Stream Processing

Lecture 1

2022/2023

Table of Contents

The need for stream processing

- Applications dealing with continuously flowing data, from geographically distributed sources, at unpredictable rates, that need to obtain timely responses to complex queries
  - Wireless sensor networks
  - Financial tickers
  - Fraud detection
  - Traffic management
  - Logistics systems, etc...

Why is this different?

- The concepts of timeliness and flow processing are crucial for justifying the need for a new class of systems
- Traditional DBMSs:
  - Require data to be (persistently) stored and indexed before it can be processed
  - Mostly designed to process data only when explicitly asked by the users, i.e., asynchronously with respect to its arrival
- Example: Detecting fire in a building by using temperature and smoke sensors
  - A fire alert has to be notified as soon as the relevant data becomes available
  - There is no need to store sensor readings if they are not relevant to fire
    - The relevant data can be discarded as soon as the fire is detected, if it does not have any extrinsic value to the fire detection application.

Dynamic Data Example

- **Using a sensor network measuring temperature and smoke, for fire alerts**
  - We want data to be processed continuously for detecting the fire prone conditions, and not only when users query
  - We don’t want to store all the measurements, especially those that have nothing to do with fire conditions.
  - Even those that alert for fire, are only needed until the fire alert is emitted; after that we may discard them
- **Implementing this in a system designed for static data (e.g. a DBMS) is not adequate**

Tools for processing streams

- Complex event processing
- Stream processing systems
- Time-series databases

Complex event processing

- CEP typically:
  - Goal: more oriented towards detecting patterns of events
  - Use high-level declarative language like SQL, or a graphical user interface
  - CEP engine performs the required matching, emitting event when the pattern is detected

  - Roots: publish-subscribe messaging systems; continuous queries in database systems

Stream processing systems

- Stream processing typically:
  - Goal: more oriented towards producing aggregations and statistical metrics
  - Moving from low-level interfaces to declarative languages

  - Roots: modern stream processing systems derive from Big data parallel processing frameworks

Time-series databases

- Time-series databases typically:
  - Goal: monitor the operation of machines, processes, etc.
  - Moving to declarative languages
- Roots: special purpose monitoring software, continuous query databases

Distributed Stream Processing Systems

- Why distributed stream processing systems?
  - Scalability
    - Impossible to process all events in a single machine
  - Provide fault-tolerance
    - Need to tolerate server failures
  - Latency
    - Need to provide results fast, in a timely manner
  - Data is distributed
    - E.g.: processing sensor data

Roadmap for the first part of the course

- Intro to big data frameworks
- Stream processing systems
  - Non-structured programming
  - Structured programming and SQL
  - Continuous streaming
- Stream processing ecosystem and IoT
- Storage for streamable data

Table of Contents

- Introduction
- Big data frameworks: map-reduce
- Big data stream processing intro

Google's MapReduce: summary

- "a programming **model** and an associated **implementation** for processing **large datasets**"
- "runs on a large cluster of **commodity machines** … a typical … computation processes many terabytes of data on **thousands** of machines"
- "a new abstraction that allows us to expresses **simple computations** we were trying to perform but **hides the messy details** of parallelization, fault-tolerance, data-distribution and load-balancing in a library"

Programming model

- Sequence of map and reduce stages
- Map: processes input (files); emit tuples
- Reduce: process tuples grouped by key; Emit tuples

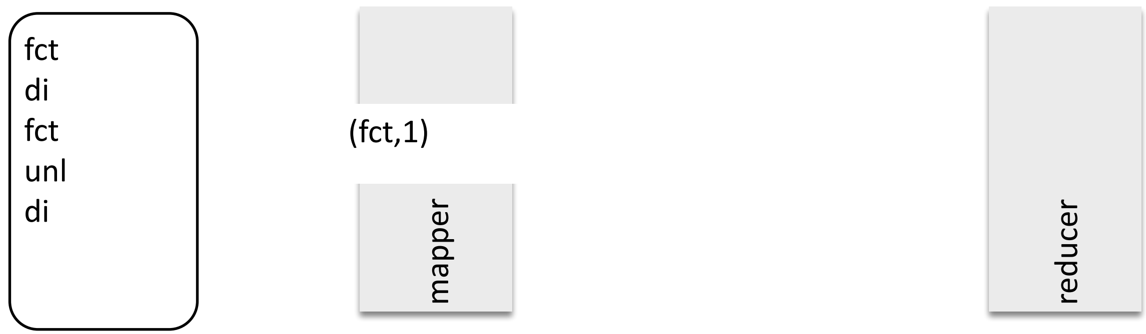
Programming model... working

- Example: count the number of times each word appears in a document (or documents)

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

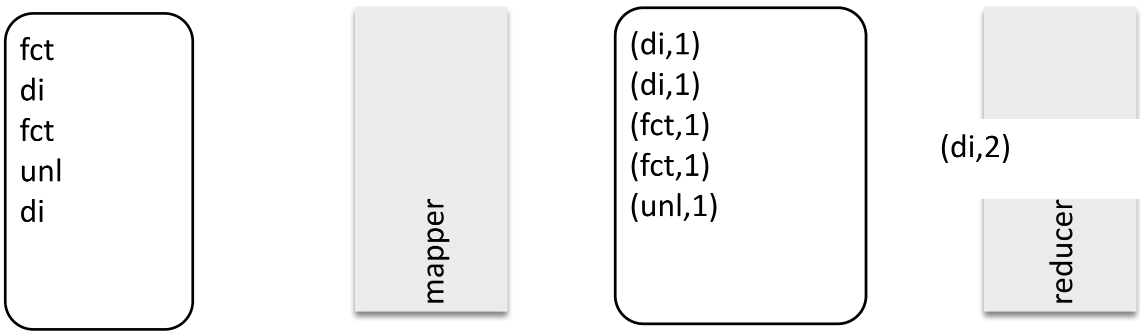
Programming model… working




```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, “1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

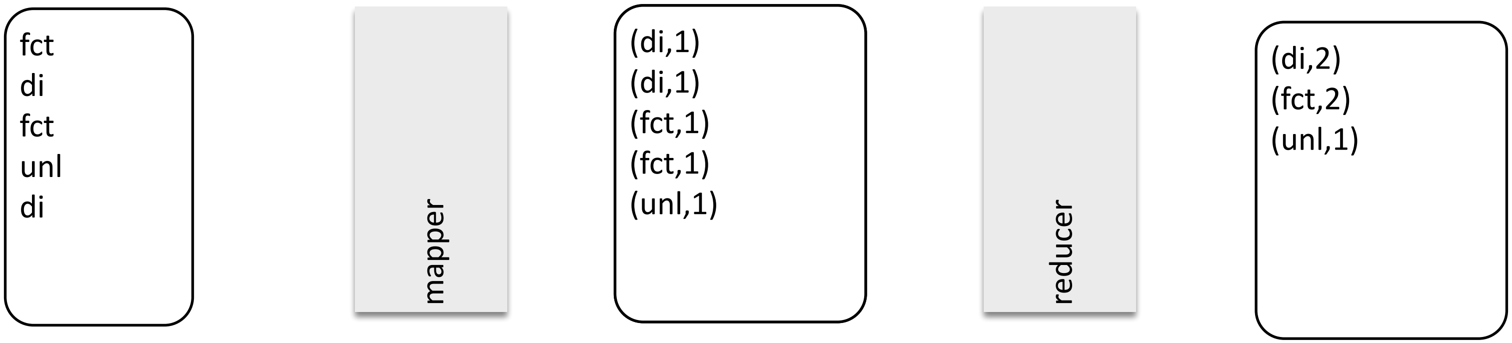
Programming model… working




```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

Programming model… working



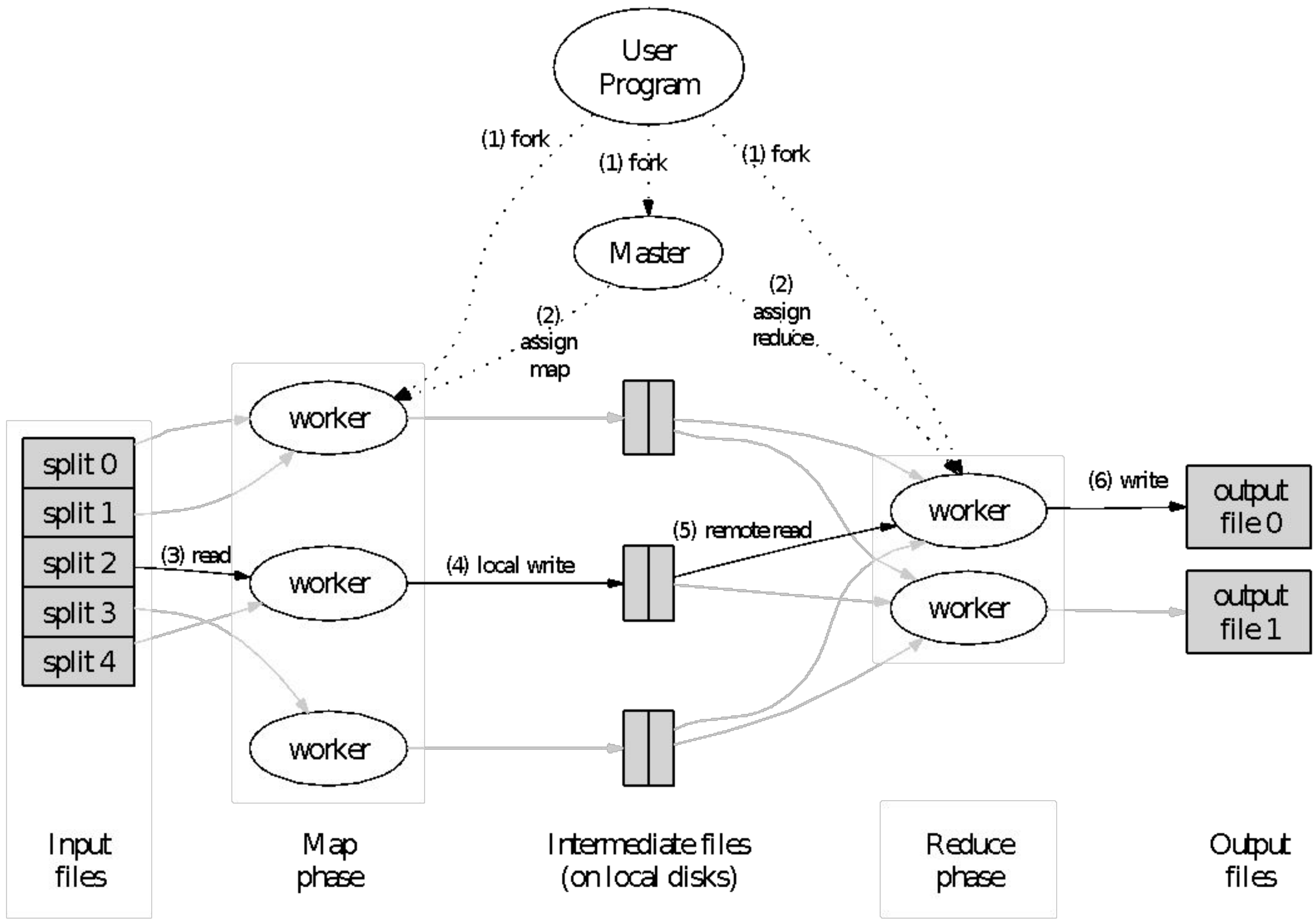

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, “1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

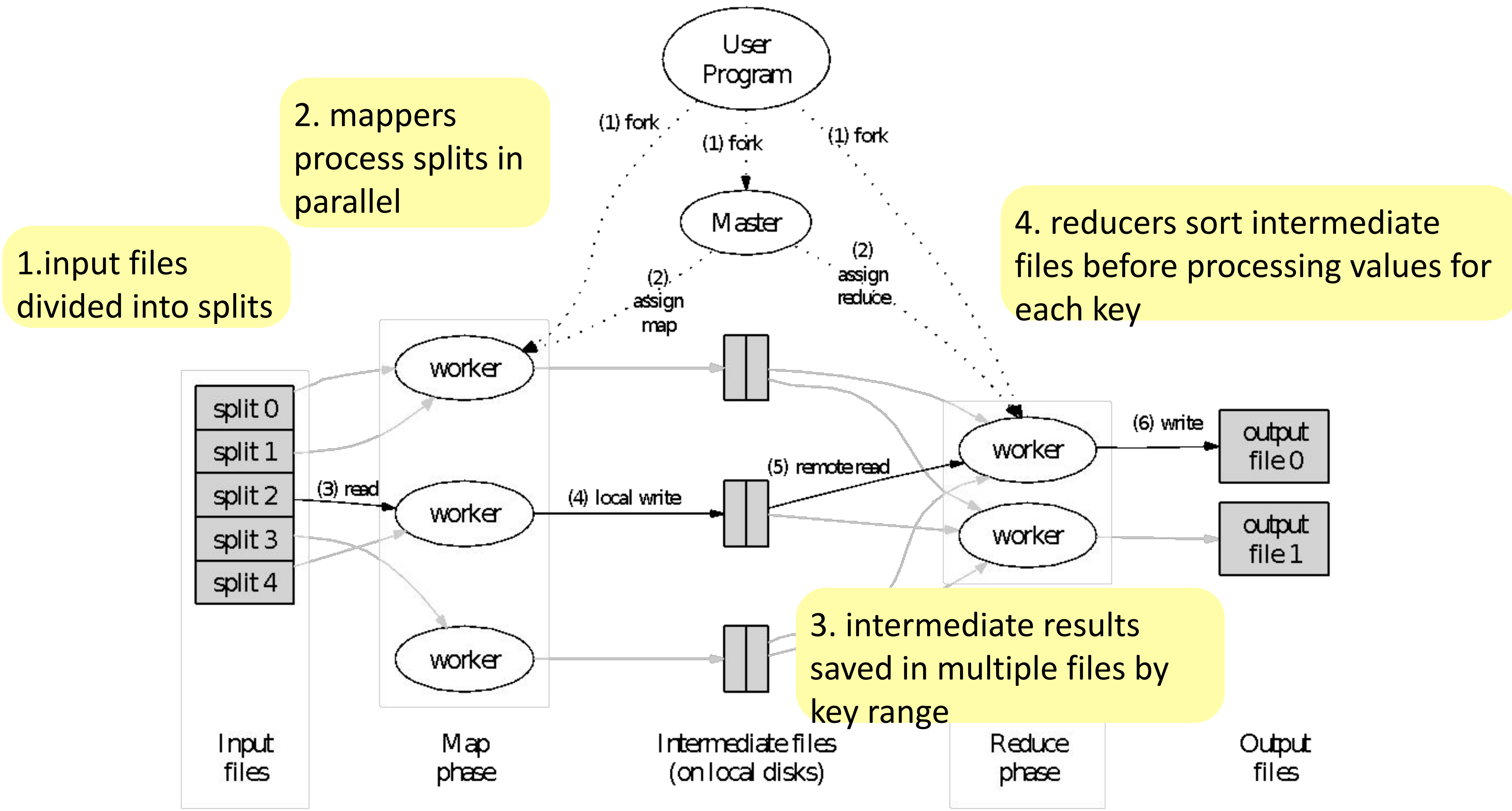
Programming model is not everything

- Programming model is simple, but…
- …how to run computations efficiently?

Map-reduce execution model



Map-reduce execution model



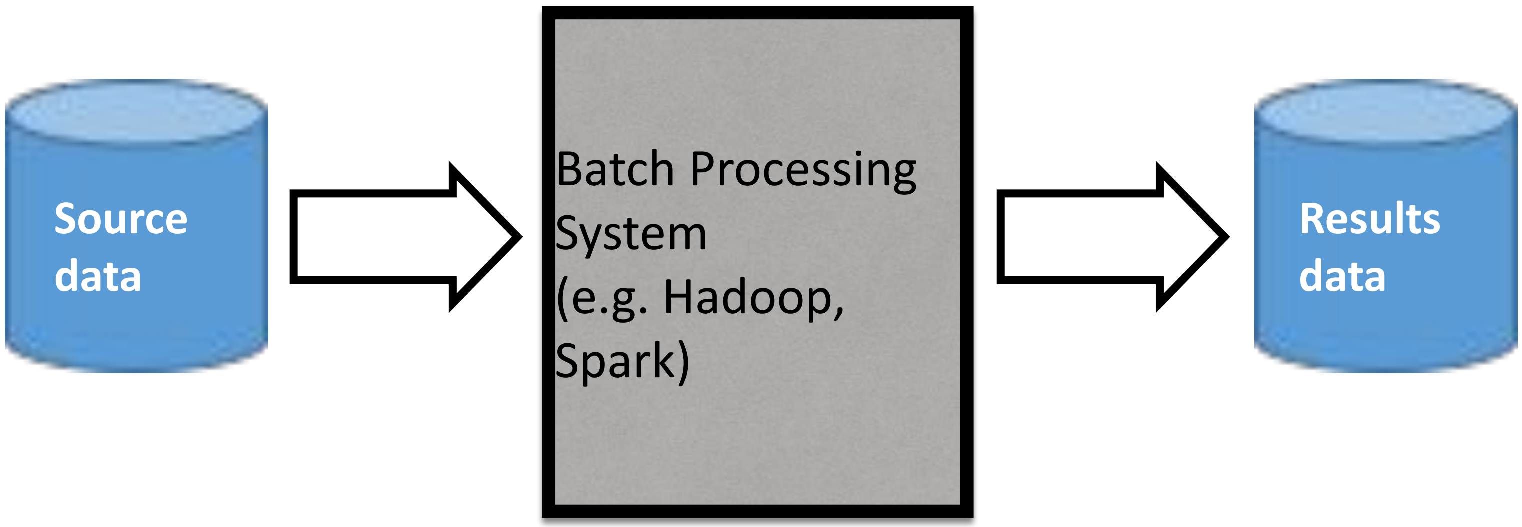
Limitations of map-reduce

- Scalable, but slow
  - Data stored on disk after each step
- Low-level programming
  - Simple programming model with no abstractions for helping writing programs
- Batch processing model not adequate for some applications
  - Need stream processing

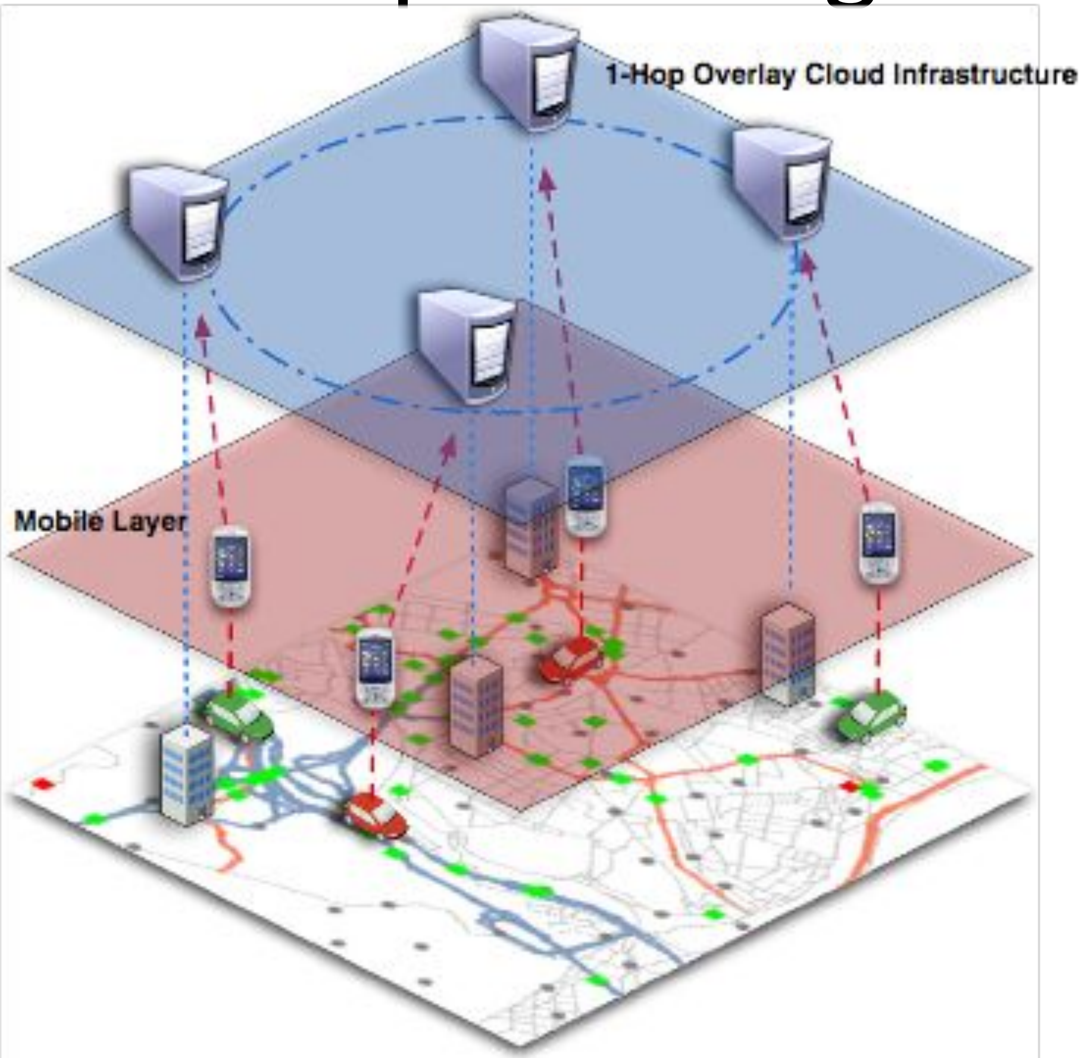
Table of Contents

Big Data / Batch processing

- All data known at the time of processing
- Goal: Execute computation over data and produce result
- Problem: what if new data arrives continuously, and new results should be computed continuously?



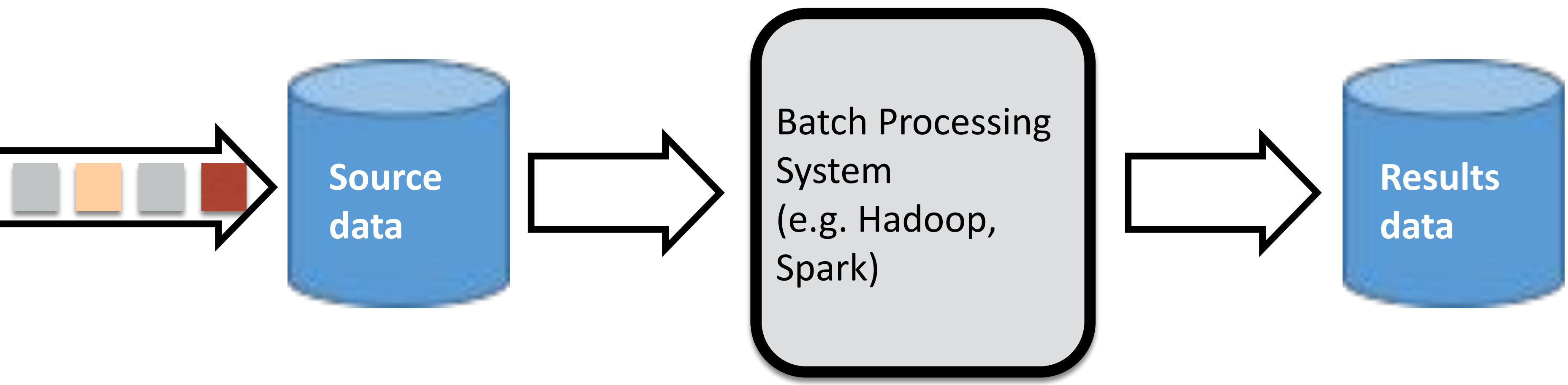
Examples of Big *Streaming* Data



Producing information on traffic, based on information collected from users' mobile phones

Big Streaming Data

- Can we use (batch) big data processing tools?
  - Save data as it arrives
  - Execute computation periodically - e.g. every hour
  - Problems?
  - Long delay for results, computation not incremental, …

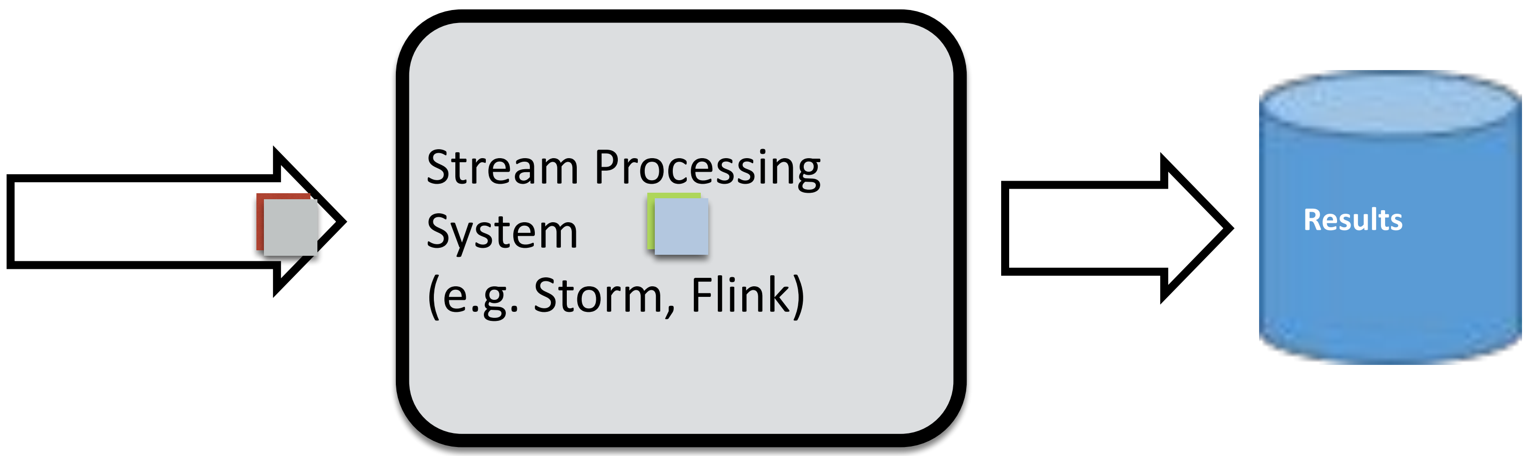


Big Streaming Data: requirements

- Need to process data as it arrives (or at most with a very small delay)
- Need to be able to process data from multiple sources
- Need to tolerate faults

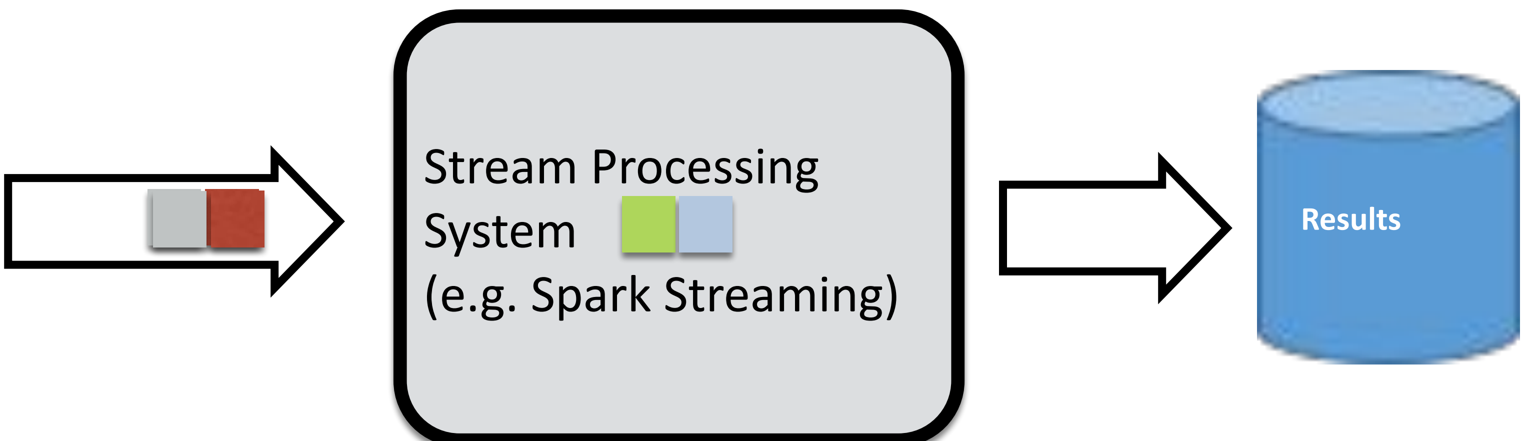
Two processing models (1)

- Continuous
  - Each tuple processed as it arrives
  - Processing system may keep state for executing window computations and incremental computations



Two processing models (2)

- Mini-batches
  - Tuples received for each X ms grouped in a mini-batch
  - Process mini-batches
  - Processing system may keep state for executing window computations and incremental computations



Stream processing: some issues

- Semantics
  - Reasoning about time
  - Joining multiple streams
- Performance
  - Latency
  - Fault tolerance
  - Sampling

Reasoning about time

- Stream processing often need to deal with time, but notion is tricky.
  - e.g.: compute X over the last five minutes. What does it mean?

Reasoning about time: event time

- Use the time of the event. Problems?

- Delay to start processing
  - Delays of event propagation
  - Have to deal with stragglers
    - Ignore straggler events
    - Issue correction of results
  - Have to deal with failures

Reasoning about time: process time

- Use the time the event reached the stream processing system. Problems?

- Combine events from different time periods
  - Delays of event propagation
  - Fault tolerance

Joining multiple streams

- Often needs to join events from multiple streams
  - e.g., in a website, associate search query with click on search.
- Stream-stream join
  - Need to be able to join an event with an event in the past
- Stream-table join
  - Store data in a table; join stream with data in a table

Stream processing: some issues

- Semantics
  - Reasoning about time
  - Joining multiple streams
- Performance
  - Latency
  - Fault tolerance
  - Determinism
  - Sampling

Latency in stream processing

- Some applications impose real time or bounded latency constraints on processing
- Results need to be produced at a rate compatible with the ingress rate
- Effects of fault-tolerance should be transient (and perceived as jitter, rather than accumulate).
- Partitioning can speed up computations, via parallelism, but can lead to some stragglers.
  - Not easy to anticipate. May be too late upon detection
  - Sensitive to input / improper partitioning

Fault tolerance in stream processing

- Batch processing
  - In worst case, can tolerate faults by re-computing everything
- Stream processing
  - Not usually feasible to replay the stream(s) from the very beginning
  - Implies some form of periodic checkpointing (or replication)

Determinism in stream processing

- Redundant processing is useful in some scenarios…
  – Can provide fault tolerance;
  – Mitigate the impact of stragglers in latency.
- Processing the same stream twice should yield the same stream of results.
- Algorithm should not depend on factors external to the data

Sampling in stream processing

- Execute processing over a fraction of the data. Why is this acceptable?
- For high ingress data rates, sampling may be employed to meet desired processing latency
- Sampling is not straightforward and impacts on the accuracy and interpretation of the processing results

Systems for stream processing

- Continuous processing
  - Apache Storm
    - Open sourced by Twitter
    - API: proprietary, SQL-like
  - Apache Flink
    - API: proprietary, table-based, SQL-like
- Mini-batch processing
  - Spark streaming
    - API: proprietary, table-based, SQL-like

Bibliography

- Martin Kleppmann. Designing data-intensive applications. Chapter 11.