Big Data Systems

Djellel Difallah

Spring 2023

Lecture 11 – Stream Processing

(Spark Streaming)

Outline

- Introduction
- Motivation
- Use Case (Telecom)
- Data Streaming Systems
- Spark Streaming

Big Velocity

- Data Arrival Speed – *How fast new data arrive?*
  - 10K to 1M of tuples per second (and beyond?)

- Data Arrival Pattern – *How does the data arrive to the infrastructure?*
  - Tuples arrive in streams (continuous, consistent or following some distribution)

- Data Processing – *What type of queries are executed?*
  - Identify complex patterns in the stream
  - (Near-)Real-time data processing
  - **Output**: alerts, anomalies, trends etc.

→ **Data Stream Management System (DSMS)**

Application Domains

Real-time and *Near Real-time* data analytics

- Web
  - Click streams (ad placement)
- Monitoring
  - Anomaly detection, intrusion (networks), fraud (credit card usage)
- Financial services
  - High Frequency Trading (HFT), Market feed processing (News, Social Media)
- Sensor-based environment monitoring
  - Weather conditions, air quality, car traffic
  - Civil engineering, military applications, etc.
- Medical applications
  - Patient monitoring, equipment tracking

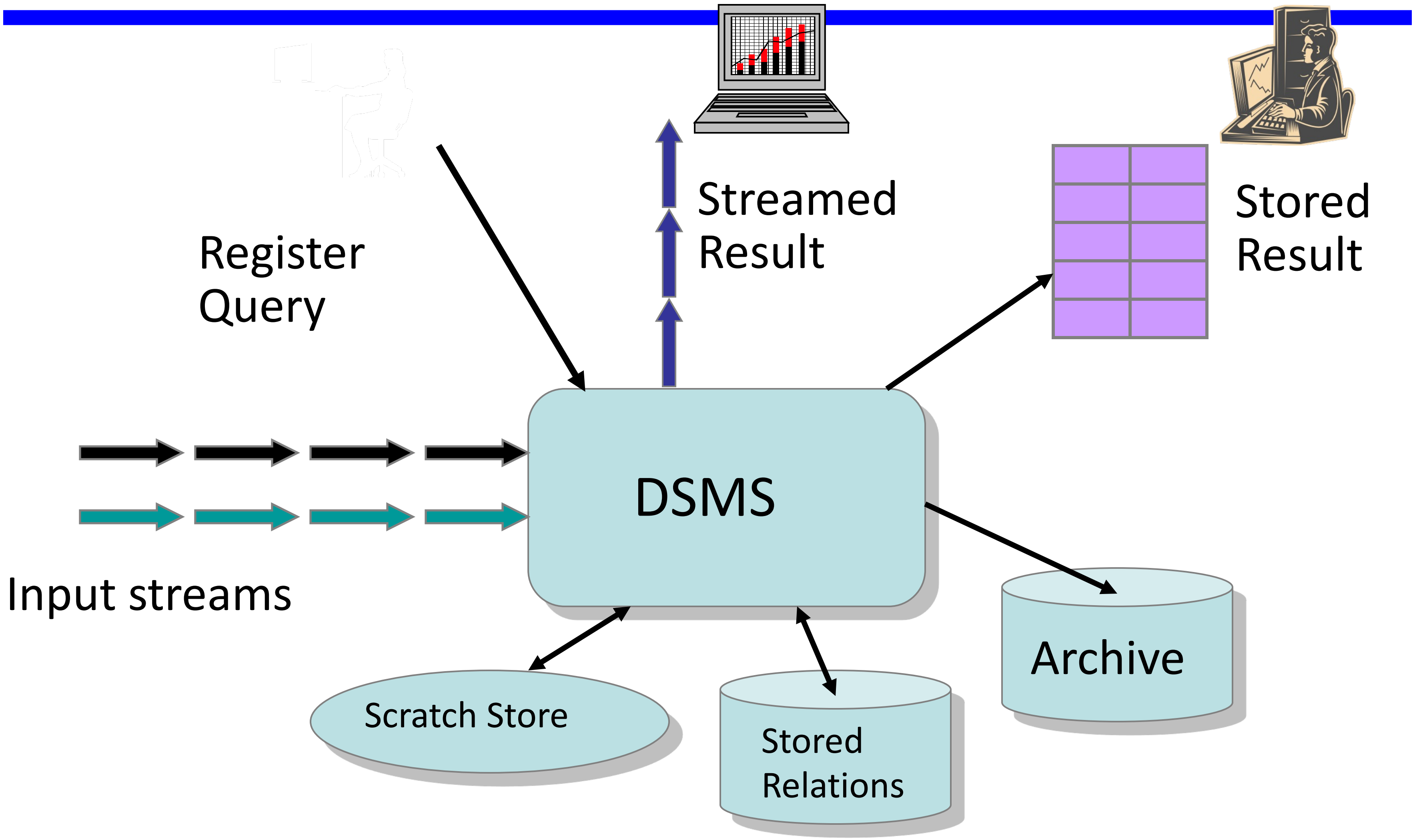
DBMS versus DSMS

DBMS	DSMS
• Persistent relations	• Transient streams (and persistent relations)
• One-time queries	• Continuous queries
• Random access	• Sequential access
• Access plan determined by query processor and physical DB design	• Unpredictable data characteristics and arrival patterns

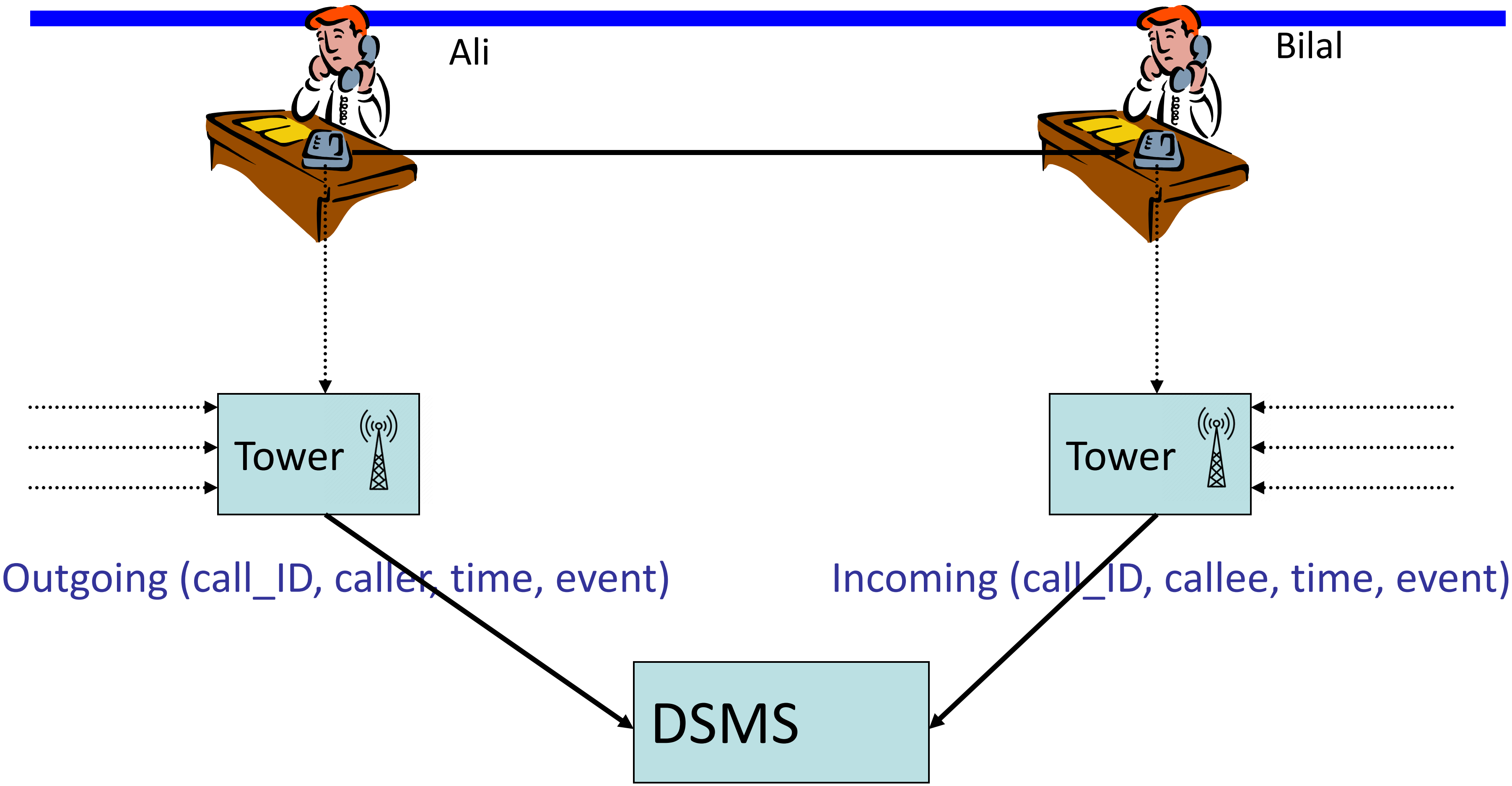
Continuous Queries

- DBMS:
  - One time queries: Run once to completion over the current data set
- DSMS
  - Continuous queries: Issued once and then continuously evaluated over the data.
  - Example:
    - Notify me when the temperature drops below X
    - Tell me when prices of stock Y > 300

The (Simplified) Big Picture



Making Things Concrete



event = **START** or **END**

Self-Join

- Find all outgoing calls longer than 2 hours

```
SELECT  O1.call_ID, O1.caller
FROM    Outgoing O1, Outgoing O2
WHERE   O2.time – O1.time > 2
AND    O1.call_ID = O2.call_ID
AND    O1.event = "start"
AND    O2.event = "end"
```

- We want to have this information continuously
- More importantly, output after 2hours of call time, i.e., before the call is over

→ Database overload (continuous join)

Group-by aggregation

- Total connection time for each caller

```
SELECT   O1.caller, sum(O2.time – O1.time)
FROM     Outgoing O1, Outgoing O2
WHERE    O1.call_ID = O2.call_ID
AND      O1.event = "start"
AND      O2.event = "end"
GROUP BY O1.caller
```

- Provide current value continuously
- Trigger the computation automatically upon each new call

→ Database overload (keep in memory stats for all users)

Architectural Differences

DSMS

- Resource (memory, per-tuple computation) limited
- Reasonably complex, near real time, query processing
- Query Operator: One pass
- Query Plan: Adaptive

DBMS

- Resource (memory, disk, per-tuple computation) rich
- Extremely sophisticated query processing, analysis
- Query Operator: Arbitrary
- Query Plan: Fixed

DSMS Challenges

- Must cope with:
  - Stream rates that may be **high**, variable, **bursty**
  - Stream data type that may be variable
  - Query load may be **high**, variable

→**Overload – need to use resources very carefully.**

→Changing conditions – adaptive strategy.

Impact of Limited Memory

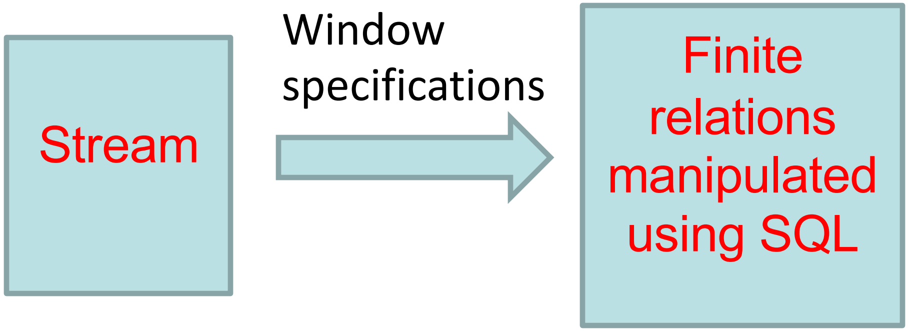
- Continuous streams grow unboundedly
- Queries may require unbounded memory (self join!)
- Solution / Tradeoff:
  - Approximate query evaluation

Approximate Query Evaluation

- Why?
  - Handling load – streams arrive too fast
  - Avoid unbounded storage and computation
  - Ad hoc queries need approximate
  - Follows the principle of “eventual consistency”
- How?
  - Sliding windows, synopsis, samples, load-shed
- Major Issues?
  - Set-valued queries (how to evaluate precision/recall?)
  - Composition of approximate operators
  - How is it understood/controlled by user?
  - Integrate into query language
  - Query planning and interaction with resource allocation

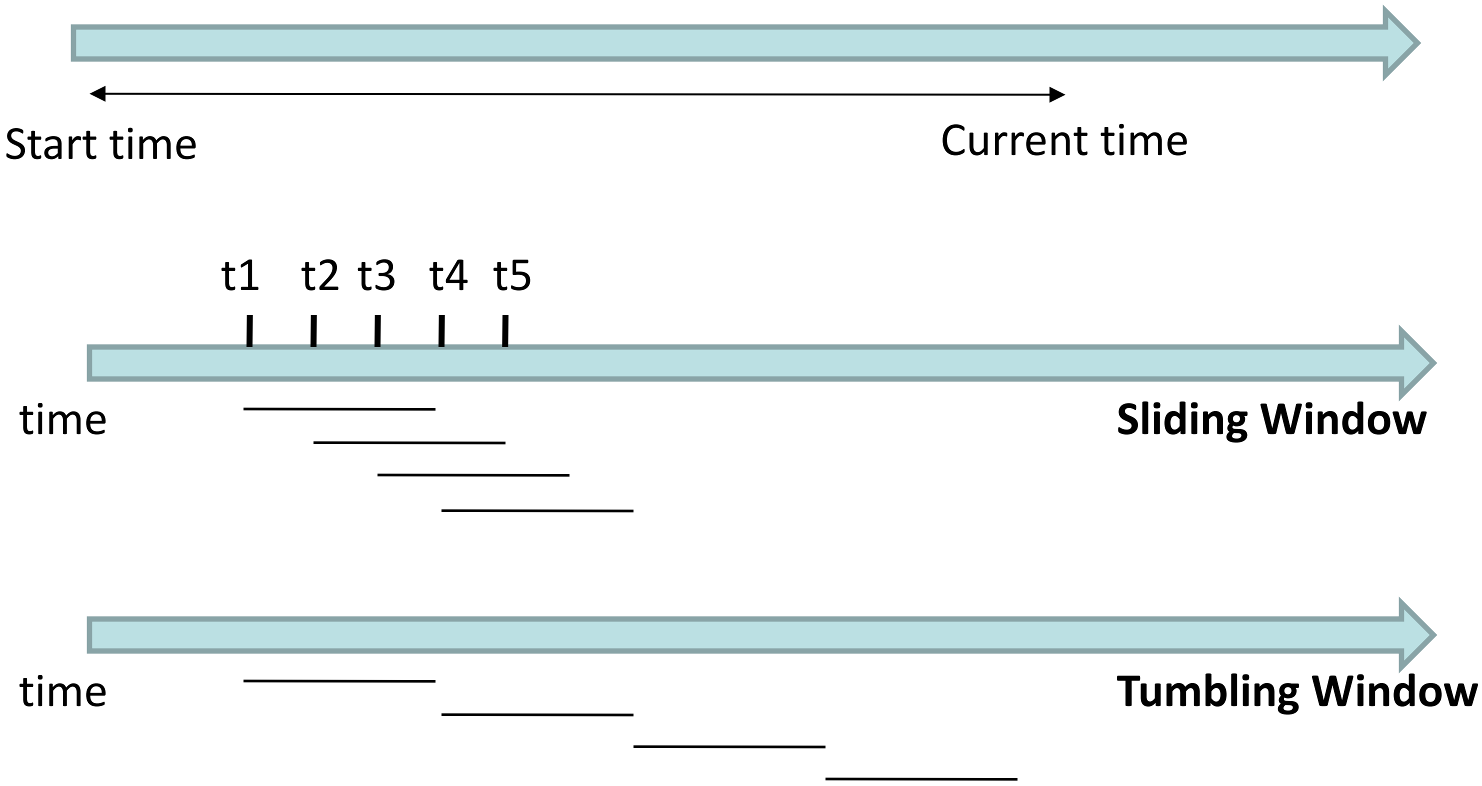
Window

- Mechanism for extracting a finite relation from an infinite stream
- Various window proposals for restricting operator scope.
  - Windows based on ordering attribute (e.g. time)
  - Windows based on tuple counts
  - Windows based on explicit markers (e.g. punctuations)
  - Variants (e.g., partitioning tuples in a window)



Windows

- Terminology



Query Operators

- Selections and Projections on streams - straightforward
  - Local per-tuple processing
  - Projection may need to include ordering attribute.
- Group by – a bit more complex, but generally easy to implement
  - Window-based group by
  - Sampling-based group-by
- Joins – Problematic
  - May need to join tuples that are arbitrarily far apart, or on different streams.
  - The majority of the solutions use Window-based joins

Basic Functionalities

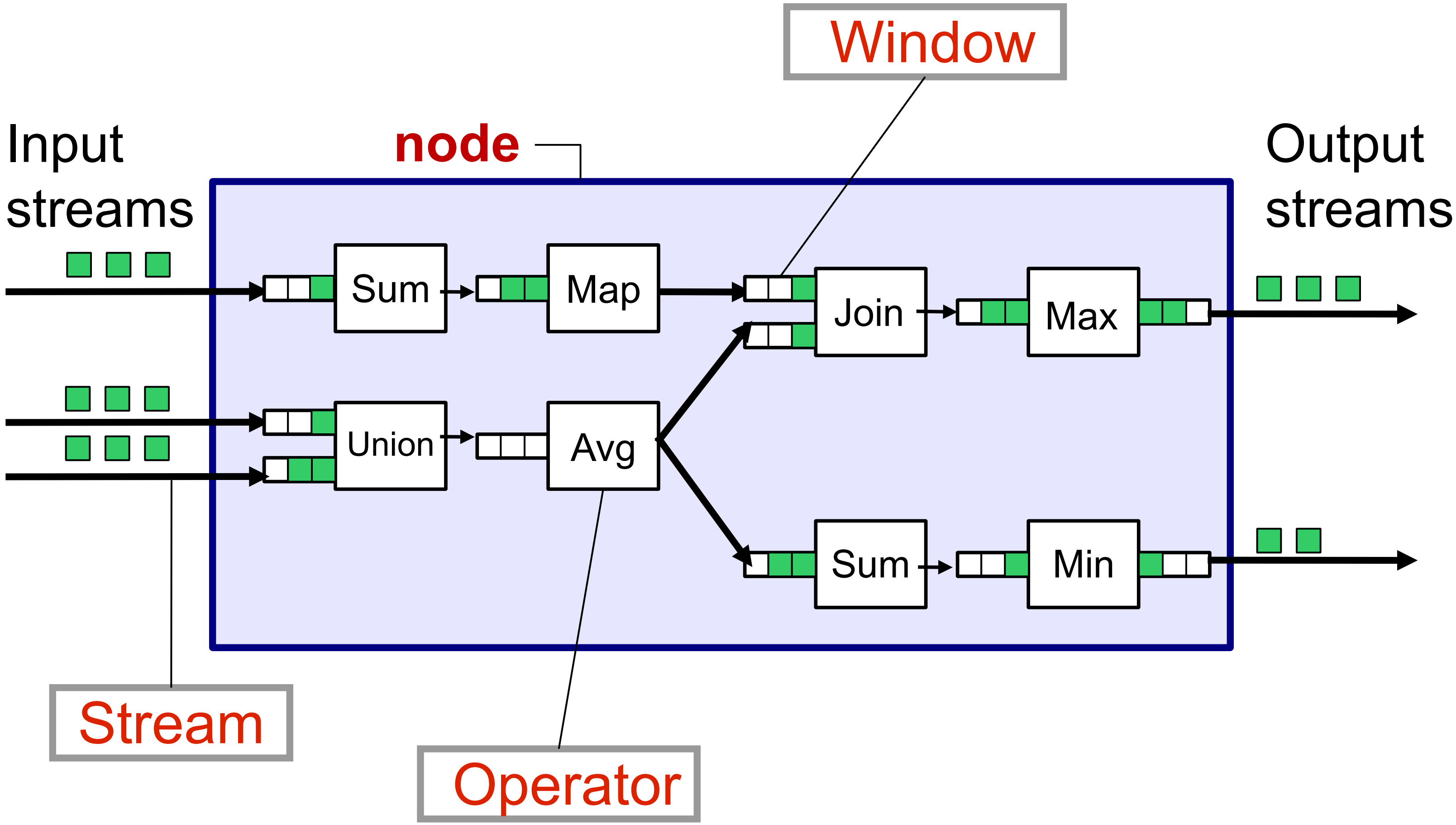
DATA STREAMING SYSTEM

Simple (Wide) Data Model

Tuple: $(\underbrace{\texttt{\textbf{timestamp}},}_{\text{header}}\ \underbrace{\texttt{v}_1,\ldots,\texttt{v}_n}_{\text{data}})$

- Stream: append-only sequence of tuples
- All tuples on a stream have the same schema

Query Model Example



Operators

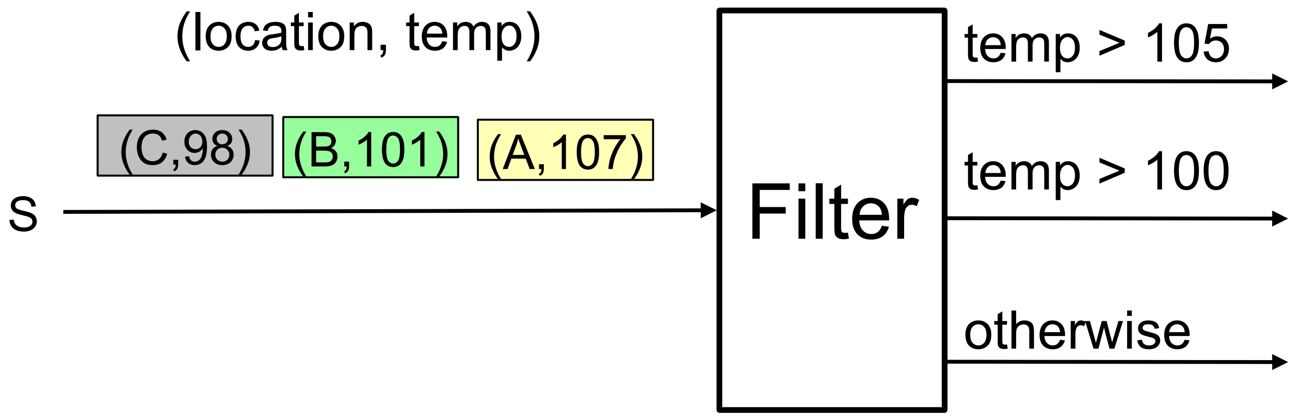
- Order-agnostic
  - Filter
  - Map
  - Union
- Order-sensitive
  - Aggregate (bursts)
  - Join (unaligned streams)
  - Sort (latest value), Sample (skew)
- **Why do we need new operators?**
  - Ops cannot block & cannot accumulate state that grows with input

Filter Example

Input tuples

Output tuples

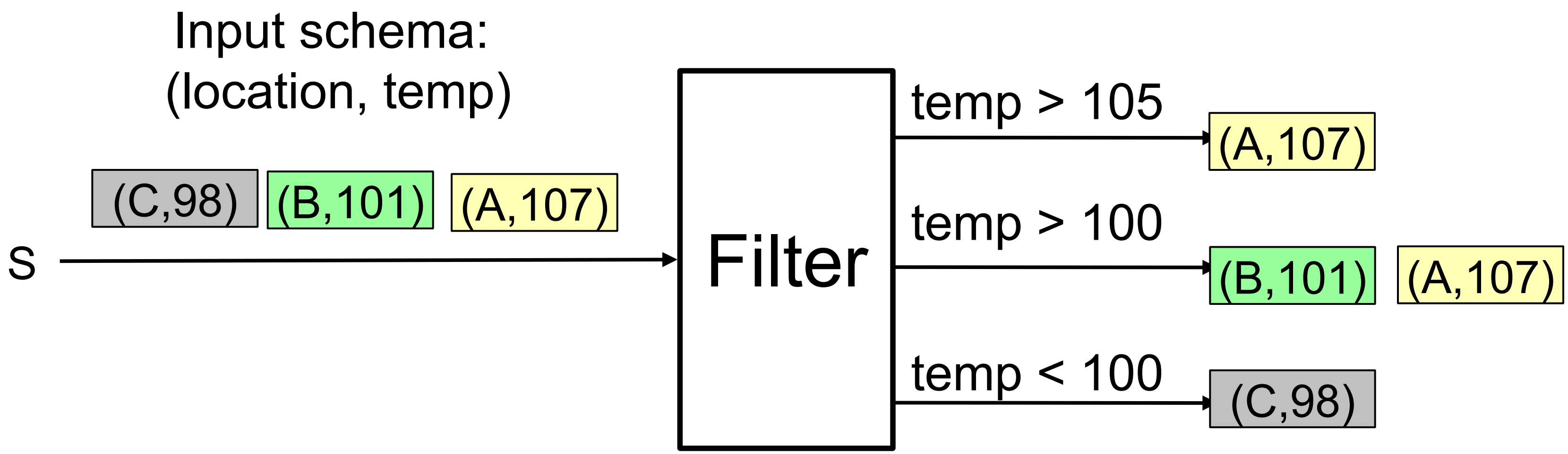
Input schema:
(location, temp)



Filter Example

Input tuples

Output tuples



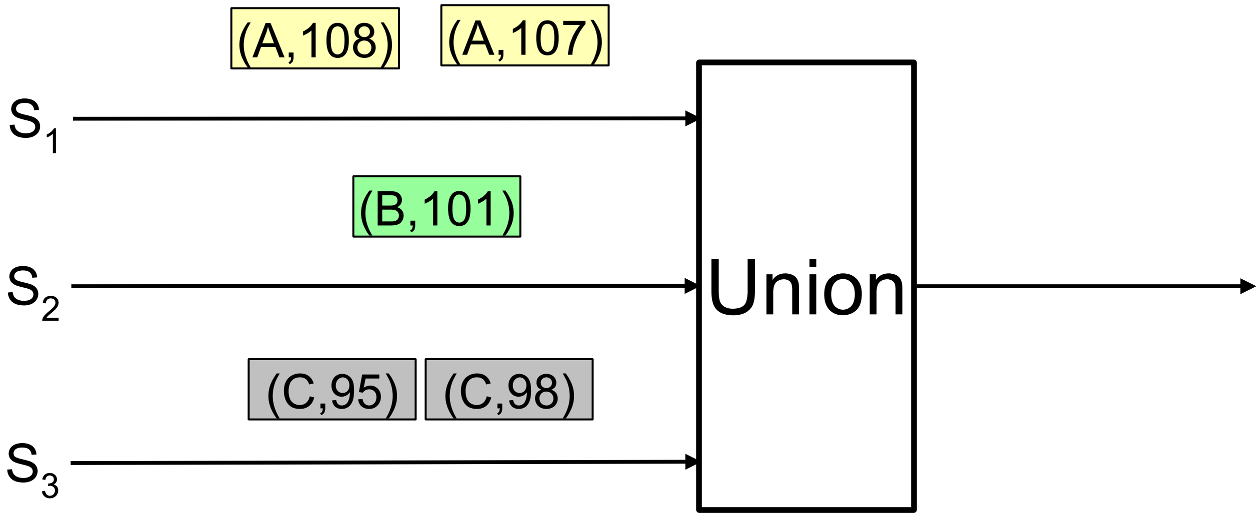
Map Example

new.location = old.location
new.temp_celcius = 5/9*(old.temp - 32)

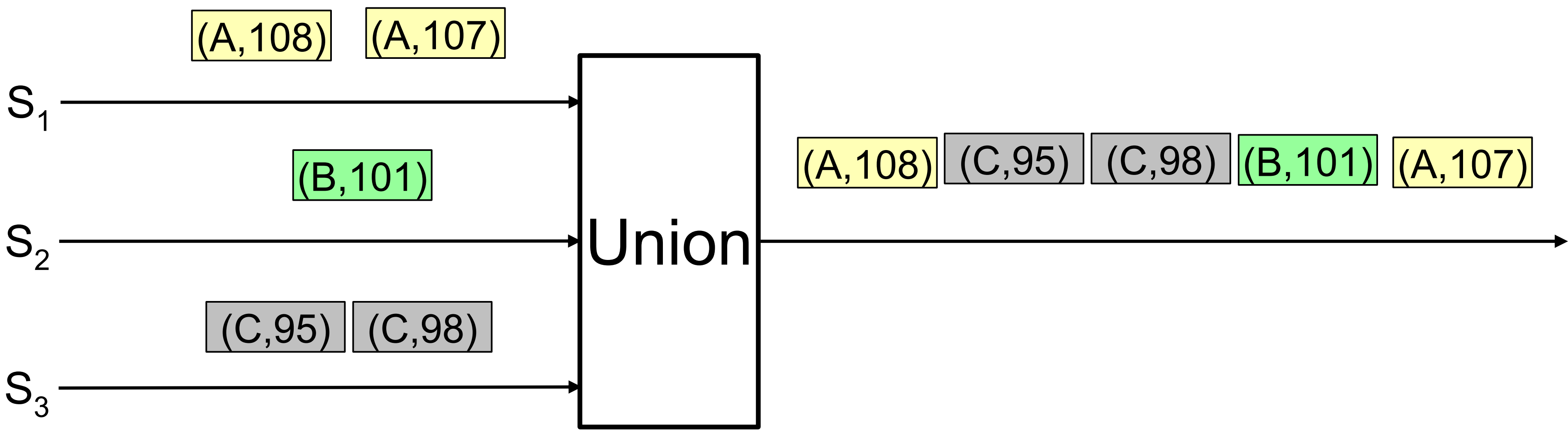
Map Example

new.temp_celcius = 5/9*(old.temp - 32)

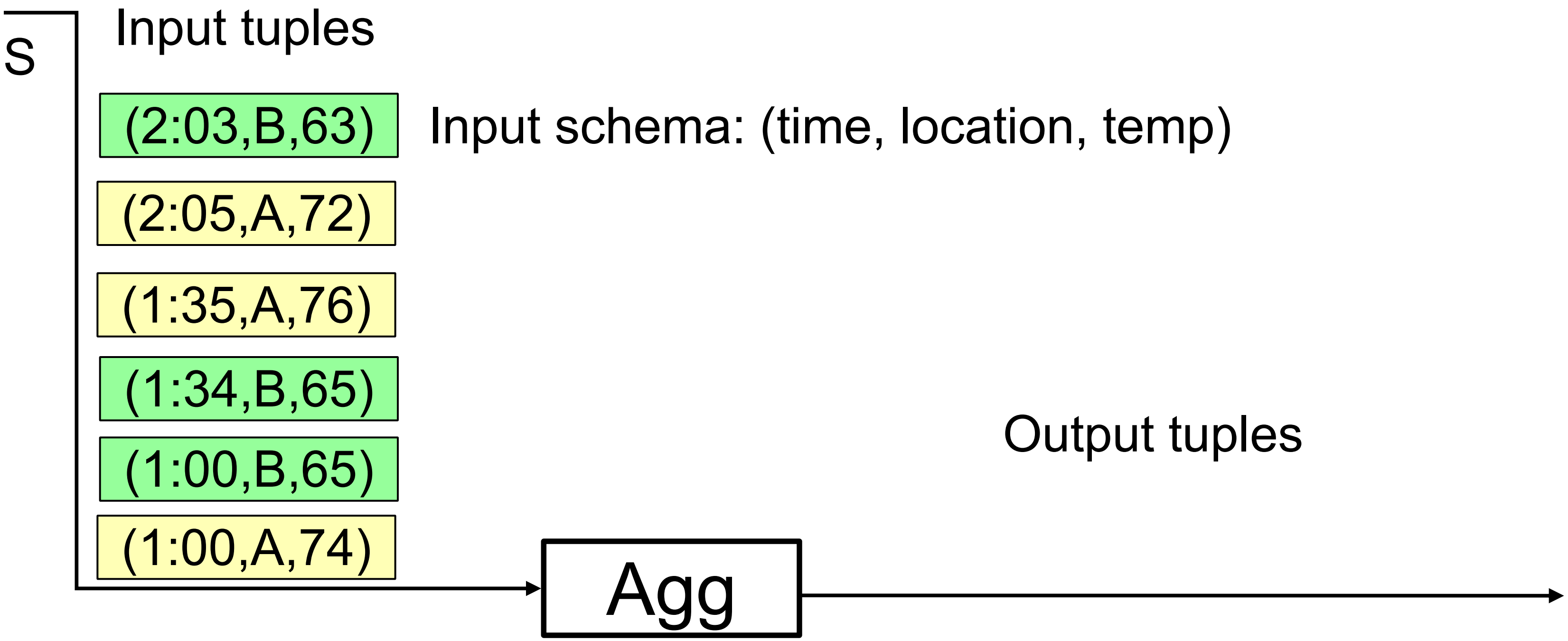
Union Example



Union Example

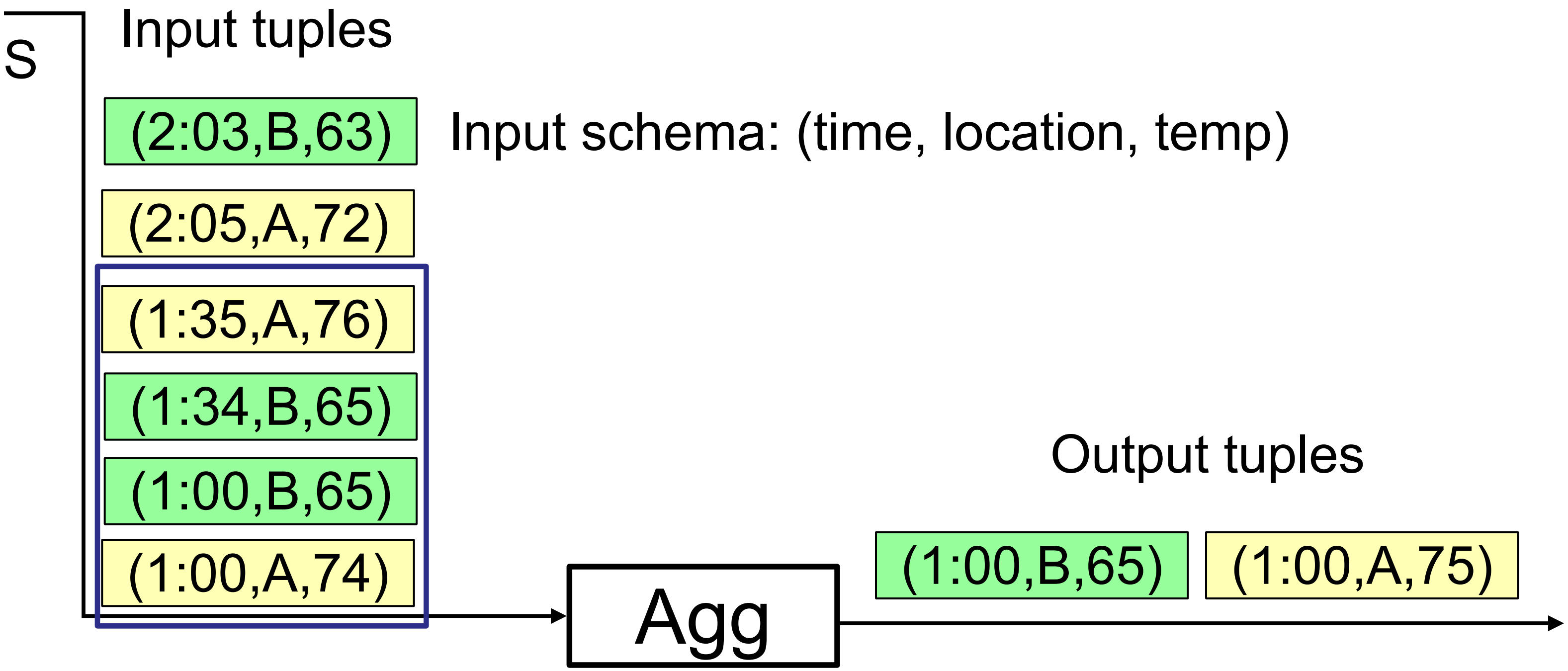


Aggregate Example



Operator parameters:
- group by location
- average temp,
- order on time, window size 1 h, advance 1h

Aggregate Example

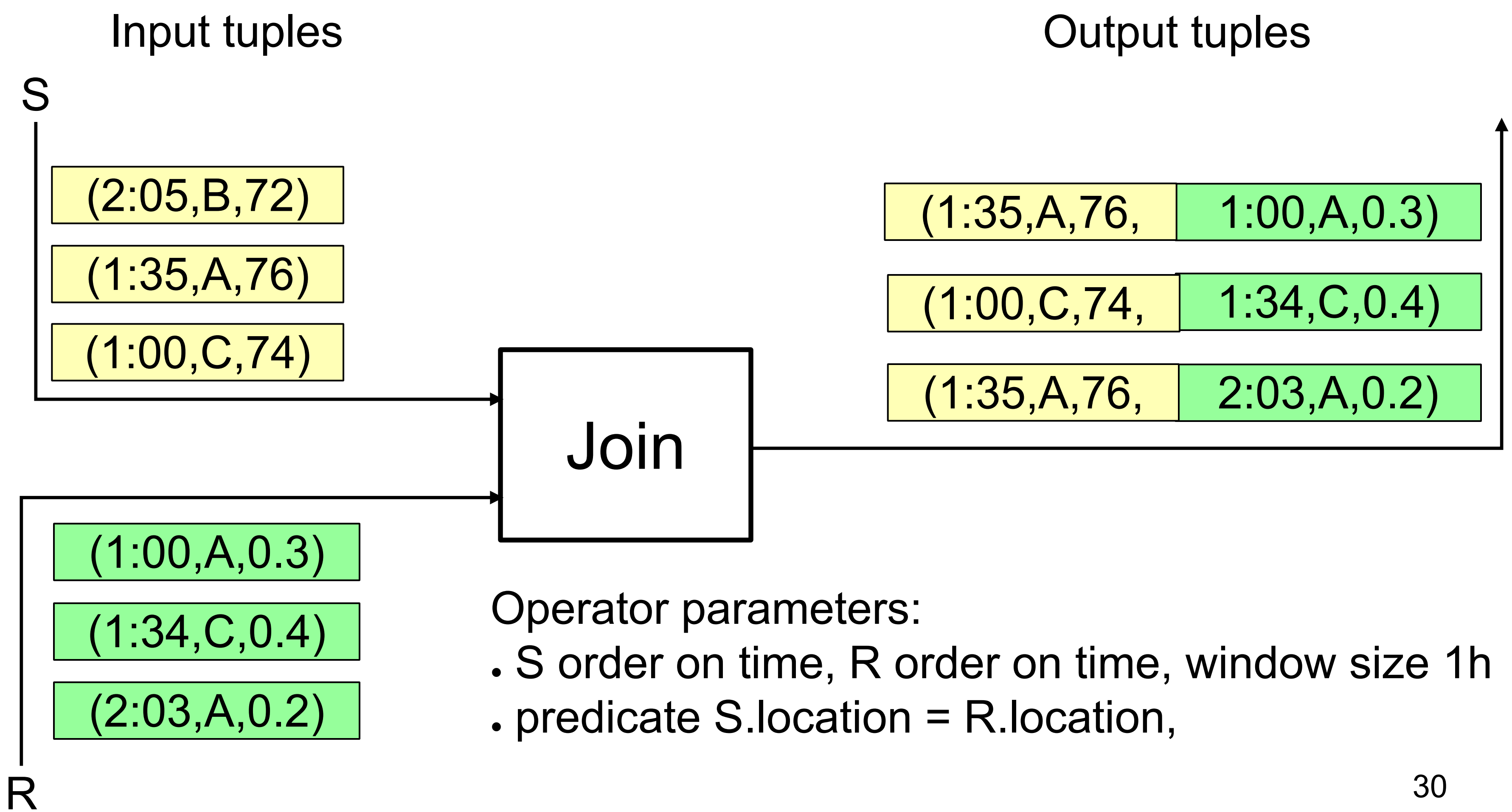


Input schema: (time, location, temp)

Operator parameters:

- group by location
- average temp,
- order on time, window size 1 h, advance 1h

Join Example



Operator parameters:

- S order on time, R order on time, window size 1h
- predicate S.location = R.location,

The distributed perspective

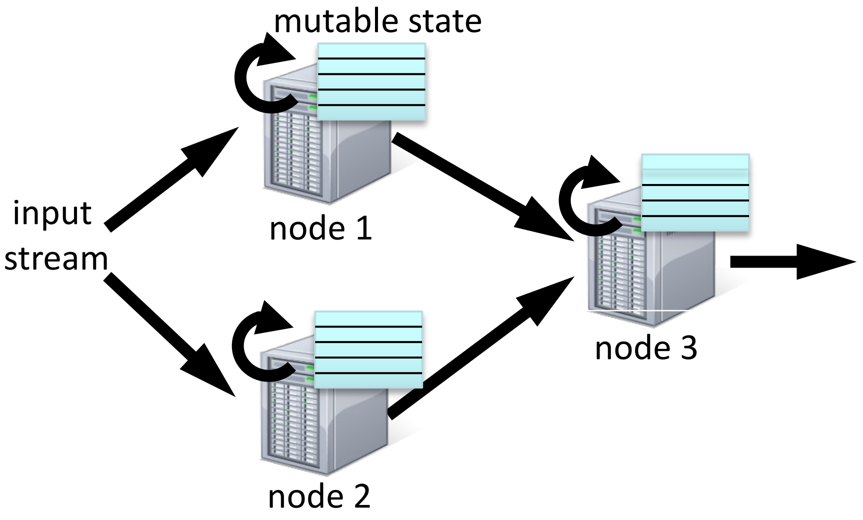
SPARK STREAMING

Challenges in DSMSs

- Dynamic Scaling
  - Scale by adding new resources
- Fault Tolerance
  - Crashes (node goes out of service)
  - Stragglers (one of the nodes is slower than others)

Challenges in DSMSs (cont.)

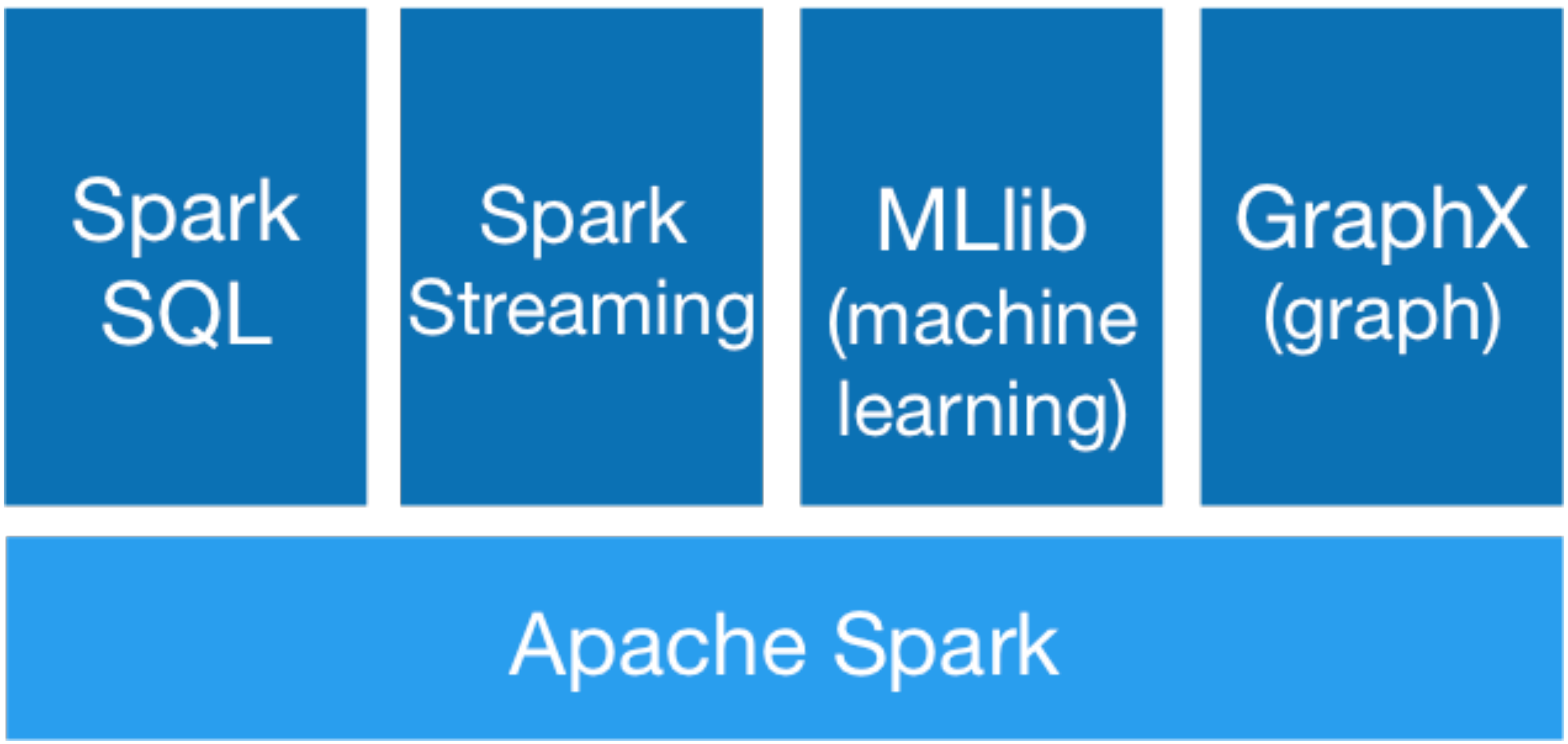
- DSMSs usually have an event-driven record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state & send new records
- Replication
  - Duplicate each node and each input stream
  - Fast recovery but 2x hardware cost
- Upstream backup
  - Replay the stream when failure occurs
  - Slow recovery
- Synchronization is an issue in these solutions
- Neither approach handle stragglers



What is Spark Streaming?

- Extends Spark to support stream processing
- Efficient and fault-tolerant stream processing system
- Scales to 100s of nodes and achieves sub-second latencies
- Batch-like API for implementing complex algorithms

Spark Ecosystem



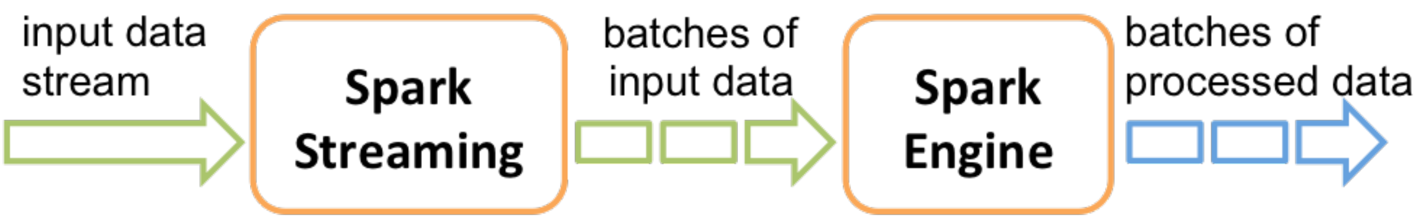
Observations

- Lessons from Hadoop (MR / Spark)
  - Data processing models in clusters provide fault tolerance efficiently
  - Divide jobs into deterministic tasks
  - Rerun failed/slow tasks in parallel on other nodes

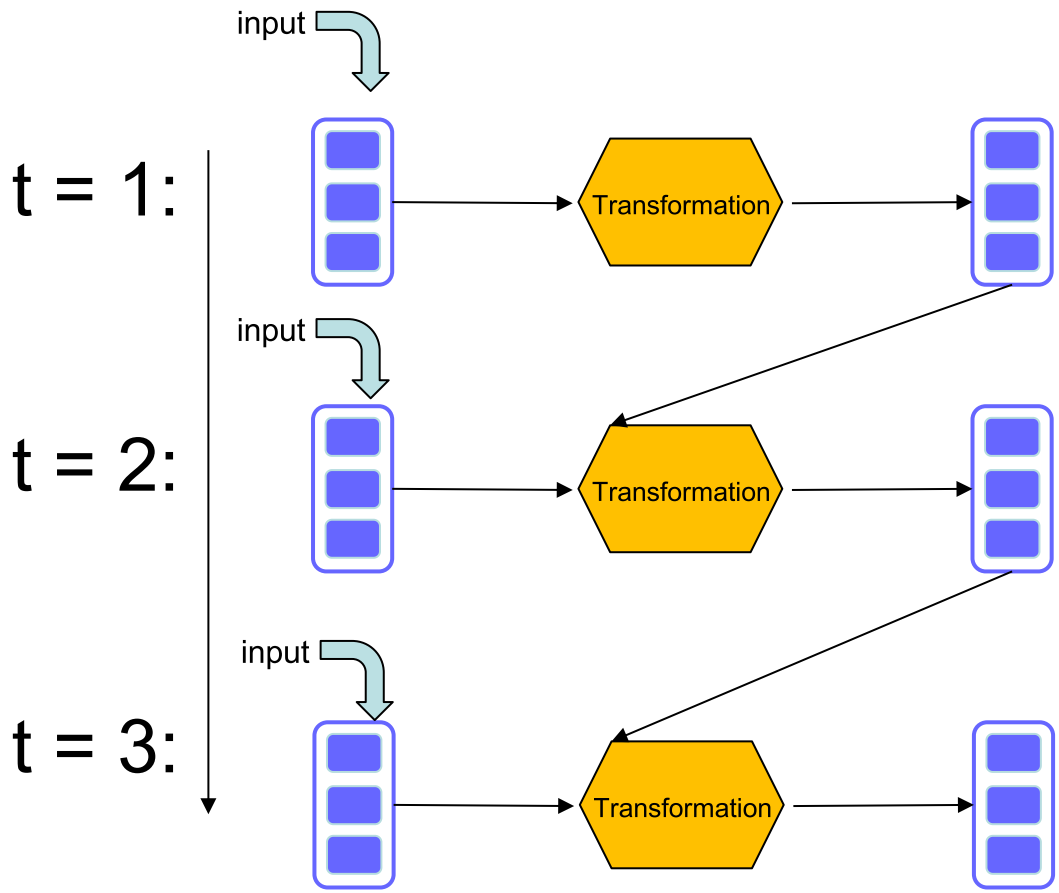
- Idea: *Run streaming computation as a series of small, deterministic batch jobs*
  - Same recovery scheme at much smaller timescale
  - Store state in RDD

Key Concepts

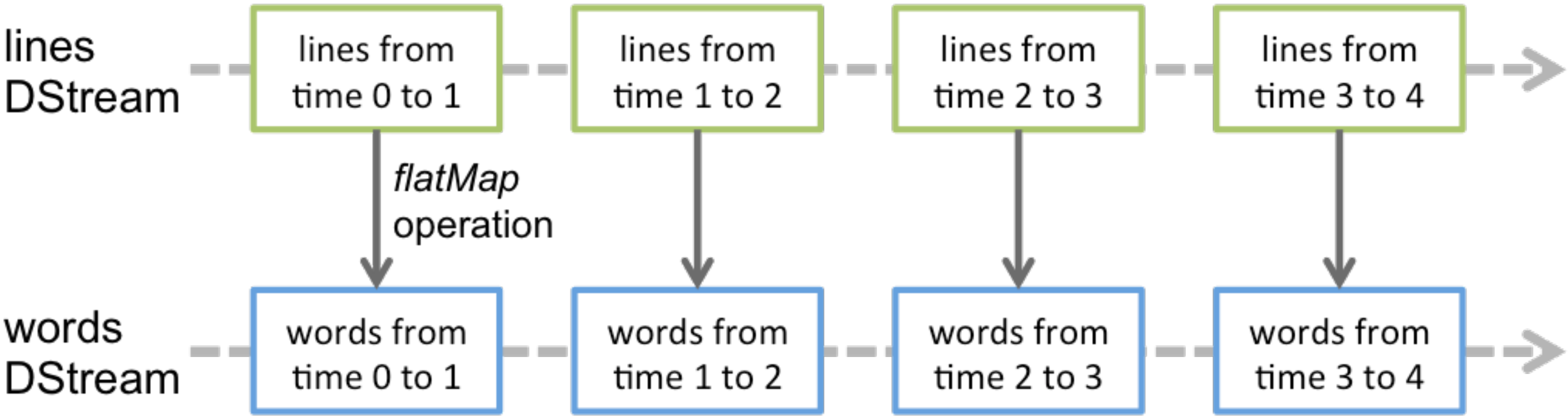
- **DStream** – sequence of RDDs representing a stream of data
  - Abstraction: DStream containing RDDs for each time window
  - Data sources such as: Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results



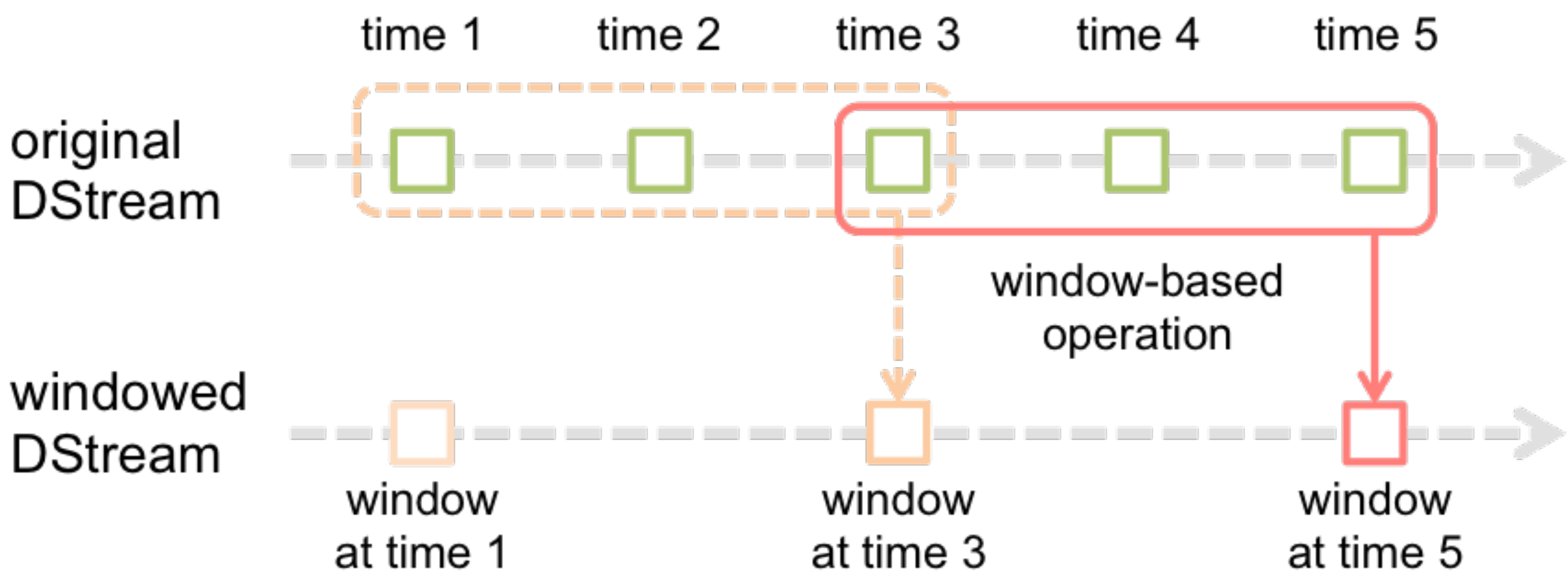
Discretize Stream Processing



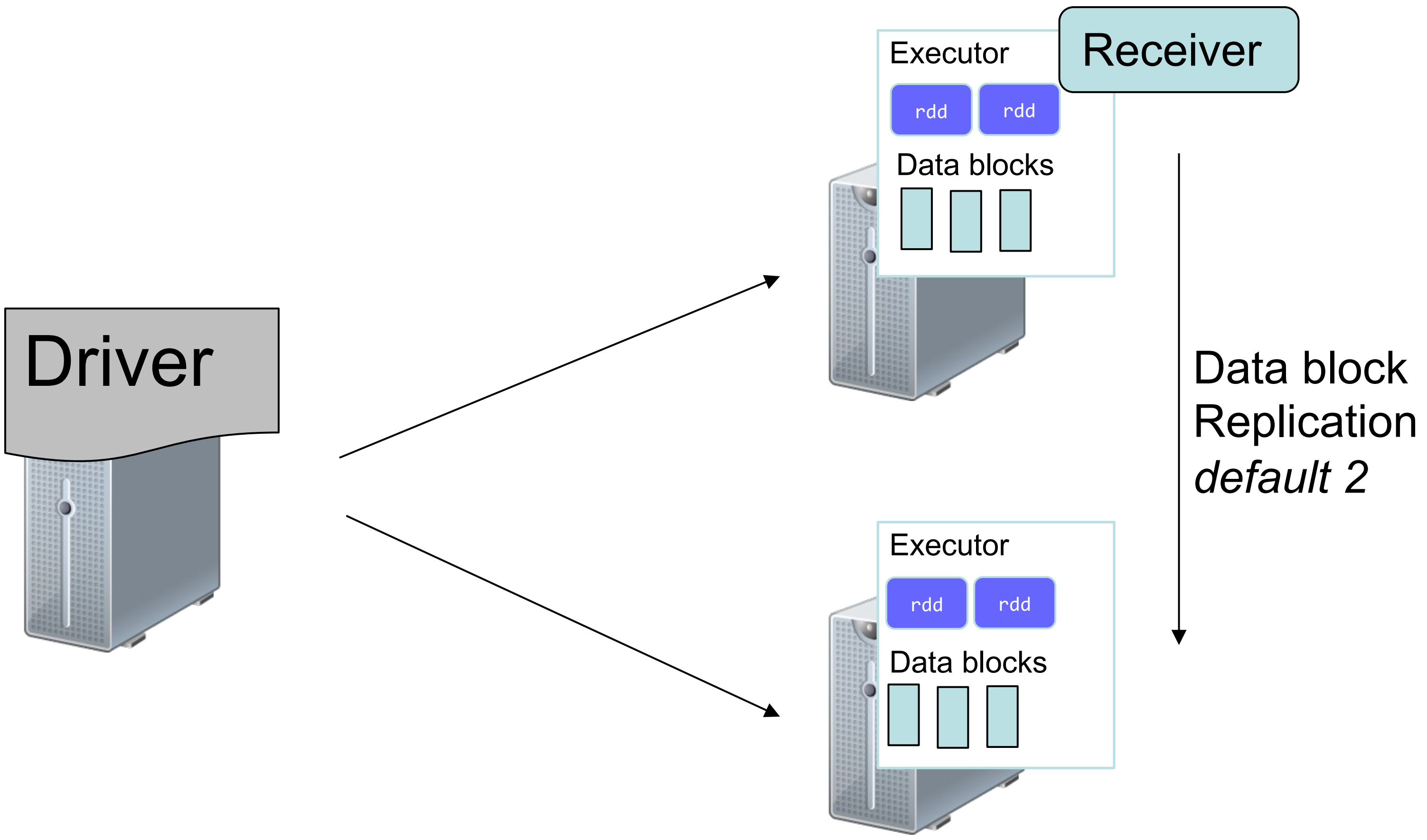
Discretize Stream Processing



Window Operations



Spark Streaming Internals



Fault Tolerance

- Executor with a receiver crashes
  - Restart receiver on a new executor
  - Use replicated data blocks

- RDD Checkpointing
  - Stateful stream processing can lead to long RDD lineages
  - Long lineage = bad for fault-tolerance, too much re-computation
  - RDD checkpointing saves RDD data to the fault-tolerant storage to limit lineage and re-computation

- Driver crashes
  - DStream checkpointing: save DAG periodically to storage (HDFS)
  - How about the lost data blocks?
    - Use Write Ahead Logs (WAL) i.e., write the data blocks to hdfs and read from it upon restart.
    - If Kafka is used, replay the log (next lecture!)