Distributed Information Management

Daniel Kocher

Salzburg, Summer semester 2022

Department of Computer Science
University of Salzburg

database research group

PARIS LODRON UNIVERSITY SALZBURG

Final Exam

Date/time: June 29, 2022, 12:20 - 13:40 pm CET

Topics: Everything until June 20 and 22, 2022 inclusive, respectively

Part II

Data Processing

Literature, Sources, and Credits

Literature:

- Silberschatz et al. *Database System Concepts.* McGraw Hill, Sixth Edition, 2010. In particular Chapter 10 – *Big Data.*

Credits[1]**:** These slides are partially based on slides of other lectures.

- Slides of Silberschatz et al. *Database System Concepts.* McGraw Hill, Sixth Edition, 2010. In particular Chapter 10 – *Big Data.*

[1] Feedback credits: Werner Dähn

Introduction

Motivation

“Big data” that does not fit on a single machine needs to be **processed**
⇒ **Higher degree of distribution and parallelism**

Big Data Properties - “The three Vs”:

- **Volume:** Thousands of machines (nodes) are required to store and process the data (e.g., volume increase by factor 300× from 2005 to 2020).
- **Velocity:** Data arrives at a very high pace (*fast data*) and needs to be processed immediately to respond to certain events (e.g., real-time, batches, …).
- **Variety:** Different data formats are used for different purposes and may need to be processed collectively (e.g., logs, the actual data of an application, video, pdf, …).

Data Sources

The web and its various applications ⇒ Web logs.

- Recommendations
- User interaction patterns
- Advertisement
- …

Smartphone apps and data about the user interactions.

Sensors report data continuously and at a very high pace ⇒ **“Internet of things”.**

Data from **social media platforms.**

Metadata in communication networks to predict/prevent problems.

Data Processing Frameworks

Similar to database systems, there exist different solutions to process large amounts of data.

Problems?

- Satisfying the performance requirements is not easy.
  - Parallelism.
  - Load balancing.
  - …
- Dealing with failures in the distributed environment is not trivial.
- …

Data Processing Frameworks

Goal: A framework that implements these **functionalities transparently.**

- Allow **complex** data processing **tasks**.
- Transparent and automatic **parallelization** of the tasks.
- Built-in and transparent **fault tolerance.**

Data Storage

In *Part I – Data Management*, we covered:

- Different **models** and **systems** to store complex data.
- **Parallel** and **distributed** database systems.
- **Fragmentation** (aka *sharding)* and **replication.**

We have **not yet** heard about **distributed file systems.**

Distributed File Systems

Every computer/operating system has a **local file system (FS).** Effectively, the file system takes care of how the data is stored on your hard disk and how the user can retrieve it. Furthermore, the FS implements a common interface to access the files.

A **distributed file system (DFS)** provides the **same functionality** across a **cluster of nodes transparently**, i.e., the user interacts with the distributed FS as if it would be a local FS.

Examples: The Google File System (GFS) and the Hadoop File System (HDFS).

Distributed File Systems

Designed to store very large files (up to hundreds of gigabytes).

A **file** is split into **multiple blocks**, which are then **distributed across multiple nodes.** Techniques like fragmentation and replication are often used in combination to provide high availability.

Functionality:

- Hierarchical organization (i.e., directory structures).
- File reconstruction (i.e., mapping a filename to the distributed blocks).
- Access to a distributed file (through the filename).

The MapReduce Framework

Introduction

A generic framework (or paradigm) for a common situation in parallel computing:
Apply a function to each of our data items.

Specifically, we want to **apply two functions** one after another:

1. Apply a first function to each data item, the `map()` function.
2. Apply a second function to each result item of (1), the `reduce()` function.

Example – WordCount

Task: Count the occurrence of each word in a collection of files.

1. Single file on a single machine (node)
2. Multiple files on multiple nodes

Example – WordCount

Task: Count the occurrence of each word in a collection of files.

1. Single file on a single machine (node) $\Rightarrow$ Straightforward.
2. Multiple files on multiple nodes $\Rightarrow$ Not that easy …

Example – WordCount with MapReduce

Input file:

```
There is only one Lord of the Ring, only one who can bend it to his will.
```

Desired Result:

Word	Count	Word	Count	Word	Count	Word	Count
There	1	is	1	only	2	one	2
Lord	1	of	1	the	1	Ring	1
who	1	can	1	bend	1	it	1
to	1	his	1	will.	1		

Specify the **core logic** through two complementary functions, `map()` and `reduce()`.

Example – WordCount with MapReduce

Step 1: The `map()` function is **invoked on each input record**, and produces one or more **intermediate data items.** Each intermediate data item is a **key-value pair** (**rkey**, **value**).

Example – WordCount with MapReduce

The `map()` Function:

```
# Pseudocode in Python-like syntax.
def map(line):
  # We consider each line a record and split it by whitespace.
  for word in line.split():
    # Output the intermediate data item.
    # emit(x, y) is a pseudo function that outputs a pair (x, y).
    emit(word, 1)
```

Output:

```
("There", 1), ("is", 1), ("only", 1), ("one", 1), ("Lord", 1),
("of", 1), ("the", 1), ("Ring", 1), ("only", 1), ("one", 1),
("who", 1), ("can", 1), ("bend", 1), ("his", 1), ("will.", 1)
```

Example – WordCount with MapReduce

Step 1: The `map()` function is **invoked on each input record**, and produces one or more **intermediate data items.** Each intermediate data item is a **key-value pair** (**rkey**, **value**).

Step 2: (*rkey*, *value*) pairs are **grouped based on the key**, i.e., data items with the same key are grouped together. This results in **one list per key,** (**rkey**, **valuelist**).

Example – WordCount with MapReduce

(*rkey*, *value*) Pairs:

```
("There", 1), ("is", 1), ("only", 1), ("one", 1), ("Lord", 1),
("of", 1), ("the", 1), ("Ring", 1), ("only", 1), ("one", 1),
("who", 1), ("can", 1), ("bend", 1), ("his", 1), ("will.", 1)
```

(*rkey*, *valuelist*) Pairs:

```
("There", [1]), ("is", [1]), ("only", [1,1]), ("one", [1,1]), ("Lord", [1]),
("of", [1]), ("the", [1]), ("Ring", [1]), ("who", [1]), ("can", [1]), ("bend", [1]),
("his", [1]), ("will.", [1])
```

Example – WordCount with MapReduce

Step 1: The `map()` function is **invoked on each input record**, and produces one or more **intermediate data items.** Each intermediate data item is a **key-value pair** (**rkey**, **value**).

Step 2: (*rkey*, *value*) pairs are **grouped based on the key**, i.e., data items with the same key are grouped together. This results in **one list per key,** (**rkey**, **valuelist**).

Step 3: The `reduce()` function is **invoked on each** (**rkey**, **valuelist**) **pair** and typically **aggregates** the results for a specific *rkey* (i.e., word).

Example – WordCount with MapReduce

The reduce() Function:

```
# Pseudocode in Python-like syntax.
def reduce(rkey, valuelist):
  count = 0 # total number of occurrences
  for value in valuelist:
    count = count + value
  # Output the final word count.
  # emit(x, y) is a pseudo function that outputs a pair (x, y).
  emit(rkey, count)
```

Final Result:

("There", 1), ("is", 1), ("only", 2), ("one", 2), ("Lord", 1),
("of", 1), ("the", 1), ("Ring", 1), ("who", 1), ("can", 1), ("bend", 1),
("his", 1), ("will.", 1)

What about multiple files on multiple machines?

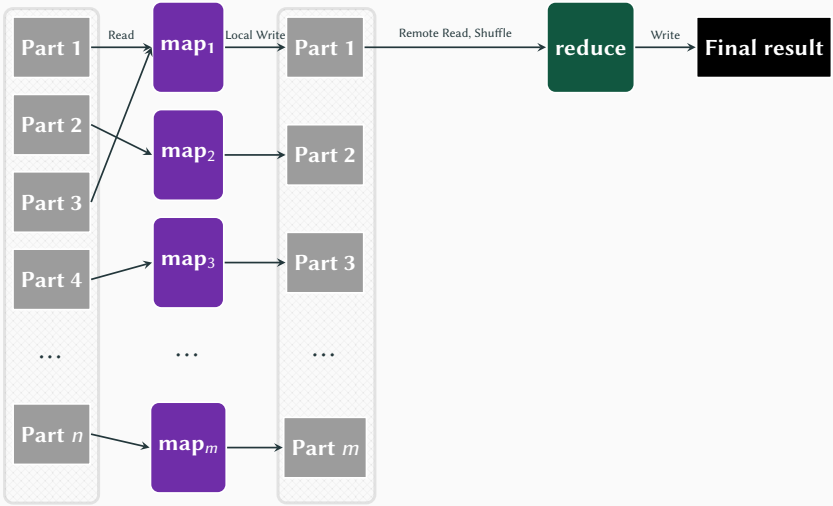
What about parallelism?

The MapReduce Framework

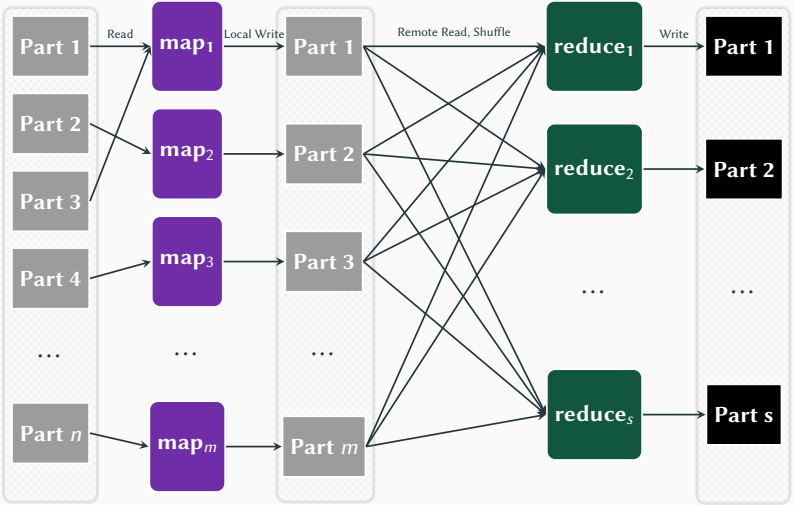
The MapReduce Framework

Input data on DFS

The MapReduce Framework



The MapReduce Framework



The MapReduce Framework

Each task (map/reduce) runs on a node, i.e., a node can be mapper *and* reducer.

Traditionally, MapReduce is **disk-based**, i.e., the input data for a map/reduce task is read from hard disk and the (intermediate) result is flushed back onto hard disk.

Disclaimer: MapReduce is not the solution to all problems.

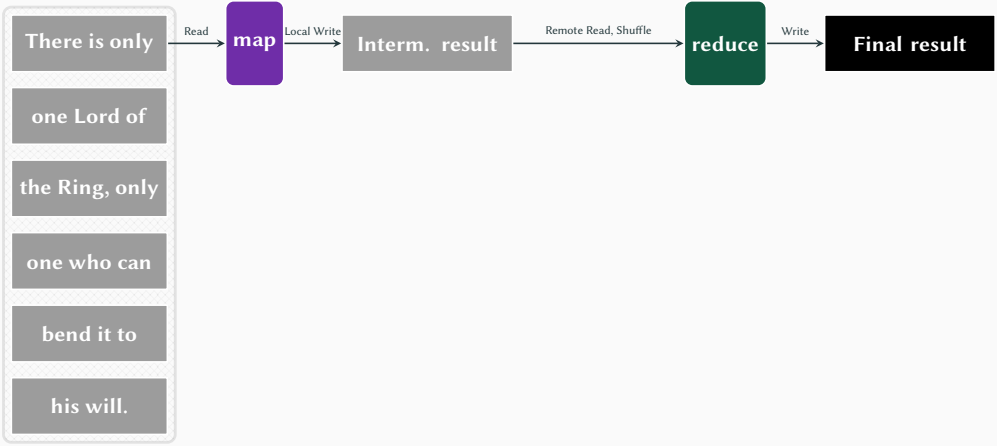
- Other systems (incl. DBSs) may be beneficial for particular problems.
- **MapReduce is stateless**, i.e., mappers/reducers are unaware of other mappers/reducers $\Rightarrow$ Not ideal for iterative algorithms.

Many parallel programming frameworks are based on the idea of MapReduce [2], e.g., Apache Hadoop, Apache Spark, Apache Flink, ...

[2] https://research.google/pubs/pub62/

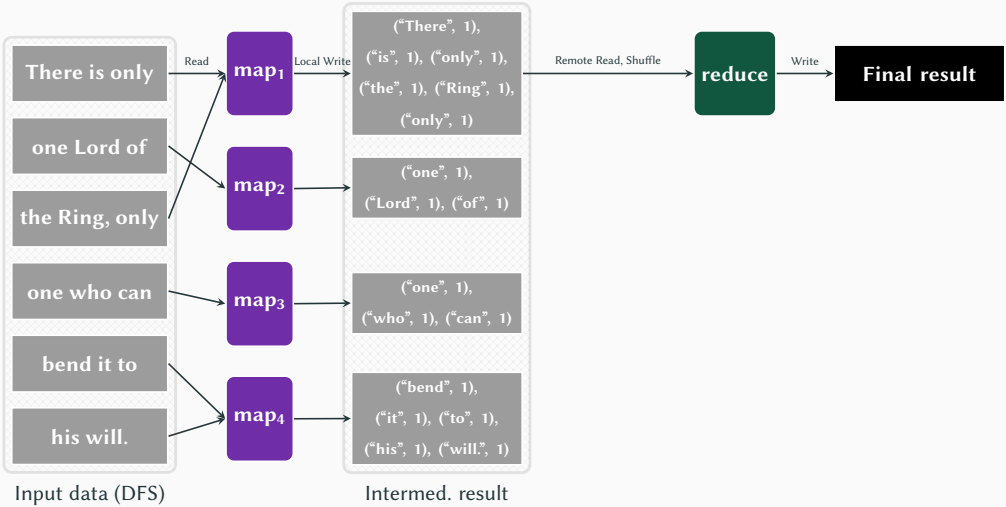
WordCount with Parallel MapReduce

WordCount with Parallel MapReduce

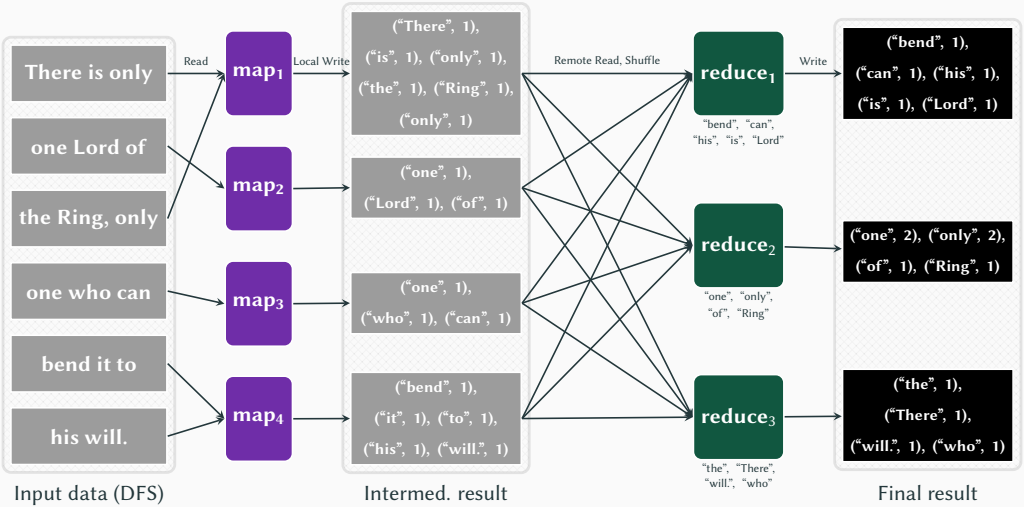


Input data (DFS)

WordCount with Parallel MapReduce



WordCount with Parallel MapReduce

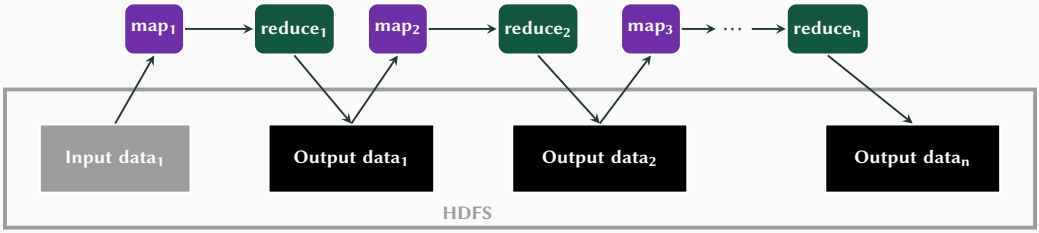


MapReduce in MongoDB

The MapReduce Framework – Caveat

Communication: Jobs run in isolation.

Shuffling large amounts of data: Performance and horizontal scalability often suffers due to communication overhead (bound by network I/O speed), e.g., two clients $\Rightarrow \frac{1}{2}$ bandwidth, four clients $\Rightarrow \frac{1}{4}$ bandwidth.



Workloads and Challenges

Batches vs. Streams – Revisited

Batch Data: A **batch** is a large but **bounded static dataset.** Before data can be processed, all data must be completely available (e.g., on hard disk).

Streaming Data: A **stream** is an **unbounded evolving dataset.** Data items are processed as they stream into the system one after another, i.e., the data does not have to be completely available.

Batch Processing

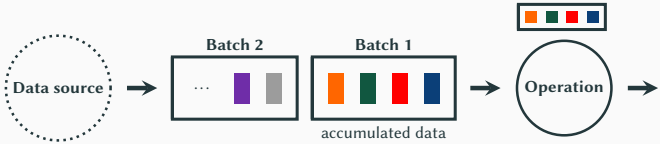
We wait until a **batch of data** (i.e., a block of data) is accumulated and then we **process the data in the batch all at once.** For example, we could analyze the data that accumulates over one hour.

Data is stored but not processed at arrival. In some scenarios, we must rely on these batches, e.g., when the “full” batch provides more insights.

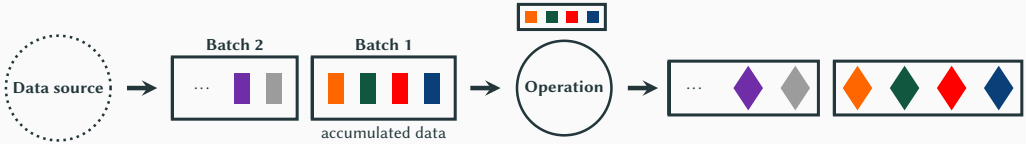
A **state is often transferred** from one batch to the next.

Batch Processing

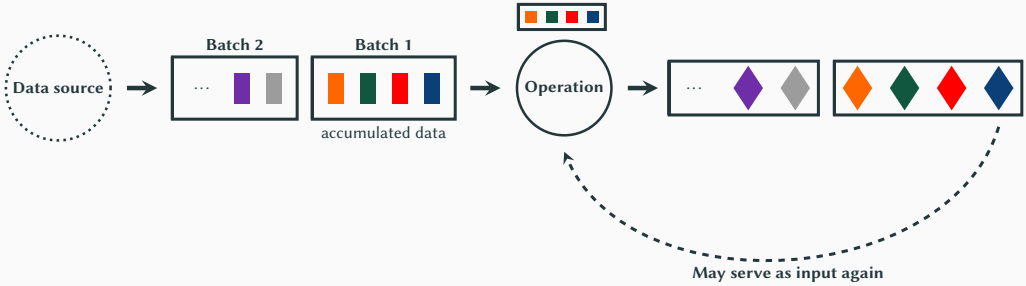
Batch Processing



Batch Processing



Batch Processing



Stream Processing

We do not wait for the data to accumulate but **process each single data item continuously** (at arrival). This allows a real-time response and typically involves simple transformations.

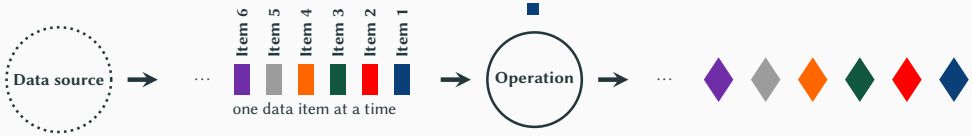
Stream processing is used **if the data naturally arrives** in a continuous **stream** (e.g., twitter) or if we build a **data-driven system** that needs to **respond quickly** (e.g., fraud detection).

Traditional stream processing is stateless, but modern systems (e.g., Apache Flink) also implement stateful stream processing.

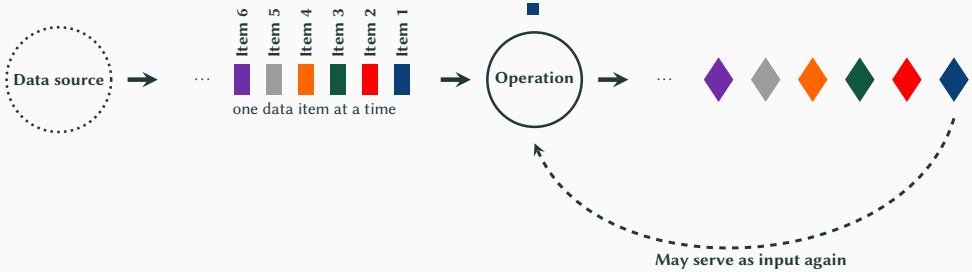
Stream Processing

Stream Processing

Stream Processing



Stream Processing



Stateless vs. Stateful

Stateless Processing: The **current operation processes** the **input data independently**, i.e., without considering preceding executions. The independence of the state makes it easier to scale.

Stateful Processing: Preceding executions may influence the outcome of the current execution, i.e., processing history is taken into account. Recording and respecting the state makes it harder to scale.

Systems Potpourri

Apache Hadoop [4]

Open-source implementation of the **MapReduce paradigm** that is designed as **batch processing system.**

- Supports a **linear data flow** but does not support iterative processing (i.e., loops).
- Is a **disk-based system** (HDFS), thus typically slower than in-memory systems.
- Scales to tens of thousands of machines (with commodity hardware).
- The **Hadoop ecosystem** [3] is quite large.

[3] https://hadoopecosystemtable.github.io/
[4] https://hadoop.apache.org/

Apache Spark [5]

Open-source **parallel processing system** that is designed as **micro-batch processing** system mainly for analytics operations.

During **computation**, the **data** is kept **in main memory (RAM)**, thus Spark is typically faster than Apache Hadoop. If the data does not fit into RAM, it falls back to disk storage (e.g., using HDFS) and provides similar performance to disk-based systems.

- Supports **iterative processing** (e.g., machine learning).
- **Generalizes MapReduce** and integrates into the Scala programming language.
- **Supports stream processing** with micro-batches (time-based windows).
- Performance heavily relies on main memory.

[5] https://spark.apache.org/

Apache Spark

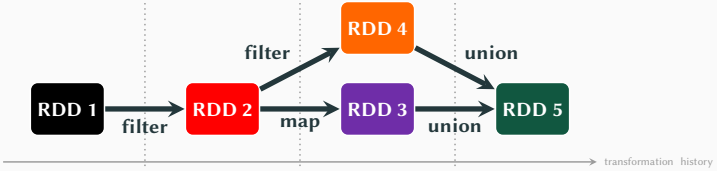
Spark implements the concept of so-called **resilient distributed datasets (RDDs).** An RDD is an immutable distributed collection of data elements that is partitioned across multiple nodes (for fault tolerance).

RDDs allow **in-memory transformations and actions.** Transformations are **applied in a lazy fashion**, i.e., they are not executed immediately but tracked in a **lineage graph.** This improves performance and implements the fault tolerance.

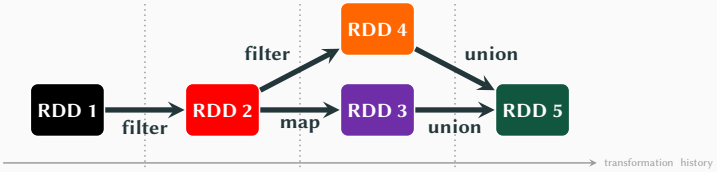
For the interested reader, we refer to the official publications on Apache Spark [6] [7].

[6] https://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf
[7] https://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

Lineage Graph: A graph that encodes how an RDD was derived (usually from stable storage), e.g., RDD 2 was derived from RDD 1 (which may represent some input file).

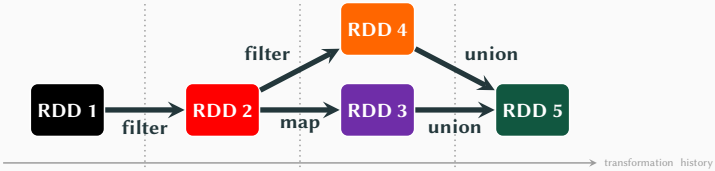


Lineage Graph: A graph that encodes how an RDD was derived (usually from stable storage), e.g., RDD 2 was derived from RDD 1 (which may represent some input file).



Lazy Evaluation: Only actions trigger execution, transformations are recorded in the lineage graph ⇒ More potential for optimizations (all transformations are known).

Lineage Graph: A graph that encodes how an RDD was derived (usually from stable storage), e.g., RDD 2 was derived from RDD 1 (which may represent some input file).



Lazy Evaluation: Only actions trigger execution, transformations are recorded in the lineage graph ⇒ More potential for optimizations (all transformations are known).

Fault Tolerance: Lost RDDs can be recomputed from other RDDs using the lineage graph, i.e., lost data is recovered without replication.

Apache Spark

For a **table-like abstraction**, Spark implements **dataframes.** A dataframe is an immutable distributed collection of data elements (like RDDs), but the **data is organized in columns** (RDDs store unstructured data).

```
df = spark.read.json("members.json")
df.show()
# Prints the "schema", i.e., the types of keys + values
df.printSchema()
```

```
+-------------+-----------+----------+
|name         |employment |department|
+-------------+-----------+----------+
|Daniel Kocher|Postdoc.   |CS        |
|Thomas Hütter|Postdoc.   |CS        |
|Tijn De Vos  |PhD Student|CS        |
|...          |...        |...       |
+-------------+-----------+----------+
```

```
root
 |-- name: string (nullable = true)
 |-- employment: string (nullable = true)
 |-- department: string (nullable = true)
```

Apache Spark – Lazy Evaluation

```
# Lazy evaluation: .filter(.) and .select(.) are tracked in the lineage graph
dffil = df.filter($"employment" == "PhD Student")
dfsel = dffil.select($"name",$"department")

# Only .show() triggers actual execution.
dfsel.show()
```

```
+-------------+--------+
|name         |department|
+-------------+----------+
|Tijn De Vos|CS         |
+-------------+----------+
```

Apache Kafka [9]

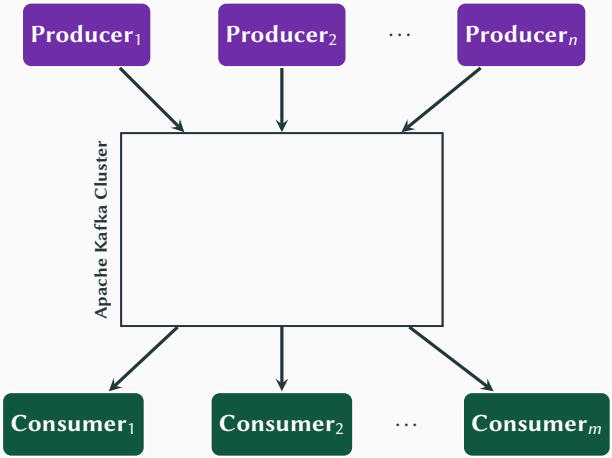
Initially developed by LinkedIn, Apache Kafka [8] is a **powerful building block** in many large-scale **data processing pipelines.** At its core, it is a **distributed and fault-tolerant logging system** (distributed transaction log).

The internals are comparable to the logging mechanism in a database system (i.e., log entries are stored in order in an append-only fashion).

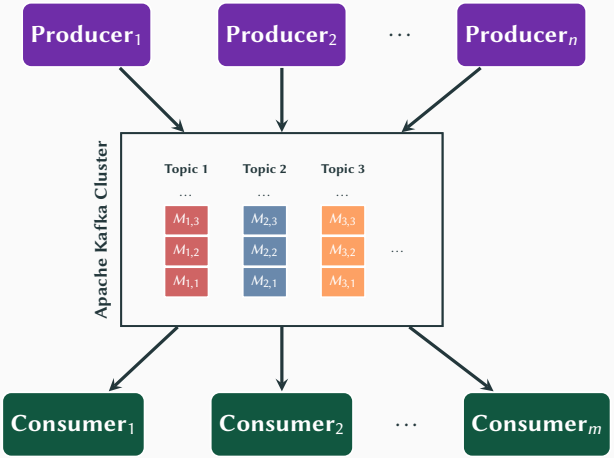
Producer-Consumer Paradigm: *Producer* applications send (produce) messages to a Kafka node. Messages are stored by "topic" and *consumer* applications subscribe to topics to retrieve (consume) the messages from the queues they subscribed to.

[8] https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf
[9] https://kafka.apache.org/

Apache Kafka

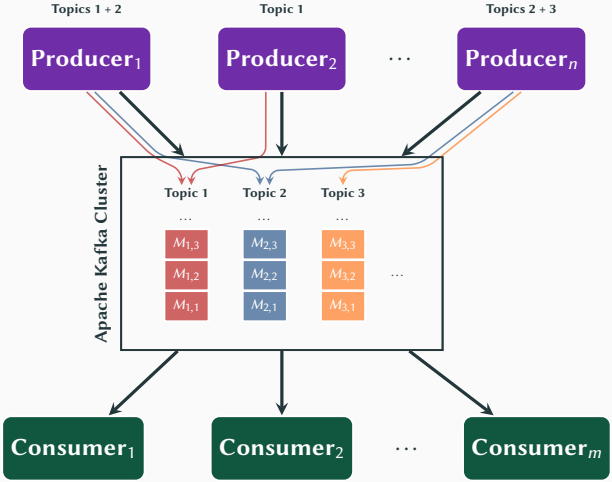


Apache Kafka



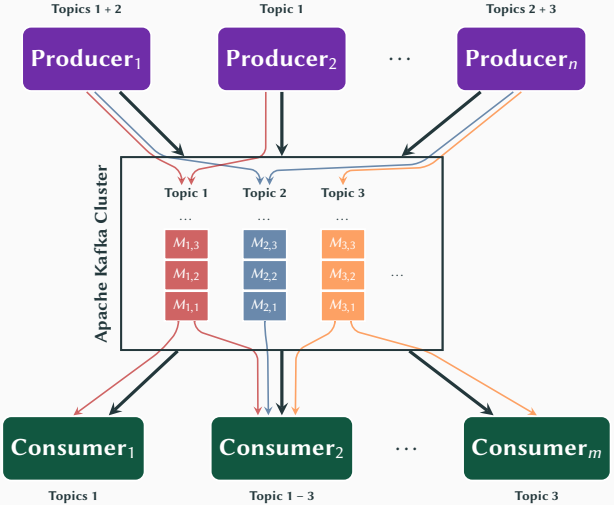
$M_{i,j}$ ...j-th message of topic i

Apache Kafka



$M_{i,j}$ …j-th message of topic i

Apache Kafka



$M_{i,j}$ …j-th message of topic i

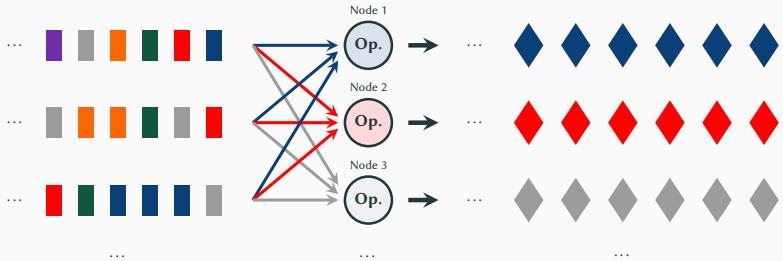
Apache Flink [10]

Open-source **parallel processing system** that is designed as **native stream processing system.**

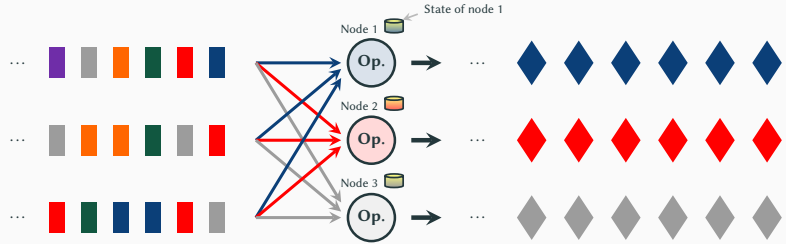
- The streaming architecture supports **iterative processing** (e.g., machine learning).
- Unified framework for processing batches and streams.
- Can operate in a **stateful or stateless** computation mode.
- Implements fault tolerance through checkpoints/snapshots.

[10] https://flink.apache.org/

Distributed Stream Processing: Data items in the streams are grouped and distributed based on some key (cf. colors), and each node is responsible for some key range.



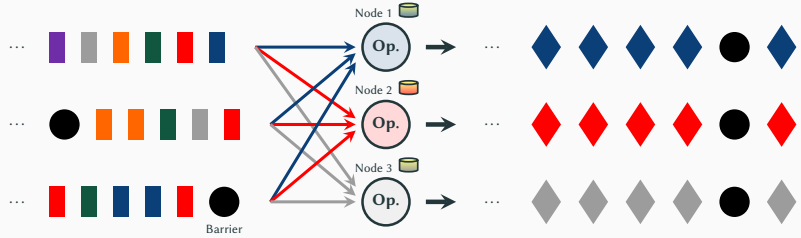
***Stateful* Distributed Stream Processing** [11]**:** The **state** is **accumulated** and **maintained over time** in a distributed manner by **co-locating it** (i.e., storing it on the node that runs the operation).



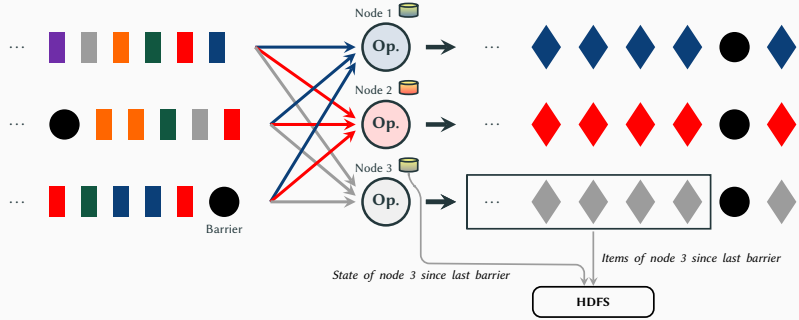
[11] This is a very simplified description. For a detailed description, please check https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html

Apache Flink

Fault Tolerance [12]**:** Special items called **barriers** are injected into the streams and force the nodes to **write a checkpoint of data and state** onto (distributed) durable storage (e.g., HDFS). **Node *i* records** its data and state since the **last barrier** was processed.



Fault Tolerance [12]: Special items called **barriers** are injected into the streams and force the nodes to **write a checkpoint of data and state** onto (distributed) durable storage (e.g., HDFS). **Node *i* records** its data and state since the **last barrier** was processed.



[12] This is a very simplified description. For a detailed description, please check
https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/stream_checkpointing.html

Apache SystemDS [15]

Formerly known as SystemML (developed by IBM). Apache SystemDS is a **distributed machine-learning (ML) system** that scales to large clusters. Its focus is on the **integration of the entire data science lifecycle** (i.e., data integration/cleaning/preparation, ML model training, serving the data).

SystemDS [13] [14] bridges the gap from simple ML algorithms written in R/Python to executing the ML algorithm at scale on a large cluster. It provides a **declarative language for ML** and can execute in-memory on a single machine or on a large Spark cluster.

[13] https://www.youtube.com/watch?v=n3JJP6UbH6Q
[14] http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf
[15] https://systemds.apache.org/