

CLOUD COMPUTING SYSTEMS

Lectures 6-7

Nuno Preguiça

(nuno.preguica_at_fct.unl.pt)

OUTLINE

Computing services

1. First generation batch processing: Map-reduce
2. Second generation batch processing: Spark
3. Stream processing

OUTLINE

Computing services

1. **First generation batch processing: Map-reduce**
 - **Programming model**
 - Execution model
 - Handling faults
2. Second generation batch processing: Spark
3. Stream processing

MAPREDUCE

“A new abstraction that allows us to express **simple computations** we were trying to perform but **hides the messy details** of parallelization, fault-tolerance, data-distribution and load-balancing in a library”

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

MAPREDUCE (2)

“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.”

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

EXAMPLE APPLICATION

Consider you have a huge text.

Goal: find out the words that appear more frequently in a text.

Can be transformed into:

Goal 1: Count the number of times each word appears in the text.

Goal 2: Order the words by frequency.

Is this a useless example?

Not really... e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

MAPREDUCE: OVERVIEW

Sequentially read a lot of data

Map phase:

- Extract the important information

Group by key: Sort and Shuffle the output of the map phase

Reduce phase:

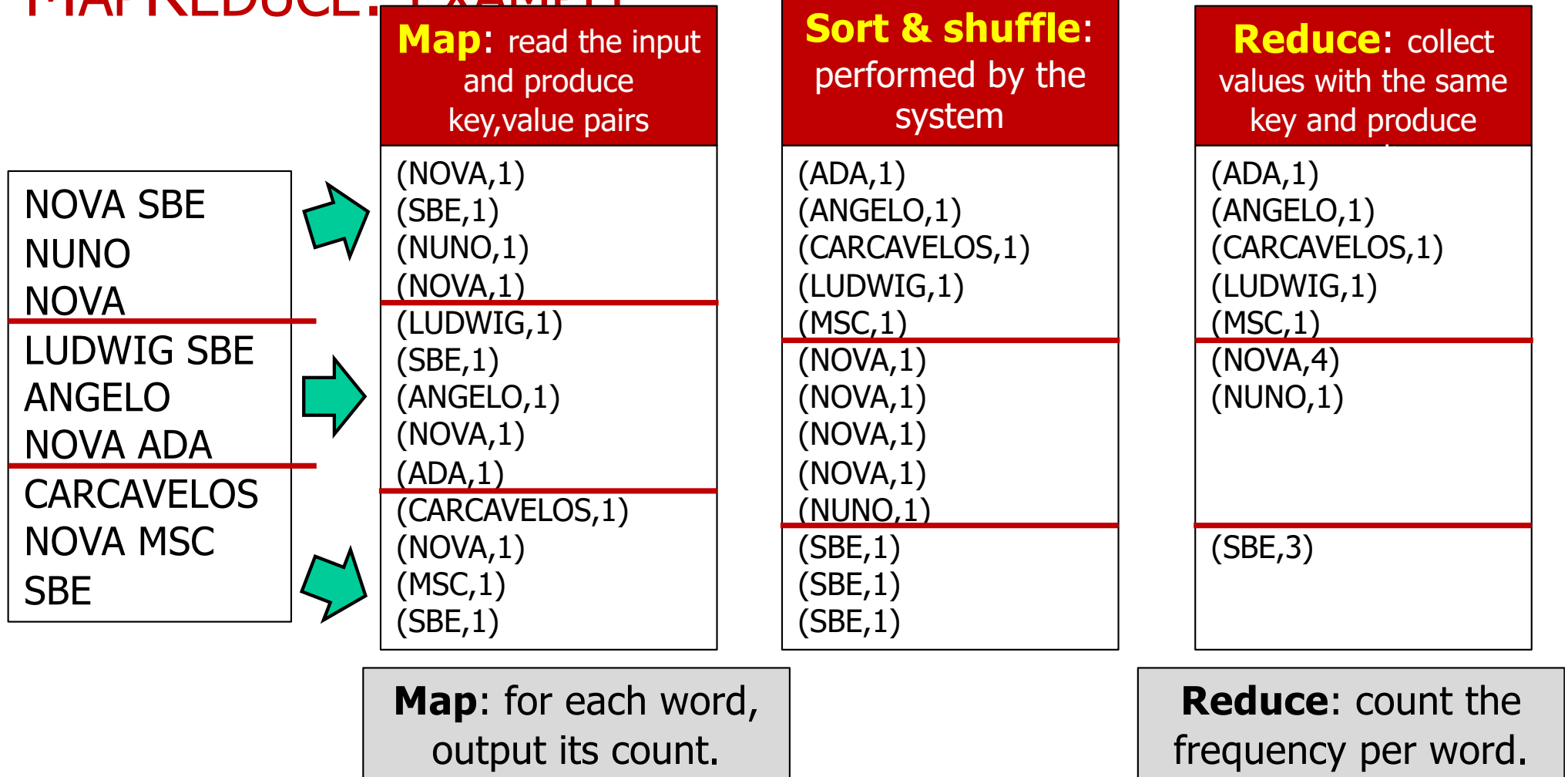
- Aggregate, summarize, filter or transform

Write the result

Each computation step is composed of a map and a reduce steps.

A computation is a sequence of map-reduce computations.

MAPREDUCE: EXAMPLE



WORD COUNT USING MAPREDUCE

```
map(key, value): // key: document name; value: text of the document
    for each word w in value:
        emit(w, 1)
```

```
reduce(key, values): // key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
```

MAPREDUCE MODEL

Input: a set of key-value pairs

Programmer specifies two methods:

- **Map(k, v)** $\rightarrow \langle k', v' \rangle^*$
 - Takes a key-value pair and outputs a set of key-value pairs
 - E.g., key is the filename, value is a single line in the file
 - Map is called for every (k, v) pair
- **Reduce($k', \langle v' \rangle^*$)** $\rightarrow \langle k', v'' \rangle^*$
 - All values v' with same key k' are reduced together and processed in v' order
 - Reduce is called for each unique key k'

EXAMPLE APPLICATION

Consider you have a huge text.

Goal: find out the words that appear more frequently in a text.

Can be transformed into:

Goal 1: Count the number of times each word appears in the text.

Goal 2: Order the words by frequency.

Is this a useless example?

Not really... e.g. analyze web logs to find popular URLs, analyze social media posts to find trending topics, etc.

GOAL 2: ORDER THE WORDS BY FREQUENCY.

Can we sort the values of the reduce before returning them?

NO !!!

Each reduce will be processed independently (by a different machine).

Also bad idea because it requires storing potentially large amount of data.

Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)

(NOVA,4)
(NUNO,1)

(SBE,3)

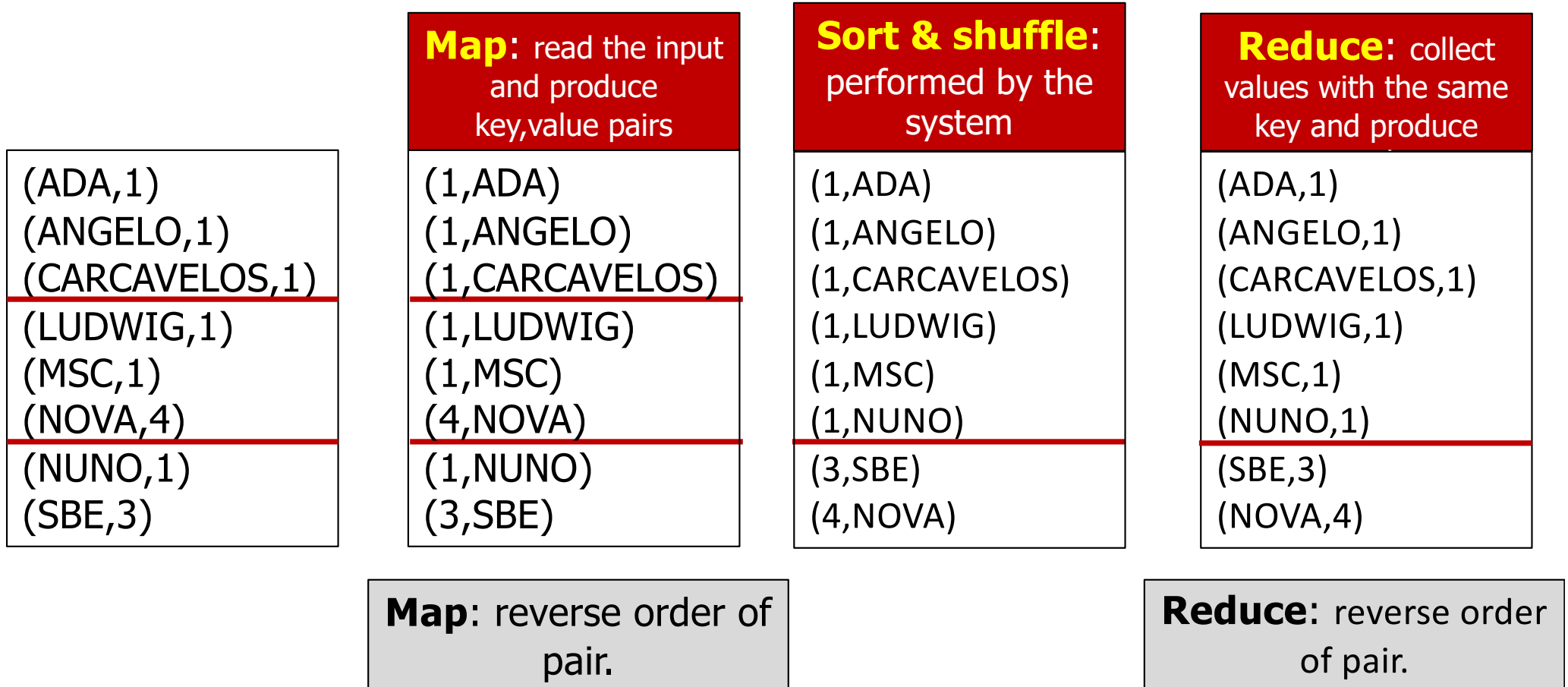
Reduce: collect values with the same key and produce

(ADA,1)
(ANGELO,1)
(CARCAVELOS,1)
(LUDWIG,1)
(MSC,1)

(NUNO,1)
(NOVA,4)

(SBE,3)

MAPREDUCE: EXAMPLE



WORD COUNT SORT USING MAPREDUCE

```
map(key, value): // key: word; value: word count  
    emit(value, key)
```

```
reduce(key, values): // key: word count; value: word  
    for each v in values:  
        emit(v, key)
```

MAPREDUCE

Programmer responsible for:

- **Map** function
- **Reduce** function

MapReduce system responsible for:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **sort by key & shuffle** step
- Handling machine **failures**
- Managing required inter-machine **communication**

OUTLINE

Computing services

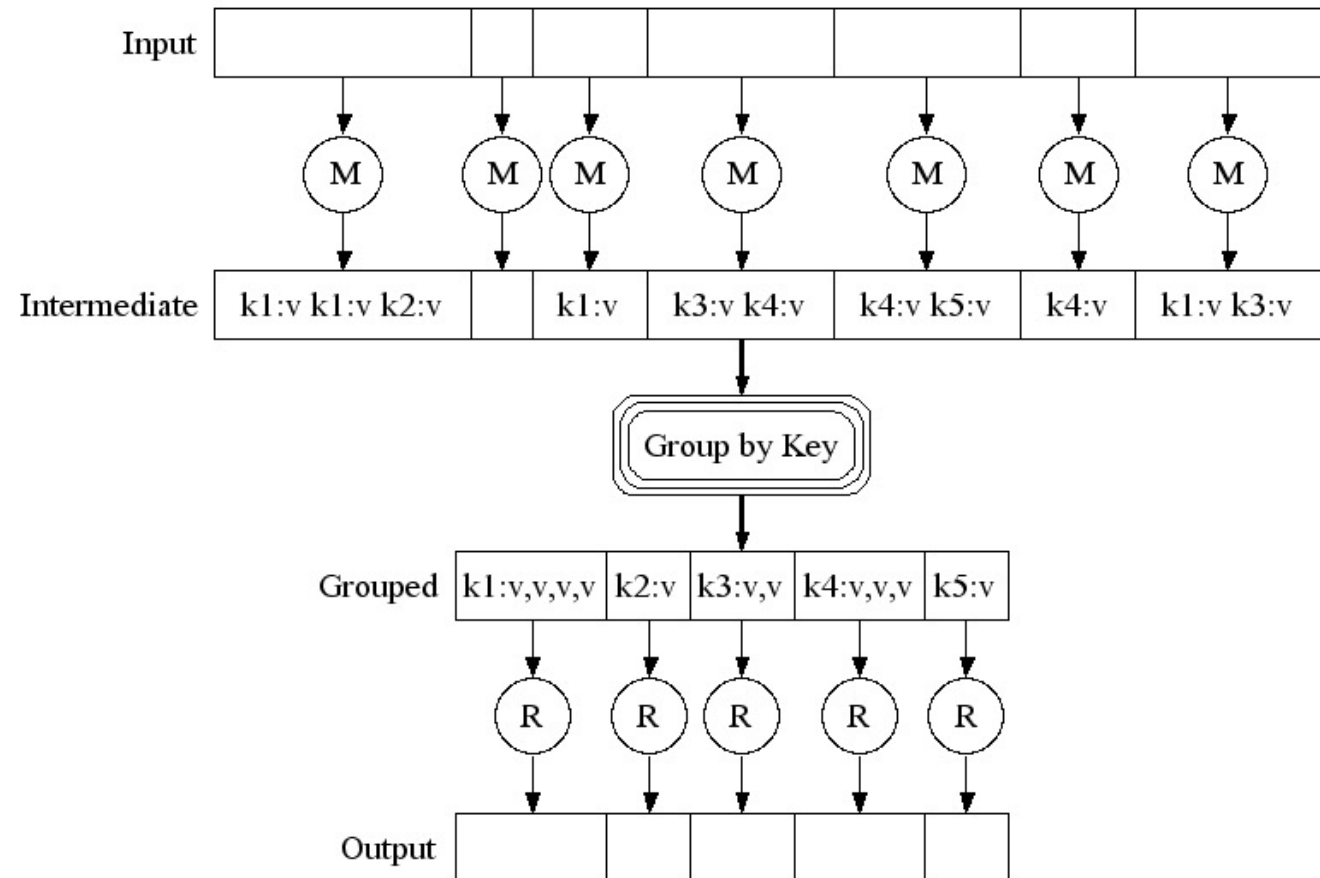
1. **First generation batch processing: Map-reduce**
 - Programming model
 - **Execution model**
 - **Handling faults**
2. Second generation batch processing: Spark
3. Stream processing

MAPREDUCE: LOGICAL EXECUTION...

Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

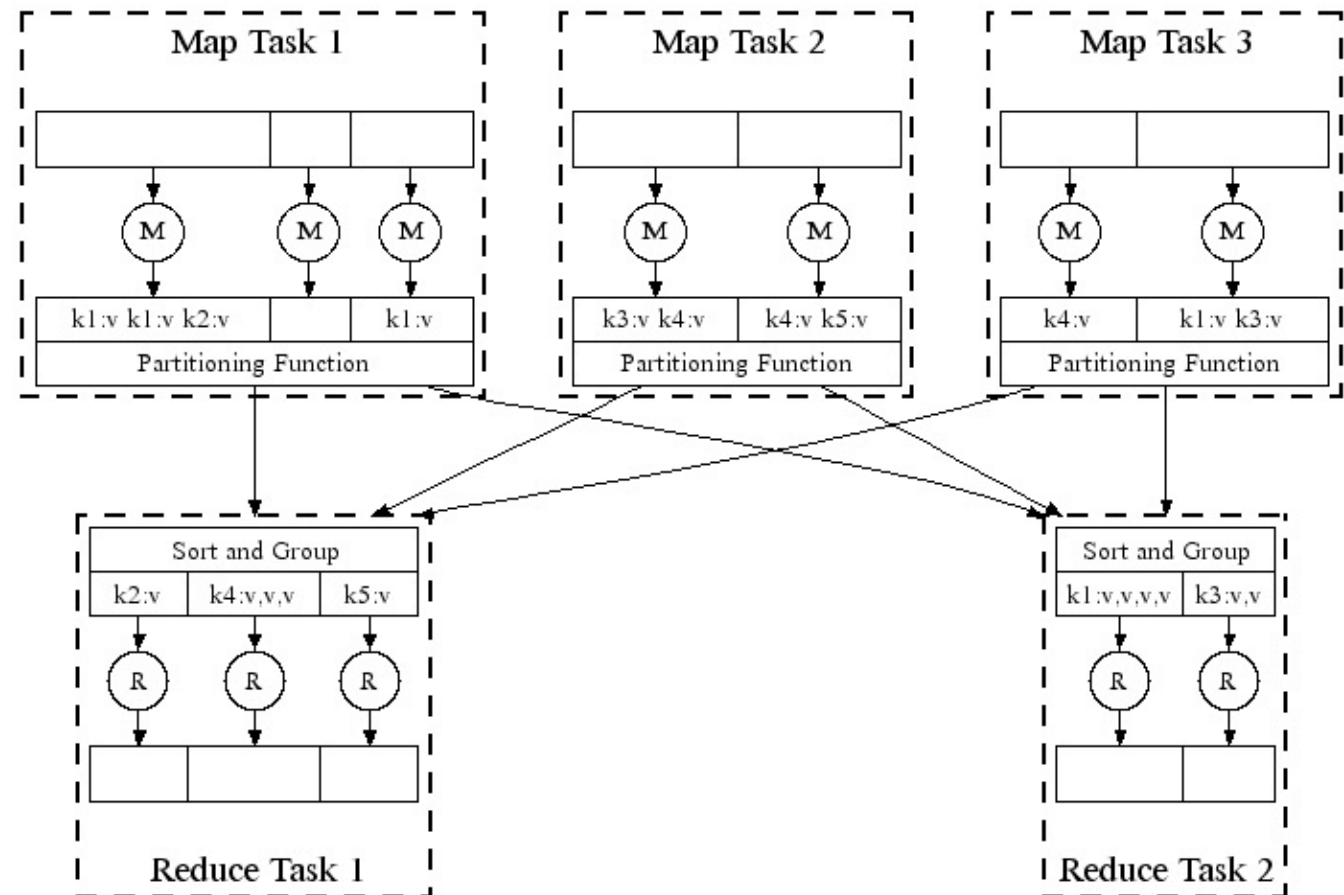


MAPREDUCE: DISTRIBUTED EXECUTION...

Map: read the input and produce key,value pairs

Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce

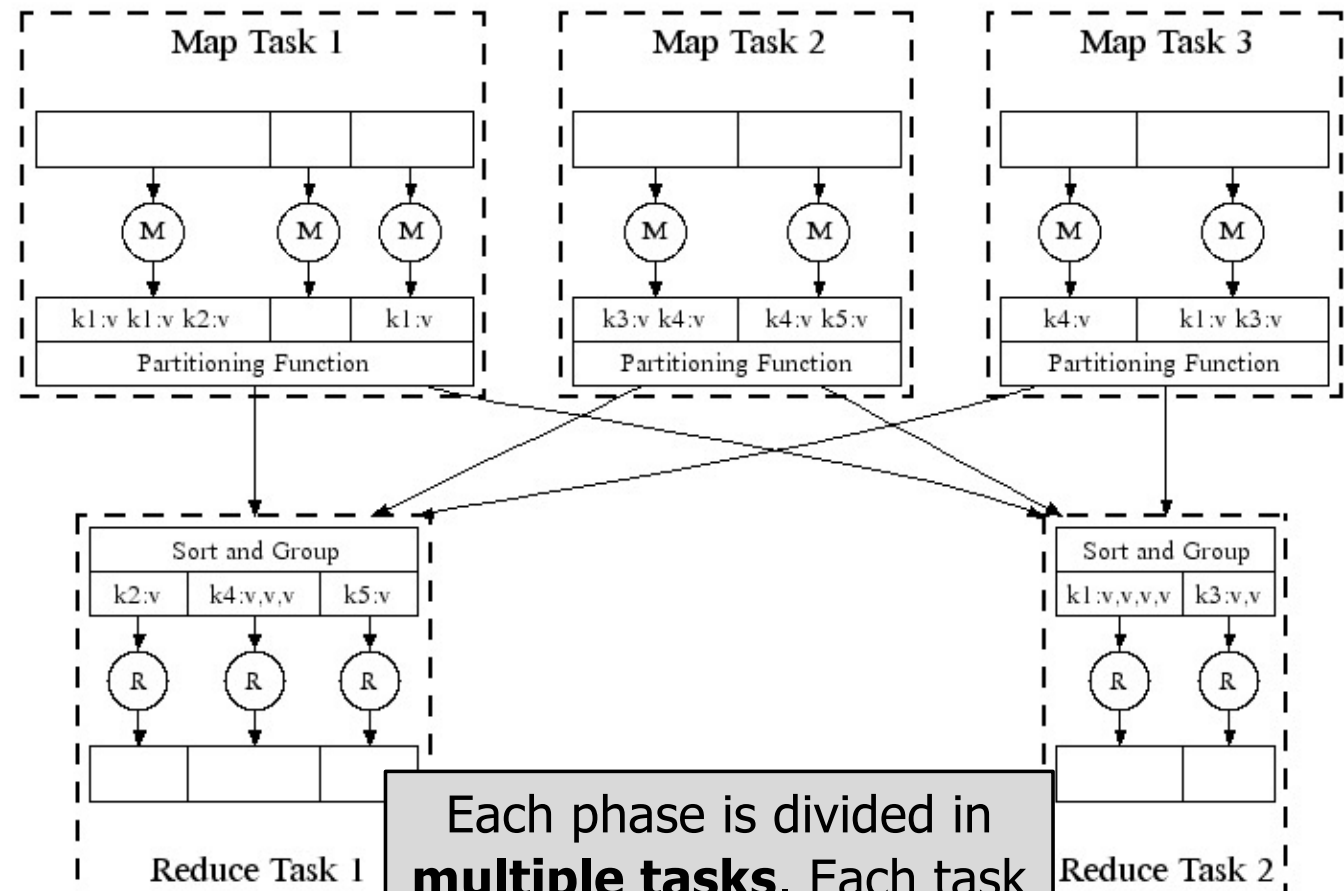


MAPREDUCE: DISTRIBUTED EXECUTION...

Map: read the input and produce key,value pairs

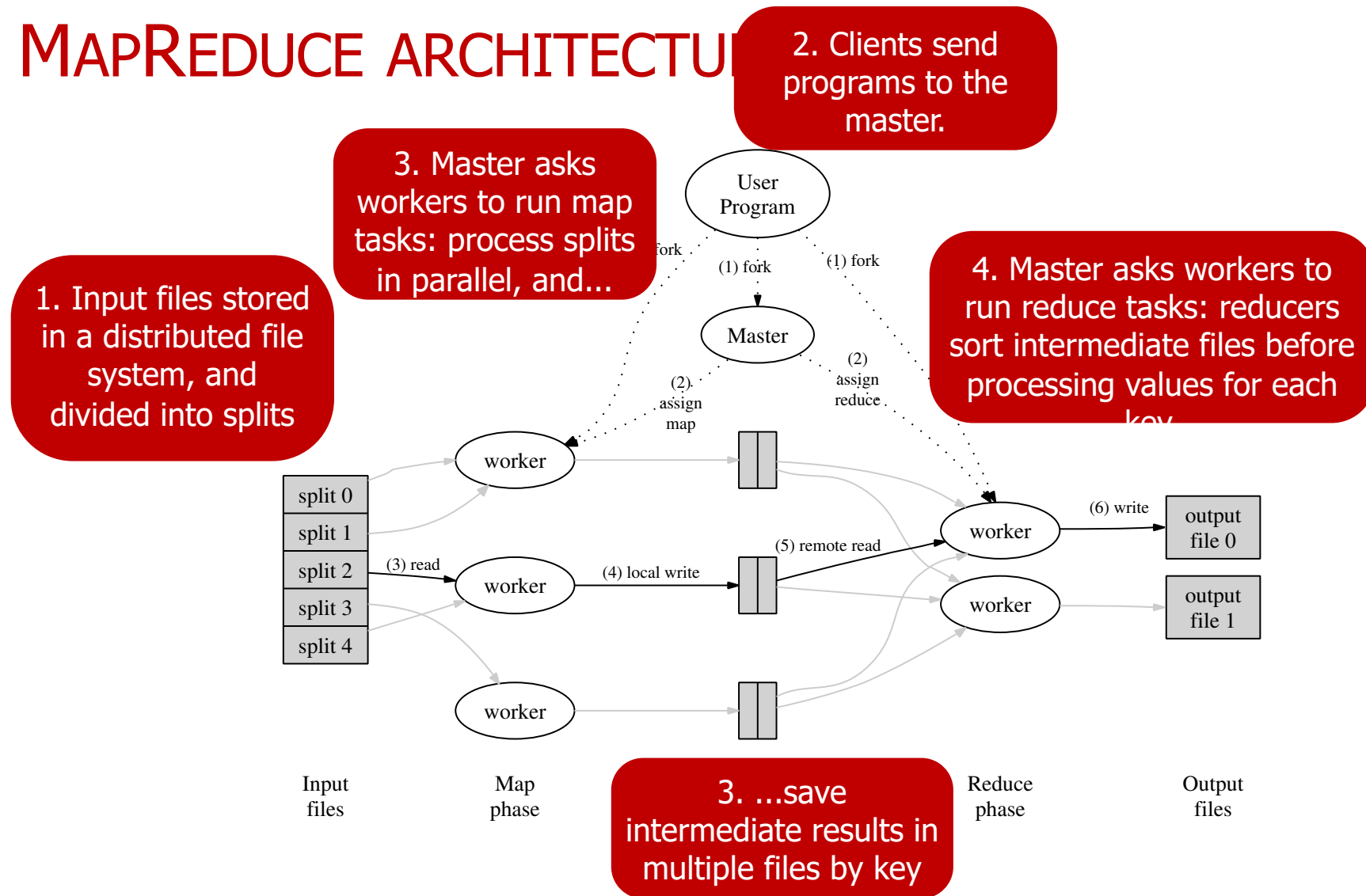
Sort & shuffle: performed by the system

Reduce: collect values with the same key and produce



Each phase is divided in **multiple tasks**. Each task is executed independently on a **different nodes**.

MAPREDUCE ARCHITECTURE



MAPREDUCE SYSTEM: MASTER NODE

Master node coordinates the execution:

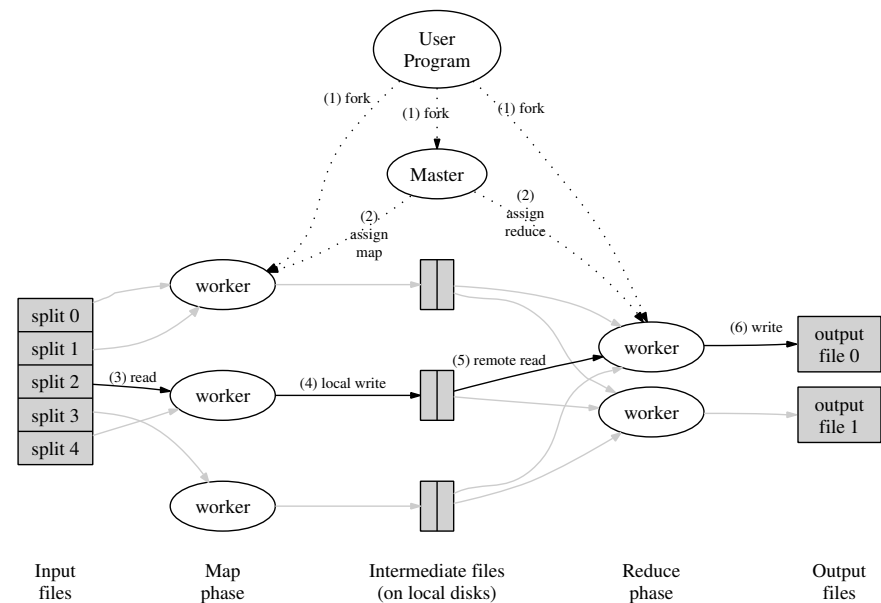
Task status: (idle, in-progress, completed)

Idle tasks get scheduled as workers become available

When a map task completes, it sends the master the location and sizes of its intermediate files, one for each reducer

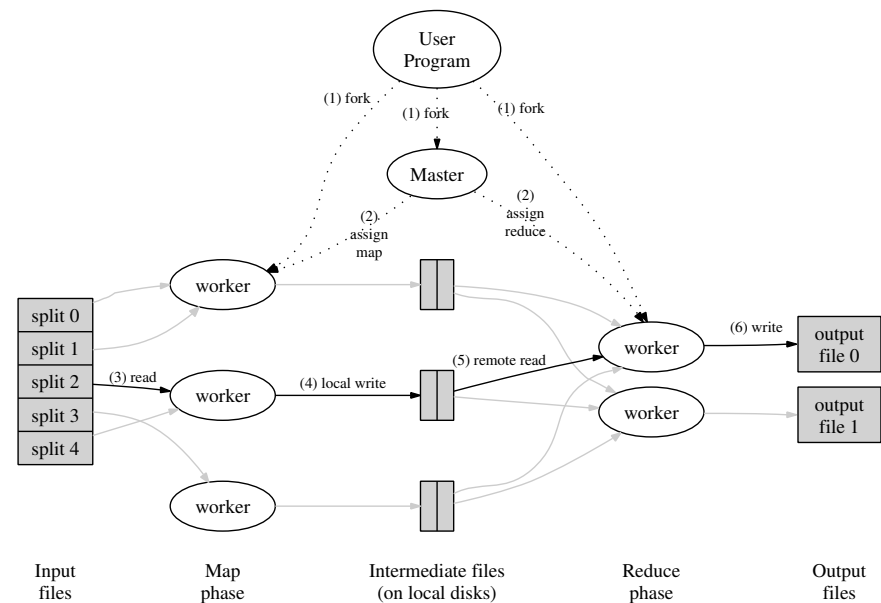
Master pushes this info to reducers

Master pings workers periodically to detect failures



MAPREDUCE SYSTEM: WORKER

Worker node performs map or reduce tasks, as requested by the coordinator.



MAPREDUCE SYSTEM: HANDLING FAULTS

Map worker failure

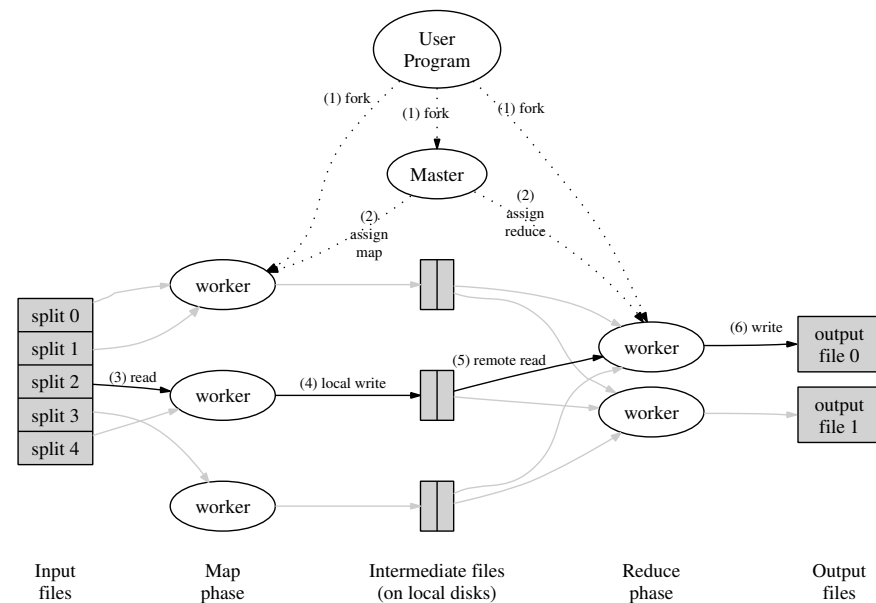
Upon detection of the failure of a worker, map tasks restarted in different worker

Reduce worker failure

Reduce task is restarted in other worker

Stragglers (slow workers)

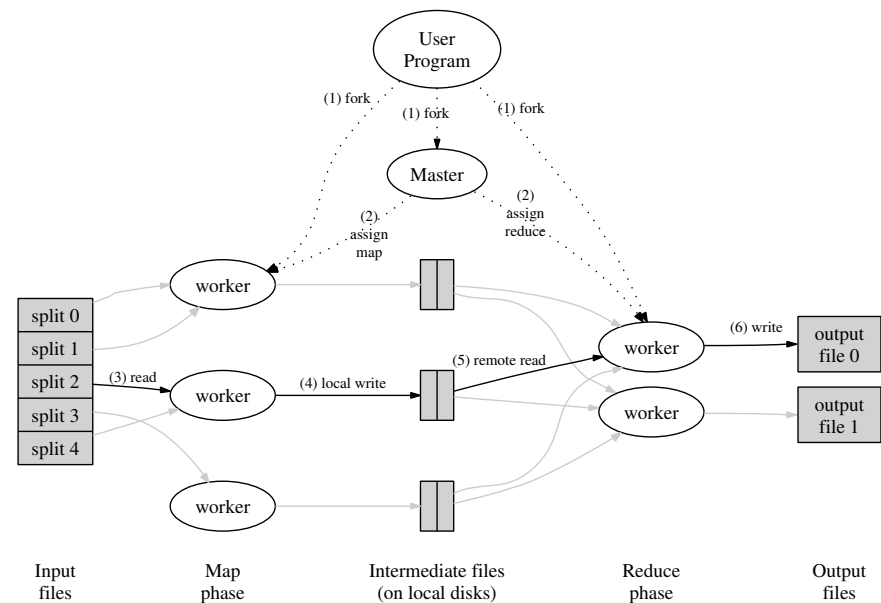
If a task is taking too long to complete, it is launched in other worker. First result used.



MAPREDUCE SYSTEM: HANDLING FAULTS (2)

Master failure

MapReduce task is aborted
and client is notified

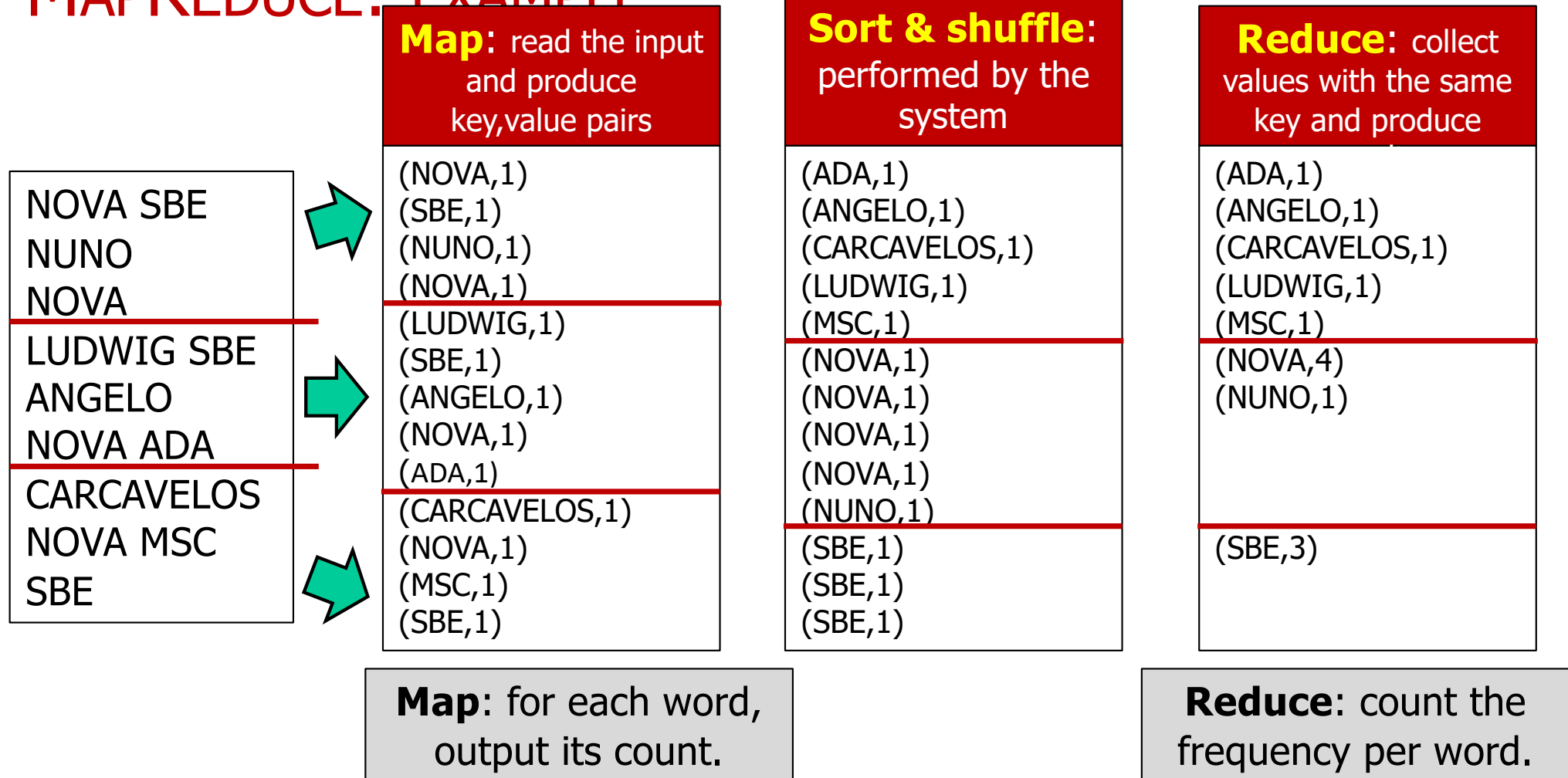


OUTLINE

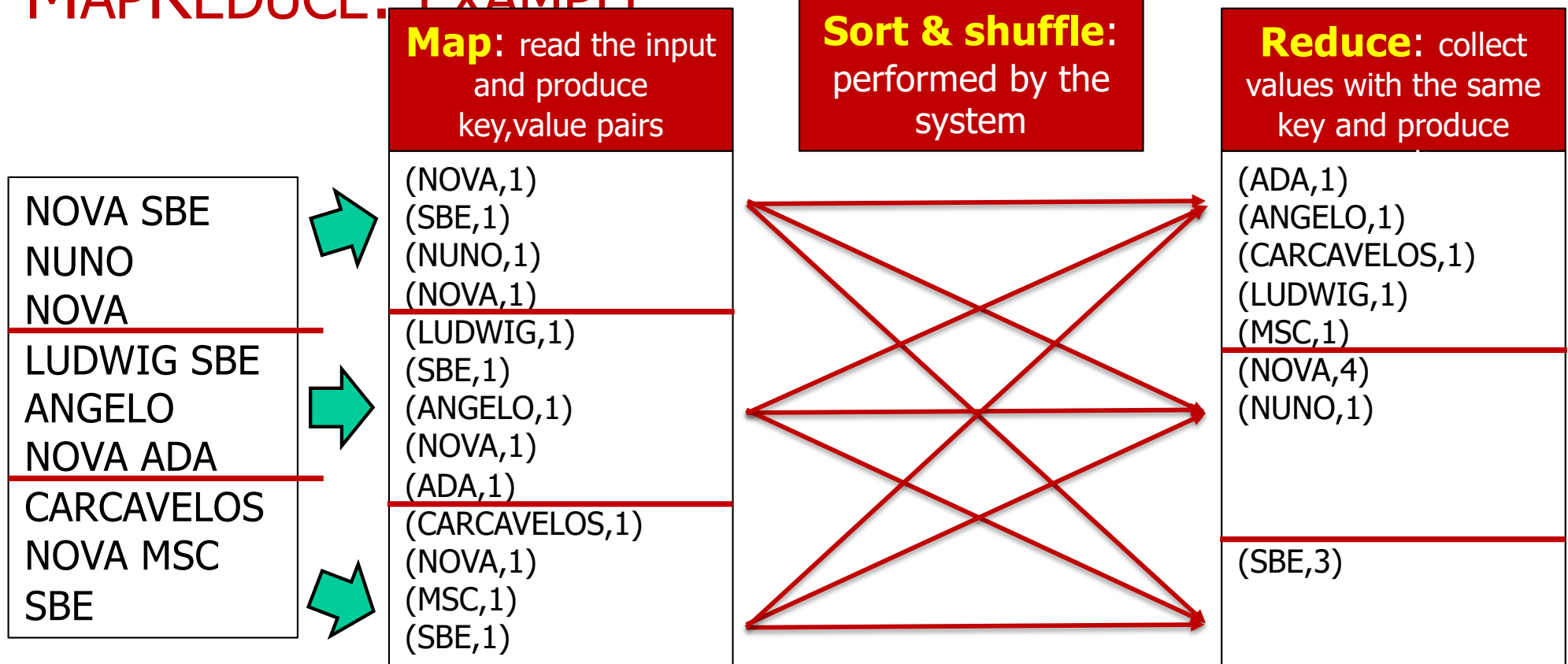
Computing services

1. **First generation batch processing: Map-reduce**
 - **Programming model**
 - Execution model
 - Handling faults
2. Second generation batch processing: Spark
3. Stream processing

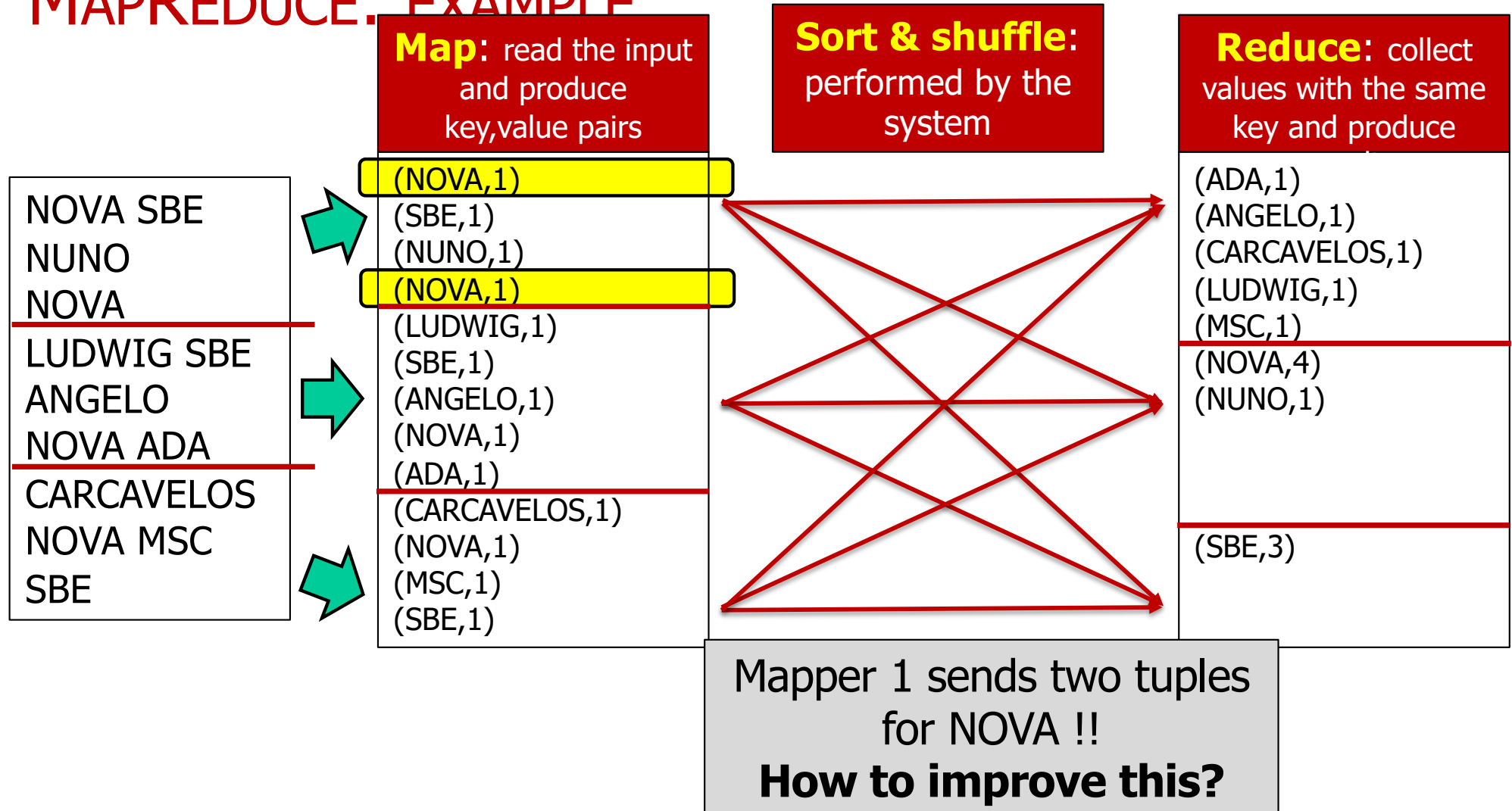
MAPREDUCE: EXAMPLE



MAPREDUCE: EXAMPLE



MAPREDUCE: EXAMPLE



IMPROVING MAPREDUCE: COMBINER

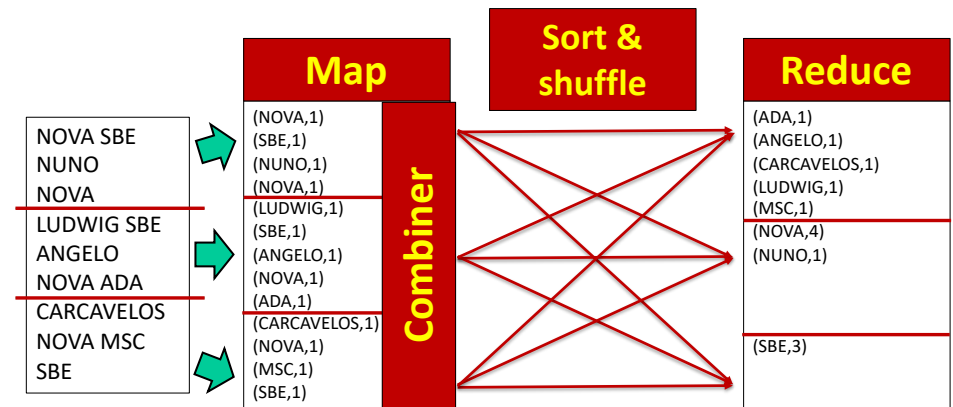
Combiner allows to pre-aggregate values in the mapper.

Combine(k, <v>*) \rightarrow <k, v'>

All values v with same key k are combined and processed in v order

Combine is called at each mapper for each unique key k

Combiner function is usually the same as the reduce function.



WHY WAS MAP-REDUCE SO POPULAR?

Distributed computation before MapReduce:

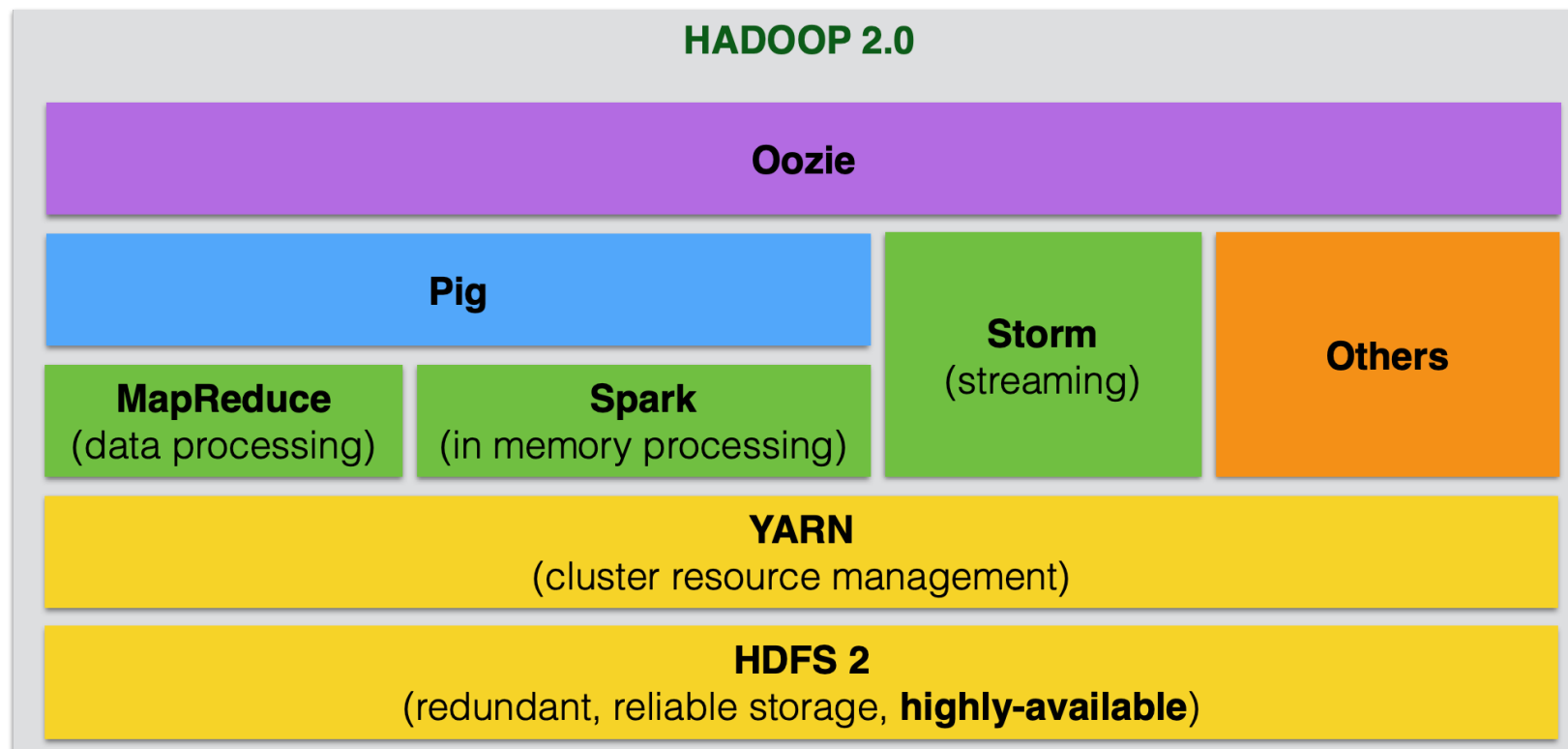
- how to divide the workload among multiple machines?
- how to distribute data and program to other machines?
- how to schedule tasks?
- what happens if a task fails while running?
- ... and ... and ...

Distributed computation after MapReduce

- how to write Map function?
- how to write Reduce function?
- systems to efficiently execute map-reduce jobs: Hadoop.

APACHE HADOOP 2.0

Multi Purpose
(batch + streaming, etc.)



OUTLINE

Computing services

1. First generation batch processing: Map-reduce
2. **Second generation batch processing: Spark**
3. Stream processing

MAPREDUCE: CHAINING PROGRAMS

MapReduce requires complex computations to be split into successive MapReduce jobs

These complex programs can experience **high latency** due to several factors, including:

- need to **read and write files**
- underlying filesystem **replication** (for writes)
- one job must finish before the next can be started...

Apache Spark tackles these limitations.

APACHE SPARK

Apache Spark provides in-memory, fault-tolerant distributed processing.

Key ideas:

- Spark programs comprise **multiple chained data transformations**, using a high-level functional programming model;
- Spark defines a distributed collection data-structure : **Resilient Distributed Dataset** (RDD).

DATA MODEL AND APIS

RDDs are immutable data

- logically a RDD is an **immutable collection of data tuples**;
- **physically distributed** (partitioned) across many nodes;
- upon a failure (or cascade of failures), RDDs can be **recreated** automatically and efficiently **from the dependencies**.

Spark Dataframes

- DataFrames are distributed collections of data that is grouped into named columns.
- DataFrames can be seen as RDDs with a schema that names the fields of the underlying tuples.

Spark SQL

- SQL for specifying computations

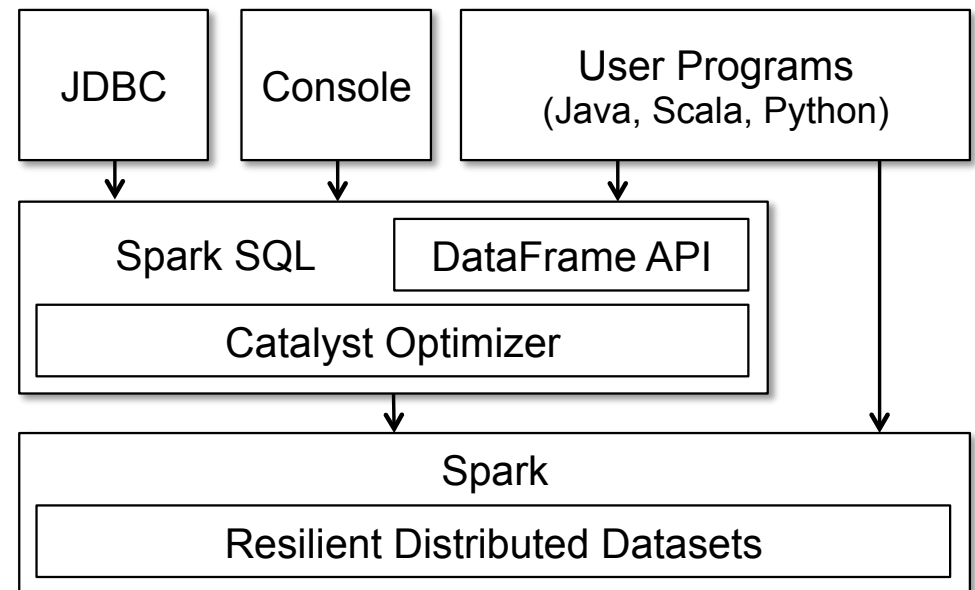
SPARKSQL ARCHITECTURE

Programs using SQL/DataFrames are **translated** into Spark programs.

Programs are **optimized** to execute efficiently.

Based on the techniques used in database systems.

Libraries for advanced analytics algorithms such as **graph processing** and **machine learning**.



FIRST EXAMPLE

SparkSession.builder. ...

- A SparkSession represents the entry point to submit programs to a Spark cluster.
- `master("local")` : defines where the master Spark node is located – local means running on local mode, i.e., not connected to a cluster.

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder \
    .master("local") \
    .appName("Simple test") \
    .getOrCreate()
```

```
try:
    df = spark.read.text("doc.txt")

    df.printSchema()
    df.show()
finally:
    spark.stop()
```

FIRST EXAMPLE (2)

spark.stop()

- Shutdown the underlying SparkContext.
- You should stop a SparkContext in the end, as only a single SparkContext may exist – we are doing this in a finally clause to guarantee this.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local") \
    .appName("Simple test") \
    .getOrCreate()

try:
    df = spark.read.text("doc.txt")

    df.printSchema()
    df.show()
finally:
    spark.stop()
```

FIRST EXAMPLE: CREATING DATAFRAME FROM TEXT FILE

dataframe = spark.read.text(filename)

- Creates a Dataframe from a text file. The Dataframe includes a single column named "value", and each line is a row of the DataFrame.

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder \
    .master("local") \
    .appName("Simple test") \
    .getOrCreate()
```

```
try:
```

```
    df = spark.read.text("doc.txt")
```

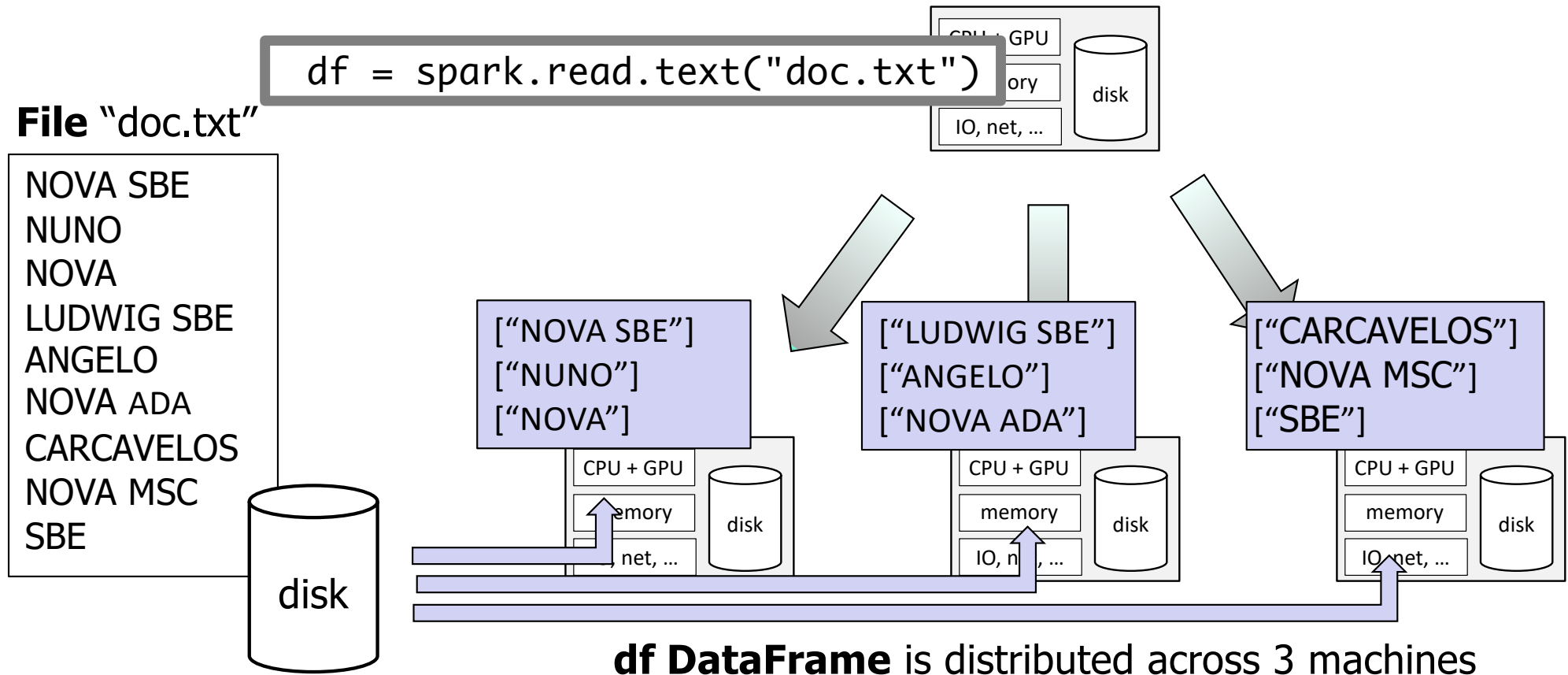
```
    df.printSchema()
```

```
    df.show()
```

```
finally:
```

```
    spark.stop()
```

FIRST EXAMPLE: DISTRIBUTED EXECUTION



FIRST EXAMPLE: CREATING DATAFRAME FROM TEXT FILE

dataframe.show()

- Displays the contents of the DataFrame.
- To show the values of a DataFrame, it is necessary to collect them – remember that a DataFrame might be distributed over multiple machines, and your program is running in a single machine.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local") \
    .appName("Simple test") \
    .getOrCreate()

try:
    df = spark.read.text("doc.txt")

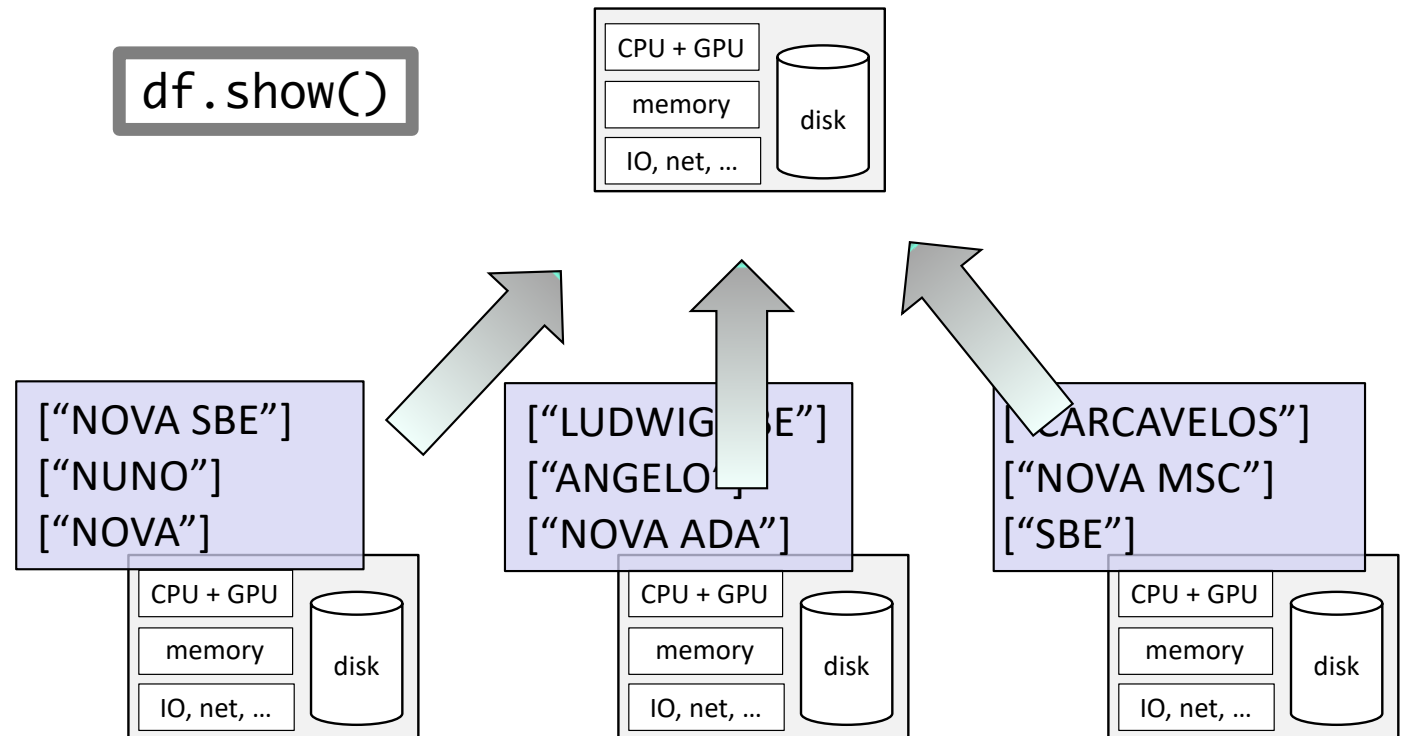
    df.printSchema()
    df.show()
finally:
    spark.stop()
```

FIRST EXAMPLE: DISTRIBUTED EXECUTION

Value of
variable **res**

```
["NOVA SBE",  
"NUNO",  
"NOVA",  
"LUDWIG SBE",  
"ANGELO",  
"NOVA ADA",  
"CARCAVELOS",  
"NOVA MSC",  
"SBE"]
```

```
df.show()
```



df DataFrame is distributed across 3 machines

PROGRAMMING MODEL

Spark Dataframe programs describe the flow of transformations that creates a DataFrame from another, usually in several steps.

Spark programs, encode the dependencies among the various DataFrames (and underlying RDDs):

- this is known as the **lineage graph**

SECOND EXAMPLE

Count the number of occurrences of each word and print those that appear more than once.

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))
result = df2.groupBy(df2.word) \
    .count() \
    .where(col("count") > 1)
```

SECOND EXAMPLE: EXPLODE + SPLIT

split(column, delimiter)

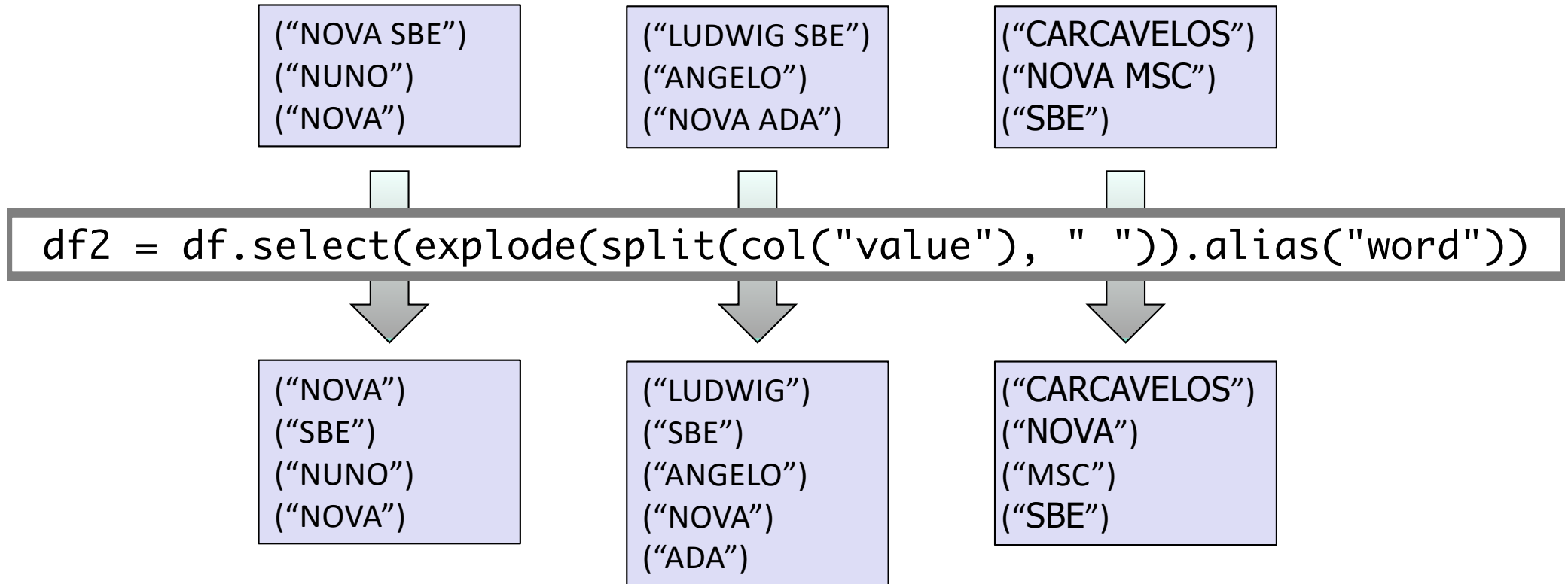
- Divides the value of the column by delimiter, creating an array of values

explode(column).alias(name)

- Flattens the array, making each value an independent row, with name the result column.

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))
result = df2.groupBy(df2.word) \
    .count() \
    .where(col("count") > 1)
```

SECOND EXAMPLE: FLATMAP



SECOND EXAMPLE: GROUPBY

groupBy(column)

- Groups the rows using the value of the given column

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))  
result = df2.groupBy(df2.word) \  
              .count() \  
              .where(col("count") > 1)
```

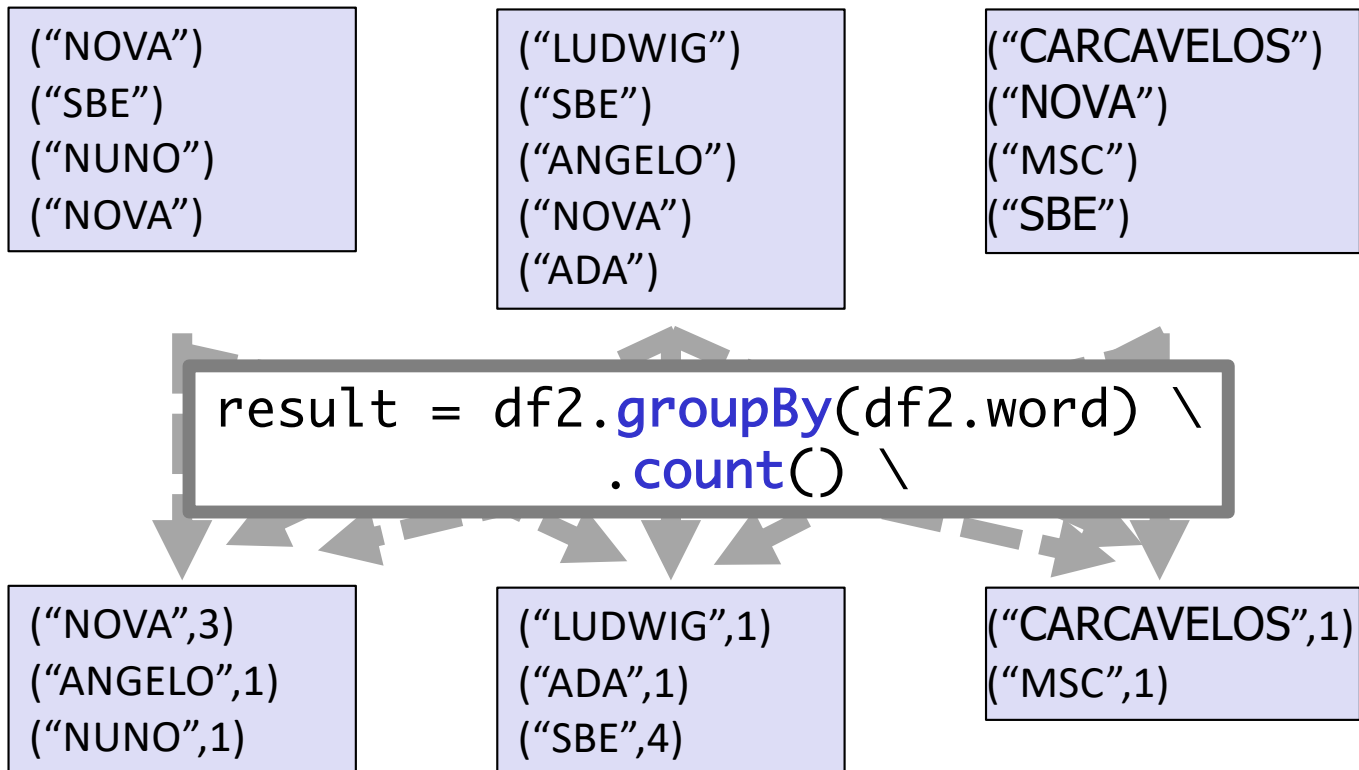
SECOND EXAMPLE: GROUPBY().COUNT()

groupBy(column).count()

- Counts the number of rows in the group, adding a column with name "count".

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))  
result = df2.groupBy(df2.word) \  
            .count() \  
            .where(col("count") > 1)
```


SECOND EXAMPLE: REDUCEBYKEY



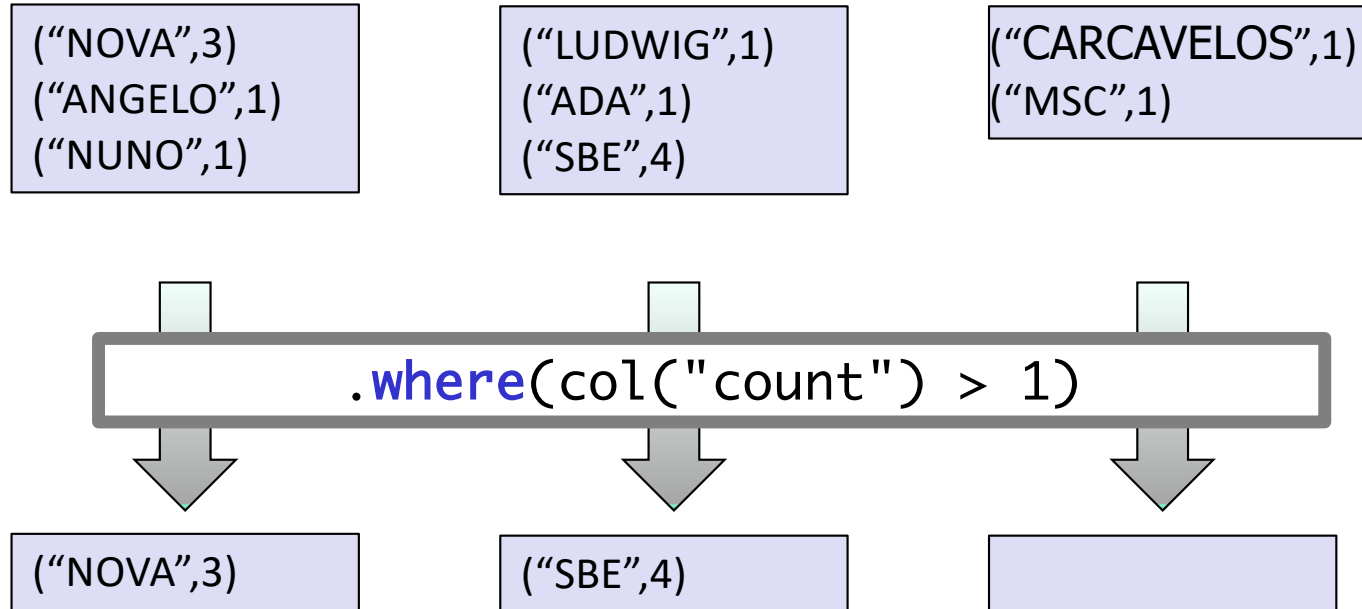
SECOND EXAMPLE: WHERE

where (condition)

- Returns a DataFrame with the rows that satisfy the given condition.

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))  
result = df2.groupBy(df2.word) \  
             .count() \  
             .where(col("count") > 1)
```

SECOND EXAMPLE: FILTER



PROGRAMMING AND EXECUTION MODEL

DataFrame programs are converted into RDD programs, which involve:

- **Transformations:** RDD -> RDD
- **Actions:** RDD -> Result (directly available to the client application)

Execution consists in applying the transformations in all the partitions of an RDD in parallel

- Performance is best when a RDD partition result does not require data from input RDD partitions located in different nodes (i.e., avoids shuffles)

FROM DATAFRAME TO RDDs

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))  
result = df2.groupBy(df2.word) \  
              .count() \  
              .where(col("count") > 1)
```

```
freq = doc.flatMap (lambda s: s.split(' '))  
              .map(lambda s: (s,1))  
              .reduceByKey (lambda v1,v2: v1+v2)  
              .filter(lambda t: t[1] > 1)
```

FROM DATAFRAME TO RDDs (2)

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))  
result = df2.groupBy(df2.word) \  
              .count() \  
              .where(col("count") > 1)
```

```
freq = doc.flatMap (lambda s: s.split(' '))  
              .map(lambda s: (s,1))  
              .reduceByKey (lambda v1,v2: v1+v2)  
              .filter(lambda t: t[1] > 1)
```

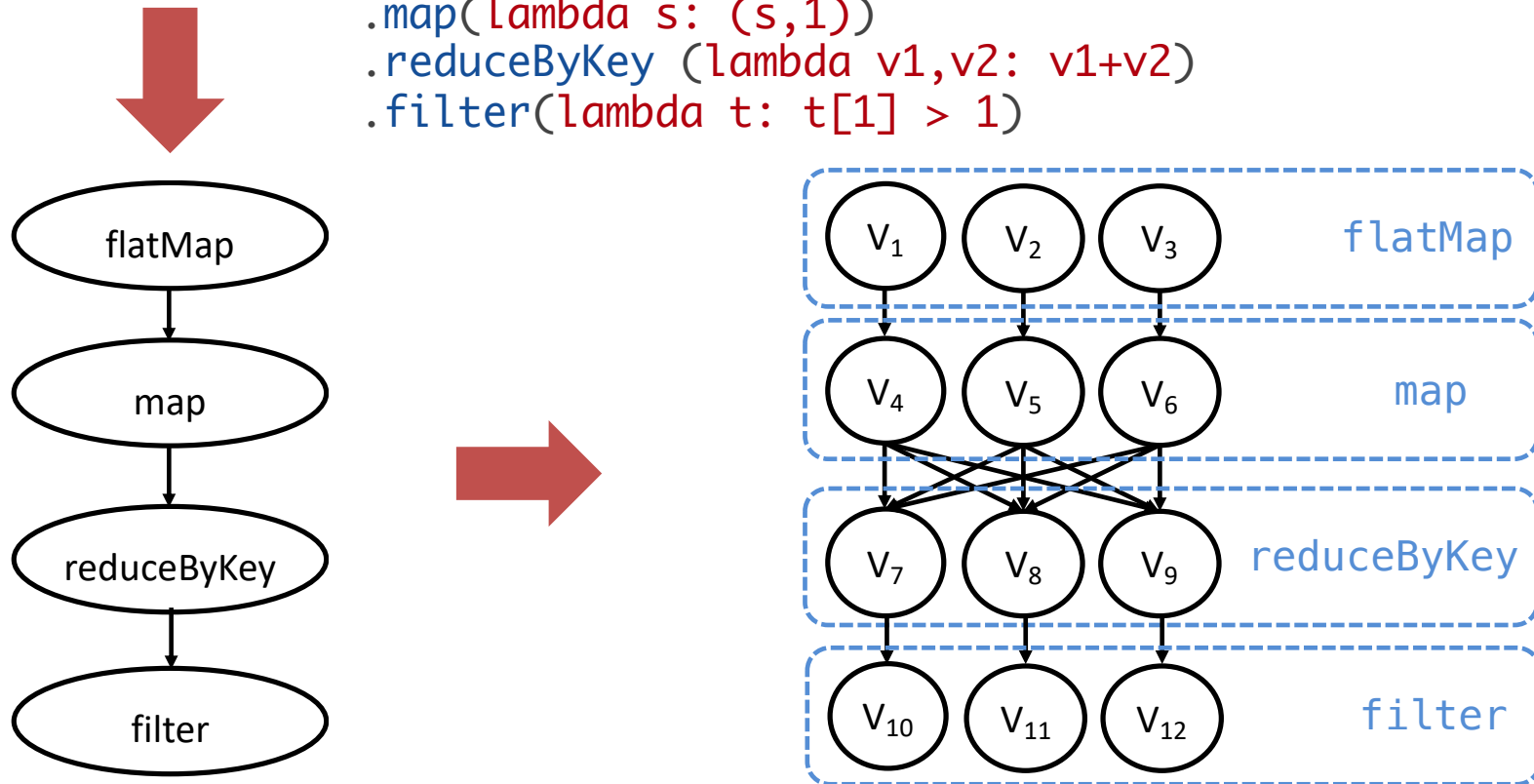
FROM DATAFRAME TO RDDs (2)

```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))
result = df2.groupBy(df2.word) \
    .count() \
    .where(col("count") > 1)
```

```
freq = doc.flatMap (lambda s: s.split(' '))
    .map(lambda s: (s,1))
    .reduceByKey (lambda v1,v2: v1+v2)
    .filter(lambda t: t[1] > 1)
```

SECOND EXAMPLE: COMPLETE EXECUTION

```
freq = doc.flatMap (lambda s: s.split(' '))  
           .map(lambda s: (s,1))  
           .reduceByKey (lambda v1,v2: v1+v2)  
           .filter(lambda t: t[1] > 1)
```



APACHE SPARK: (SCALA) API EXCERPT

Transformations	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	$count() : RDD[T] \Rightarrow Long$ $collect() : RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T$ $lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}$

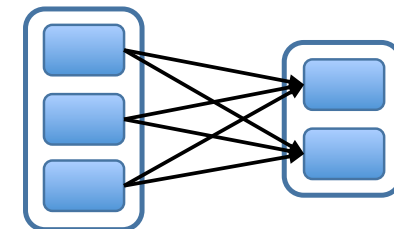
Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

PROGRAMMING MODEL: DEPENDENCIES

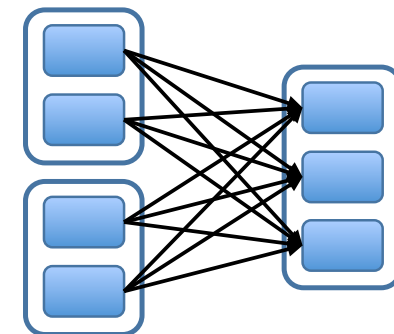
Wide-Dependencies are produced when an RDD partition depends on multiple partitions stored on different nodes

groupBy, join

Expensive due to high cost of network bandwidth



groupByKey



join with inputs not
co-partitioned

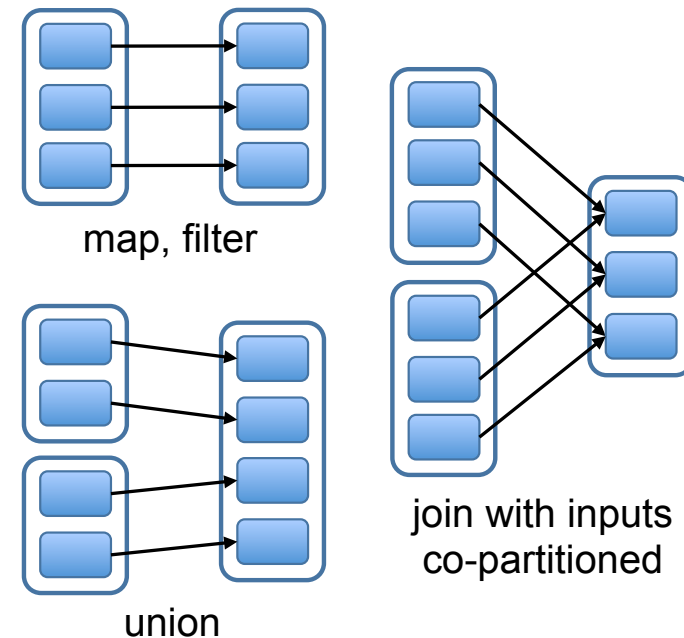
PROGRAMMING MODEL: DEPENDENCIES (2)

Narrow-dependencies

are produced when a RDD partition depends on data that is co-located (in the same node).

Filter (where), map

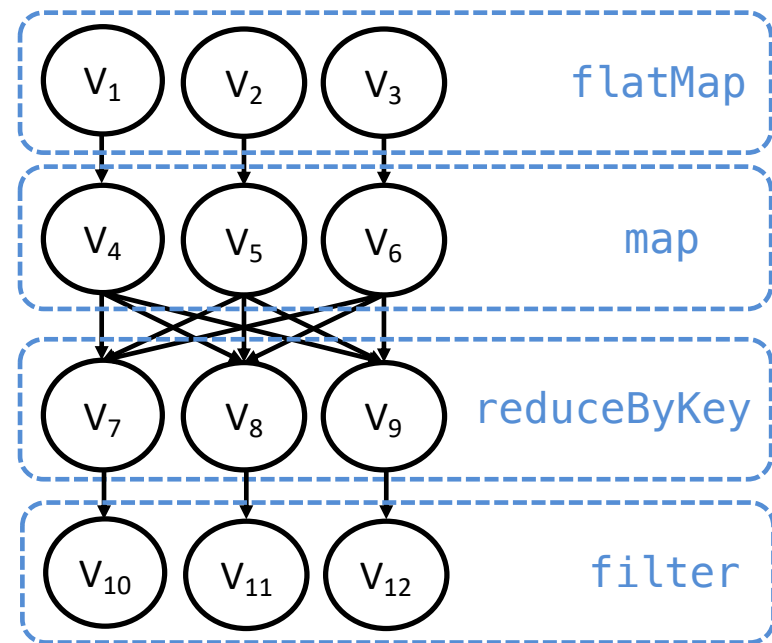
Fast as executed in the same machine.



FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost partitions**, using lineage information.

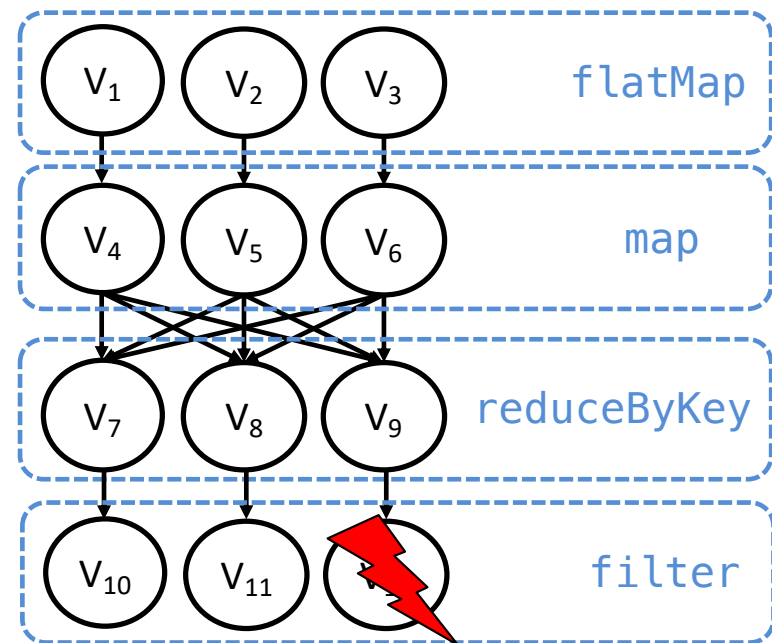
Optimized by persisting intermediate RDDs.



FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost partitions**, using lineage information.

Optimized by persisting intermediate RDDs.

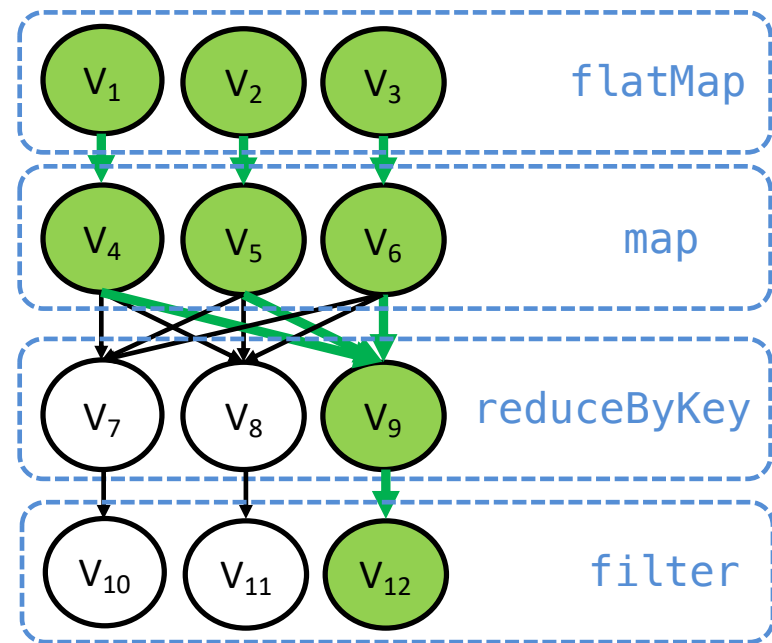


FAULT-TOLERANCE

Sparks deals with node failures by **recomputing lost partitions**, using lineage information.

Optimized by persisting intermediate RDDs.

In the example, if V_9 is persisted, lots of recomputation would be saved.



EXERCISES

Consider the information about products, stored in file "shopdata.csv", with the following format (where elements are separated by a tab):

` `` `store product price` `` ` , where elements are separated by a tab.

6Ave Express LLC	13.3 MacBook Air (Mid 2017, Silver)	892.49
Amazon.com	13.3 MacBook Air (Mid 2017, Silver)	979
Best Buy	13.3 MacBook Air (Mid 2017, Silver)	899.99
bhphotovideo.com	13.3 MacBook Air (Mid 2017, Silver)	799

LOAD CSV FILE

dataframe = spark.read.csv(filename)

- Creates a Dataframe from a CSV file.
- Option "header" specifies if the first line is the header of the table.
- Option "inferSchema" instructs Spark to infer data type for each column.

```
df = spark.read.option("header", True) \  
                .option("inferSchema", True) \  
                .csv("shopdata.csv")
```


REGISTER DATAFRAME AS SQL VIEW

dataframe.createOrReplaceTempView(table_name)

Registers a DataFrame as a SQL view / table. The table is available for the SparkSession.

After registering the table, it is possible to issue SQL statements.

```
df = spark.read.option("header", True) \  
                .option("inferSchema", True) \  
                .csv("shopdata.csv")
```

```
df.createOrReplaceTempView("products")
```

```
result = spark.sql("SELECT * FROM products")  
result.show()
```

EXECUTING SQL OPERATIONS

dataframe = spark.sql(SQL statement)

Execute SQL statement. The result is a DataFrame.

```
df = spark.read.option("header", True) \
                .option("inferSchema", True) \
                .csv("shopdata.csv")
```

```
df.createOrReplaceTempView("products")
```

```
result = spark.sql("SELECT * FROM products")
result.show()
```

EXECUTING SQL OPERATIONS

dataframe = spark.sql(SQL statement)

Execute SQL statement. The result is a DataFrame.

```
df = spark.read.option("header", True) \
                .option("inferSchema", True) \
                .csv("shopdata.csv")

df.createOrReplaceTempView("products")

result = spark.sql("SELECT * FROM products")
result.show()
```

EXECUTING SQL OPERATIONS

dataframe

Execute

df = spark

df.create

result =

result.show

```
try:
    df = spark.read.option("header", True).option("inferSchema", True).csv("shopdata.csv")

    df.createOrReplaceTempView("products")

    result = spark.sql("SELECT * FROM products")

    result.show()

finally:
    spark.stop()
```

	shop	product	price
	[AIM USA]	Spartan - 3-Targe...	310.79
	[AIM USA]	Spartan - 3-Targe...	329.36
	[AIM USA]	Spartan - 3-Targe...	399.0
	[AIM USA]	Spartan - 3-Targe...	307.62
	\$aveTronix - Walm...	SanDisk - Ultra 3...	11.0
1	SHOP DIRECT	JBL Clip2 Portabl...	44.99
1	Stop Electronic...	Hisense - 55 Clas...	916.4

SIMPLE STATISTICS (1)

Let's assume data is registered under view name products.

Find the **minimum** price for each product.

```
result = spark.sql("""SELECT product, min(price) AS min_price FROM products
                        GROUP BY product""")

result.show()
```

SIMPLE STATISTICS (2)

Find the **average** price for each product.

```
result = spark.sql("""SELECT product, mean(price) AS avg_price FROM products
                        GROUP BY product""")

result.show()
```

SIMPLE STATISTICS (3)

Find the **minimum** price **and shop** for each product.

```
result = spark.sql("""SELECT m.product, p.shop, m.min_price FROM
  (SELECT product, min(price) AS min_price FROM products GROUP BY product) m
  JOIN products p ON m.product = p.product AND m.min_price = p.price
  ORDER BY m.product""")
```

OUTLINE

Computing services

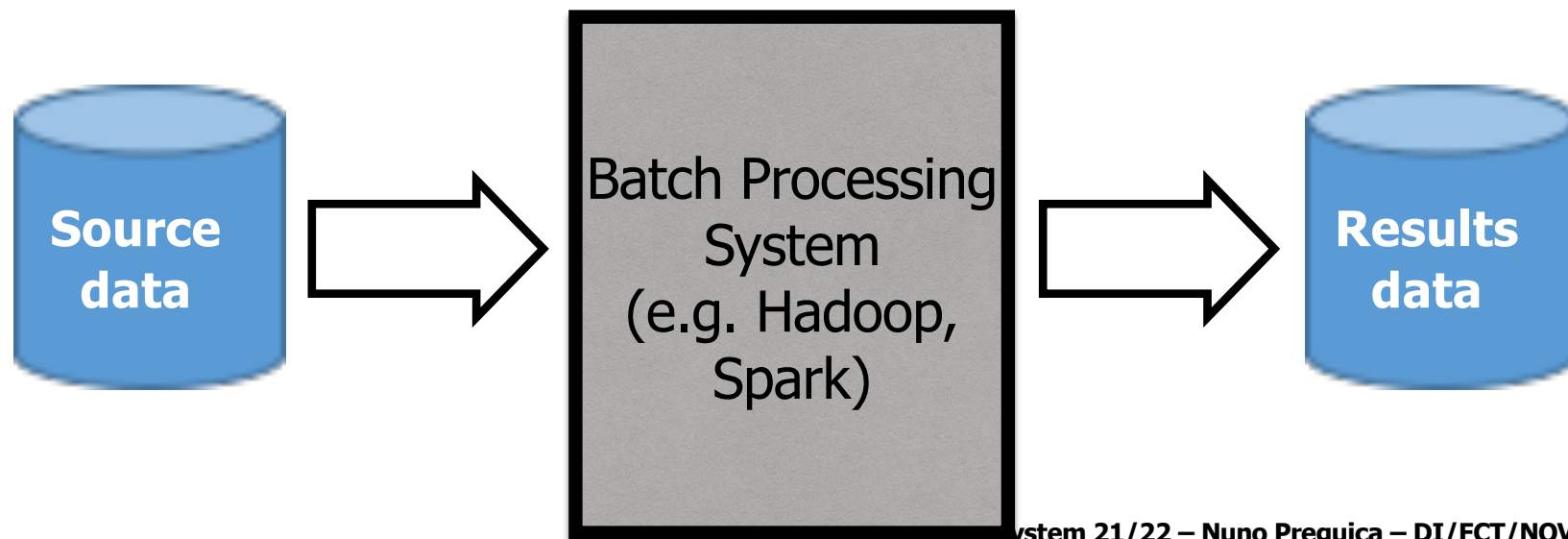
1. First generation batch processing: Map-reduce
2. Second generation batch processing: Spark
3. **Stream processing**

BIG DATA / BATCH PROCESSING

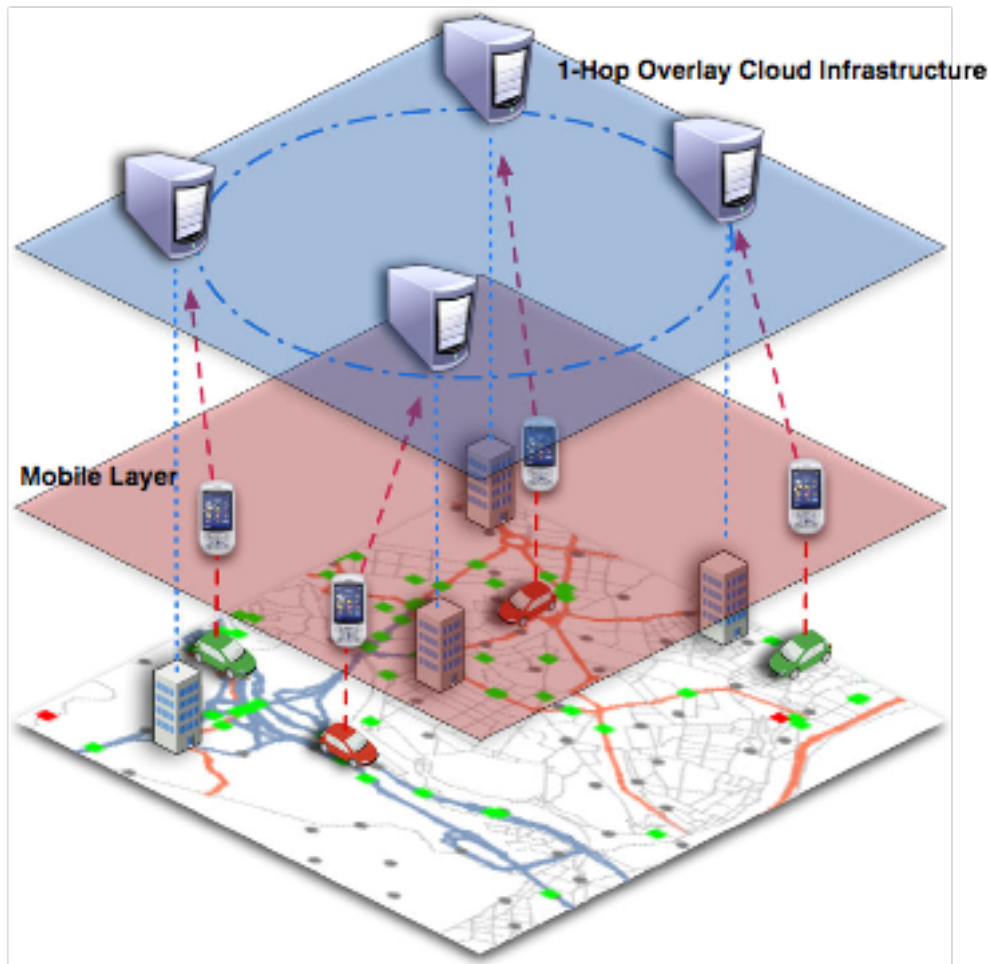
All data known at the time of processing

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

STREAMING PROCESSING: REQUIREMENTS

Need to process data as it arrive (or at most with a very small delay)

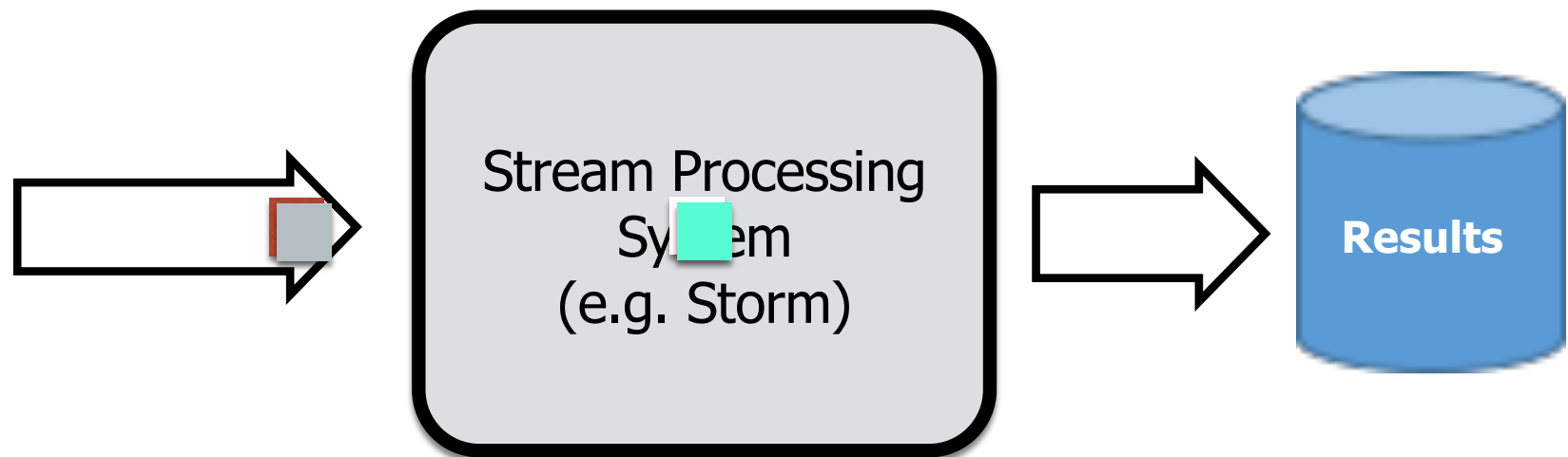
Need to be able to process data from multiple sources

Need to tolerate faults

TWO PROCESSING MODELS (1)

Continuous

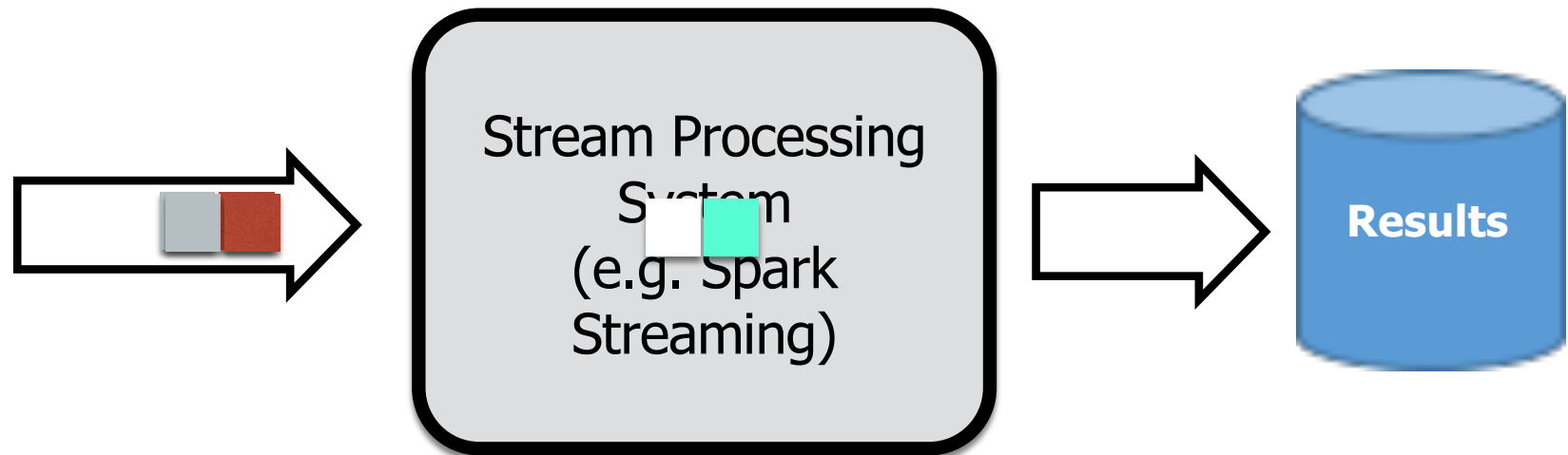
- Each tuple processed as it arrives
- Processing system may keep state for executing window computation and incremental computation



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 computation and incremental computation

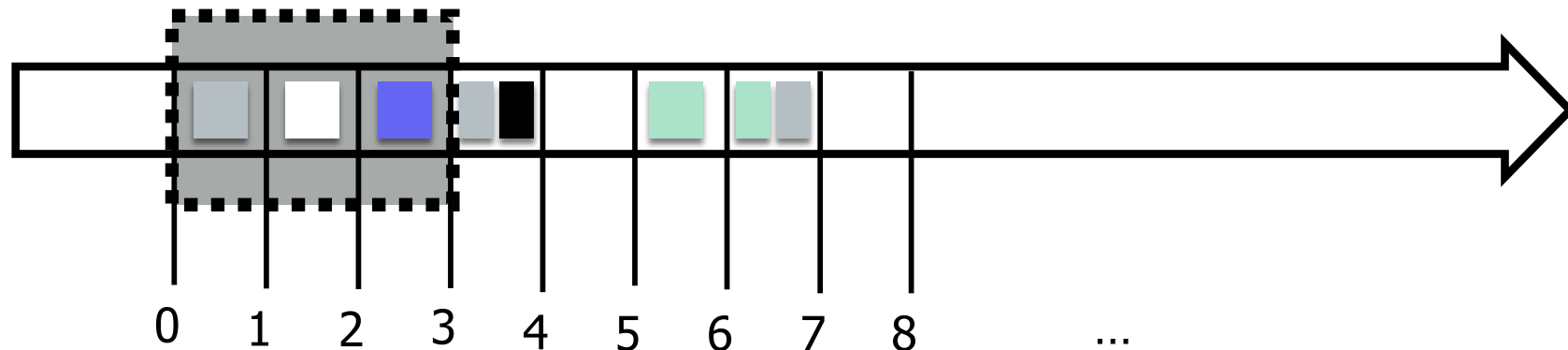


WINDOWING

When doing stream processing, it is often interesting to compute results based on data from a given interval, but compute results more frequently than the time interval — for example, process data of last 3 minutes, but produce results every minutes.

System for stream processing support the definition of sliding time windows.

E.g. In SparkStreaming, `s.window("3s")` would output results comprising the records in intervals: $[0,3)$, $[1,4)$, $[2,5)$, ...



SYSTEMS FOR STREAM PROCESSING

Continuous processing

- Apache Storm
 - Open sourced by Twitter
 - API: proprietary, SQL-like
- Apache Flink
 - API: proprietary, table-based (similar to DataFrames), SQL-like

Mini-batch processing

- Spark streaming
 - API: proprietary, table-based, SQL-like

SPARK STREAMING

NOTE: slides with Spark Streaming intro are just for those wanting to know a little more on this topic.

Spark Streaming is an extension of the core Spark API to enable scalable, high-throughput, fault-tolerant stream processing of live data streams.

Matei Zaharia, et. al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In Proc. SOSP'13.

http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

<http://spark.apache.org/streaming/>

OUTLINE

Computing services

1. First generation batch processing: Map-reduce
2. Second generation batch processing: Spark
3. Stream processing

Computing services @ Azure

ANALYTICS @ AZURE

IF YOU WANT...	USE THIS
Limitless analytics service with unmatched time to insight (formerly SQL Data Warehouse)	Azure Synapse Analytics
A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for Azure	Azure Databricks
A fully managed cloud Hadoop and Spark service backed by 99.9% SLA for your enterprise	HDInsight
A data integration service to orchestrate and automate data movement and transformation	Data Factory
Open and elastic AI development spanning the cloud and the edge	Machine Learning
Real-time data stream processing from millions of IoT devices	Azure Stream Analytics
A fully managed on-demand pay-per-job analytics service with enterprise-grade security, auditing, and support	Data Lake Analytics
Enterprise grade analytics engine as a service	Azure Analysis Services
A hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events	Event Hubs
Fast and highly scalable data exploration service	Azure Data Explorer
A simple and safe service for sharing big data with external organizations	Azure Data Share

AZURE HDINSIGHT

Azure HDInsight is a managed open-source analytics service.

Azure HDInsight is a cloud distribution of Hadoop components.

Open-source frameworks available: Hadoop, Apache Spark, Apache Hive, LLAP, Apache Kafka, Apache Storm, R, and more.

AZURE HDINSIGHT CLUSTER

To use HDInsight, a user needs to create a cluster.

A cluster is comprised by a set of machines: head nodes + worker nodes.

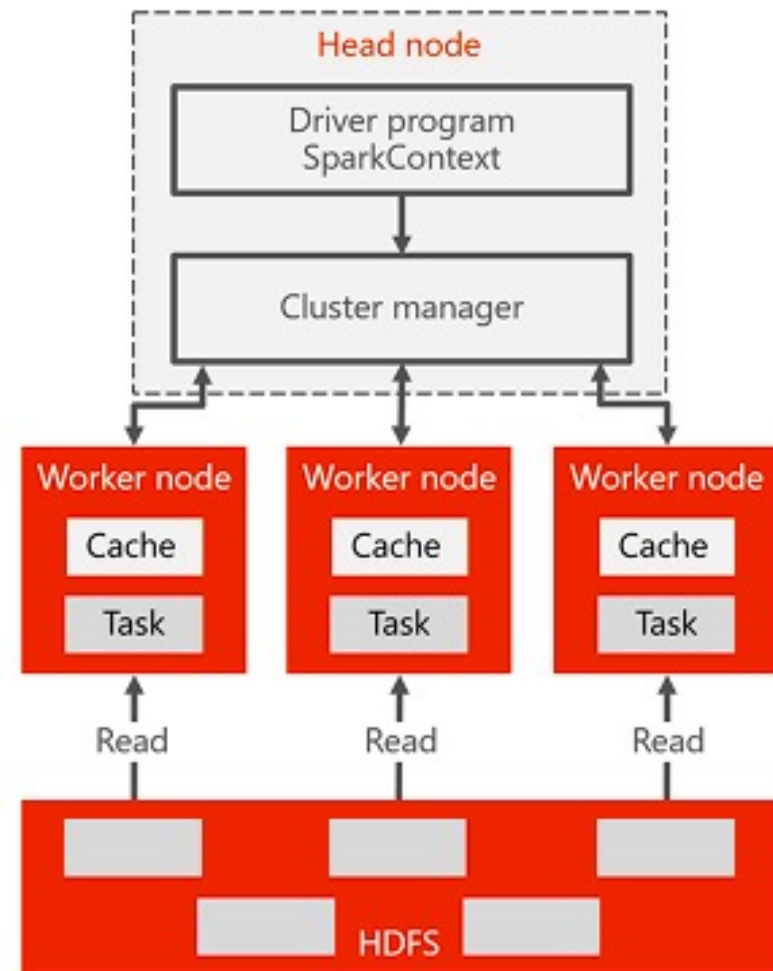
SPARK @ HDINSIGHT : ARCHITECTURE

Spark applications run as independent sets of processes on a cluster.

The SparkContext in the main program connects to a YARN cluster manager, which allocate resources across applications

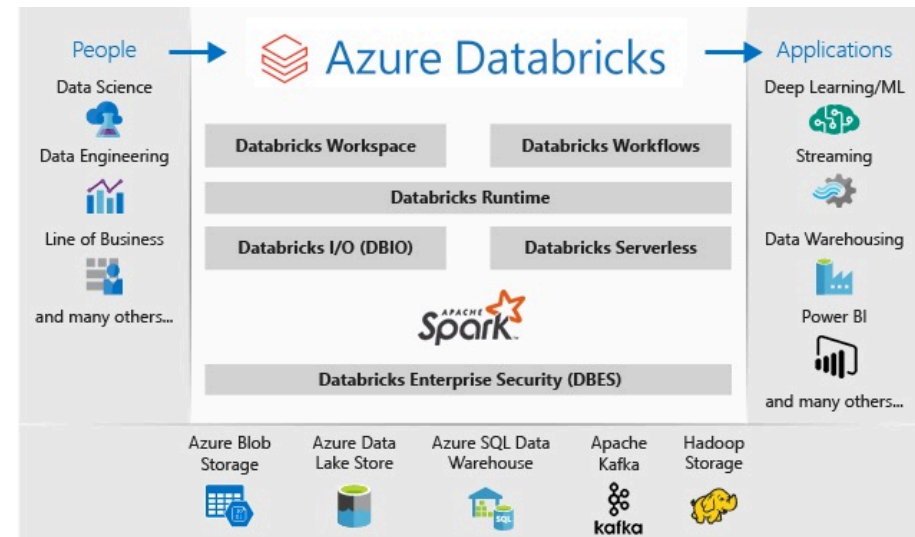
Once connected, Spark acquires executors on workers nodes in the cluster.

Spark sends the application code to the executors. Finally, SparkContext sends tasks to the executors to run.



AZURE DATABRICKS

“**Azure Databricks** is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Designed with the founders of Apache Spark, Databricks is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive workspace that enables collaboration between data scientists, data engineers, and business analysts.”

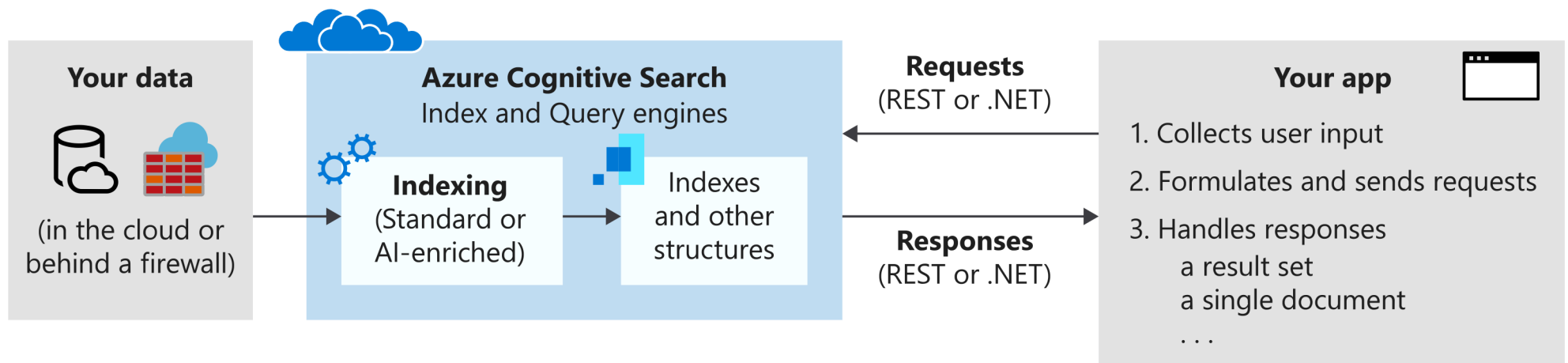


AZURE COGNITIVE SEARCH

Azure Cognitive Search is a search-as-a-service cloud solution.

- Text, Images, etc.
- Image recognition, OCR, etc.

Applications invoke data ingestion (indexing) to create and load an index. Optionally, it is possible to add cognitive skills to apply AI processes during indexing.



TO KNOW MORE

J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, OSDI'04.

M. Zaharia, et. al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI'12.

M. Zaharia, et. al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. SOSP'13.

<https://spark.apache.org/>

<https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-overview>

ACKNOWLEDGMENTS

Some text and images from Microsoft Azure online documentation and AWS online documentation.