APACHE FLINK™
STREAM AND BATCH PROCESSING IN A SINGLE ENGINE

Akshaya Kalyanaraman

WHAT IS APACHE FLINK?

Data Stream Processing

realtime results from data streams

Batch Processing

process static and historic data

Event-driven Applications

data-driven actions and services

MOTIVATION

- In Lambda Architecture: Two separate execution engines for batch and streaming
- Unification of Batch and Stream Processing in a single framework, Flink.
- Apache Flink provides a highly flexible windowing mechanism.
- Flink supports different notions of time.

STREAM ANALYTICS

NOTIONS OF TIME

Event Time

Time when event happened.

12:23 am

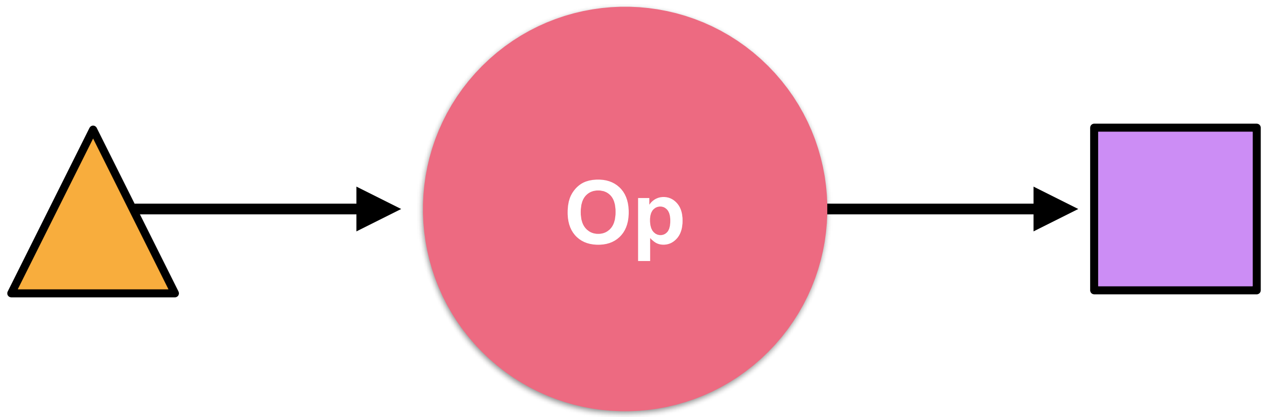
1:37 pm

Processing Time

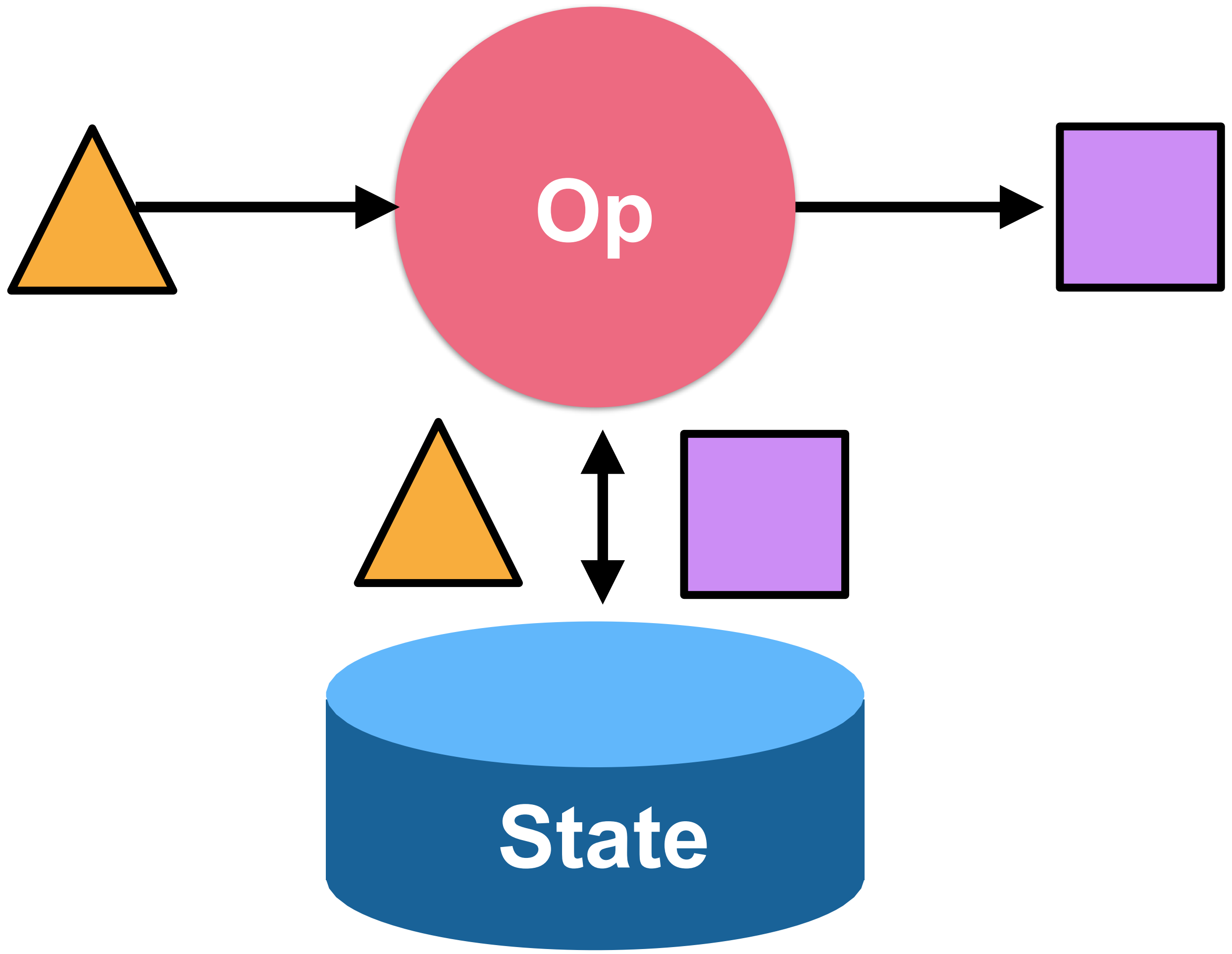
Time measured by system clock

STATEFUL STREAMING

Stateless Stream Processing



Stateful Stream Processing



PROCESSING SEMANTICS

At-least once
May over-count after failure

Exactly Once
Correct counts after failures

End-to-end exactly once
Correct counts in external system (e.g. DB, file system) after failure

PROCESSING SEMANTICS

Flink guarantees **exactly once**

End-to-end exactly once with specific sources and sinks (e.g. Kafka -> Flink -> HDFS)

Internally, Flink periodically takes **consistent snapshots** of the state without ever stopping computation

WINDOWING

- Window configured using **assigner** and optionally **trigger** and **evictor.**
- **Assigner:** assigns each record to logical windows.
- **Trigger:** defines when the operation associated with the window definition is performed.
- **Evictor:** determines which records to retain within each window.

WINDOWING : EXAMPLE

- Below is a window definition with a range of 6 seconds that slides every 2 seconds (the assigner).
- The window results are computed once the watermark passes the end of the window (the trigger).

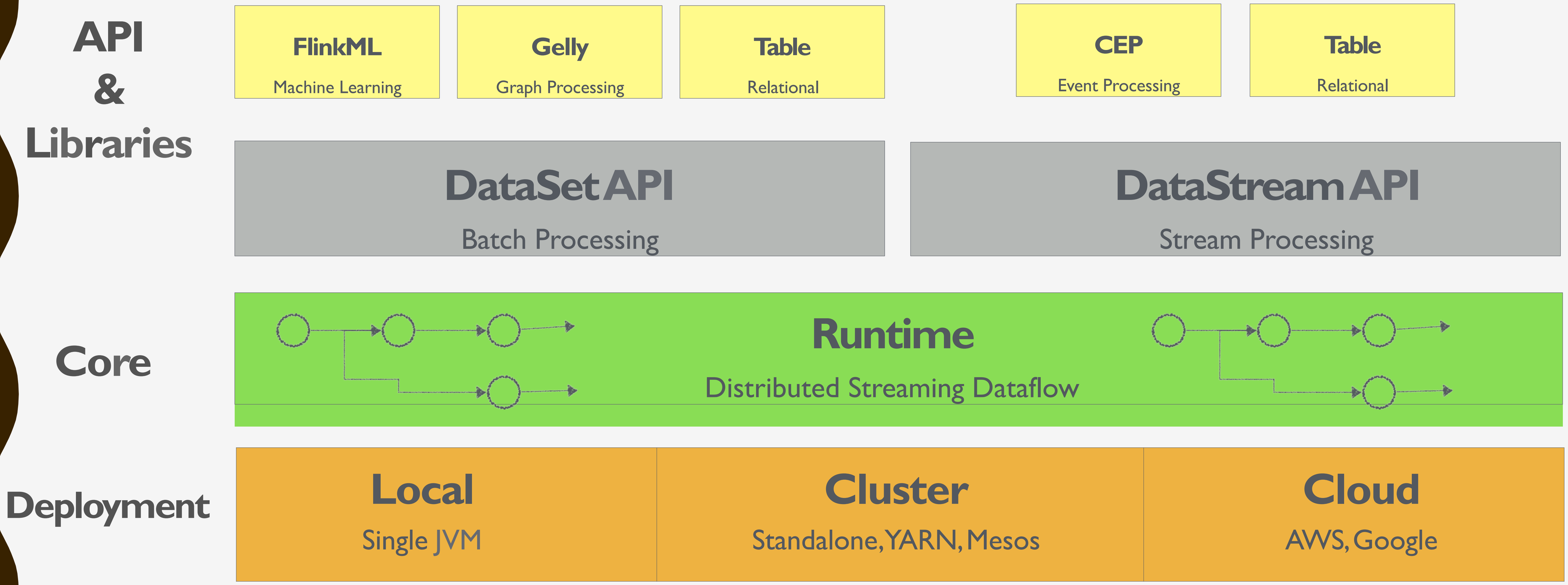
```
stream
.window(SlidingTimeWindows.of(Time.of(6,
SECONDS), Time.of(2, SECONDS))
.trigger(EventTimeTrigger.create())
```

WINDOWING : EXAMPLE

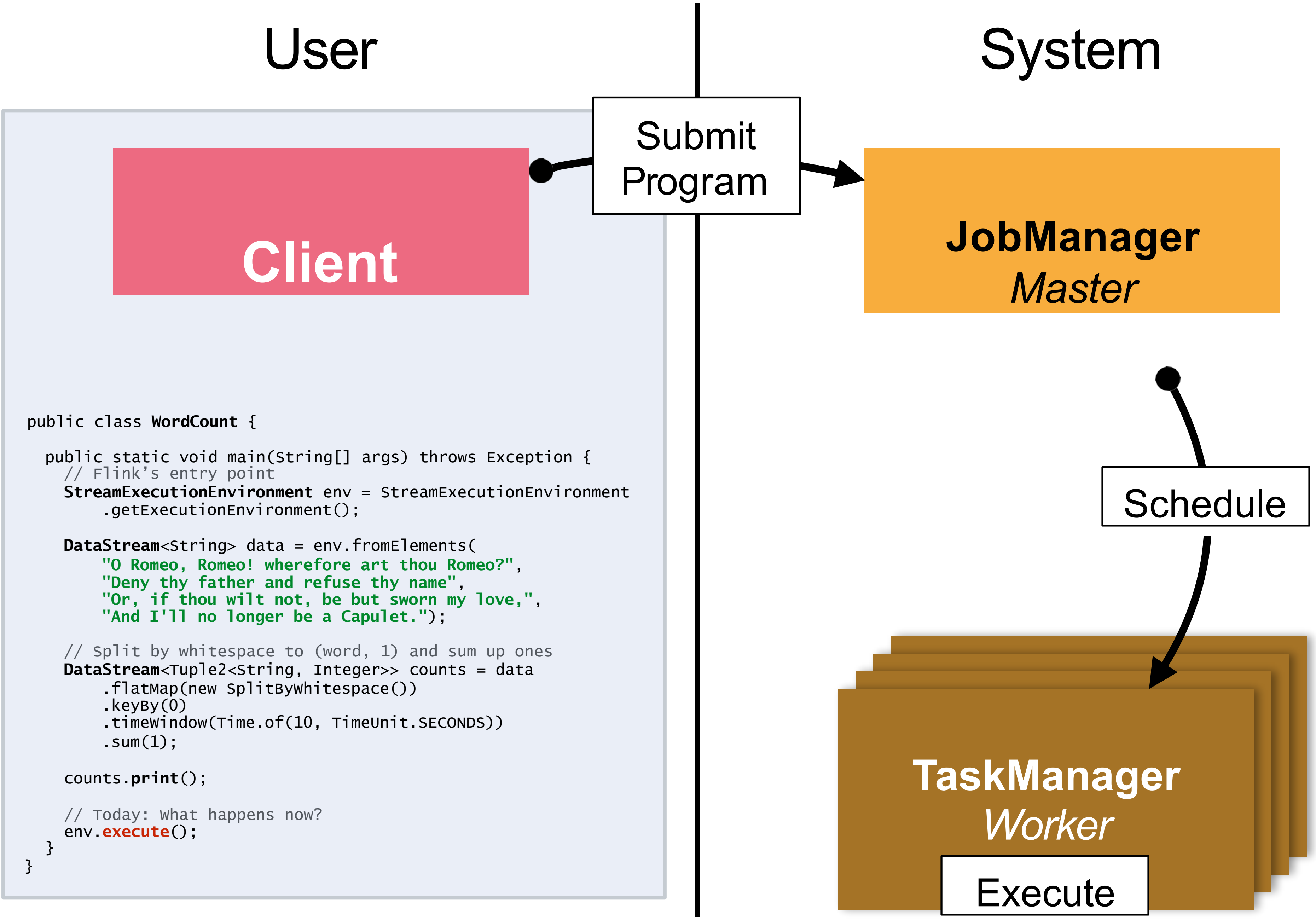
- A global window creates a single logical group.
- The example defines a global window (i.e., the assigner) that invokes the operation on every 1000 events (i.e., the trigger) while keeping the last 100 elements (i.e., the evictor).

```
stream
.window(GlobalWindow.create())
.trigger(Count.of(1000))
.evict(Count.of(100))
```

FLINK STACK


API
&
Libraries
FlinkML
Machine Learning
Gelly
Graph Processing
Table
Relational
CEP
Event Processing
Table
Relational
DataSet API
Batch Processing
DataStream API
Stream Processing
Core
Runtime
Distributed Streaming Dataflow
Deployment
Local
Single JVM
Cluster
Standalone, YARN, Mesos
Cloud
AWS, Google

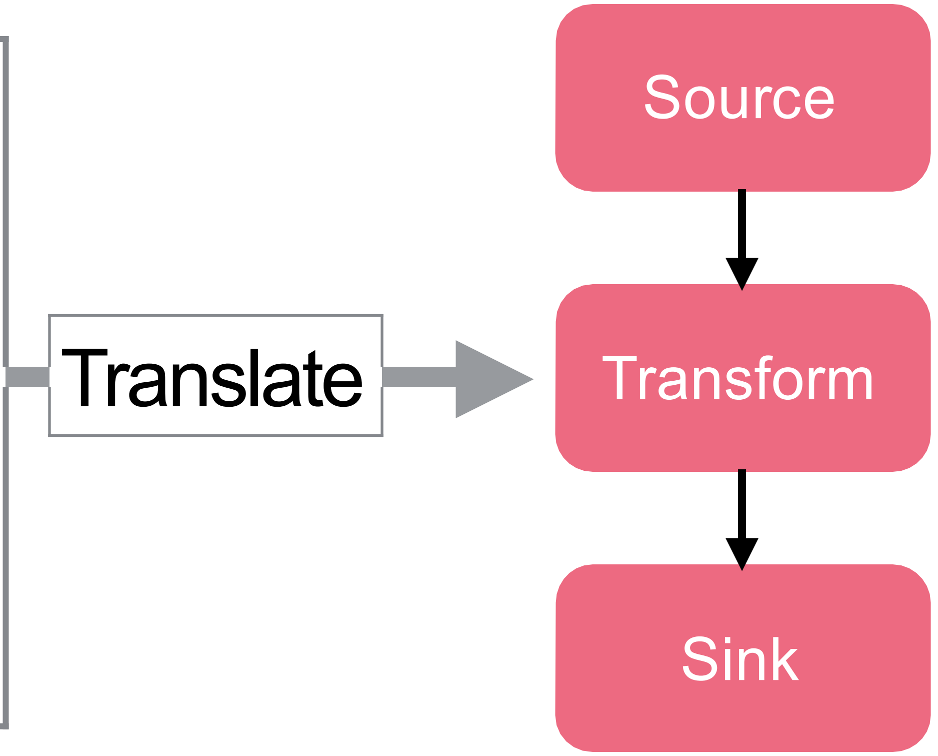
FLINK PROCESS MODEL




```java
public class WordCount {

  public static void main(String[] args) throws Exception {
    // Flink's entry point
    StreamExecutionEnvironment env = StreamExecutionEnvironment
        .getExecutionEnvironment();

    DataStream<String> data = env.fromElements(
        "O Romeo, Romeo! wherefore art thou Romeo?",
        "Deny thy father and refuse thy name",
        "Or, if thou wilt not, be but sworn my love,",
        "And I'll no longer be a Capulet.");

    // Split by whitespace to (word, 1) and sum up ones
    DataStream<Tuple2<String, Integer>> counts = data
        .flatMap(new SplitByWhitespace())
        .keyBy(0)
        .timeWindow(Time.of(10, TimeUnit.SECONDS))
        .sum(1);

    counts.print();

    // Today: what happens now?
    env.execute();
  }
}
```

CLIENT

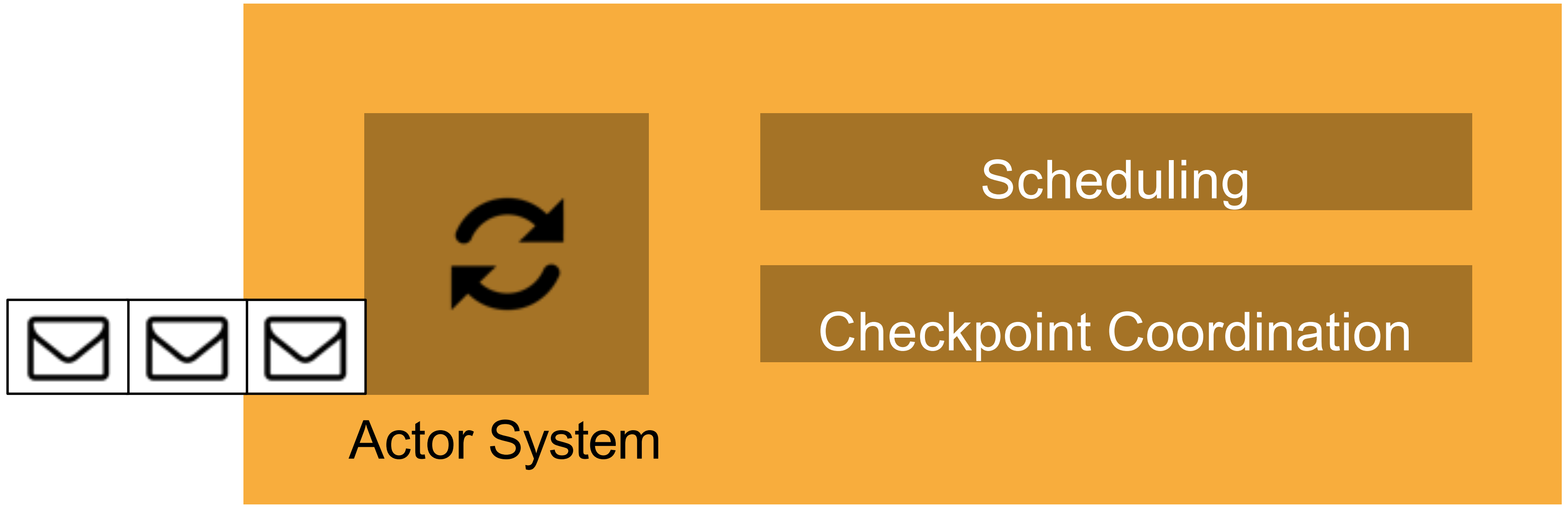
```
public class WordCount {

   public static void main(String[] args) throws Exception {
     // Flink's entry point
     StreamExecutionEnvironment env = StreamExecutionEnvironment
          .getExecutionEnvironment();

     DataStream<String> data = env.fromElements(
          "O Romeo, Romeo! wherefore art thou Romeo?",
          "Deny thy father and refuse thy name",
          "Or, if thou wilt not, be but sworn my love,",
          "And I'll no longer be a Capulet.");

     // Split by whitespace to (word, 1) and sum up ones
     DataStream<Tuple2<String, Integer>> counts = data
          .flatMap(new SplitByWhitespace())
          .keyBy(0)
          .timeWindow(Time.of(10, TimeUnit.SECONDS))
          .sum(1);

     counts.print();

     // Today: what happens
     now?  env.execute();
   }
}
```



Translates the API code to
a *data flow graph* called **JobGraph** and
submits it to the JobManager.

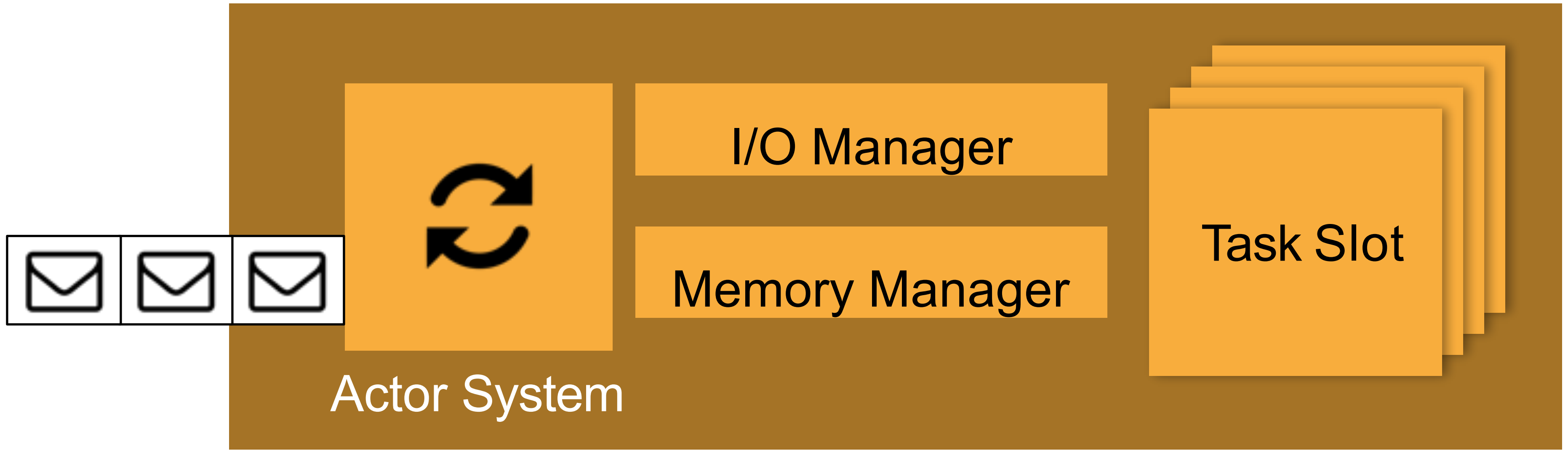
JOBMANAGER

- All coordination via JobManager (master):
  - Scheduling programs for execution
  - Checkpoint coordination
  - Monitoring workers



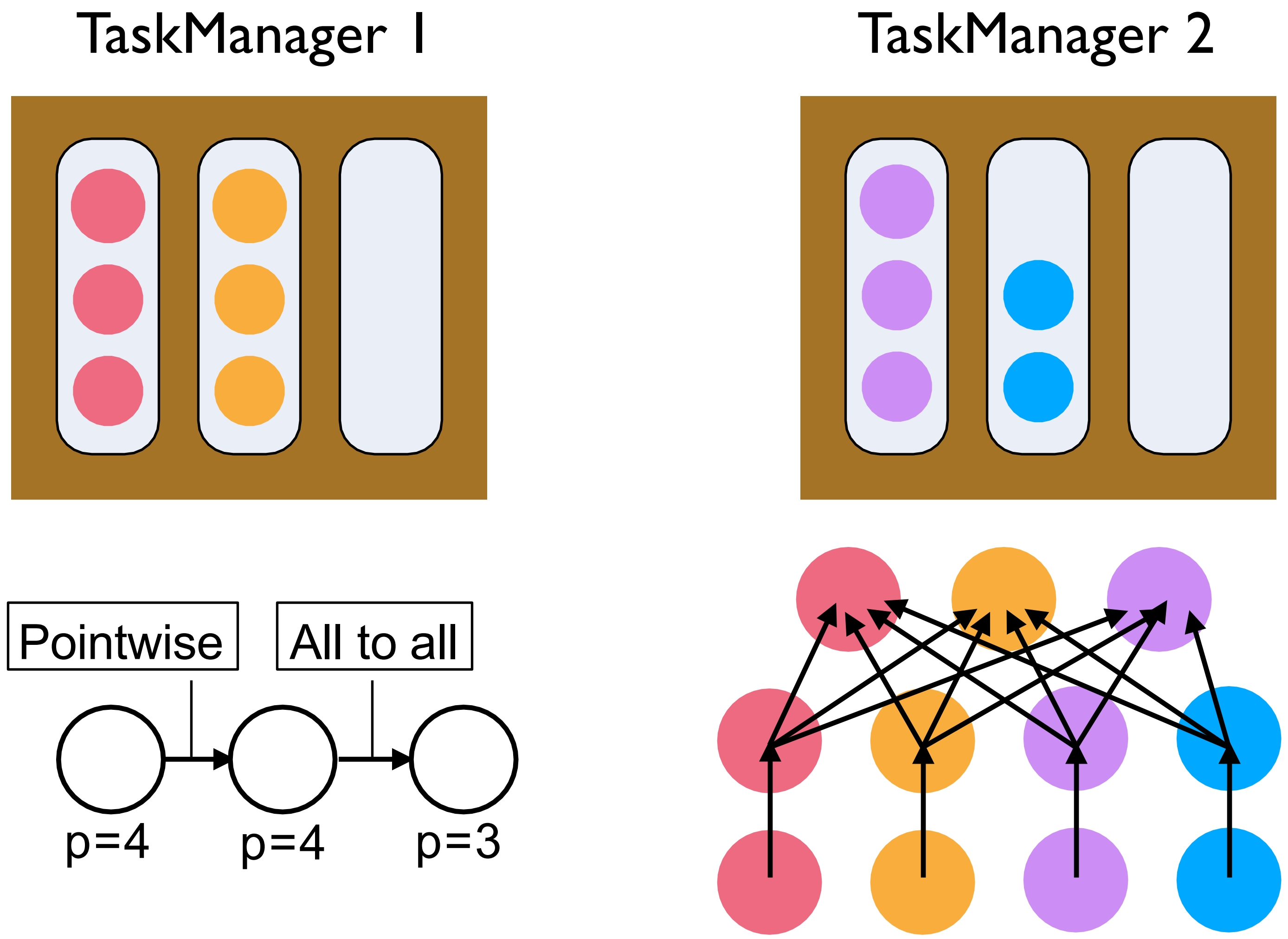
TASK MANAGER

- All data processing in TaskManager (worker):
  - Communicate with JobManager via Actor messages
  - Exchange data between themselves via dedicated data connections
  - Expose task slots for execution



SCHEDULING

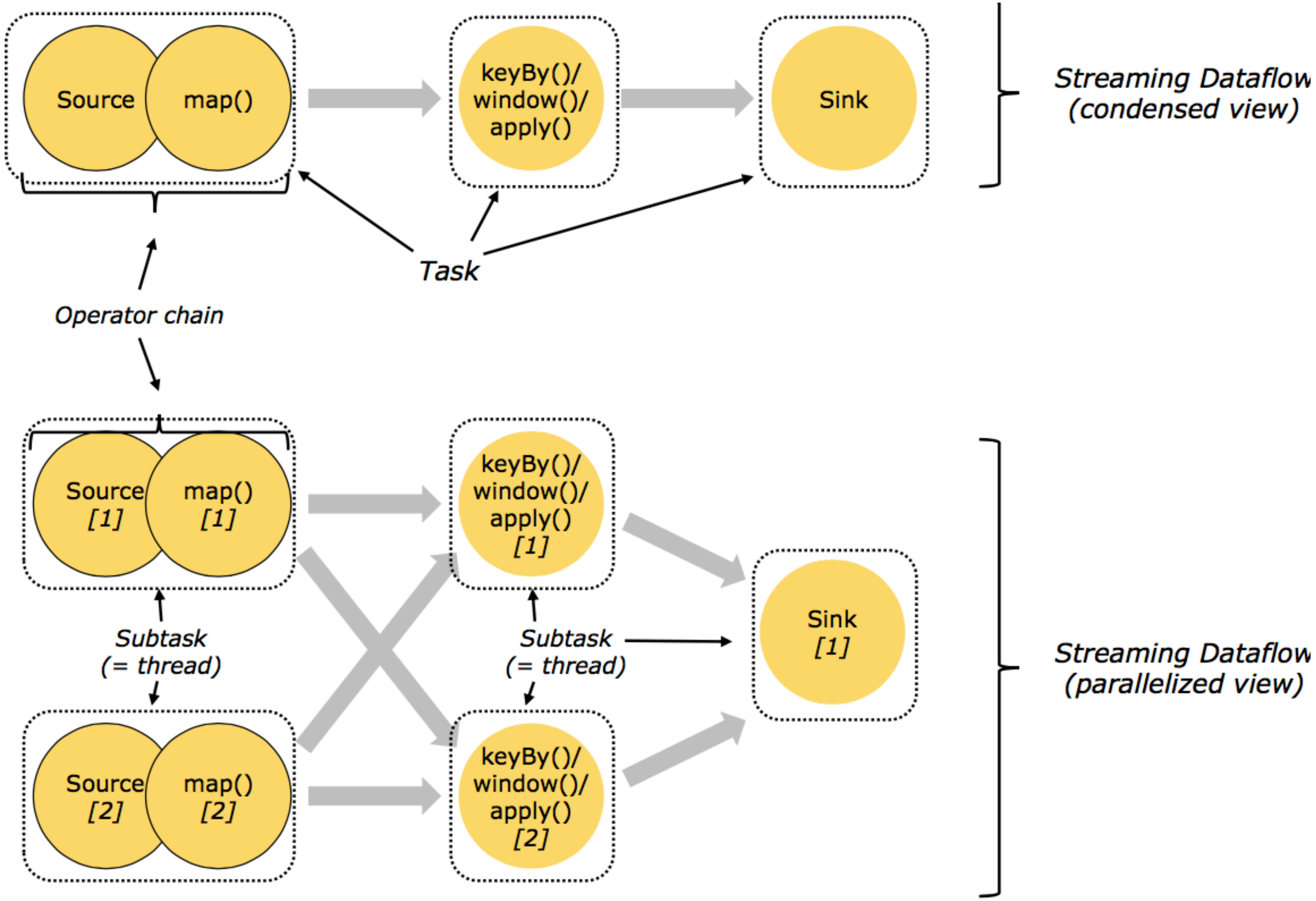
- Each ExecutionVertex will be executed one or more times
- The JobManager maps Execution to task slots
- Pipelined execution in same slot where applicable



SAMPLE QUERY

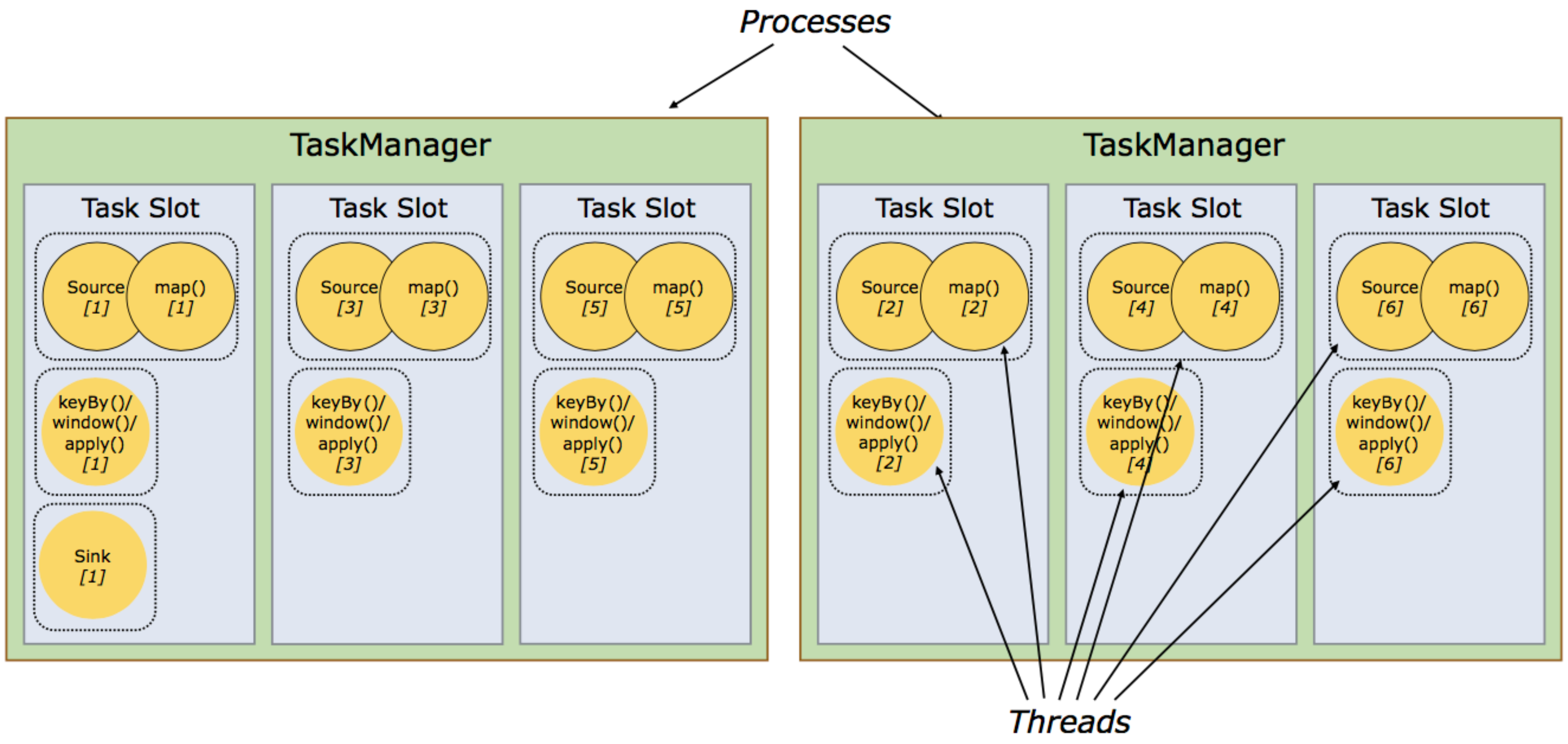
- dataStream
- count = input**.map** {m.split(”1")};
- **.keyBy**(count%2)**; //keys by odd count (1) or even count (0)**
- **.window(**TumblingEventTimeWindows.of**(**Time.seconds**(3)));**
- **.apply (new** CoGroupFunction **() {…});**
- **.reduce**(count**);**

SCHEDULING

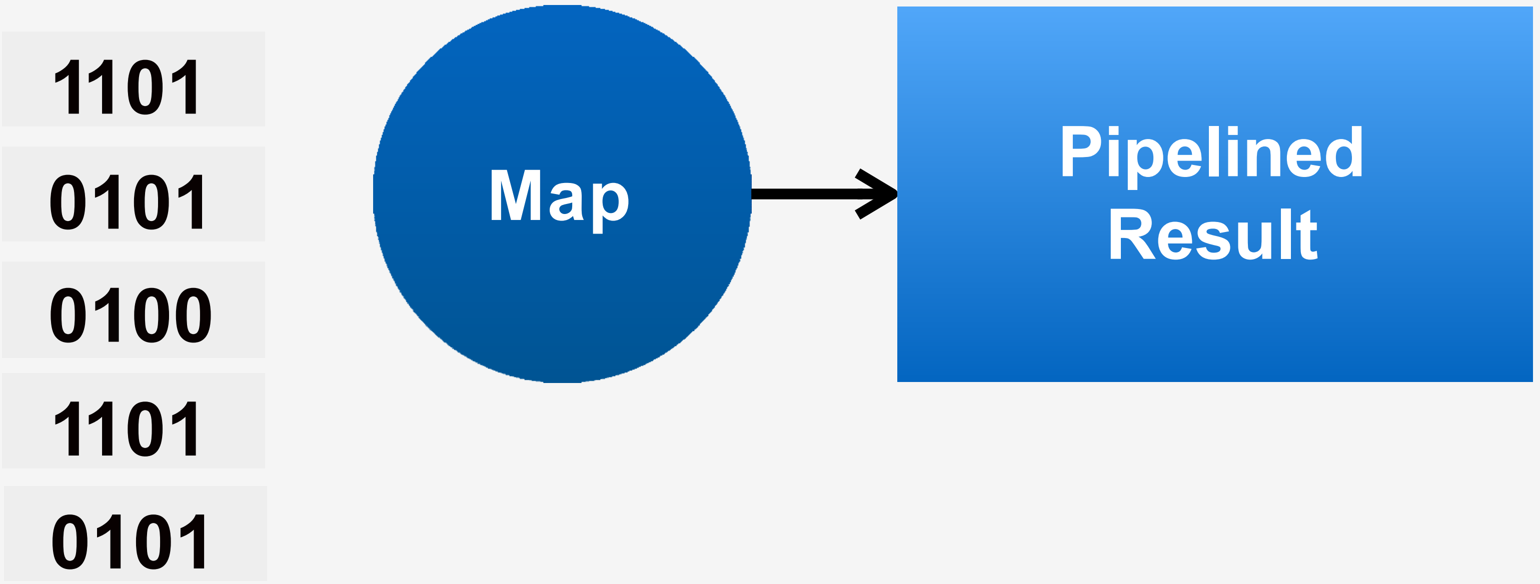


Source
map()
keyBy()/
window()/
apply()
Sink
Streaming Dataflow
(condensed view)
Task
Operator chain
Source
[1]
map()
[1]
keyBy()/
window()/
apply()
[1]
Subtask
(= thread)
Subtask
(= thread)
Sink
[1]
Source
[2]
map()
[2]
keyBy()/
window()/
apply()
[2]
Streaming Dataflow
(parallelized view)

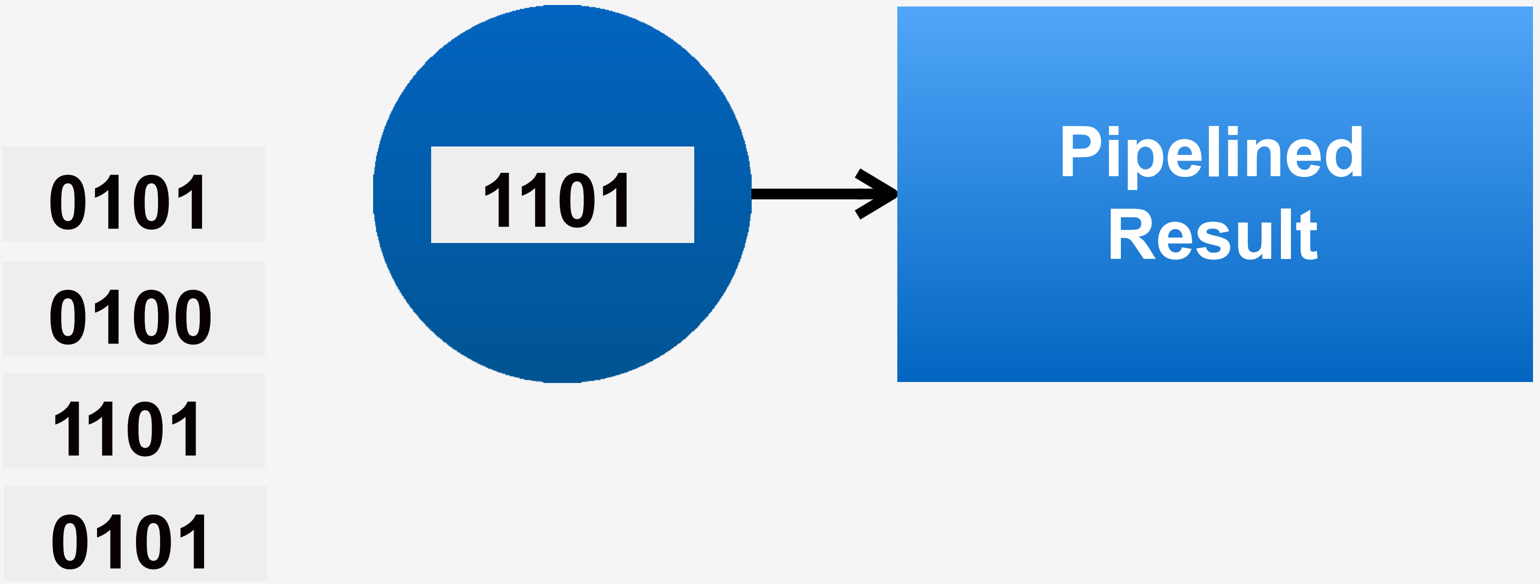
EXECUTION IN SLOTS



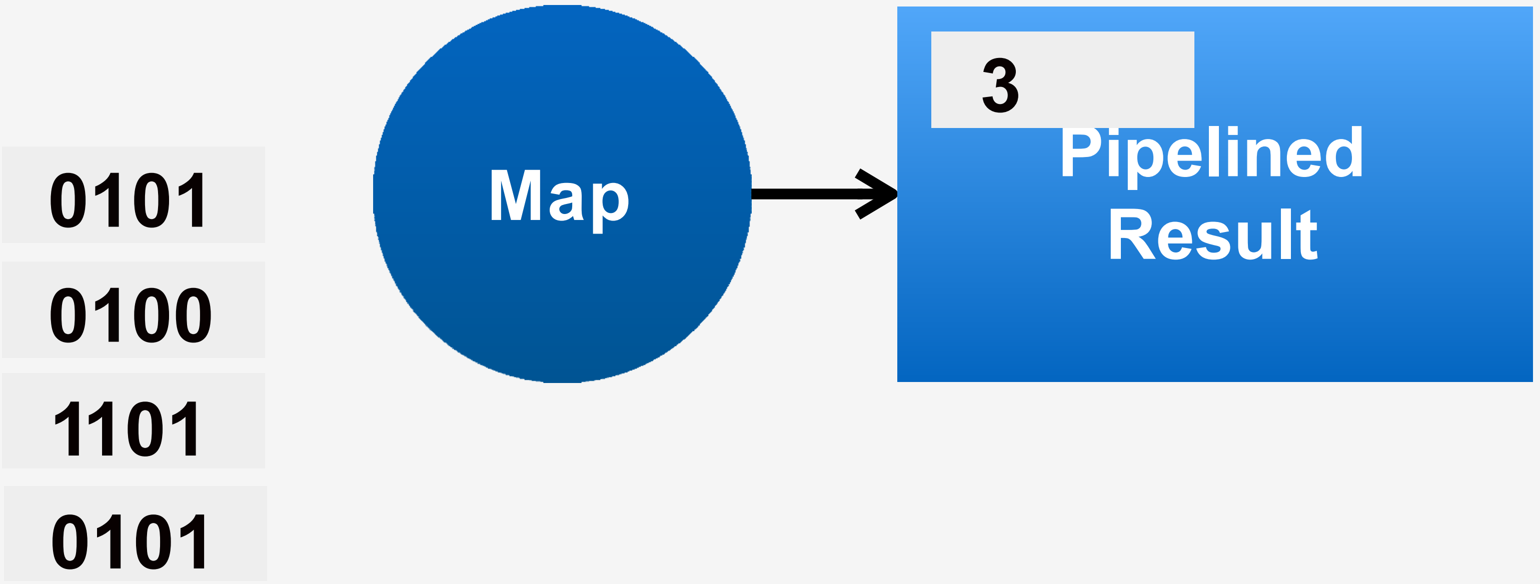
PIPELINED RESULTS



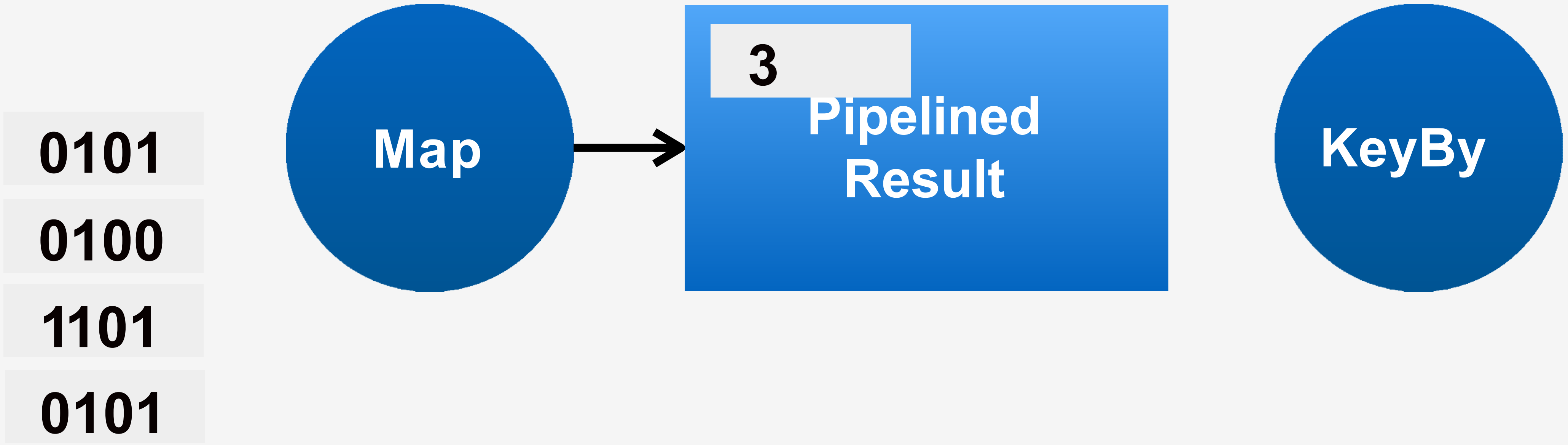
PIPELINED RESULTS



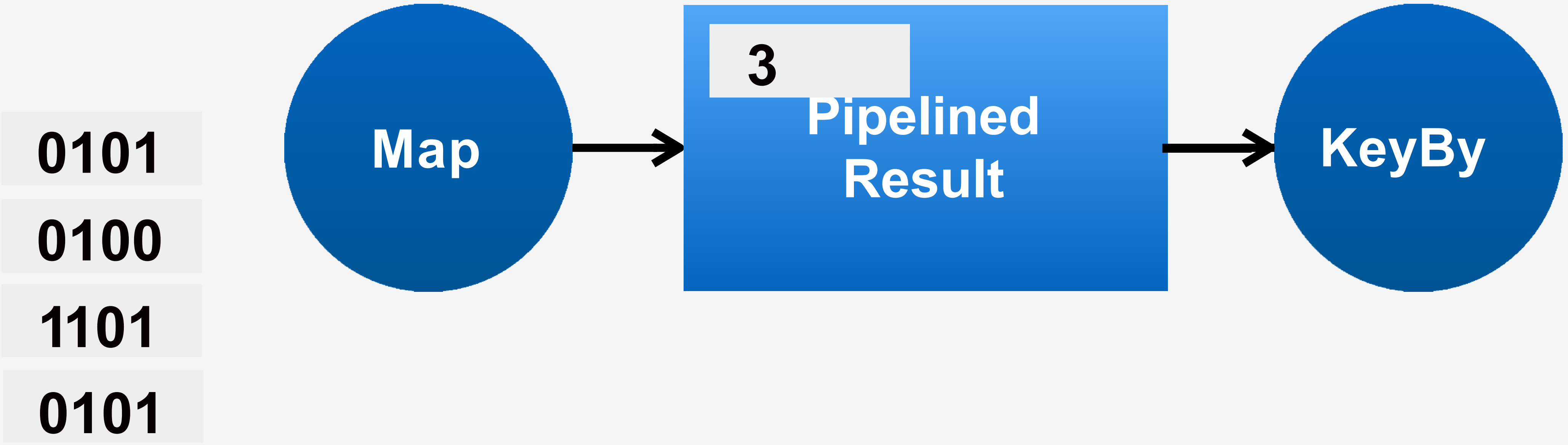
PIPELINED RESULTS



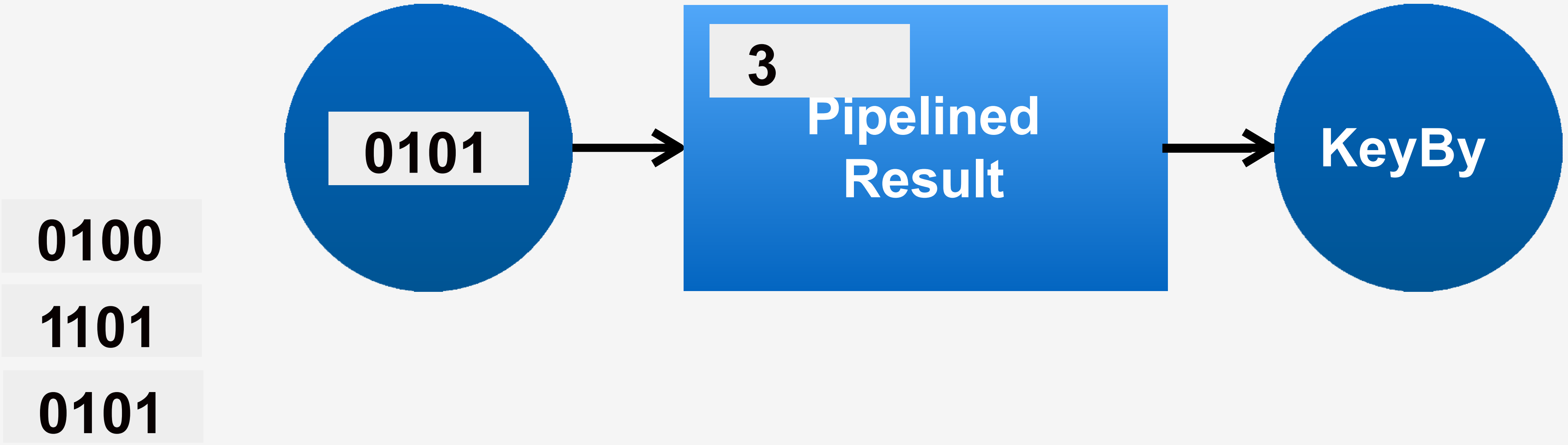
PIPELINED RESULTS



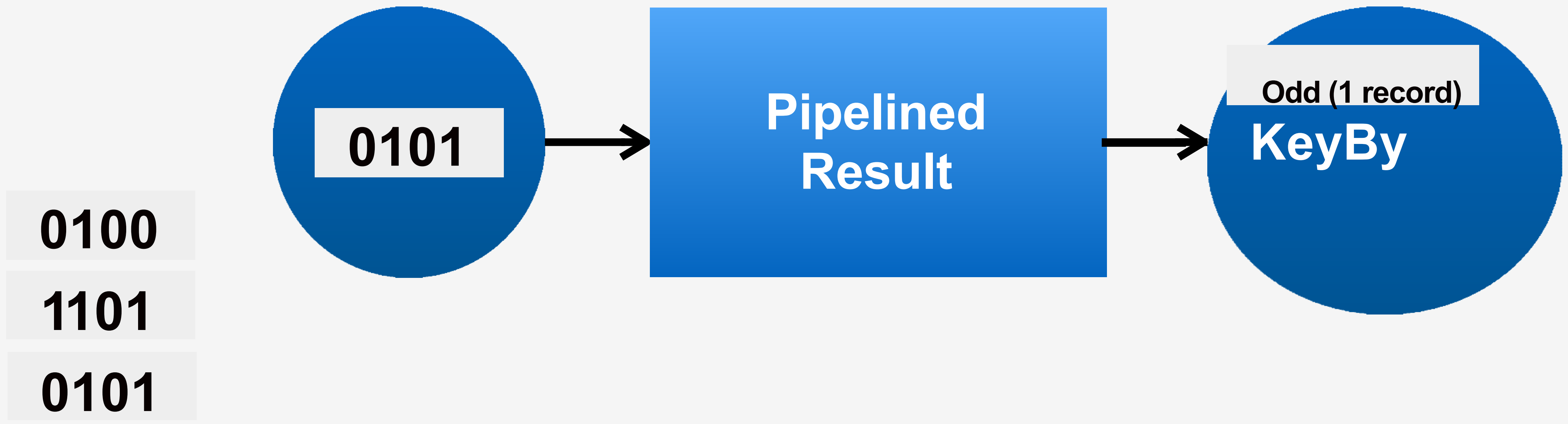
PIPELINED RESULTS



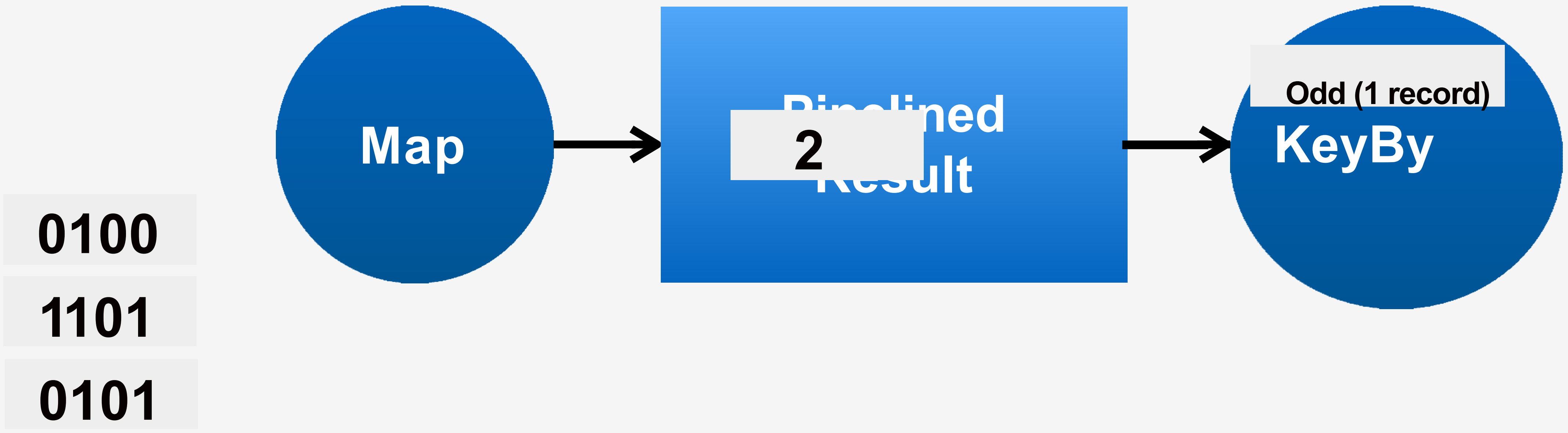
PIPELINED RESULTS



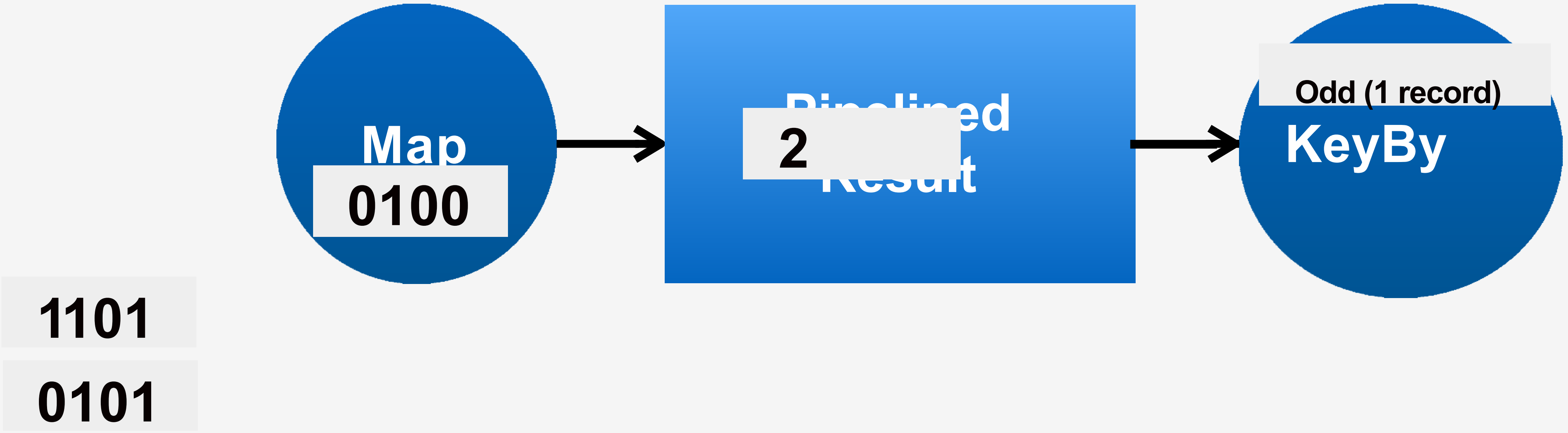
PIPELINED RESULTS



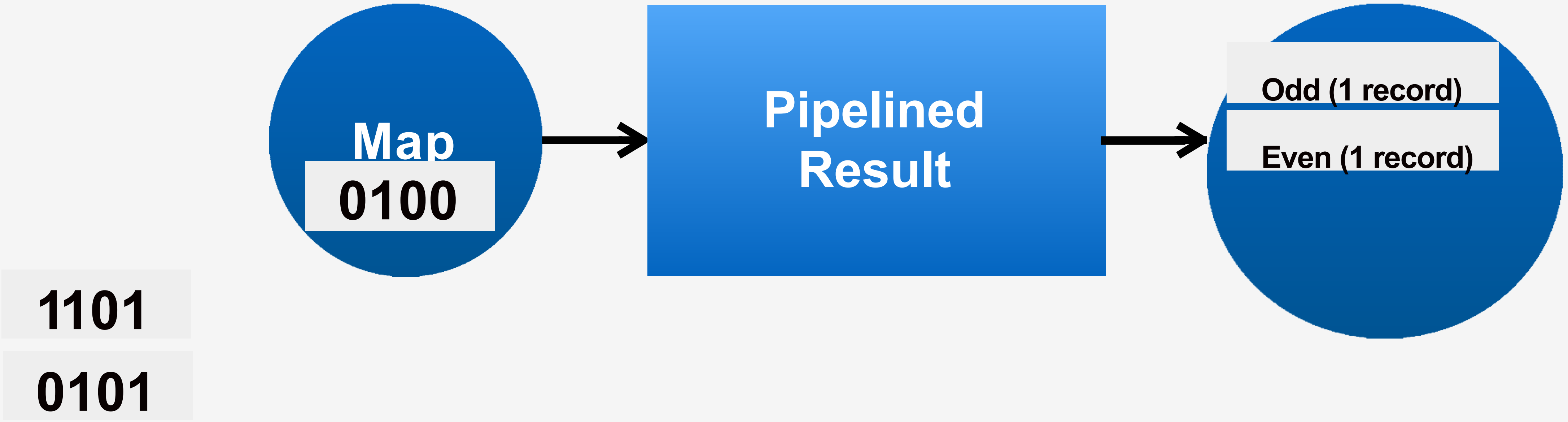
PIPELINED RESULTS



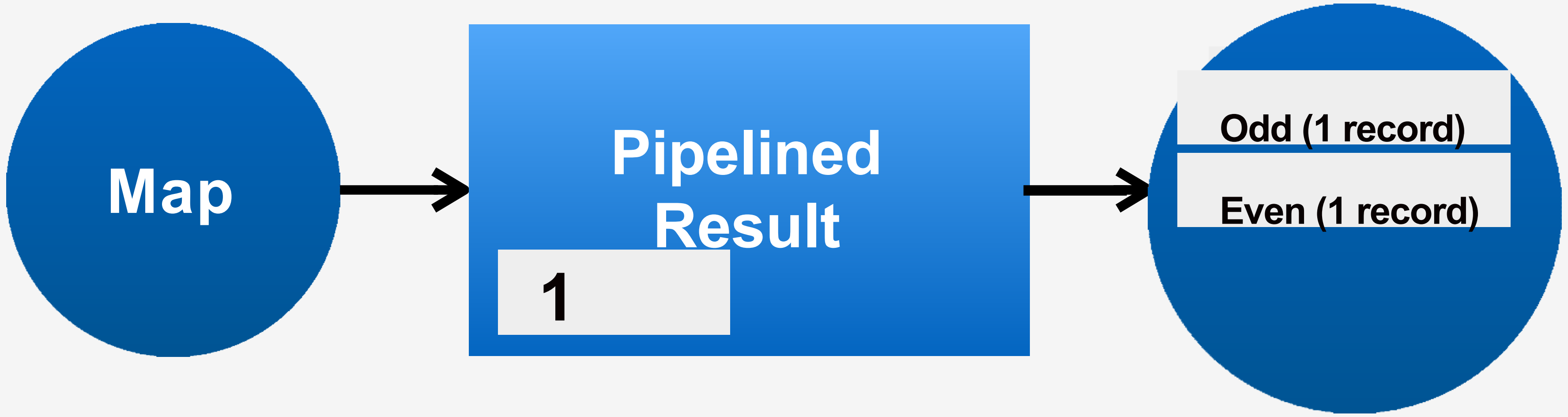
PIPELINED RESULTS



PIPELINED RESULTS



PIPELINED RESULTS



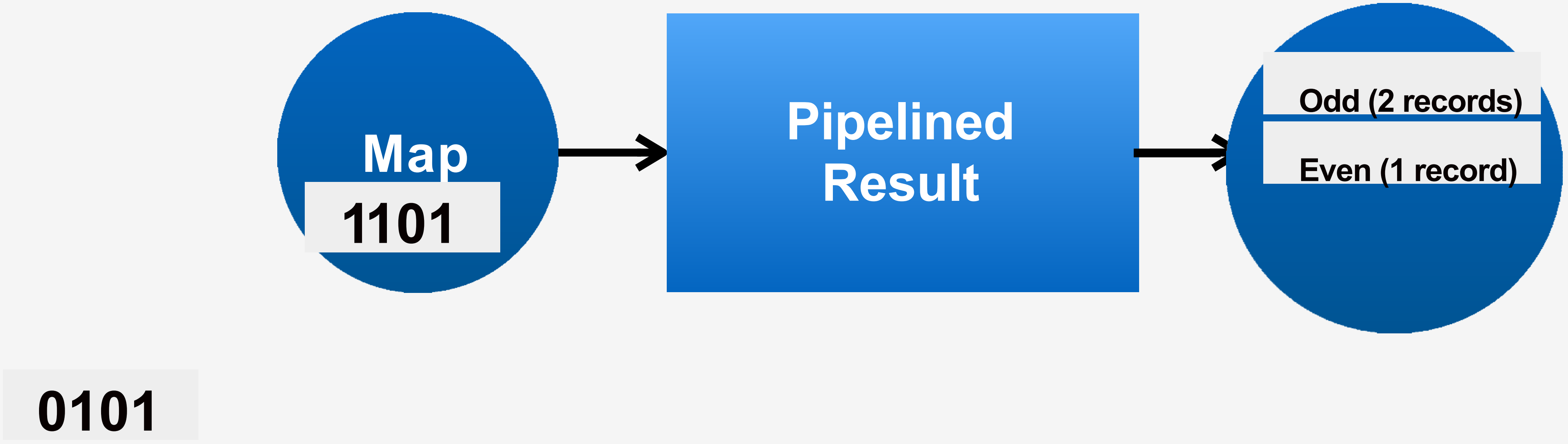
1101

0101

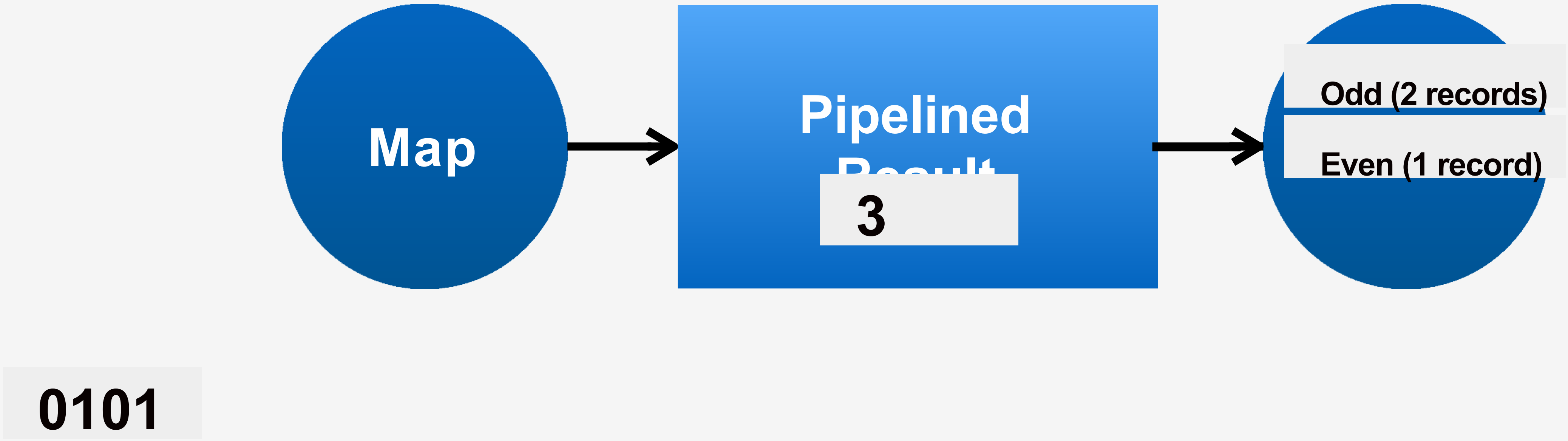
PIPELINED RESULTS

0101

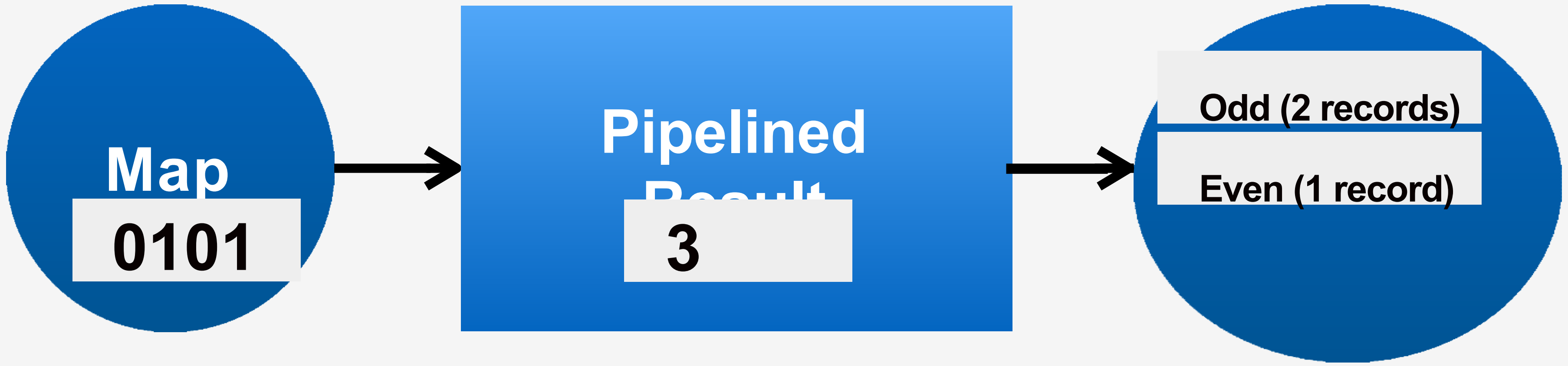
PIPELINED RESULTS



PIPELINED RESULTS

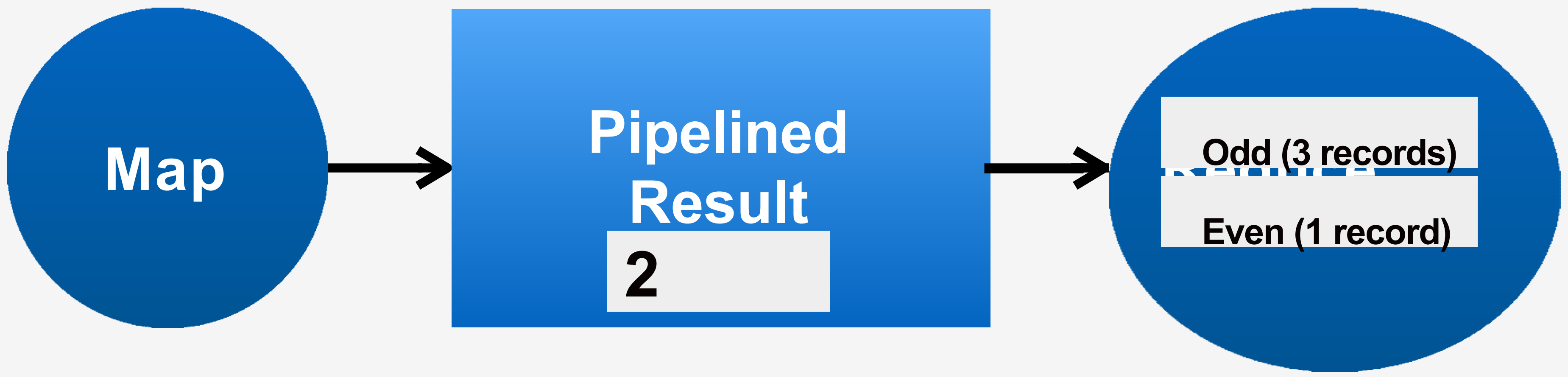


PIPELINED RESULTS

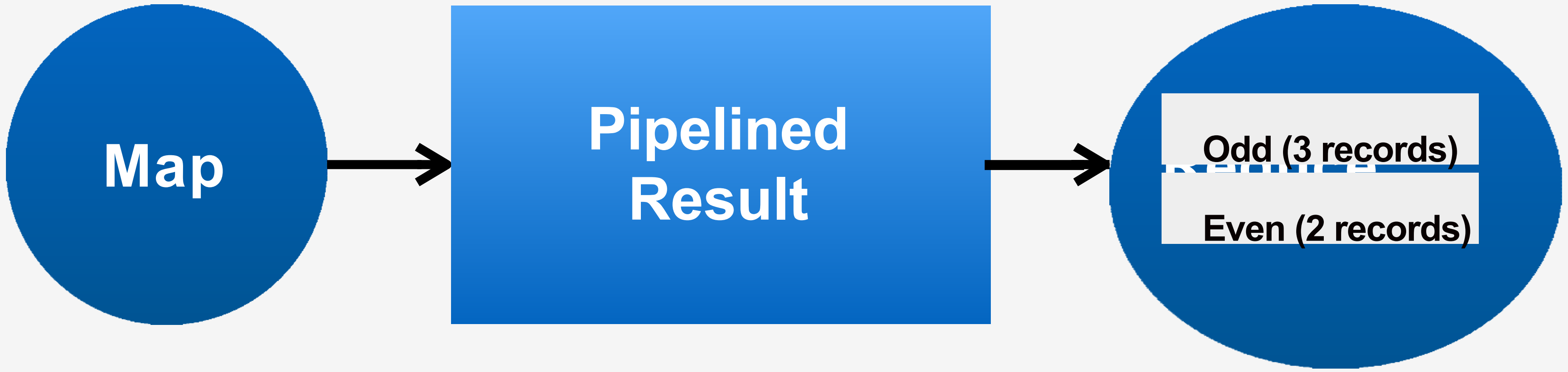


PIPELINED RESULTS

PIPELINED RESULTS



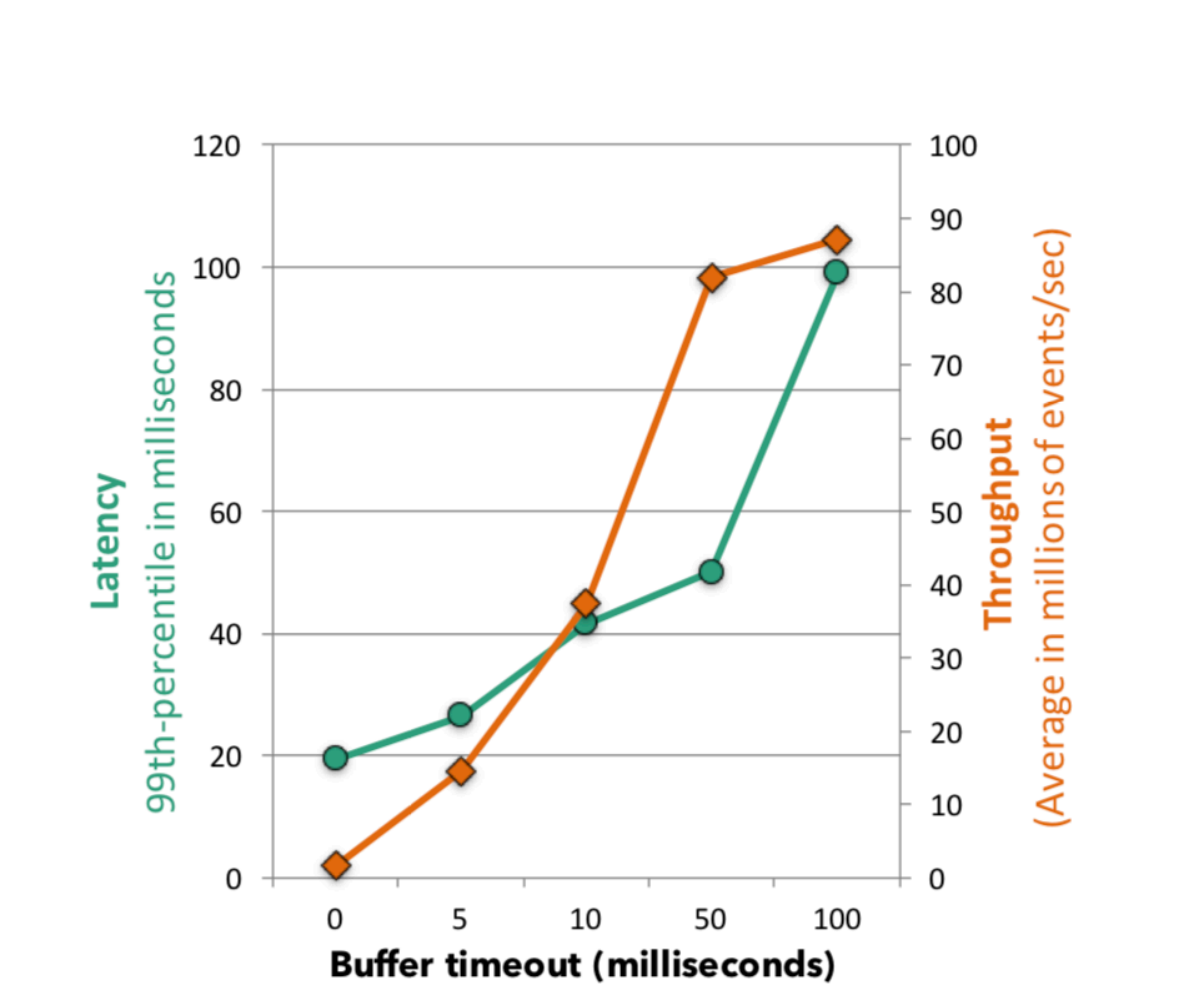
PIPELINED RESULTS



LATENCY AND THROUGHPUT

- When a data record is ready on the producer side, it is serialized and split into one or more buffers.
- A buffer is sent to a consumer either when it is full or when a timeout condition is reached.
- High throughput and low latency is achieved.

LATENCY AND THROUGHPUT

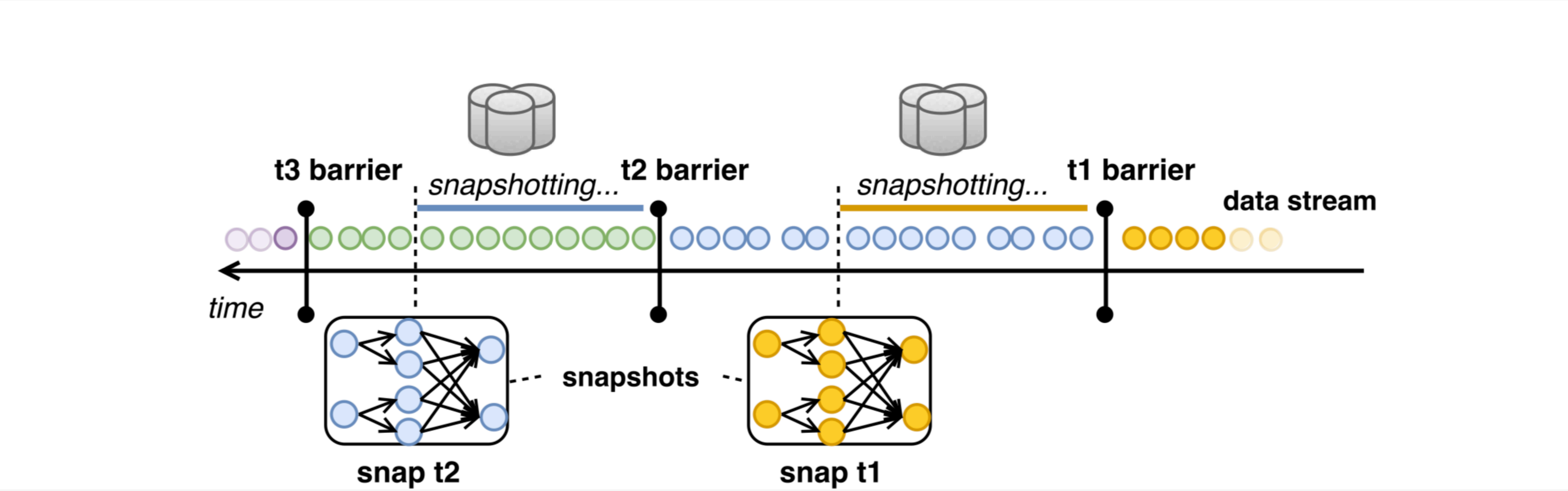


FAULT TOLERANCE

ASYNCHRONOUS BARRIER SNAPSHOTTING

- An operator receives barriers from upstream and first performs an alignment phase.
- Then, the operator writes its state to durable storage.
- Once the state has been backed up, the operator forwards the barrier downstream.
- Eventually, all operators will register a snapshot of their state and a global snapshot will be complete.

FAULT TOLERANCE



COMPARISON WITH NAIAD

- Both Flink and Naiad make use of snapshotting mechanism for fault tolerance.
- Both Apache Flink and Naiad frameworks combine batch processing and stream processing.
- Both the frameworks support high throughput and low latency.
- NAIAD performs iterative and incremental computations, while Flink performs primarily data processing of stream and batch data.

CONCLUSION

- Apache Flink is designed to perform both stream and batch analytics.
- The streaming API provides the means to keep recoverable state and to partition, transform, and aggregate data stream windows.
- Flink treats batch computations by optimizing their execution using a query optimizer.

QUESTIONS?