Eötvös Loránd University (ELTE)
Faculty of Informatics (IK)
Pázmány Péter sétány 1/c
1117 Budapest, Hungary

CONSISTENCY IN STREAM PROCESSING PIPELINES

Stream mining (SM)

Imre Lendák, PhD, Associate Professor

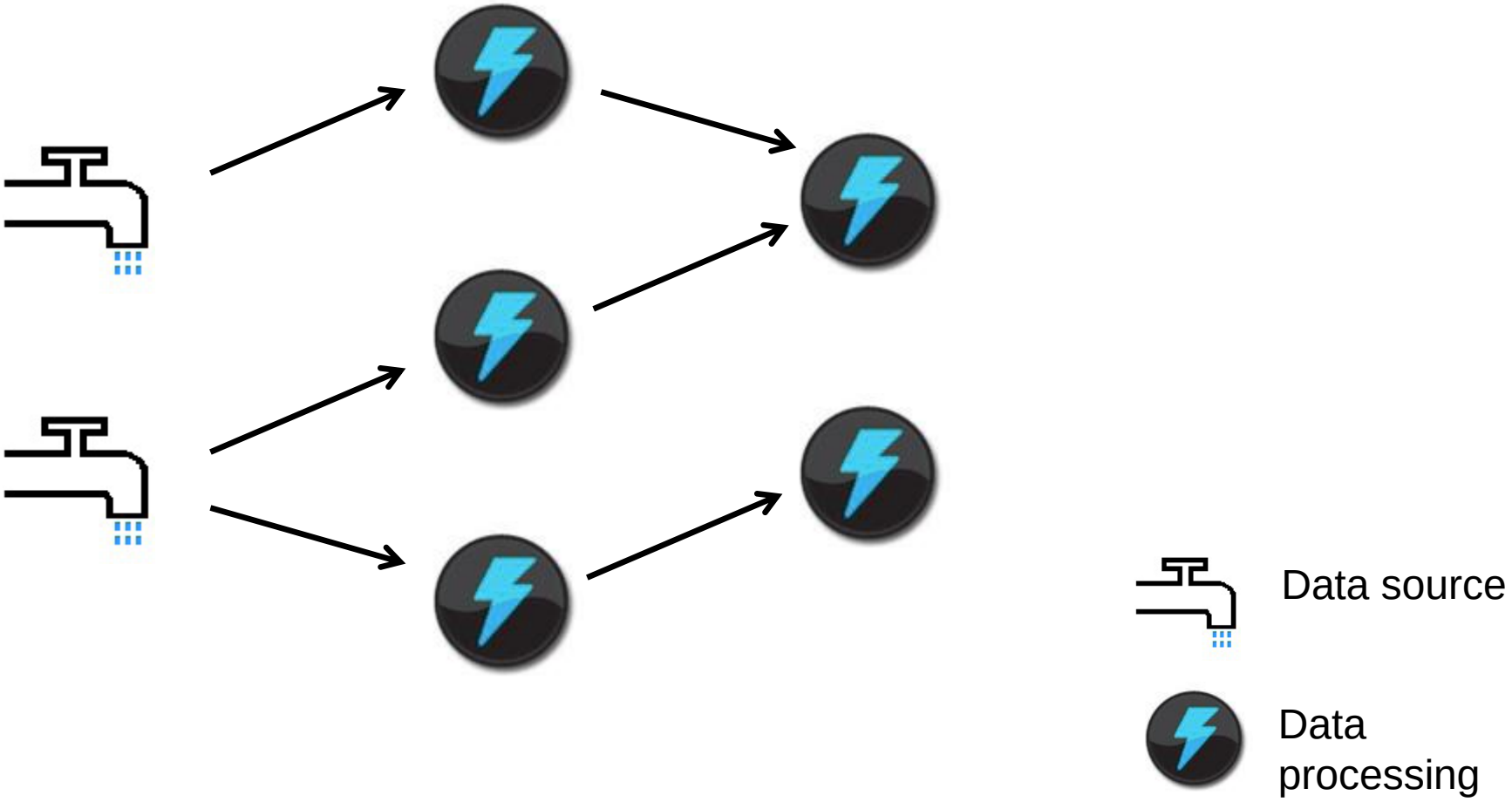
Péter Kiss, PhD candidate

2020
Budapest, Hungary

Outline

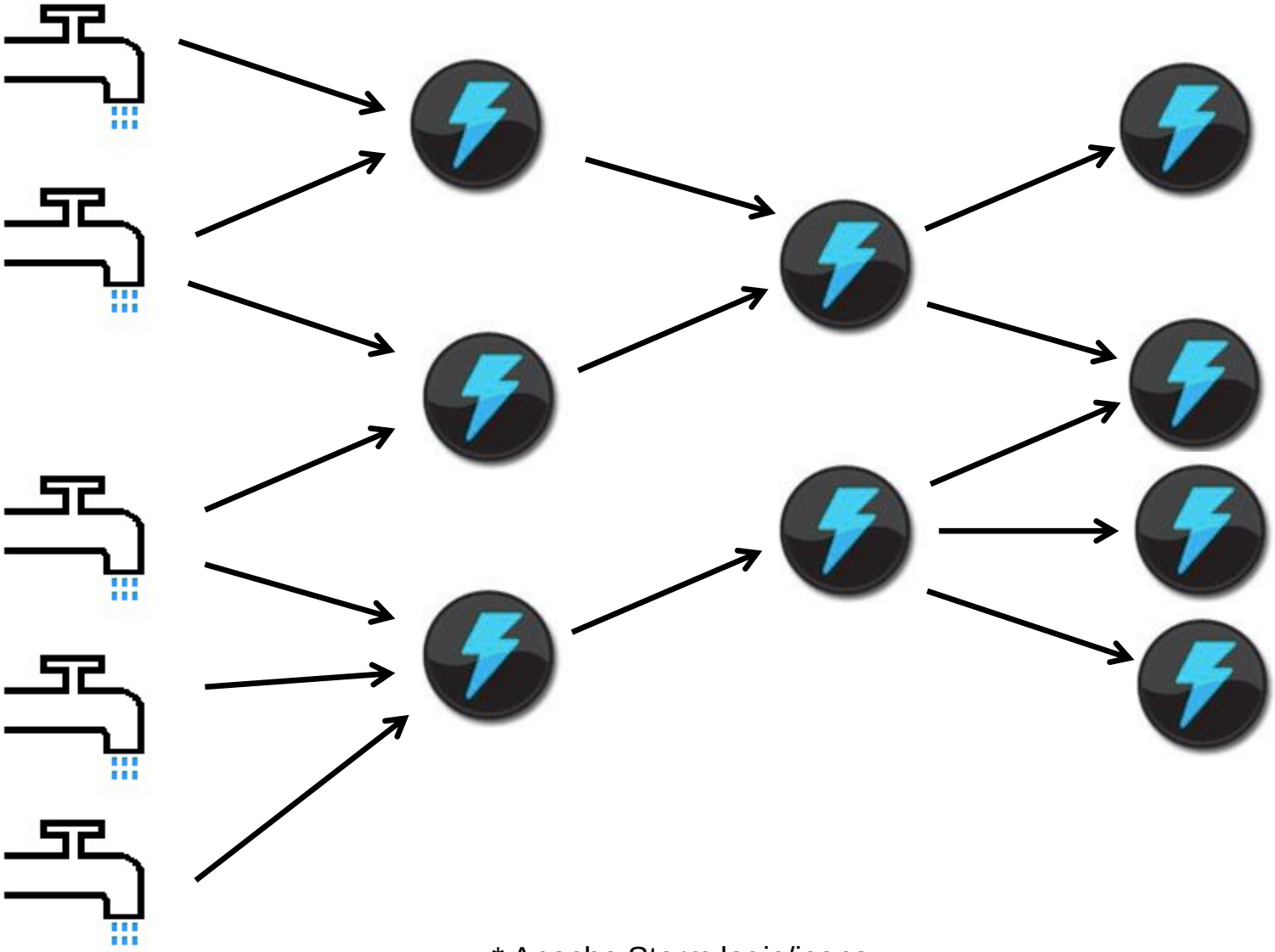
- Consistency intro
- Consistency ‘levels’
- Checkpoints
- Platforms & consistency

Simple stream processing pipeline



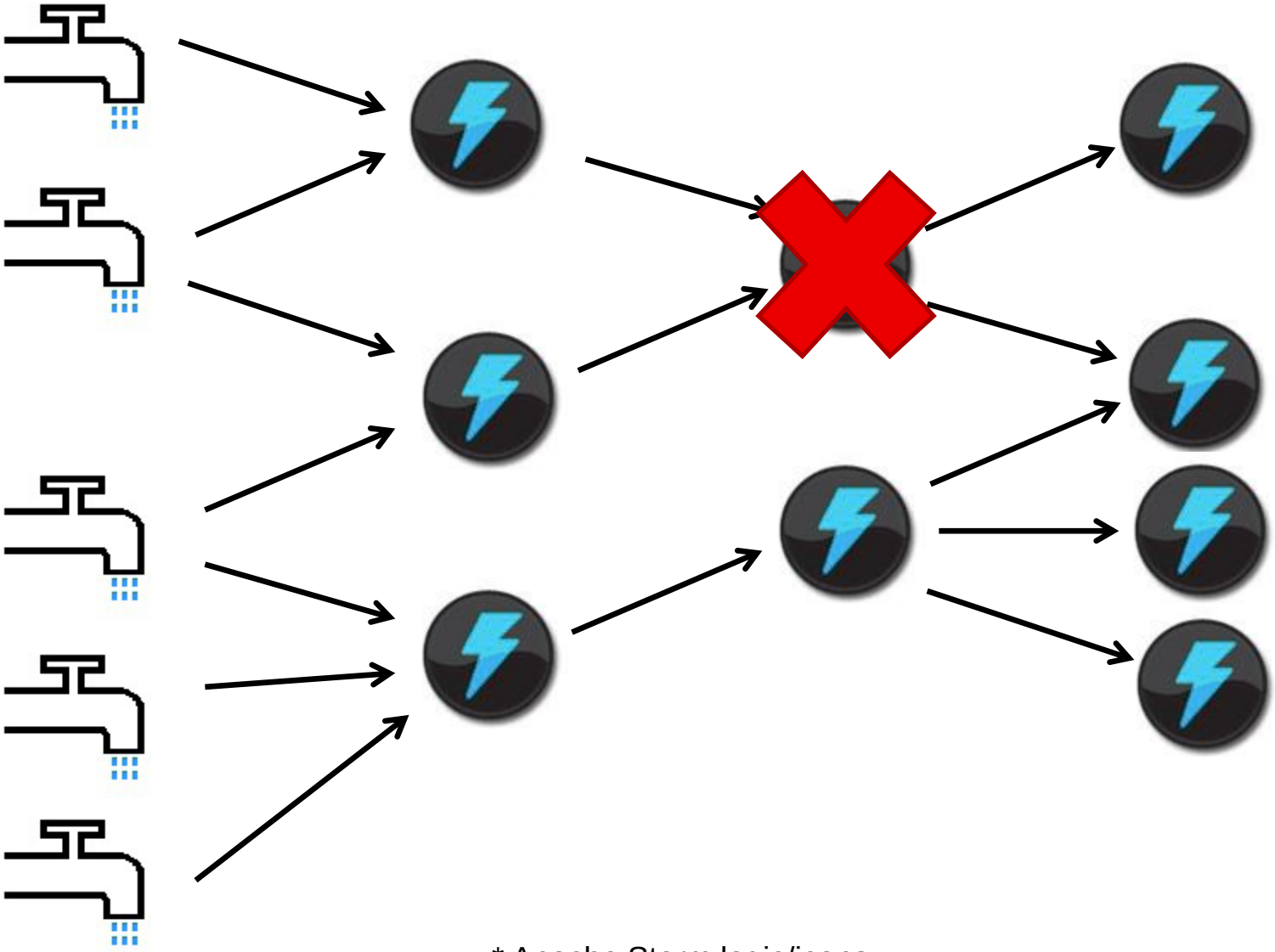
* Apache Storm logic/icons

Complex stream processing pipeline



* Apache Storm logic/icons

Outage in a processing pipeline



* Apache Storm logic/icons

Planned service interruptions

- Hardware maintenance downtimes, e.g. replacement of outdated/failed gear
- Code changes, e.g. migrating from Spark 1.x to a newer version
- Issues can be caused by the human element as well
  - E.g. misconfiguration, configuring/unplugging the wrong device

Unplanned service interruptions

- Hardware failures are inevitable in any distributed system
  - CPU/memory/IO device/power supply failure
  - Communication channel related failure, e.g. cabling, misconfigured switches/routers
- Software failures are inevitable in any distributed system
  - Firmware in networking gear, e.g. switches, routers
  - Operating systems
  - Middleware
  - Applications

Detecting unplanned interruptions

Fault types in DS

- Crash failure
  - Processing node crashes
- Omission failure
  - Input data lost due to communication error
- Timing failure
  - Allowed lateness missed
- Response failure
  - Output buffer write error
- Byzantine fault
  - Processing node generates unplanned outputs

Fault detection types

- Active → based on heartbeat signal
- Passive → based on polling
- Centralized detection
  - Watchdog process monitors processing node statuses
- Distributed detection
  - Detectors assigned to processing nodes
  - Hard to implement
  - Numerous control messages

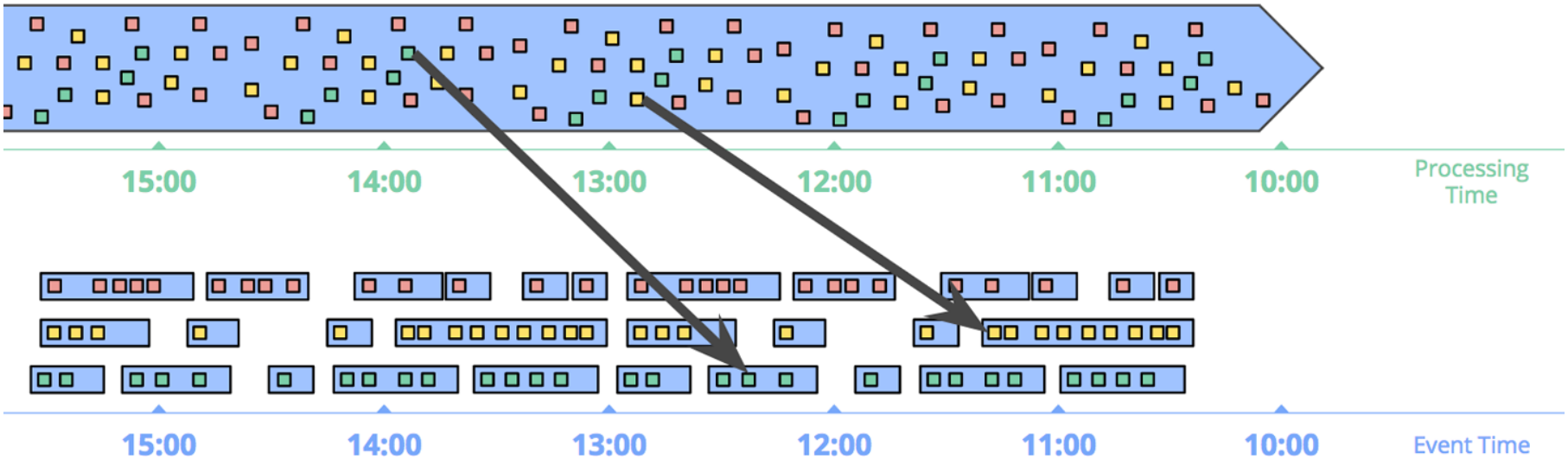
Key challenges

- The assumption of **batch processing systems** (e.g. RDBMS) was that data is stored in persistent storage and processing can be repeated
- The assumption of the 1st **generation streaming systems** was that data is ephemeral (i.e. not stored) and processed at least once if received
- Both above use cases:
  - Assume that failures are infrequent
  - Accept the extra cost of recomputation
- The key change brought by the **2nd generation streaming systems** (since ~2011-2012) is that they provide consistency guarantees
  - A few systems provide strong consistency guarantees

TABLES AND STREAMS

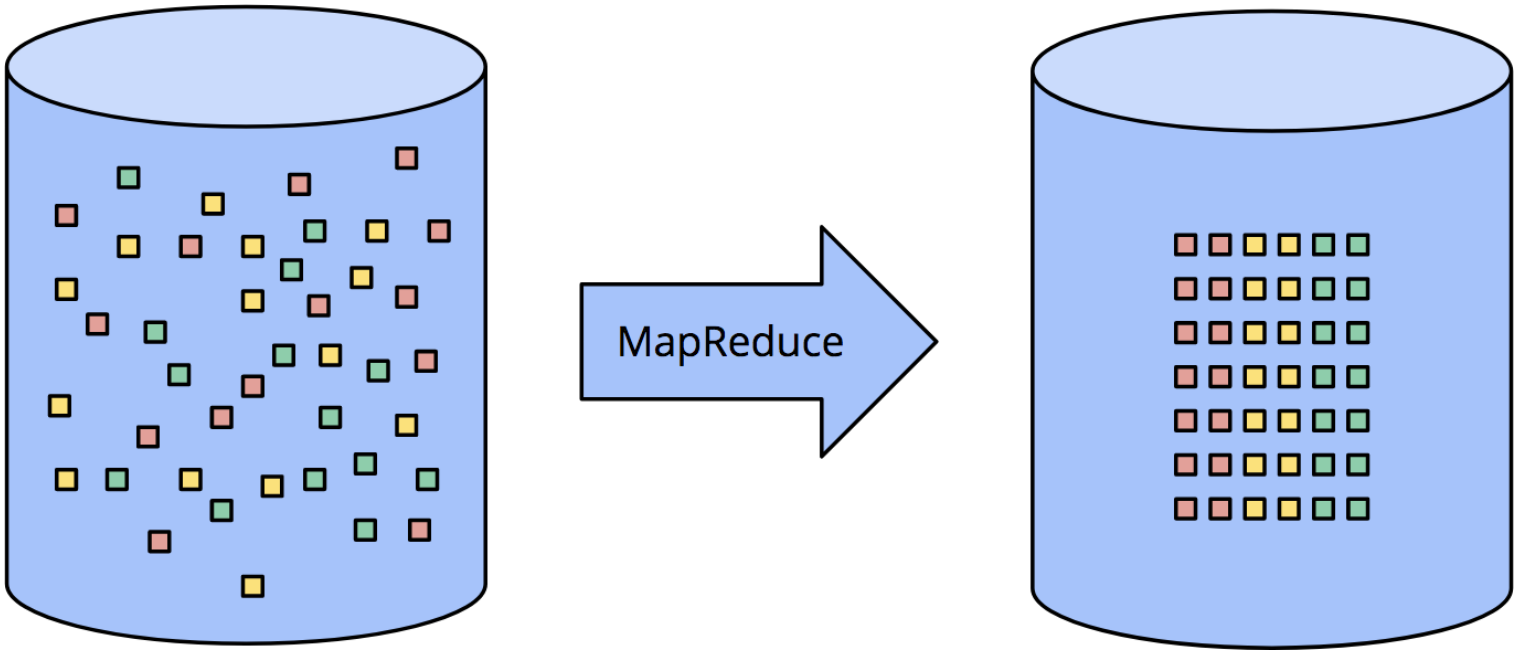
Streams

- **Cardinality:** **Unbounded data** is infinite in size.
- **Constitution:** A **stream** is an element-by-element view of the evolution of a dataset over time.
- **Alternate stream definition:** A data stream is an ordered (not necessarily always) and potentially infinite sequence of data points (e.g. numbers, words, sequences).

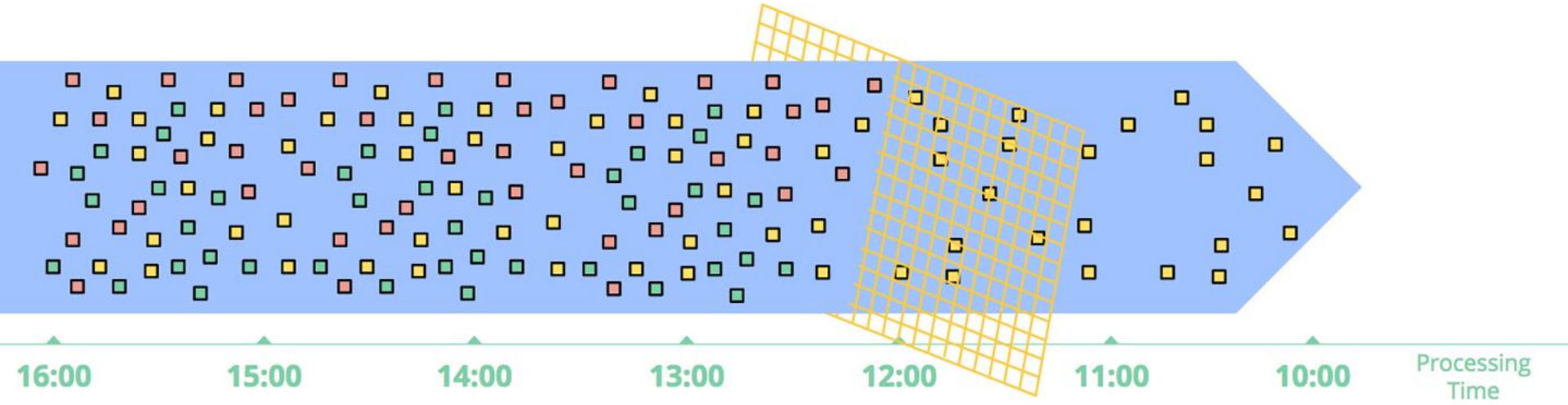


Tables

- **Cardinality:** **Bounded data** is finite in size.
- **Constitution:** A **table** represents a holistic (~complete) view of a dataset at a specific point in time.



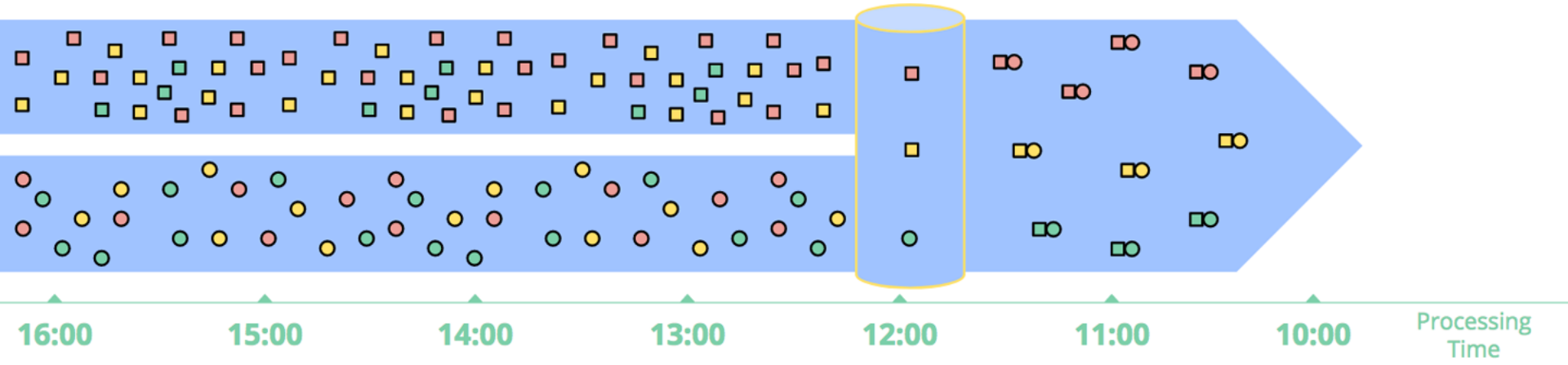
Non-grouping operations



http://streamingsystems.net/fig/1-5

- Non-grouping operations in stream processing pipelines accept streams as inputs and transform them into new streams
- **Input:** **stream** of records/observations
- **Output:** transformed **stream** of records

Grouping operations



http://streamingsystems.net/fig/1-6

- Grouping operations in stream processing pipelines accept streams, transform and group them
  - E.g. joins, aggregations, ML training, histogram creation
- **Input:** **stream** of records/observations
- **Output:** a **table**

Triggers (revisited)

- **DEF:** Triggers declare when output for a window should happen in processing time, i.e. when data should be processed
- Trigger types:
  - **Repeated update triggers** = most common in streaming systems, generate periodic updates for a window. Updates materialized for
    - Every new record → too many processing → high CPU load
    - After some processing time delay, e.g. 5 minutes
  - **Completeness triggers** = materialize each time a window is (believed to be) complete to certain degree, e.g. ~80% of data, ~90% of data, etc.
    - Allow reasoning about missing and late data

Triggers in table-to-stream conversion

- Grouping operations = **stream-to-table conversion** (see previous slides)
- After (stream) data is grouped into windows, triggers dictate when to process and send data downstream
- In stream processing pipelines the (above) downstream data triggered is again a stream → triggers drive **table-to-stream** conversion in streaming systems
- **Alternate trigger definition:** Triggers are special procedures applied to a that materializes transformed data in response to relevant events
- **Note:** A parallel can be drawn between triggers in streaming systems and in traditional database management systems → they are the same thing in different systems

PROCESSING GUARANTEES

Processing guarantees

- There are four levels of processing guarantees in general distributed systems
  - No guarantee → we do not know if delivery will succeed or not similarly to UDP over IP
  - At-least-once
  - At-most-once
  - Exactly-once
- These processing guarantees apply in streaming systems, which are just a specific type of DS

At-most-once

- In general-purpose distributed systems the at-most-once guarantee means that communication messages are **delivered zero or one times**
  - English: each message may be lost, but not duplicated
- In streaming systems this translates to each data record being processed zero or one times

- **Pro:** high performance, simple
- **Contra:** no correctness guarantees, non-deterministic systems

At-least-once

- In general purpose distributed systems at-least-once means that a communication message will be **delivered at least once to each recipient** (multiple recipients in message-oriented architectures!)
  - Potentially multiple attempts are made at delivering the communication messages → messages might be duplicated
- In streaming systems this means that the data record (i.e. observation) is delivered at least once to each processing node
- It might happen that the message is delivered **more than once**!
- Possible reasons leading to more-than-once:
  - Timing error – the acknowledge message 'came' with a delay longer than the configured timeout
  - Upstream processing or source failure
  - ...
- **Pro:** higher level of consistency
- **Contra:** complexity, duplicates, a bit more latency

Exactly-once

- In general-purpose distributed systems exactly-once delivery is a guarantee that each communication message is delivered exactly once to each recipient of that message
  - English: the message cannot be lost and/or duplicated
- In streaming systems an exactly-once guarantee means that each data record in the handled streams will be **processed exactly one time**
- **Pro:** correctness
- **Contra:** complexity, added latency

- Additional reading: https://www.ververica.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink

Exactly-once in sources

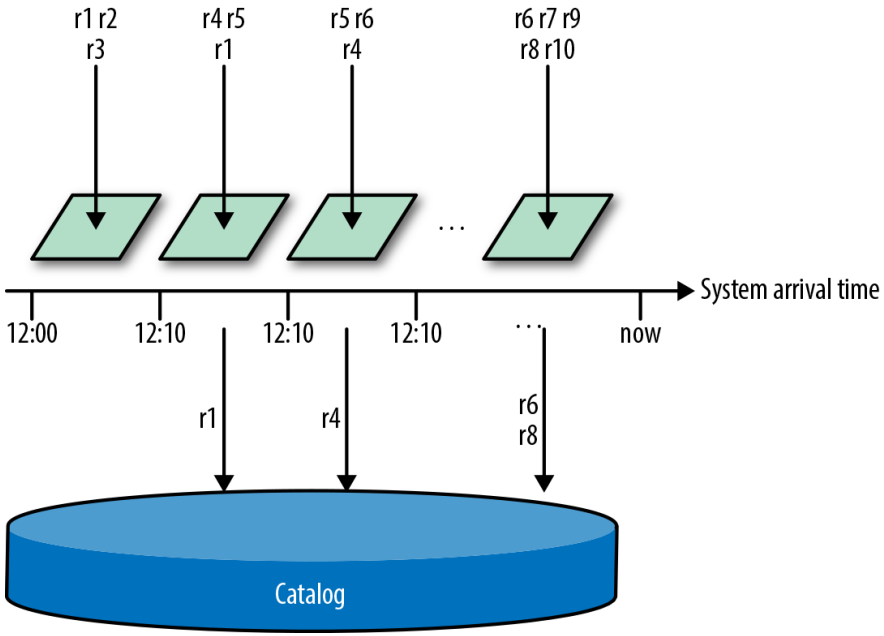
- Exactly-once is relatively straightforward to implement in **deterministic sources**
  - The records in a log file are deterministic, always located at the same offset
  - Some streaming transports also provide such deterministic guarantees, e.g. Kafka (discussed later)
- Exactly-once is a non-trivial task with **non-deterministic sources**, which might emanate different data records and send them to different recipients
  - Publish-subscribe message brokers are usually non-deterministic → if processing node fails, it may or may not send the same messages to the same subscribers in the same order
  - Solution: de-duplication handled in the processing nodes

Exactly-once in sinks

- Built-in sinks in modern streaming systems provide an **exactly-once output guarantee**
  - This means that output records are written to sinks with exactly-once guarantees in the presence of failures in the processing pipeline
- If it is necessary to implement a custom sink, then exactly-once can be guaranteed by implementing a **2-phase commit in custom sinks**
  - Phase I: prepare all necessary output data
  - Phase II: write output data to the sink(s)

Bloom filters vs duplicates

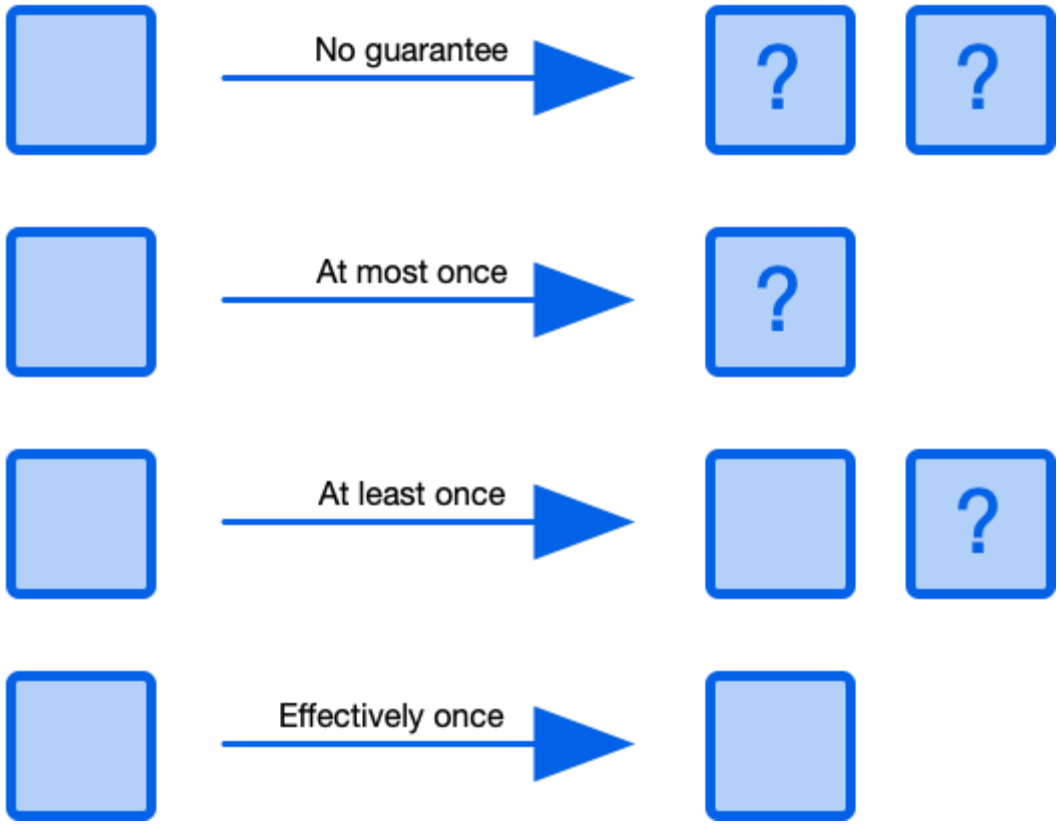
- **DEF: Bloom filters** are simple data structures designed for efficient set membership checks
- Characteristics of Bloom filters
  - They might return false positives
  - They never return false negatives
- In de-duplication they are used as they are
  - capable to always spot non-duplicates, and
  - sometimes trigger on false duplicates (i.e. false positives)
- Bloom filters fill up over time and the false positive rate increases → reconstruct periodically, e.g. every 10 min



Garbage collection

- In streaming systems it is usually not feasible to persist state for an extensive time at each processing stage
- Possible approaches to **garbage collection**, i.e. the deletion of data records which are no longer necessary for correctness
  - Assign unique, strictly increasing identifiers assigned to each record
  - Ingress (processing) timestamps assigned to each data record → as observed by the processing node
- If ingress timestamps are used, a **garbage collection watermark** can be calculated to trigger the optimal execution of garbage collection, i.e. delete observed inputs

Guarantee levels revisited



https://medium.com/@andy.bryant/processing-guarantees-in-kafka-12dd2e30be0e

CHECKPOINT WHAT? AND WHY?

Distributed streaming systems

- **DEF: Distributed systems** are complex systems designed to achieve one or more stated goals. They consist of hardware, software, documentation and people.
- Types of distributed systems
  - **Distributed computing systems**, e.g. streaming systems
  - Distributed information systems, e.g. global bank
  - Distributed pervasive systems, e.g. large-scale sensor network → Internet of Things
- Algorithms run in distributed computing systems
  - The processes executing the **basic algorithm** exchange basic messages and execute part of the distributed computation → processing nodes
  - The processes executing the **control algorithm** exchange control messages and oversee the basic algorithm's execution

Checkpoint intro

- **DEF:** Checkpoints are defined as snapshots of a complete or partial state of a distributed system
  - The DS consists of process nodes and communication channels and its state consists of process node and channel state fragments
  - Checkpoints allow **backward recovery** after failures
- It is a challenging task to create a checkpoint in any DS
  - **Storage:** Where to store the checkpoint?
  - **Timing:** When to initiate and how to sync the checkpoint?
  - **Performance:** How to minimize the additional load in the DS during the checkpoint creation process?
- Checkpoint creation types:
  - **Autonomous:** process nodes create checkpoint fragments separately (i.e. not synced)
  - **Coordinated:** selected elements create checkpoint fragments in a coordinated fashion

Message logging

- **DEF: Message logging** in distributed systems is the process of storing communication messages which alter the state of the DS
  - Note: messages storage in persistent storage
- Recovery can be a combination of **checkpoint + message logging** since the last checkpoint
  - Phase I: Revert the system to the last checkpoint
  - Phase II: Replay all messages since the last (complete or partial) checkpoint

- **Preconditions:**
  - the underlying system is **deterministic** → message M received by processing node P in state S1 will always transition P to state S2
  - all (relevant) communication messages (i.e. data records in streming systems) are **persisted**

Checkpoint levels

- Processing nodes might implement different checkpointing strategies
- Any such checkpointing strategy needs to balance between to extremes
  - Always-persist-everything → good for consistency, bad for efficiency
  - Never-persist-anything → bad for consistency, good for efficiency
- We will analyze the following strategies
  - Always-persist-everything
  - Persist groupings/increments
  - Generalized state

Always persist everything

- In the ‘always-persist-everything’ scenario the stream processing pipeline is capable to store
  - each piece of data
  - at each processing node
- How does it work? Atomic elements of the data stream are appended to the list of all prior elements received
- **Pro:** excellent consistency
- **Contra:**
  - Not a viable solution in real-life scenarios with extremely high loads, e.g. monitoring websites with extremely large numbers of users
  - High CPU load with periodic processing triggers

Persist groupings/increments

- In the persist groupings/increments strategy the processing nodes persist groupings of the inputs
  - Also known as ‘incremental combining’ in the Streaming systems book
  - E.g. a SUM processing stage can store the partial sum of observed inputs and the number of inputs
- It is necessary that the operation performed by the processing stage is both commutative and associative
  - OP(a,b) == OP(b,a)
  - OP(OP(a,b),c) == OP(a, OP(b,c))
- These use cases are excellent for parallelization on multiple physical or virtual computers

Generalized state

- If it is infeasible to buffer all data records and/or the (processing) operation is not commutative and/or associative we need more flexibility → generalized state with (the below) three levels of flexibility
- Flexibility in the use of **data structures** → in generalized state the process node is not limited to storing a list, number or pair of numbers. It should be allowed to store different data types, e.g. set, map
- Flexibility and **write and read granularity** → the processing node should be capable to choose the amount and type (see above) of data stored for optimal efficiency. It should be also possible to do large read/write in parallel.
- Flexibility in **scheduling** processing → ability to choose the time at which processing occurs:
  - **Completeness triggers** bound to watermarks (event time)
  - **Repeated update triggers** in processing time
  - **Timers** bind processing to a specific point in time in either of the two supported time domains

PLATFORMS AND CONSISTENCY

Spark Streaming & consistency

- Apache Spark Streaming implements a microbatch architecture for stream analysis
  - Under the hood Spark Streaming groups the data into a series of Resilient Distributed Datasets (RDDs) which are processed sequentially → all this happens in the processing time domain
  - Spark implements an **exactly-once guarantee via batch processing**

- **Pro:** exactly-once
- **Contra:** latency (due to batch processing) → clever tuning necessary to avoid long delays

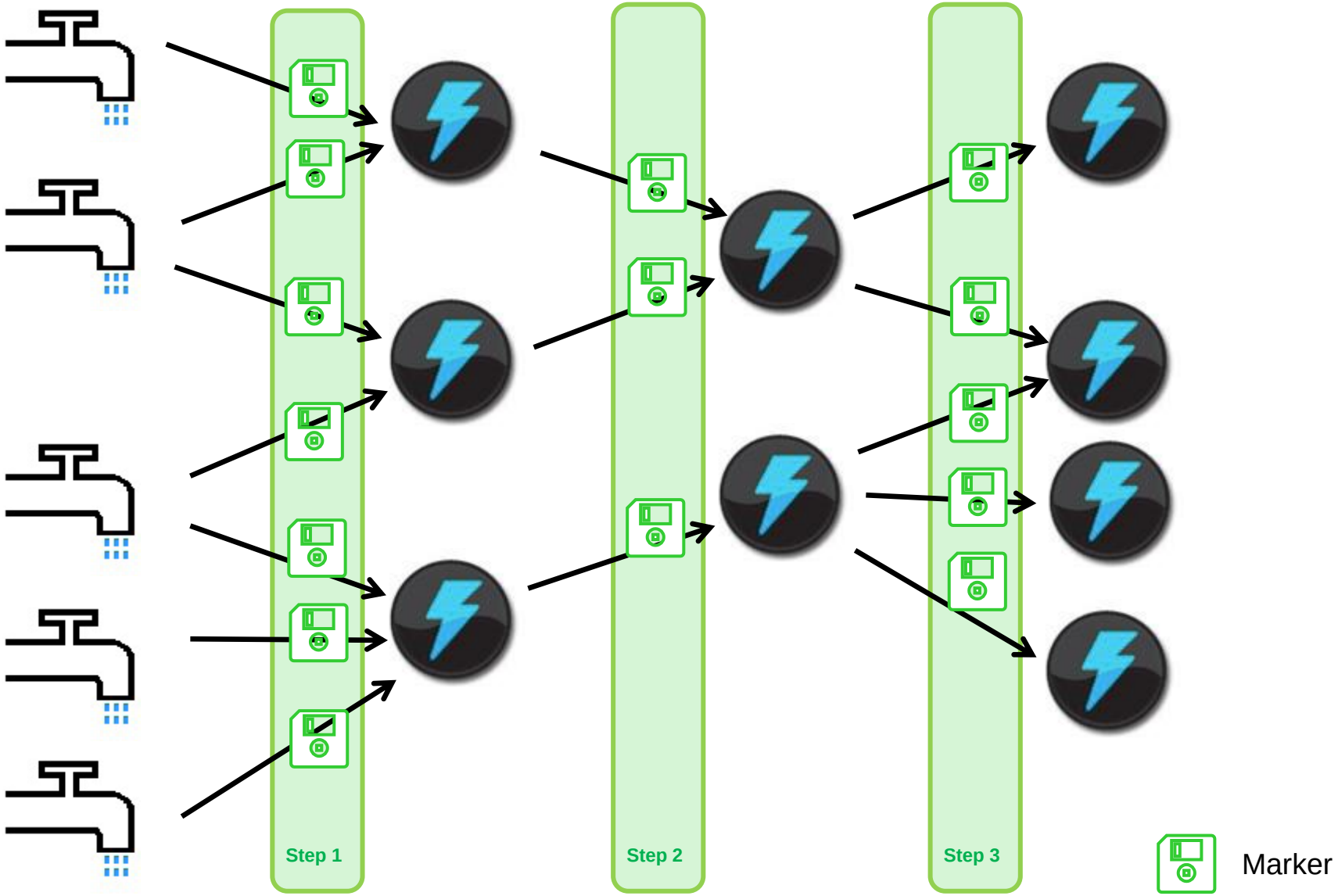
Kafka & consistency

- **DEF:** Kafka is a **(streaming) transport layer**, not a full-blown stream (processing) framework
  - Kafka is a persistent streaming transport implemented as a set of partitioned logs
- Kafka introduced **durable, replayable input sources** in 2011-2012 (among the 1st of its kind)
- Prior to Kafka streaming systems handled streams as ephemeral data, which meant:
  - Lack of durability → streams not persisted
  - Lack of replayability → streams could not be (easily) repeated for testing, (ML) training or during fault recovery
- **Note:** Kafka is often used as part of modern stream processing pipelines as a transport layer

Flink & consistency

- Apache Flink processing pipelines periodically compute **consistent snapshots** → point-in-time state of the entire stream processing pipeline
  - Flink snapshots are computed without halting the base algorithm
  - Assumption: processing tasks are statically allocated to ‘workers’
- Flink **adds markers** to data records flowing through the stream processing pipeline which aid its snapshot creation process
- When a processing node receives a snapshot marker, it **persists its state** and **hands the marker** to downstream processing nodes (via emitted data records)
  - The snapshot is completed when all nodes do the above
- **Sinks wait until snapshot completion** before sending their results to the outside world
- **Pro:** exactly-once in a truly streaming system manner
- **Contra:** added latency

Marker-based snapshot creation



Summary

- Consistency levels
- Tables and streams
- Checkpoints
- Platforms & consistency

Thank you for your attention!