

CS4225/CS5425 Big Data Systems for Data Science

Streaming Processing

Ai Xin
School of Computing
National University of Singapore
aixin@comp.nus.edu.sg



Today's Plan

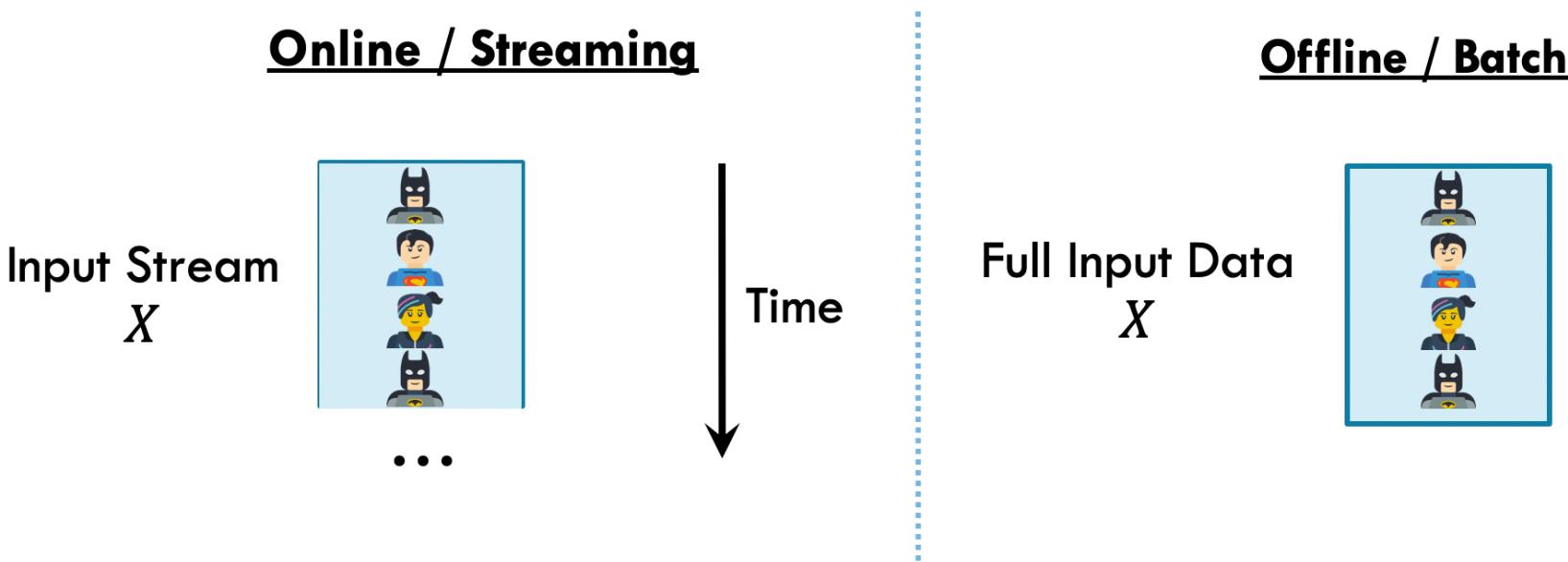
- **Introduction**

- **Spark**

- **Flink**

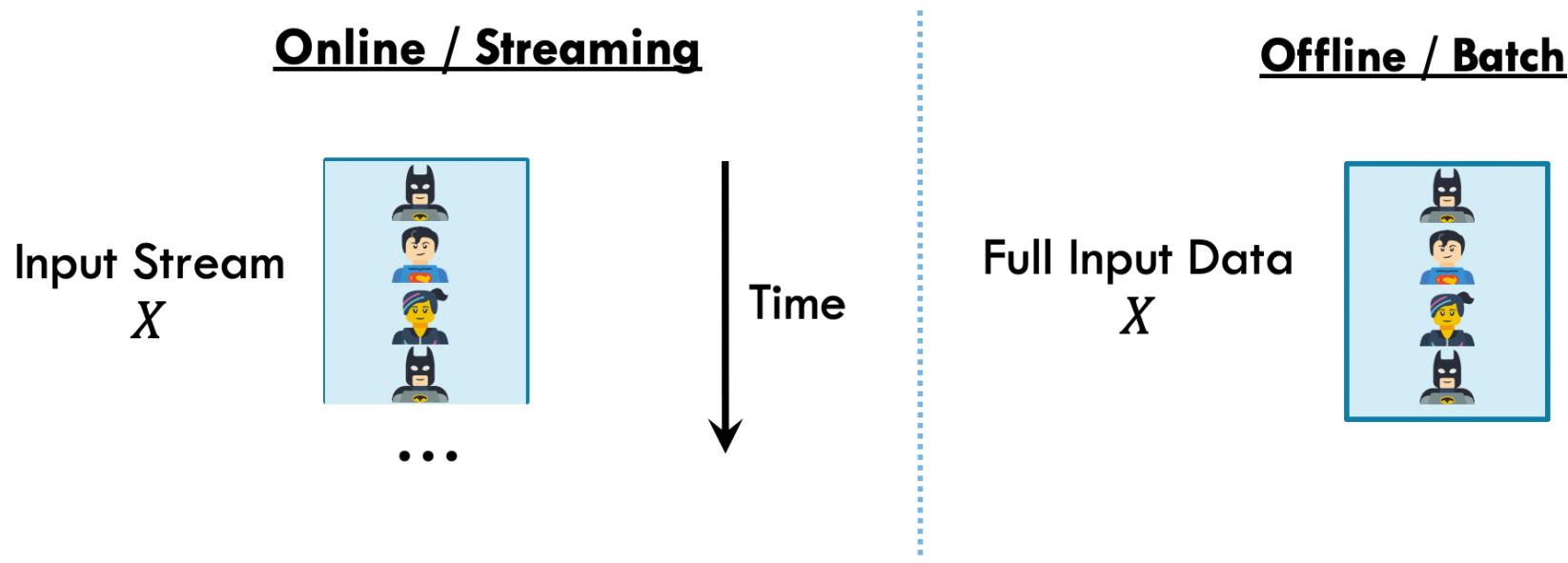
Streams: Motivation

- In many settings, the data is **arriving over time**; it is not received all at once
 - Streaming (or “online”) approaches are designed to process their input **as it is received**
 - This is in contrast to offline or batch approaches that operate on the full dataset, all at once



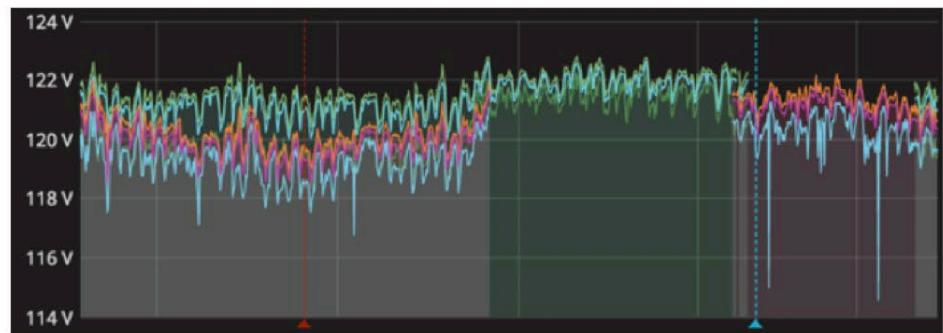
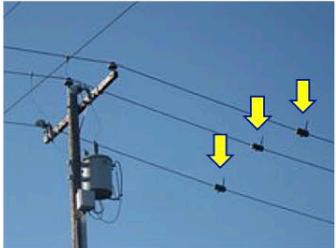
Streaming Data

- Input elements enter at a rapid rate from **input ports** (e.g. receiving data from a sensor, from a TCP connection, from a file stream, or from a message queue)
 - Elements of the stream are sometimes referred to as ‘tuples’
 - The stream is potentially infinite; **the system cannot store the entire stream accessibly** (due to limited memory)



Example Applications

- In many real settings, data is **high volume** and **constantly arriving over time**



Sensor data (industrial, environmental, medical, etc)

Example Applications

- In many real settings, data is **high volume** and **constantly arriving over time**



Online user activity data

Example Applications

- In many real settings, data is **high volume** and **constantly arriving over time**



Financial Transactions
(e.g. credit card fraud detection)



Stock Trades Data
(e.g. trades, orders, prices)

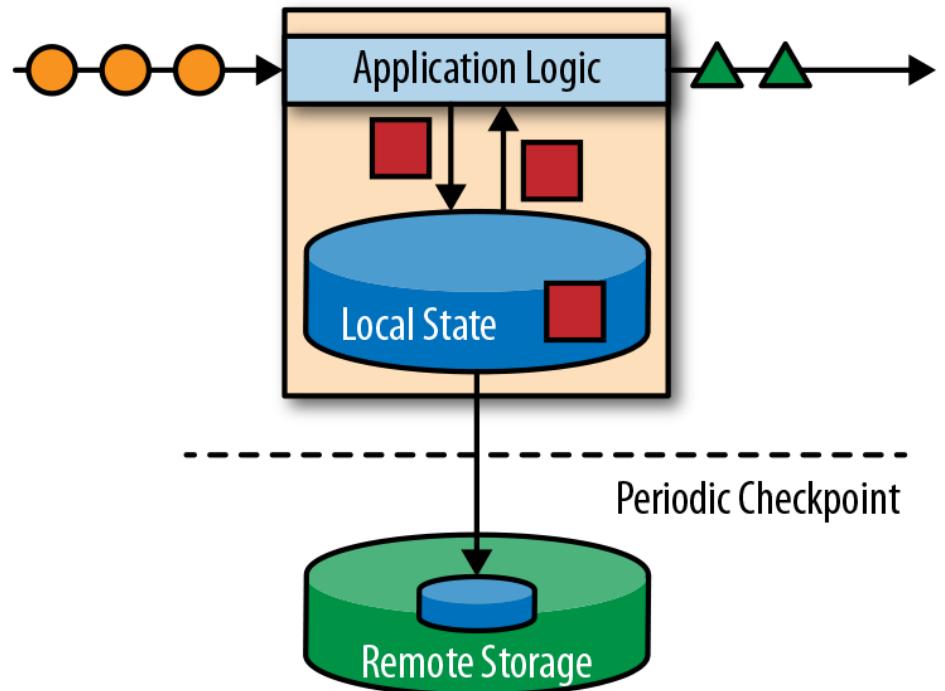
02 Sep 2020	OMO STORE		\$S\$3.30
02 Sep 2020	FACEBK	V2 650-5434800 CA USD2.00	\$S\$2.81
02 Sep 2020	FACEBK	! 650-5434800 CA USD3.00	\$S\$4.22
02 Sep 2020	FACEBK	650-5434800 CA USD2.00	\$S\$2.81
02 Sep 2020	FACEBK	650-5434800 CA USD3.00	\$S\$4.22
02 Sep 2020	FACEBK	650-5434800 CA USD5.00	\$S\$7.03
02 Sep 2020	FACEBK	? 650-5434800 CA USD50.00	\$S\$70.45
02 Sep 2020	FACEBK	' 650-5434800 CA USD10.00	\$S\$14.10
03 Sep 2020	FACEBK	.50-5434800 CA USD50.00	\$S\$70.45
03 Sep 2020	FACEBK	650-5434800 CA USD75.00	\$S\$105.84
03 Sep 2020	FACEBK	650-5434800 CA USD1.33	
03 Sep 2020	OMO STORE		\$S\$3.00
03 Sep 2020	GRAB*GRAB*	[REDACTED] SINGAPORE SG	\$S\$50.00

Do you need any help?



Stateful Stream Processing

- not just perform trivial record-at-a-time transformations
- the ability to store and access intermediate data
- state can be stored and accessed in many different places including program variables, local files, or embedded or external databases



Today's Plan

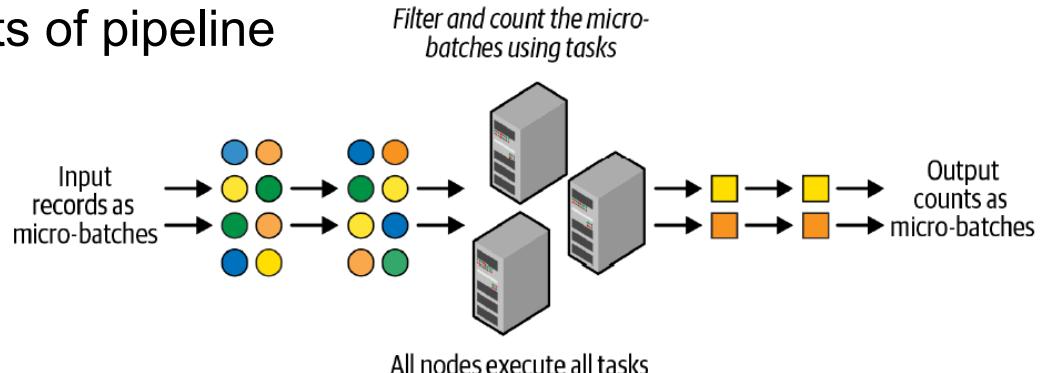
- Introduction

- Spark

- Flink

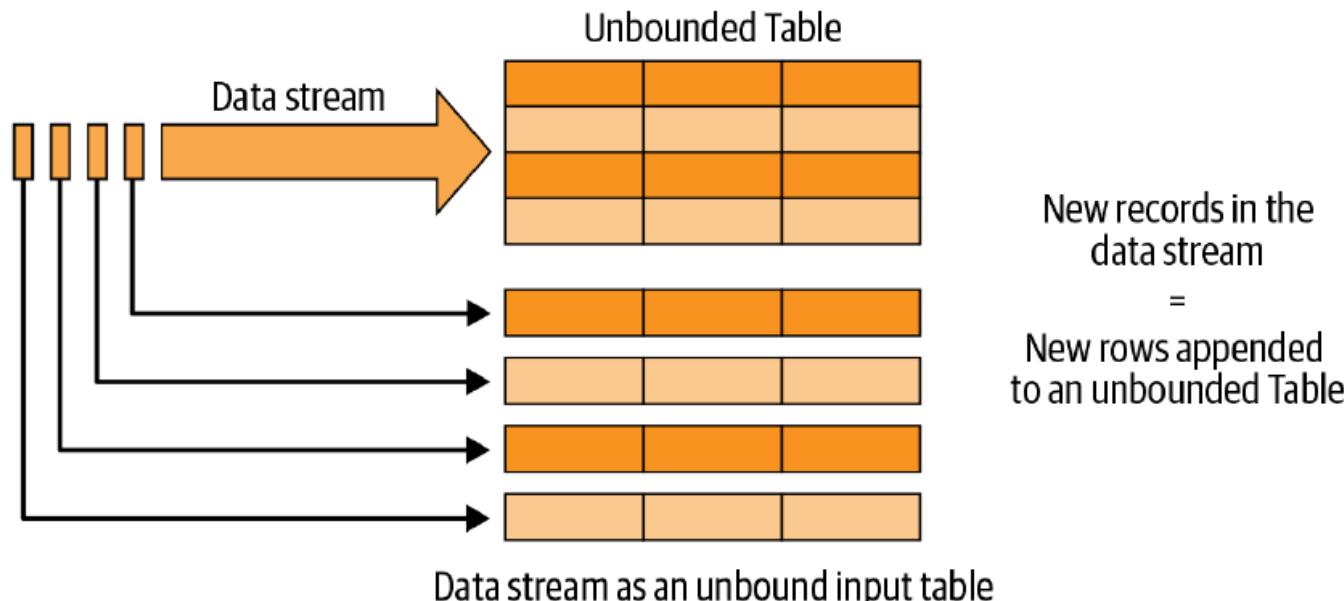
Micro-Batch Stream Processing

- Structured Streaming uses a micro-batch processing model
 - divides the data from the input stream into micro batches
 - each batch is processed in the Spark cluster in a distributed manner
 - small deterministic tasks generate the output in micro-batches
- Advantages
 - quickly and efficiently recover from failures
 - deterministic nature ensures end-to-end exactly-once processing guarantees
- Disadvantages: latencies of a few seconds
 - This is actually OK for many applications
 - Application may incur more than a few seconds delay in other parts of pipeline

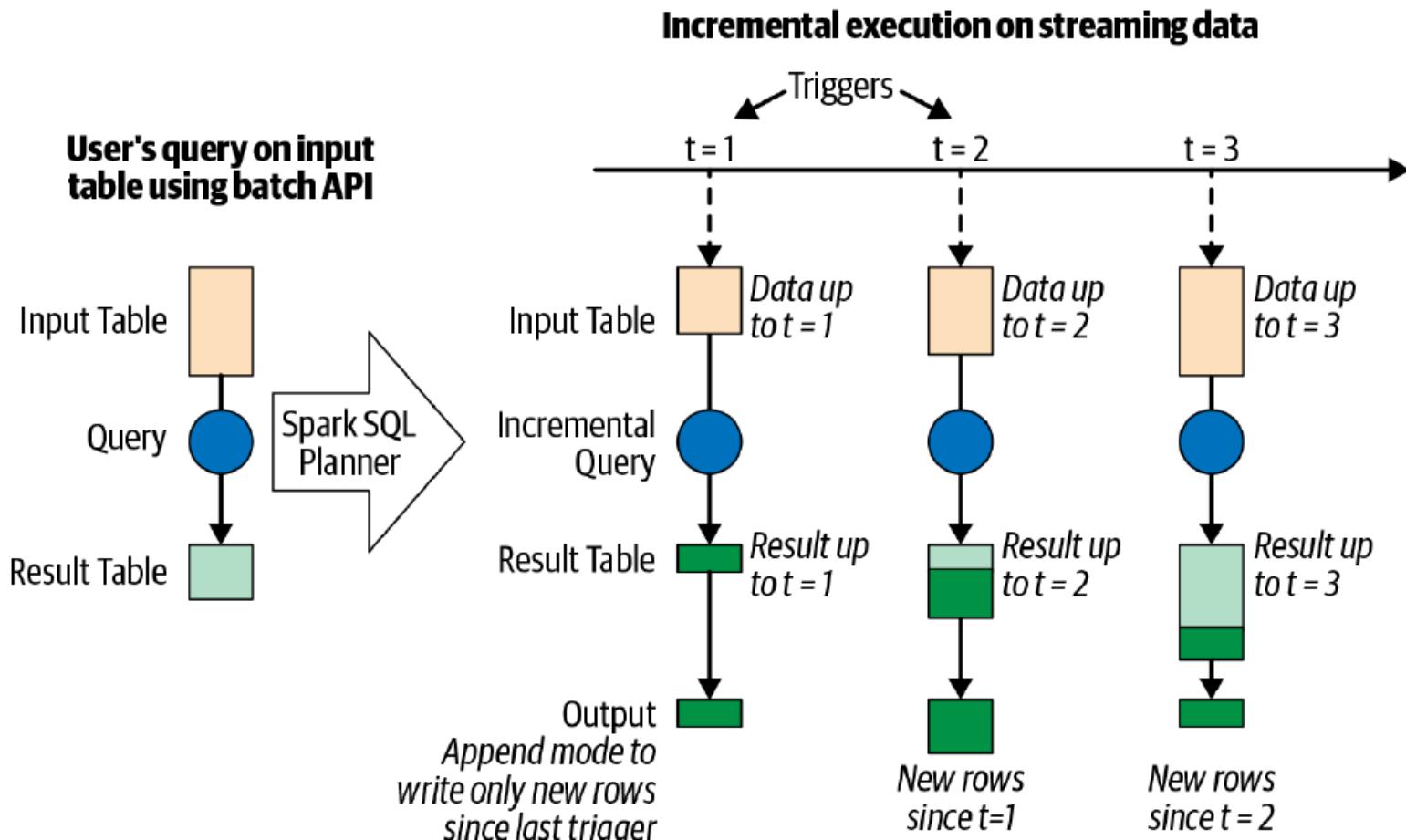


The Philosophy of Structured Streaming

- For developers, writing stream processing pipelines should be as easy as writing batch pipelines.
 - A single, unified programming model and interface for batch and stream processing
 - A broader definition of stream processing
- The Structured Streaming programming model: data stream as an unbounded table



The Structured Streaming processing model



Users express query on streaming data using a batch-like API and Structured Streaming incrementalizes them to run on streams.

Five Steps to Define a Streaming Query

- Step 1: Define input sources
- Step 2: Transform data
- Step 3: Define output sink and output mode
 - Output writing details (where and how to write the output)
 - Processing details (how to process data and how to recover from failures)
- Step 4: Specify processing details
 - Triggering details: when to trigger the discovery and processing of newly available streaming data.
 - Checkpoint Location: store the streaming query process info for failure recovery
- Step 5: Start the query

Practical 3: Spark Streaming

Practical 3: Spark Streaming

This notebook provides a structure streaming example using Spark.

Source: <https://github.com/databricks/Spark-The-Definitive-Guide>

Cmd 2

```
1 spark.conf.set("spark.sql.shuffle.partitions", 5)
```

Command took 0.11 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 4:24:58 PM on Test

Cmd 3

```
1 static = spark.read.json("/databricks-datasets/definitive-guide/data/activity-data/")
2 dataSchema = static.schema
3
```

▶ (3) Spark Jobs

▶ 📄 static: pyspark.sql.dataframe.DataFrame = [Arrival_Time: long, Creation_Time: long ... 8 more fields]

Command took 38.53 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 4:25:06 PM on Test

Cmd 4

```
1 streaming = spark.readStream.schema(dataSchema).option("maxFilesPerTrigger", 1) \
2   .json("/databricks-datasets/definitive-guide/data/activity-data")
3
```

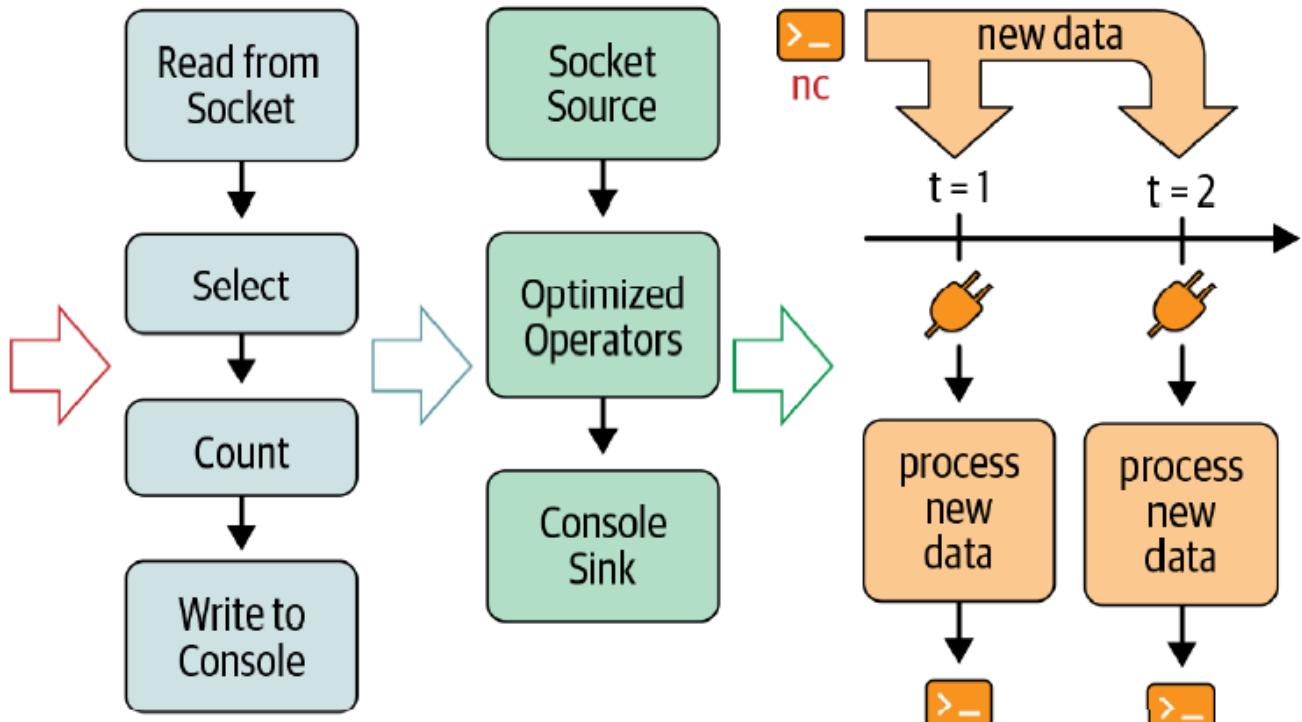
▶ 📄 streaming: pyspark.sql.dataframe.DataFrame = [Arrival_Time: long, Creation_Time: long ... 8 more fields]

Command took 0.52 seconds -- by aixin@comp.nus.edu.sg at 2/13/2023, 4:03:19 PM on Test

Cmd 5

Incremental execution of streaming queries

```
val lines = spark  
  .readStream  
  .format("socket")  
  ...  
  
val words = lines  
  .select(...)  
  
val counts = words  
  .groupByKey("word")  
  .count()  
  
val streamingQuery =  
  counts  
  .writeStream  
  .format("console")  
  ...
```



User Code

Logical Plan

Optimized Plan

Series of Incremental Execution Plans

Data Transformation

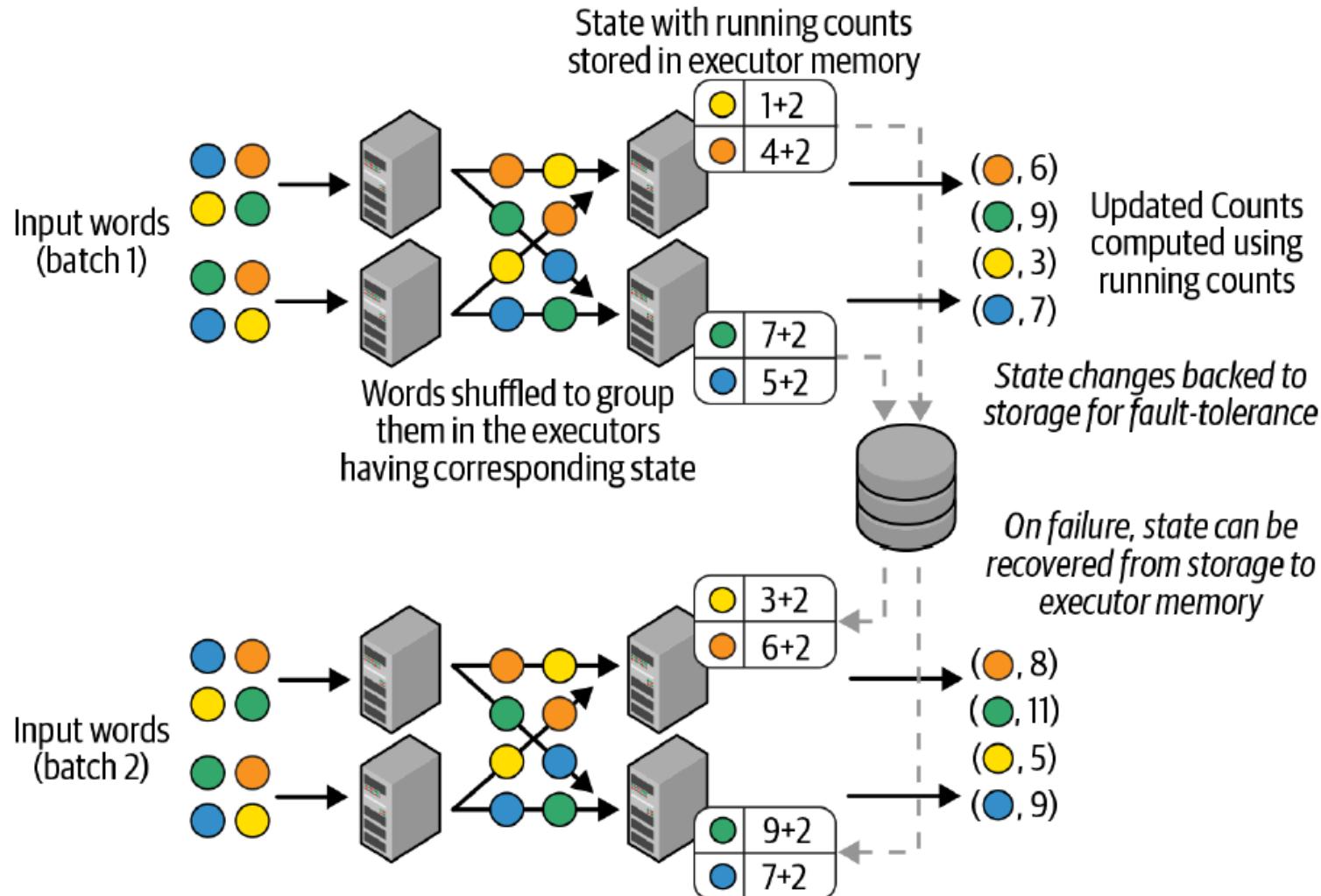
- Stateless Transformation

- Process each row individually without needing any information from previous rows
- Projection operations: `select()`, `explode()`, `map()`, `flatMap()`
- Selection operations: `filter()`, `where()`

- Stateful Transformation

- A simple example: `DataFrame.groupBy().count()`
- In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch
- The partial count communicated between plans is the state
- The state is maintained in the memory of the Spark executors and is checkpointed to the configured location to tolerate failures.

Distributed state management in Structured Streaming



Stateful Streaming Aggregations

- Aggregations Not Based on Time

- Global aggregations

```
runningCount = sensorReadings.groupBy().count()
```

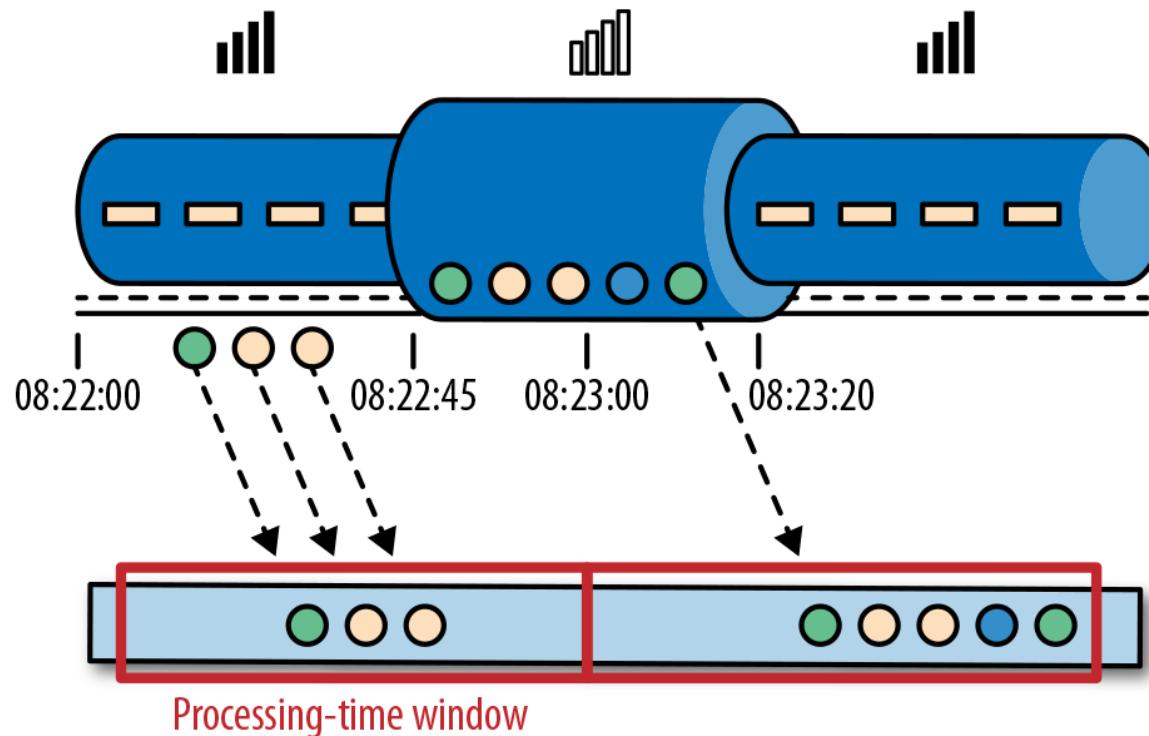
- Grouped aggregations

```
baselineValues = sensorReadings.groupBy("sensorId").mean("value")
```

- All built-in aggregation functions in DataFrames are supported
 - sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct(), and etc.

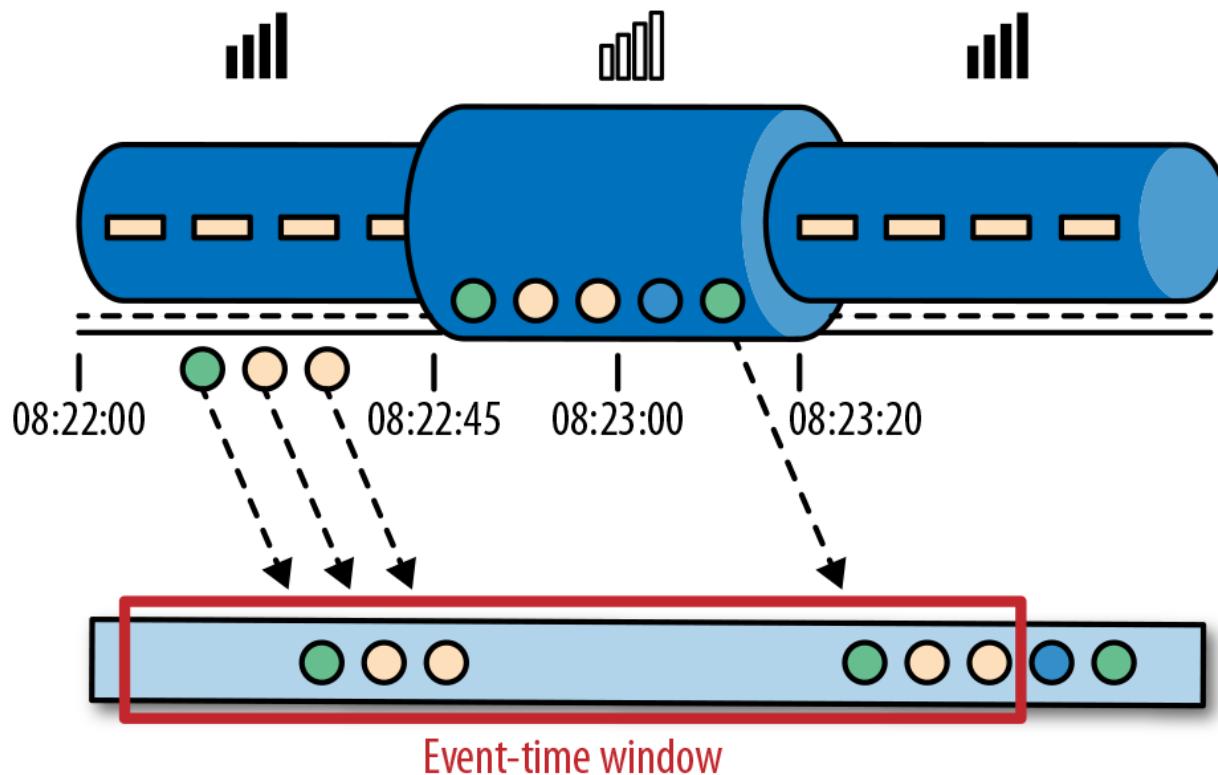
Time Semantics

- Processing Time: the time of stream processing machine



Time Semantics

- Event Time: the time an event actually happened



Time Semantics

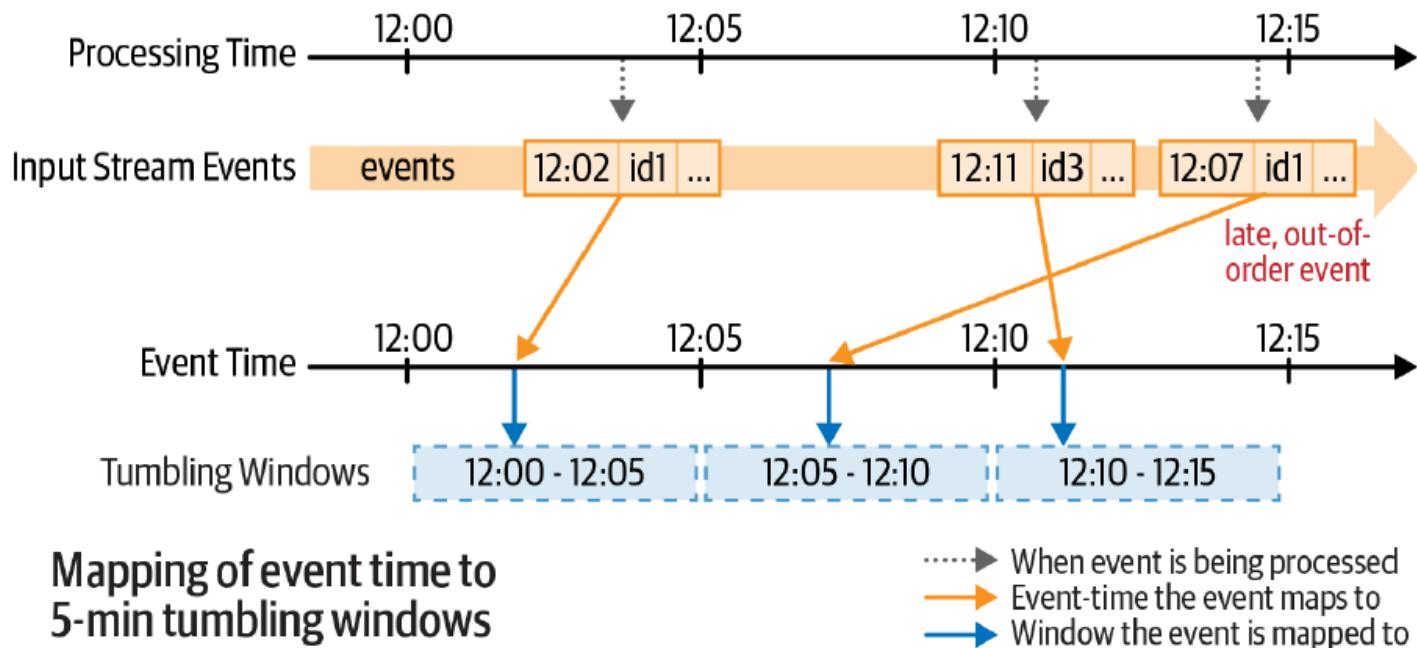
- Event time completely decouples the processing speed from the results.
- Operations based on event time are predictable and their results are deterministic.
- An event time window computation will yield the same result no matter how fast the stream is processed or when the events arrive at the operator.
- how long do we have to wait before we can be certain that we have received all events that happened before a certain point of time?

watermarks

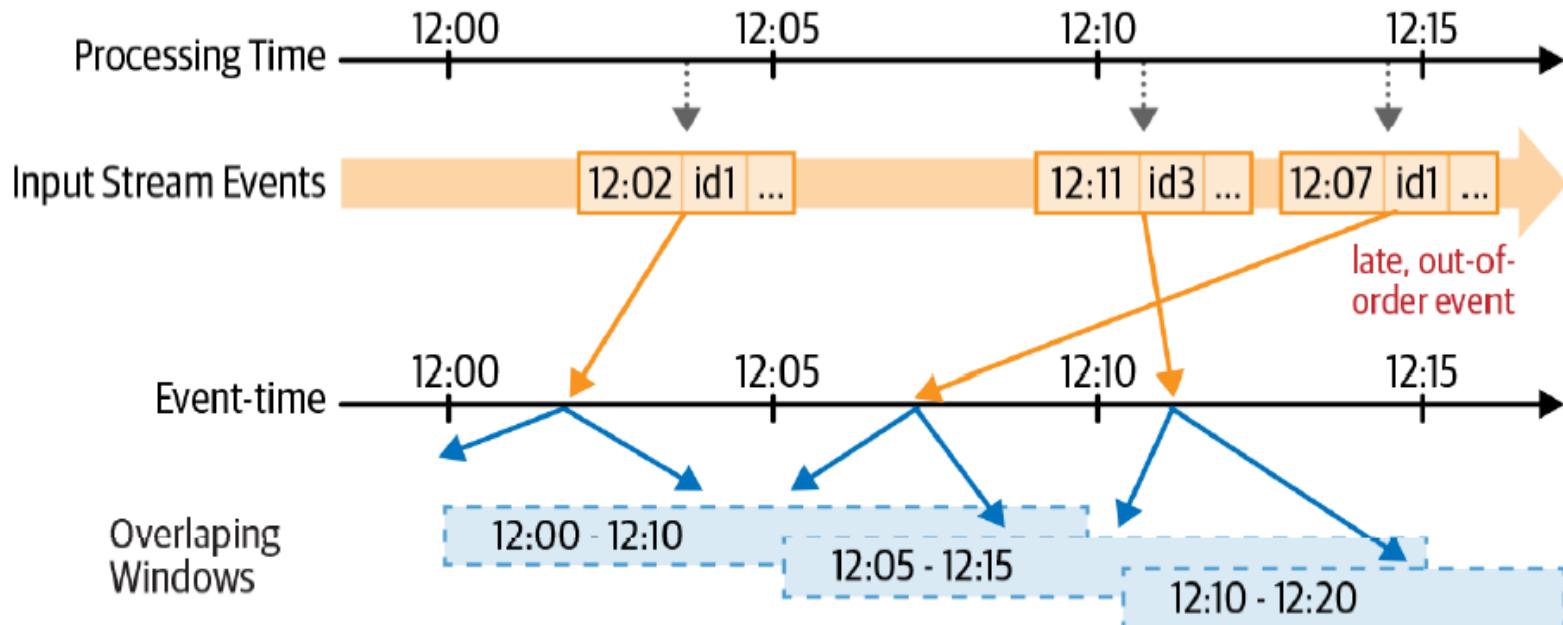
Stateful Streaming Aggregations

- Aggregations with Event-Time Windows

```
(sensorReadings
    .groupBy("sensorId", window("eventTime", "5 minute"))
    .count())
```



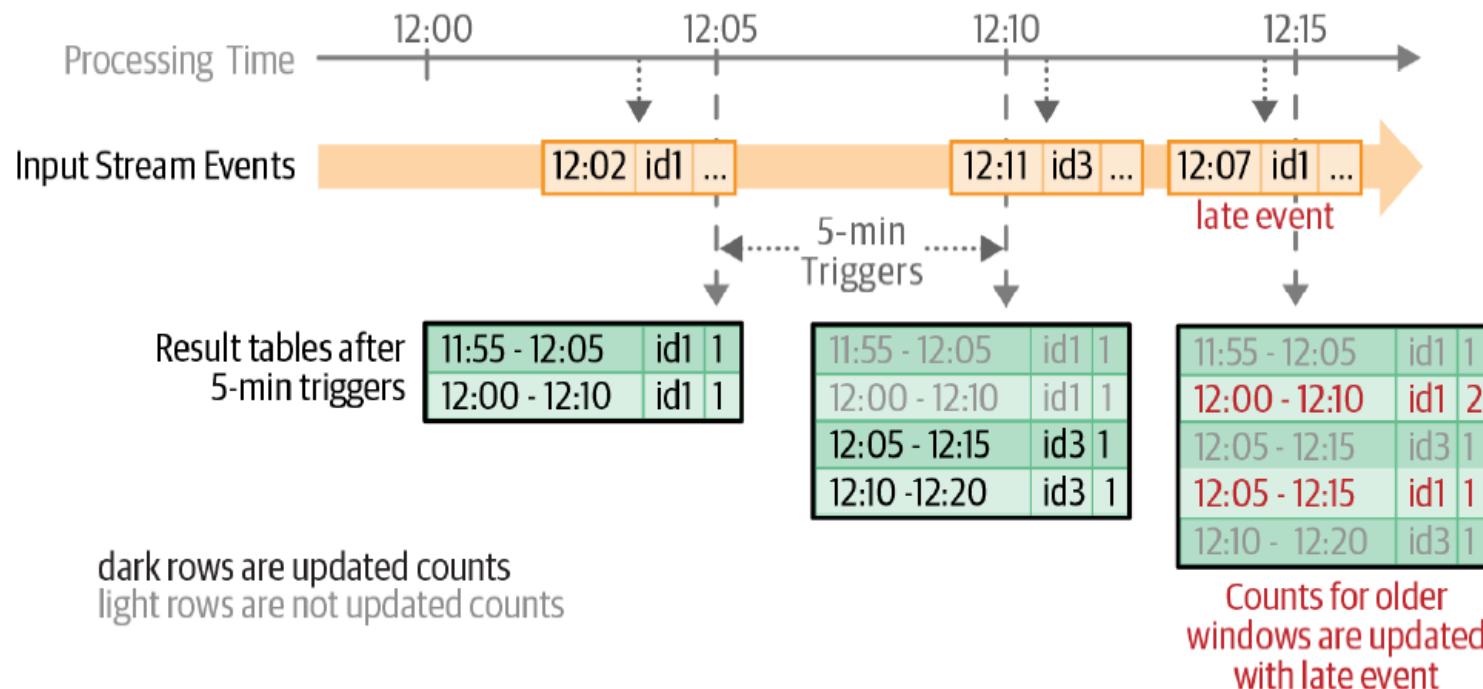
```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```



Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins

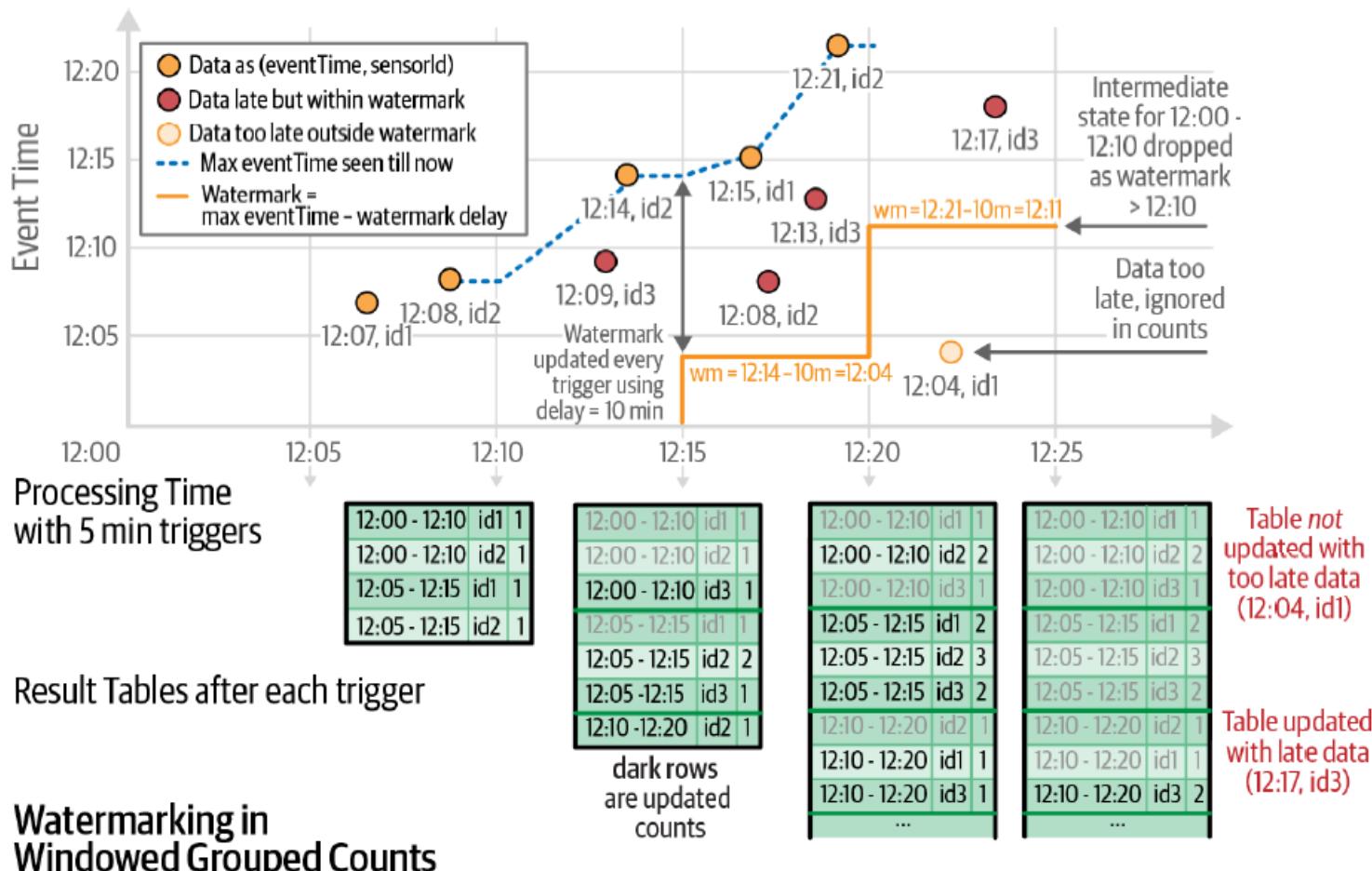
- → When event is being processed
- Event-time the event maps to
- Window the event is mapped to

- Updated counts in the result table after each five-minute trigger



Handling Late Data with Watermarks

```
(sensorReadings
    .withWatermark("eventTime", "10 minutes")
    .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
    .count())
```



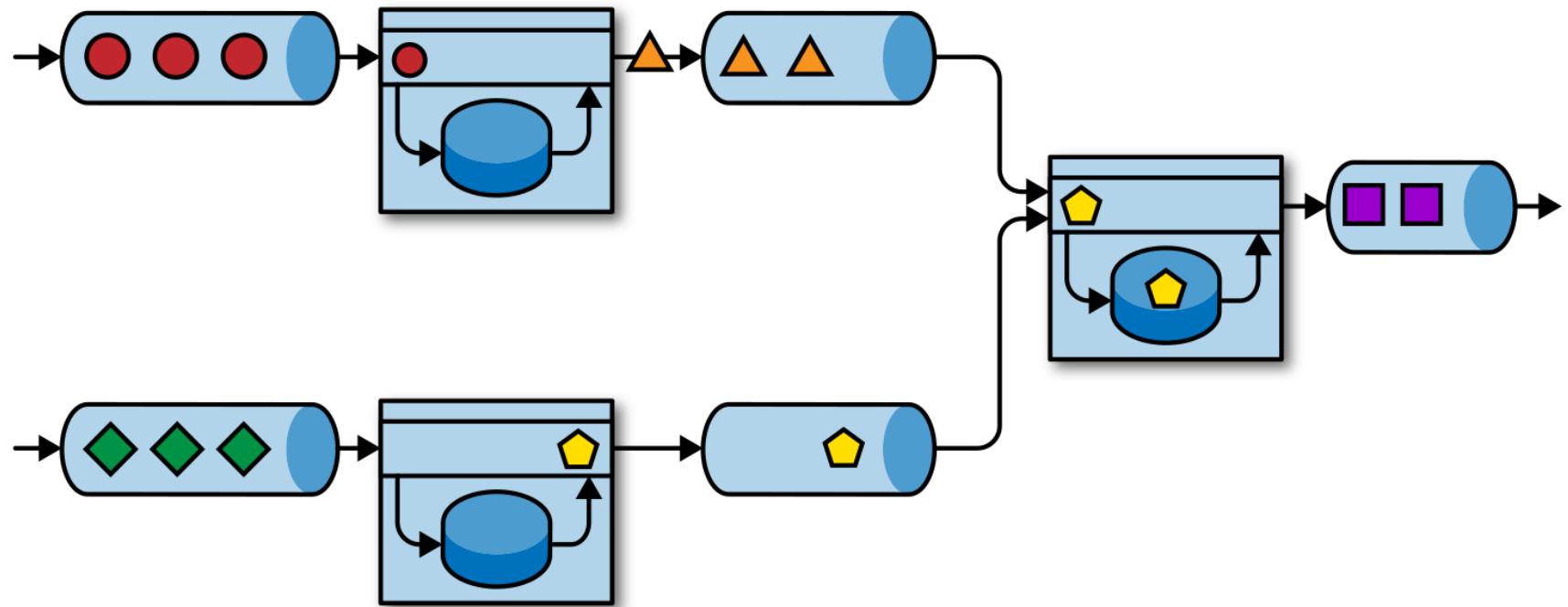
Performance Tuning

- Besides tuning Spark SQL engine, a few other considerations
 - Cluster resource provisioning appropriately to run 24/7
 - Number of partitions for shuffles to be set much lower than batch queries
 - Setting source rate limits for stability
 - Multiple streaming queries in the same Spark application

Today's Plan

- Introduction
- Spark
- Flink

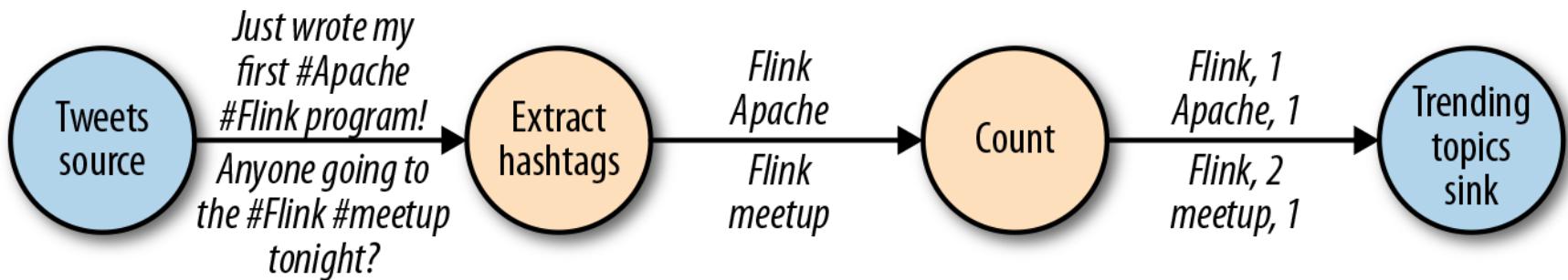
Event-driven streaming application



Dataflow Model

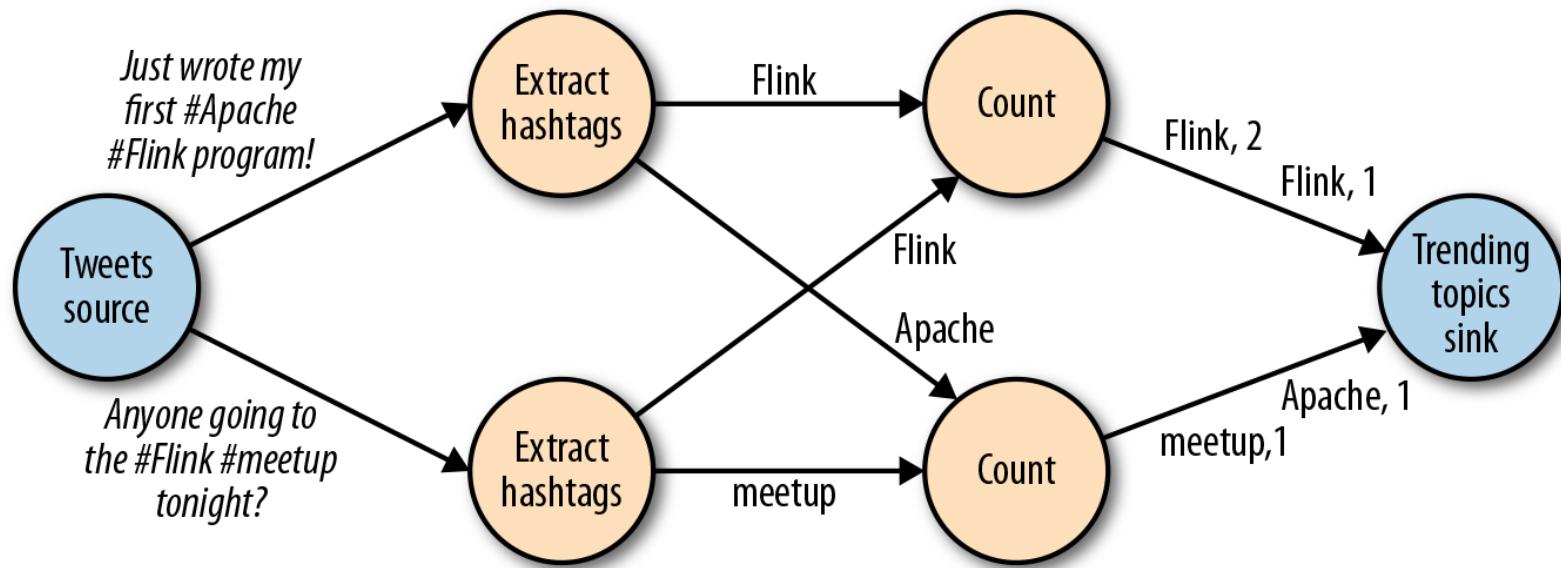
- Dataflow Graph

- A logical dataflow graph to continuously count hashtags (nodes represent operators and edges denote data dependencies)



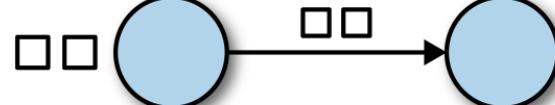
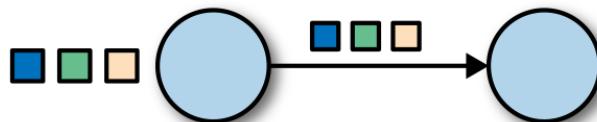
Dataflow Model

- A physical dataflow plan for counting hashtags (nodes represent tasks)

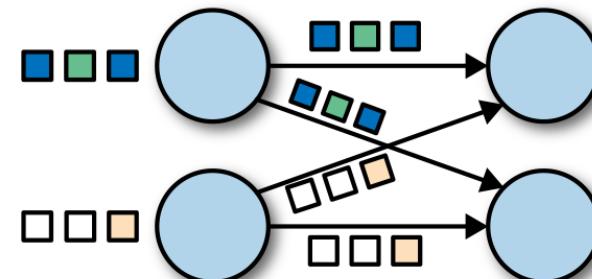


Dataflow Model

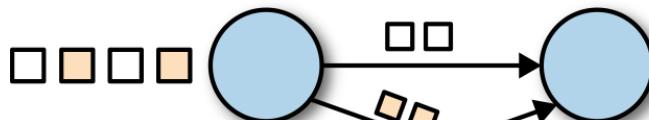
- Data Exchange Strategies



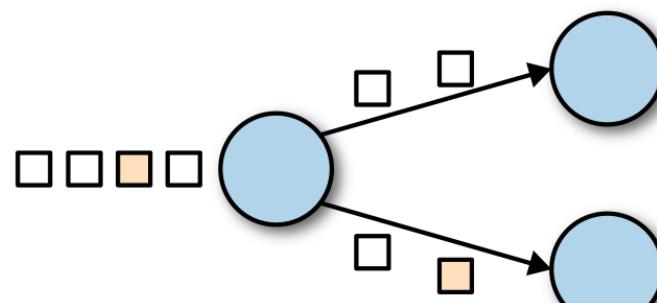
Forward



Broadcast



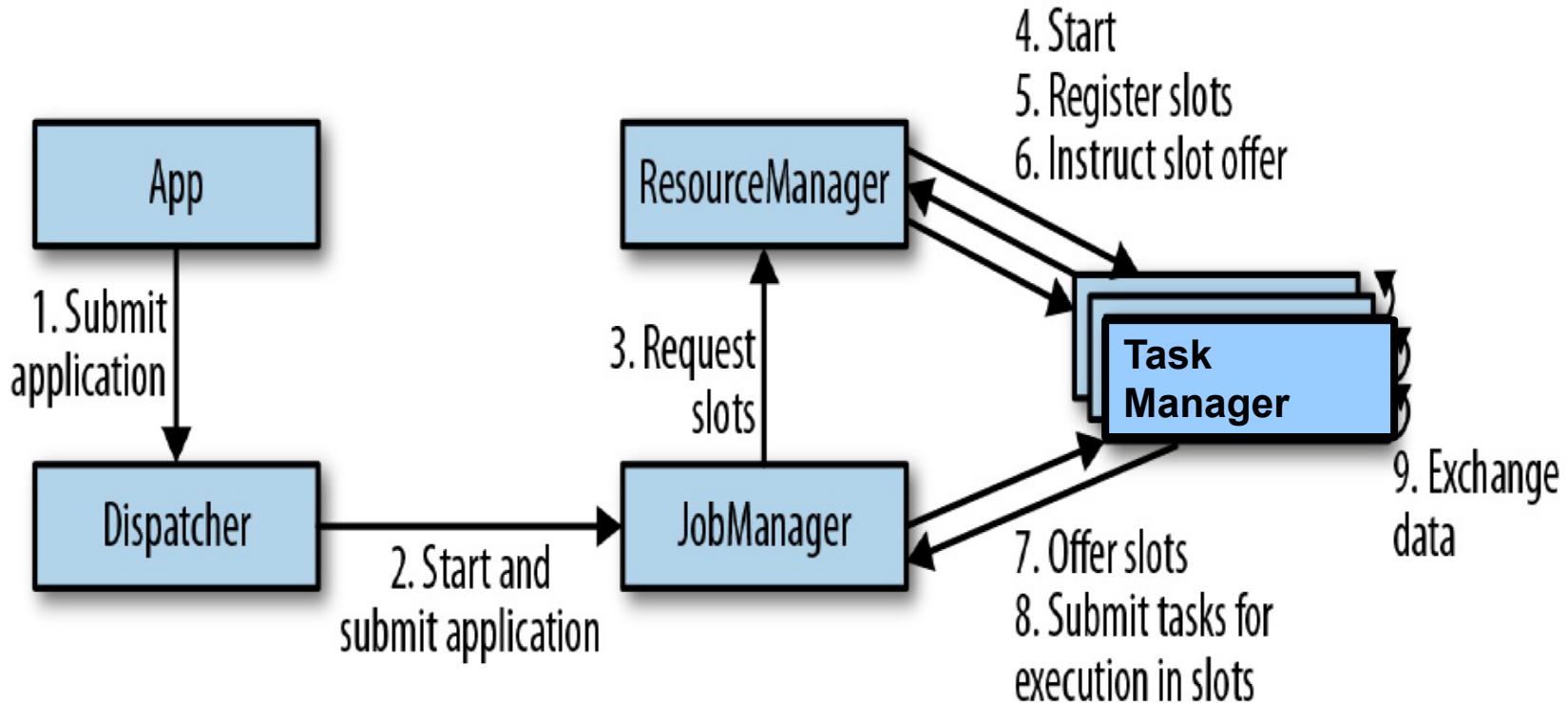
Key-based



Random

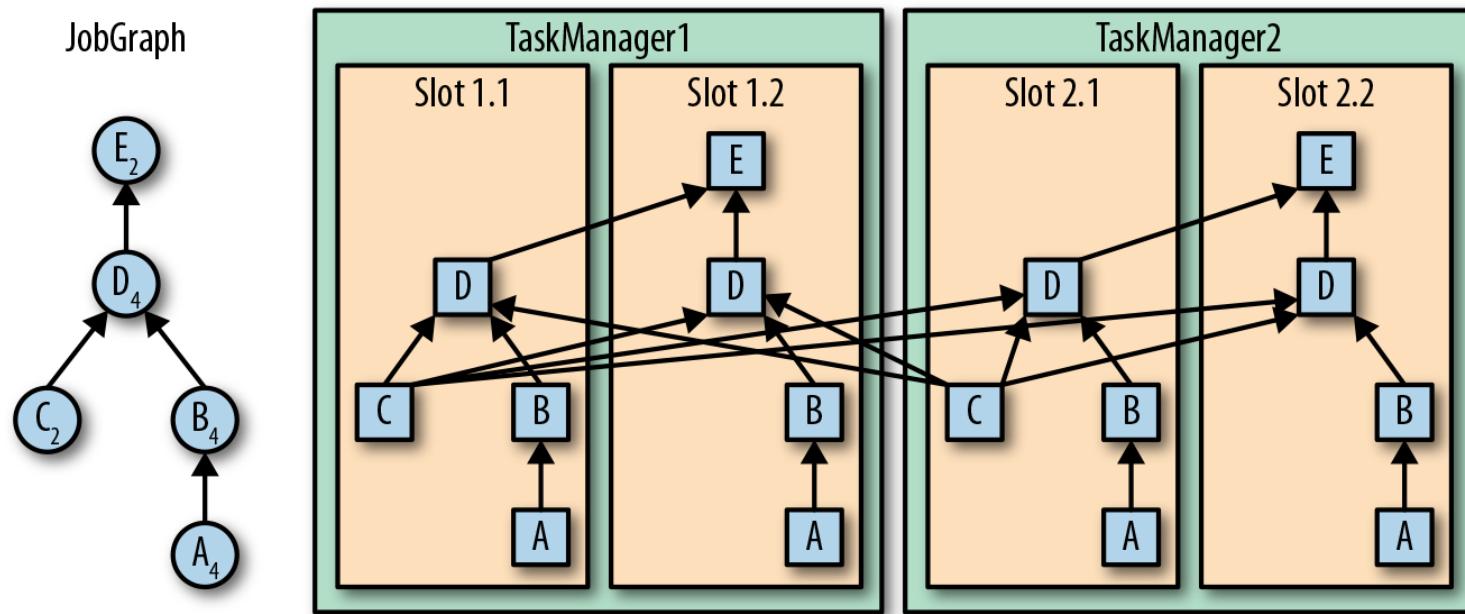
System Architecture

- Flink is a distributed system for stateful parallel data stream processing.



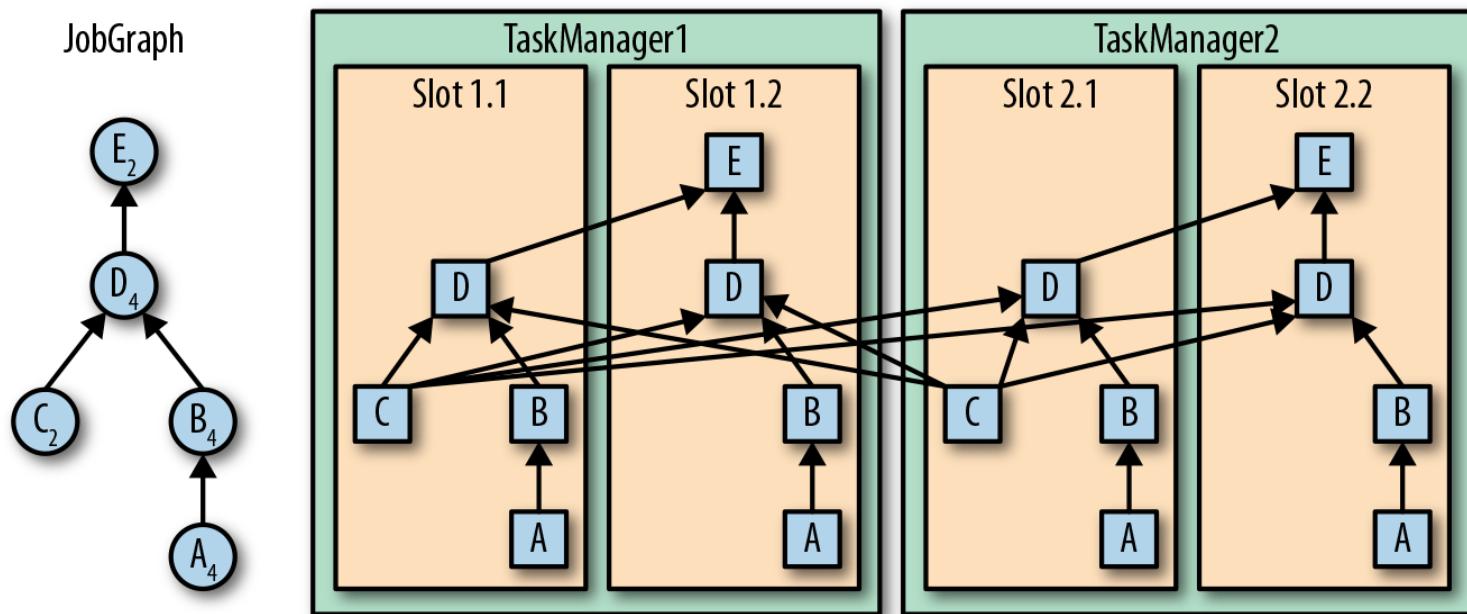
Task Execution

- A TaskManager can execute several tasks at the same time:
 - tasks of the same operator (data parallelism)
 - tasks of different operators (task parallelism)
 - tasks from a different application (job parallelism)



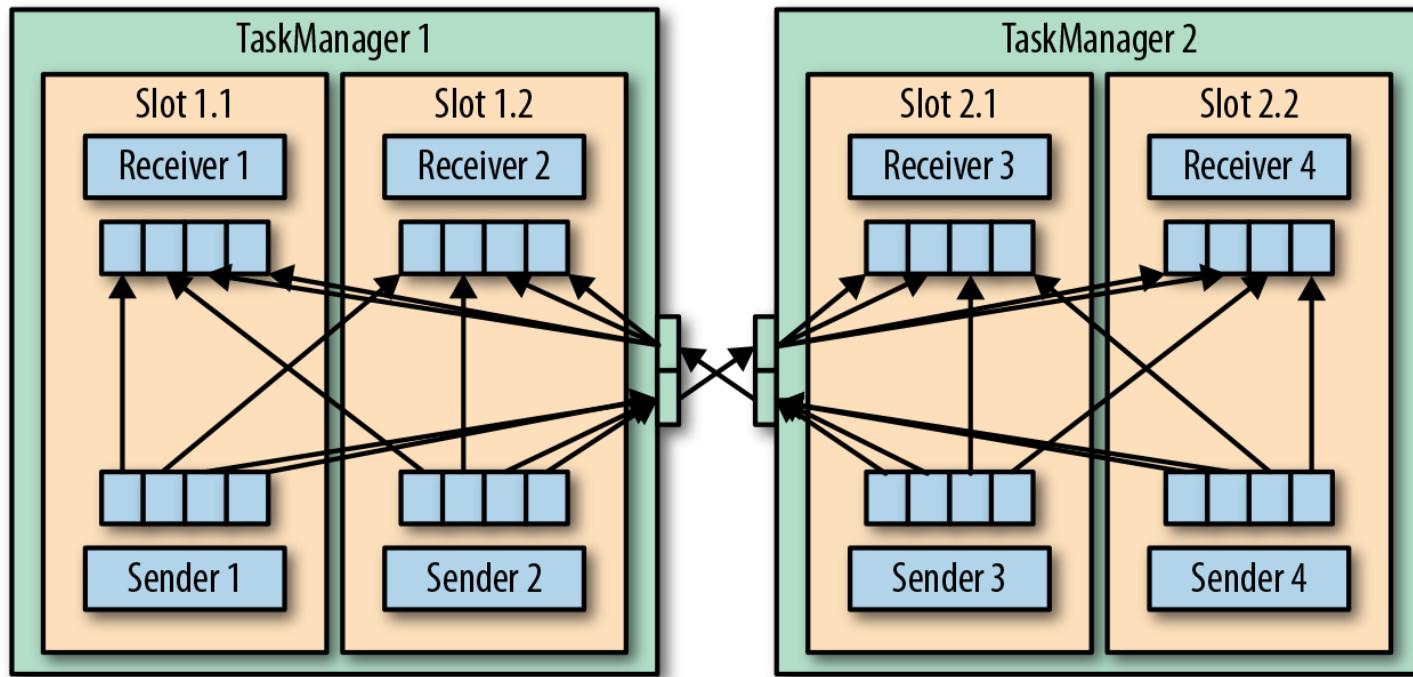
Task Execution

- A TaskManager
 - offers a certain number of processing slots to control the number of tasks it is able to concurrently execute.
- A processing slot
 - can execute one slice of an application—one parallel task of each operator of the application.



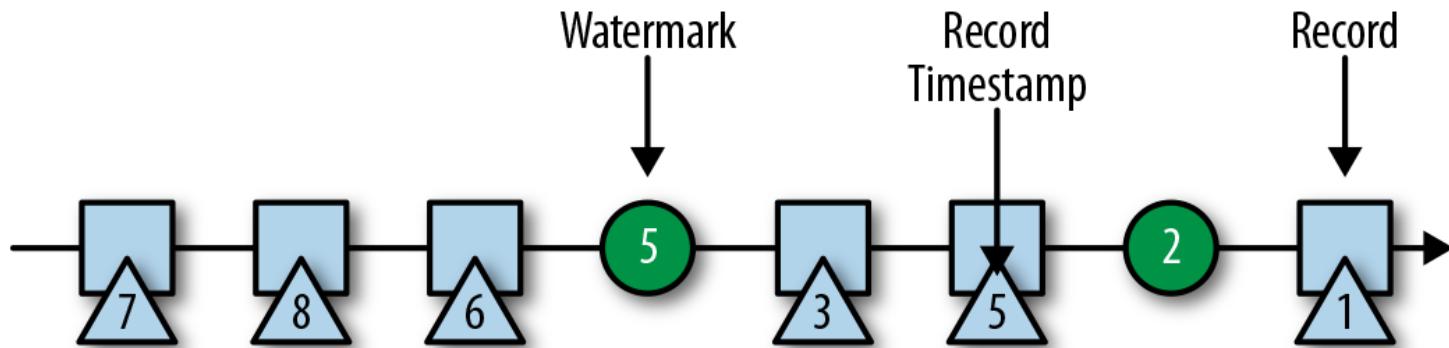
Data Transfer in Flink

- The tasks of a running application are continuously exchanging data.
- The TaskManagers take care of shipping data from sending tasks to receiving tasks.
- The network component of a TaskManager collects records in buffers before they are shipped.



Even-Time Processing

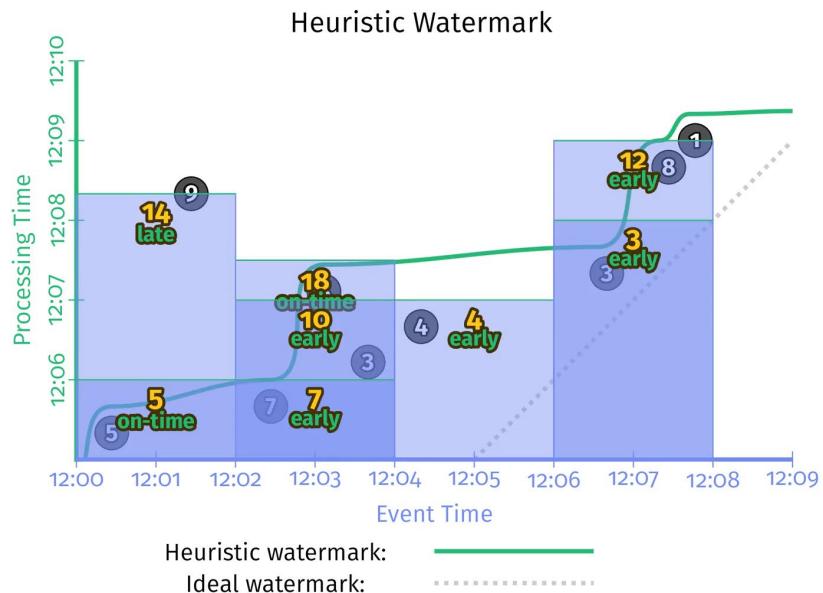
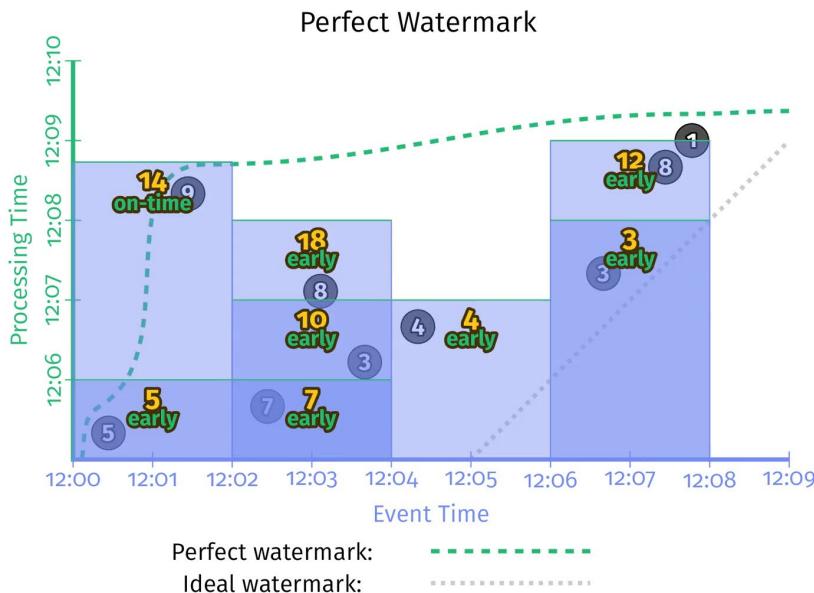
- Timestamps
 - Every record must be accompanied by an event timestamp
- Watermarks
 - a Flink event-time application must also provide watermarks
 - Watermarks are used to derive the current event time at each task in an event-time application
 - In Flink, watermarks are implemented as special records holding a timestamp as a Long value. Watermarks flow in a stream of regular records with annotated timestamps.



Optional

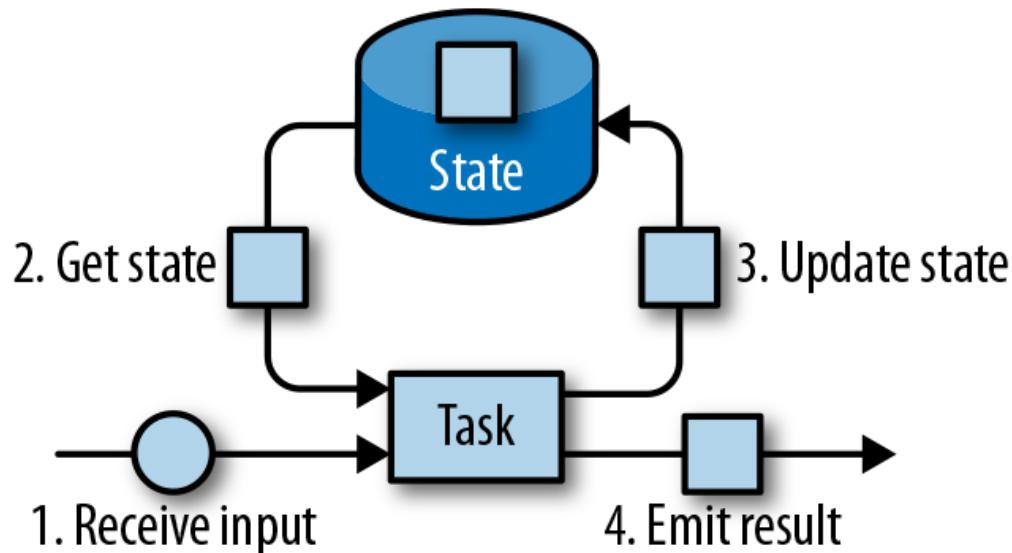
A Simple Counting Example

```
PCollection<KV<Team, Integer>> totals = input
    .apply(Window.into(FixedWindows.of(TWO_MINUTES)))
        .triggering(AfterWatermark()
            .withEarlyFirings(AliignedDelay(ONE_MINUTE))
            .withLateFirings(AfterCount(1))))
    .apply(Sum.integersPerKey());
```



State Management

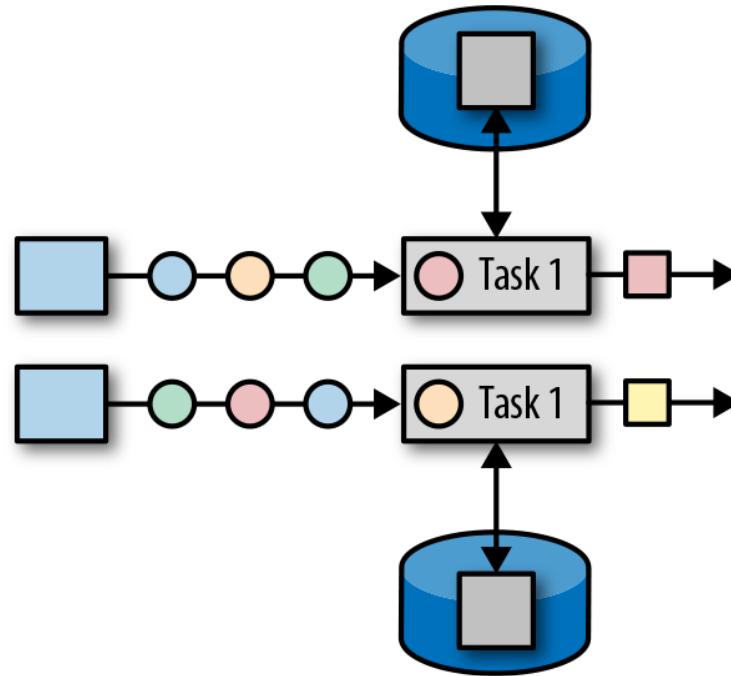
- A stateful stream processing task
 - all data maintained by a task and used to compute the results of a function belong to the state of the task



State Management

- Operator State

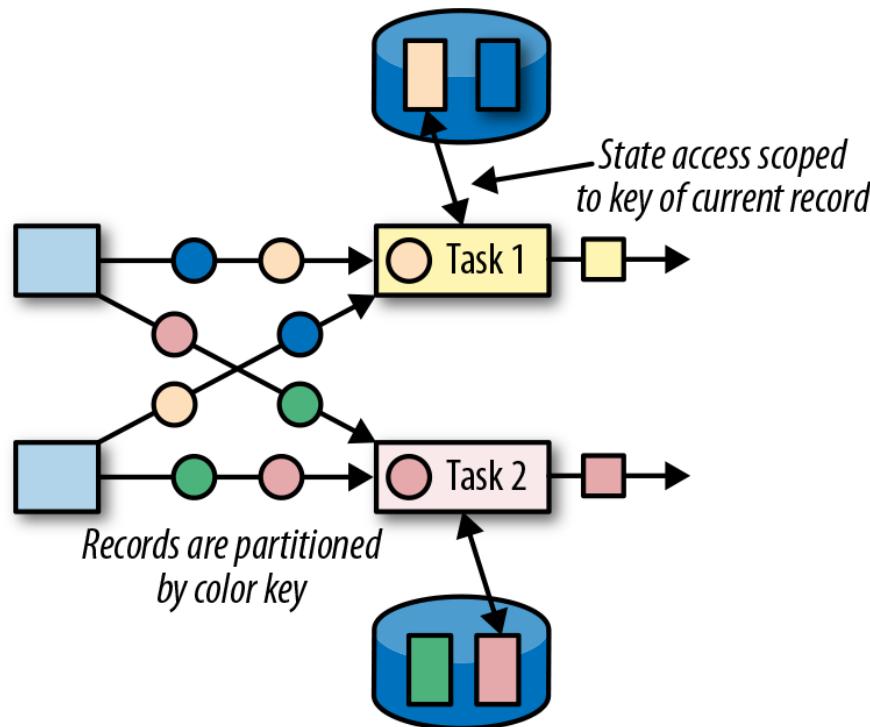
- Scoped to an operator task
- All records processed by the same parallel task have access to the same state
- Operator state cannot be accessed by another task of the same or a different operator



State Management

- Keyed State

- maintains one state instance per key value
- partitions all records with the same key to the operator task that maintains the state for this key



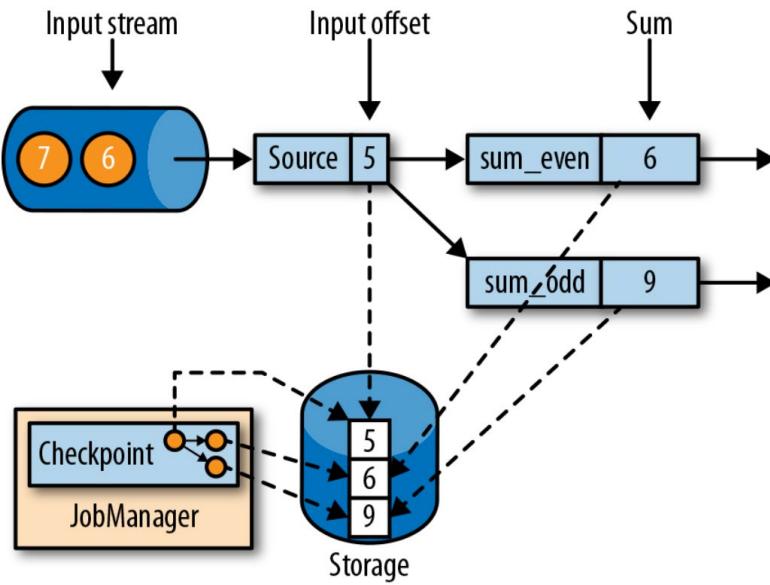
State Management

- State Backend
 - Local State Management
 - a task of a stateful operator typically reads and updates its state for each incoming record.
 - each parallel task locally maintains its state in memory to ensure fast state accesses
 - Checkpointing
 - A TaskManager process may fail at any point, hence its storage must be considered volatile
 - Checkpointing the state of a task to a remote and persistent storage
 - The remote storage for checkpointing could be a distributed filesystem or a database system

Checkpoints

- Consistent checkpoints: similar to Spark micro-batch checkpoints
 1. Pause the ingestion of all input streams
 2. Wait for all in-flight data to be completely processed, meaning all tasks have processed all their input data.
 3. Take a checkpoint by copying the state of each task to a remote, persistent storage. The checkpoint is complete when all tasks have finished their copies.
 4. Resume the ingestion of all streams.

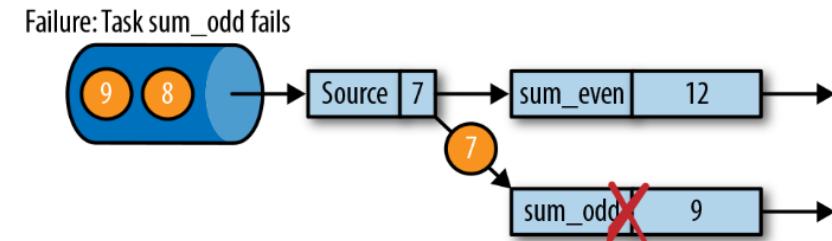
Stop-the-world



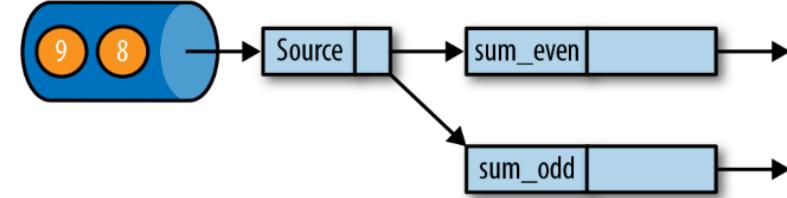
A consistent checkpoint

Failure Recovery from a consistent checkpoint

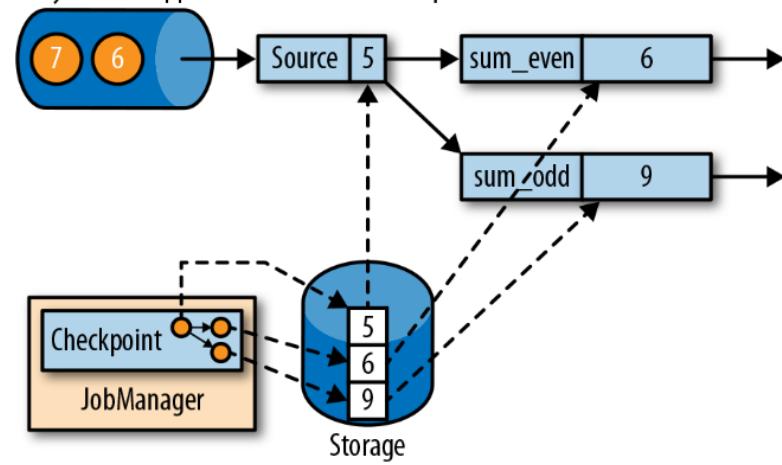
1. Restart the whole application.
2. Reset the states of all stateful tasks to the latest checkpoint.
3. Resume the processing of all tasks.



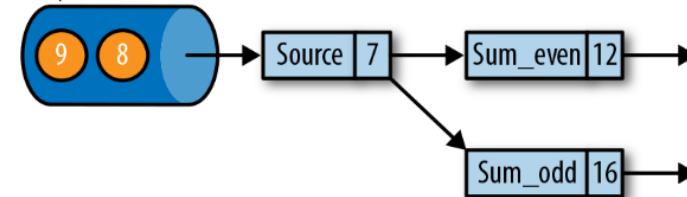
Recovery 1: Restart application



Recovery 2: Reset application state from Checkpoint



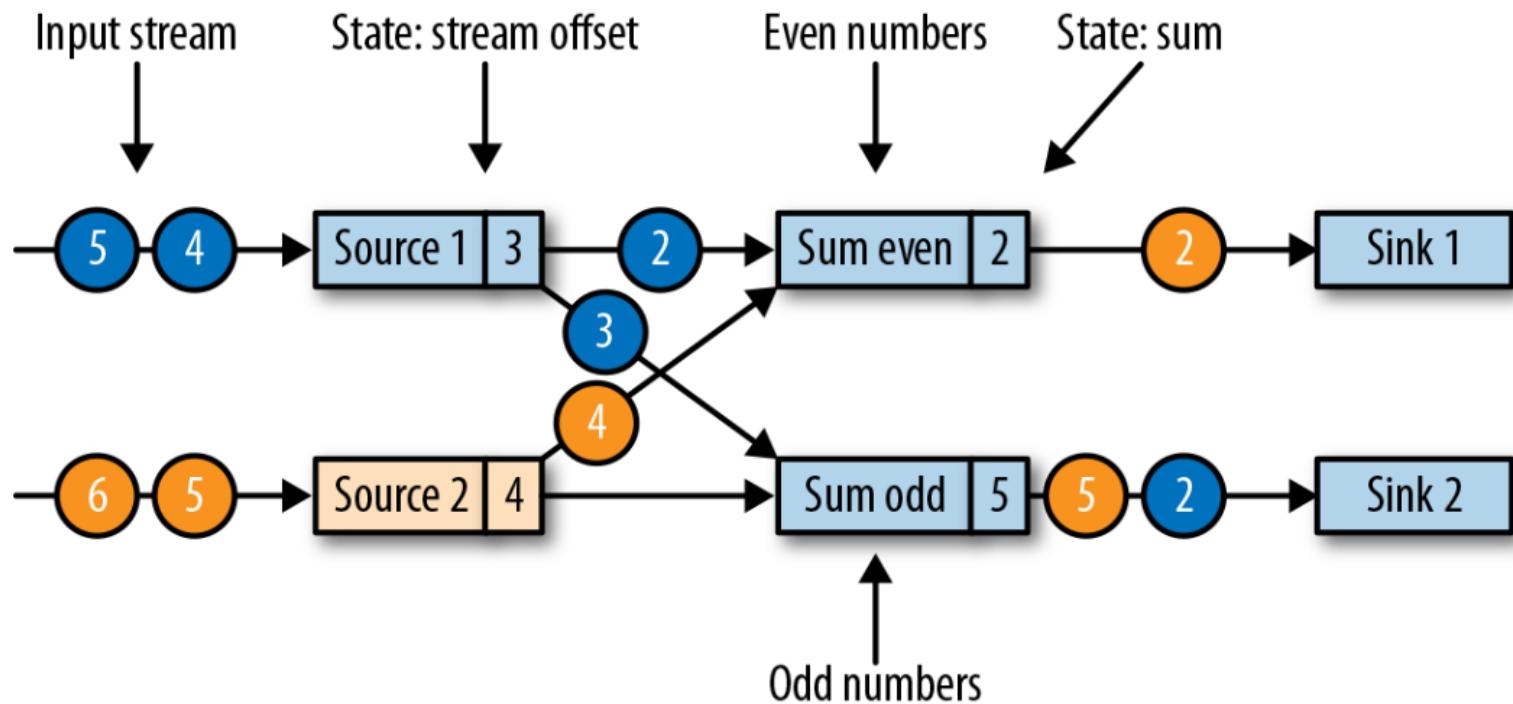
Recovery 3: Continue processing



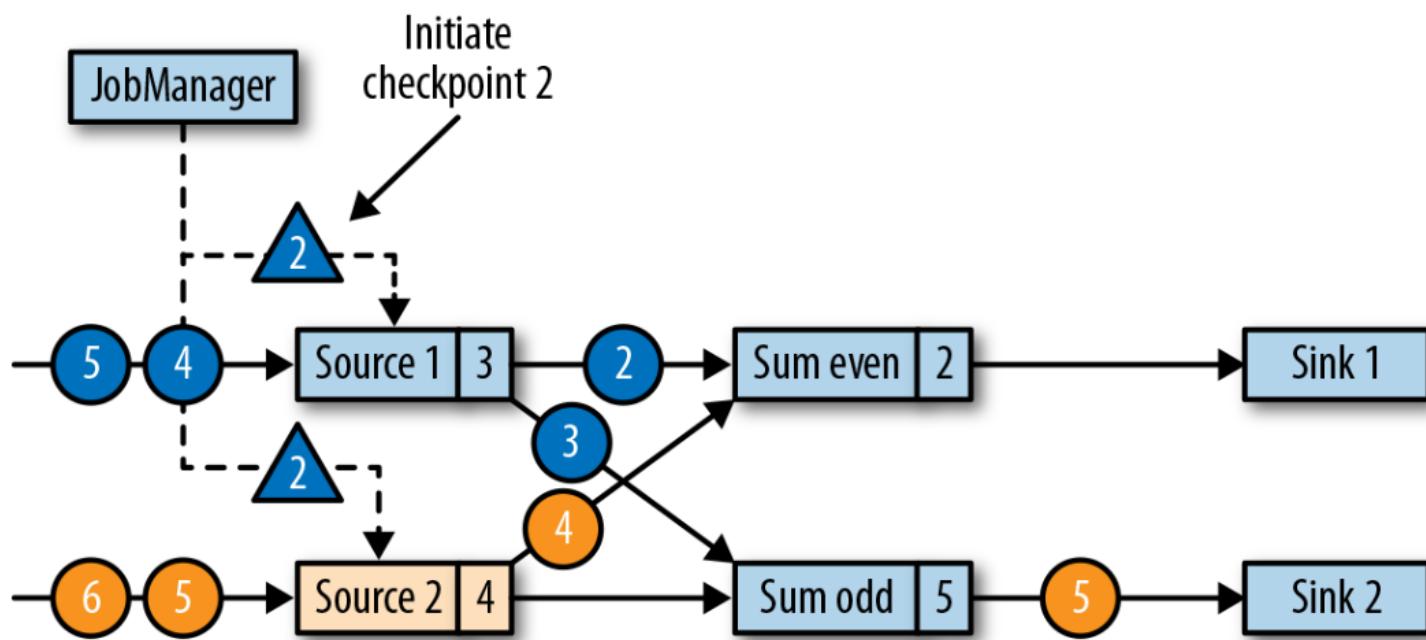
Checkpoints

- Flink's Checkpointing Algorithm
 - based on the Chandy–Lamport algorithm for distributed snapshots
 - does not pause the complete application but decouples checkpointing from processing
 - some tasks continue processing while others persist their state
- More details
 - uses a special type of record called a *checkpoint barrier*
 - checkpoint barriers are injected by source operators into the regular stream of records and cannot overtake or be passed by other records
 - A checkpoint barrier carries a checkpoint ID to identify the checkpoint it belongs to and logically splits a stream into two parts
 - All state modifications due to records that precede a barrier are included in the barrier's checkpoint and all modifications due to records that follow the barrier are included in a later checkpoint.

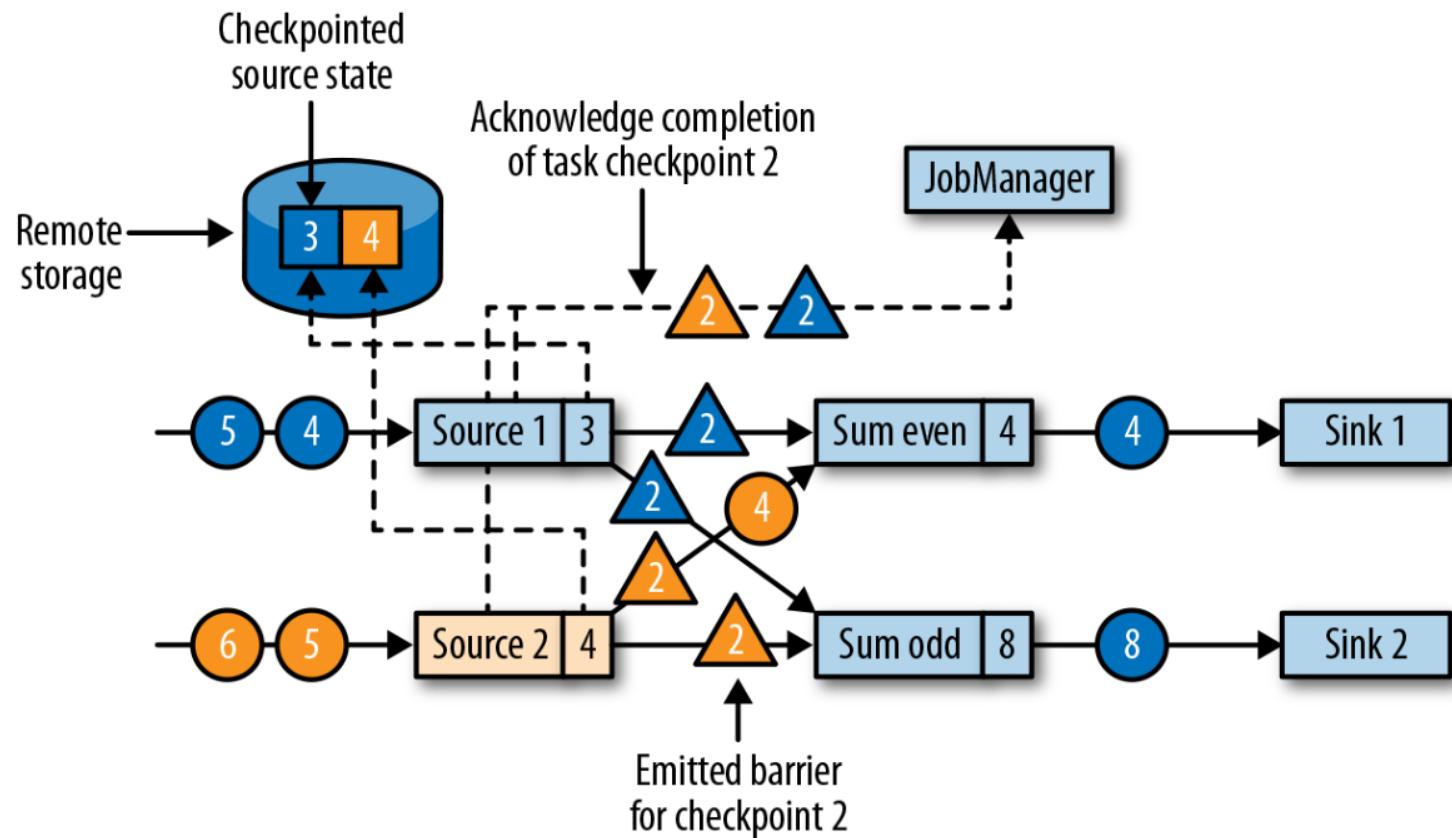
- Streaming application with two stateful sources, two stateful tasks, and two stateless sinks



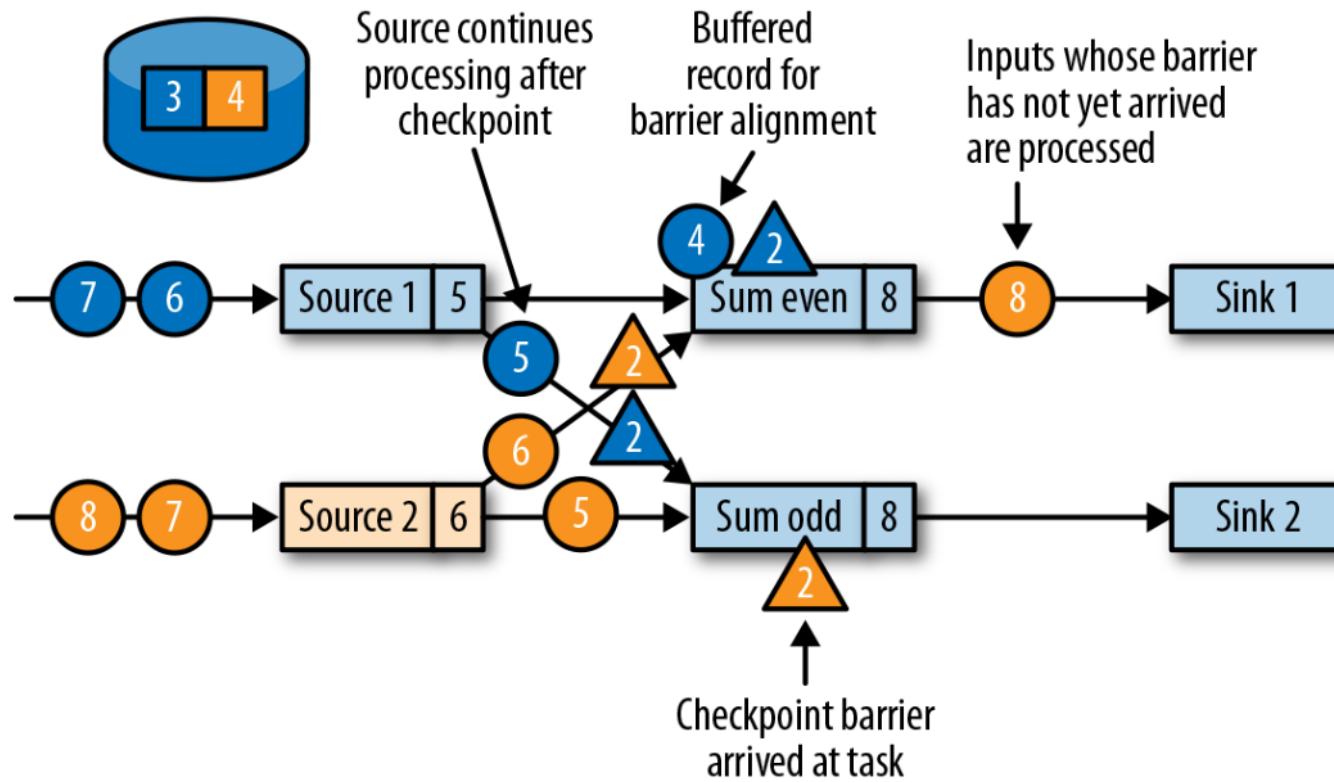
- JobManager initiates a checkpoint by sending a message to all sources



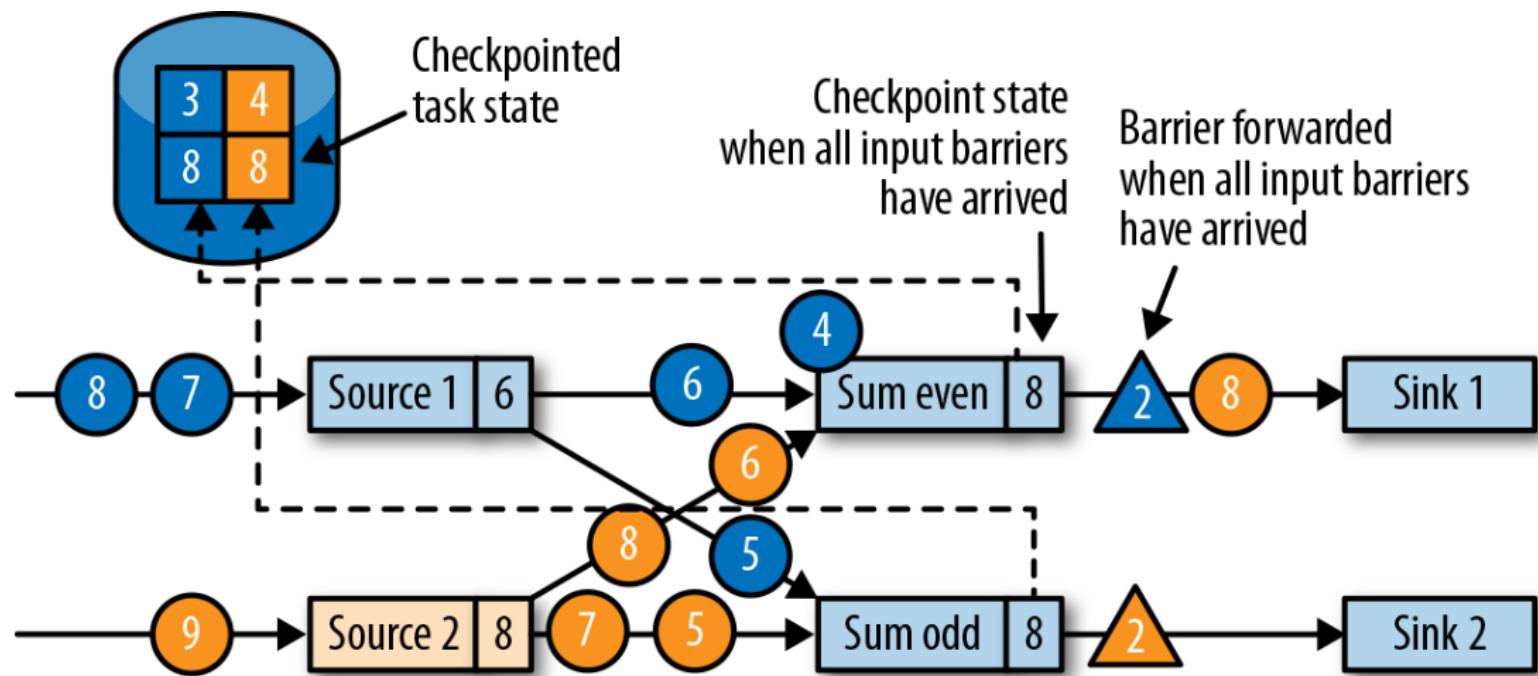
- Sources checkpoint their state and emit a checkpoint barrier



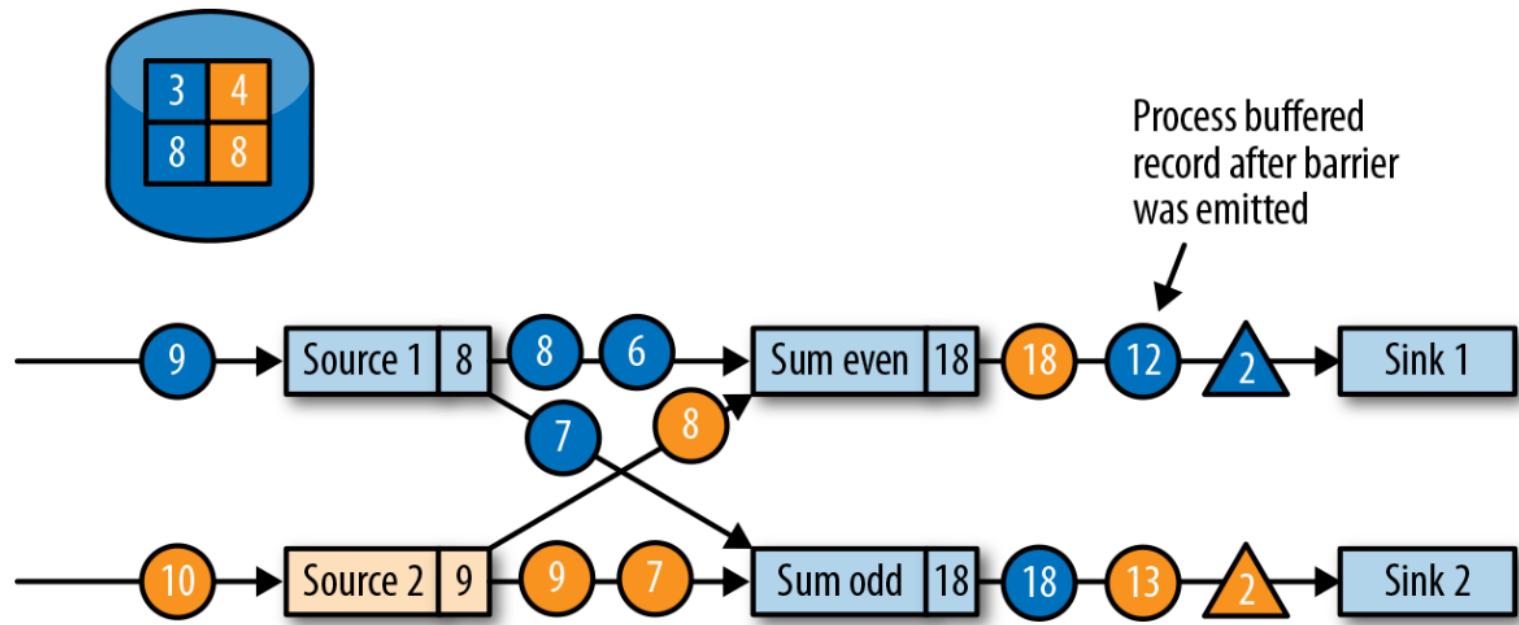
- Tasks wait to receive a barrier on each input partition
- Records from input streams for which a barrier already arrived are buffered
- All other records are regularly processed



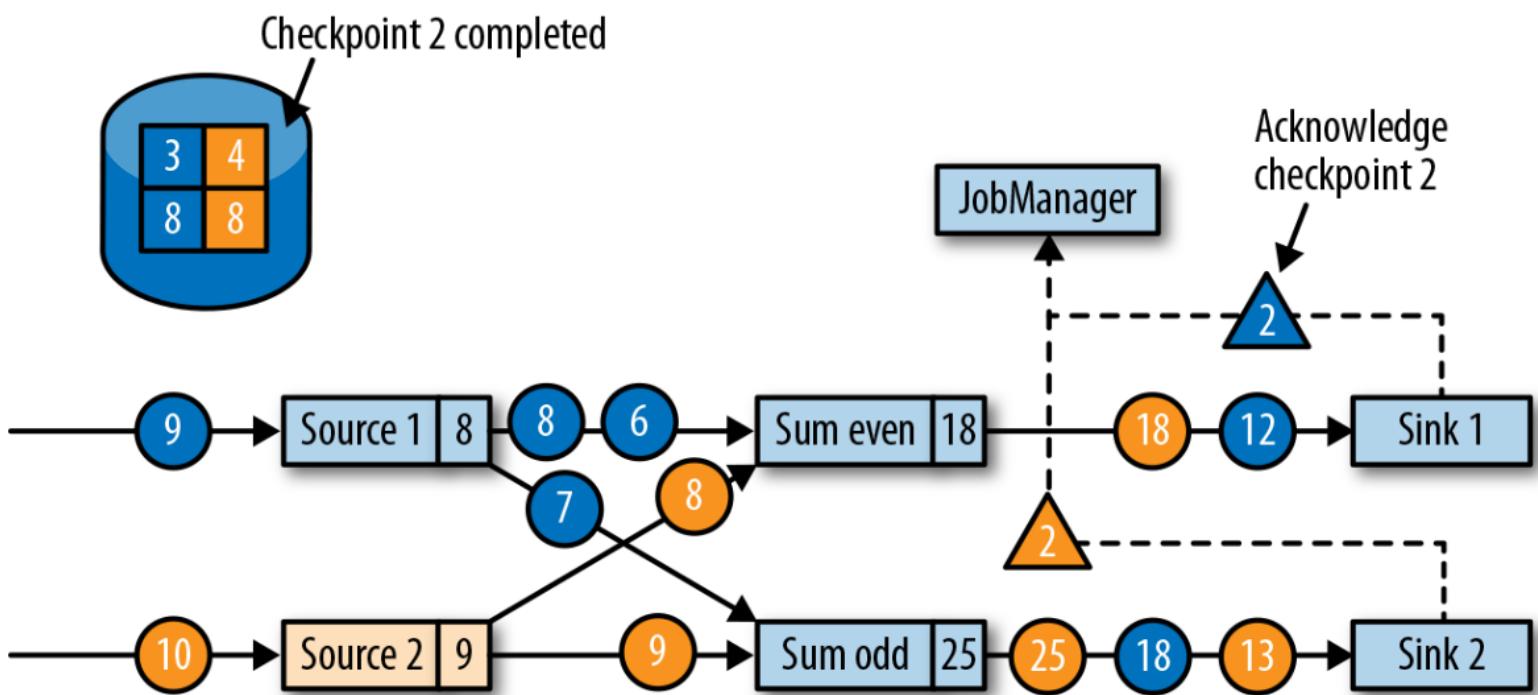
- Tasks checkpoint their state once all barriers have been received, then they forward the checkpoint barrier



- Tasks continue regular processing after the checkpoint barrier is forwarded



- Sinks acknowledge the reception of a checkpoint barrier to the JobManager
- a checkpoint is complete when all tasks have acknowledged the successful checkpointing of their state



Conclusion: a comparison between Spark vs. Flink

- Spark

- Microbatch streaming processing (latency of a few seconds)
- Checkpoints are done for each microbatch in a synchronous manner (“stop the world”)
- Watermark: a configuration to determine when to drop the late events

- Flink

- Real-time streaming processing (latency of milliseconds)
- Checkpoints are done distributedly in an asynchronous manner (more efficient → lower latency)
- Watermark: a special record to determine when to trigger the even-time related results
 - Flink uses late handling functions (related to watermark) to determine when to drop the late events

Acknowledgements

- CS4225 slides by He Bingsheng and Bryan Hooi
- Jules S. Damji, Brooke Wenig, Tathagata Das & Denny Lee, “Learning Spark: Lightning-Fast Data Analytics”
- Bill Chambers, Matei Zaharia, “Spark: The Definitive Guide”
- Fabian Hueske and Vasiliki Kalavri, “Stream Processing with Apache Flink”, 2019
- Tyler Akidau, Slava Chernyak and Reuven Lax, “Streaming Systems”, 2018