Intro to Apache Spark

Distributed computing: Definition

A distributed computing system is a system including several computational entities where:

- Each entity has its own local memory
- All entities communicate by message passing over a network

Each entity of the system is called a node.

Distributed computing: Motivation

There are several reasons why one may want to distribute data and processing:

- Scalability
  - ✔ The data do not kept in the memory/storage of one node
  - ✔ The processing power of more processor can reduce the time to solution
- Fault tolerance / availability
  - ✔ Continuing delivering a service despite node crashes.
- Latency
  - ✔ Put computing resources close to the users to decrease latency

Programming distributed systems

Challenges

Context of execution

- Large number of resources
- Resources can crash (or disappear)
  ✔ Failure is the norm rather than the exception.
- Resources can be slow

Objectives

- Run until completion
  ✔ And obtain a correct result :-)
- Run fast

The Big Data approach

talent sprint

Provide a distributed computing execution framework

- Simplify parallelization
  - ✔ Define a programming model
  - ✔ Handle distribution of the data and the computation
- Fault tolerant
  - ✔ Detect failure
  - ✔ Automatically takes corrective actions
- Code once (expert), benefit to all

Limit the operations that a user can run on data

- Inspired from functional programming (eg, MapReduce)
- Examples of frameworks:
  - ✔ Hadoop MapReduce, Apache Spark, Apache Flink, etc.

Apache Hadoop

In a few words

- Built on top of the ideas of Google
- A full data processing stack
- The core elements
  - ✔ A distributed file system: HDFS (Hadoop Distributed File System)
  - ✔ A programming model and execution framework: Hadoop MapReduce

MapReduce

- Allows simply expressing many parallel/distributed computational algorithms

Hadoop MapReduce

Key/Value pairs

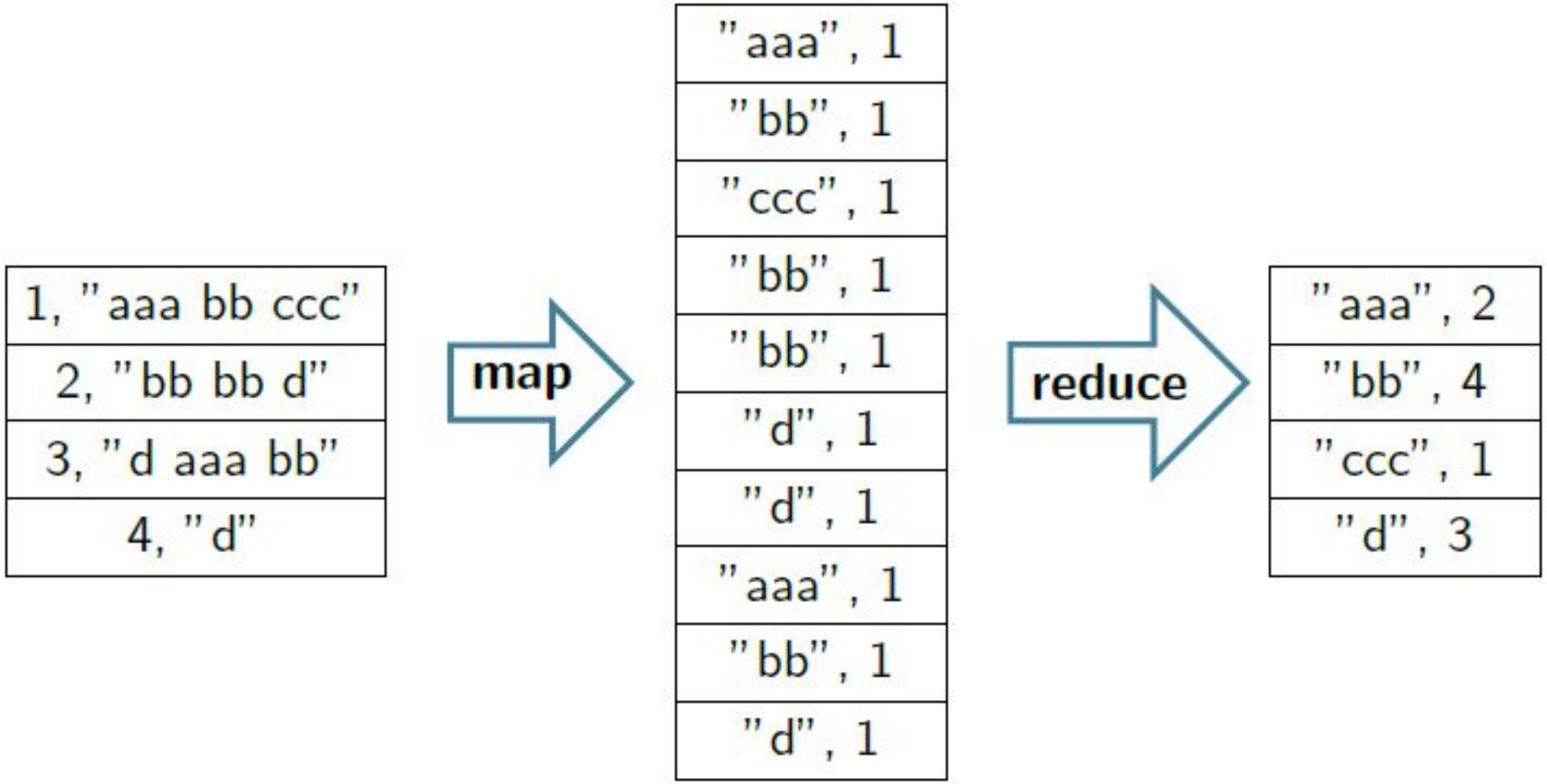
- MapReduce manipulate sets of Key/Value pairs
- Keys and values can be of any types

Functions to apply

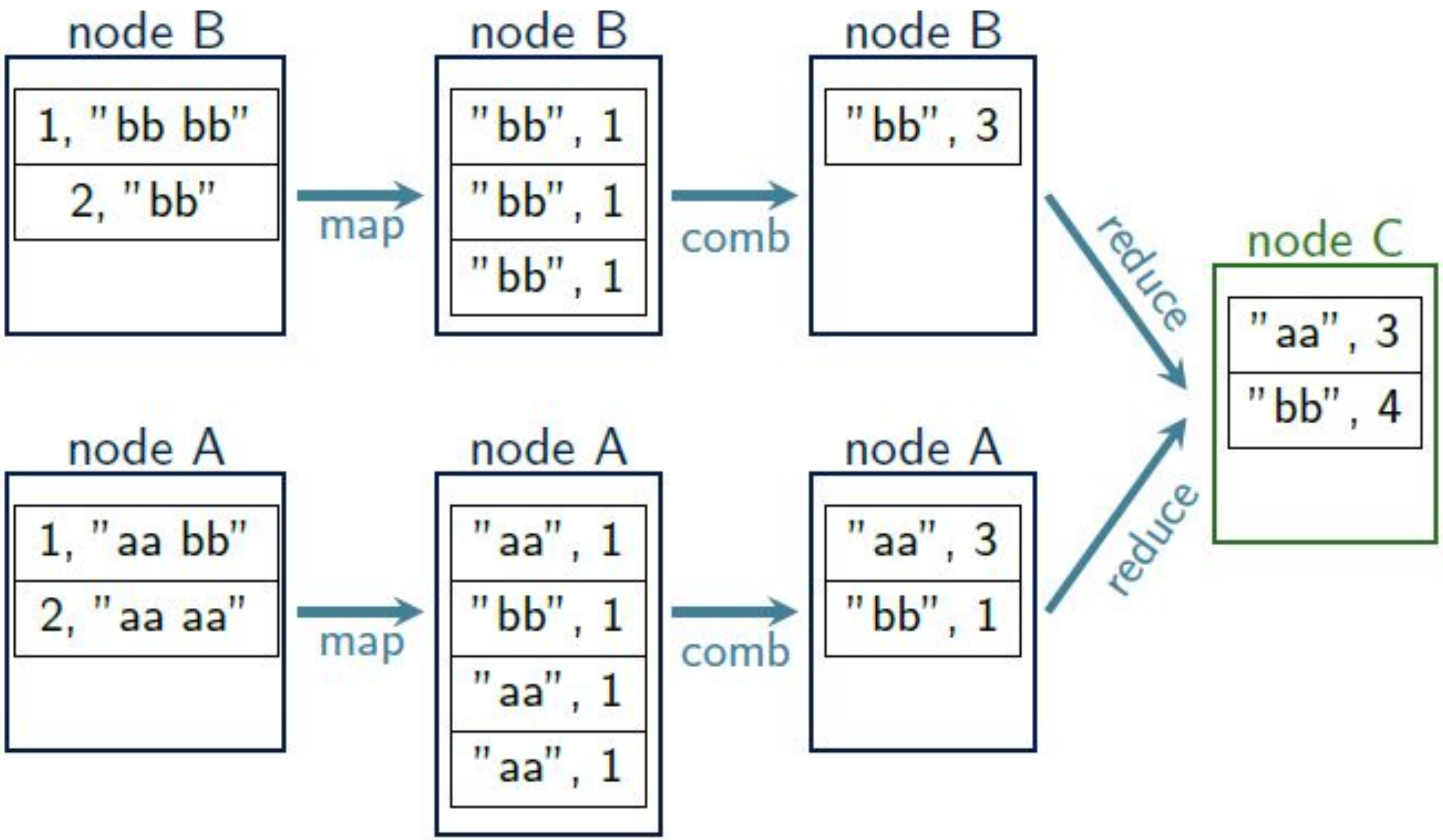
- The user defines the functions to apply
- In Map, the function is applied independently to each pair
- In Reduce, the function is applied to all values with the same key

A First MapReduce program

Word Count



Distributed execution of Word Count



Example: Web index

Description

Construct an index of the pages in which a word appears.

- Input: A set of web pages
  - ✔ Pairs $<$ URL, content of the page $>$
- Output: A set of pairs $<$ word, set of URLs $>$

Hadoop Distributed File System (HDFS)

Main ideas

- Running on a cluster of commodity servers
  - ✔ Each node has a local disk
  - ✔ A node may fail at any time
- The content of les is stored on the disks of the nodes
  - ✔ Partitioning: Files are partitioned into blocks that can be stored in different Datanodes
  - ✔ Replication: Each block is replicated in multiple Datanodes
    - Default replication degree: 3
  - ✔ A Namenode regulates access to files by clients
    - Master-worker architecture

Hadoop MapReduce issues

talent sprint

- Only some type of computations supported
- Not easy to program
- Missing abstractions for advanced workflows
- Streamed data, interactive data, DAG workflows, heterogeneous tasks - difficult

Apache Spark: the concept

- Written in Scala
- Scala code compiles into JVM bytecode
- Focused on in-memory processing
- In memory, 100x faster than MapReduce
- 10x faster on disk
- Allows for a wide range of workflows. More flexible and easy in programming
- Leverages a lot of Hadoop infrastructure: Mesos, YARN, HDFS

Apache Spark

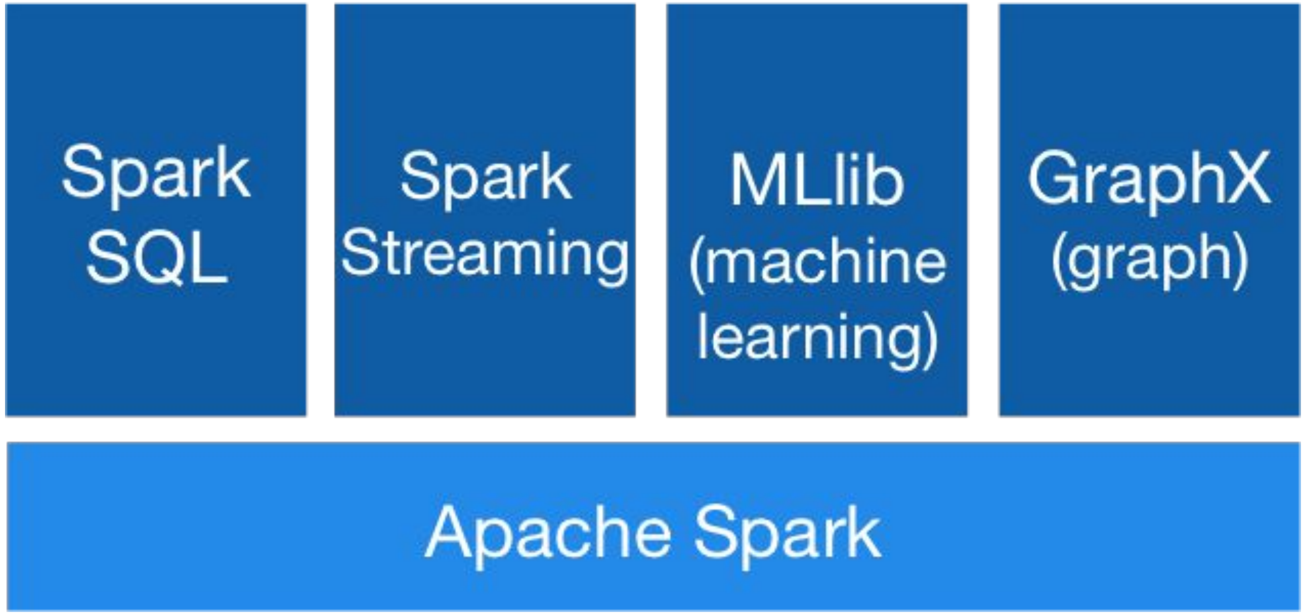
- History
  - UC Berkeley's AMPLab in 2009
  - Donated to Apache 2013. Now among the most vibrant Apache projects in version 2.4
- Industrial ecosystem
  - Apache Software Foundation: maintains the code base
  - Databricks: provides commercial support and more (Unified Analytics Platform)
  - Hortonworks: employs Hadoop creators and provides services and software around Hadoop; HDP (NASDAQ)
  - Cloudera: originally a Hadoop distribution, incorporated 2008, now trades as CLDR at NYSE since 2017

Apache Spark: key terms

- DAG, direct acyclic graph
- RDD, resilient data set
- SparkContext
- Mesos, YARN
- Workers, Executors

Apache Spark EcoSystem

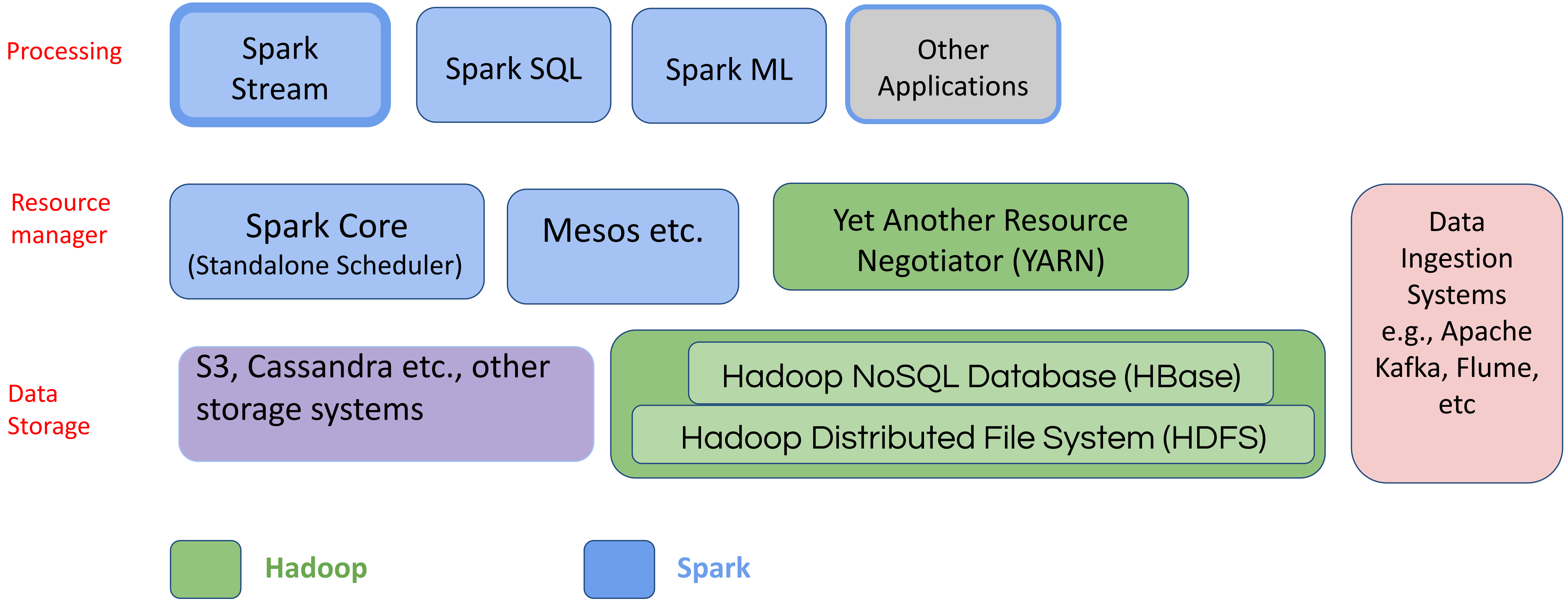
- **Apache Spark**
  - RDDs
- **Spark SQL**
  - Once known as “Shark”
  - before completely integrated
  - into Spark
  - For SQL, structured and
  - semi-structured data
  - processing
- **Spark Streaming**
  - Processing of live data
  - streams
- **MLlib/ML**
  - Machine Learning Algorithms



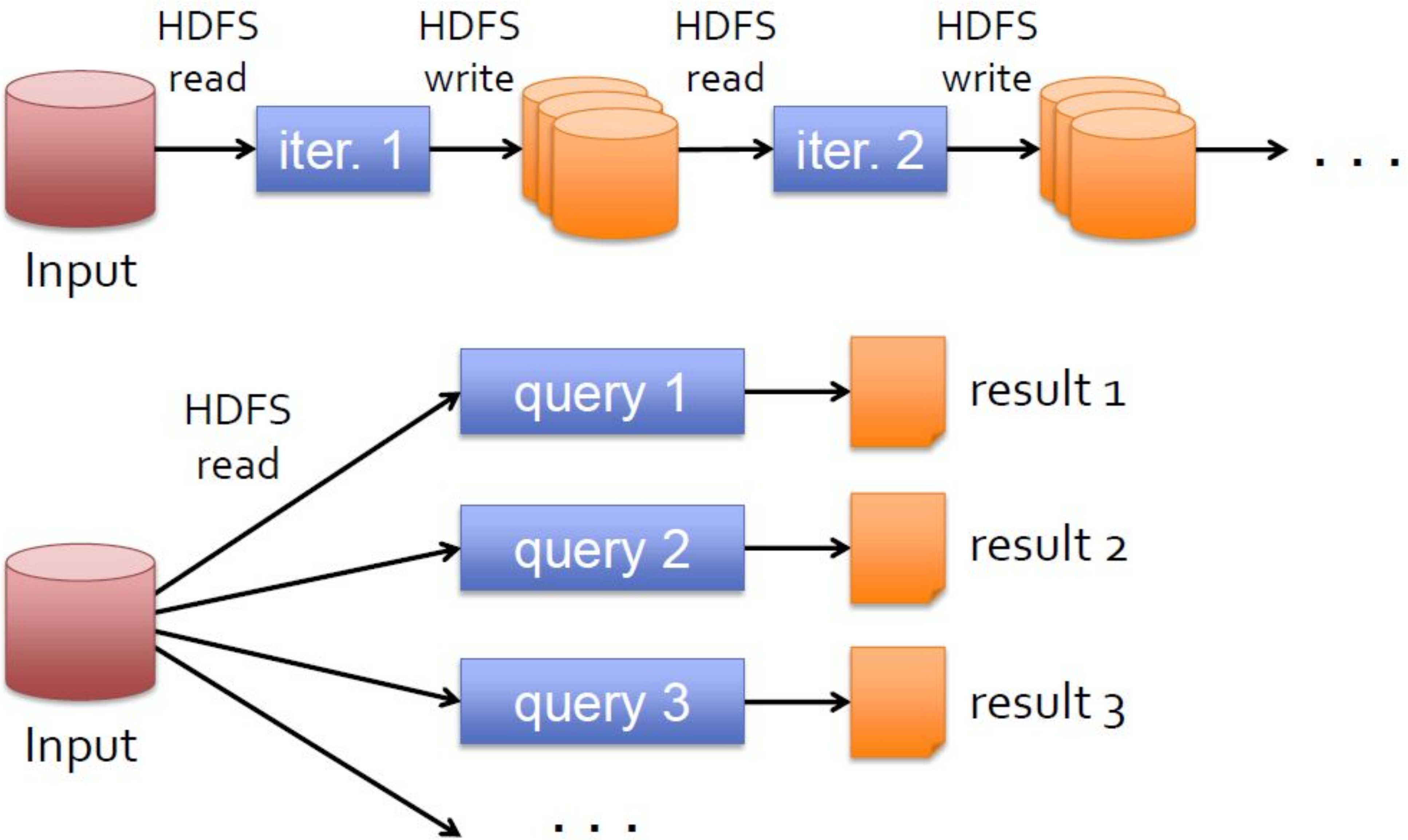
Apache Spark, Apache Spark Ecosystem
http://spark.apache.org/images/spark-stack.png

Apache Spark

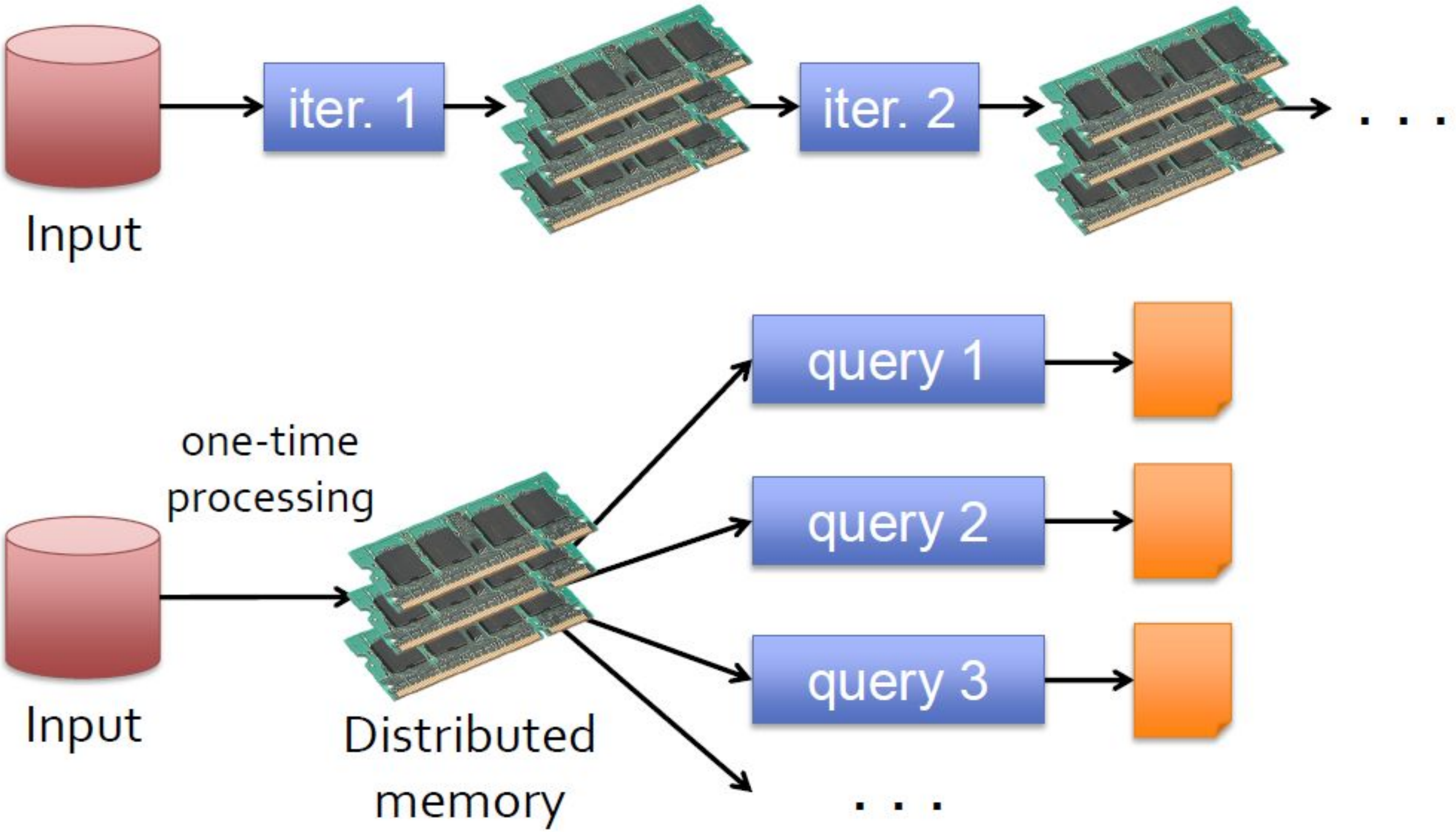
** Spark can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos or YARN)



Data Sharing in MapReduce



Data Sharing in Spark



~10 × faster than network and disk

MapReduce vs. Spark (Performance) (Cont.)

		Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Dayton Gray 100 TB sorting results
https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

Running Spark Jobs

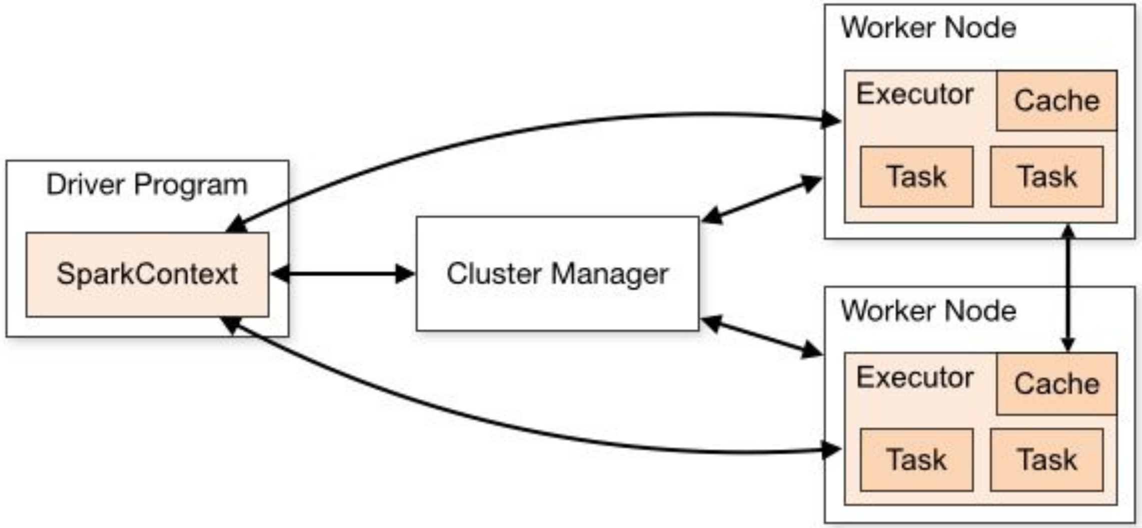
- **Shell**
  - Shell for running Scala Code
    - $ spark-shell
  - Shell for running Python Code
    - $ pyspark
  - Shell for running R Code
    - $ sparkR

- **Submitting (Java, Scala, Python, R)**
  - $ spark-submit --class {MAIN_CLASS} [OPTIONS] {PATH_TO_FILE} {ARG$_0$} {ARG$_1$}… {ARG$_N$}

SparkContext

- A Spark program first creates a SparkContext object
  - Spark Shell automatically creates a SparkContext as the **sc** variable
- Tells spark how and where to access a cluster
- Use SparkContext to create RDDs
- Documentation
  - https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext

Spark Architecture



http://spark.apache.org/docs/latest/img/cluster-overview.png

RDDs

- Primary abstraction object used by Apache Spark
- **Resilient Distributed Dataset**
  - Fault-tolerant
  - Collection of elements that can be operated on in parallel
  - Distributed collection of data from any source
- **Contained in an RDD:**
  - Set of dependencies on parent RDDs
    - Lineage (Directed Acyclic Graph – DAG)
  - Set of partitions
    - Atomic pieces of a dataset
  - A function for computing the RDD based on its parents
  - Metadata about its partitioning scheme and data placement

RDDs (Cont.)

- RDDs are Immutable
  - Allows for more effective fault tolerance
- Intended to support abstract datasets while also maintain
- MapReduce properties like automatic fault tolerance, locality-aware
- scheduling and scalability.

RDDs (Cont.)

- RDD API built to resemble the Scala Collections API
- Programming Guide
  - http://spark.apache.org/docs/latest/quick-start.html
- Lazy Evaluation
- Waits for action to be called before distributing actions to worker node

Word Count Example

Scala

```scala
val textFile = sc.textFile("/path/to/file.txt")
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
counts.saveAsTextFile("/path/to/output")
```

Python

```python
text_file = sc.textFile("/path/to/file.txt")
counts = text_file.flatMap(lambda line: line.split(""))\
                  .map(lambda word: (word, 1))\
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/path/to/output")
```

Word Count Example (Java 8)

```
JavaRDD<String> textFile = sc.textFile("/path/to/file.txt");
JavaPairRDD<String, Integer> counts = lines
            .flatMap(line -> Arrays.asList(line.split(" ")));
            .mapToPair(w -> new Tuple2<String, Integer>(w, 1))
            .reduceByKey((a, b) -> a + b);

counts.saveAsTextFile("/path/to/output");
```

Fault Tolerance

- RDDs contain lineage graphs (coarse grained updates/transformations) to help it rebuild partitions that were lost
- Only the lost partitions of an RDD need to be recomputed upon failure.
- They can be recomputed in parallel on different nodes without having to roll back the entire app
- Also lets a system tolerate slow nodes (stragglers) by running a backup copy of the troubled task.
- Original process on straggling node will be killed when new process is complete
- Cached/Check pointed partitions are also used to re-compute lost partitions if available in shared memory

Spark Streaming

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

http://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark SQL

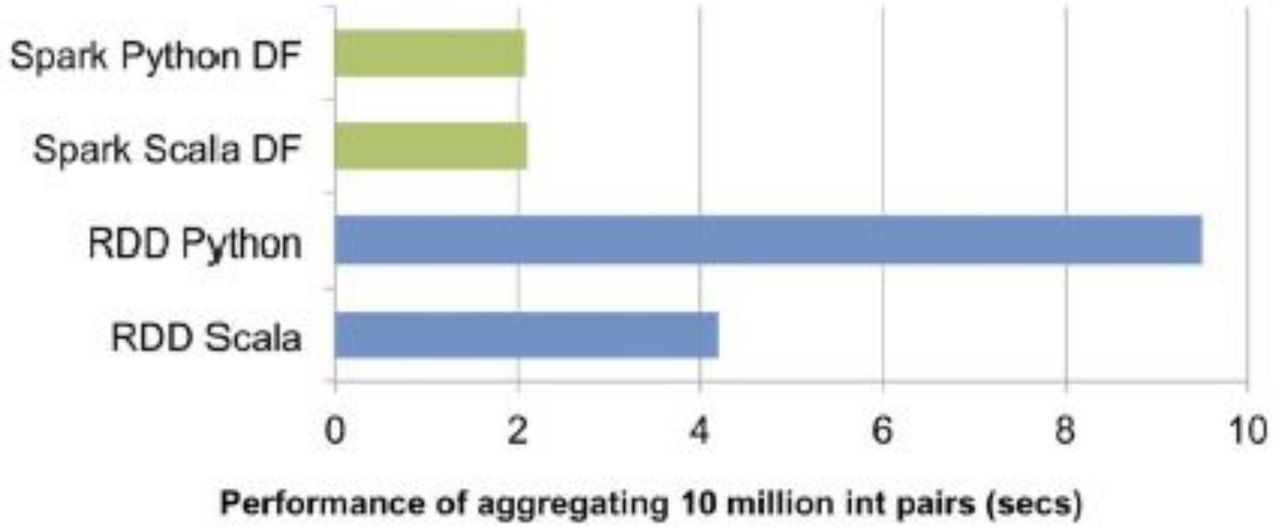
- Distributed in-memory computation on massive scale (Just like Spark!)
- Can use all data sources that Spark supports natively:
  - Can import data from RDDs
  - JSON/CSV files can be loaded with inferred schema
  - Parquet files - Column-based storage format
    - Supported by many Apache systems (big surprise!)
  - Hive Table import
    - A popular data warehousing platform by Apache

Spark SQL

- SQL using Spark as a “Database”
    - Spark SQL is best optimized for retrieving data
    - Don’t UPDATE, INSERT, or DELETE
- Optimization handled by a newer optimization engine, **Catalyst**
    - Creates physical execution plan and compiles directly to JVM bytecode
- Can function as a compatibility layer for firms that use RDBMS systems

Spark DataFrames

- Dataset organized into named columns
- Similar to structure as Dataframes in Python (i.e. Pandas) or R
- Lazily evaluated like normal RDDs
- Tends to be more performant than raw RDD operations



Pandas DataFrame

Does in-memory computing, but:

- Not scalable by itself.
- Not fault tolerant.

```
import pandas as pd

df = pd.read_csv("/path/to/data.json")

df
```

```
      first_name last_name  age  preTestScore     postTestScore
0     Jason Miller     42   4    25,000
1     Molly Jacobson   52   24   94,000
2     Tina  .     36   31   57
3     Jake  Milner     24   .    62
4     Amy   Cooze 73   .    70
```

Spark DataFrames

- When to prefer RDDs over DataFrames:
  - Need low-level access to data
  - Data is mostly unstructured or schema less
- When to prefer DataFrames over RDDs:
  - Operations on structured data
  - If higher-level abstractions are useful (i.e. joins, aggregation, etc.)
  - High-performance is desired, and workload fits within DataFrame APIs
    - Catalyst optimization makes DataFrames more performant on average

Spark Datasets

- Strongly-typed DataFrames
- Only accessible in Spark2+ using Scala
- Operations on DataFrames are all statically typed, so you catch type errors at compile-time

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Data Ingest (RDD)

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
users_rdd = sc.parallelize([[1, 'Alice', 10], [2, 'Bob', 8]])
users = sqlContext.createDataFrame(users_rdd,['id', 'name', 'num_posts'])
users.printSchema()
```

```
#root
# |-- id: long (nullable = true)
# |-- name: string (nullable = true)
# |-- num_posts: long (nullable = true)
```

Data Ingest (JSON)

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
users = sqlContext.read.json("/path/to/users.json")
users.printSchema()


# root
# |-- id: long (nullable = true)
# |-- name: string (nullable = true)
# |-- num_posts: long (nullable = true)
```

SQL API

```
# Register users DataFrame as a table called "users"
users.createOrReplaceTempView( 'users')
# Query the table
sqlContext.sql(
'SELECT * FROM users WHERE name="Bob"'
).collect()
# [Row(id=2, name='Bob', num_posts=8)]
```

DataFrame API

```
# Same query can be done with DataFrame API
users.filter(users.name=='Bob').collect()
# [Row(id=2, name='Bob', num_posts=8)]
users.filter(users.name=='Eve').select('num_posts').collect()
# [Row(num_posts=10)]
```

Thanks!!

Questions?