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

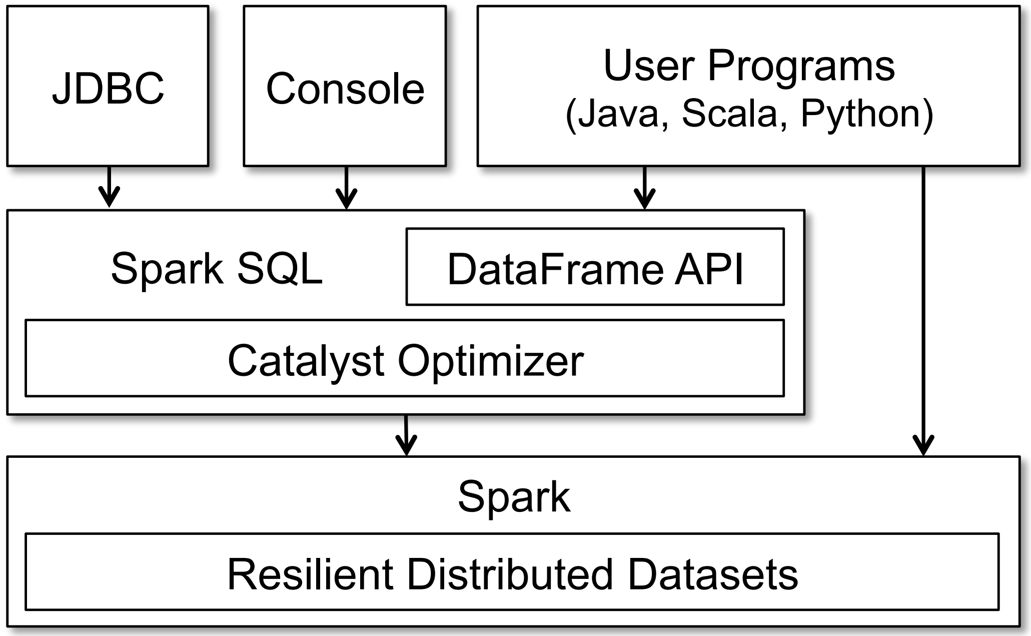
SparkSQL Architecture

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)

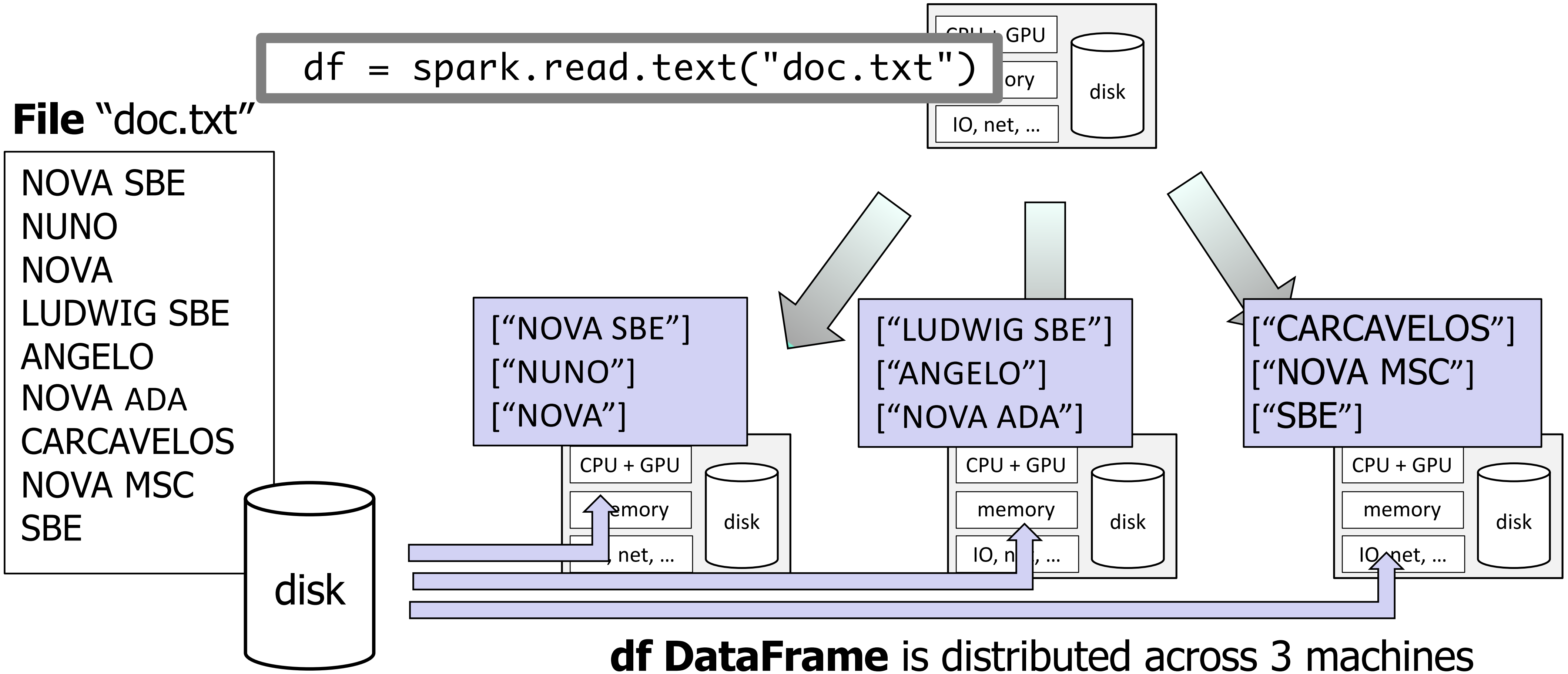
- Creates a Dataframe from a text file. The Dataframe includes a single column named "value", and each line is a row of the DataFrame.

```
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

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

File "doc.txt"



df DataFrame is distributed across 3 machines

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 DataFrme might be distributed over multiple machines, and your program is running in a single machine.

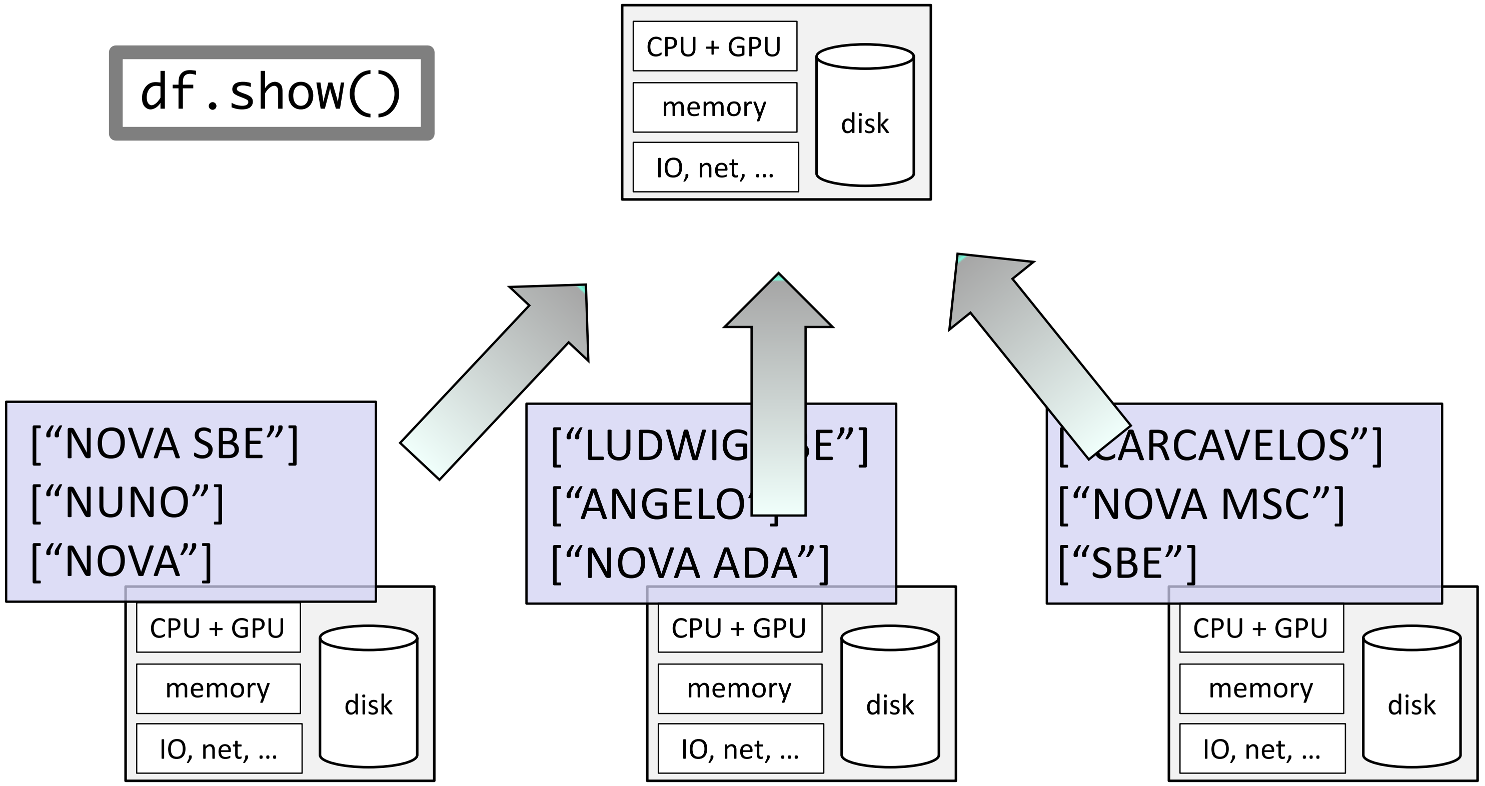
```
from pyspark.sql import SparkSession

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, delimeter)

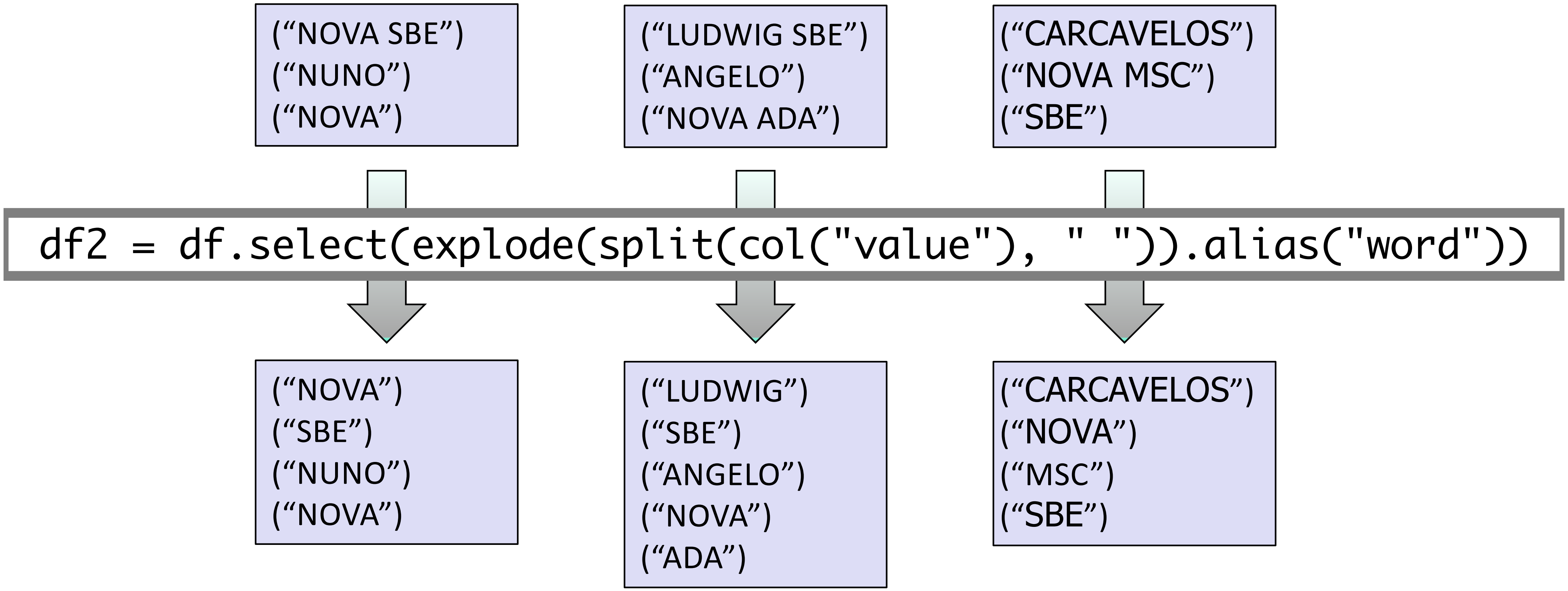
- Divides the value of the column by delimiter, creating an array of values

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



```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))
```

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)
```

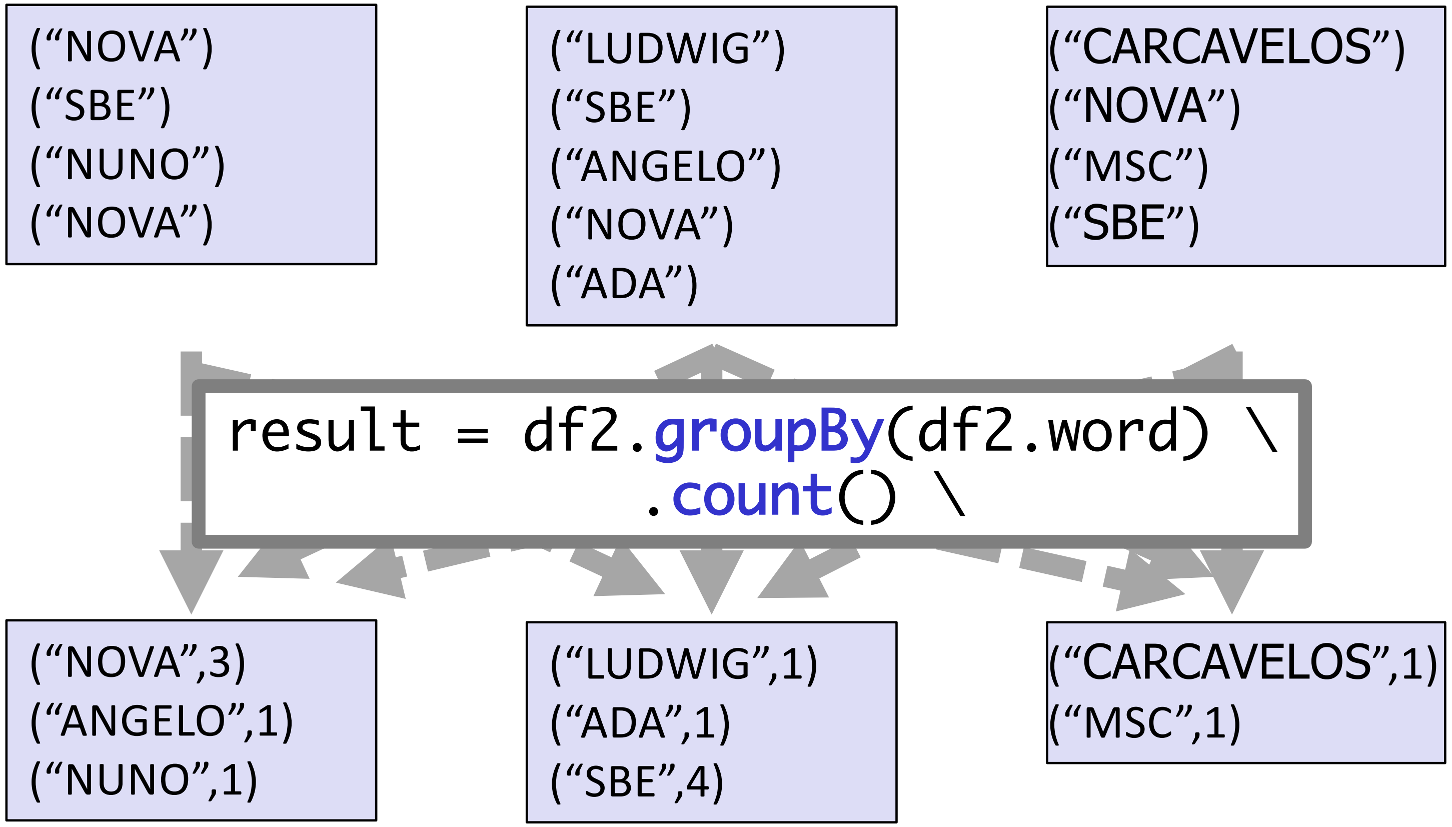
Second example: groupBy().count()

groupBy(column).count()

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

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

SECOND EXAMPLE: REDUCEBYKEY



```
result = df2.groupBy(df2.word) \
            .count() \
```

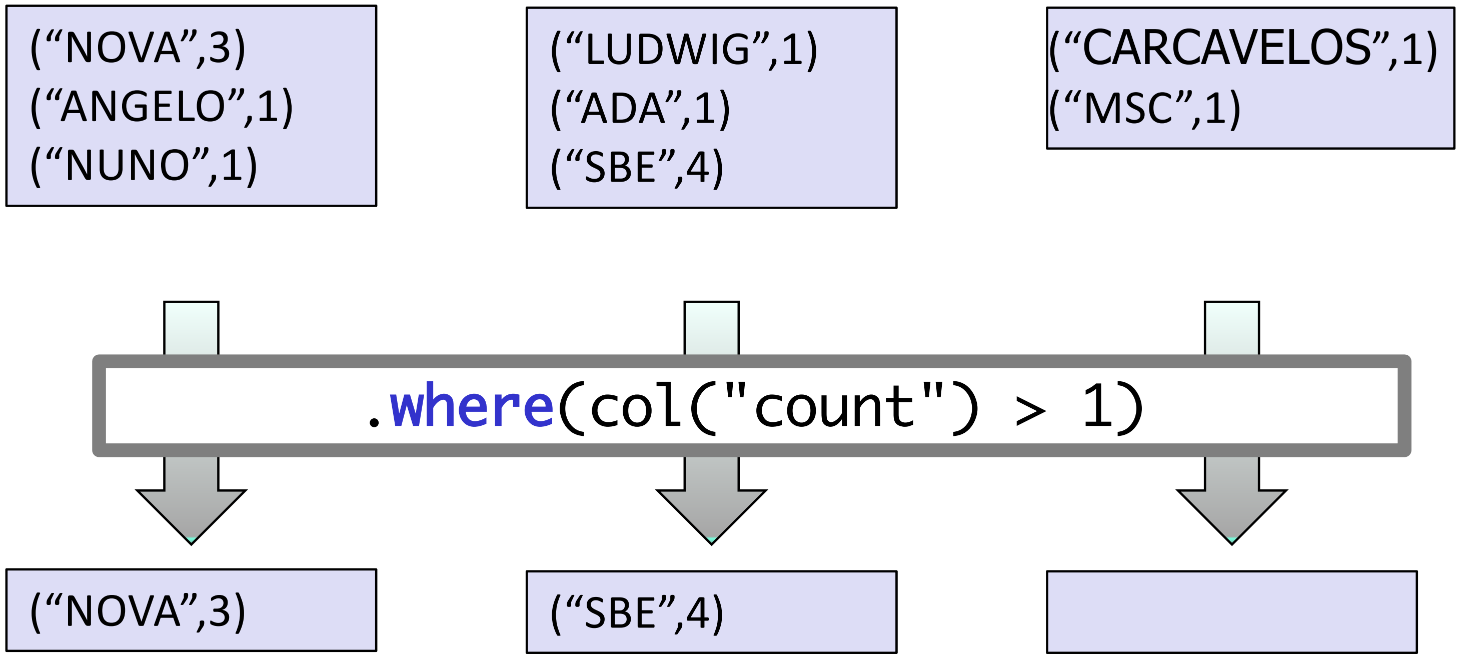
SECOND EXAMPLE: WHERE

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)
```

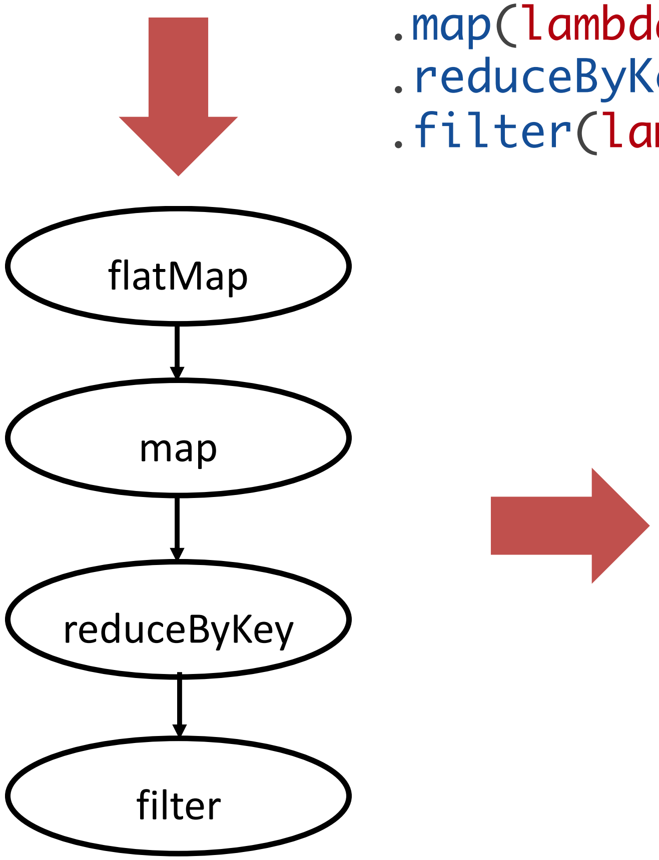
From DataFrame to RDDs (2)

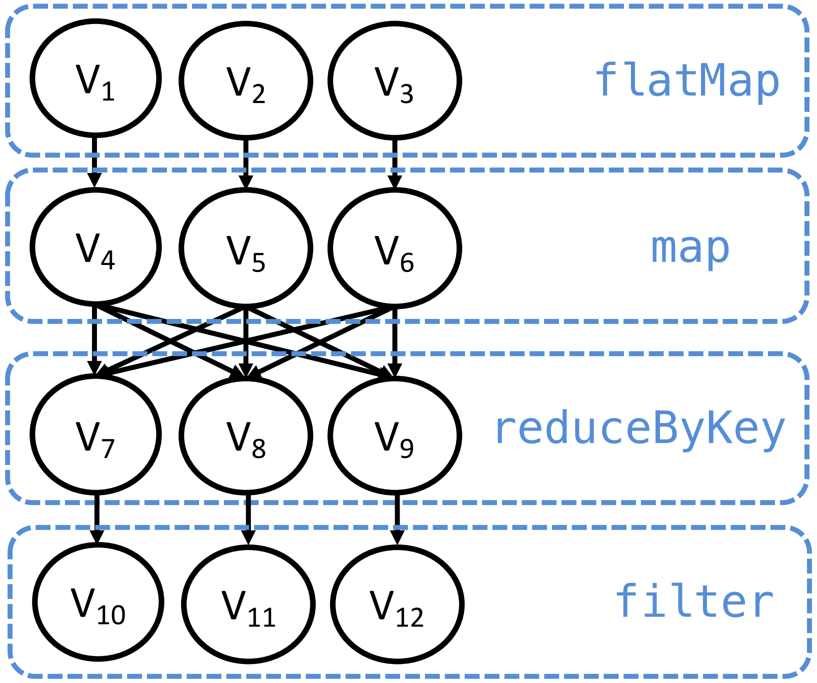
```
df2 = df.select(explode(split(col("value"), " ")).alias("word"))
result = df2.groupBy(df2.word) \
            .count() \
            .where(col("count") > 1)
```

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

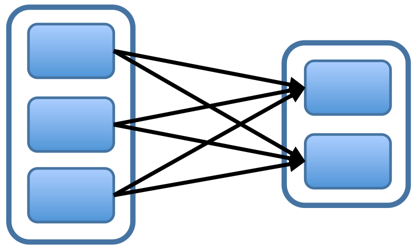
Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

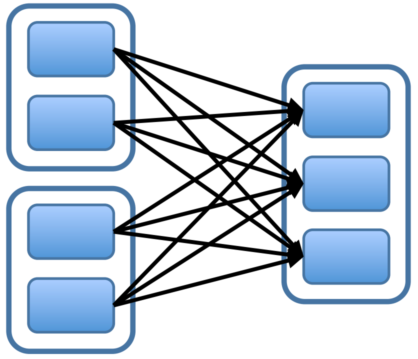
PROGRAMMING MODEL: DEPENDENCIES

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

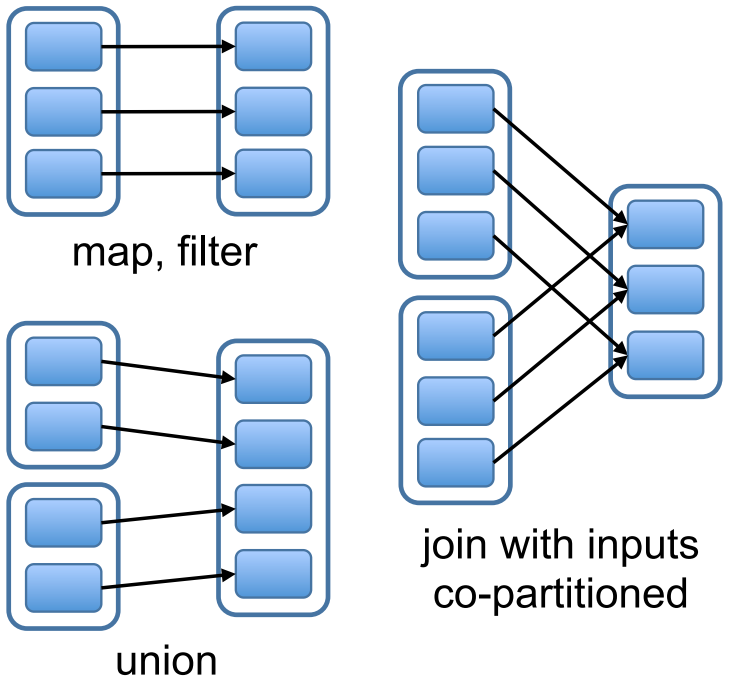




Programming Model: Dependencies (2)

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

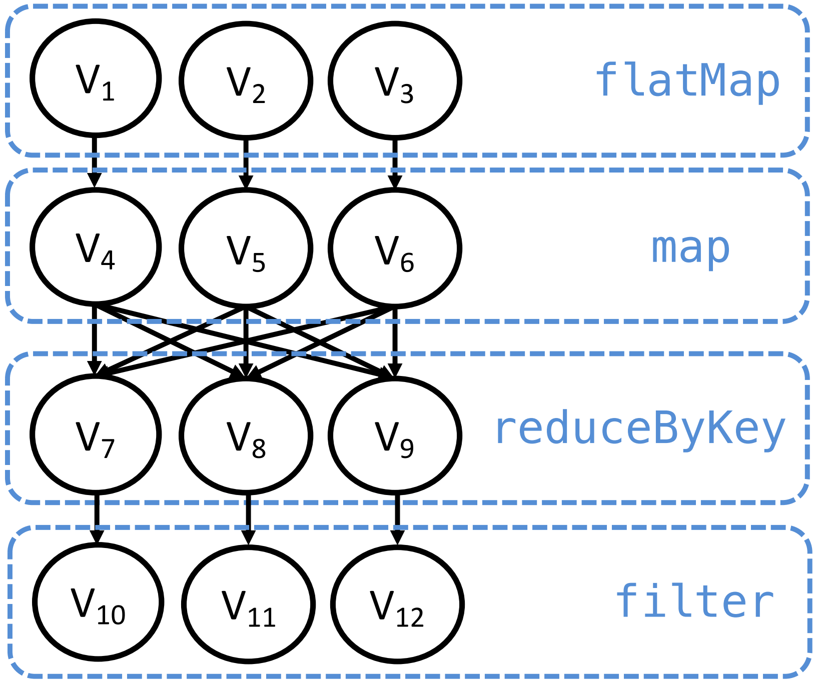
- Filter (where), map
- Fast as executed in the same machine.



FAULT-TOLERANCE

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

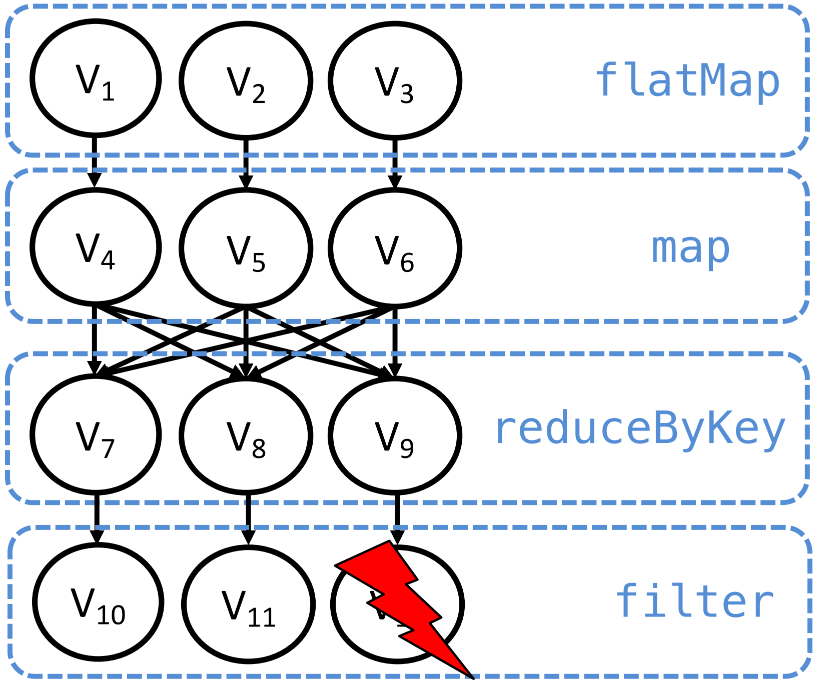
- Optimized by persisting intermediate RDDs.



FAULT-TOLERANCE

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

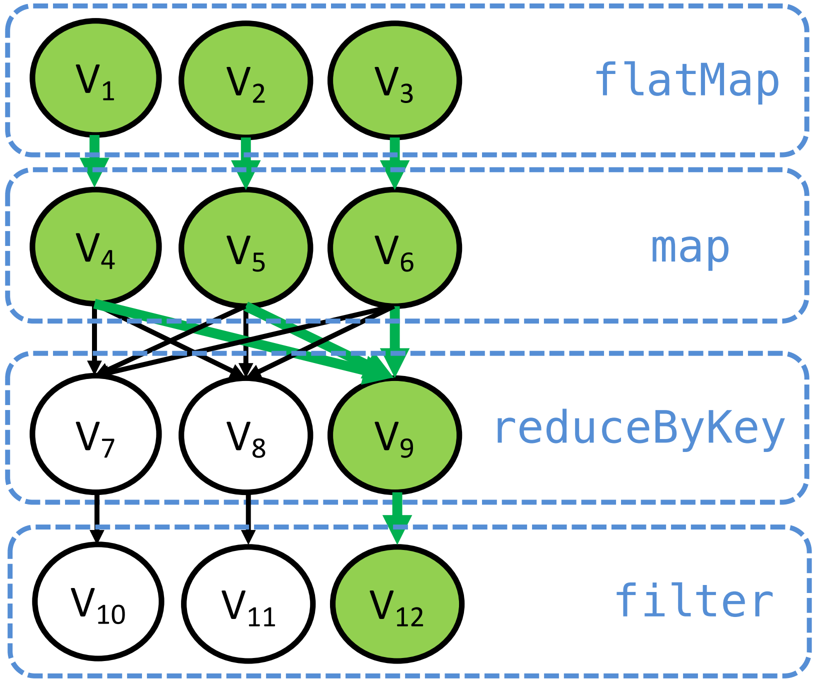
Optimized by persisting intermediate RDDs.



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.

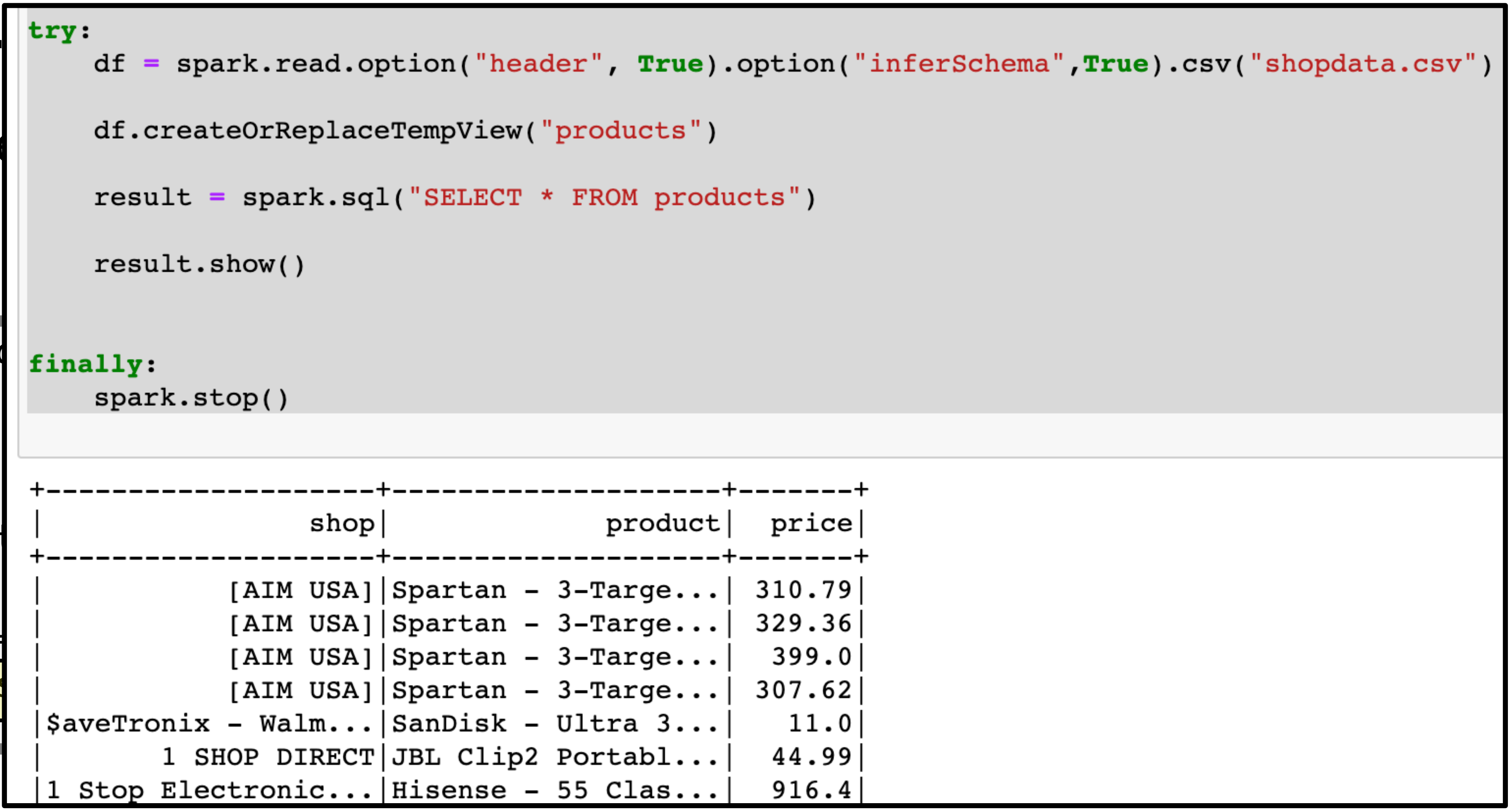
```
df = spark.read.option("header", True) \
               .option("inferSchema",True) \
               .csv("shopdata.csv")

df.createOrReplaceTempView("products")

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

EXECUTING SQL OPERATIONS

datafr

Execute

```
df = sp

df.creat

result =
result.s
```



```
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

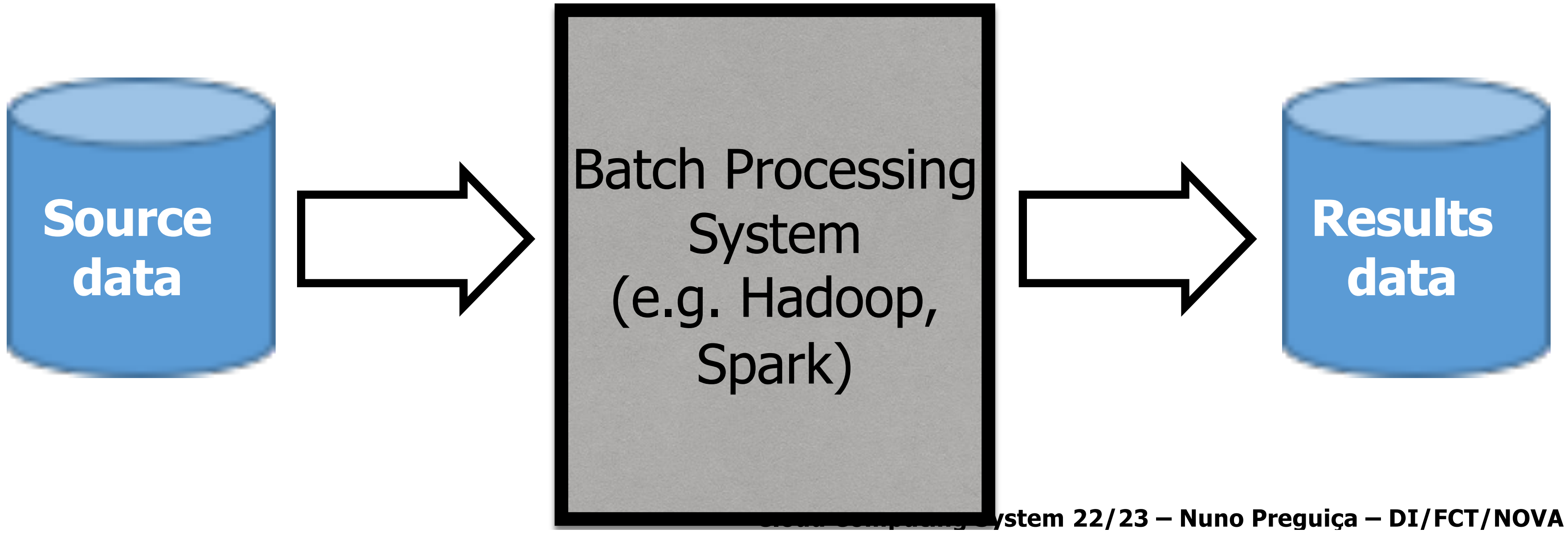
1. First generation batch processing: Map-reduce
2. Second generation batch processing: Spark
3. **Stream processing**

BIG DATA / BATCH PROCESSING

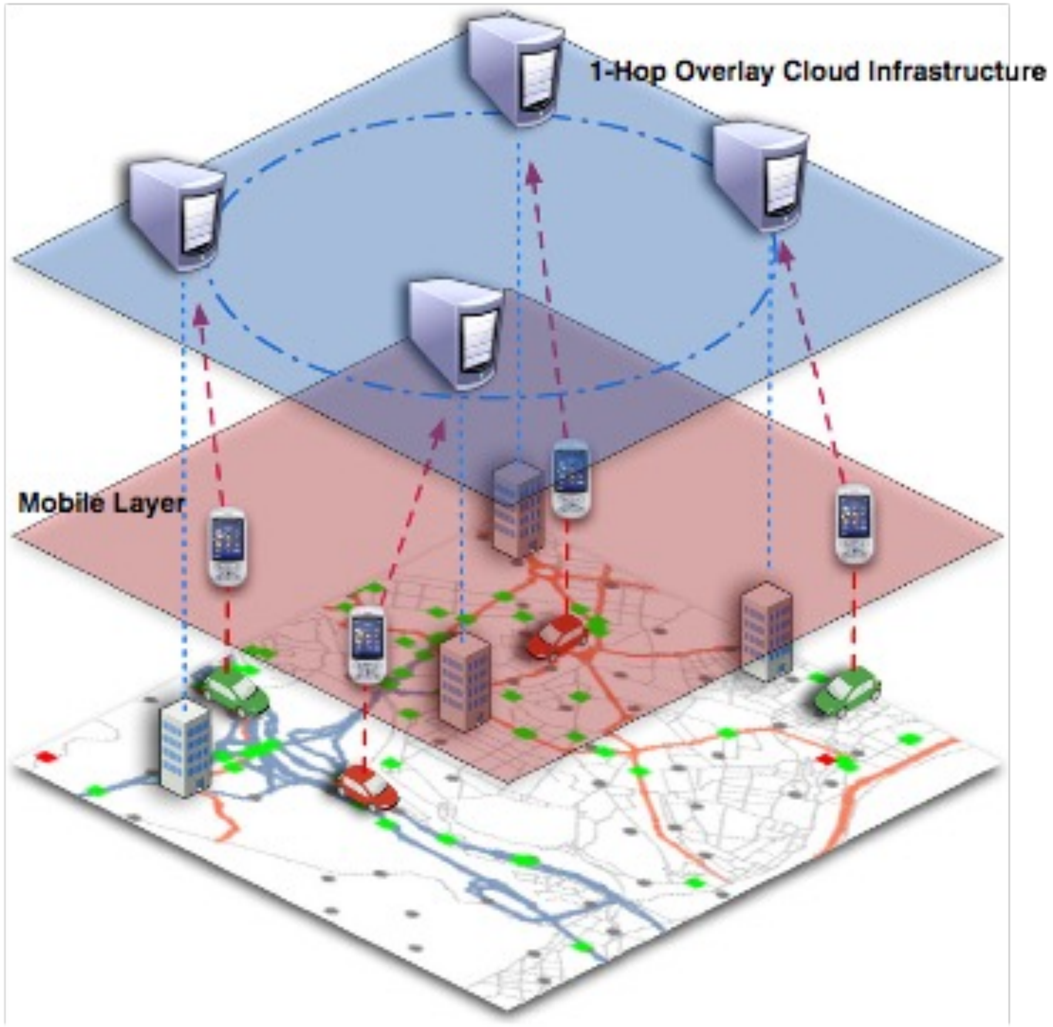
All data known at the time of processing

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)

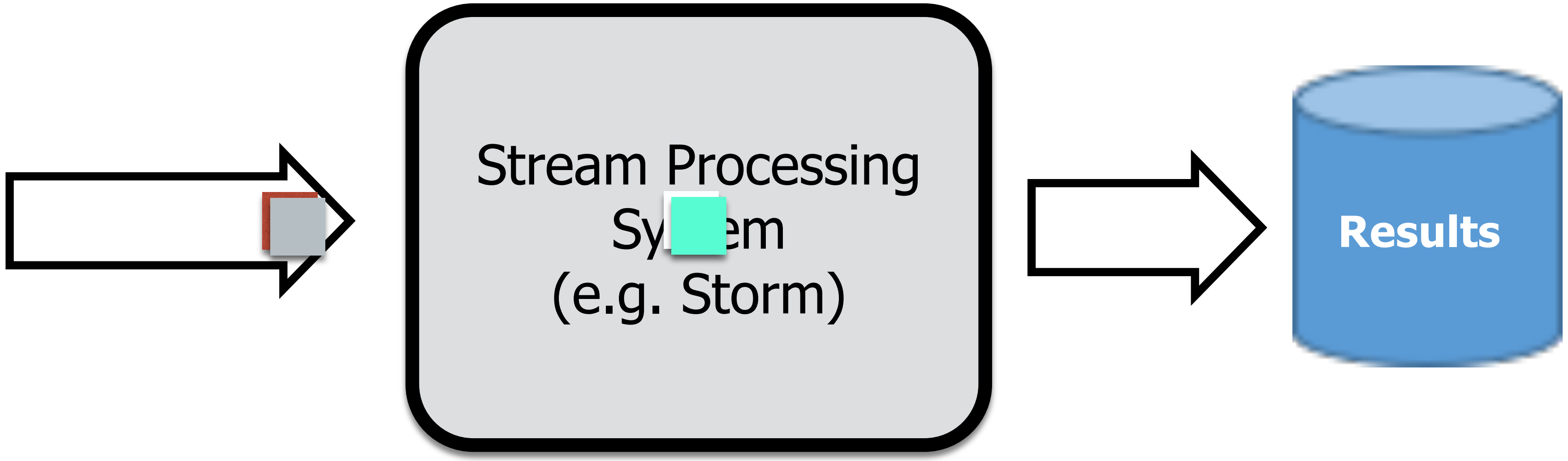
Need to be able to process data from multiple sources

Need to tolerate faults

TWO PROCESSING MODELS (1)

Continuous

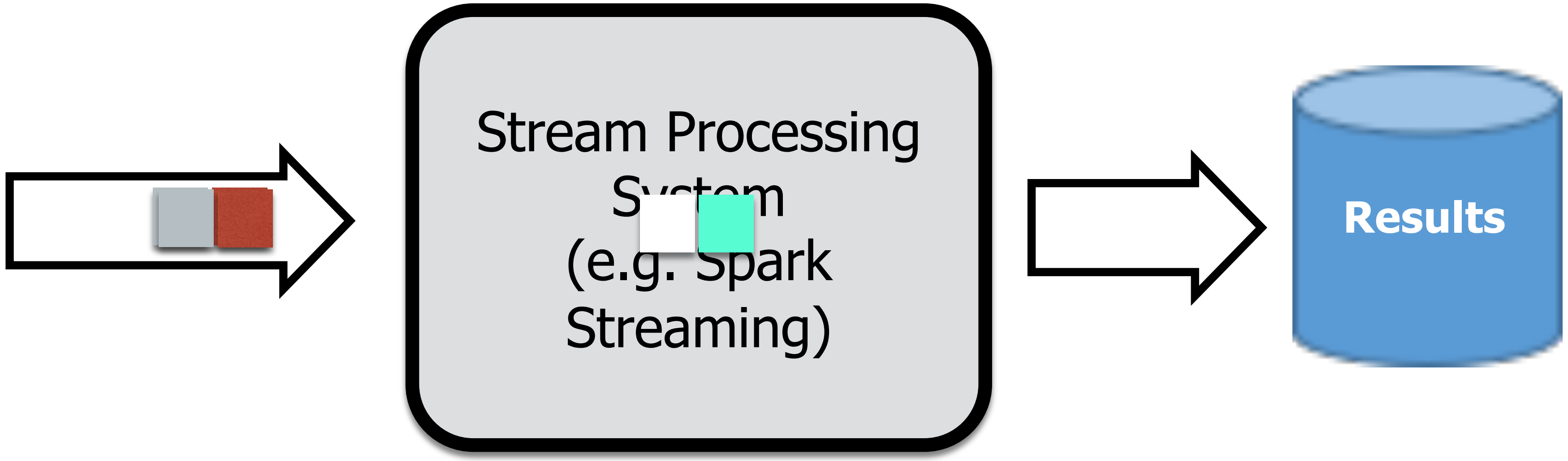
- Each tuple processed as it arrives
- Processing system may keep state for executing window computation and incremental computation



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

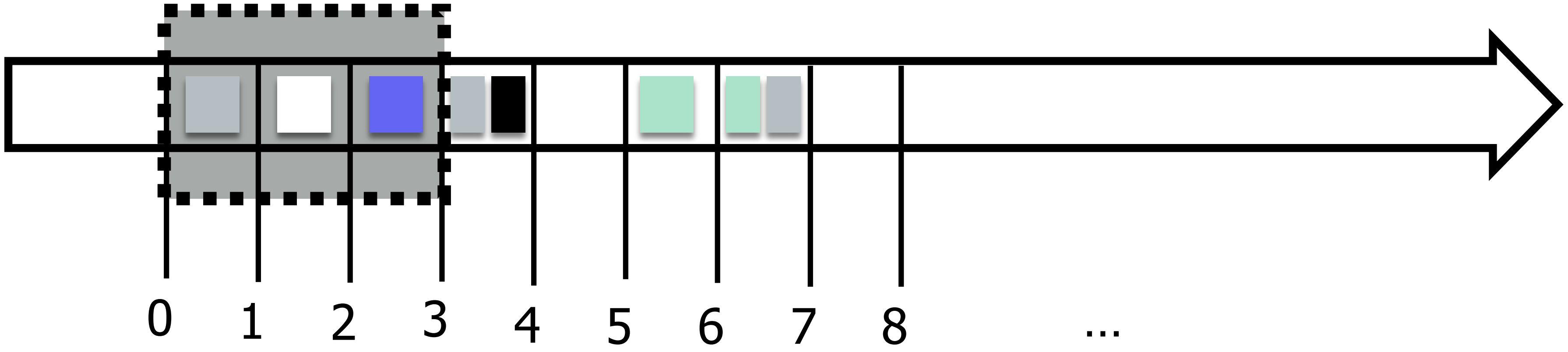


WINDOWING

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/

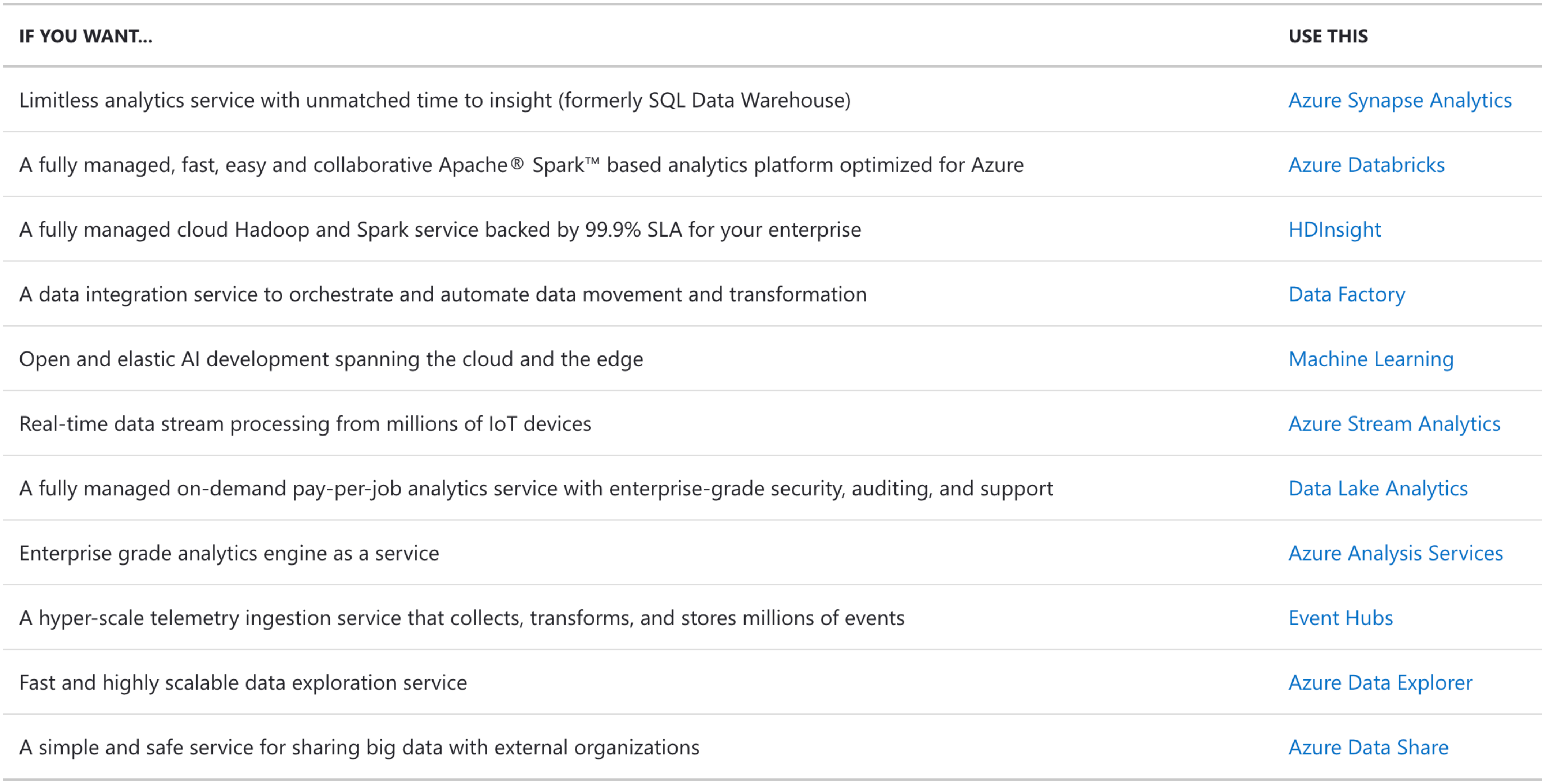
Outline

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.

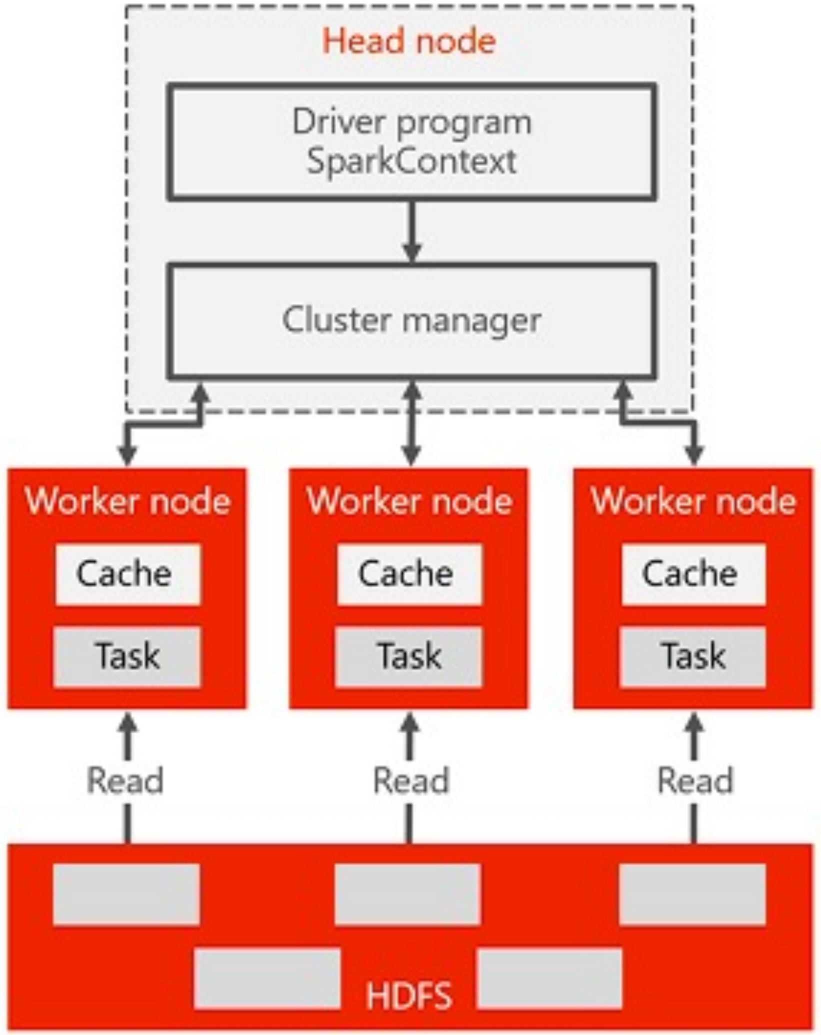
SPARK @ HDINSIGHT : ARCHITECTURE

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

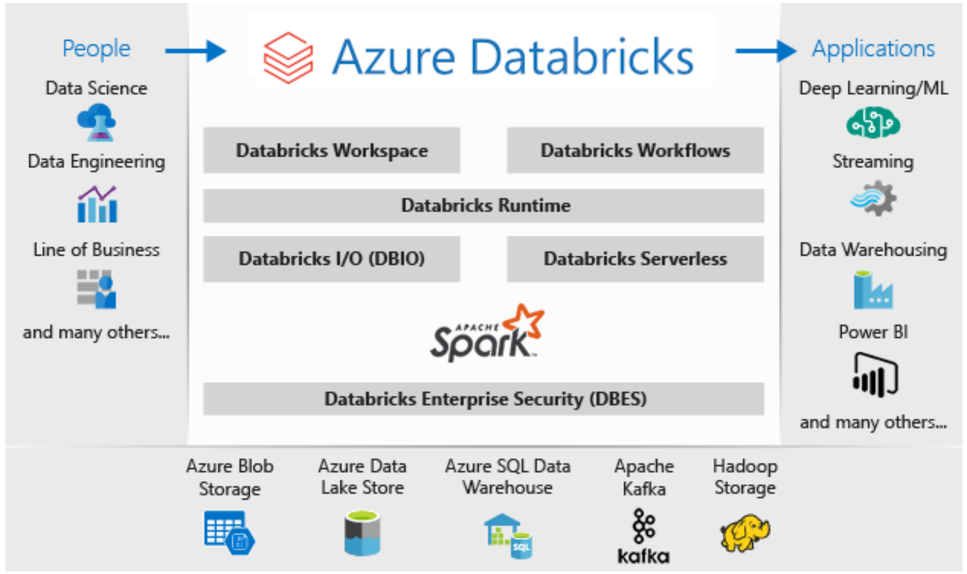
Once connected, Spark acquires executors on workers nodes in the cluster.

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."

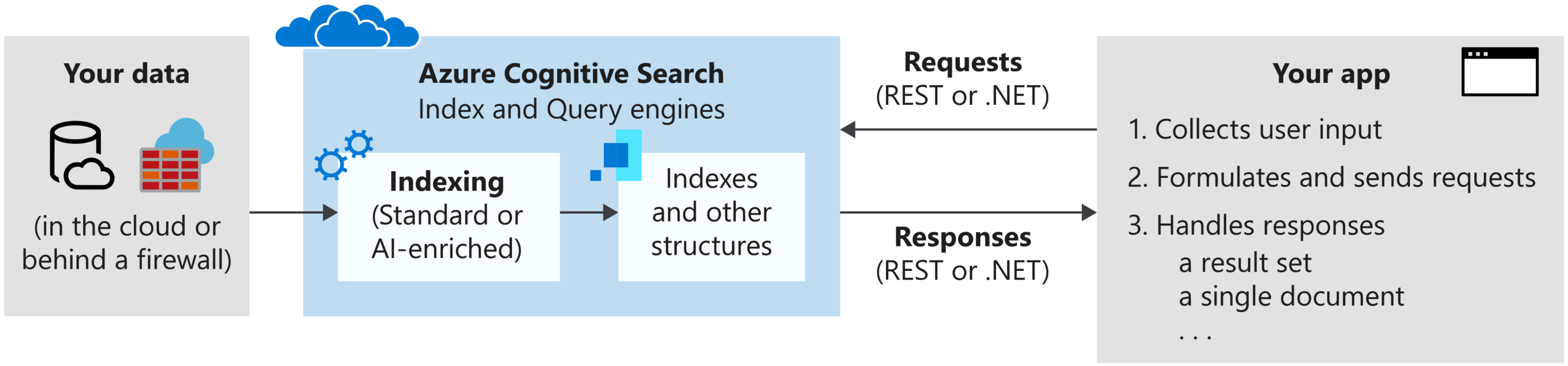


Azure Cognitive Search

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.