if it's out of coverage.
- Just writing the reducer operators to assume data arrives in order of time fields will not work—they need to be prepared to receive out-of-order data, and to update the results in MySQL accordingly.

THIS IS WHY IN OUR COURSE WE RELY ON STRUCTURED STREAMING MODEL

- To tackle the issue of semantics head-on by making a strong guarantee about the system: at any time, the output of the application is equivalent to executing a batch job on a prefix of the data.
- For example, in our monitoring application, the result table in MySQL will always be equivalent to taking a prefix of each phone's update stream (whatever data made it to the system so far) and running the SQL query we showed above.
- There will never be "open" events counted faster than "close" events, duplicate updates on failure, etc. Structured Streaming automatically handles consistency and reliability both within the engine and in interactions with external systems (e.g. updating MySQL transactionally).