

BIG DIVE

DATA SCIENCE & ANALYTICS

Designed for

INTESA  SANPAOLO

Hadoop Training

Powered by



About us



Agile Lab is an **Italian company** that realizes software systems using the most advanced **Big Data technologies** and **machine learning algorithms** for all the companies which want to manage and generate **real value from their data** even if that data are "big" or "small".

Some people address us as a «Fintech» company as we have made many experiences in that sector, but actually we operate in several business sectors always with the same common approach: **bringing innovation and driving forward our clients** through technology, competence, positive and "agile" thinking.



«We are entering in a world where data may be more important than software»

Quote: Tim O'Reilly

Approach



Agile Lab understands rapidly the key features of the Customer's market which has become even more demanding, quick and complex.

We have a deeper understanding of the **real big data cases for production environments**, that allows us to think in a cross-pollination logic and to offer ideas, new methodologies and technology approaches to cross-sector Customers, without fear for any challenge.



«Knowledge speaks, but wisdom listens»

Quote: Jimi Hendrix

R&D



Agile Lab's spirit is based on **research, innovation and development** but always with a very strong focus on the understanding of the Customer's Business.

Our team is made up of specialists and we work constantly with Research Institutes and Universities to be always ahead with new technologies and innovative analysis methodologies.



«Think about the future, think about what you could do, don't fear anything»

Quote: Rita Levi Montalcini

What we do



MACHINE LEARNING
Engine



INFRASTRUCTURE
for Big Data



«I'm not a speed reader, I'm a speed understander»

Quote: Isaac Asimov



Introduction to Big Data

- 1. When did it started?**
2. What is Big Data
3. Why Big Data
4. Data Value
5. NoSQL
6. Main technologies

When did it started?

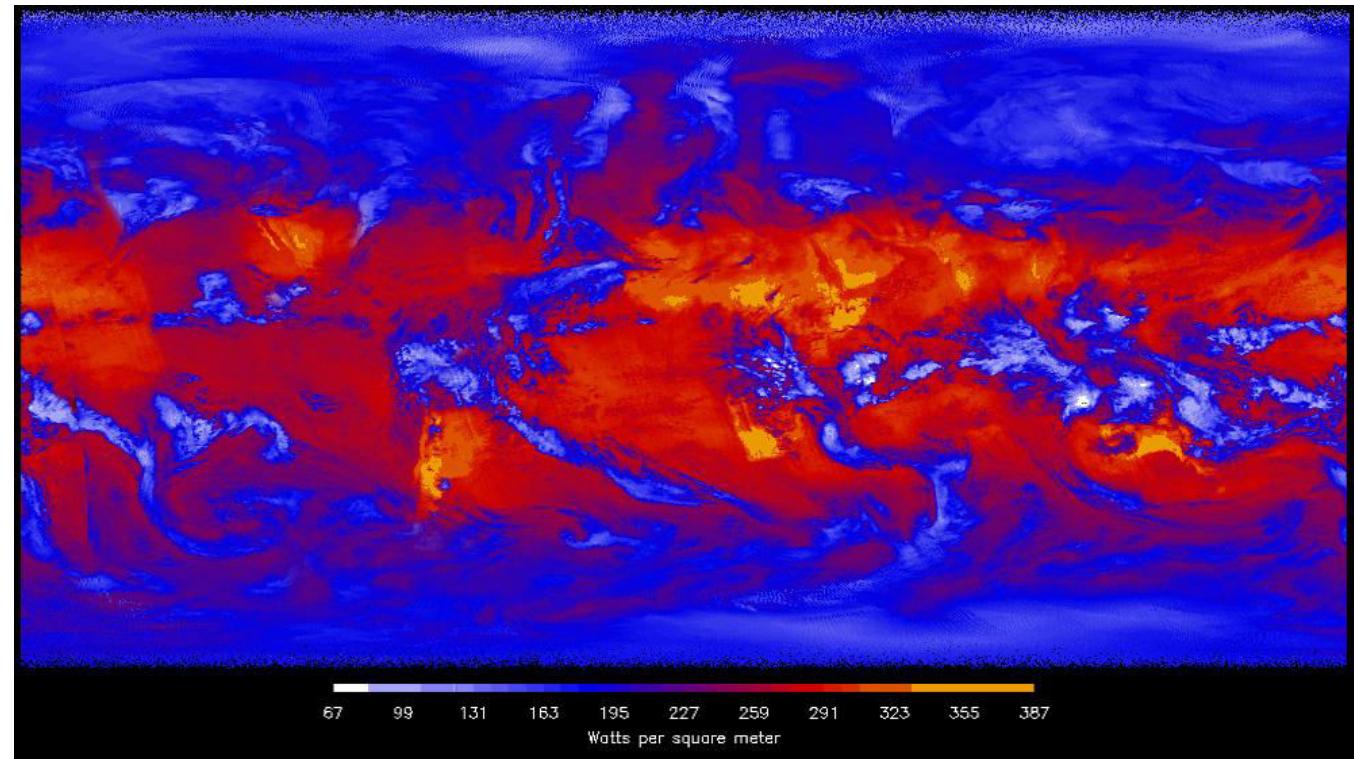


«Visualization provides an interesting challenge for computer systems:

data sets are generally quite large, taxing the capacities of main memory, local disk, and even remote disk.

We call this the problem of big data.»

Michael Cox and David Ellsworth @ NASA
July 1997



Source: [Application-Controlled Demand Paging for Out-of-Core Visualization](#).

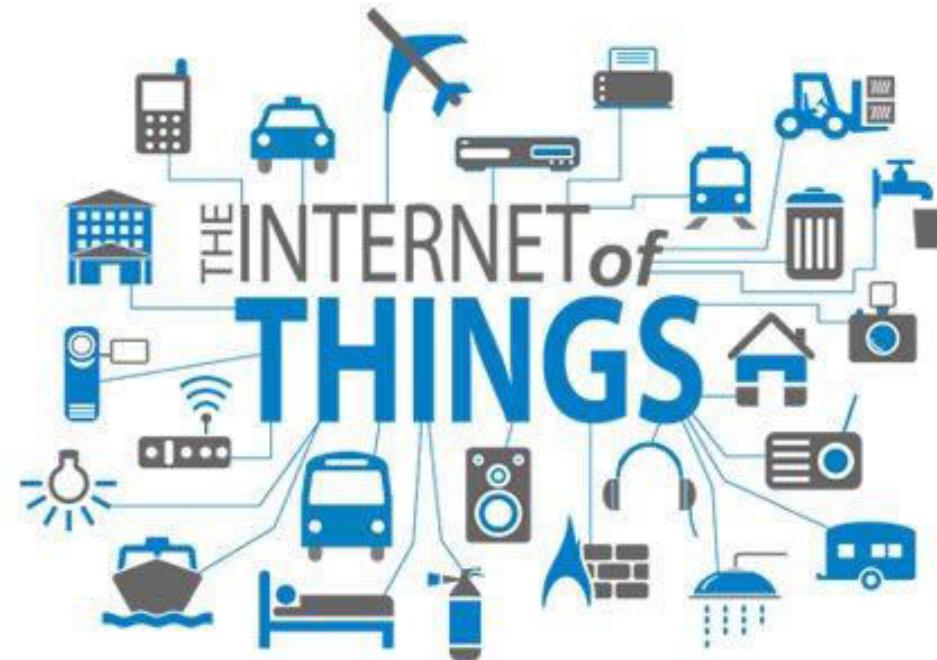
New data sources: IoT



“Internet of things” terms used for the first time in 1999

«Today computers - and, therefore, the Internet - are almost wholly dependent on human beings for information. [...] If we had computers that knew everything there was to know about things - using data they gathered without any help from us - we would be able to track and count everything, and greatly reduce waste, loss and cost.»

Kevin Ashton, Co-Founder of the Auto-ID Center @ MIT



Source: [RFID Journal](#).

Data growing

CSC

THE STORY OF BIG DATA

EXPLORE ALL
THREE SECTIONS ▶

1 > 2 > 3



1

THE RAPID GROWTH OF GLOBAL DATA

The production of data is expanding at an astonishing pace. Experts now point to a 4300% increase in annual data generation by 2020. Drivers include the switch from analog to digital technologies and the rapid increase in data generation by individuals and corporations alike.

- Size of Total Data
- Enterprise Managed Data
- Enterprise Created Data



WHAT IS A
ZETTABYTE? +

DATA PRODUCTION WILL BE
44 TIMES GREATER
IN 2020 THAN IT WAS IN 2009

More than 70% of the digital universe is generated by individuals. But enterprises have responsibility for the storage, protection and management of 80% of it.*



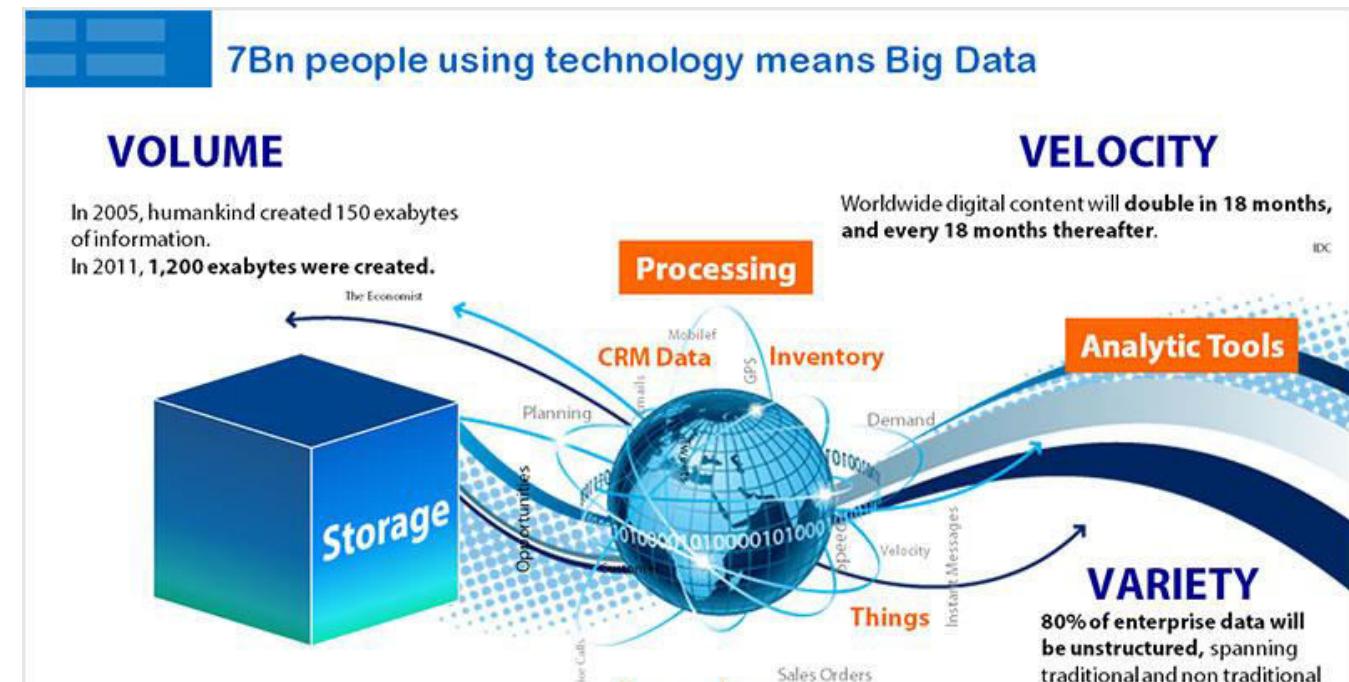
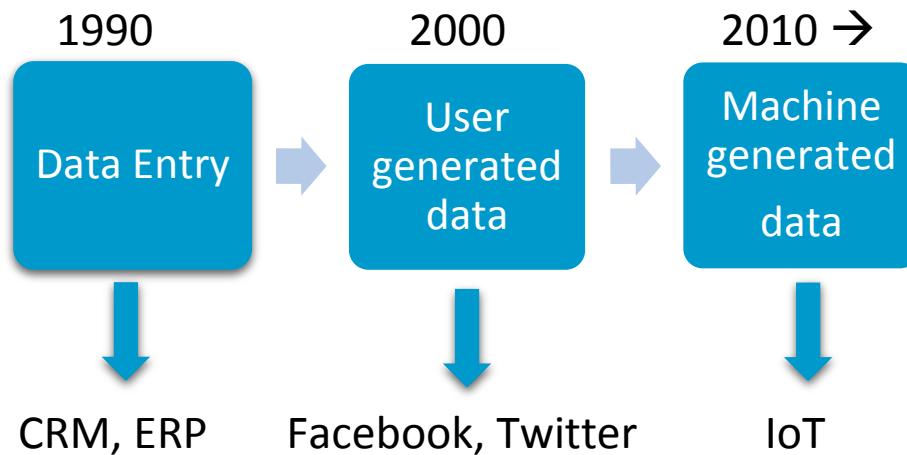
Introduction to Big Data

1. When did it started?
2. **What is Big Data**
3. Why Big Data
4. Data Value
5. NoSQL
6. Main technologies

What is Big Data



90% of world data has been generated in the last two years



What are big data systems?



Applications involving the "three V's"

- Volume: gigabytes, growing to terabytes and beyond
- Velocity: sensor data, click streams, financial transactions
- Variety: data must be ingested from many different formats

Characteristics requiring

- multi-region availability
- very fast and reliable response
- no single point of failure

Why not relational data?



Relational model provides

- Normalized table schema
- Cross table joins
- ACID compliance

But, at very high cost

- Big data table joins – billions of rows, or more – require massive overhead
- Sharding tables across systems is complex and fragile

Modern applications have different priorities

- Needs for speed and availability trump "always on" consistency
- Commodity server racks trump massive high-end systems
- Real world need for transactional guarantees is limited



Introduction to Big Data

1. When did it started?
2. What is Big Data
- 3. Why Big Data**
4. Data Value
5. NoSQL
6. Main technologies

Why Big Data



Cost reduction.

Big data technologies such as Hadoop and cloud-based analytics bring significant cost advantages when it comes to storing large amounts of data – plus they can identify more efficient ways of doing business.

Why Big Data



Faster, better decision making.

With the speed of Hadoop and in-memory analytics, combined with the ability to analyze new sources of data, businesses are able to analyze information immediately – and make decisions based on what they've learned.

Why Big Data



New products and services.

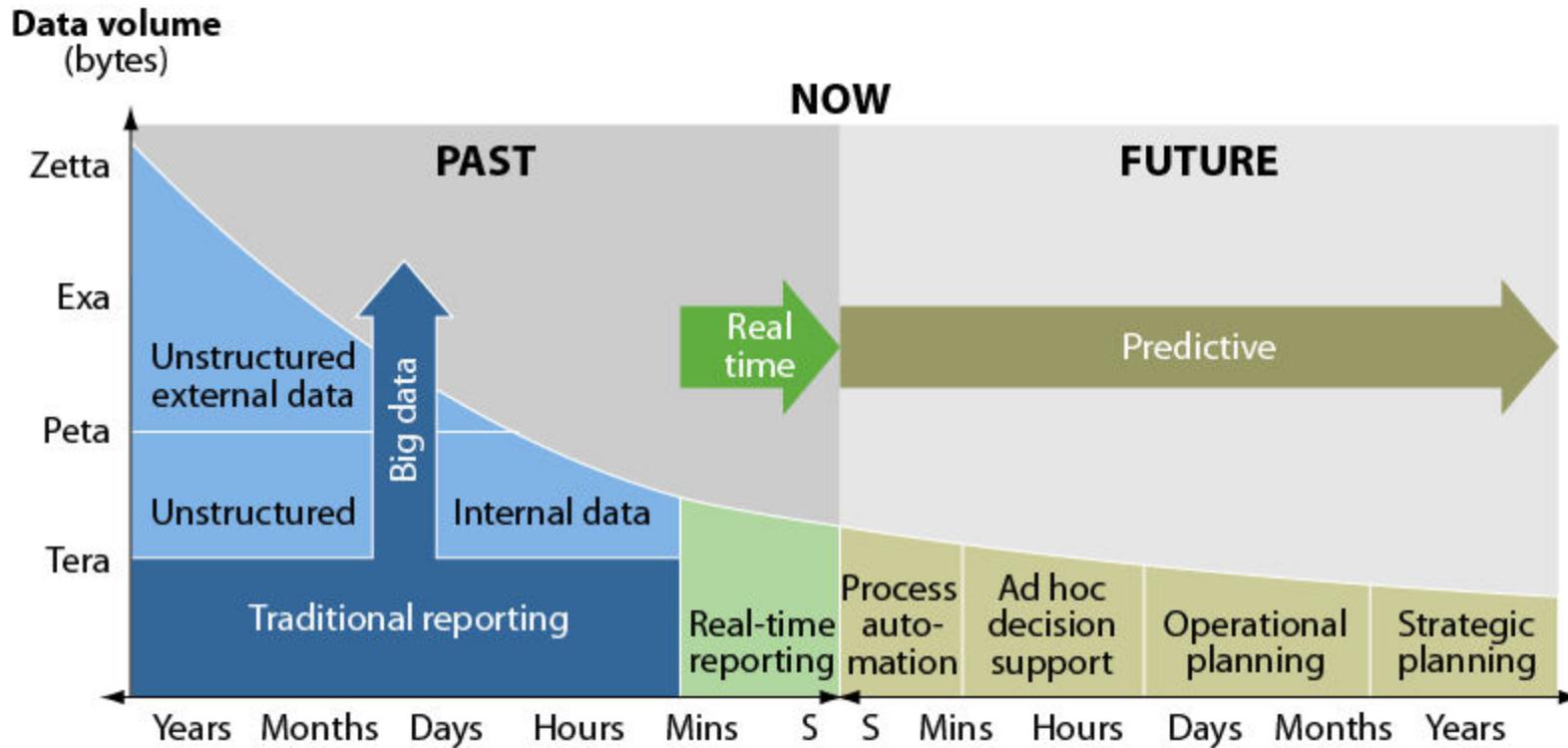
With the ability to gauge customer needs and satisfaction through analytics comes the power to give customers what they want. Davenport points out that with big data analytics, more companies are creating new products to meet customers' needs.



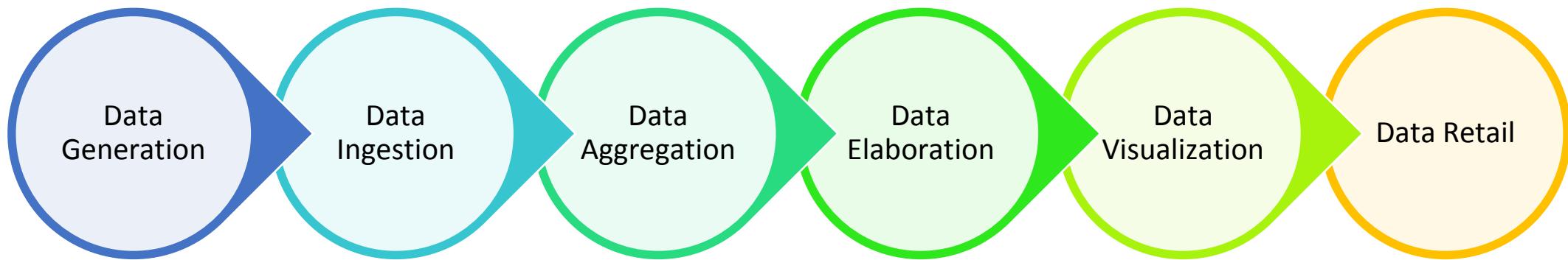
Introduction to Big Data

1. When did it started?
2. What is Big Data
3. Why Big Data
- 4. Data Value**
5. NoSQL
6. Main technologies

Data Value



Data Value Chain





Introduction to Big Data

1. When did it started?
2. What is Big Data
3. Why Big Data
4. Data Value
5. **NoSQL**
6. Main technologies

NoSQL landscape



Four broad classes of non-relational database

- Graph: data elements each relate to n others in a graph/network
- Key-Value: keys map to arbitrary values of any data type
- Document: document sets (JSON) queryable in whole or part
- Column Family: keys mapped to sets of n-number of typed columns

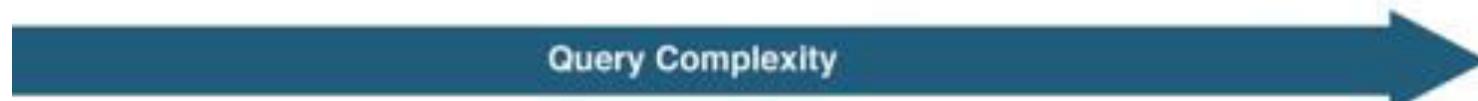
Three key factors help navigate the landscape

- Consistency: do you get identical results, regardless which node is queried?
- Availability: can the cluster respond to very high write and read volumes?
- Partition Tolerance: is the cluster still available when part of it goes dark?

NoSQL landscape



Simple Key-Value Reads / Writes	Sparse Column Families (~Tables)	Document Level Reads / Writes	Transactional Reads / Writes	
No Analytics	No Analytics	No Joins, Some Aggregates	Limited Joins, Some Aggregates	Scalable Joins and Aggregates

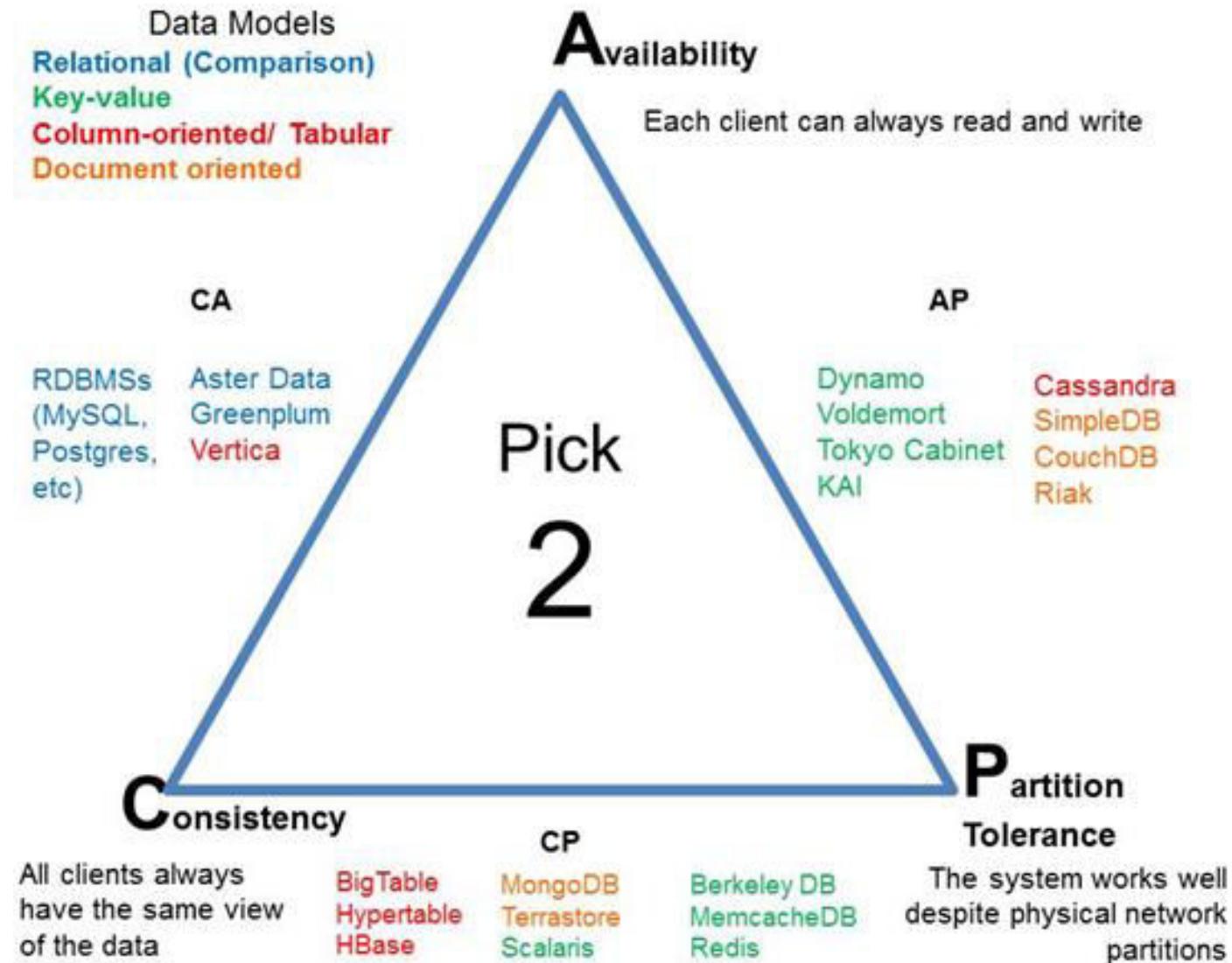


NoSQL features



- Shared-nothing architecture
- Non-locking concurrency control: workload separation
- Much higher per-node performance than SQL-based
- Scalable replication and distribution
- Schema-less Data Model
- Mostly query and Few updates

CAP Theoreme



BASE: not ACID



Basically **A**vailable: guarantee availability

SoftState: state of the system may change over time – because of eventual consistency

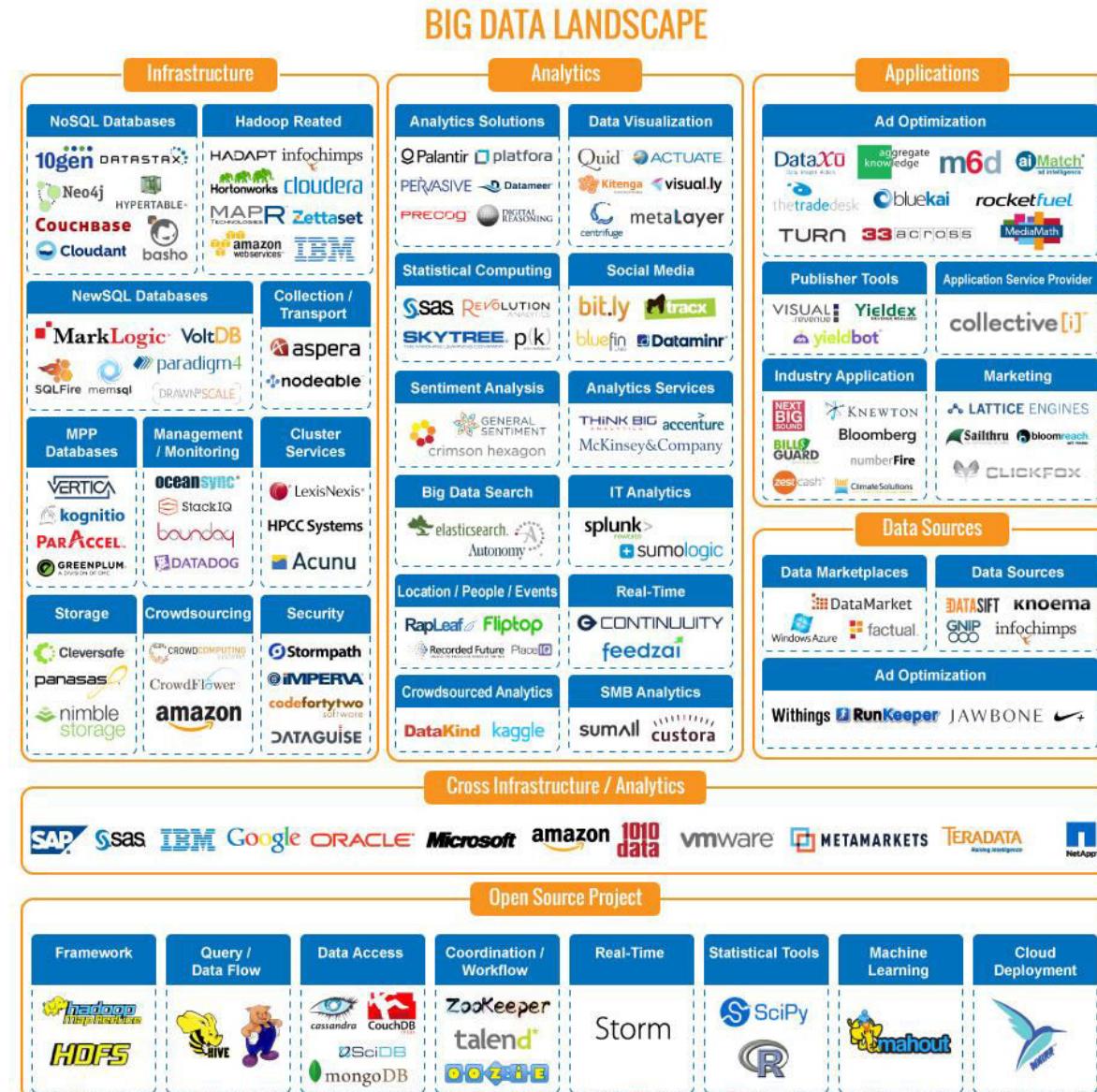
Eventual Consistency: will become consistent over time



Introduction to Big Data

1. When did it started?
2. What is Big Data
3. Why Big Data
4. Data Value
5. NoSQL
- 6. Main technologies**

Big Data Technologies



Summary



- "Big data" can bring high value to enterprises if correctly processed.
- Velocity is a key factor in Big Data processing for real time decisions.
- There are a lot of technologies that cover various aspects of the data lifecycle.
- Hadoop related technologies are key actors in the actual big data scenario.

Hadoop Training

Powered by





Hadoop in an Enterprise

- 1. How to introduce it**
2. Common errors
3. Required skills and teams
4. Enterprise architecture

Hadoop in an Enterprise



- Some companies start with a data science project, show great ROI and then expand it to a full data lake
- Other companies start by off-loading some existing functions to Hadoop and then add more data and expand into a data lake
- Yet a third set of companies start with a new operational project on Hadoop and then decide to add analytics
- Finally, some companies build data lake as a central point of governance

Which approach is right for you?

Data Science driving



Find a highly visible problem (**Pilot**) that:

- Can be solved through machine learning or advanced analytics
- Is well defined and well understood
- Can show quick, measurable benefit
- Requires data that the team can easily procure

Then:

- Invest into a small Hadoop cluster
- Bring Data Science consultants
- Quickly produce results and show the value

Then:

- Gain budgets
- Enlarge the cluster and go on

Off-loading



- Choose an expensive functionality in your company
 - Re-implement it in Hadoop trying to leverage its cost-effectiveness
-
- You don't need business sponsor because cost is coming out entirely out of IT budget
 - Success is entirely up to IT
 - Off-loading should be transparent to the business users

New Operational Project



- Try to support new operational project such as IoT or Social Media analytics
- The advantage of a new project is that it creates new visible value for the company
- But it requires additional budget
- A project failure even if it has nothing to do with Hadoop can taint enterprise's view of big data technology and negatively affect its adoption

Some new opportunities with Hadoop:

- Financial Services: portfolio risk analysis, regulations, anti-money laundering
- Healthcare: governance and compliance, medical research, IoT for medical devices
- Manufacturing: IoT smart sensors, quality and preventive maintenance, supply chain
- Education: admissions
- Retail: pricing

Central Point of Governance



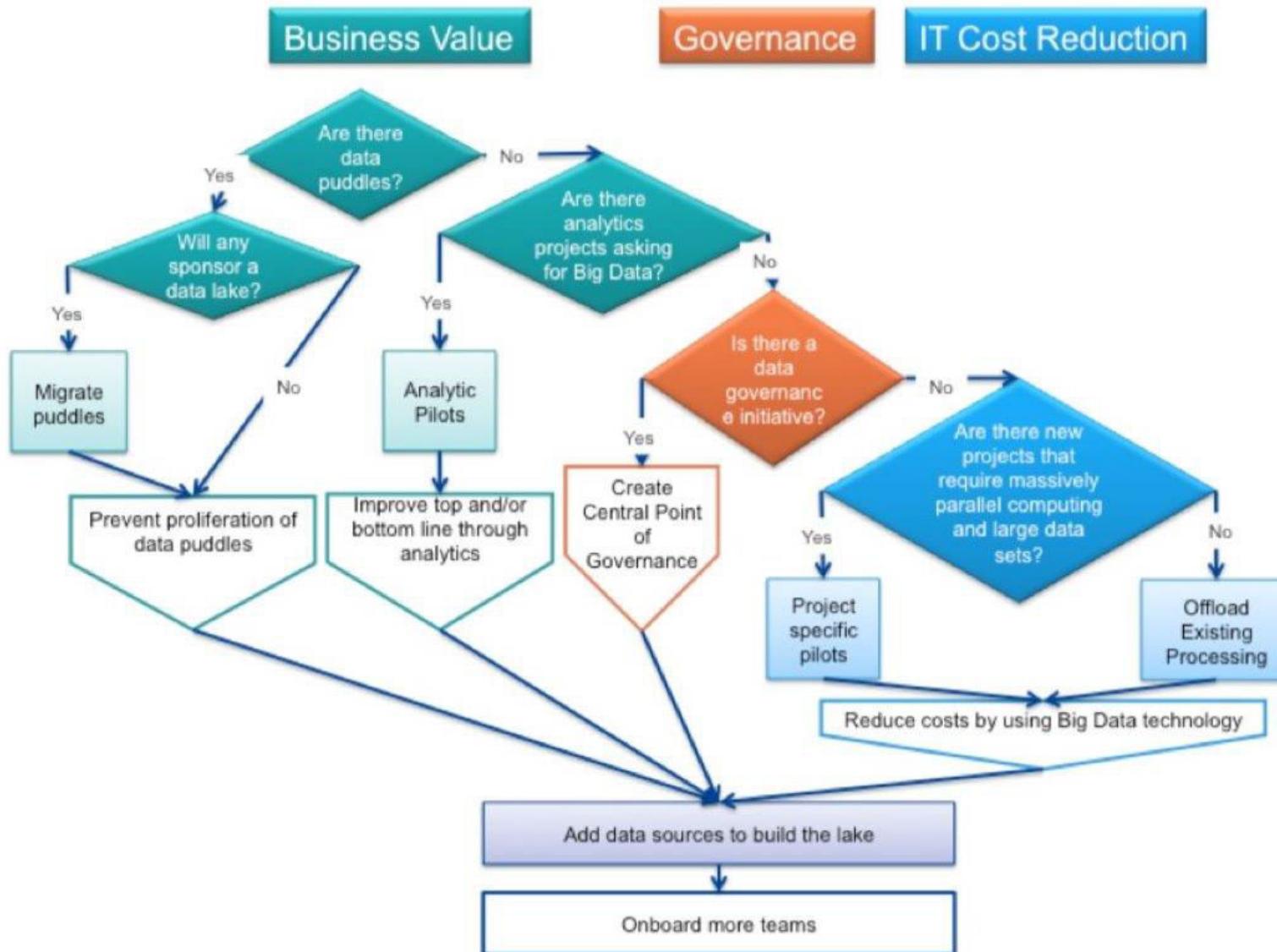
- Secure and managed access to data
- Centralized data quality, data lifecycle management, data lineage
- Really expensive approach
- Ends with creating different zones with different degrees of governance: Raw data → Cleaned Data → Consumer Data

Which is the Right for me ?



- Any one of these approaches can lead to a successful data lake
- It depends on your role, your budget and the allies you can recruit
- Generally, it is easiest to start a data lake by using the budget that you control
- In any case you need a sustainable plan to convince stakeholders start using it for their projects

Which is the Right for me ?





Hadoop in an Enterprise

1. How to introduce it
2. **Common errors**
3. Required skills and teams
4. Enterprise architecture

Common errors



- **Go big (cluster) and hope for the best**
- **Go with Hadoop without skills**
- **Go with Hadoop without planning**
- **Data Silos**

Avoid Data Silos



- Hadoop is not for local business teams or for a single project
- Quick wins hide big problems
- If many business groups have their own «shadow IT» and their own little Hadoop cluster you have a problem
- Proliferation of data silos is bad for data quality and costs
- An Hadoop cluster **needs a permanent skilled IT team** to manage it

You should get ahead of the train and build a data lake, so when business teams decide that they need Hadoop, they have both the compute resources and the data for their projects already available in the data lake.



Hadoop in an Enterprise

1. How to introduce it
2. Common errors
- 3. Required skills and teams**
4. Enterprise architecture

Required Teams



Teams:

- **IT**
 - manage IT infrastructure,
 - Hadoop operations
 - Security
- **Data Ingestion**
 - Build the data catalogue
 - Ingest data
 - Data quality
 - Data lake management
- **Analytics**
 - Develop big data applications
 - Develop API
 - Handle functional requirements
- **DataScience**
 - Prototype algorithms
 - Produce custom reports
 - New Ideas

Required Skills



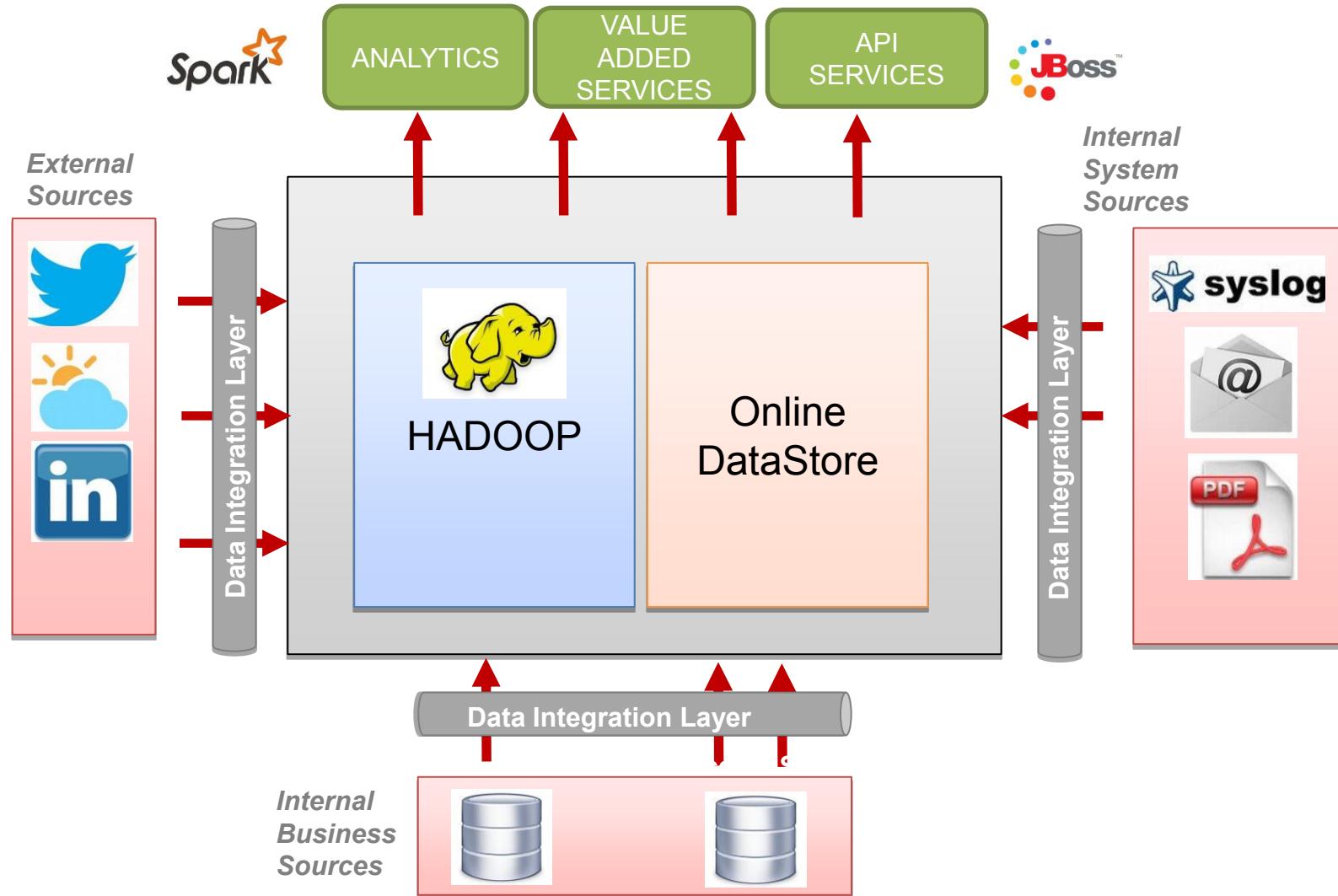
- IT Architect
- Hadoop System Admin
- Software Architect
- Hadoop Developer
- Spark Developer
- Data Scientist with programming skills: Python – R
- Security expert
- Functional Analyst



Hadoop in an Enterprise

1. How to introduce it
2. Common errors
3. Required skills and teams
4. **Enterprise architecture**

Enterprise architecture



Hadoop Training

Powered by





Hadoop

- 1. What is Hadoop**
2. Hadoop Key Properties
3. Brief History
4. Why Hadoop
5. Hadoop vs RDBMS
6. Hadoop Architecture
7. Hadoop components

Hadoop



«A framework that allows reliable, scalable, distributed computing using simple programming models»

- Scales up from single servers to thousands of machines
- Detects and handles failures at the application layer



Hadoop



Hadoop moves the data-processing tasks where the data are actually stored rather than OpenMPI-like software

Hadoop



- Several companies offer **distributions** of Hadoop, with commercial support and additional software to manage it and the services than run on top of it
- The three big ones:





Hadoop

1. What is Hadoop
- 2. Hadoop Key Properties**
3. Brief History
4. Why Hadoop
5. Hadoop vs RDBMS
6. Hadoop in an enterprise Env
7. Hadoop Architecture
8. Hadoop components

Scalability



Extreme Scalability: At most enterprises, data only grows and often exponentially. This growth requires more and more compute power to process. Hadoop is designed to keep scaling by simply adding more nodes. It is used at some of the largest clusters in the world at places like Yahoo and Facebook. This ability to keep scaling is often referred to as “scaling out”.

Cost effectiveness



Cost-effectiveness: Hadoop is designed to work with off-the shelf hardware and run on top of Linux and use many free open source projects. This makes it very cost-effective

Modularity



Modularity: Traditional data management systems are monolithic. For example a traditional relational database does not expose its data except through a relational query. Hadoop has been designed to be modular. You can access your data in several ways. This makes Hadoop **extremely attractive as a long-term platform for managing data**

Schema-on-read



Loose schema coupling or schema-on-read: Unlike a traditional relational database, Hadoop does not enforce any sort of schema when the data is written. This allows what's called “frictionless ingest”. We can avoid paying the high price of processing for data we may not need and, potentially, not processing it correctly for the future applications.

Hadoop Target



if we are building a long-term storage and analytics system for our data, we would want it to be – cost effective, highly scalable and available, require minimal work to add data and be extensible to support future technology, applications and projects, then **Hadoop fits the bill beautifully**



Hadoop

1. What is Hadoop
2. Hadoop Key Properties
- 3. Brief History**
4. Why Hadoop
5. Hadoop vs RDBMS
6. Hadoop Architecture
7. Hadoop components

Brief History



Hadoop came from the Google File System paper that was published in October 2003.

This paper spawned another research paper from Google: MapReduce

Development started on the Apache Nutch project, with Doug Cutting, who was working at Yahoo!, named it after his son's toy elephant.

The first version of Hadoop was released in April 2006.

In 2009 Yahoo runs 17 clusters with 24000 machines.

In June 2010 Facebook manages 40 petabytes with Hadoop



Hadoop

1. What is Hadoop
2. Hadoop Key Properties
3. Brief History
- 4. Why Hadoop**
5. Hadoop vs RDBMS
6. Hadoop Architecture
7. Hadoop components

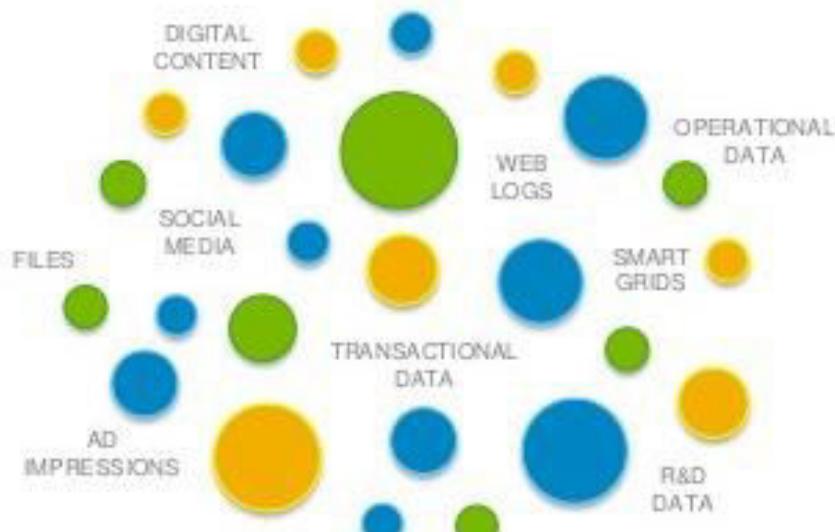
Hadoop



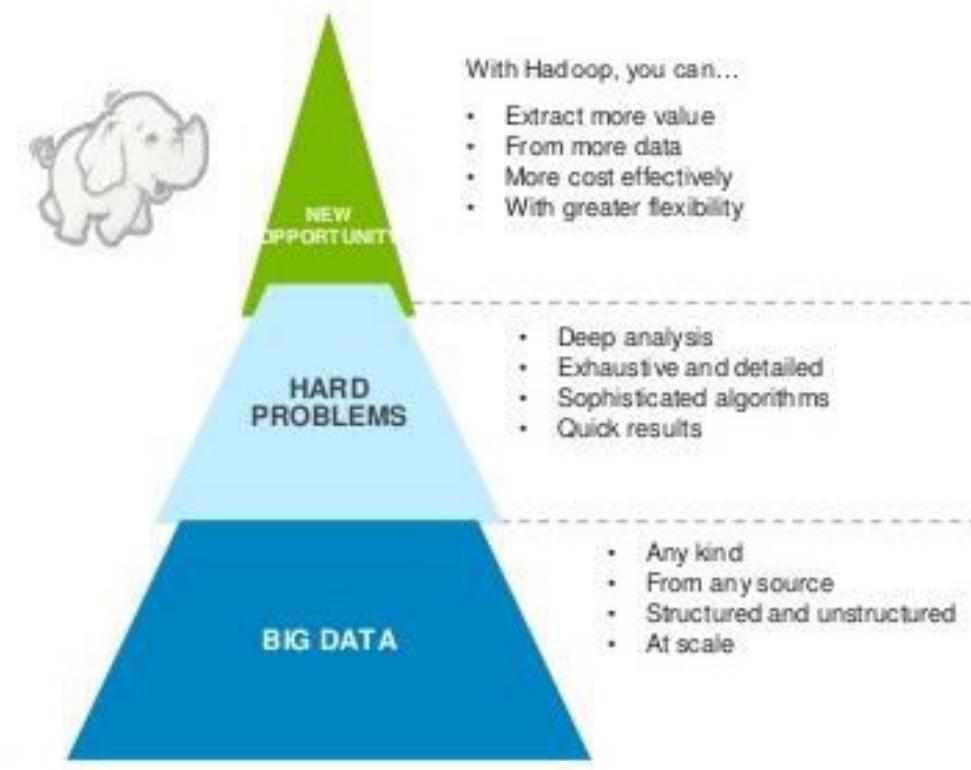
Exploding data volumes & types

LEADS TO

Dramatic changes in
enterprise data management



It's difficult to handle data this diverse at this scale.
Traditional platforms can't keep pace.



Why and when Hadoop ?



- Data is growing, we want to be able to scale-out
- Use cheaper hardware to scale horizontally
- Tolerates failures
- Store all your data from all your systems

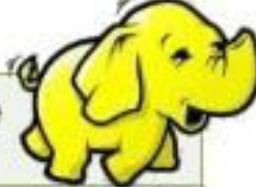


Hadoop

1. What is Hadoop
2. Hadoop Key Properties
3. Brief History
4. Why Hadoop
- 5. Hadoop vs RDBMS**
6. Hadoop Architecture
7. Hadoop components

Hadoop vs RDMS



	Traditional RDBMS 	Hadoop / MapReduce 
Data Size	Gigabytes (Terabytes)	Petabytes and greater
Access	Interactive and Batch	Batch – NOT Interactive
Updates	Read / Write many times	Write once, Read many times
Structure	Static Schema	Dynamic Schema
Integrity	High (ACID)	Low
Scaling	Nonlinear	Linear
Query Response Time	Can be near immediate	Has latency (due to batch processing)

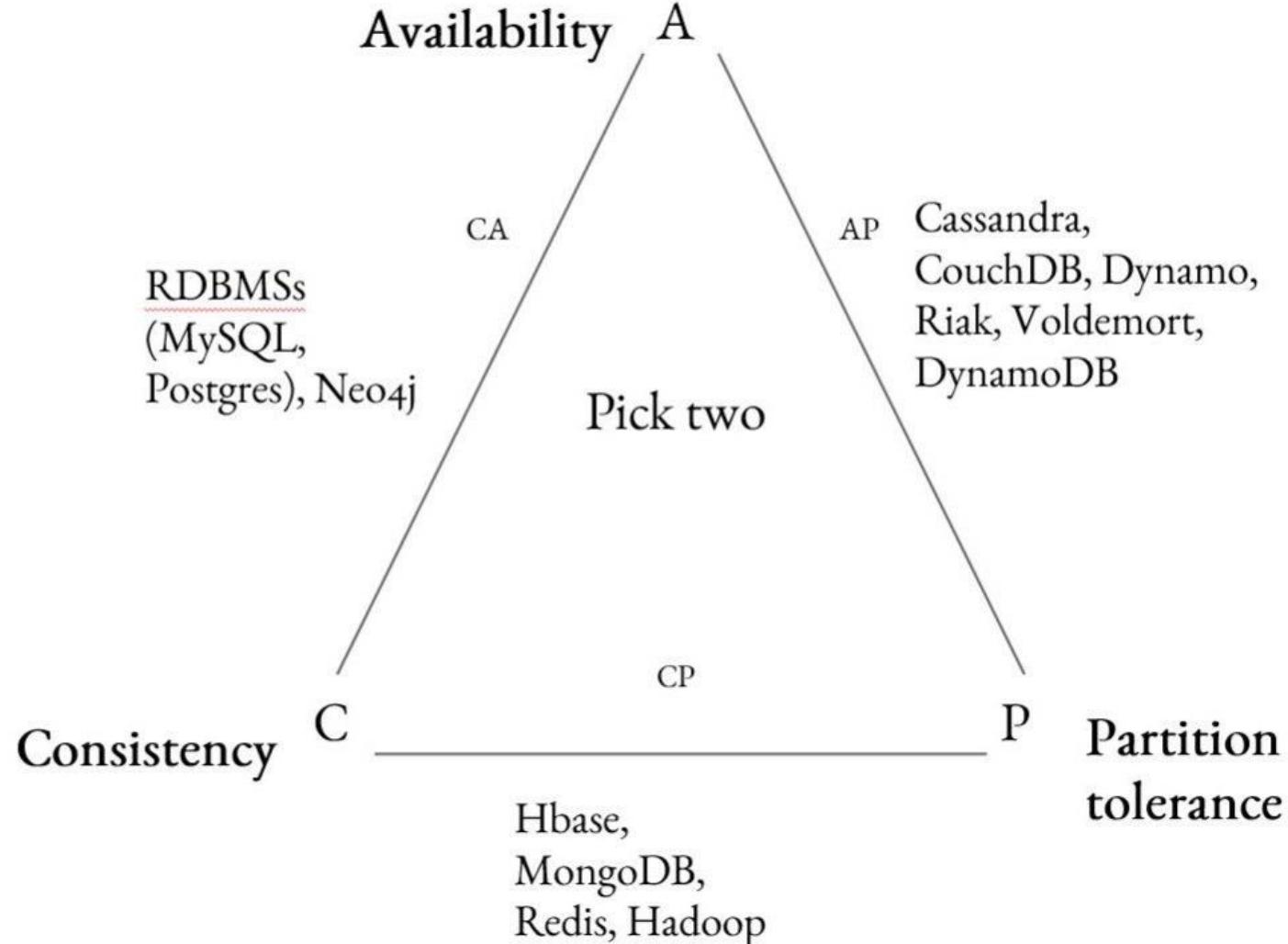
Hadoop vs RDMS



- Hadoop is not a DBMS, but an ecosystem for data management
- HDFS + Hive/Impala is a DBMS
- **Stop comparing them** from the first day, because they are deeply different
- **Use the right tool for the right job**

Features	RDBMS	Hive	Impala
Insert individual records	Yes	No	Yes
Update and delete records	Yes	No	No
Transactions	Yes	No	No
Role-based authorization	Yes	Yes	Yes
Stored procedures	Yes	No	No
Index support	Extensive	Limited	None
Latency	Very Low	High	Low
Data size	Terabytes	Petabytes	Petabytes
Complex data types	No	Yes	No
Storage cost	Very High	Very Low	Very Low

Hadoop vs RDMS - CAP

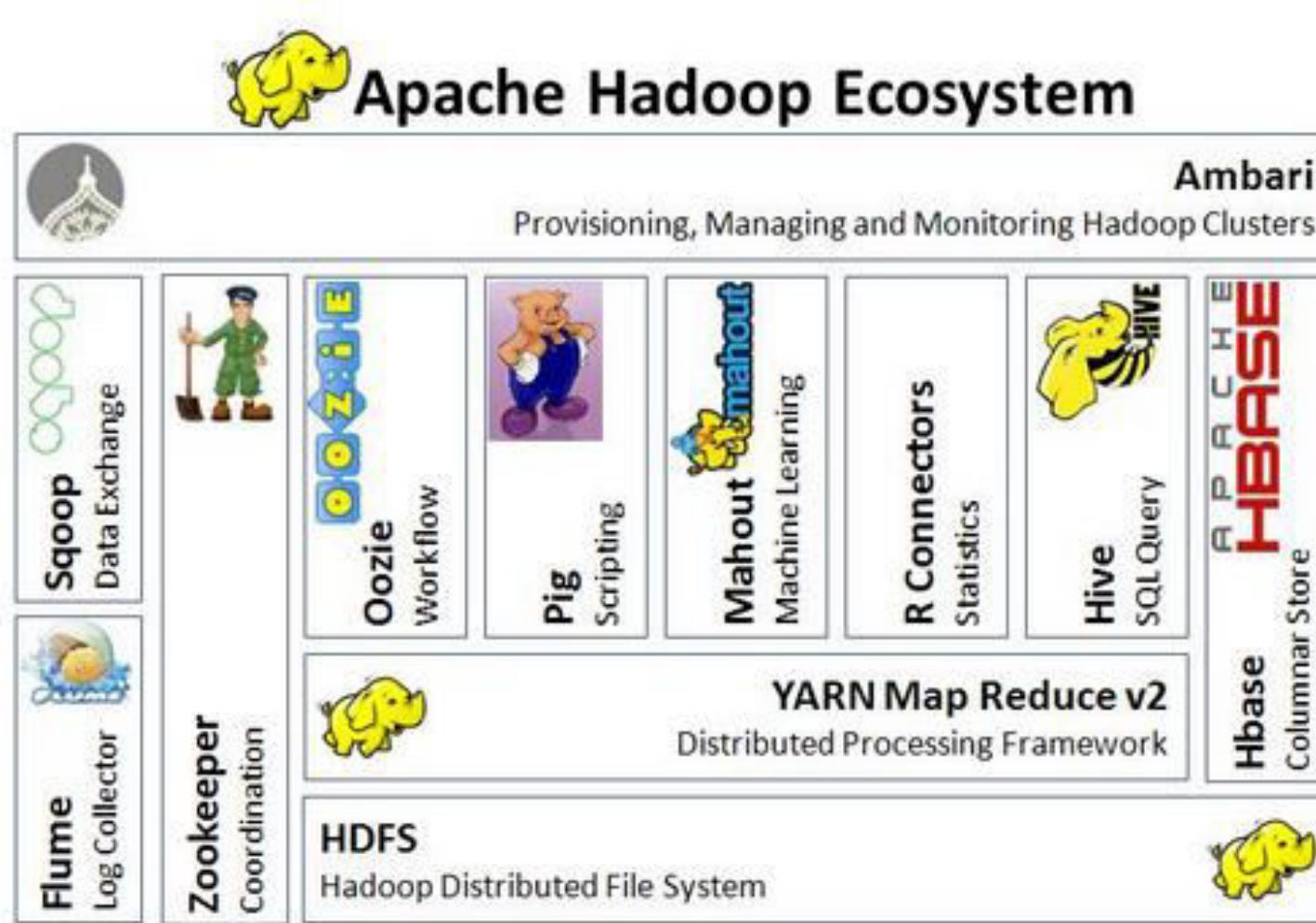




Hadoop

1. What is Hadoop
2. Hadoop Key Properties
3. Brief History
4. Why Hadoop
5. Hadoop vs RDBMS
- 6. Hadoop Architecture**
7. Hadoop components

Hadoop architecture



What Hadoop can do



- Handle large data volume
 - Run queries spanning days/months – **long running**
 - GB/TB/PBs
- Structured, Semi Structured, Unstructured data
- **Computationally intensive**
 - Deep analytics
 - Machine learning algorithms

What Hadoop can NOT do



Low latency processing (Hadoop is batch oriented)



Hadoop

1. What is Hadoop
2. Hadoop Key Properties
3. Brief History
4. Why Hadoop
5. Hadoop vs RDBMS
6. Hadoop Architecture
- 7. Hadoop components**

HDFS



- Inspired by the Google File System paper
- Scalable, resilient distributed file system
- Master/slave architecture
 - **NameNode** holds metadata
 - **DataNodes** run on nodes, hold data
- Data is replicated to resist DataNode failures
- NameNode resilient with standby NameNodes
- Clients interact with DataNodes directly to read/write data

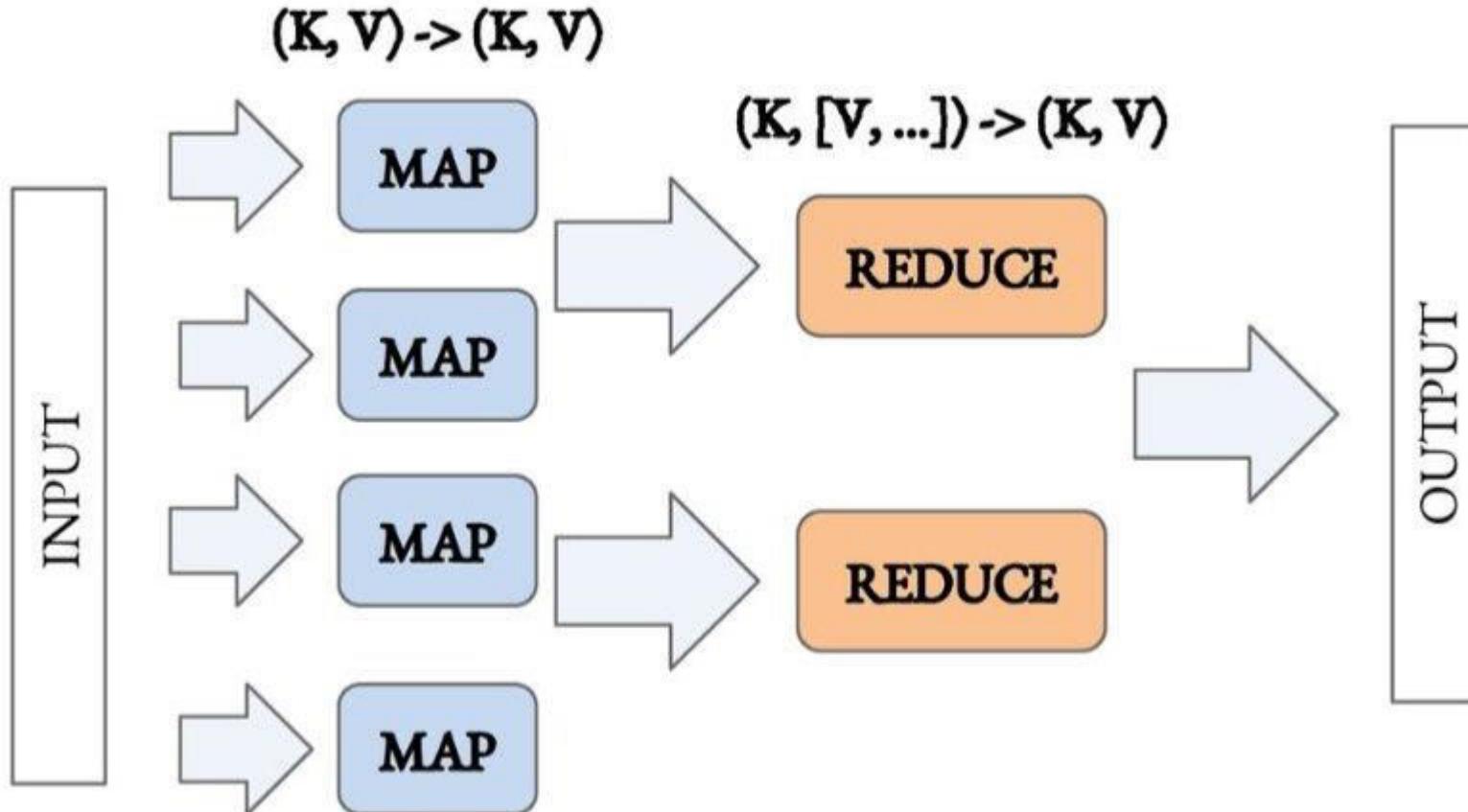
Map Reduce



- Inspired by Google Map Reduce paper
- Runs on top of YARN, implements the familiar distributed computing paradigm

It's a pretty restrictive paradigm.

Map Reduce



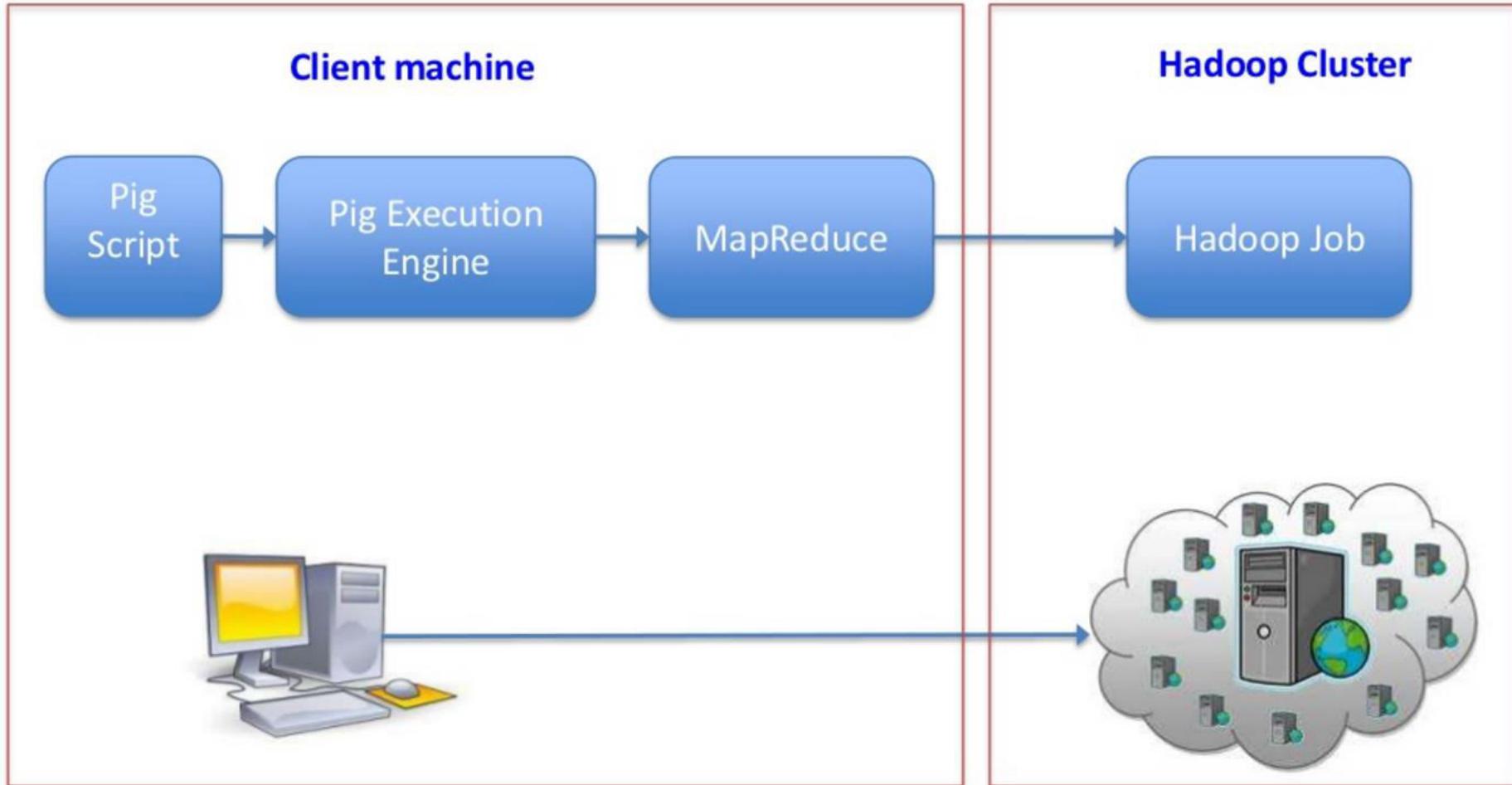
Pig



- Sub project of Hadoop
- Platform for analyse large data sets
- Includes a data flow language: Pig-Latin
- Originally developed at Yahoo!
- User can write custom UDF
- 5% of the code, 5% of the time
- It is procedural and fits very naturally with the pipeline paradigm
- Describes a Direct Acyclic Graph (DAG)



Pig



Pig



```
A = LOAD 'page_views_text' USING PigStorage(' ','-tagFile')
AS (filename:chararray, lang:chararray, page_name:chararray,
view_count:long, page_size:long);

B = FOREACH A {
    date = REGEX_EXTRACT(filename, '.*?pagecounts-([0-9]*-[0-9]*)', 1);
    date_object =ToDate(date,'yyyyMMdd-HHmmss','+00:00');
    unixtime = ToUnixTime(date_object);

    GENERATE (long)unixtime AS datetime,
    lang, page_name,
    view_count, page_size;
}

STORE B INTO 'page_views_parquet' USING parquet.pig.ParquetStorer;
```



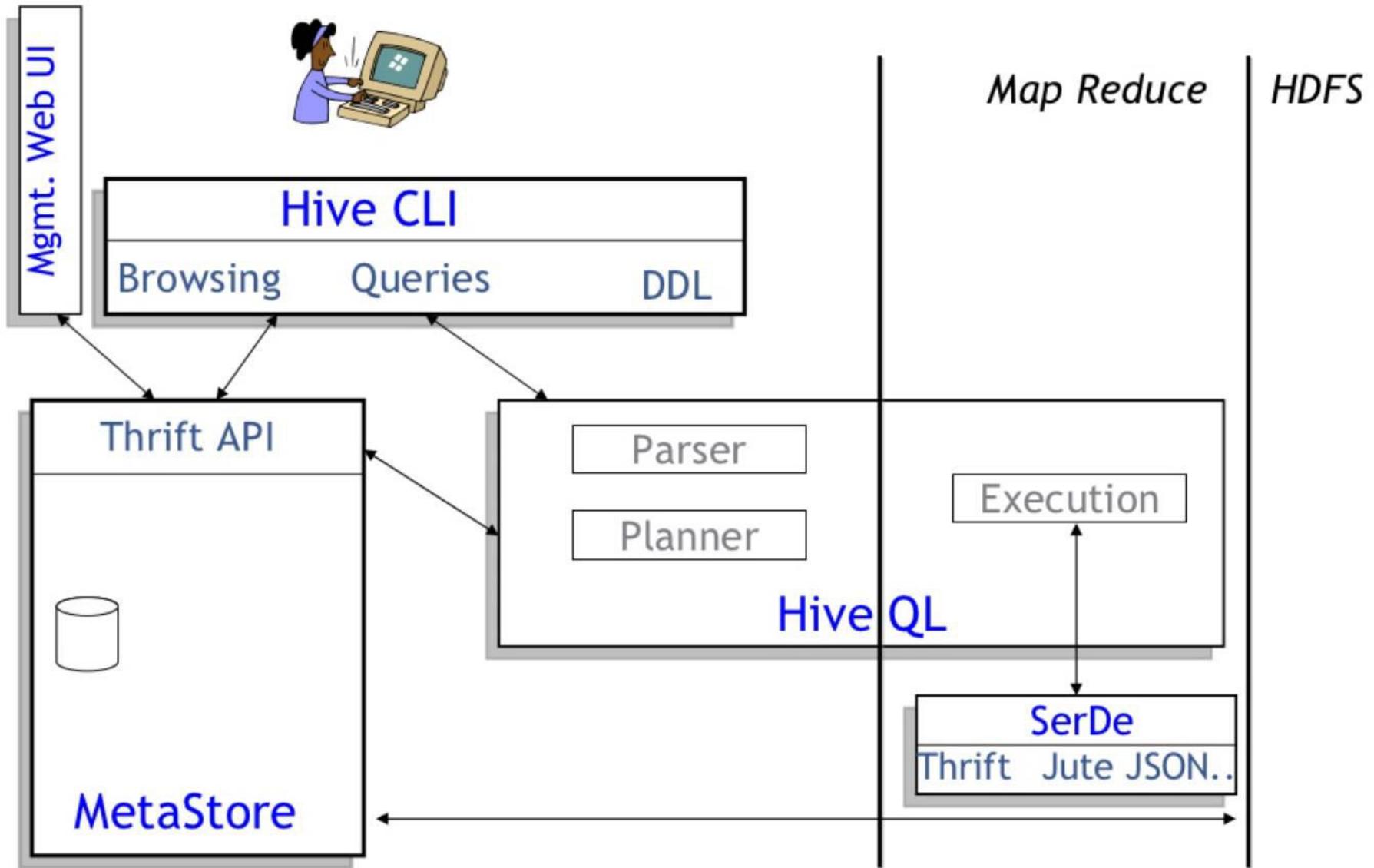
Hive



- Petabyte scale data warehouse system
- **SQL Syntax**
- Batch DWH procedures
- No referential integrity constraints
- Requires **schema**
- Extensible with User Defined Function (**UDF**)
- Multiple formats and HDFS interoperable
- Not offer real-time queries and row level updates
- Not designed for online transaction processing



Hive



Hive



```
CREATE EXTERNAL TABLE page_counts_gz(lang STRING, page_name STRING,  
          page_views BIGINT, page_weight BIGINT  
)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '  
STORED AS TEXTFILE  
LOCATION '/user/admin/pagecounts_text';
```

```
CREATE EXTERNAL TABLE page_counts_parquet(lang STRING, page_name STRING,  
          page_views BIGINT, page_weight BIGINT, datetime BIGINT)  
STORED AS PARQUET  
LOCATION '/user/admin/pagecounts_parquet';
```



Hive



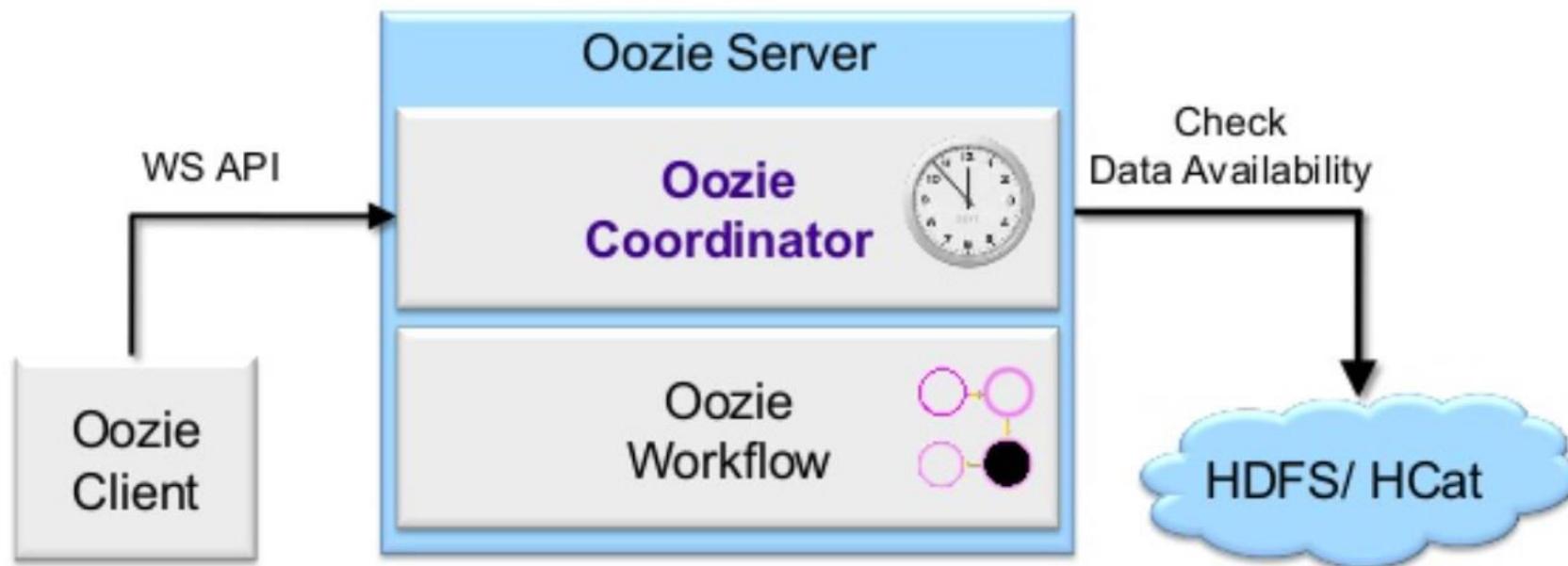
```
INSERT OVERWRITE TABLE page_counts_parquet
SELECT pcg.lang,
       pcg.page_name, pcg.page_views, pcg.page_weight,
       (unix_timestamp(
           regexp_extract(INPUT_FILE_NAME,
           '.*?pagecounts - ([0-9]*-[0-9])*'),
           'yyyyMMdd - HHmmss+)' * 7200)) as datetime;
FROM page_counts_gz pcg
```



Oozie



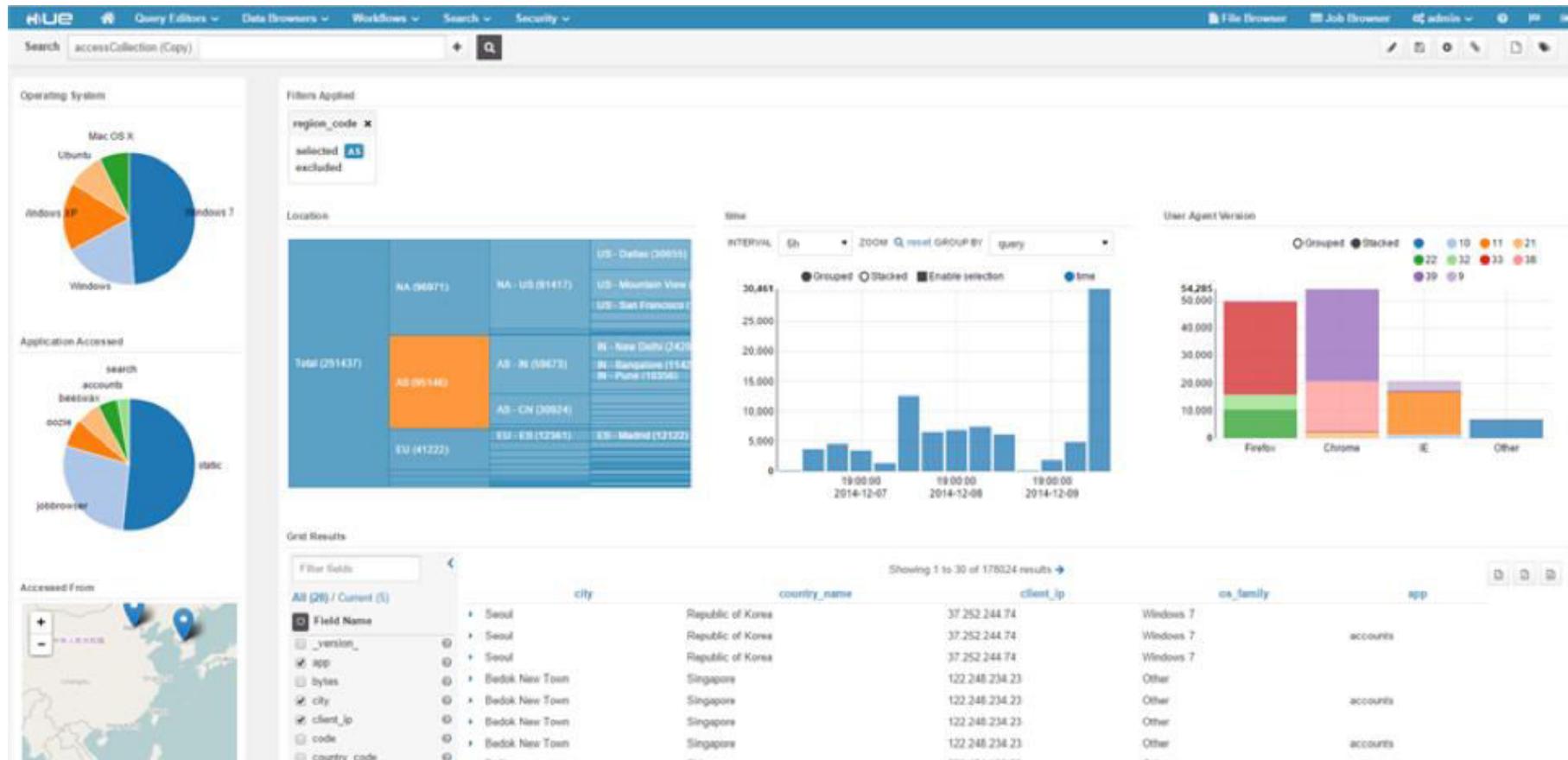
- Workflow management
- Coordinator
- Scheduler



Hue



- Web application over Hadoop
 - Application management
 - Search, Discovery, Monitoring

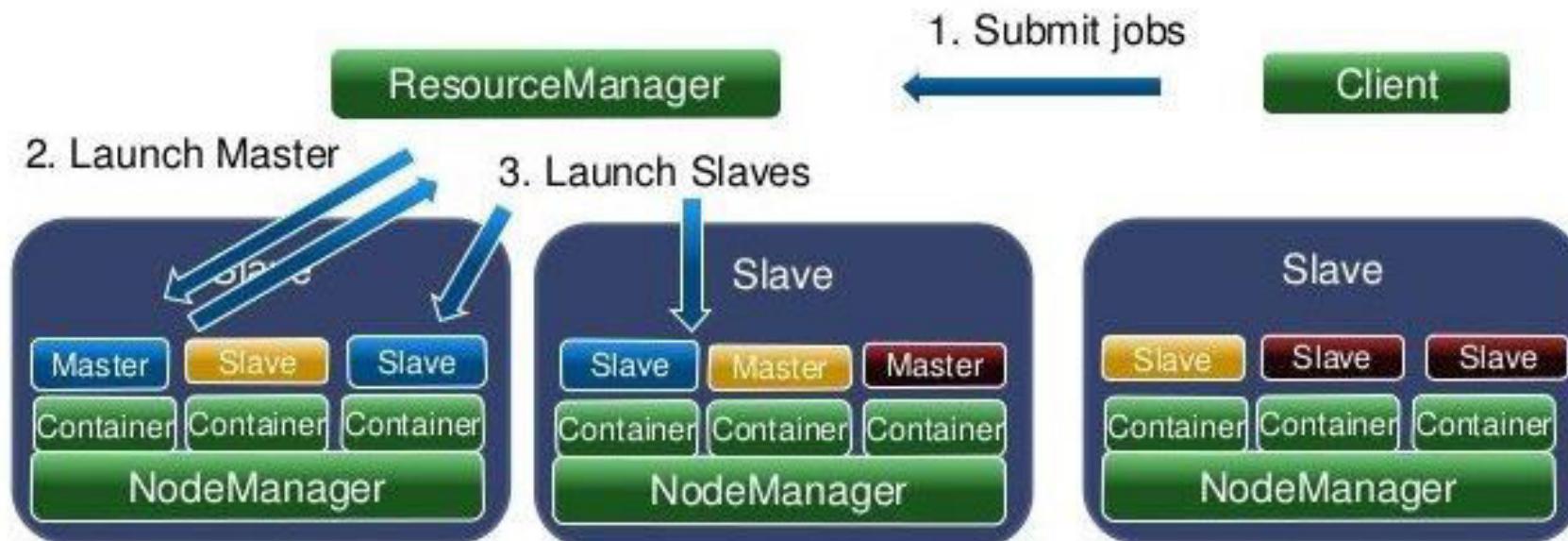


YARN



- Resource management framework for clusters
- Master/slave architecture
 - **ResourceManager** does scheduling
 - **NodeManagers** run on nodes, manage containers
- Containers have well-defined resource constraints and are isolated from each other
- Applications run inside containers
- Each application spawns an Application Master that interacts with YARN

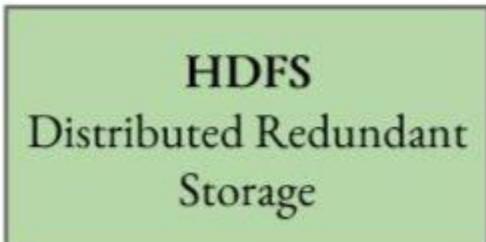
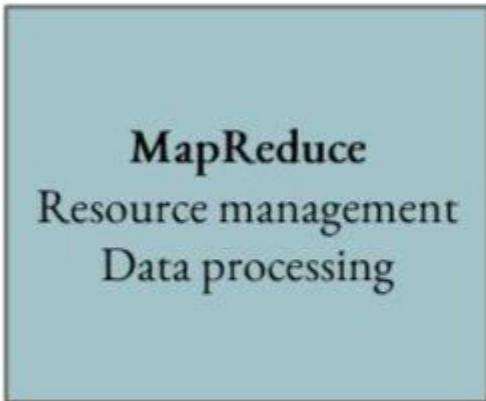
YARN



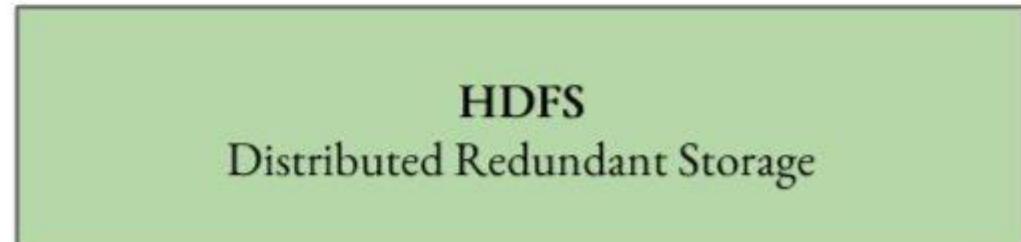
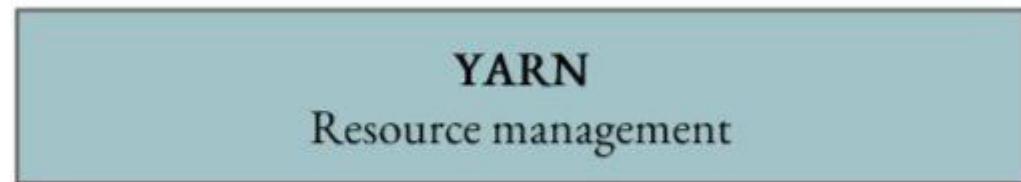
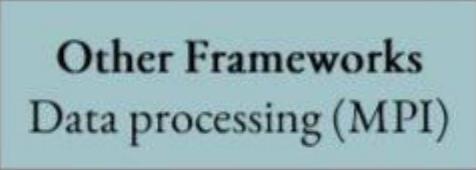
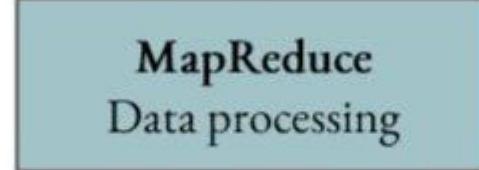
YARN



Hadoop V 1



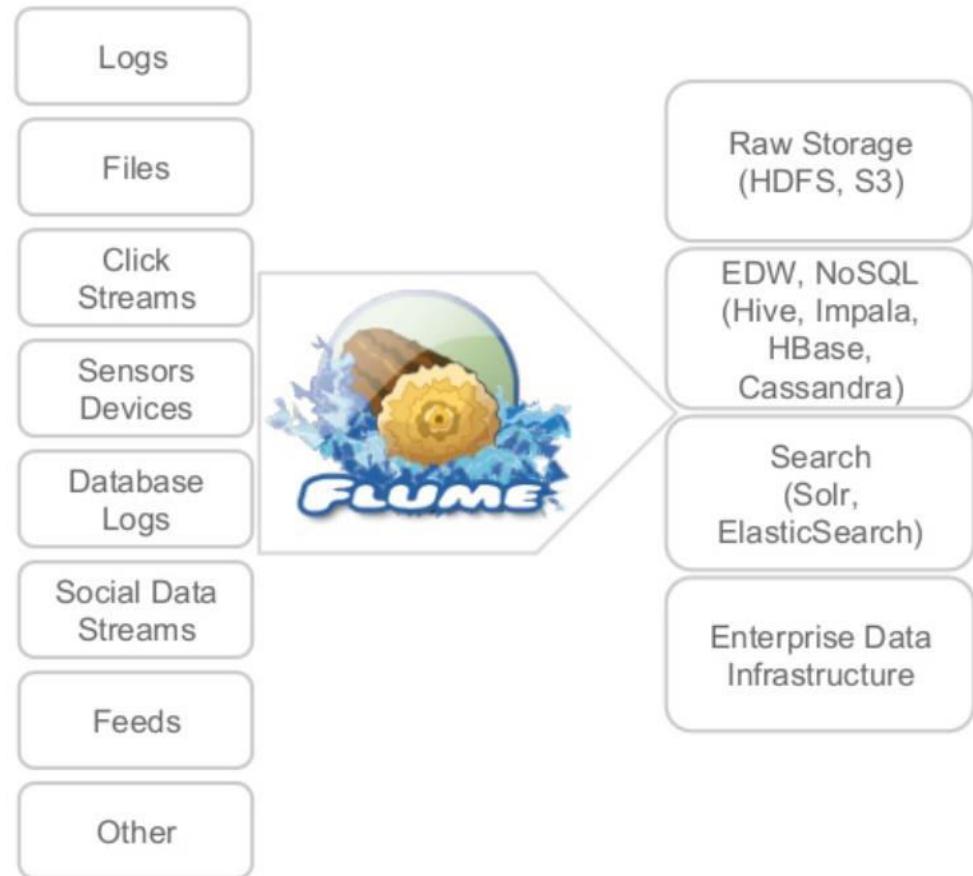
Hadoop V 2



Flume



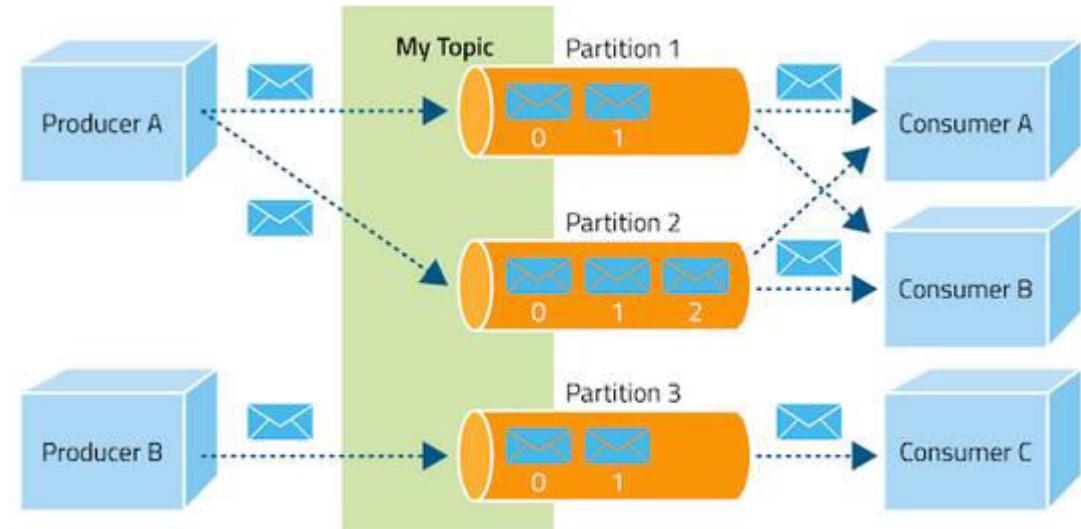
- Move large amount of event data to Hadoop
- Real time solution for streaming data
- Can apply business logic and transformations to incoming data
- Distributed, Scalable, Reliable
- Integrated with Hadoop
- **Does not replicate events !!** - in case of flume-agent failure, you will lose events in the channel



Kafka



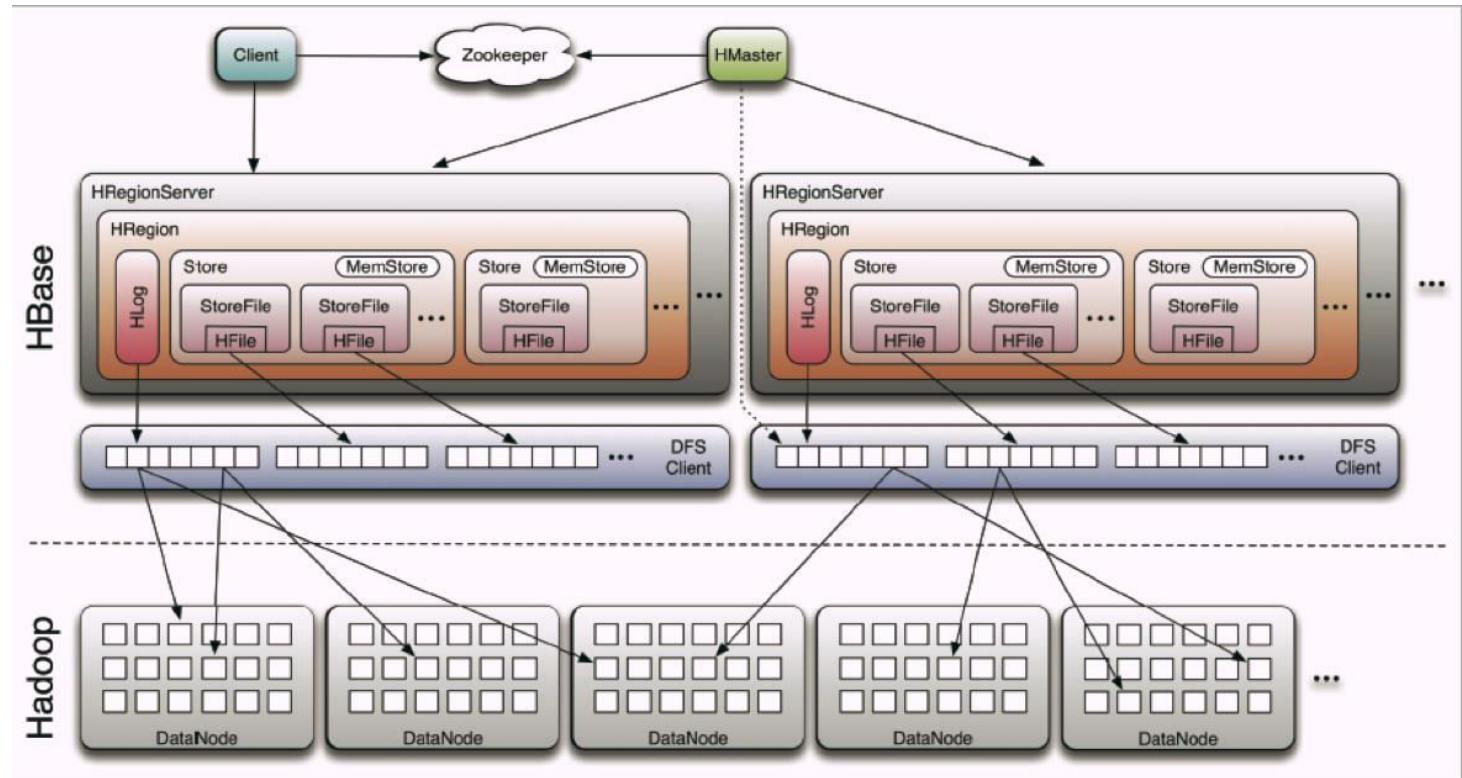
- Distributed publish-subscribe messaging system
- Fast
- Scalable
- Durable
- Log oriented – append only
- Can be combined with Flume
- Should be the foundation of all streaming architectures



HBase



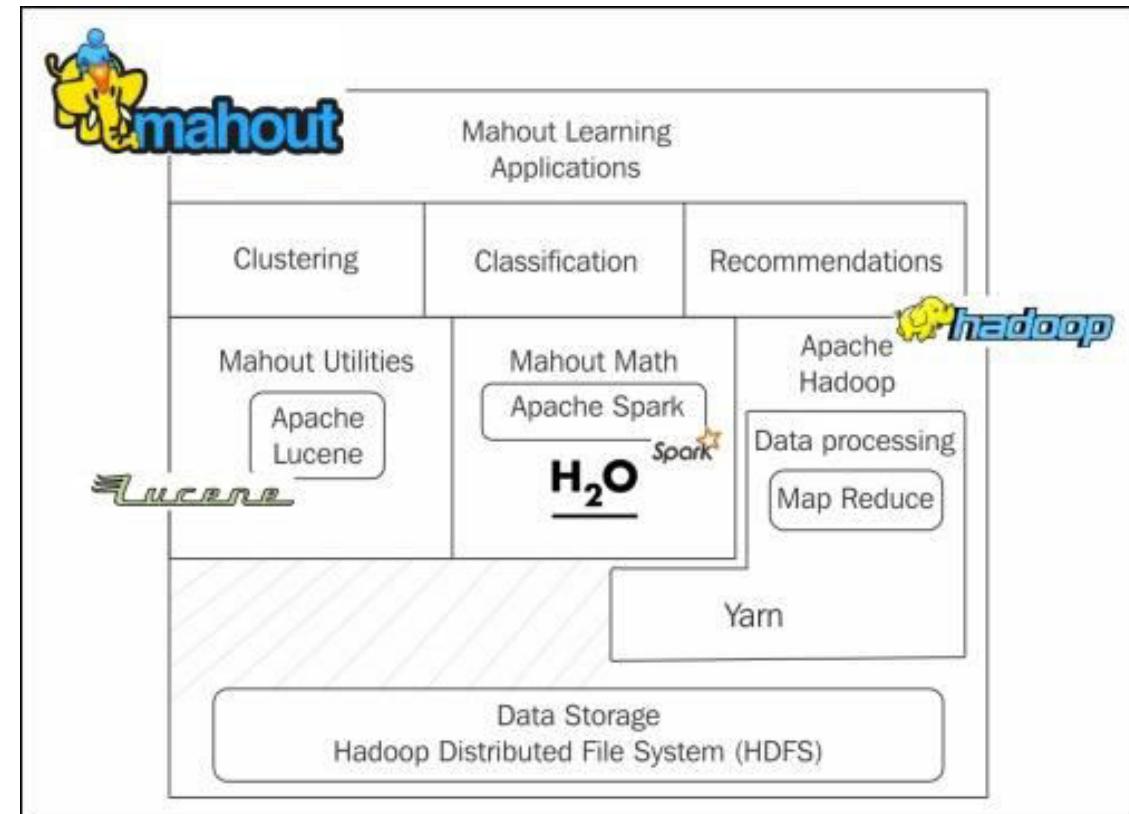
- Key-value store (NoSQL)
- Run on top of HDFS
- Low latency
- Scales linearly
- Poor query syntax
- Fault tolerant
- Consistent
- High availability
- Terrible API



Mahout



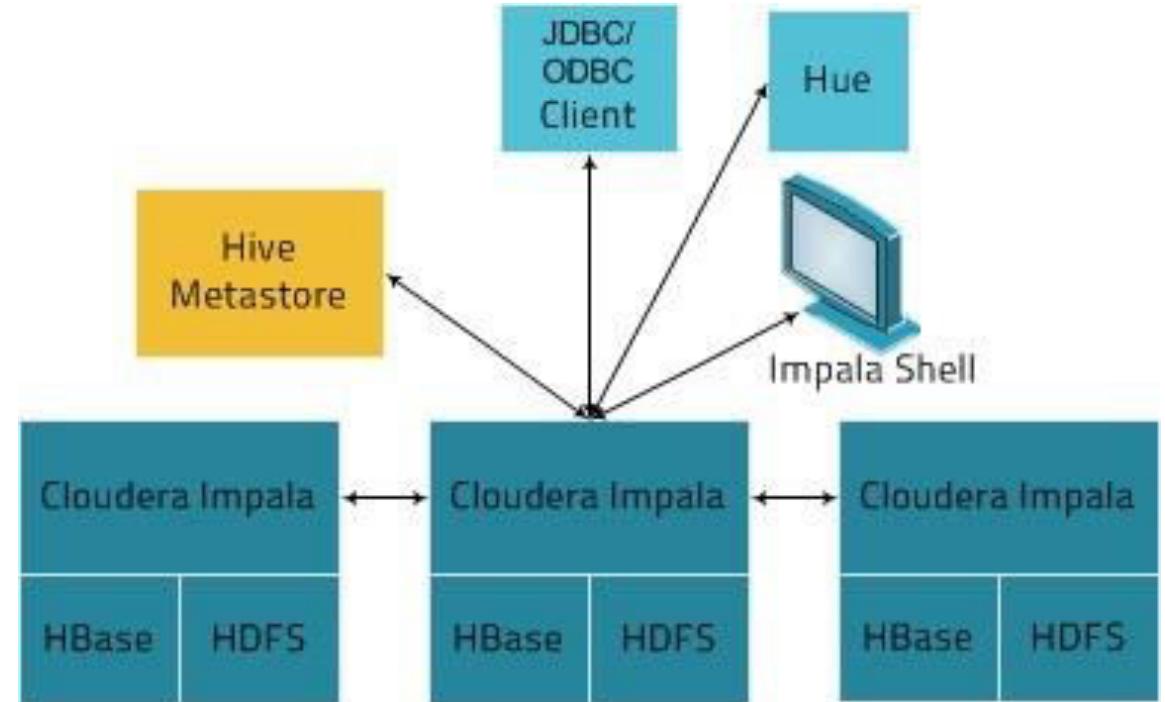
- Is a scalable machine learning library for Hadoop
- It uses MapReduce paradigm which in combination with Hadoop can be used as an inexpensive solution to solve machine learning problems (MapReduce has been replaced by Spark)
- Clustering, Classification, Reccommendation and Math Library



Impala



- Fast MPP Query Engine
- Doesn't rely on MapReduce
- SQL + custom analytics functions
- Lowest latency
- Best concurrency
- Built for BI
- No OLTP style operations



Choosing the best tool for the job



- **Map Reduce** → Low-level processing and analysis – time-consuming, error-prone, best when control matters
- **Hive** → SQL-based queries executed using MapReduce – Batch processing and ETL
- **Impala** → High-performance SQL-based queries using a custom MPP execution engine – BI and SQL Analytics (pay attention to JDBC driver)
- **Hbase** → NoSQL storage for random read / random write data access – OLTP workloads



DEMO



Hadoop Training

Powered by





Data analysis

1. Map Reduce concepts
2. Developing a Map Reduce application
3. Map Reduce Input and Output formats
4. MR Advanced Concepts
5. Data analysis with SQL
6. In-depth Hive

Map Reduce



- Inspired by Google Map Reduce paper
- Programming model for processing and generating large data sets with a parallel and distributed algorithm on a cluster
- Runs on top of YARN, implements the familiar distributed computing paradigm
- Map and reduce functions produce input and output

Map Reduce Terminology



- Job – A full program – an execution of a Mapper and Reducer across a data set
- Task – An execution of a Mapper or a Reducer on a slice of data
 - A.k.a. Task-In-Progress (TIP)
- Task Attempt – A particular instance of an attempt to execute a task on a node

Map Reduce - Functional



```
val a = Array(1,2,3)
```

```
Map  
a.map(_ + 1)  
Array[Int] = Array(2, 3, 4)
```

Reduce

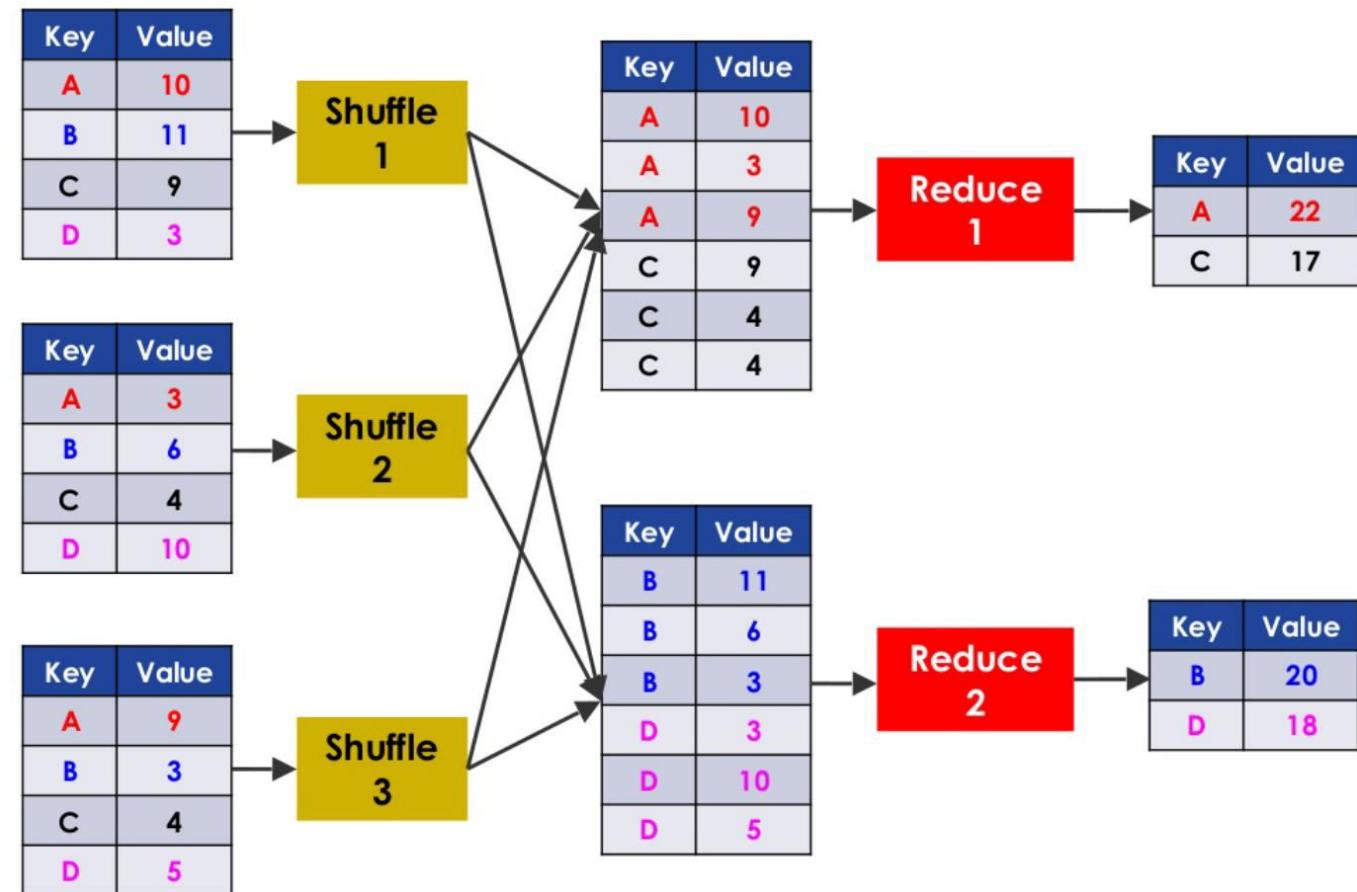
```
a.reduce(_ + _)  
Int = 6
```



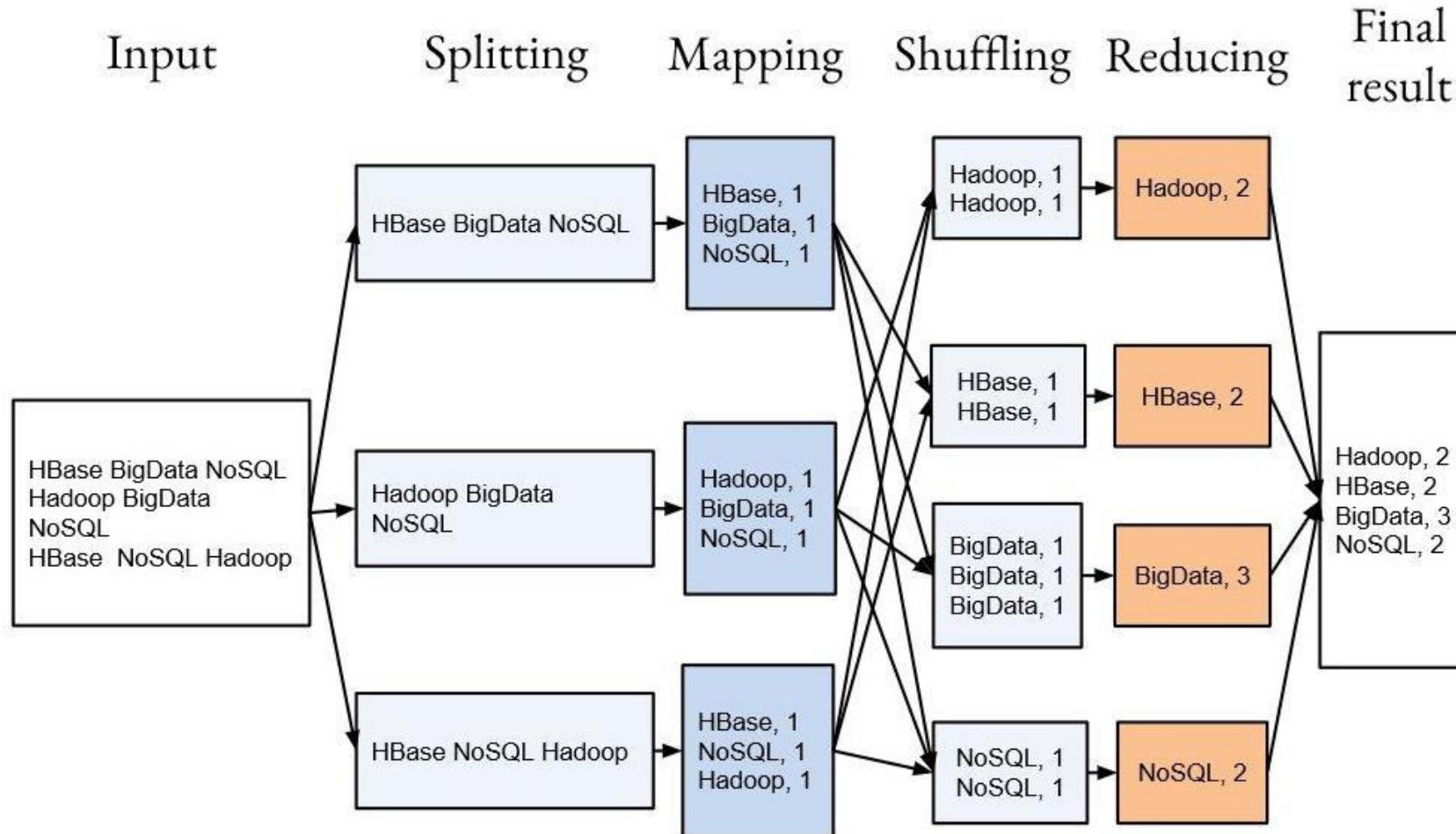
Map Reduce



MapReduce takes an input, splits it into smaller parts, execute the code of the mapper on every part, then gives all the results to one or more reducers that merge all the results into one



Map Reduce

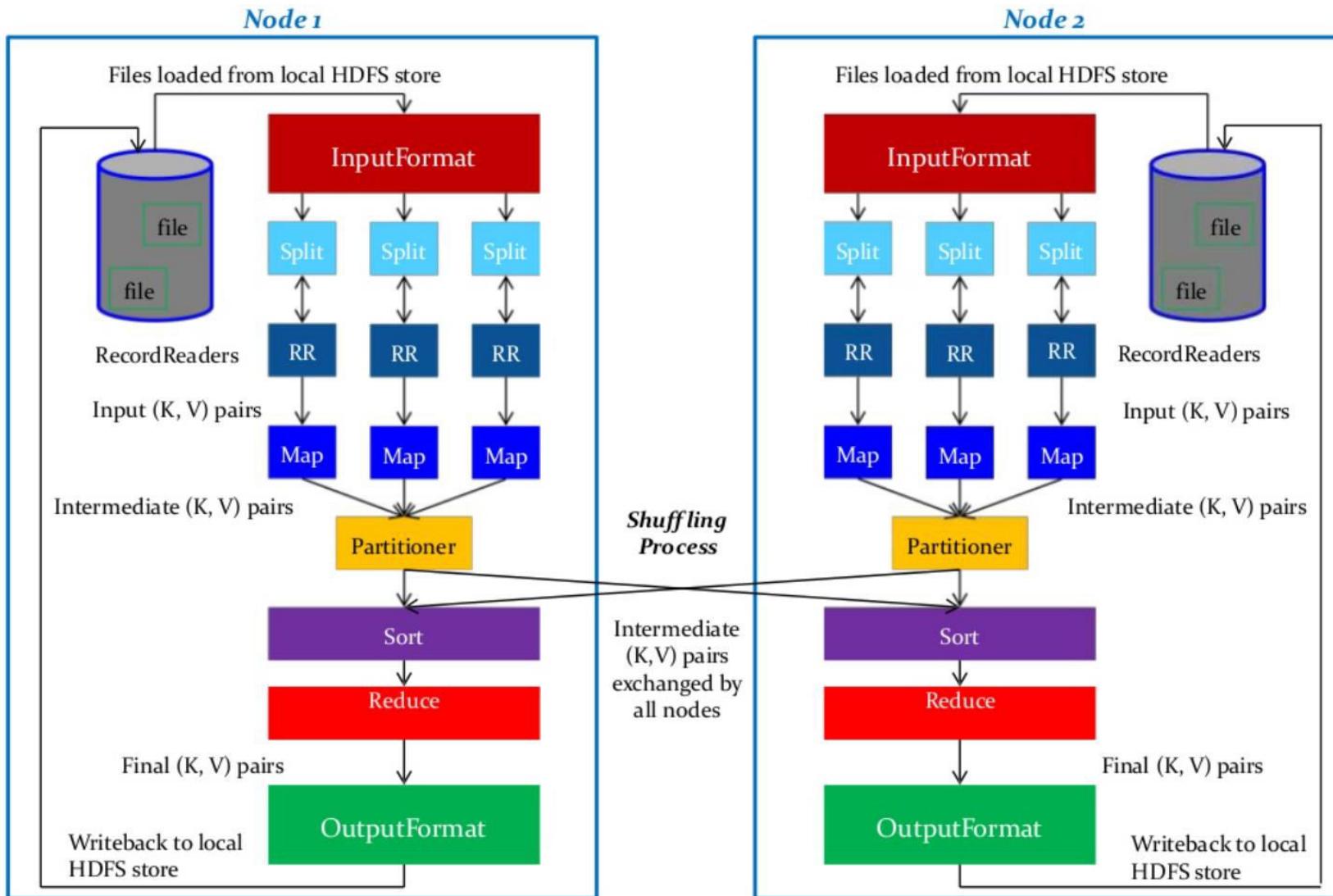


How does it work ? – a closer look



- **Init:** Hadoop divides the input file stored on HDFS into splits and assigns every split to a different mapper, trying to assign every split to the mapper where the split physically resides
- **Mapper:** locally, Hadoop read the split line by line and call the method map() for every line passing it as key/value parameters. Then mapper computes its application logic and emits other key/value pairs
- **Shuffle and sort:** locally, Hadoop's partitioner divides the emitted output of the mapper into partitions, each of those is sent to a different reducer. Hadoop collects all the different partitions received from the mappers and sort them by key
- **Reducer:** locally, Hadoop reads the aggregated partitions line by line and calls the reduce() method on the reducer for every line of the input. Reducer computes its application logic and emits other key/value pairs. Hadoop writes emitted pairs output to HDFS

How does it work ? – a closer look





Data analysis

1. Map Reduce concepts
2. **Developing a Map Reduce application**
3. Map Reduce Input and Output formats
4. MR Advanced Concepts
5. Data analysis with SQL
6. In-depth Hive

Map Reduce - dev



```
public class WordCount extends Configured implements Tool {  
  
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
        static enum Counters { INPUT_WORDS }  
  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(LongWritable key, Text value,  
                        OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {  
  
            String line = (caseSensitive) ? value.toString() : value.toString().toLowerCase();  
  
            for (String pattern : patternsToSkip) {  
                line = line.replaceAll(pattern, "");  
            }  
  
            StringTokenizer tokenizer = new StringTokenizer(line);  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                output.collect(word, one);  
                reporter.incrCounter(Counters.INPUT_WORDS, 1);  
            }  
  
            if ((++numRecords % 100) == 0) {  
                reporter.setStatus("Finished processing " + numRecords + " records " + "from the input file: " + inputFile);  
            }  
        }  
    }  
}
```

Map Reduce - dev



```
public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(other_args.get(0)));
    FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
    JobClient.runJob(conf);
    return 0;
}

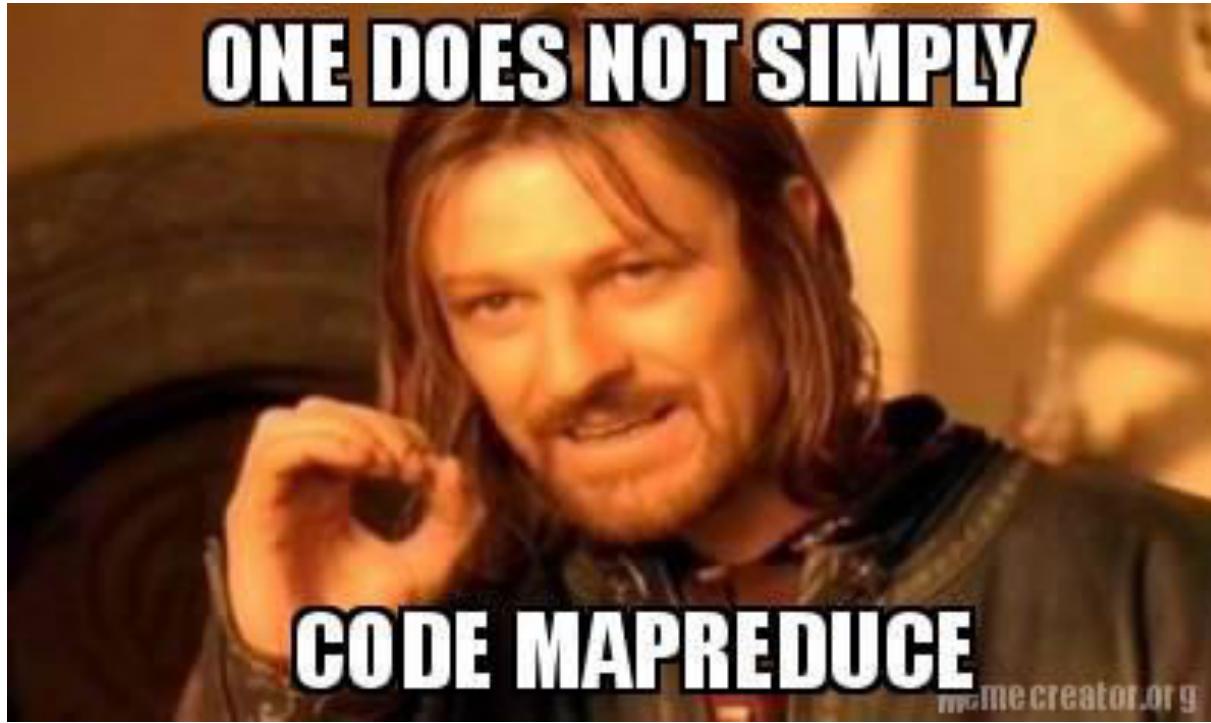
public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(), args);
    System.exit(res);
}
```

Map Reduce



DEMO

Map Reduce





Data analysis

1. Map Reduce concepts
2. Developing a Map Reduce application
3. **Map Reduce Input and Output formats**
4. MR Advanced Concepts
5. Data analysis with SQL
6. In-depth Hive

Hadoop Formats



Basically Hadoop need data types which support both Serialization (for efficient read and write) and comparability (to sort keys within sort and shuffle phase)

- XML
- JSON
- SequenceFile
- Avro
- Parquet
- Thrift
- Protocol Buffers
- Custom Formats

XML, JSON and Sequence File

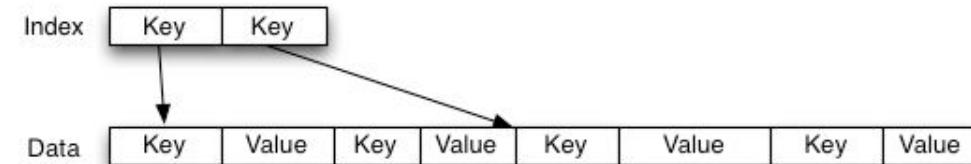


- XML: MapReduce doesn't have native XML support. But you can use Mahout XMLInputFormat
- JSON: MapReduce doesn't come with an InputFormat that works with JSON. But you can use ElephantBird LzoJsonInputFormat
- SequenceFile: It was created specifically for MapReduce tasks, it is row oriented key/value file format. Well supported by Hive and Pig

SequenceFile File Layout

Data	Key	Value	Key	Value	Key	Value	Key	Value
------	-----	-------	-----	-------	-----	-------	-----	-------

MapFile File Layout



Avro



Avro is a RPC and data serialization framework developed by Hadoop in order to improve data interchange, interoperability and versioning in MapReduce.

- Isn't key/value but record based file format
- Support code generation and schema evaluation
- Use JSON to define schema
- Ver good support in MapReduce api
- Cross-language
- Support also for dynamic access
- Support for schema evolution

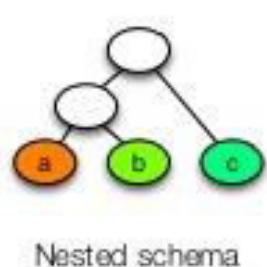


Parquet



Parquet is a columnar storage format(I didn't say file format).

- Parquet physically store data column by column, instead of row by row (e.g Avro)
- In case when you need projection by columns or you need to do operation (avg, min, max) only on the specific columns, it's more effective to store data in columnar format, because accessing data become faster than in case of row storage
- It support schema evaluation but doesn't support code generation
- Well supported by lots of Hadoop projects (Pig, Hive, Spark, Hbase e.t.c)



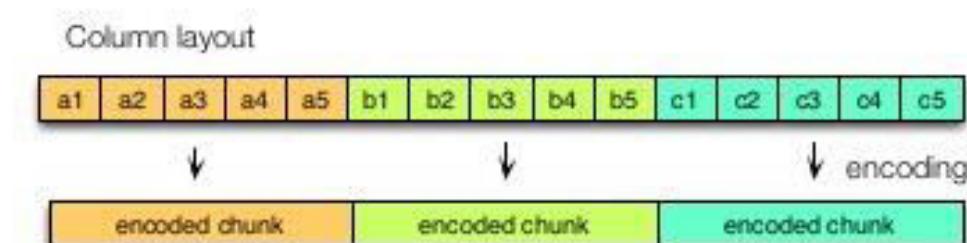
Nested schema

Logical table representation

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Row layout

a1	b1	c1	a2	b2	c2	a3	b3	c3	a4	b4	c4	a5	b5	c5
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

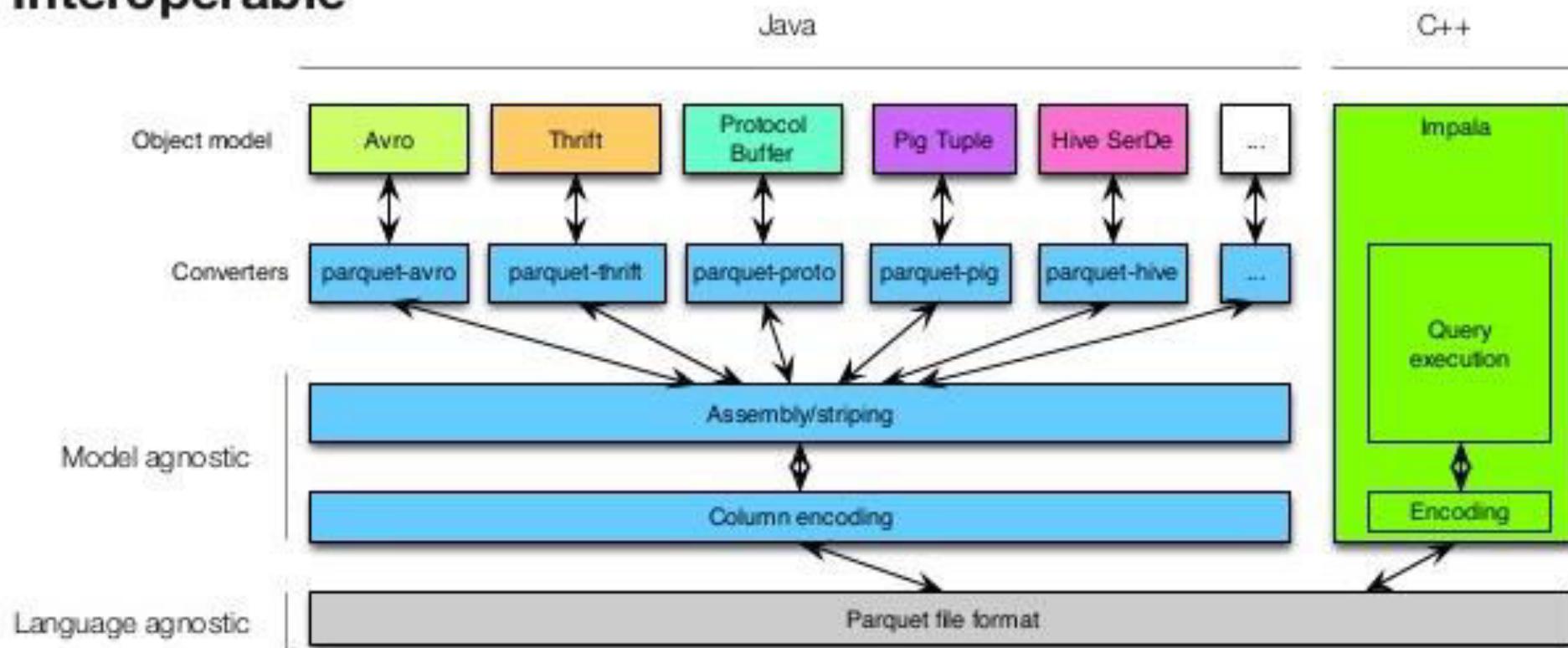


Parquet



Parquet doesn't have his own object model, instead it has object model converters
(to Avro, to Thrift, To Hive, etc...)

Interoperable



Custom Formats (CSV)



You can read the custom file formats with TextInputFormat class and implement reading and writing in your map and reduce tasks appropriately. But if you want write reusable and convenient code for specific file format (e.g CSV) you need to implement your own:

- CSVInputFormat which extend FileInputFormat
- CSVRecordReader which extend RecordReader
- CSVOutputFormat which extend TextOutputFormat
- CSVRecordWriter which extend RecordWriter



Data analysis

1. Map Reduce concepts
2. Developing a Map Reduce application
3. Map Reduce Input and Output formats
- 4. MR Advanced Concepts**
5. Data analysis with SQL
6. In-depth Hive

Job Scheduling



- In MapReduce, an application is represented as a job
- A job encompasses multiple map and reduce tasks
- MapReduce in Hadoop comes with a choice of schedulers
 - FIFO
 - Multi-user called the Fair Scheduler which aims to give every user a fair share the cluster capacity over time

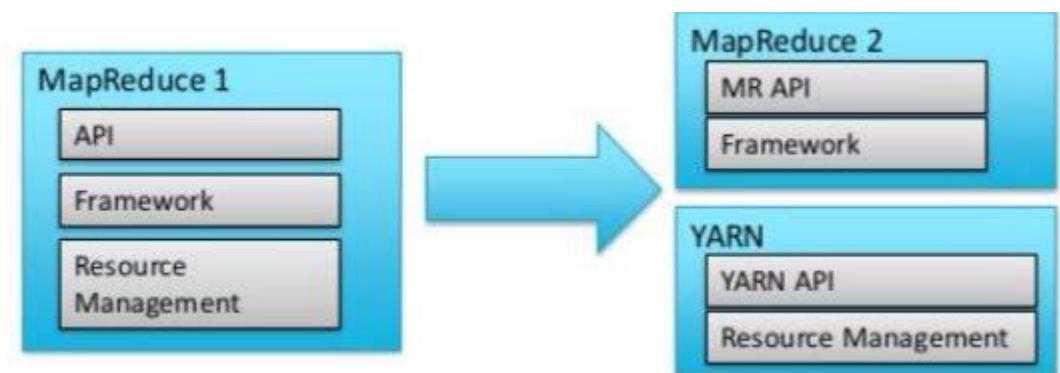
MRv1 and MRv2



MapReduce 1 («Classic») has three main components

- API – for user-level programming of MR applications
- Framework – runtime services for running Ma and Reduce process, shuffling and sorting, etc
- Resource management – infrastructure to monitor nodes, allocate resources and schedule jobs

MapReduce2 moves Resource Management into YARN





Data analysis

1. Map Reduce concepts
2. Developing a Map Reduce application
3. Map Reduce Input and Output formats
4. MR Advanced Concepts
- 5. Data analysis with SQL**
6. In-depth Hive

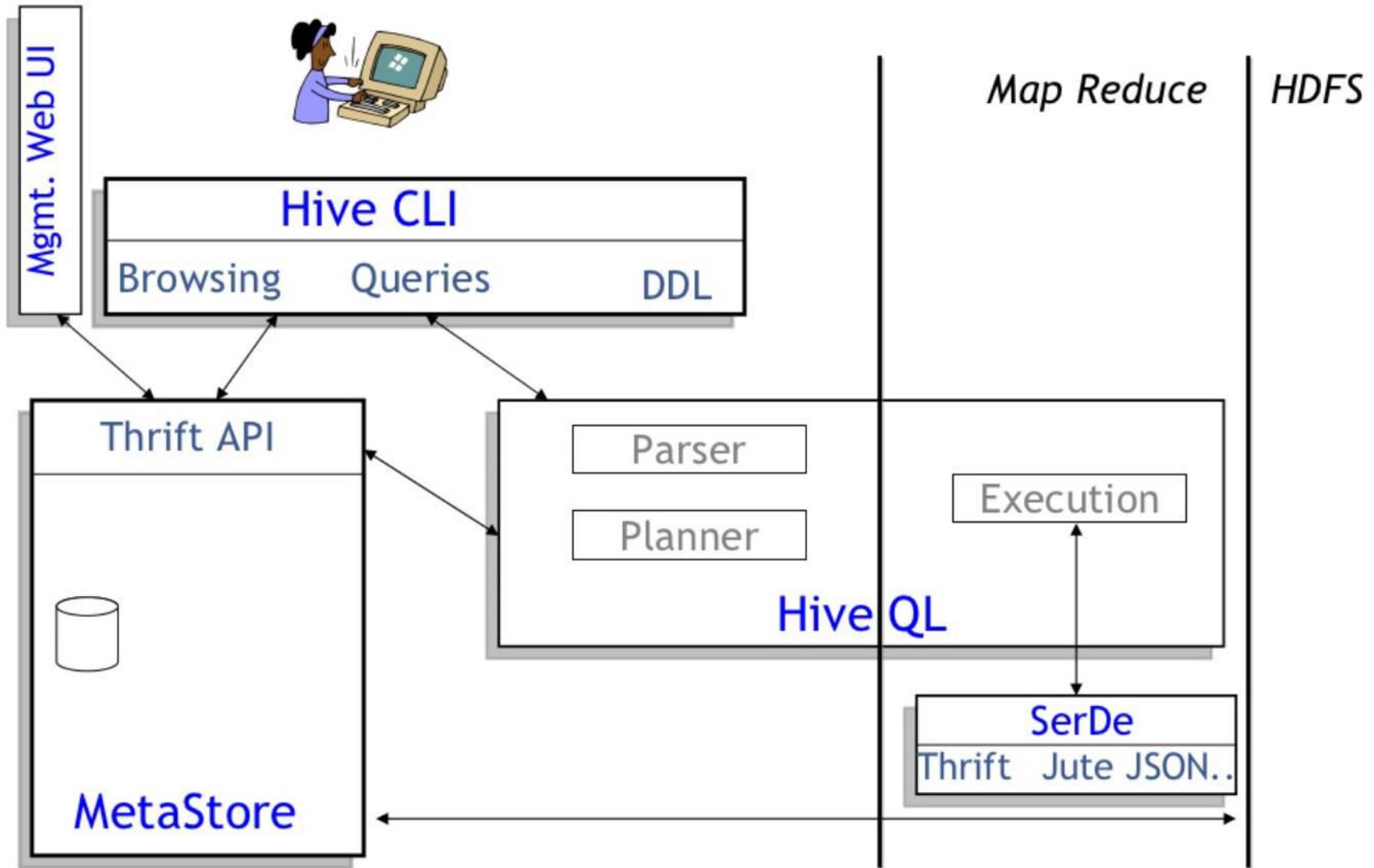
Hive



- Petabyte scale data warehouse system
- SQL Syntax
- Batch DWH procedures
- No referential integrity constraints
- Requires **schema**
- Extensible with User Defined Function (UDF)
- Multiple formats and HDFS interoperable
- Not offer real-time queries and row level updates
- Not designed for online transaction processing



Hive



Hive



```
CREATE EXTERNAL TABLE page_counts_gz(lang STRING, page_name STRING,  
          page_views BIGINT, page_weight BIGINT  
)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '  
STORED AS TEXTFILE  
LOCATION '/user/admin/pagecounts_text';
```

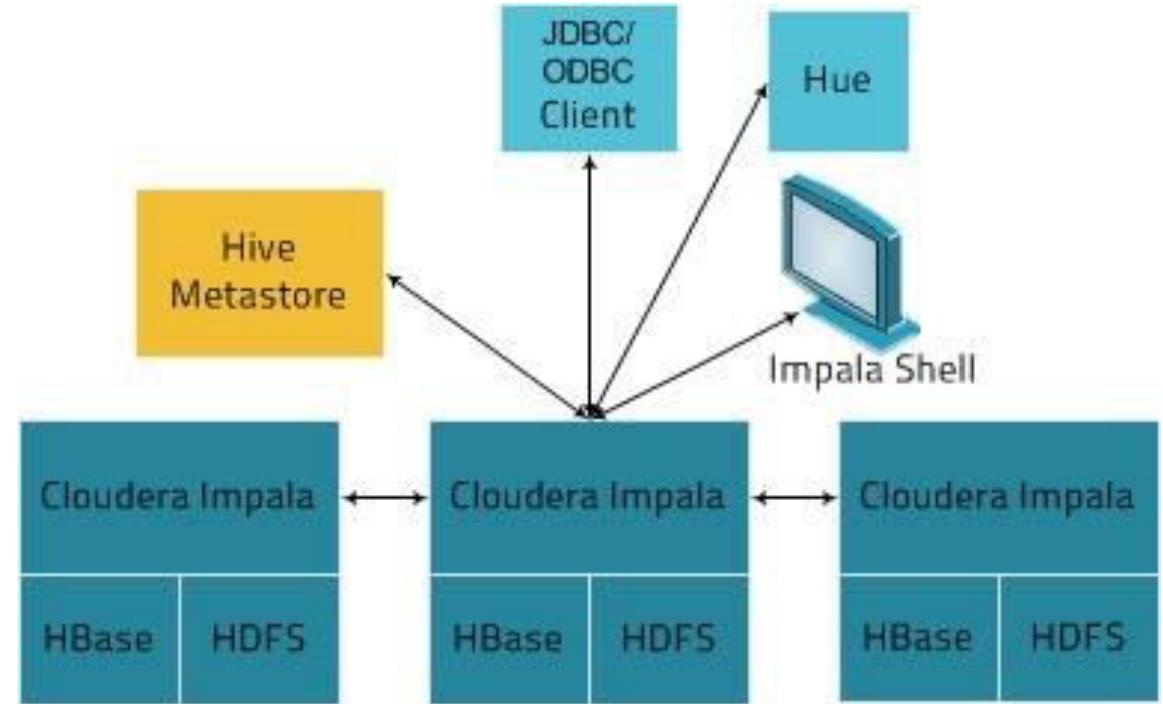
```
CREATE EXTERNAL TABLE page_counts_parquet(lang STRING, page_name STRING,  
          page_views BIGINT, page_weight BIGINT, datetime BIGINT)  
STORED AS PARQUET  
LOCATION '/user/admin/pagecounts_parquet';
```



Impala



- Fast MPP Query Engine
- Doesn't rely on MapReduce
- SQL + custom analytics functions
- Lowest latency
- Best concurrency
- Built for BI
- No OLTP style operations



Choosing the best tool for the job



- **Map Reduce** → Low-level processing and analysis – time-consuming, error-prone, best when control matters
- **Hive** → SQL-based queries executed using MapReduce – Batch processing and ETL
- **Impala** → High-performance SQL-based queries using a custom MPP execution engine – BI and SQL Analytics (pay attention to JDBC driver)
- **Hbase** → NoSQL storage for random read / random write data access – OLTP workloads



Data analysis

1. Map Reduce concepts
2. Developing a Map Reduce application
3. Map Reduce Input and Output formats
4. MR Advanced Concepts
5. Data analysis with SQL
6. In-depth Hive

Hive: supported data



- Files on HDFS

SEQUENCEFILE

TEXTFILE

RCFILE

Available in Hive 0.6.0+

ORC

Available in Hive 0.11.0+

PARQUET

Available in Hive 0.13.0+

AVRO

Available in Hive 0.14.0+

- Other

HBASE

Handling Text file



You can create tables with a custom SerDe or using a native SerDe. A native SerDe is used if ROW FORMAT is not specified or ROW FORMAT DELIMITED is specified.

Use the SERDE clause to create a table with a custom SerDe to handle:

- RegEx
- JSON
- CSV/TSV

SerDe Examples - RegEx



RegEx

```
CREATE TABLE apachelog (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
    "input.regex" = "([^\"]*) ([^\"]*) ([^\"]*) (-|\\[^\\"\\]*\\]) ([^ \"]*|[\"[^\"]*\"])(-|[0-9]*)(-|[0-9]*)(?: ([^ \"]*|\".*\") ([^ \"]*|\".*\"))?"
)
STORED AS TEXTFILE;
```

SerDe Examples – JSON, CSV/TSV



JSON

```
CREATE TABLE my_table(a string, b bigint, ...)  
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'  
STORED AS TEXTFILE;
```

CSV/TSV

```
CREATE TABLE my_table(a string, b string, ...)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  
WITH SERDEPROPERTIES (  
    "separatorChar" = "\t",  
    "quoteChar" = "",  
    "escapeChar" = "\\\""  
)  
STORED AS TEXTFILE;
```

Partitioning



- Could help in data life-cycle (e.g. deleting portion of data)
- Faster access (especially for row-based file format like avro)

- Example DDL

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
    page_url STRING, referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
STORED AS SEQUENCEFILE;
```

- HDFS data file organization

```
<Path to sequencefiles>/page_view/dt=201601/country=IT
<Path to sequencefiles>/page_view/dt=201601/country=FR
<Path to sequencefiles>/page_view/dt=201602/country=US
<Path to sequencefiles>/page_view/dt=201602/country=IT
<Path to sequencefiles>/page_view/dt=201603/country=SP
```

...

Skewed tables optimization



Hive allows to optimize table with high skewed values on one or more columns

By specifying the values that appear very often (heavy skew) Hive will split those out into separate files

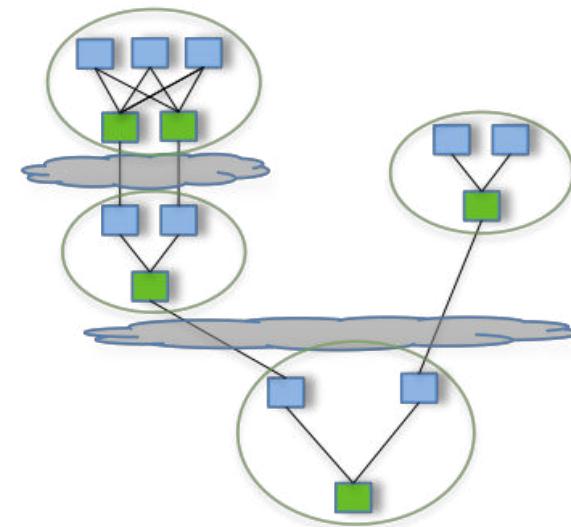
- Example DDL

```
CREATE TABLE people(  
    name STRING,  
    age INT,  
    gender CHAR)  
SKEWED BY (gender) ON ('M', 'F');
```

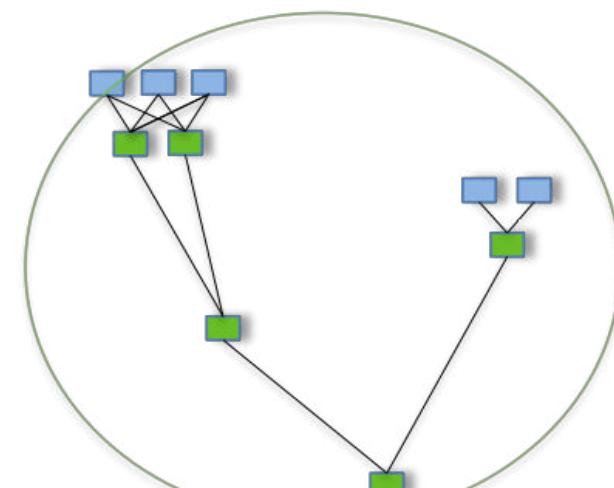
Hive Execution Engines



- MapReduce
 - Legacy execution engine
 - Slower
- Apache Tez
 - Build complex directed-acyclic-graph of tasks for processing data
 - Plan reconfiguration at runtime
 - Dynamic physical data flow decisions
- Apache Spark
 - In-memory computing
 - Better performance



Pig/Hive - MR



Pig/Hive - Tez

Spark Training

Powered by





Spark

- 1. Overview and features**
- 2. Main components**
- 3. Spark users**
- 4. History & Community**

Overview



Apache Spark™ is a **fast and general** engine for large-scale data processing, with built-in modules for streaming, SQL, machine learning and graph processing

Computation diversity



Computation Scopes	MapReduce Model	Spark
Analytics	✓	✓
Interactive data queries	✓*	✓
Stream processing	✗	✓
Iterative algorithms	✗	✓

- Computation in memory makes it easy!
- Workloads can also be combined

Computation diversity



- How fast?
 - Daytona GraySort 2014 <http://sortbenchmark.org/>

	Hadoop MR	Spark	Spark 1PB
Data size	102.5TB	100TB	1PB
Nodes	2100	206	190
Cores	50400 (phy)	6592 (virt)	6080 (virt)
Time	72 mins	23 mins	234 mins
Rate	1.42TB/min	4.27TB/min	4.27TB/min
Rate (node)	0.67GB/min	20.7GB/min	22.5GB/min

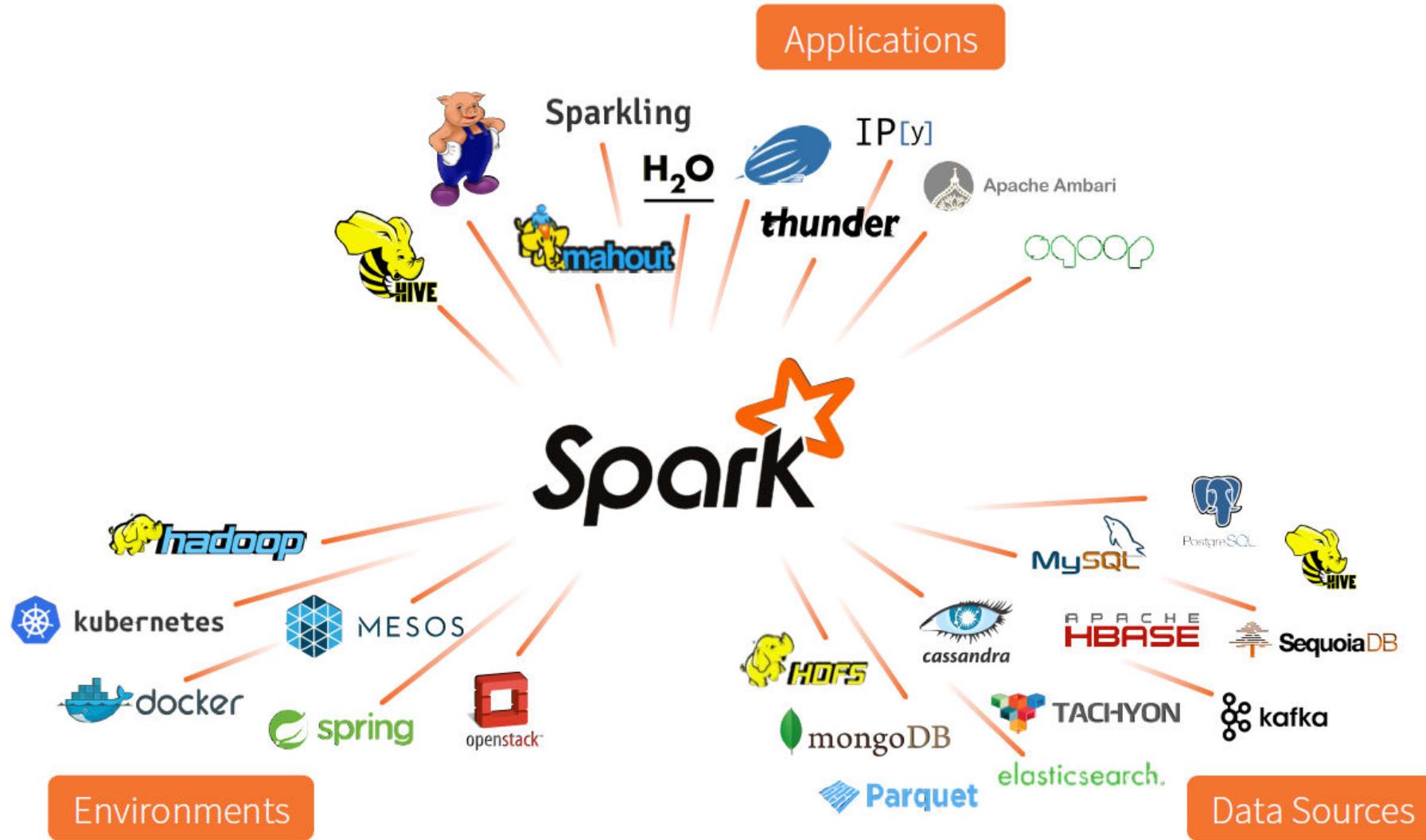
High accessibility



Simple and rich APIs



Well integrated and connected



Support available



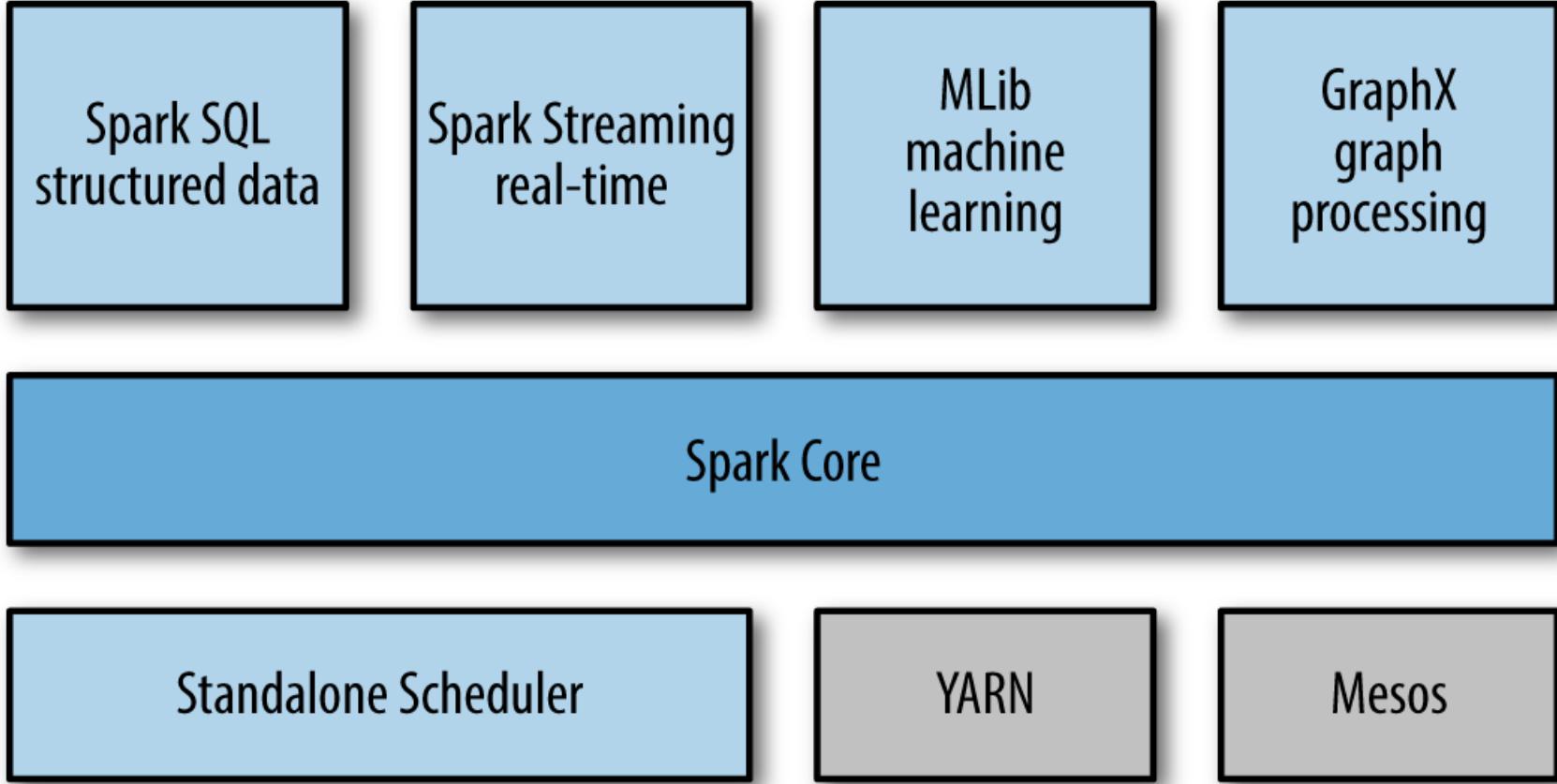
- Founded by Spark's core developers in 2013
- Contributes heavily to the Apache Spark project – 75% of code in 2014
- Offers commercial and technical support



Spark

1. Overview and features
2. **Main components**
3. Spark users
4. History & Community

Spark Unified Stack



Spark Core



Basic functionalities of Spark:

- Task scheduling
- Memory management
- Fault recovery
- Storage system connectors
- RDD APIs

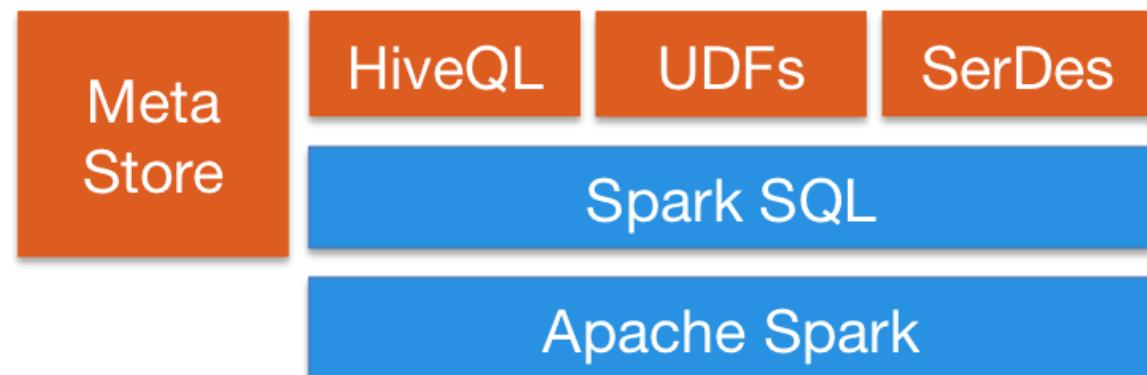
Resilient Distributed Datasets are Spark's main programming abstraction.
A deeper discussion will follow

Spark SQL



Spark's package for working with **structured data**

- Querying via SQL or HQL
- Supports many data source
 - Hive tables
 - Parquet
 - JSON
 - ...
- Possibility to combine SQL with complex analytics

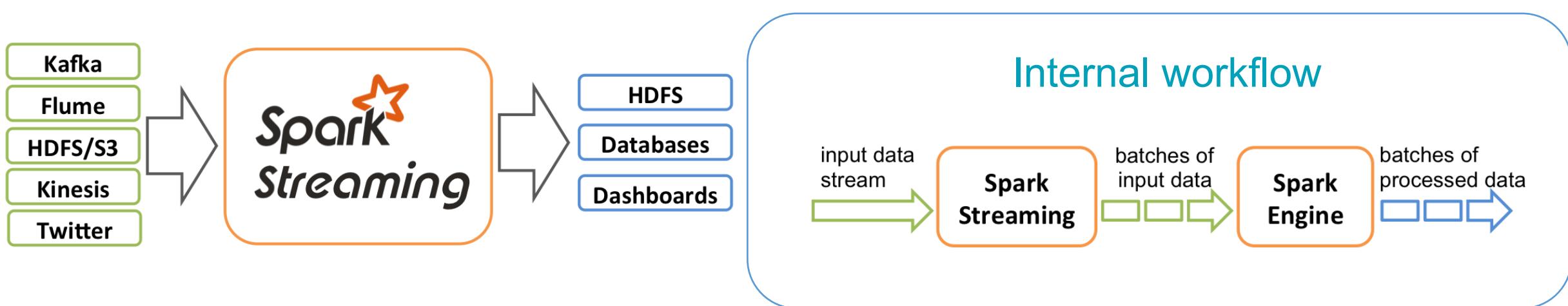


Spark Streaming



Spark component to **process live streams** of data

- Dedicated API to manipulate data streams
- Designed to be comparable to Spark Core in terms of:
 - Fault tolerance
 - Throughput
 - Scalability
- Well combined with batch computation and interactive queries

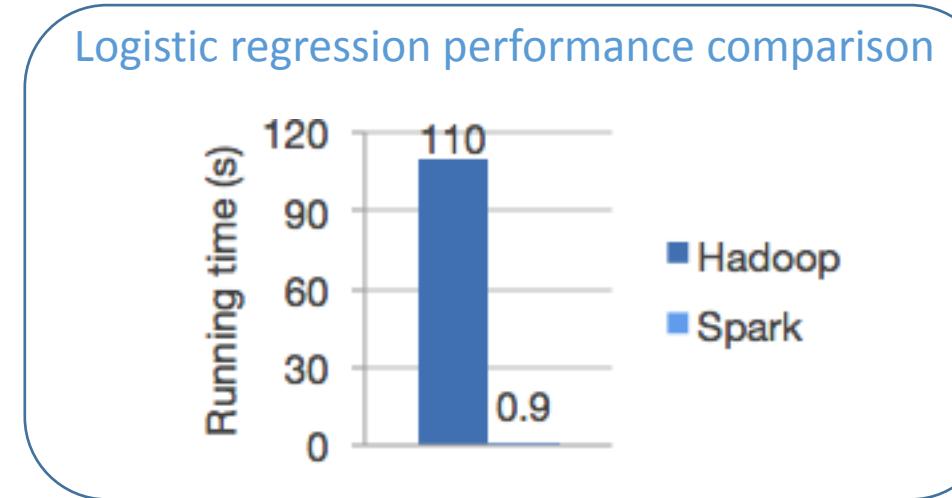


Spark MLlib



Spark's scalable machine learning library

- Designed to scale out across a cluster
- Many ready-to-use known algorithms and utilities are available
 - **Classification:** logistic regression, naive Bayes
 - **Regression:** generalized linear regression, isotonic regression
 - Decision trees, random forests, and gradient-boosted trees
 - **Recommendation:** alternating least squares (ALS)
 - **Clustering:** K-means, Gaussian mixtures (GMMs)
 - **Feature transformations:** standardization, normalization, hashing
 - Model evaluation and hyper-parameter tuning
 - ML persistence: saving and loading models and Pipelines
 - Statistics: summary statistics, hypothesis testing
 - Many others!



APIs:

- Java
- Scala
- Python (ver. 0.9+)
- R (ver. 1.5+)

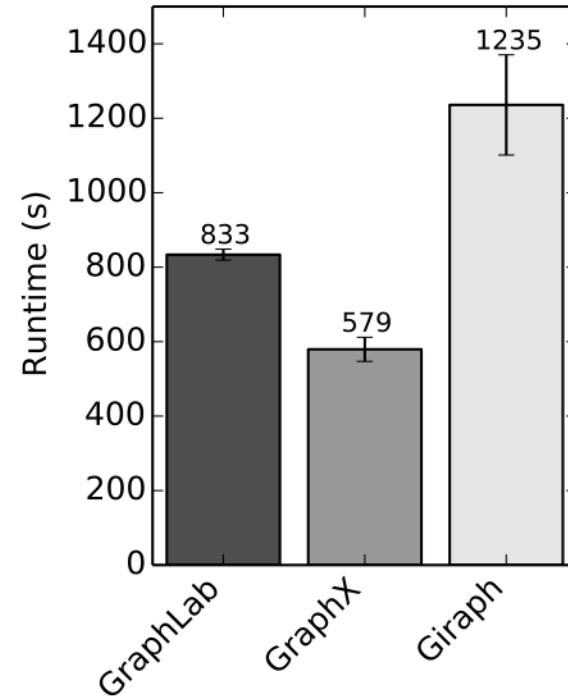
Spark GraphX



Spark's library for manipulating graphs

- Graph-parallel computations
- APIs for manipulating graphs
- Common graph algorithms available (e.g. PageRank, Triangle count)
- High performance retaining Spark's flexibility, fault tolerance, and ease of use.

Page rank performance comparison



End-to-end PageRank performance
(20 iterations, 3.7B edges)

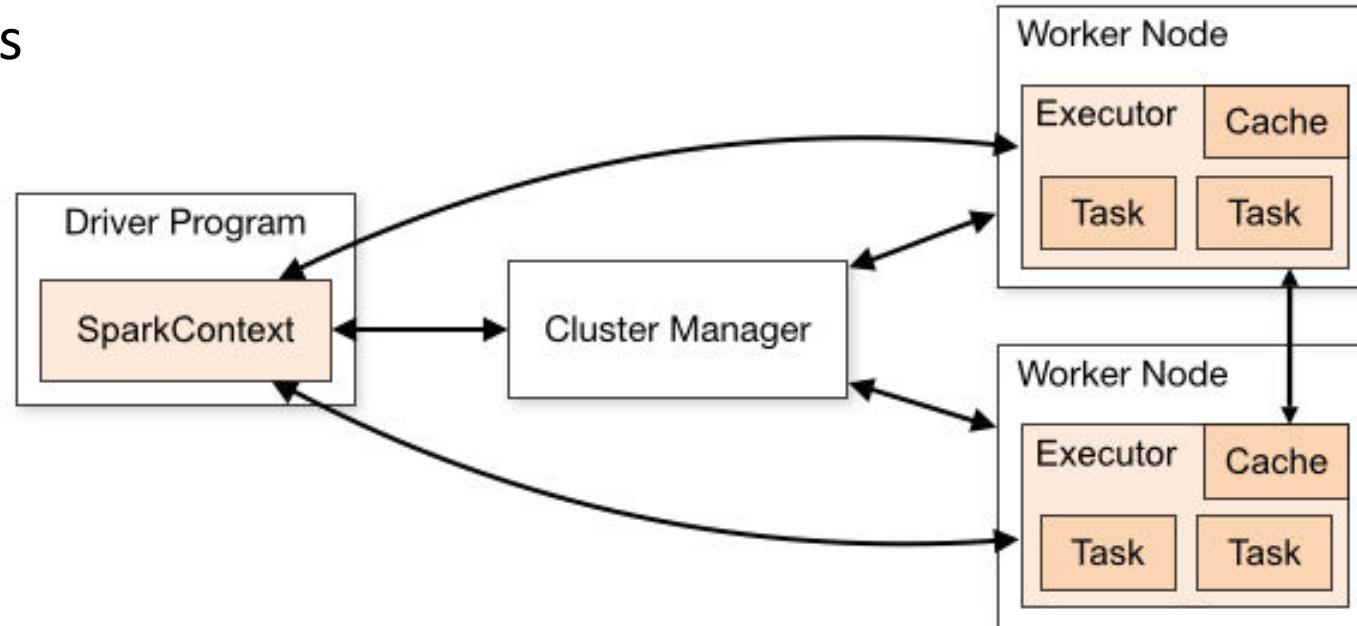
Cluster Managers



Spark is designed to efficiently scale up from 1 to thousand compute nodes.

It can run over different cluster managers:

- Standalone Scheduler (included Spark)
- Hadoop YARN
- Apache Mesos





Spark

1. Overview and features
2. Spark Unified Stack
- 3. Spark users**
4. History & Community

Who uses Spark?



Two main task categories

- Data Science
- Data Applications

	Data Science	Data Applications
Who	Data Scientist	Engineer/Developer
What	Discovering, Analyzing and Modeling Data	Building data processing applications in Production environments
How does Spark help	Supports Python, R, SQL	Java/Scala available
	Fast interactive queries	Complexity of distributed systems, Network communication and fault tolerance hidden
	Machine learning ready-to-use	Mature and reliable for production environments
	Interactive shells availability	Monitoring, inspection and tuning easy
	Connection with R, Matlab external programs	Modular APIs



Spark

1. Overview and features
2. Spark Unified Stack
3. Spark users
- 4. History, Community, Ecosystem**

History



- Started in 2009 as a Berkeley AMPLab research project.
- Matei Zaharia's Ph.D. research.
- First open sourced in March 2010
- Now a top-level [Apache project](#) (since 2013)

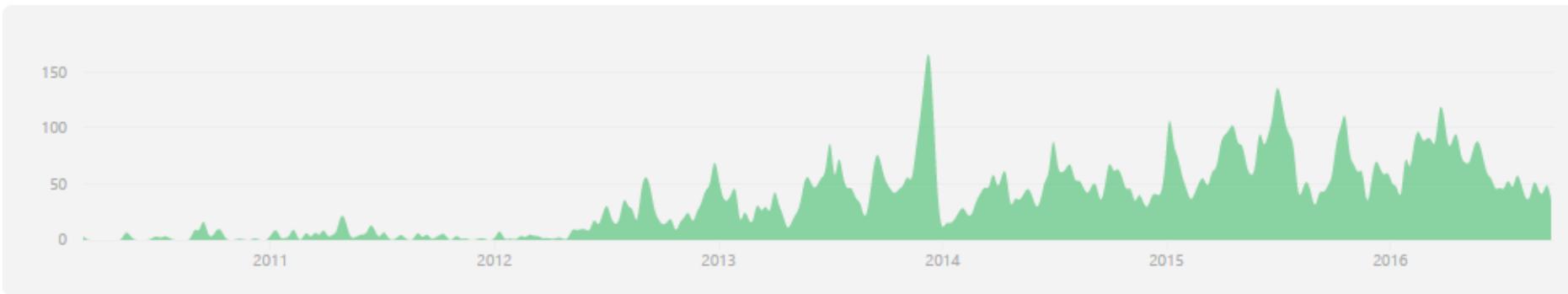


<https://amplab.cs.berkeley.edu/software/>

An open source success



Most active open source project in data processing



Commits on master branch
From March 2010 to October
2016

	2014	2015
Contributors	255	730
Contributors/month	75	135
Lines of code	175'000	400'000 (mostly libraries)

Spark enterprise users



Widely used in production by different industries!

Worldwide active community



Conferences and meetups worldwide

- Spark summits in America and Europe
- Community Meetups* worldwide (500+ with over 260K users)



* Check out our new Meetup group in Milan <https://www.meetup.com/it-IT/Spark-More-Milano/>

Spark Resources



- Spark HUB: <https://sparkhub.databricks.com/>
- Safari O'reilly: <https://www.safaribooksonline.com>
- Mailing List: User & Dev
- Spark Packages: <https://spark-packages.org/>

Spark Packages



- Plug & play libraries for spark
- More than 250 packages !!!
- Connectors, Algorithms, Utilities, Applications, Examples
- <https://spark-packages.org/>

Popular Spark Packages



spark-indexedrdd: An efficient updatable key-value store for Apache Spark

Magellan: Geo Spatial Data Analytics on Spark

spark-corenlp: A Stanford CoreNLP wrapper for Apache Spark

dl4j-spark-ml: Deep Learning for Spark ML

Mean-Shift-LSH: Spark implementation of Nearest Neighbours Mean Shift using LSH

Sparkxgboost: gradient boosting tree with arbitrary user-defined loss function

Snappydata: SnappyData: OLTP + OLAP Database built on Apache Spark

spark-to-tableau: Spark DataFrame to Tableau Data Extract library

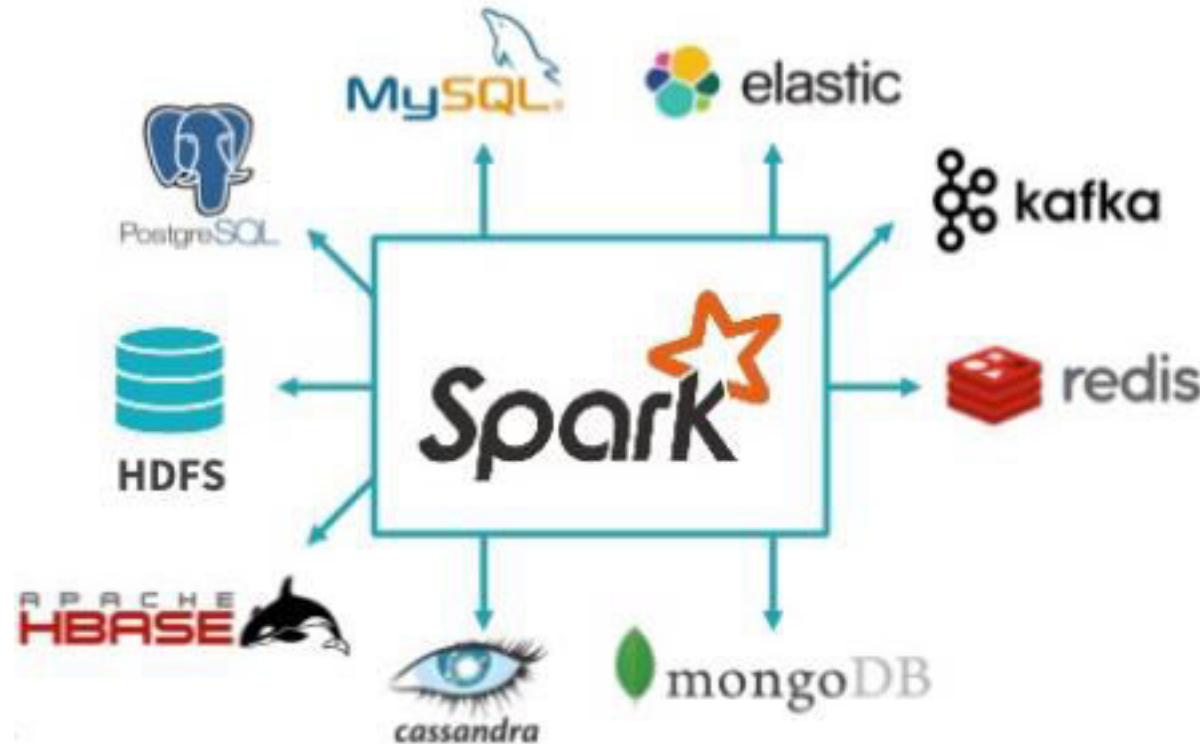
spark-hyperloglog: Algebird's HyperLogLog support for Apache Spark

GPUEnabler: Provides GPU awareness to Spark

spark-lucenerdd: Lucene query capabilities for Spark

spark-timeseries: A library for time series analysis on Apache Spark

Spark Connectors



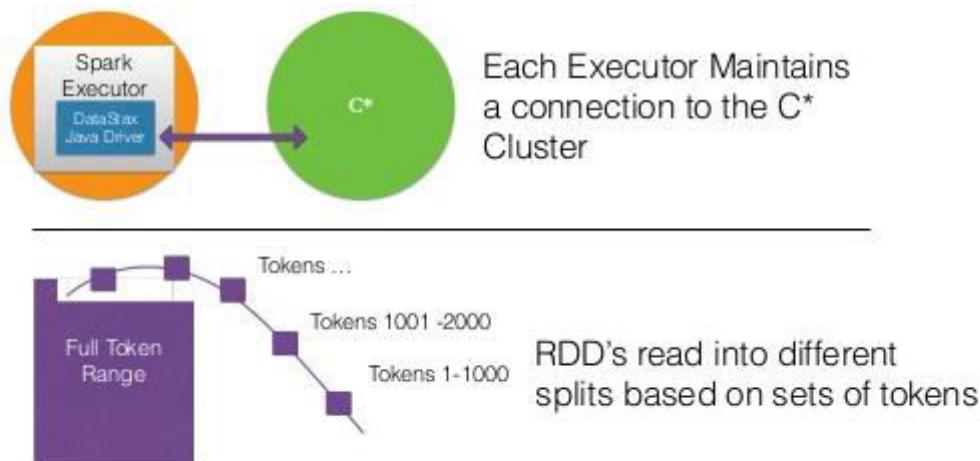
Spark Cassandra Connector



This library lets you expose Cassandra tables as Spark RDDs, write Spark RDDs to Cassandra tables, and execute arbitrary CQL queries in your Spark applications

```
val conf = new SparkConf(true)
    .set("spark.cassandra.connection.host", "192.168.123.10")
    .set("spark.cassandra.auth.username", "cassandra")
    .set("spark.cassandra.auth.password", "cassandra")
```

Spark Cassandra Connector uses the DataStax Java Driver to
Read from and Write to C*

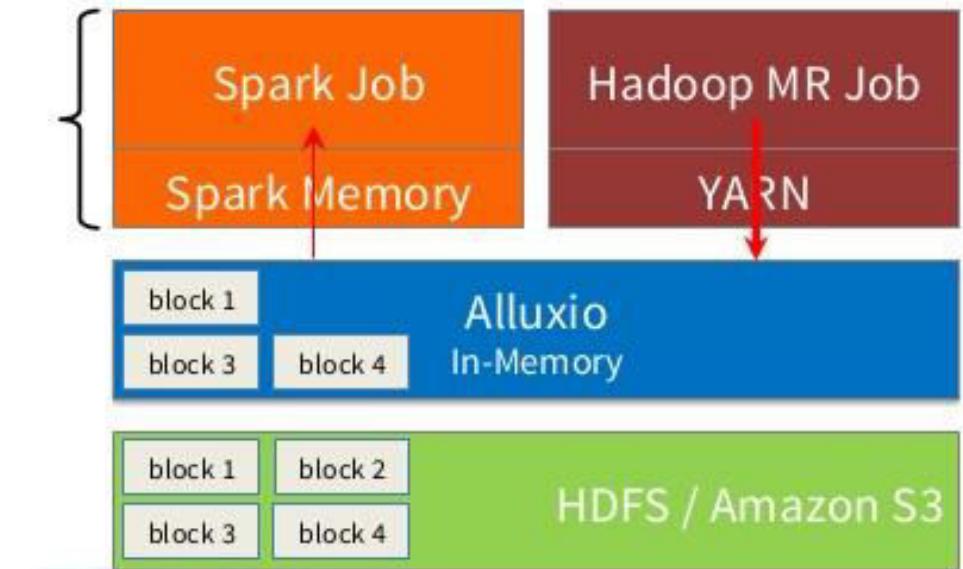
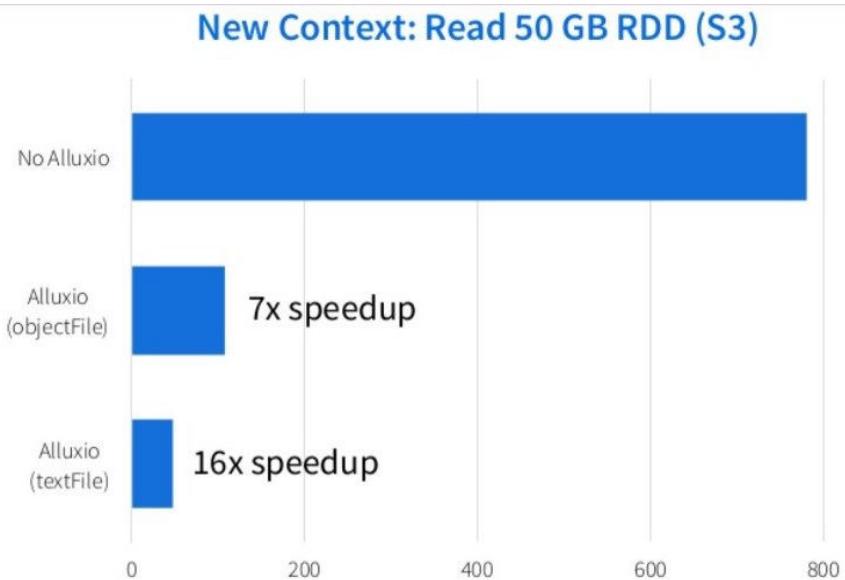


Spark Alluxio



- Memory speed virtual distributed storage
- Enables virtualized data across multiple types of storage
- Data locality guarantees
- Decoupling from storage

```
> val s = sc.textFile("alluxio-ft://stanbyHost:19998/LICENSE")
> val double = s.map(line => line + line)
> double.saveAsTextFile("alluxio-ft://activeHost:19998/LICENSE2")
```



Spark Training

Powered by





Using spark

- 1. How does it work**
2. Install
3. Spark shells
4. Notebooks
5. Run Java/Scala applications
6. Spark on Yarn

How does it work? (1/3)

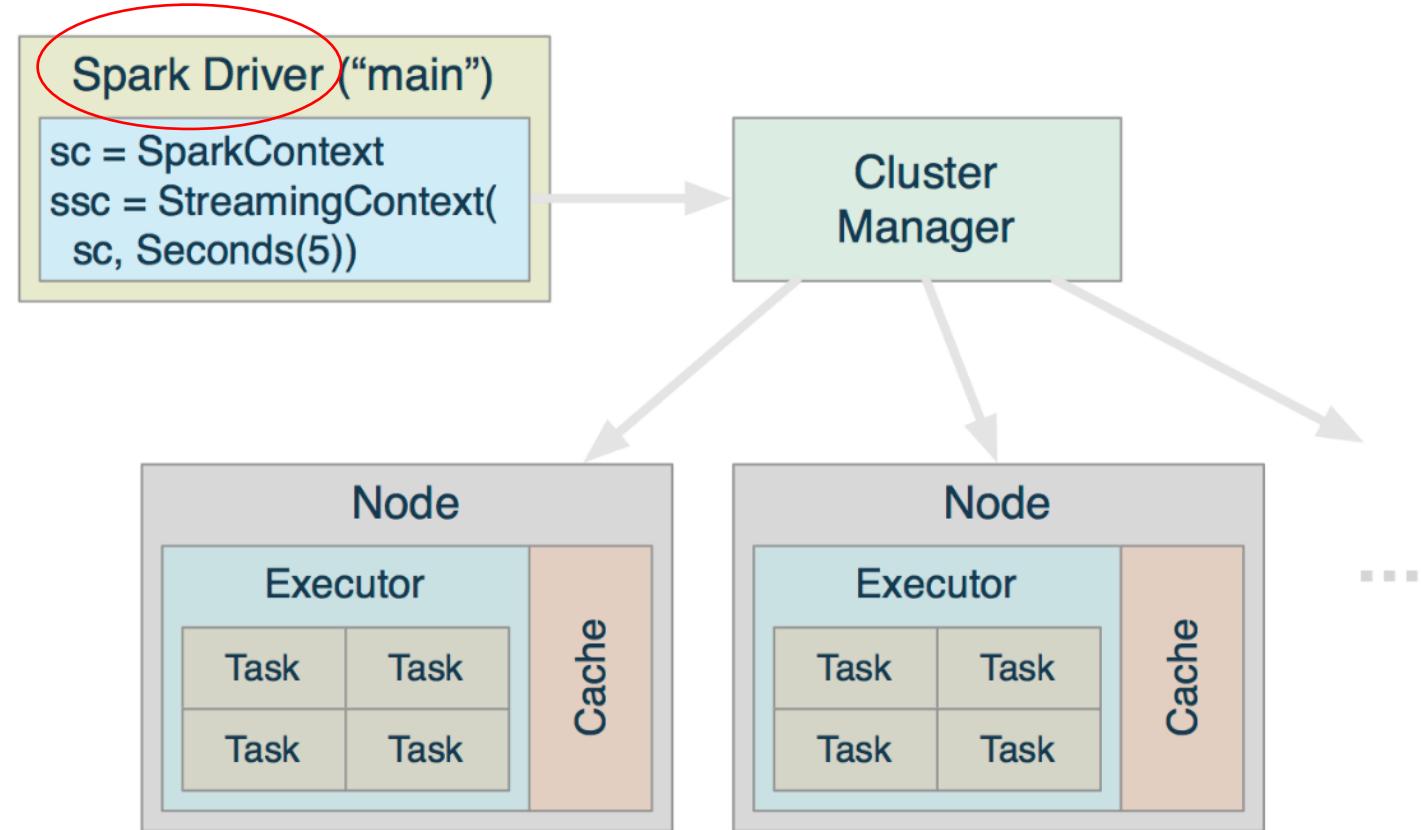


Spark is written in Scala and runs on the Java Virtual Machine.

Every Spark application consists of a **driver program**:

It contains the main function, defines distributed datasets on the cluster and then applies operations to them.

Driver programs access Spark through a `SparkContext` object.

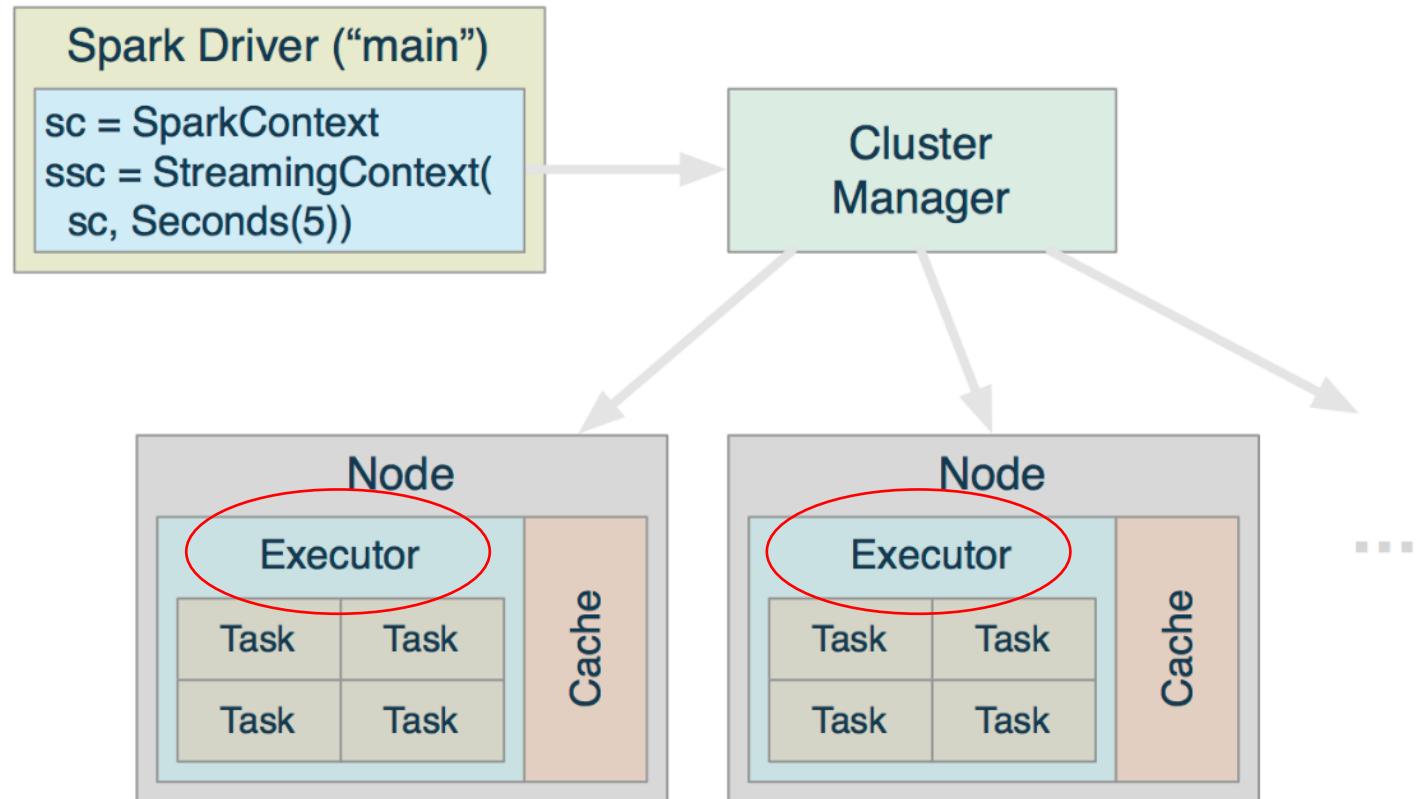


How does it work? (2/3)



To run the operations defined in the application the driver typically manages a number of nodes called **executors**.

These operations result in **tasks** that the executors have to perform.

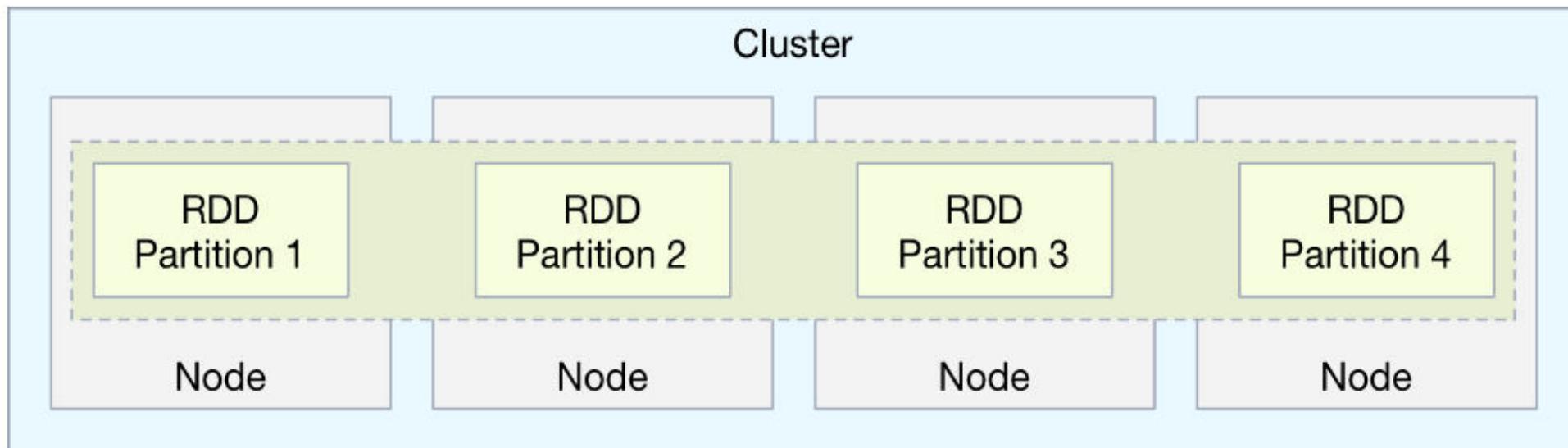


How does it work?(3/3)



Managing and manipulating datasets distributed over a cluster writing just a driver program without taking care of the distributed system is possible because of:

- Cluster managers (resources management, networking...)
- SparkContext (task definition from more abstract operations)
- RDD (Spark's programming main abstraction to represent distributed datasets)





Using spark

1. How does it work
- 2. Install**
3. Spark shells
4. Notebooks
5. Run Self-contained applications
6. Spark on Yarn

Download Spark on local machine



- Download page: <http://spark.apache.org/downloads.html>
- In 'bin' folder you can find
 - spark-shell
 - pyspark
 - sparkR
 - spark-submit



Using spark

1. How does it work
2. Install
- 3. Spark shells**
4. Notebooks
5. Run Self-contained applications
6. Spark on Yarn

Spark shells



- Interactive shells available
 - *spark-shell* - Extended Scala REPL (Spark imports already in scope)
 - *spark-sql* - SQL REPL like Hive's.
 - *pyspark* - Python shell.
 - *sparkR* - Shell with R API (since 1.4)
- Spark shells run in local system or distributed clusters

<path to spark folder>/bin/spark-shell --master local[2]

Demo



Using spark

1. How does it work
2. Install
3. Spark shells
- 4. Notebooks**
5. Run Self-contained applications
6. Spark on Yarn

Notebooks



- Data Scientists use interactive notebooks for exploring data, including graphing results.
- Developers are less attracted from notebooks but they also provide an excellent learning platform, as well as a useful workplace tool, even for developer tasks.
- There are several Spark-compatible Notebook systems available
 - Spark Notebook (<http://spark-notebook.io/>)
 - Zeppelin (<https://zeppelin.apache.org/>)

Demo



Using spark

1. How does it work
2. Install
3. Spark shells
4. Notebooks
- 5. Run Self-contained applications**
6. Spark on Yarn

Self-contained applications



To build self-contained application we need to

- Import spark-core and the other spark packages needed
- Initialize a Spark Context

To run a self-contained application we need to

- Use the spark-submit command

Import packages



- **Maven**

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.6.2</version>
  </dependency>
</dependencies>
```

- **SBT**

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.6.2" % "provided",
  "org.apache.spark" %% "spark-sql" % "1.6.2" % "provided"
)
```

Initializing the SparkContext



```
#Python
"""Spark Count.py"""
from pyspark import SparkContext
sc = SparkContext("local", "Simple App")
#Word count logic
```

```
//Scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SparkWordCount {
  def main(args: Array[String]) {
    // create Spark context with Spark configuration
    val sc = new SparkContext(new SparkConf().setAppName("Spark Count"))
    // Word count logic
    /* ... */
```

```
//Java
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.SparkConf;

public class JavaWordCount {
  public static void main(String[] args) {
    // create Spark context with Spark configuration
    JavaSparkContext sc = new JavaSparkContext(new SparkConf().setAppName("Spark Count"));
    // Word count logic
    /* ... */
```

Run Self-contained applications (1/2)



Python application

```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
--master local[4] \
SimpleApp.py
```

SBT based application

```
# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.11/simple-project_2.11-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
--class "SimpleApp" \
--master local[4] \
target/scala-2.11/simple-project_2.11-1.0.jar
```

Run Self-contained applications (2/2)



Maven based application

```
# Package a JAR containing your application
$ mvn package
...
[INFO] Building jar: {..}/{..}/target/simple-project-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
--class "SimpleApp" \
--master local[4] \
target/simple-project-1.0.jar
```



Using spark

1. How does it work
2. Install
3. Spark shells
4. Notebooks
5. Run Self-contained applications
- 6. Spark on Yarn**

Why Spark on YARN



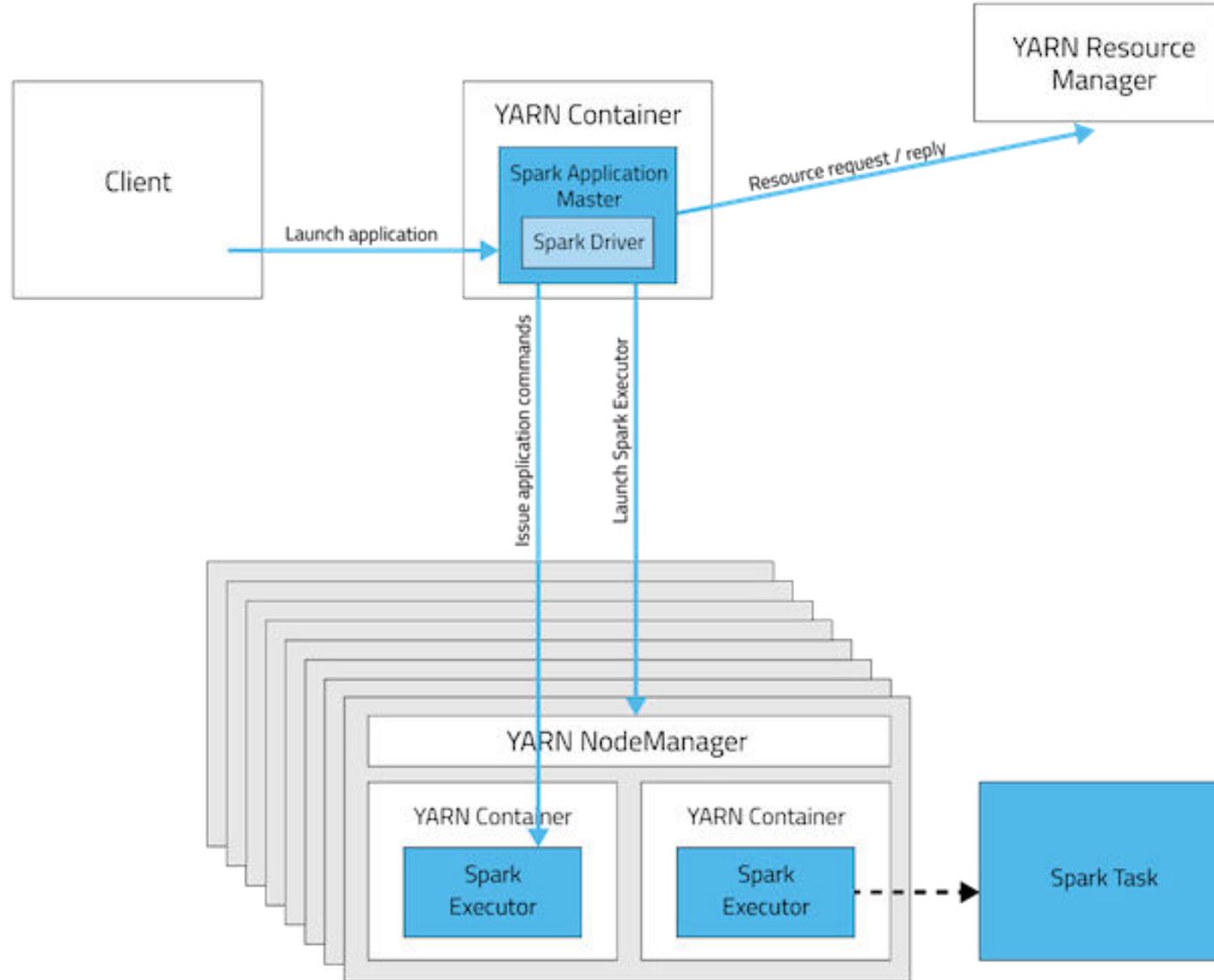
- YARN allows you to dynamically share and centrally configure the same pool of cluster resources between all frameworks that run on YARN.
- You can take advantage of [all the features of YARN schedulers](#) for categorizing, isolating, and prioritizing workloads.
- Spark standalone mode requires each application to run an executor on every node in the cluster, whereas with YARN, you choose the number of executors to use.
- Finally, YARN is the only cluster manager for Spark that supports security. With YARN, Spark can run against Kerberized Hadoop clusters and uses secure authentication between its processes.

Spark on YARN setup

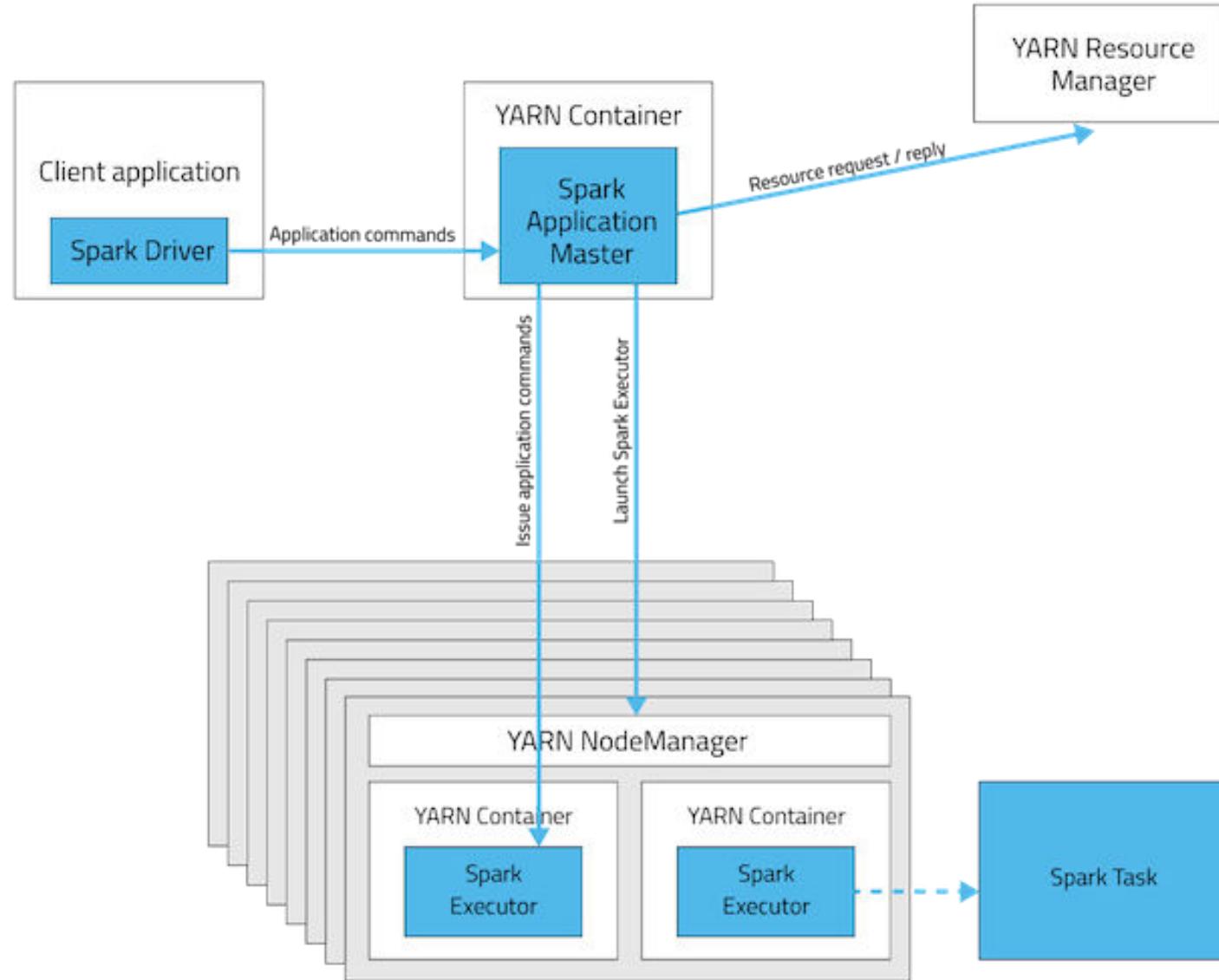


- Available from Spark 0.6.0
- HADOOP_CONF_DIR or YARN_CONF_DIR must point to the directory which contains the (client side) configuration files for the Hadoop cluster.
 - Confs used to write to HDFS and connect to YARN Resource Manager
- Set parameter --master to yarn
 - The Resource Manager URL is taken from the Hadoop configuration
- Example command
 - *./bin/spark-submit --class path.to.your.Class --master **yarn** --deploy-mode **cluster** [options] <app jar> [app options]*

Spark on YARN: Cluster Mode



Spark on YARN: Client Mode



Spark on YARN: Client Mode



Mode	YARN Client Mode	YARN Cluster Mode
Driver runs in	Client	ApplicationMaster
Requests resources	ApplicationMaster	ApplicationMaster
Starts executor processes	YARN NodeManager	YARN NodeManager
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers
Supports Spark Shell	Yes	No

Spark Training

Powered by





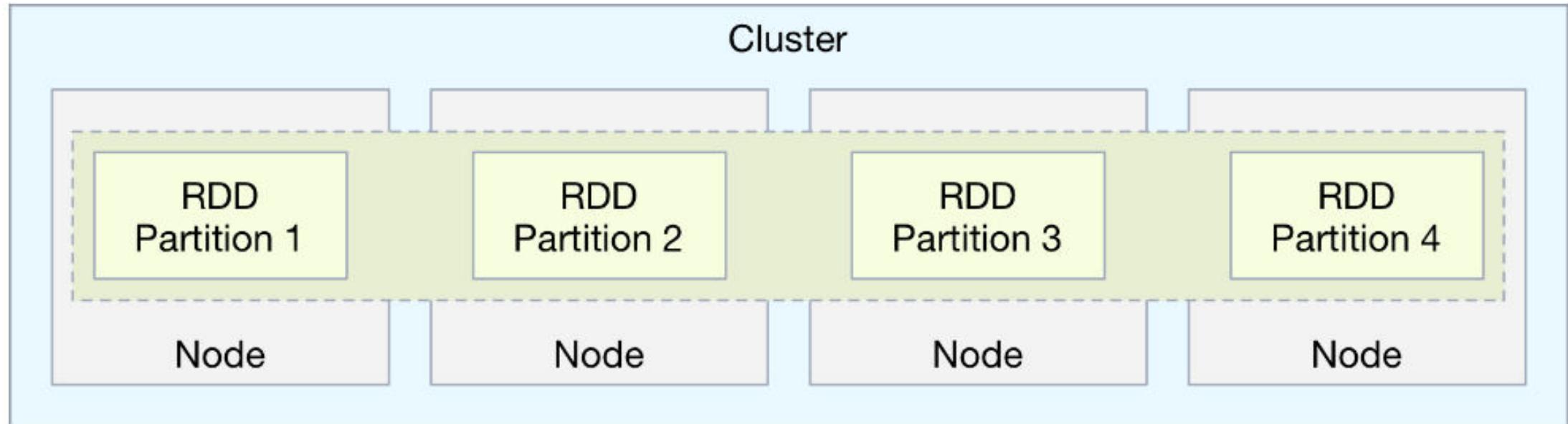
Programming with RDDs

- 1. RDD basics**
2. RDD Operations
3. Lazy evaluation
4. RDD Persistence
5. Spark Job Scheduling
6. Shared variables

What is a RDD?



A **Resilient Distributed Dataset** is an immutable collection of elements splitted in multiple **partitions** distributed over a number of nodes.



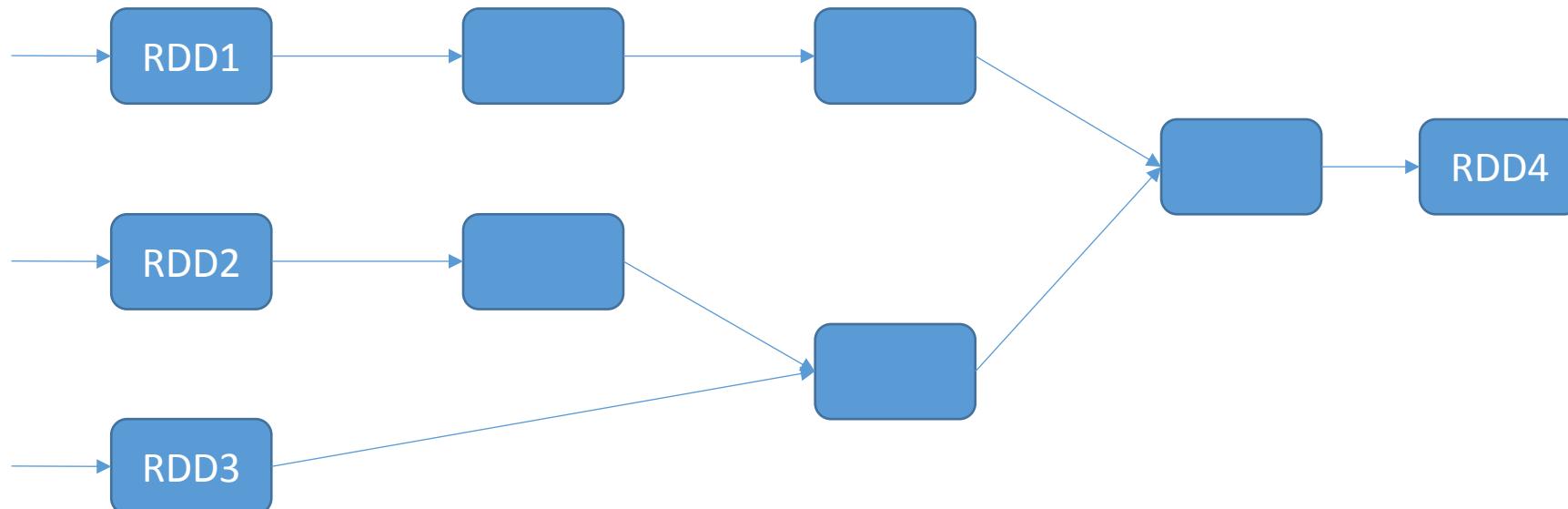
RDD



RDDs are:

- **Immutable** - Each step of a data pipeline will create a new RDD
- **Lazy** – The actual data is processed only when the results are requested
- **Aware of its parents and ancestors** – the «lineage» that brings to that RDD is known
- **Resilient** – A partition of a RDD can be reconstructed if lost

A Direct Acyclic Graph (DAG) of computation defines the data pipeline in Spark



Creating a RDD



Two ways to create a RDD

1. Loading an external dataset

```
#Python
lines = sc.textFile("/path/to/InputText.txt")
//Scala
val lines = sc.textFile("/path/to/InputText.txt")
//Java
JavaRDD<String> lines = sc.textFile("/path/to/InputText.txt");
```

2. Parallelizing a collection already present in the driver program

```
#Python
lines = sc.parallelize(["first", "second"])
//Scala
val lines = sc.parallelize(List("first", "second"))
//Java
JavaRDD<String> lines = sc.parallelize(Arrays.asList("first", "second"));
```

Demo



Programming with RDDs

1. RDD basics
2. **RDD Operations**
3. Lazy evaluation
4. RDD Persistence
5. Spark Job Scheduling
6. Shared variables

RDD Operations



Two types of operations:

- Transformations
 - RDDs -> RDD
- Actions
 - RDD -> Result to Driver Program
 - Save RDD on External storage

```
//Scala
val sentences = sc.parallelize(List("This is a Spark course", "This
is another sentence"))
val words = sentences.flatMap(sentence => sentence.split(" "))
val lowerCaseWords = words.filter(word => word.charAt(0).isLower)
lowerCaseWords.take(2).foreach(println)
```



(Little detour: passing functions) 1/2



- Most of transformations and many actions depend on passing in functions used by Spark to compute data
- In the three main languages that use Spark APIs it happens differently
 - Python
 - Lambda expressions
 - Top-level functions
 - Locally defined functions
 - Scala
 - Inline functions
 - References to methods
 - Static functions
 - Java
 - Objects that implement Spark's function interfaces
 - Lambda expressions (Java 8+)

(Little detour: passing functions) 2/2



```
#Python
errorsRDD = rdd.filter(lambda s: "error" in s)
# or
def containsError(s):
    return "error" in s
errorsRDD = rdd.filter(containsError)
```

```
//Scala
val errorsRDD = rdd.filter(s => s.contains("error"))
//or
def containsError = (s: String) => s.contains("error")
val errorsRDD = rdd.filter(containsError)
```

```
//Java
RDD<String> errorsRDD = rdd.filter( new Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
});
//or
class ContainsError implements Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
}
RDD<String> errors = rdd.filter(new ContainsError());
// or in Java 8
RDD<String> errors = rdd.filter( s -> s.contains("error"))
```

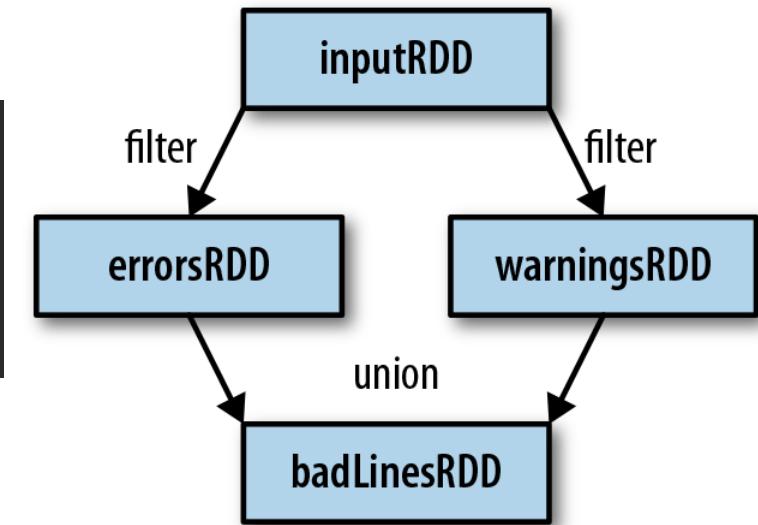
Be aware of the object
that contains the
function or fields used
in it!

RDD Transformations



- RDDs -> RDD
- Lazy evaluation
- Can operate on any number of input RDDs
- Spark keep track of the «Lineage graph» of RDDs
- Narrow vs. Wide

```
#Python
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```



Examples Input RDD : {1, 2, 3, 4, 4, 5}

Frequently used transformations



- **map()**
 - RDD[T] -> RDD[S]
 - One to one transformation
 - Need function T -> S

```
//Scala  
val mapResult = input.map(x => x + 2)
```

input: {1, 2, 3, 4, 4, 5}

map()

mapResult: {3, 5, 5, 6, 6, 7}

Frequently used transformations



- **flatMap()**
 - $\text{RDD}[T] \rightarrow \text{RDD}[S]$
 - One to many transformation
 - Need function $T \rightarrow \text{TraversableOnce}[S]$

```
//Scala
val flatMapResult = input.flatMap{x => if(x % 2 == 0) Seq(x, x/2), else Seq(x)}
```

input: {1, 2, 3, 4, 4, 5}

flatMap()

flatMapResult: {1, 2, 1, 3, 4, 2, 4, 2, 5}

Frequently used transformations



- **filter()**
 - RDD[T] -> RDD[T]
 - One to Zero-One transformation
 - Need function T -> Boolean

```
//Scala  
val filterResult = input.filter( x => x % 3 == 0)
```

input: {1, 2, 3, 4, 4, 5}

filter()

filterResult: {3}

Frequently used transformations



- **distinct()**
 - $\text{RDD}[T] \rightarrow \text{RDD}[T]$
 - Doesn't need any function

```
//Scala  
val distinctResult = input.distinct()
```

input: {1, 2, 3, 4, 4, 5}

distinct()

distinctResult: {1, 2, 3, 4, 5}

Frequently used transformations



- **union()**
 - $\text{RDD}[T]^2 \rightarrow \text{RDD}[T]$
 - Doesn't need any function
 - One RDD passed as parameter

```
//Scala  
val unionResult = input.union(newInput)
```

input: {1, 2, 3, 4, 4, 5}

newInput: {1, 4, 6}

union()

unionResult: {1, 2, 3, 4, 4, 5, 1, 4, 6}

Frequently used transformations



- **keyBy()**
 - $\text{RDD}[T] \rightarrow \text{RDD}[K, T]$
 - Requires a function $T \rightarrow K$

```
//Scala  
val keyByResult = input.keyBy(x => x / 2)
```

input: {1, 2, 3, 4, 4, 5}

keyBy()

*keyByResult: {
(0, 1), (1, 2), (1, 3), (2, 4), (2, 4), (2, 5)
}*

Transformations on Pair RDDs



- **mapValues()**
 - $\text{RDD}[(\text{K}, \text{V})] \rightarrow \text{RDD}[(\text{K}, \text{T})]$
 - Need a function $\text{V} \rightarrow \text{T}$

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

// rdd1: RDD[(String, Int)]
val rdd2 = rdd1.mapValues( x => x.toString)
// rdd2 : RDD[(String, String)]
```

rdd1: {
('A', 2), ('B', 3), ('E', 1), ('Z', 1)
}

mapValues()

rdd2: {
('A', '2'), ('B', '3'), ('E', '1'), ('Z', '1')
}

Transformations on Pair RDDs



- **reduceByKey()**
 - $\text{RDD}[(\text{K}, \text{V})] \rightarrow \text{RDD}[(\text{K}, \text{V})]$
 - Need a function $V^2 \rightarrow V$

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

val reduceByKeyResult = inputKv.reduceByKey((a, b) => a + b)
```

inputKv: {
 ('A', 2), ('B', 3), ('A', 1),
 ('C', 1), ('C', 10), ('D', 5)
}

reduceByKey()

reduceByKeyResult: {
 ('A', 3), ('B', 3), ('C', 11),
 ('D', 5)
}

Transformations on Pair RDDs



- **aggregateByKey()**
 - $\text{RDD}[(K,V)] \rightarrow \text{RDD}[(K,U)]$
 - Need a function $(U, V) \rightarrow U$
 - Need a function $U^2 \rightarrow U$
 - Need a zero-value of type U

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

val aggregateByKeyResult = inputKv.aggregateByKey
(0) ( (acc: Int, elem: String) => acc + Integer.parseInt(elem), (a,b) => a + b)
```

inputKv: {
 ('A', '2'), ('B', '3'), ('A', '1'),
 ('C', '1'), ('C', '10'), ('D', '5')
}

aggregateByKey()

aggregateByKeyResult: {
 ('A', 3), ('B', 3), ('C', 11),
 ('D', 5)
}

Transformations on Pair RDDs



- **join()**
 - $(RDD[(K,V)], RDD[(K,U)]) \rightarrow RDD[(K,(V, U))]$
 - Need the second RDD as parameter
 - The two RDD must have the same type as key

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

// rdd1: RDD[ (String, S) ]
// rdd2: RDD[ (String, T) ]
val joinResult = rdd1.join(rdd2)
// joinResult: RDD[ (String, (S,T)) ]
```

rdd1: {
 ('A', 2), ('B', 3)}

rdd1.join(rdd2)

joinResult: {
 ('A', (2, 1)), ('B', (3, 4)), ('B', (3,5))}

rdd2: {('A', 1), ('B', 4)
 ('C', 1), ('C', 10), ('B', 5)}

Transformations on Pair RDDs



- **leftOuterJoin()**
 - $(RDD[(K,V)], RDD[(K,U)]) \rightarrow RDD[(K,(V, Option[U]))]$
 - Need the second RDD as parameter
 - The two RDD must have the same type as key

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

// rdd1: RDD[ (String, S) ]
// rdd2: RDD[ (String, T) ]
val joinResult = rdd1.leftOuterJoin(rdd2)
// joinResult: RDD[ (String, (S,Option[T])) ]
```

rdd1: {
 ('A', 2), ('B', 3), ('E', 1), ('Z', 1)}

rdd1.leftOuterJoin(rdd2)

rdd2: {('A', 1), ('B', 4)
 ('C', 1), ('C', 10), ('B', 5)}

joinResult: {
 ('A', (2, Some(1))), ('B', (3, Some(4))),
 ('B', (3, Some(5))), ('E', (1, None)),
 ('Z', (1, None))}

Transformations on Pair RDDs



- **cogroup()**

- $(RDD[(K,V)], RDD[(K,W)]) \rightarrow RDD[(K,(Iterable[V], Iterable[W]))]$
- Need the second RDD as parameter
- The two RDD must have the same type as key

```
//Scala
import org.apache.spark.rdd.PairRDDFunctions

// rdd1: RDD[(String, S)]
// rdd2: RDD[(String, T)]
val cogroupResult = rdd1.cogroup(rdd2)
// cogroupResult: RDD[(String, (Iterable[S], Iterable[T]))]
```

rdd1: {
('A', 2), ('B', 3), ('E', 1), ('Z', 1)}

rdd1.cogroup (rdd2)

rdd2: {('A', 1), ('B', 4)
('C', 1), ('C', 10), ('B', 5)}

cogroupResult: {
('A', ([2], [1])), ('B', ([3], [4,5])),
('E', ([1], [])), ('Z', ([1], [])),
('C', ([], [1,10])))
}

Other transformations

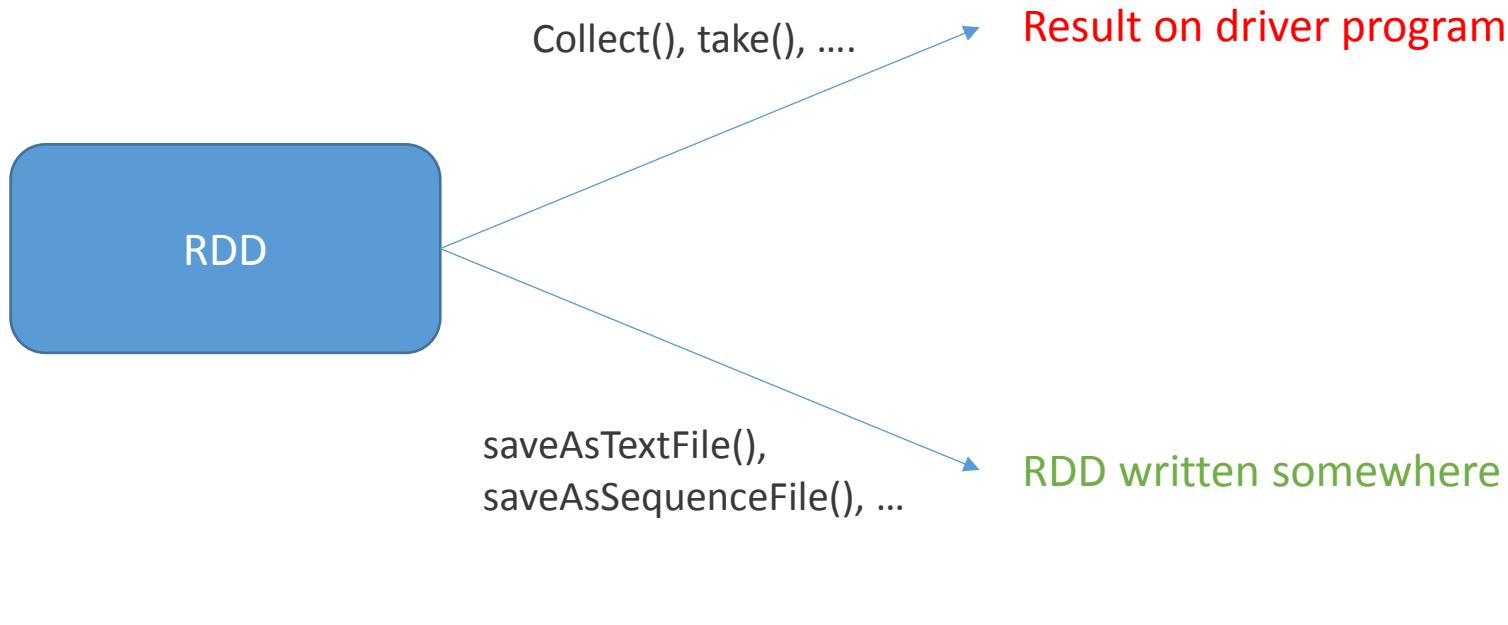


- mapPartitions(), mapPartitionsWithIndex()
- intersection(), subtract(), cartesian()
- groupByKey(), sortByKey()
- coalesce(), repartition()
- flatMapValues()
- partitionBy()
- Complete API doc:
 - <http://spark.apache.org/docs/1.6.3/api/scala/index.html#org.apache.spark.RDD>
 - <http://spark.apache.org/docs/1.6.3/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions>

RDD Actions



- RDDs -> Result on Driver OR rdd saving operations
- Force the evaluation of needed transformations
- To each action corresponds a Spark Job



Common actions



- **collect()**
 - Return the RDD on the driver as a collection
 - Be aware of RDD dimension!
- **take(n)**
 - Return n elements from the RDD to the driver as a collection
- **reduce()**
 - RDD[V] -> V
 - Need a function $V^2 \rightarrow V$

```
//Scala
val dataOnDriver = input.collect()
val rddSampleOnDriver = input.take(2)
val sumOnDriver = input.reduce((a, b) => a + b)
```

	collect()	<i>dataOnDriver: {1, 2, 3, 4, 4, 5}</i>
<i>input: {1, 2, 3, 4, 4, 5}</i>	take()	<i>rddSampleOnDriver: {2, 4}</i>
	reduce()	<i>sumOnDriver: 19</i>

Demo



Programming with RDDs

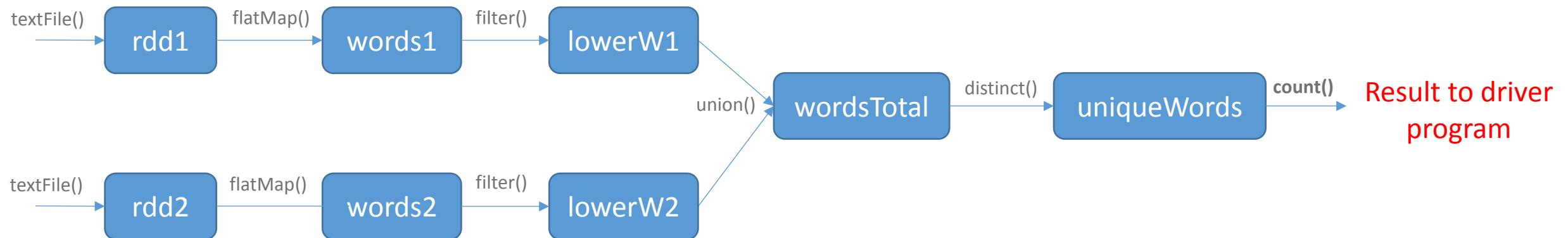
1. RDD basics
2. RDD Operations
3. **Lazy evaluation**
4. RDD Persistence
5. Spark Job Scheduling
6. Shared variables

Lazy evaluation



```
//Scala
val rdd1 = sc.textFile("/path/to/text1.txt")
val rdd2 = sc.textFile("/path/to/text2.txt")
val words1 = rdd1.flatMap(s => s.split(" "))
val words2 = rdd2.flatMap(s => s.split(" "))
val lowerW1 = words1.filter(w => w.toLowerCase() == w)
val lowerW2 = words2.filter(w => w.toLowerCase() == w)
val wordsTotal = lowerW1.union(lowerW2)
val uniqueWords = wordsTotal.distinct()

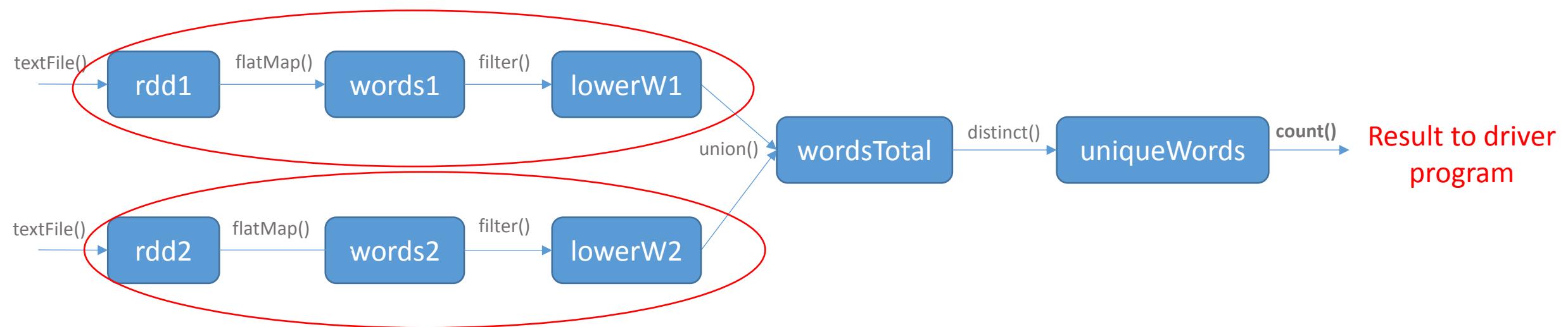
println(s"There are ${uniqueWords.count()} lowercase distinct words in the 2 text files")
```



Lazy evaluation advantages (1/3)



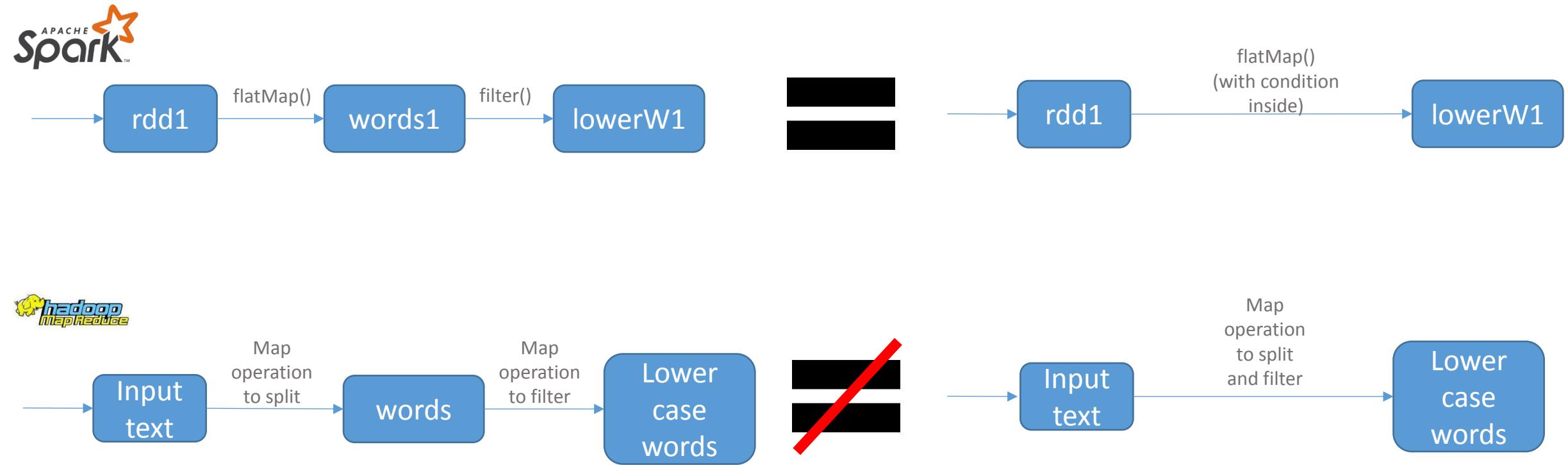
- **Performance**
 - Spark can chain operations
 - No multiple pass over data if not required



Lazy evaluation advantages (2/3)



- **Easy implementation**
 - Possible optimization are inferred by Spark
 - Easier w.r.t. MapReduce



Lazy evaluation advantages (3/3)



- **Fault tolerance**
 - Spark knows how to build each partition
 - Spark can recover lost partitions without recomputing the entire RDD
 - Recover happens also in parallel
- Instead of distributed systems, based on mutable objects and strict evaluation paradigms, provide fault tolerance by logging updates or duplicating data across machines. Spark doesn't have to store any of this information.



Programming with RDDs

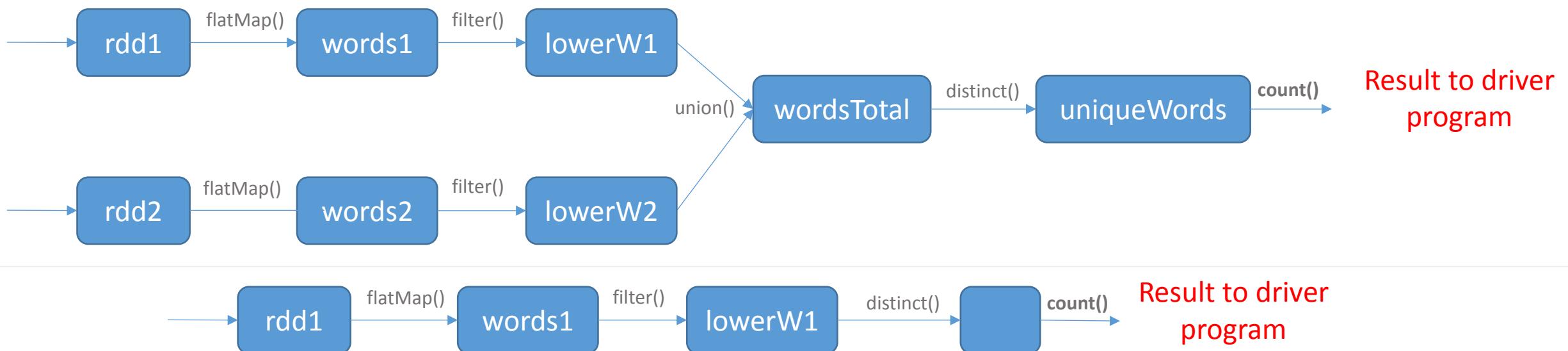
1. RDD basics
2. RDD Operations
3. Lazy evaluation
- 4. RDD Persistence**
5. Spark Job Scheduling
6. Shared variables

Previous example change



```
//Scala
val rdd1 = sc.textFile("/path/to/text1.txt")
val rdd2 = sc.textFile("/path/to/text2.txt")
val words1 = rdd1.flatMap(s => s.split(" "))
val words2 = rdd2.flatMap(s => s.split(" "))
val lowerW1 = words1.filter(w => w.toLowerCase())
val lowerW2 = words2.filter(w => w.toLowerCase())
val wordsTotal = lowerW1.union(lowerW2)
val uniqueWords = wordsTotal.distinct()

println(s"There are ${uniqueWords.count()} lowercase distinct words in the 2 text files")
//New line
println(s"There are ${lowerW1.distinct().count()} lowercase distinct words in the first text file")
```



Same computation done two times!



```
//Scala
val rdd1 = sc.textFile("/path/to/text1.txt")
val rdd2 = sc.textFile("/path/to/text2.txt")
val words1 = rdd1.flatMap(s => s.split(" "))
val words2 = rdd2.flatMap(s => s.split(" "))
val lowerW1 = words1.filter(w => w.toLowerCase())
val lowerW2 = words2.filter(w => w.toLowerCase())
val wordsTotal = lowerW1.union(lowerW2)
val uniqueWords = wordsTotal.distinct()

println(s"There are ${uniqueWords.count()} lowercase distinct words in the 2 text files")
//New line
println(s"There are ${lowerW1.distinct().count()} lowercase distinct words in the first text file")
```

'lowerW1' RDD is computed from scratch 2 times in that way!

RDD Persistence



- RDD can be persisted in memory and/or disk to be re-used multiple times
- Three options for memory management
 - In memory as deserialized Java objects
 - In memory as serialized data
 - On disk
- RDD can be unpersisted manually or by Spark with a LRU strategy

	In memory deserialized	In memory serialized	On disk
Space required	High	Low	Low
Performance	High	Medium	Very Low

The best option? It depends

RDD Persistence API



- Methods available
 - persist()
 - cache()
 - unpersist()
- Storage levels
 - MEMORY_ONLY
 - MEMORY_AND_DISK
 - MEMORY_ONLY_SER
 - MEMORY_AND_DISK_SER
 - DISK_ONLY
 - MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

```
//Scala
val persistedRdd = rdd.persist(StorageLevel.MEMORY_ONLY)
//equivalent to
val persistedRdd = rdd.cache()
```

Previous example with persistence



```
//Scala
val rdd1 = sc.textFile("/path/to/text1.txt")
val rdd2 = sc.textFile("/path/to/text2.txt")
val words1 = rdd1.flatMap(s => s.split(" "))
val words2 = rdd2.flatMap(s => s.split(" "))
val lowerW1 = words1.filter(w => w.equals(w.toLowerCase)).cache()
val lowerW2 = words2.filter(w => w.equals(w.toLowerCase))
val wordsTotal = lowerW1.union(lowerW2)
val uniqueWords = wordsTotal.distinct()

println(s"There are ${uniqueWords.count()} lowercase distinct words in the 2 text
files")
//New line
println(s"There are ${lowerW1.distinct().count()} lowercase distinct words in the first
text file")
```

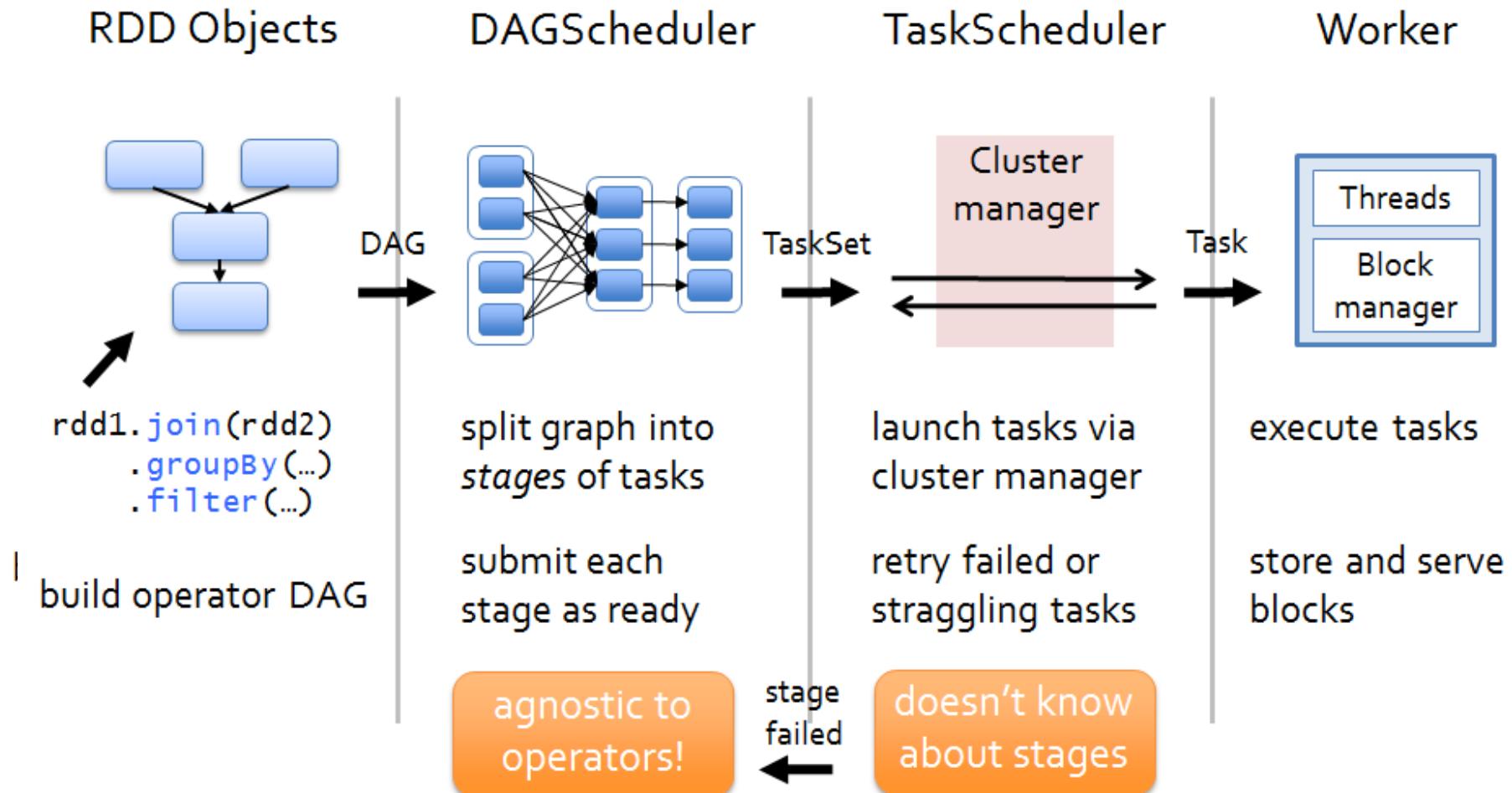
Demo



Programming with RDDs

1. RDD basics
2. RDD Operations
3. Lazy evaluation
4. RDD Persistence
- 5. Spark Job Scheduling**
6. Shared variables

Anatomy of a Spark application

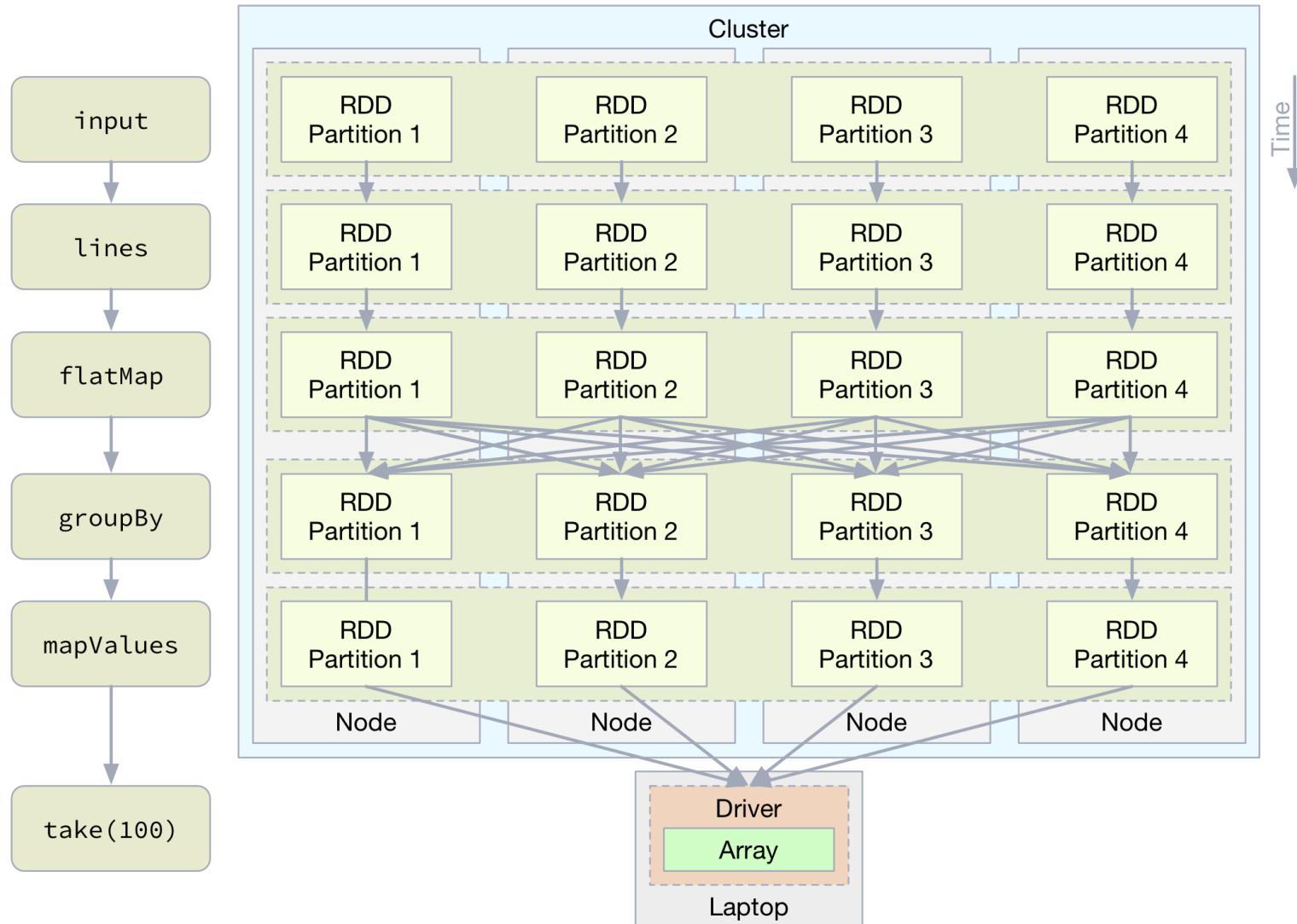


Jobs, Stages, Tasks

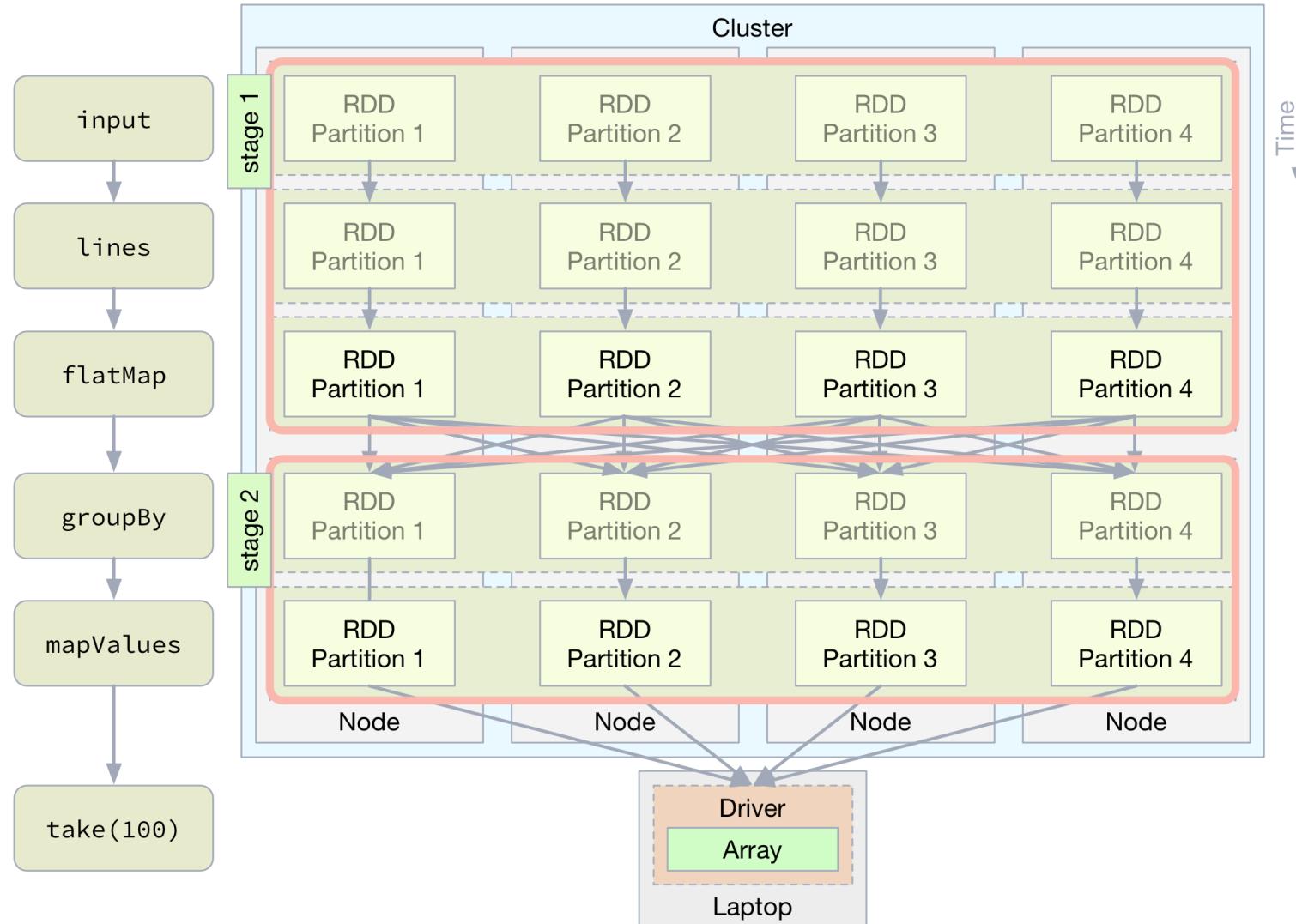


- A Spark Application consists of a driver process, with high-level logic written in it
- Each action call results in a **Spark Job**
- Every Spark Job is splitted in different **Stages**, one for each wide transformation
- A stage is a set of **tasks**, that are equal across a stage and distributed to the workers
- A task is assigned to an executor to produce a partition.

RDD Lineage



RDD Lineage



Narrow vs. Wide (1/2)

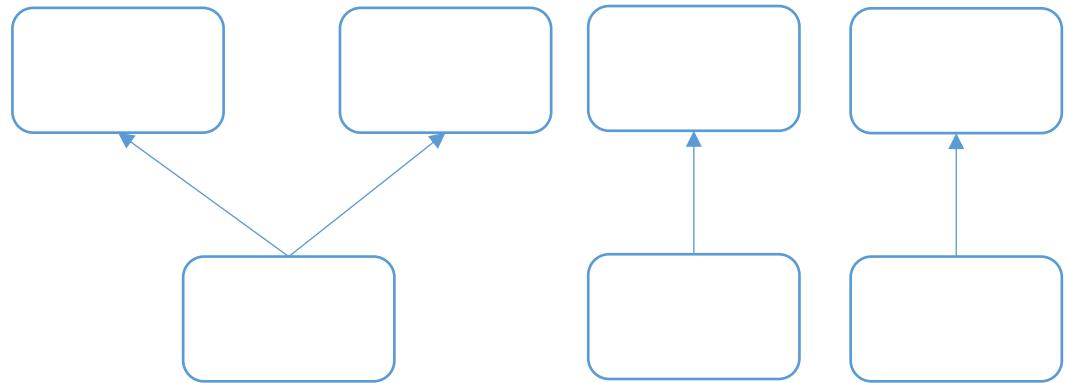


- **Narrow** transformations
 - Depend on a known set of partitions in the parent RDD
 - These partitions can be determined at design time
 - Examples: filter(), map(), ...
- **Wide** transformations
 - Cannot be executed on arbitrary rows
 - Require data to be partitioned in a particular way
 - Examples: sort(), join(), ...
- **Shuffle**: process of moving records in an RDD to accomodate partitioning requirement
 - Shuffles are expensive, proportionally with the amount of data
 - Not always wide dependencies bring to shuffle

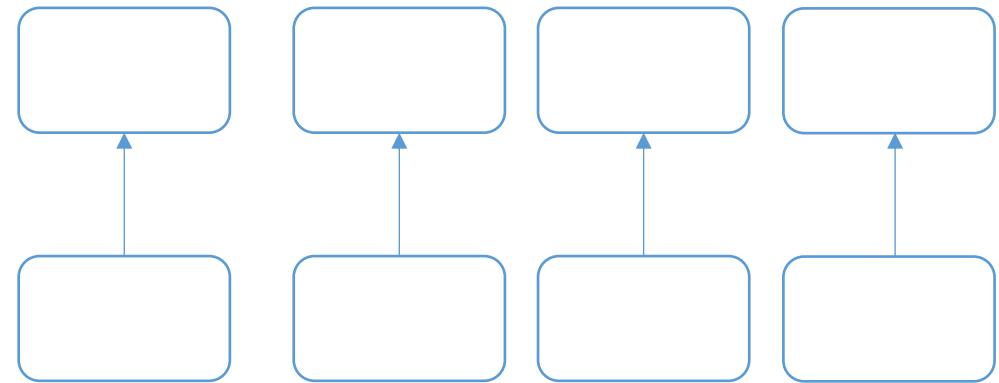
Narrow vs. Wide (2/2)



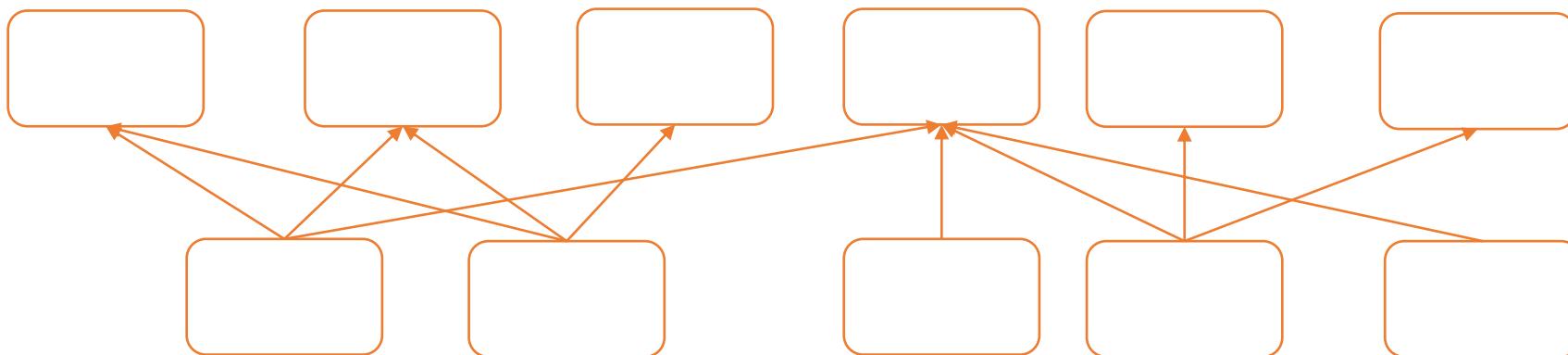
- Narrow transformations



OR



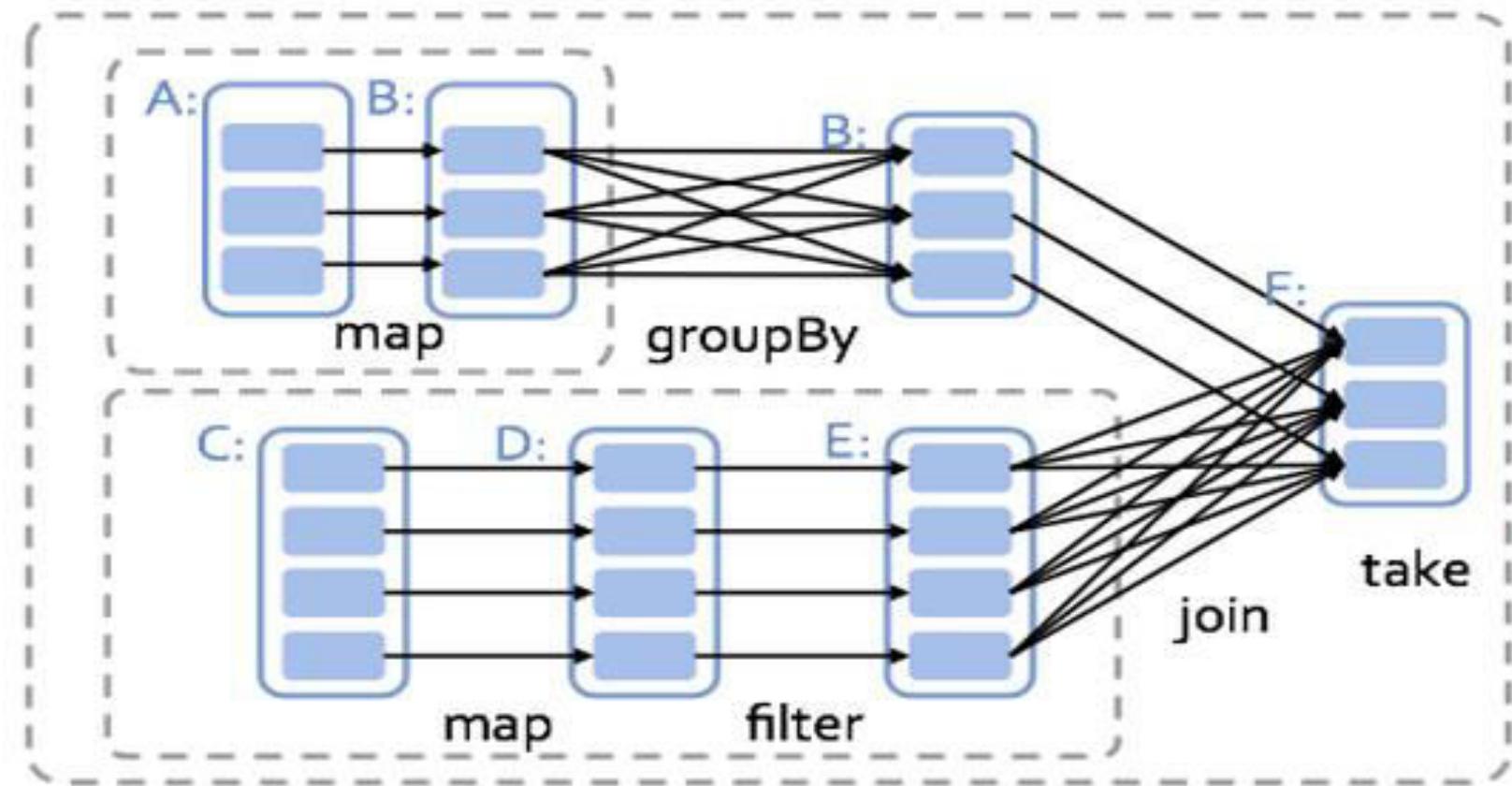
- Wide transformations



Why wide transformations are expensive?



- RDDs are only materialized at the end of each stage, when wide transformations are needed, that cause shuffle data in memory and/or disk.





Programming with RDDs

1. RDD basics
2. RDD Operations
3. Lazy evaluation
4. RDD Persistence
5. Spark Job Scheduling
6. **Shared variables**

Shared variables



- Functions passed to Spark (i.e. to a map) can use variables defined outside them
- These variables are copied to each task, no update is propagated back to the driver
- Two ways to relax this restriction:
 - **Accumulators** -> to perform aggregation of results
 - **Broadcast** -> Send large, read-only values to executors

Accumulator



Simple syntax to aggregate values from workers to the driver program

- *SparkContext.accumulator(**initial value **)* to initialize the accumulator.
- The type is a *org.apache.spark.Accumulator[T]* . *T* is inferred from the initial value.
- Accumulator can be updated using its **+=** method (*add* in Java)
- In driver program the accumulator value can be accessed by the property *value*
- Tasks cannot access the accumulator value
- Accumulators recommended only for debug purposes
- Custom accumulators can be defined

Accumulator example



```
//Scala
val rdd = sc.textFile("/path/to/text1.txt")
val blankLines = sc.accumulator(0)
val words = rdd.flatMap{
  case x: String =>
    if(x == "") blankLines += 1
    x.split(" ")
}
println(s"There are ${words.count()} words")
println(s"There are ${blankLines} blank lines")
```

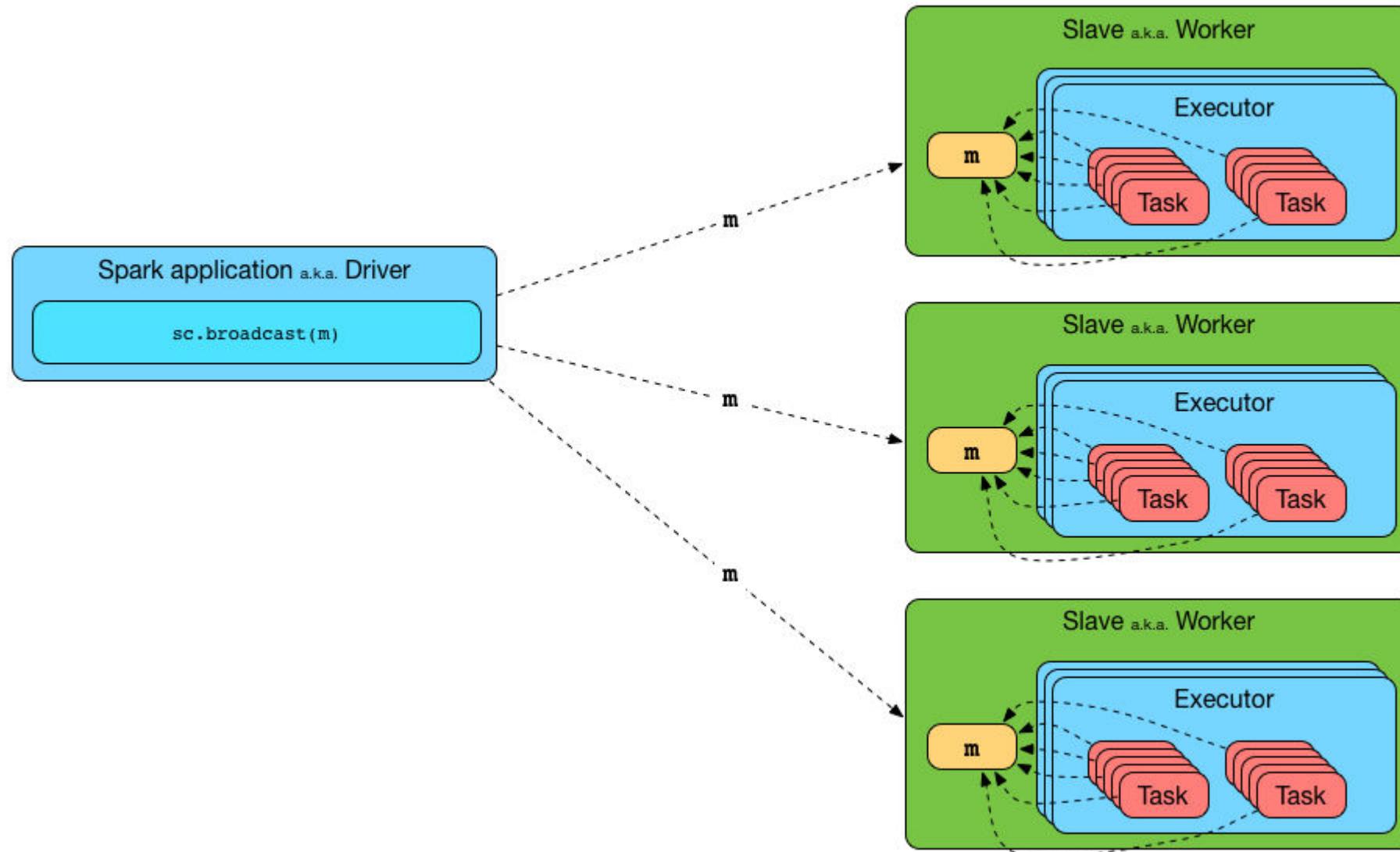
Broadcast



Broadcasts allow to send large, read-only values to all the worker nodes to be used in one or more operations

- A *Broadcast[T]* can be created by calling *SparkContext.broadcast*
- Any *T* type works as long as it is Serializable
- Value is accessed by calling the *value* property (*value()* method in Java)
- Variables are sent to each node **only once**.
- Modifications are not propagated to other nodes (read-only)

Broadcast concept



Broadcast Example



```
//Scala
val inputCities = sc.parallelize(Seq("Nice", "Warsaw", "Rome", "Amsterdam"))
val cityCountryMap = Map("rome" -> "IT", "paris" -> "FR", "milan" -> "IT", "madrid" -> "ES", "warsaw" -> "PL")

val mapBroadcast = sc.broadcast(cityCountryMap)

val italianCitiesInInput = inputCities.filter{
  case city => mapBroadcast.value.get(city.toLowerCase).exists( _ == "IT")
}

println(s"In the input list there is/are ${italianCitiesInInput.count()} italian city/cities")
```

Demo

Spark Training

Powered by





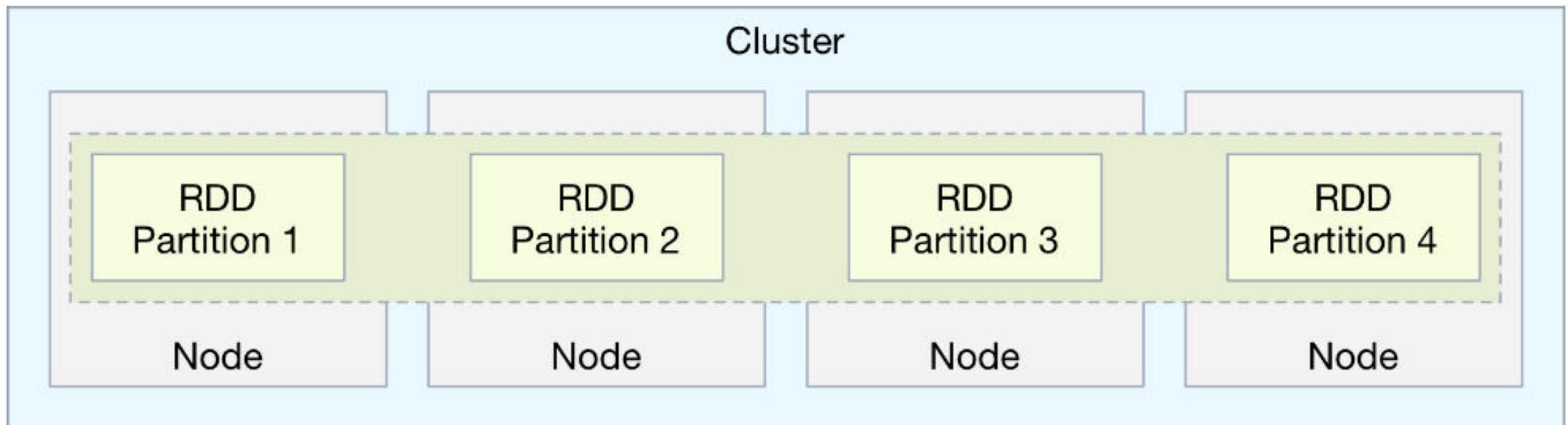
Partitioning in Spark

- 1. How it works**
2. Partitioning on Key-Value RDDs
3. Optimize partitioning

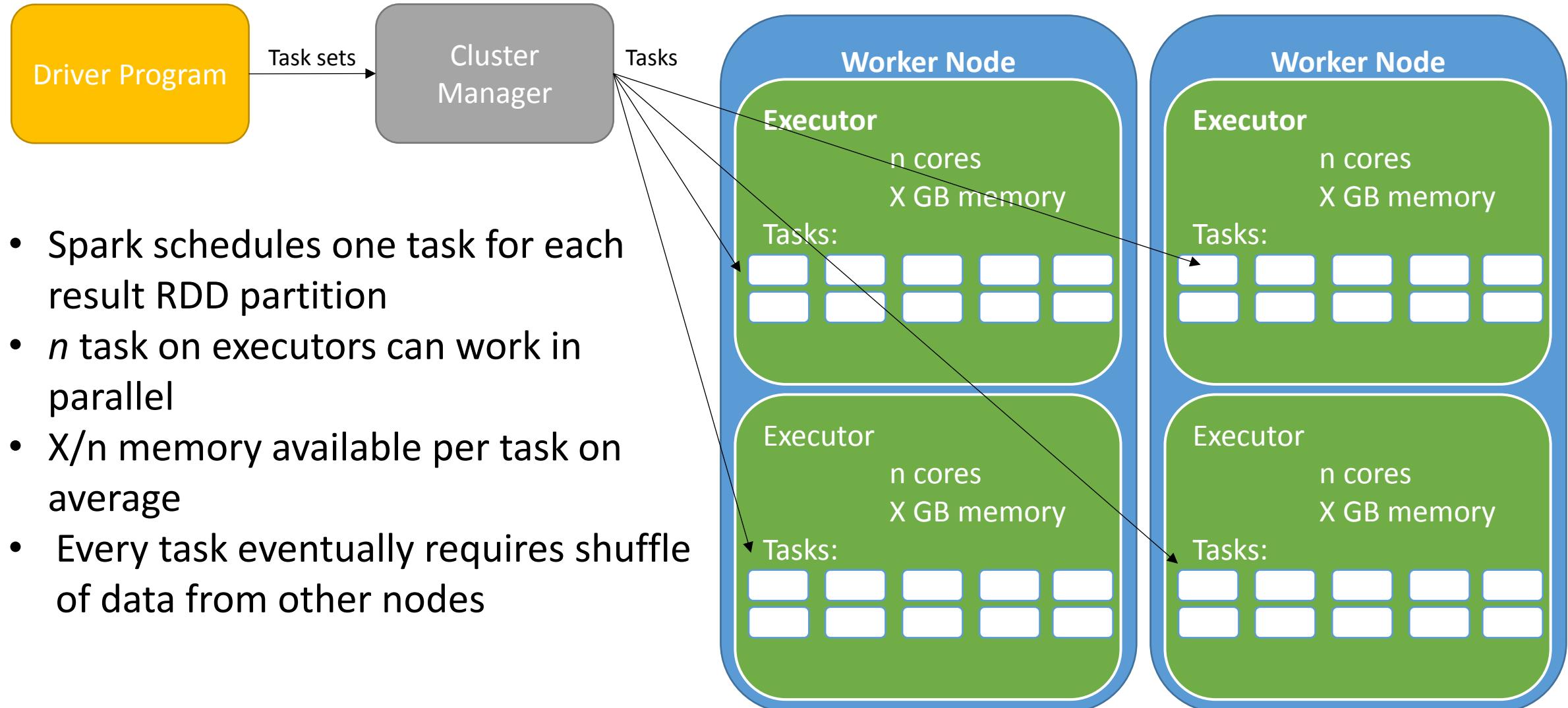
RDD Partitioning



The data of a RDD is split into multiple partitions, each of them resides on a node in the cluster.



One task per partition



Partitioning transformations



Two main dedicated transformations:

- `coalesce(n)`
- `repartition(n)`

Additional parameter ‘`numPartitions`’ for some transformations is available

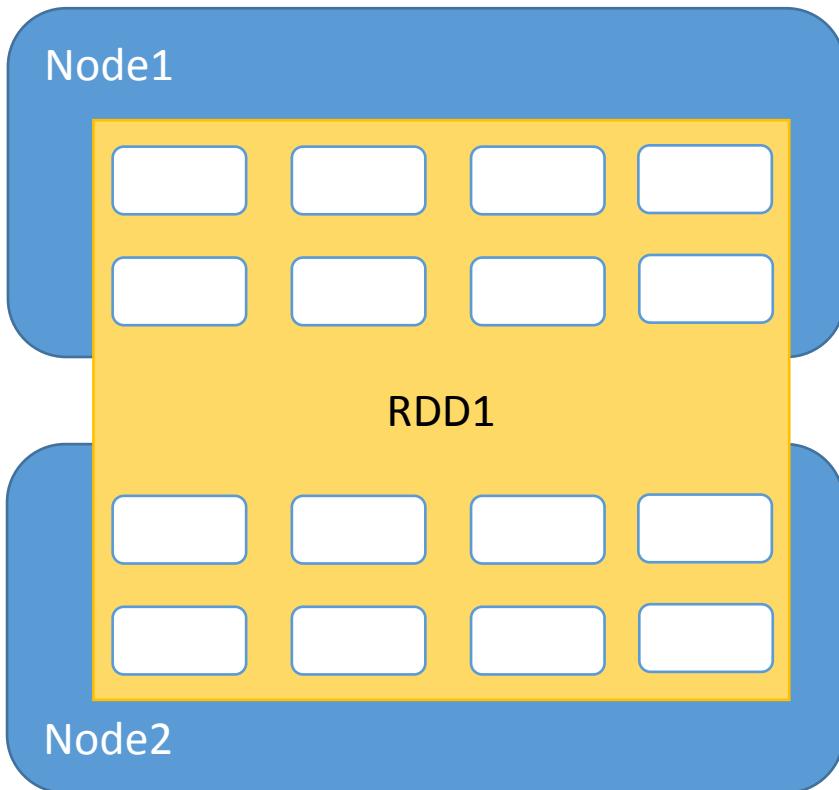
- `Join()`, `reduceByKey()`, `groupByKey()`,
- They cause a repartition

coalesce(n)



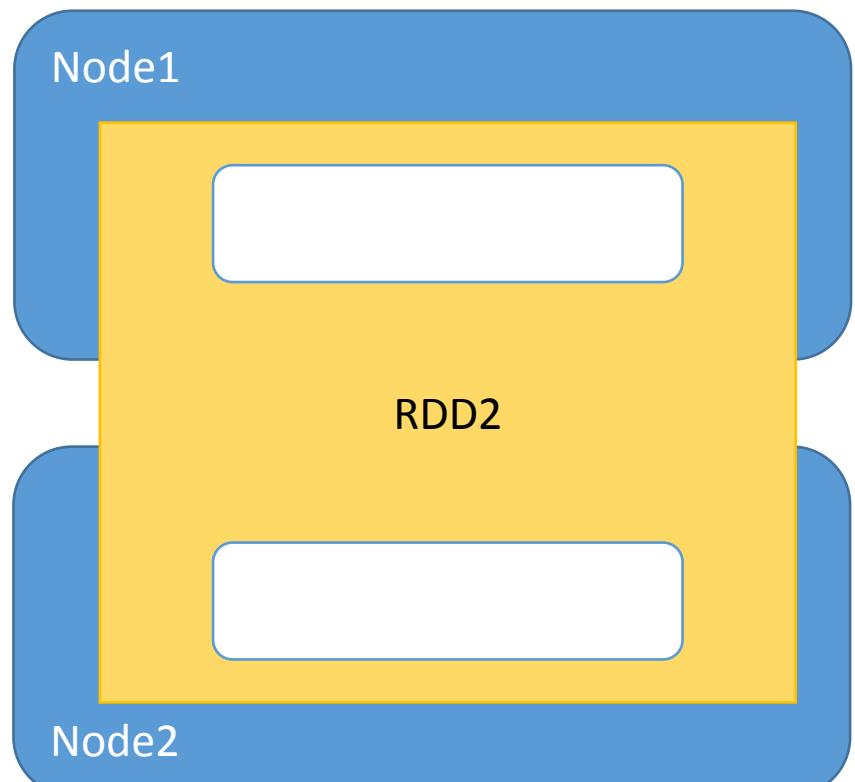
Return a new RDD with the data across 'n' partitions *

- It can only reduce the number of partitions
- N partitions are not ensured!
- No shuffle between nodes



$\text{RDD2} = \text{RDD1.coalesce(1)}$

RDD1 has 16 partitions
RDD2 has 2 partitions (not 1!)

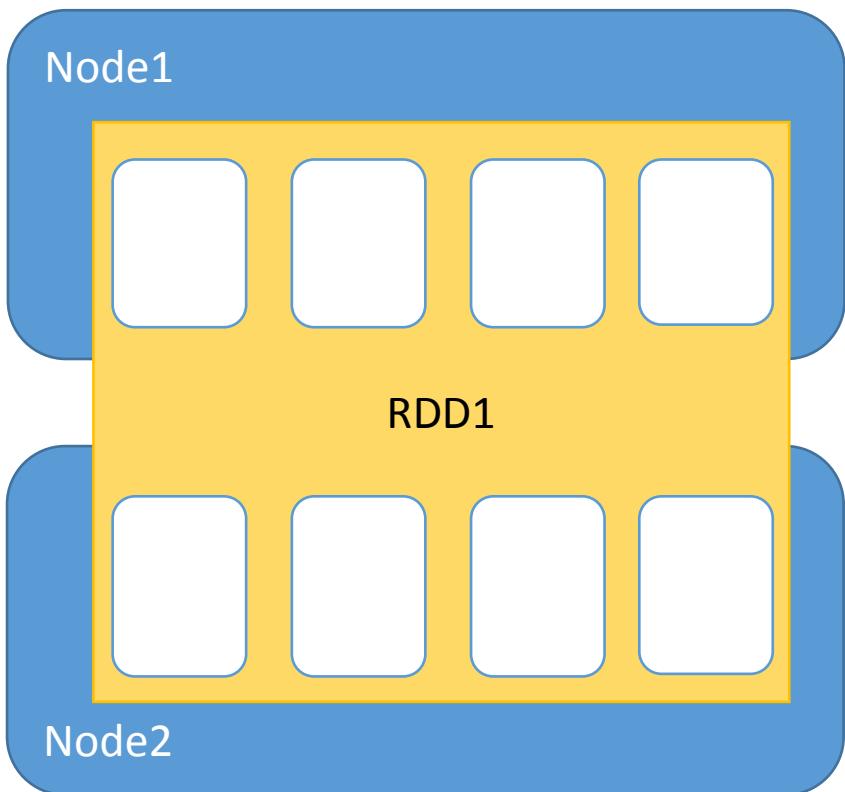


repartition(n)



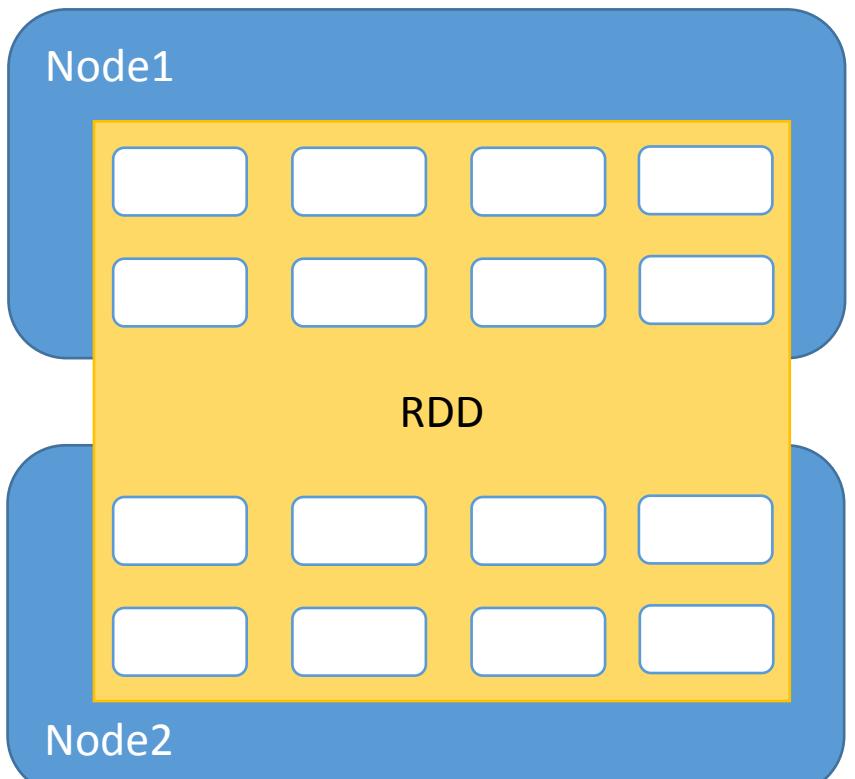
Return a new RDD with the data across ‘n’ partition

- It can reduce or increase the number of partitions
- Shuffle between nodes



`RDD2 = RDD1.repartition(16)`

RDD1 has 8 partitions
RDD2 has 16 partitions

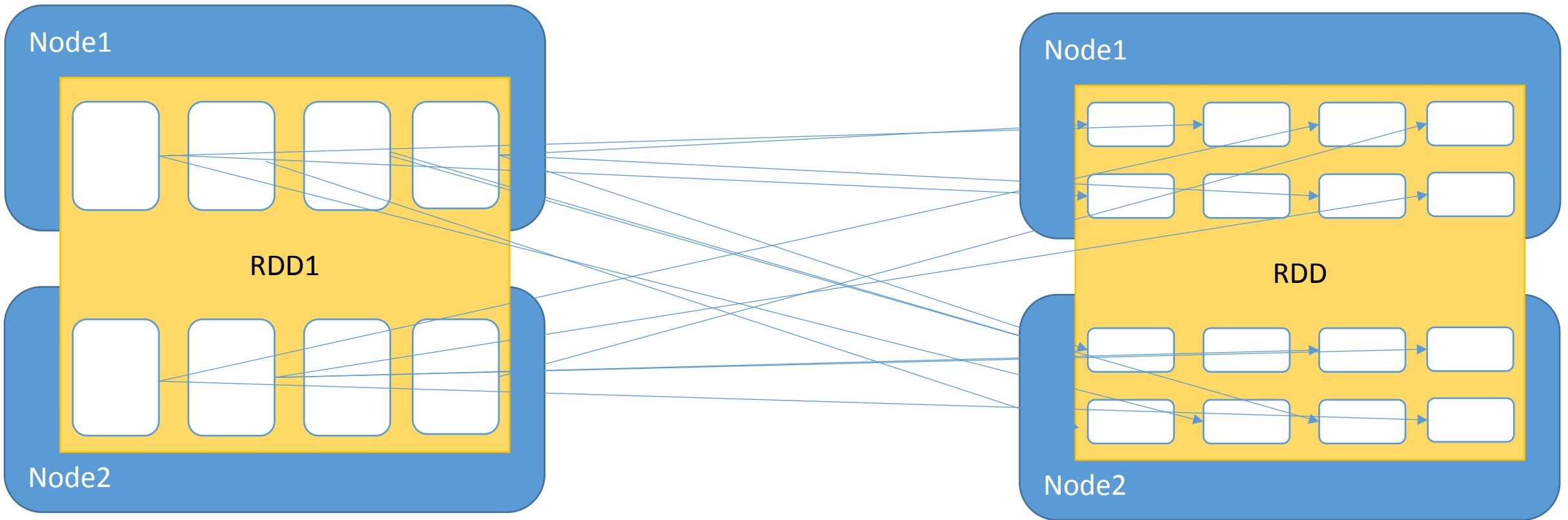


repartition(n)



Return a new RDD with the data across 'n' partition

- It can reduce or increase the number of partitions
- **Shuffle between nodes**





Partitioning in Spark

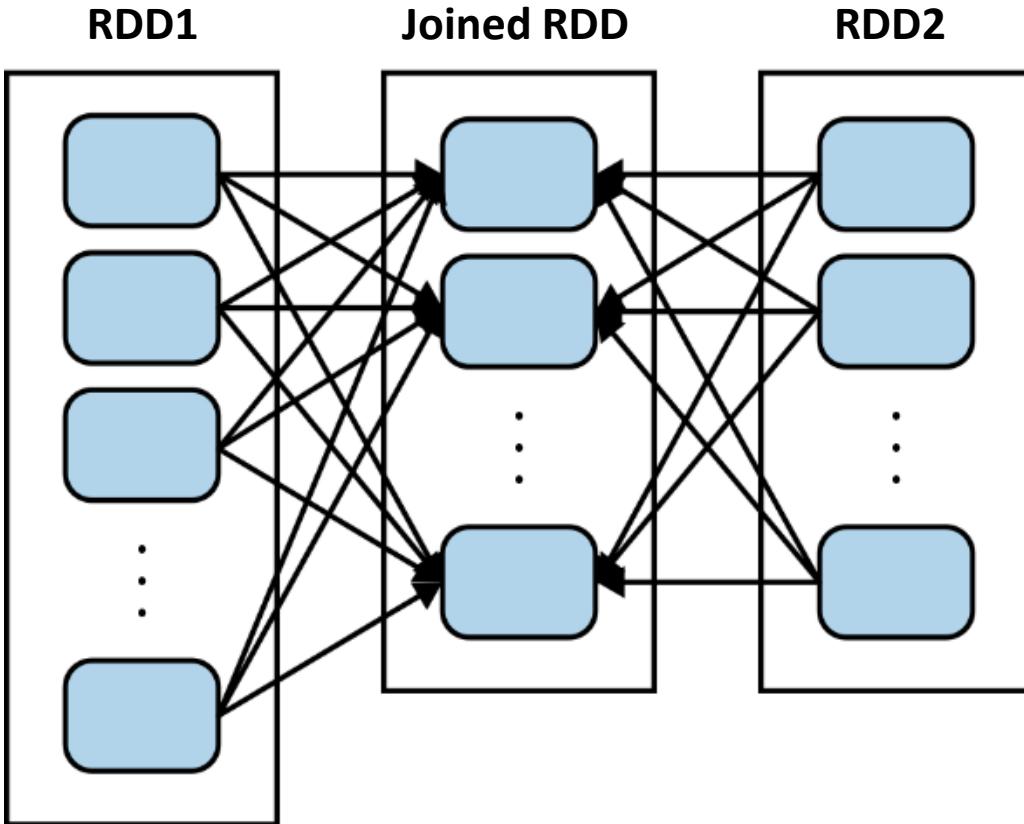
1. How it works
2. **Partitioning on Key-Value RDDs**
3. Optimize partitioning

Key-Value RDD partitioning



When a transformation requires reasoning on keys, same keys have to be on the same partition

Join example:



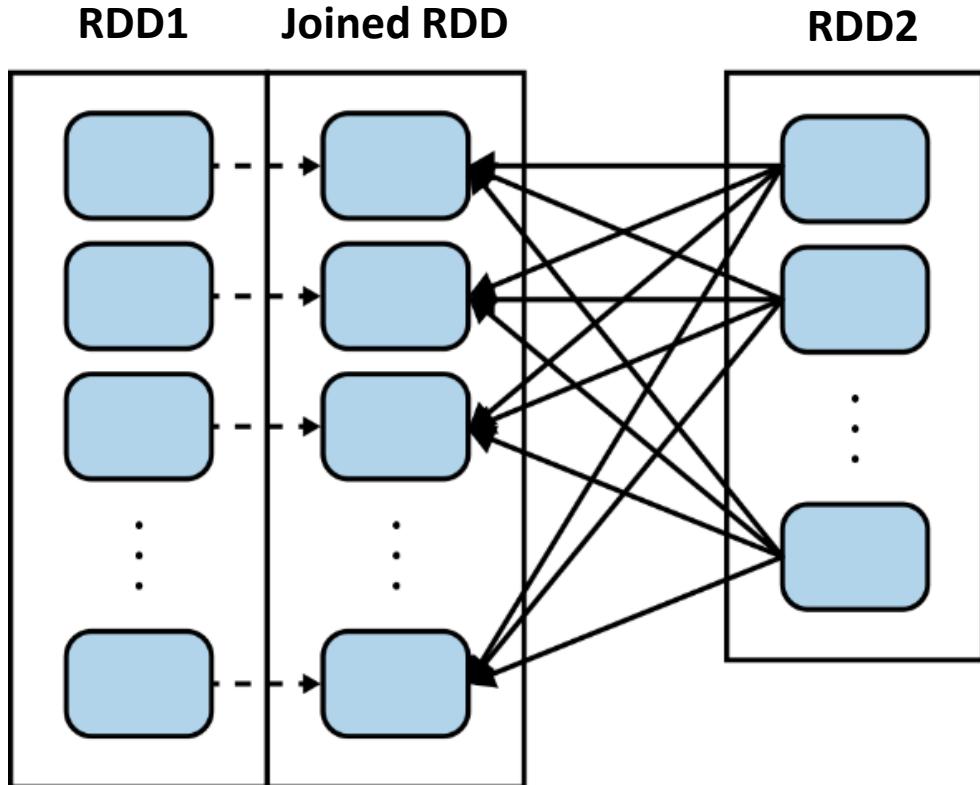
This requires a lot of shuffle!

Partitioners



- To distribute the records of a RDD according to the keys Spark use a **partitioner**
- If a partitioner is assigned to a PairRDD Sparks knows in which partitions its keys reside
- Available partitioners
 - HashPartitioner
 - RangePartitioner
 - Custom partitioners could be developed!
- A partitioner can be inherited by a RDD if result of some operations
 - Examples: mapValues(), join()

Effects of a partitioner



Only data from RDD2 is shuffled

Little detour: skew data



- Most of the time map-side transformations are not an option: **groupBy, join**
- **Pareto law** applies also for data! ‘80% of data related to 20% of possible keys’
 - Examples:
 - 80% of the transactions in a bank towards the 20% of the customers
 - 80% of traffic generated by 20% of the communication towers
- **Problem:** Hashing distributes key uniformly but not data!
 - Slow tasks
 - Data distribution unpredictable

Exploit data awareness



Custom partitioners to be implemented to exploit **data awareness**

Offline

- Ad hoc function to assign partition to keys -> **CustomPartitioner**

Online

- Data-aware Spark (by MTA SZTAKI and Ericsson)
- <https://youtu.be/aMh8KgaFWrY>
- <http://www.slideshare.net/SparkSummit/spark-summit-eu-talk-by-zoltan-zvara>

CustomPartitioner Example



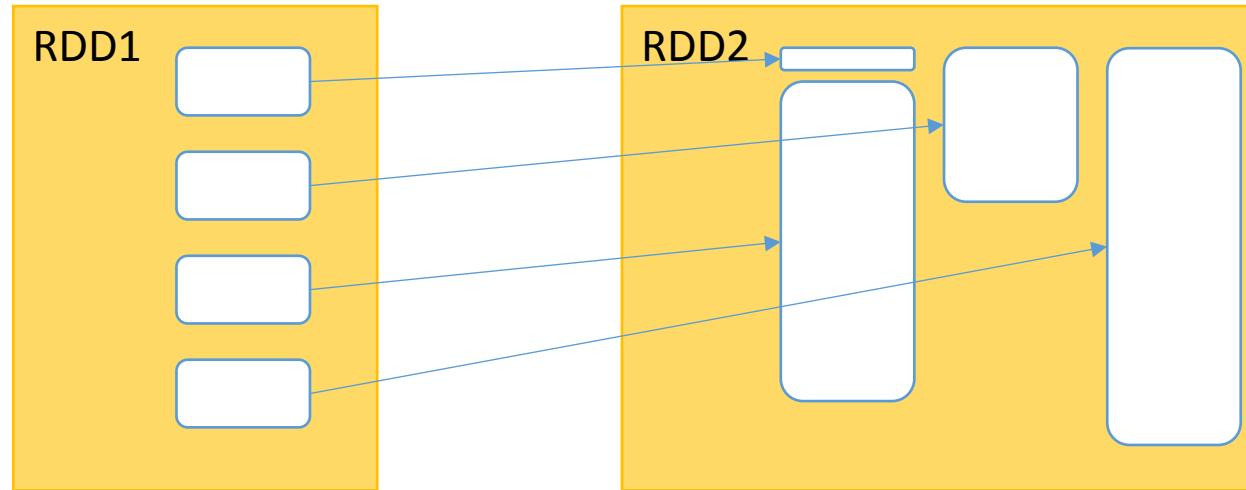
```
class MyCustomerPartitioner(numParts: Int) extends
org.apache.spark.Partitioner {
  override def numPartitions: Int = numParts
  override def getPartition(key: Any): Int =
  {
    val intKey = key.toString.toInt
    val specialKeys = Map(100 -> 0, 200 -> 1, 300 -> 2)
    specialKeys.getOrElse(intKey, intKey%(numPartitions-3) + 3 )
  }
  override def equals(other: Any): Boolean = other match
  {
    case dnp: MyCustomerPartitioner =>
      dnp.numPartitions == numPartitions
    case _ =>
      false
  }
}
```

Side effects on partitioning (1/2)



Partitions balance could be affected by transformations

- flatMap
 - Each partition from input RDD results in a partition on output RDD
 - Size of the partition on the outputRDD is unpredictable



- filter
 - Each partition from input RDD results in a partition on output RDD
 - Depending on the filter function the result partitions could be unbalanced

Side effects on partitioning (2/2)



Partitioner information can be lost also if the key doesn't change

map() vs mapValues()

flatMap() vs flatMapValues()

```
//Scala
val peopleByName = people
  .keyBy(p => p.name)
  .partitionBy(new HashPartitioner(100))

val peopleNameSurname = peopleByName.map{
  case (name, p) => name -> p.surname
}
```

```
//Scala
val peopleByName = people
  .keyBy(p => p.name)
  .partitionBy(new HashPartitioner(100))

val peopleNameSurname = peopleByName.mapValues{
  p => p.surname
}
```

From a functional point of view they are equal, but Spark behave differently!



Partitioning in Spark

1. How it works
2. Partitioning on Key-Value RDDs
3. **Optimize partitioning**

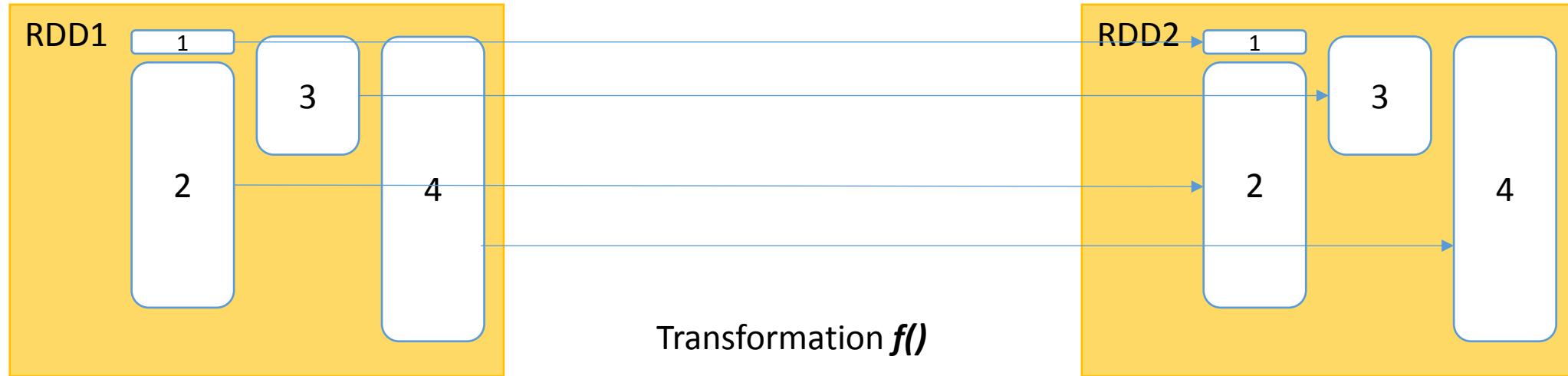
Ideal Partitioning



Ideal partitioning:

- Minimize idle time for the executors and the cores
- Reduces network overhead for shuffling

Minimize idle time



- Example case:

Assumption 4 cores running in parallel over 4 partitions

Time to execute each task T_i (with $i \in \{1,2,3,4\}$)

$T_i = n_i * K$ (with K = time to execute function $f()$ over a single record, n_i number of rows in partition i)

Time to complete the transformation $E = \text{Max}(T_i)$

To minimize the time E we have to balance the T_i , that results in balancing the RDD. **repartition()** needed

- Repartitioning is also time consuming, be careful with this optimization!

Reduce shuffling



Partitioners can be used to reduce shuffle in case of multiple joins

```
//Scala
val joinRDD1 = rdd1.join(rdd2)
val joinRDD2 = rdd1.join(rdd3)
```

In this case the rdd1 partitions are shuffled **2 times**, once for each join. Rdd2 and rdd3 partitions are shuffled too.

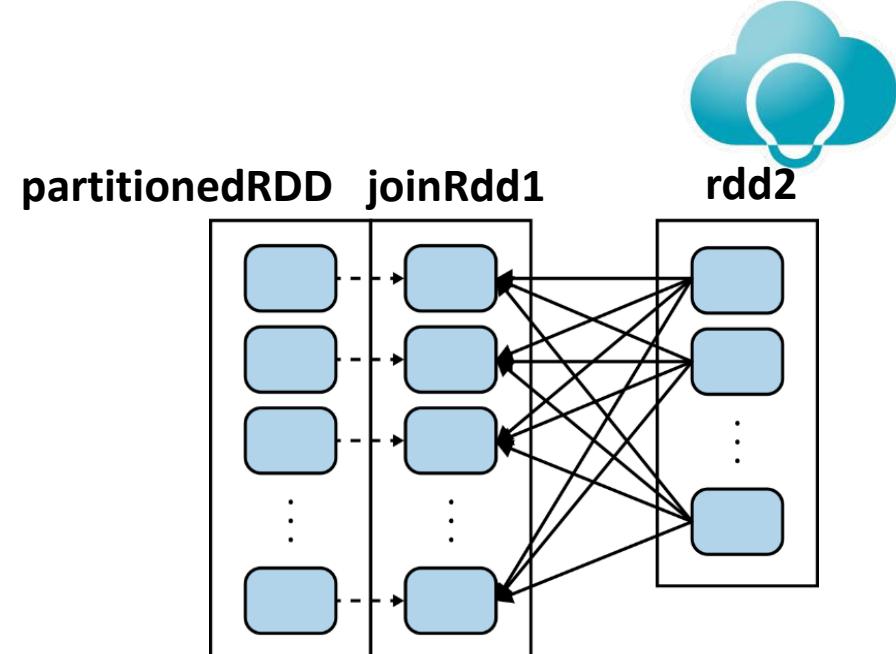
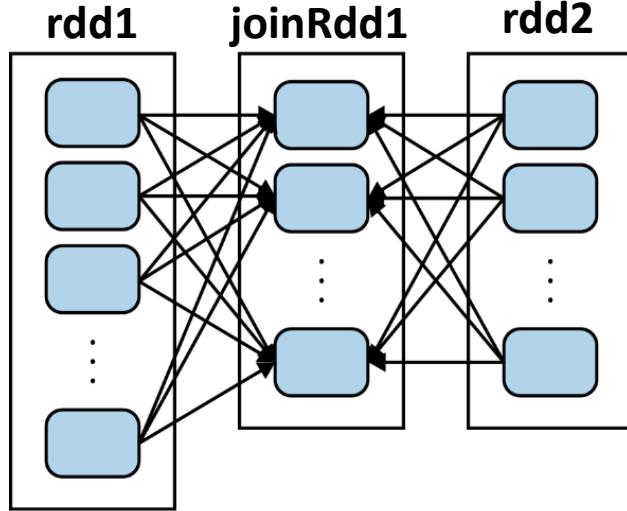
```
//Scala
val partitionedRDD = rdd1.partitionBy(new HashPartitioner(100)).cache()

val joinRDD1 = partitionedRDD.join(rdd2)
val joinRDD2 = partitionedRDD.join(rdd3)
```

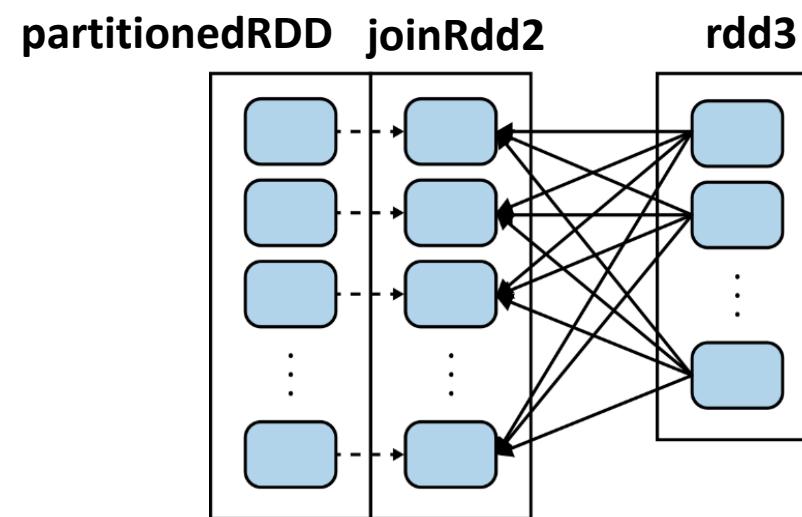
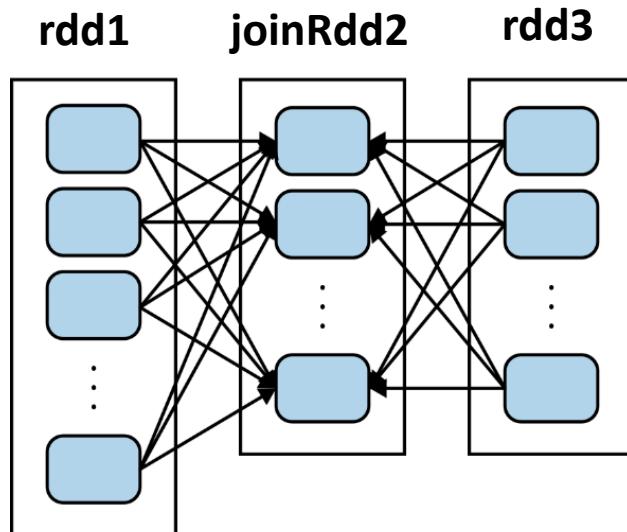
In this case the rdd1 partitions are shuffled **1 time**, once when the partitioner is set.

Rdd2 and rdd3 partitions are shuffled.

Reduce shuffling



Vs.



Understanding of transformations



- A deep understanding of transformation behavior can help you in improve performances
- Data shuffle across partition can be avoided with the right transformations

Classic example:
`groupByKey()` vs `reduceByKey()`

groupByKey() vs reduceByKey()



```
//Scala
val people = input

val avgAgeByGender = people.keyBy(p => p.gender)
  .groupByKey()
  .map{
    case (gender, peopleList) =>
      val ages = peopleList.map(_.age)
      gender -> (ages.sum / ages.size)
  }
avgAgeByGender.collect().foreach(println)
```

```
//Scala
val people = input

val avgAgeByGender =
  people.keyBy(p => p.gender)
  .map{
    case (gender, person) =>
      gender -> (person.age, 1)
  }
  .reduceByKey{
    (rec1: (Int, Int), rec2: (Int, Int)) =>
    (rec1._1 + rec2._1, rec1._2 + rec2._2)
  }.mapValues(pair => pair._1 / pair._2)

avgAgeByGender.collect().foreach(println)
}
```

Demo

Spark Training

Powered by



Spark SQL - Roadmap



1. RDD vs DataFrame vs DataSet
2. Working with DataFrames
3. Sparkling queries with Spark SQL
4. Saving and loading DataFrames
5. Optimization

Spark SQL - Roadmap



1. RDD vs DataFrame vs DataSet
2. Working with DataFrames
3. Sparkling queries with Spark SQL
4. Saving and loading DataFrames
5. Optimization

RDD vs DataFrame vs DataSet



- **RDD:**

Immutable **distributed** collection of elements of your data, **partitioned** across nodes in your cluster that can be operated in parallel with a **low-level API**, i.e. transformations and actions.

- **DataFrame:**

Immutable distributed collection of data, **organized** into named columns. It is like table in a relational database.

DataFrame: pros and cons

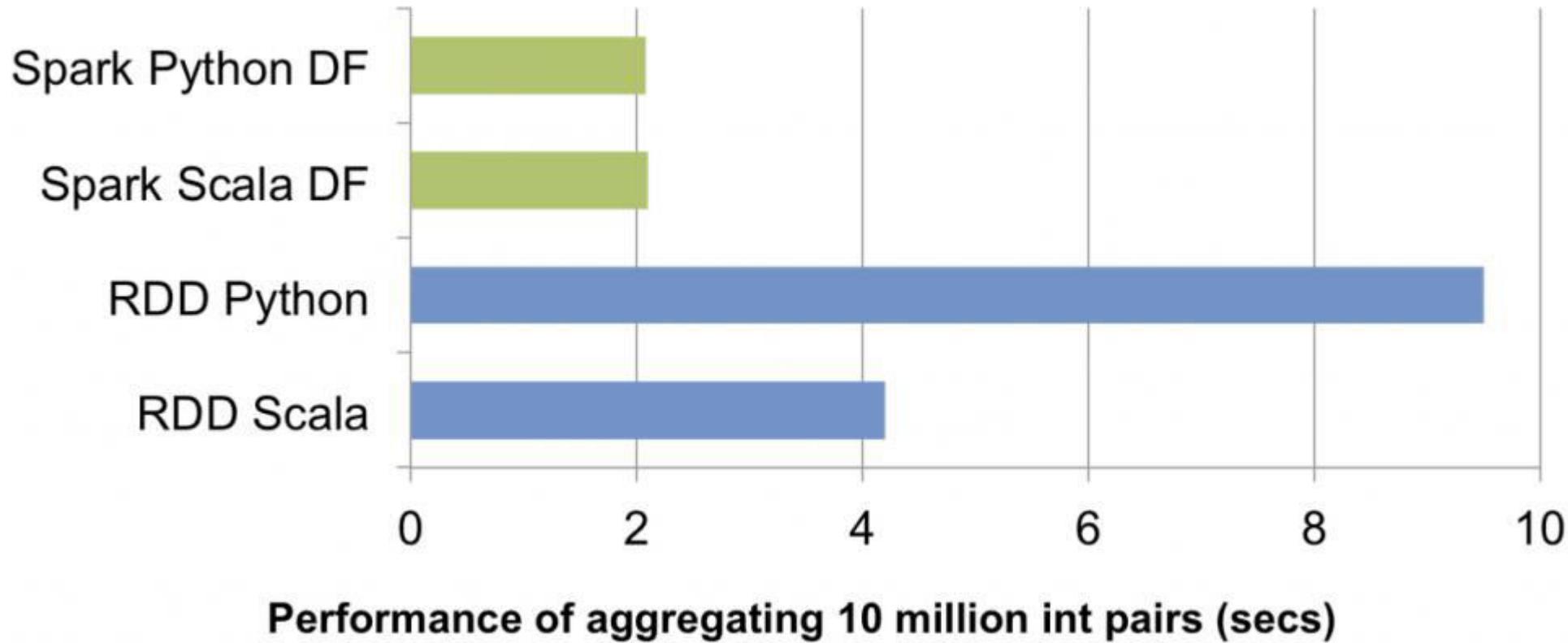


- Higher level API, which makes spark available to wider audience
- Performance gains thanks to the **Catalyst** query optimizer
- Space efficiency by leveraging the **Tungsten** subsystem

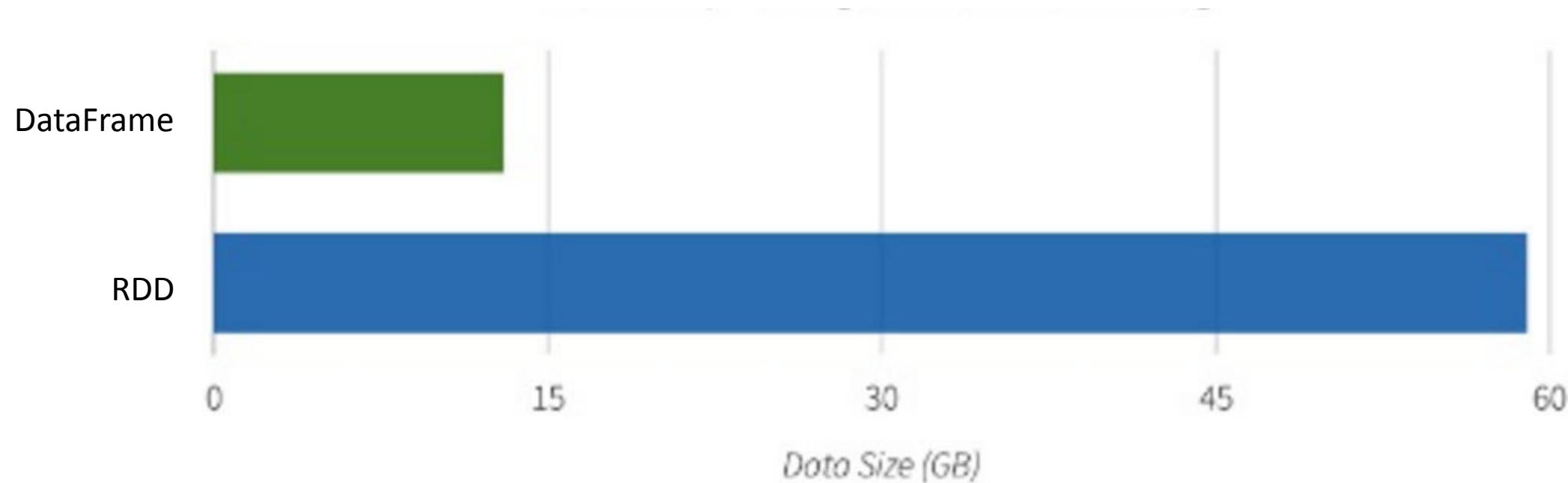


- Higher level API may limit expressiveness
- Complex transformation are better express using RDD's API

DataFrame are fast



DataFrames are memory efficient



Ditch RDDs? Not really

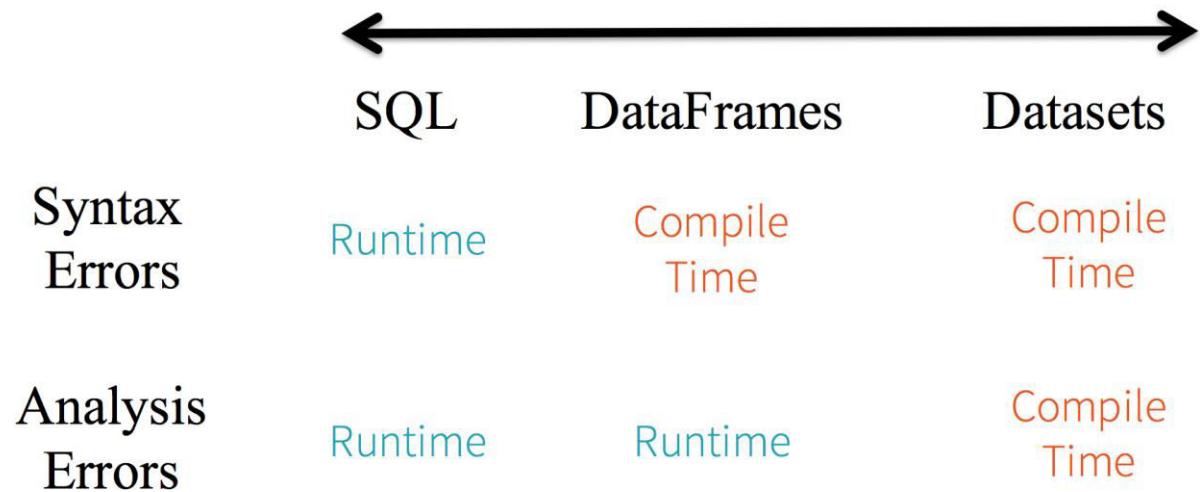


- You want low-level transformations (fp + DSL) and actions and control on your dataset
- Your data is unstructured, such as media streams or streams of text
- You don't care about imposing a schema, and you can forgo some optimization and performance benefits available with DataFrames

Datasets: the future (2.0+)



- Marked as experimental in Spark 1.6.x, stable from 2.0 +
- DataFrame api will be merged in the DataSet api
- Provides static typing and runtime type-safety



Spark SQL - Roadmap



1. RDD vs DataFrame vs DataSet
2. Working with DataFrames
3. Sparkling queries with Spark SQL
4. Saving and loading DataFrames
5. Optimization

Example



2009 StackOverflow data dump (Italian questions)

- *commentCount*—number of comments related to the question/answer
- *lastActivityDate*—date and time of the last modification
- *ownerUserId*—user ID of the owner
- *body*—textual contents of the question/answer
- *score*—total score based on upvotes and downvotes
- *creationDate*—date and time of creation
- *viewCount*—the view count
- *title*—the title of the question
- *tags*—a set of tags the question has been marked with
- *answerCount*—number of related answers
- *acceptedAnswerId*—if a question, contains the ID of its accepted answer
- *postTypeId*—type of the post; 1 is for questions, 2 for answers
- *id*—post's unique ID



How to create a DataFrame



Prerequisites:

```
import org.apache.spark.sql.SQLContext  
val sqlContext = new SQLContext(sc)  
import sqlContext.implicits._
```

- If the hive support is available you should use **HiveContext** instead of **SQLContext**
- This step is optional if you're using spark shell

How to create a DataFrame



From an existing RDD we can obtain:

1. A row (tuple) DataFrame
2. A DataFrame based on a case class
3. A DataFrame with a schema manually specified

How to create a DataFrame (1)



```
val itPostsRows = sc.textFile("italianPosts.csv")
val itPostsSplit = itPostsRows.map(x => x.split("~"))
```

- We've just created an RDD[Array[Strings]]
- The RDD class has a **df** method which returns the corresponding DataFrame[Row]
- It only works with ***string, int, long, tuple*** data types and case classes

```
val itPostsRDD = itPostsSplit.map(x => (x(0),x(1),x(2),x(3),x(4),
                                         x(5),x(6),x(7),x(8),x(9),x(10),x(11),x(12)))
val itPostsDFrame: DataFrame[Row] = itPostsRDD.toDF()
```

DataFrame[Row]: pros and cons



- Super easy to create



- No column names
- Only index based access to fields

We'll see how to overcome these shortcomings

How to create a DataFrame (2)



- Let's create a case class that reflects our dataset's structure
- Nullable fields need to be declared as Option[T]
- For timestamp columns, Spark supports java.sql.Timestamp class

```
case class Post( commentCount: Option[Int],  
                 lastActivityDate: Option[Timestamp],  
                 ownerUserId: Option[Long],  
                 body: String,  
                 score: Option[Int],  
                 creationDate: Option[Timestamp],  
                 viewCount: Option[Int],  
                 title: String,  
                 tags: String,  
                 answerCount: Option[Int],  
                 acceptedAnswerId: Option[Long],  
                 postTypeId: Option[Long],  
                 id: Long)
```

- We then map the raw string to the Post case class and we invoke the .df method

How to create a DataFrame (3)



We can use *SQLContext*'s *createDataFrame* method, which requires:

- An RDD of Row objects
- A *StructType* containing one or more *StructFields*, one for each column

```
import org.apache.spark.sql.types._  
val postSchema = StructType(Seq(  
    StructField("commentCount", IntegerType, nullable = true),  
    StructField("lastActivityDate", TimestampType, nullable = true),  
    StructField("ownerUserId", LongType, nullable = true),  
    StructField("body", StringType, nullable = true),  
    StructField("score", IntegerType, nullable = true),  
    StructField("creationDate", TimestampType, nullable = true),  
    StructField("viewCount", IntegerType, nullable = true),  
    StructField("title", StringType, nullable = true),  
    StructField("tags", StringType, nullable = true),  
    StructField("answerCount", IntegerType, nullable = true),  
    StructField("acceptedAnswerId", LongType, nullable = true),  
    StructField("postTypeId", LongType, nullable = true),  
    StructField("id", LongType, nullable = false))  
)
```

How to inspect a DataFrame



We take a peek into our freshly created DataFrame by using:

- *show(n)*: prints out the first n lines of the DataFrame, along with column names
- *printSchema*: prints out the names and the data type of each column of the Dataframe
- *columns*: returns an Array of column names
- *dtypes*: return an Array of tuple with column names and corresponding type names

Demo

DataFrame API: selection



Selecting data using the select method:

```
/*selection using column names*/
val postsIdBody = postsDf.select("id", "body")
/*selection using column objects*/
val postsIdBody = postsDf.select(postsDf.col("id"), postsDf.col("body"))
/*select all columns BUT the ones specified*/
val allButBody = postsDf.drop("body")
```

Using the implicit conversions (*sqlContext.implicits._*) we can use:

- The implicit conversion between Scala's symbols and *Column* object
- The dollar (\$) operator which converts *Strings* to *Column* objects

DataFrame API: selection



```
/*selection through scala's symbols*/  
val postsIdBody = postsDf.select(Symbol("id"), Symbol("body"))  
  
val postsIdBody = postsDf.select('id, 'body)  
/*selection using the $ operator*/  
val postsIdBody = postsDf.select($"id", $"body")
```

Scala symbols are useful because:

- At most one instance per symbol exists
- They can easily be checked for equality

DataFrame API: filtering



The *where* and *filter* method are equivalent for filtering data on a DataFrame

```
/*filtering through column expression*/
postsDf.filter('body contains "Italiano").count
/*filtering through column object*/
val noAnswer = postsDf.filter((postTypeId === 1) and ('acceptedAnswerIdisNull))
/*limiting the resulting DF*/
val firstTenQs = postsDf.filter('postTypeId === 1).limit(10)
```

- Column objects contains a rich set of SQL-like operator beside representing a Column name
- Column expression can be also used inside a select operator

DataFrame API: adding columns



withColumn method adds a column according to a given Column expression:

```
postsDf.filter('postId === 1).  
  withColumn("ratio", 'viewCount / 'score). //new DF  
  where('ratio < 35).show()  
  
/*Column renaming*/  
firstTenQs.withColumnRenamed("ownerUserId", "owner")
```

Since DataFrames are immutable both methods will return a **new** DataFrame

DataFrame API: sorting data



The *sort* and *orderBy* methods are equivalent for sorting data on a DataFrame

```
postsDf.filter('postTypeId === 1)
    .orderBy('lastActivityDate desc)
    .limit(10)
    .show
```

DataFrame API: SQL functions



Like many DBMS, Spark SQL supports a variety of built-in functions:

- Scalar functions:
a single value for each row computed on one or more columns.
- Aggregate functions:
a single value for a group of rows.
- User-defined functions (UDF):
custom scalar or aggregate functions.

```
import org.apache.spark.sql.functions._
```

DataFrame API: SQL functions



Scalar functions examples:

- Math functions: *abs, log, cbrt, exp*
- String operations: *concat, length, trim*
- Date-time operators: *year, date_add*

```
postsDf
    .filter('postTypeId === 1)
    .withColumn("activePeriod",
    datediff('lastActivityDate, 'creationDate))
    .orderBy('activePeriod desc)
    .head.getString(3)
```

Aggregate functions are used in combination with *groupBy* but can also be used on the whole dataset, inside *select* or *withColumn* statements

```
postsDf.select(avg('score), max('score), count('score)).show
```

DataFrame API: UDF functions



We can extend the collection of built-in function with the ones that suit our needs:

- A UDF takes as input zero or more columns (up to 10) and returns a single value
- It can be created either
 - Using the *udf* function (from `org.apache.spark.sql.functions`)
 - Using *SQLContext.udf.register* function

```
val countTags = udf(  
  (tags: String) => "<".r.findAllMatchIn(tags).length)  
val countTags2 = sqlContext.udf.register("countTags",  
  (tags: String) => "<".r.findAllMatchIn(tags).length)  
  
postsDf.filter('postTypeId === 1).  
  select('tags, countTags('tags) as "tagCnt").show(10, false)
```

DataFrame API: dirty data



Data is rarely ready to use: it's often filled with null values, empty fields, *NaNs* (not a number). DataFrameNaFunctions class might prove useful:

- You can drop rows with nulls or *NaNs*: the `DataFrame.na.drop()` function is quite flexible

```
//drop rows with at least one null or Nan column
postsDf.na.drop()

//drop rows with nulls/Nan in all the columns
postsDf.na.drop("all")

//drop rows with nulls/Nan in specific column list
postsDf.na.drop(Array("acceptedAnswerId"))
```

DataFrame API: dirty data



- You can choose to fill null values with a *String* or a numeric (*Double*) constant

```
//Fill all null values in String columns
postsDf.na.fill("null") //or .fill(0.0)
//fill null values of specific columns
postsDf.na.fill(0.0, Array("viewCount"))
//column specific fill strategy
postsDf.na.fill(Map(
    "acceptedAnswerId" -> "unknown",
    "viewCount" -> 1.0
))
```

- You can replace values according to a remapping table

```
postsDf.na.
    replace(Array("id", "acceptedAnswerId"), Map(1177 -> 3000))
```

DataFrame API: grouping data



Grouping data with DataFrame is performed by the `groupBy` method. It accepts a list of column names or column objects and returns a `GroupedData` object.

`GroupedData` represents groups of rows that have the same values in the columns specified when calling `groupBy`

```
postsDf.groupBy('ownerUserId, 'tags,  
    'postTypeId) //groupedData  
    .count  
    .orderBy('ownerUserId desc)  
    .show(10)
```

DataFrame API: grouping data



You can perform several aggregations on different columns with the *agg* function. It can take one or more aggregate functions that work on column expressions. Alternatively you can use a map to indicate columns and their respective aggregate function name.

```
postsDf.groupBy('ownerUserId')
      .agg(max('lastActivityDate'), max('score')).show(10)
postsDf.groupBy('ownerUserId')
      .agg(Map("lastActivityDate" -> "max", "score" -> "max")).show(10)
```

DataFrame API: join



The join method of the DataFrame supports all join flavors of standard SQL join

```
postsVotes = postsDf.join(votesDf,  
    postsDf("id") === 'postId')
```

Demo

Spark SQL - Introduction



The DataFrame DSL functionalities presented in the previous sections are also accessible through SQL commands. When you write SQL commands in Spark SQL, they get translated into operations on DataFrames.

Spark actually supports two SQL dialects:

- Spark's SQL dialect
- Hive Query Language (HQL).

The Spark community recommends HQL (Spark version 1.5) because it has a richer set of functionalities.

To use Hive functionalities, you need a Spark distribution built with Hive support

Spark SQL – Introduction (2)



Most SQL operations involve referencing a table by its name. Using SparkSQL you can reference a DataFrame by name registering it as a table



- SQLContext stores the table catalog as an in-memory map
- HiveContext uses Hive metastore and thus the DataFrame definitions are **not** lost after the spark program ends
 - You can also choose to store temporary tables in the metastore

```
postsDf.registerTempTable("posts_temp")
```

Spark SQL – Introduction (2)



Now that we've registered our table we can perform SQL queries using the `sqlContext.sql` method

```
val resultDF = sqlContext.sql("select * from posts_temp")
```

The result is again a DataFrame. All of the data manipulations performed with DataFrame DSL can also be done through SQL.

Spark also offers a SQL shell in the form of `spark-sql` command. When run without arguments, it starts a SQL shell in local mode.

Spark SQL – Hive integration



In order to successfully connect to the Hive metastore the Spark program needs access to the Hive configuration directory

- `/etc/hive/conf/hive-env.sh` sets hive environment variables
- `/etc/hadoop/conf/hadoop-env.sh` sets Hadoop environment variables

Alternatively just copy `hive-site.xml` into the Spark conf directory

- If you see the `metastore_db` directory in your working path it means that the connection to the hive metastore was not successful and a local metastore is being used instead.



Spark SQL – Impala



Impala is a MPP query engine that can execute SQL queries very fast even on large datasets. It can access Hive's metastore and read Hive's tables. If you (permanently) register a DataFrame you can use Impala for data exploration.

- You may need to execute a refresh on your newly registered table in order for impala to actually see its data.
- You might also need to issue an invalidate metadata before the refresh command for the data to be accessible for impala

Impala uses private indexes to speed up the query computation. Those indexes need to be created and updated, should the data change.

Demo

Spark SQL - Roadmap



1. RDD vs DataFrame vs DataSet
2. Working with DataFrames
3. Sparkling queries with Spark SQL
4. Saving and loading DataFrames
5. Optimization

Spark SQL – Load and save data

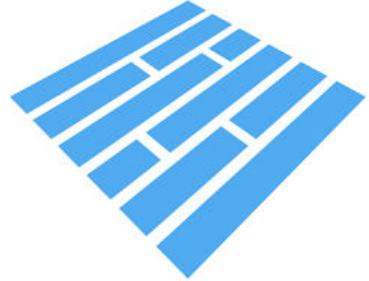


Spark has built-in support for several file formats and database backends (data sources).

- Hive, MySql, Postgres, JDBC
- Json, Orc (**Optimized Row Columnar**), Parquet

Data sources are pluggable, you can add the one that suits your need:

- SparkCSV adds native support for CSV files and hence is very useful
- SparkAvro adds support for avro files



Parquet file format



Parquet is a columnar storage format that can efficiently store nested data structures.

- Column values are stored next to each other which allows efficient encoding and less space requirements
- The data layout allows to read only portion of the whole file thus speeding up the query execution time
- The use of compression algorithms allows even more space saving
- Furthermore, low level optimizations makes this format very suitable for big data workloads

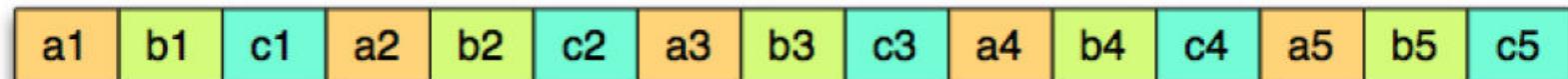
Parquet file format



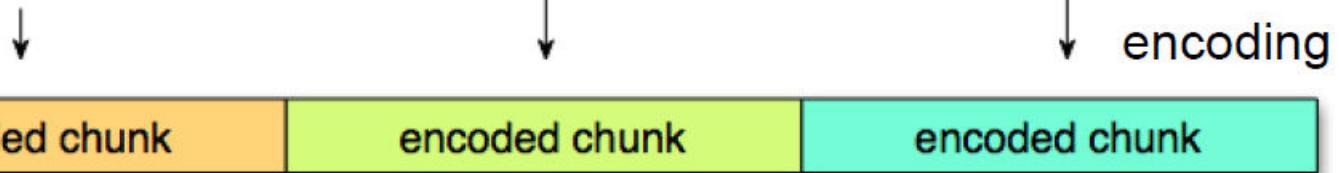
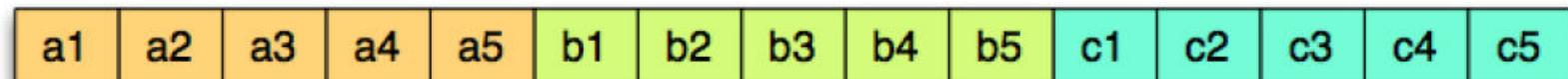
Logical table representation

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

Row layout



Column layout



Homogeneous data is stored next to each other

- Encodings: delta encoding, dictionary encoding, run-length encoding
- Compression algorithm can achieve better performance

Parquet file format



Vertical partitioning
(projection push down) + Horizontal partitioning
(predicate push down) = Read only the data
you need!

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

+

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

=

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

- Column layout allows to quickly skip unnecessary column data
- Per block stats (min-max) for each column allows to read only relevant blocks of data

Spark SQL – Save data



DataFrame's data is saved using the `DataFrameWriter` object available as DataFrame's `write` field. There are three different methods to carry out the DataFrame writing to disk:

- `save`
- `saveAsTable`
- `insertInto`

The latter two methods work with Hive and therefore need an `HiveContext`

Spark SQL – Save data



DataFrameWriter needs some configuration tweaking before being used:

- **format**: specifies the file format for saving the data: parquet, Orc, json ...
- **mode**: specifies the behavior in case of existing file
 - **overwrite**: overwrites existing data
 - **append**: appends the data
 - **ignore**: does nothing
 - **error**: throws an Exception (default behavior)
- **option** and **options**: adds a single or multiple configuration parameters to the data source configuration (see documentation for details)

Spark SQL – Save data (2)



- **partitionBy**: specifies the partitioning column(s). Parquet only

You can chain up these methods one after another to build a DataFrameWriter .

```
postsDf
    .write
    .format("parquet")
    .mode("overwrite")
    .option("myConfigOpt", "myConfigOptValue")
    .save("postdf.parquet") //action
```

Spark SQL – Save as Table



`saveAsTable` takes as a parameter only the table name. If such table exists the mode configuration determines the resulting behavior.

Hive SerDe (serialization/deserialization class), takes care of the serialization.

Table data location depend on hive's configuration.

```
postsDf  
    .write  
    .format("parquet")  
    .saveAsTable("myPostDfTable")
```

Spark SQL – Save as Table (2)



`insertTo` works with an existing table which must have the very same schema as our DataFrame. If you set the `mode` to `overwrite` the content of the table will be replaced with the data from source DataFrame

```
postsDf
    .write
    .mode(SaveMode.Overwrite) //be careful!
    .insertInto("myExistingTable")
```

Spark SQL – Load data



Data loading is accomplished by using the `DataFrameReader` object, accessible through the `read` field of `sqlContext`. It works similarly to `DataFrameWriter`.

- Reading from existing table:

```
sqlContext.read.table("posts")
sqlContext.table("posts")
```

- Reading from source files:

```
sqlContext.read.format("parquet").load("pathToMyFile")
sqlContext.read.parquet("pathToMyFile")
```

If the input source is partitioned spark will automatically add partitioning field to the DataFrame schema

Spark SQL - Roadmap



1. RDD vs DataFrame vs DataSet
2. Working with DataFrames
3. Sparkling queries with Spark SQL
4. Saving and loading DataFrames
5. Optimization

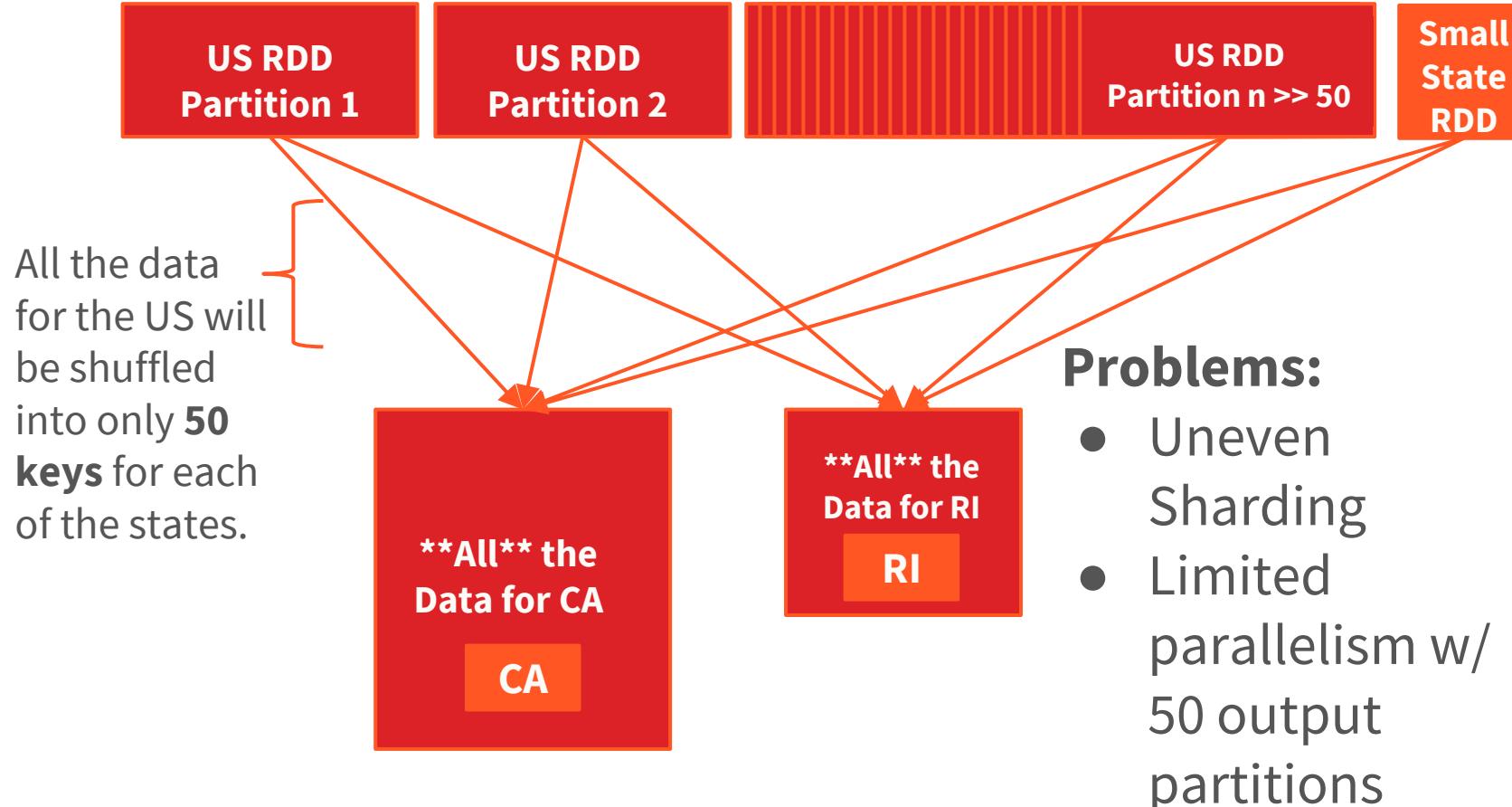
Joining a large table with a small one



```
join_rdd = sqlContext.sql("select *\n    FROM people_in_the_us\n    JOIN states\n    ON people_in_the_us.state = states.name")
```

- **ShuffledHashJoin?**
- **BroadcastHashJoin?**

Shuffled Hash Join

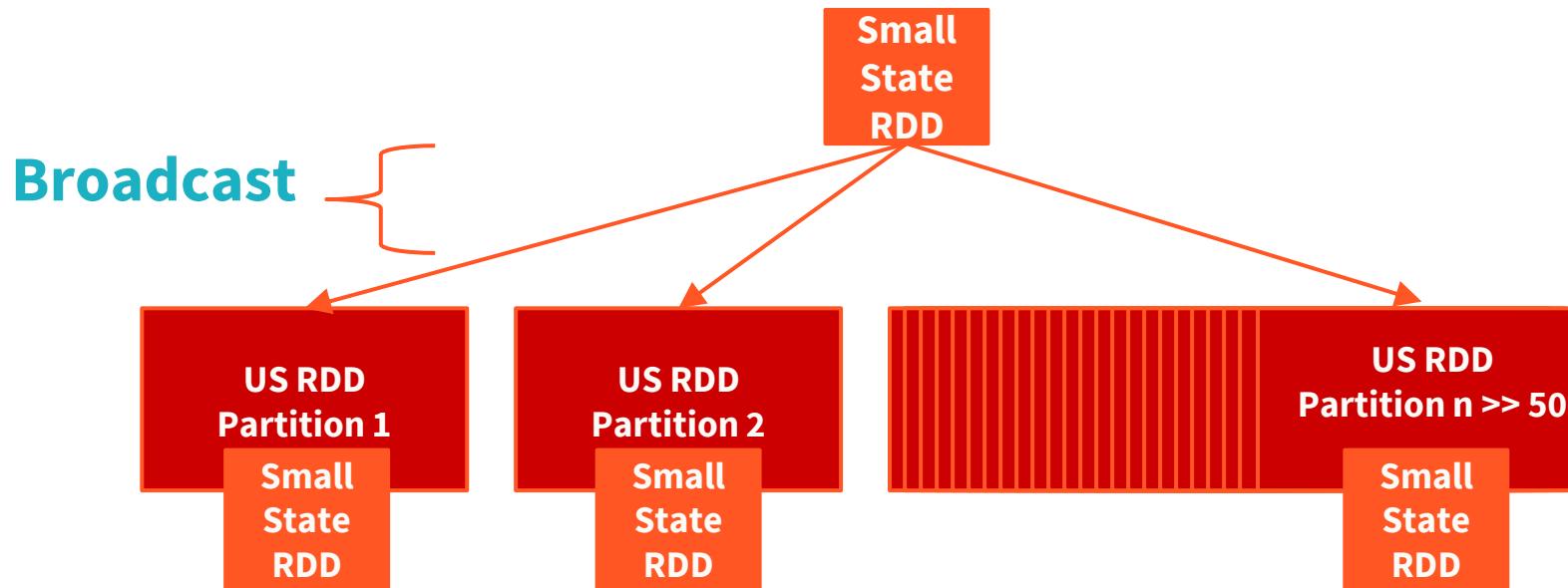


Even a larger Spark cluster will not solve these problems!

Broadcast Hash Join



Solution: Broadcast the Small RDD to all worker nodes.



Parallelism of the large RDD is maintained (n output partitions),
and shuffle is not even needed.

Broadcast Hash Join



- ❑ See the Spark SQL programming guide for your Spark version for how to configure.
- ❑ For Spark 1.3:
 - ❑ Set `spark.sql.autoBroadcastJoinThreshold`.
 - ❑ `sqlContext.sql("ANALYZE TABLE state_info COMPUTE STATISTICS noscan")`
 - ❑ Use `.toDebugString()` or `EXPLAIN` to double check.

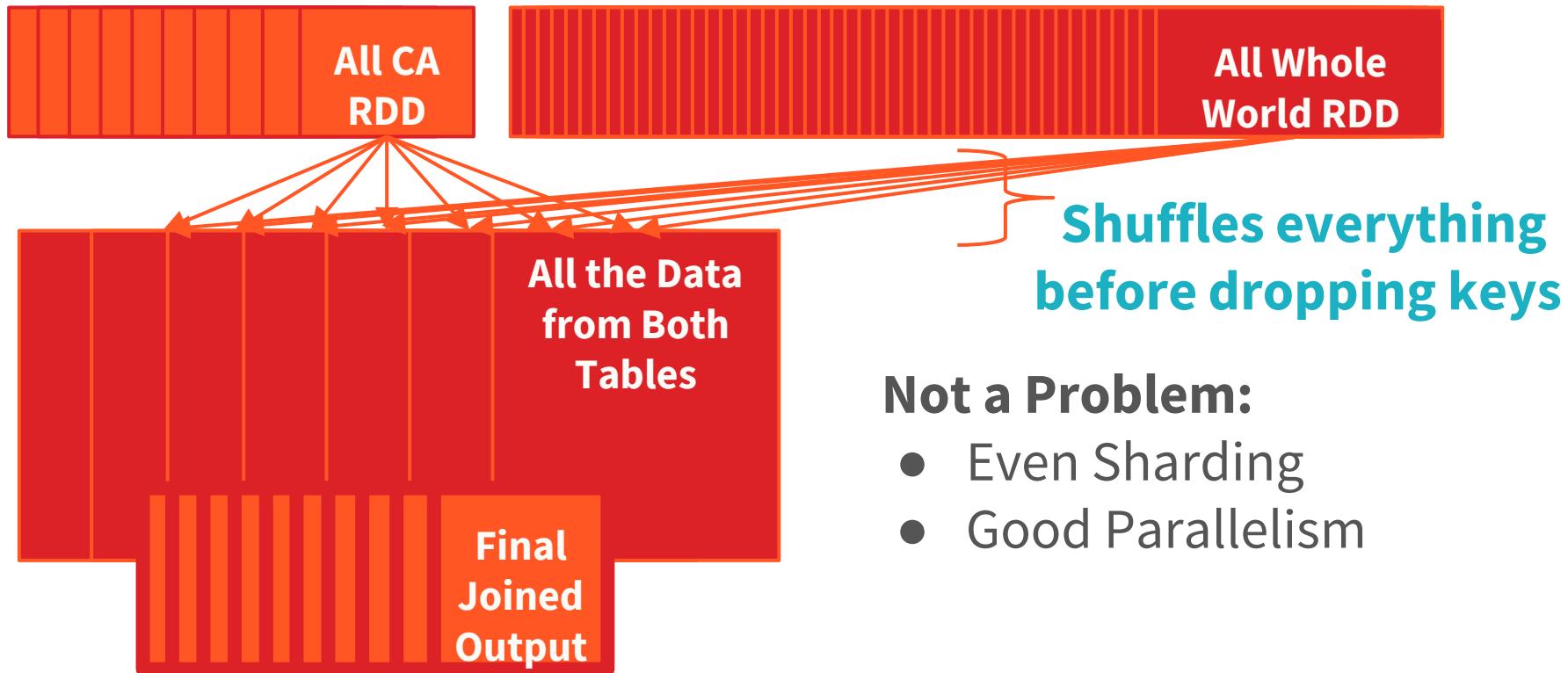
Joining a medium table with a huge one



```
join_rdd = sqlContext.sql("select *\n    FROM people_in_california\n    LEFT JOIN all_the_people_in_the_world\n    ON people_in_california.id =\n        all_the_people_in_the_world.id")
```

**Final output keys = keys people_in_california,
so this don't need a huge Spark cluster, right?**

Left join – shuffle step

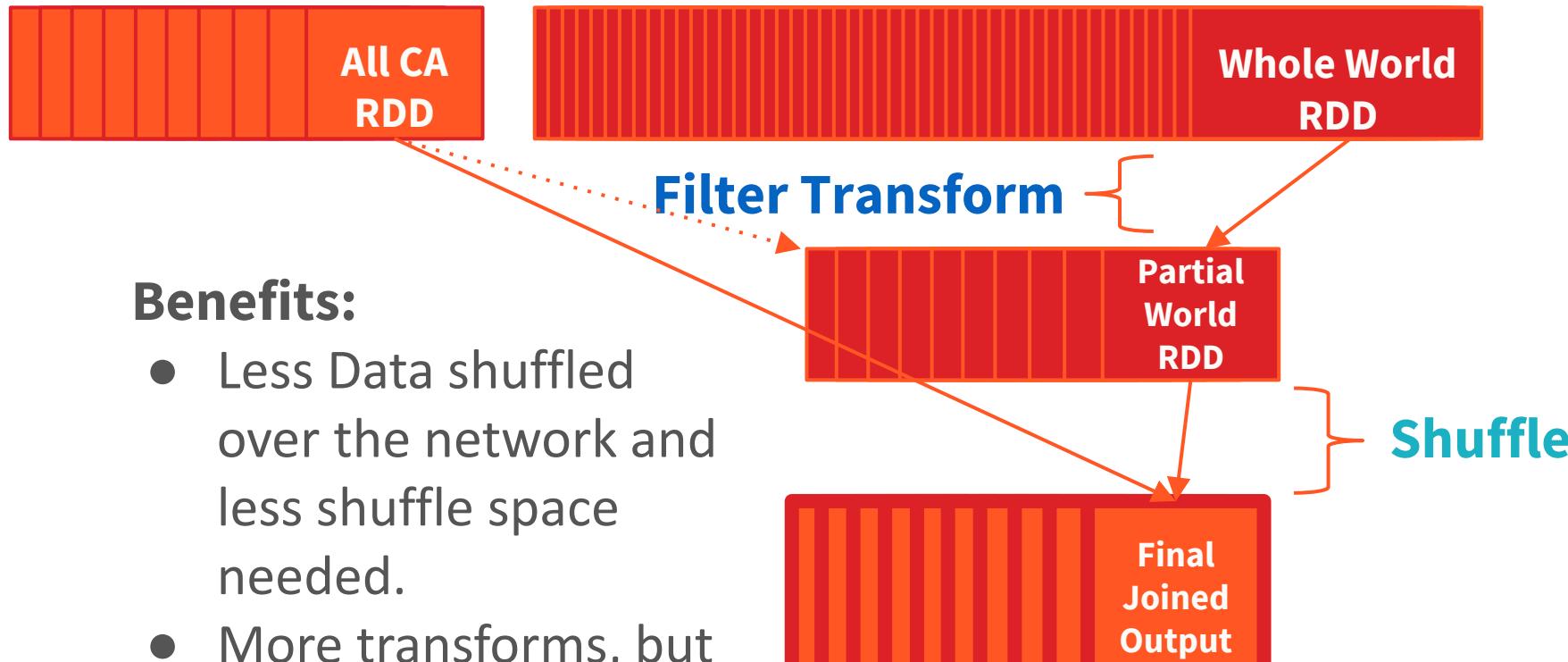


The Size of the Spark Cluster to run this job is limited by the Large table rather than the Medium Sized Table.

Better solution



Filter the Whole World RDD for only entries that match the CA ID



When big is too big?



- It's relative.
- There aren't always strict rules for optimization.

You should understand your data and its unique properties in order to best optimize your Spark Job.

Spark Training

Powered by





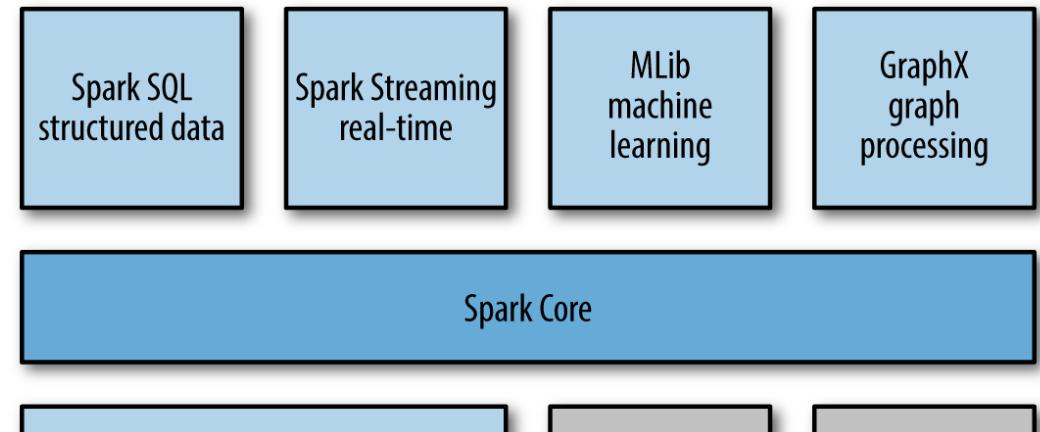
Spark Streaming

- 1. How it works**
2. Features and API
3. Kafka integration
4. Streaming engines comparison

Spark Streaming

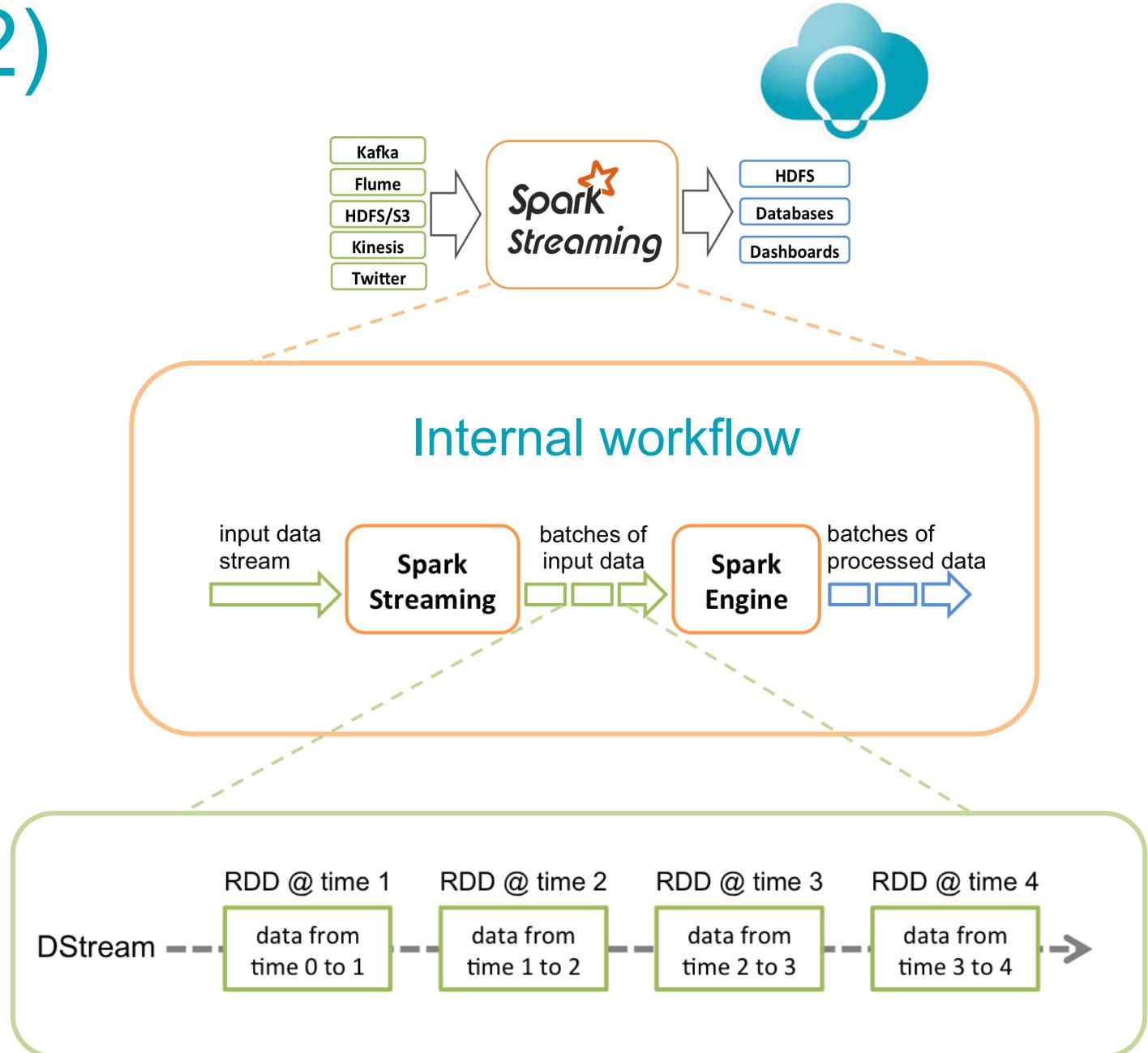


- Spark extension for real-time processing
 - Scalable
 - High-throughput
 - Fault-tolerant
- API for Scala, Java, Python
- Integrated with many sources
 - Kafka
 - Flume
 - Twitter
 - TCP sockets
 - ...



How it works (1/2)

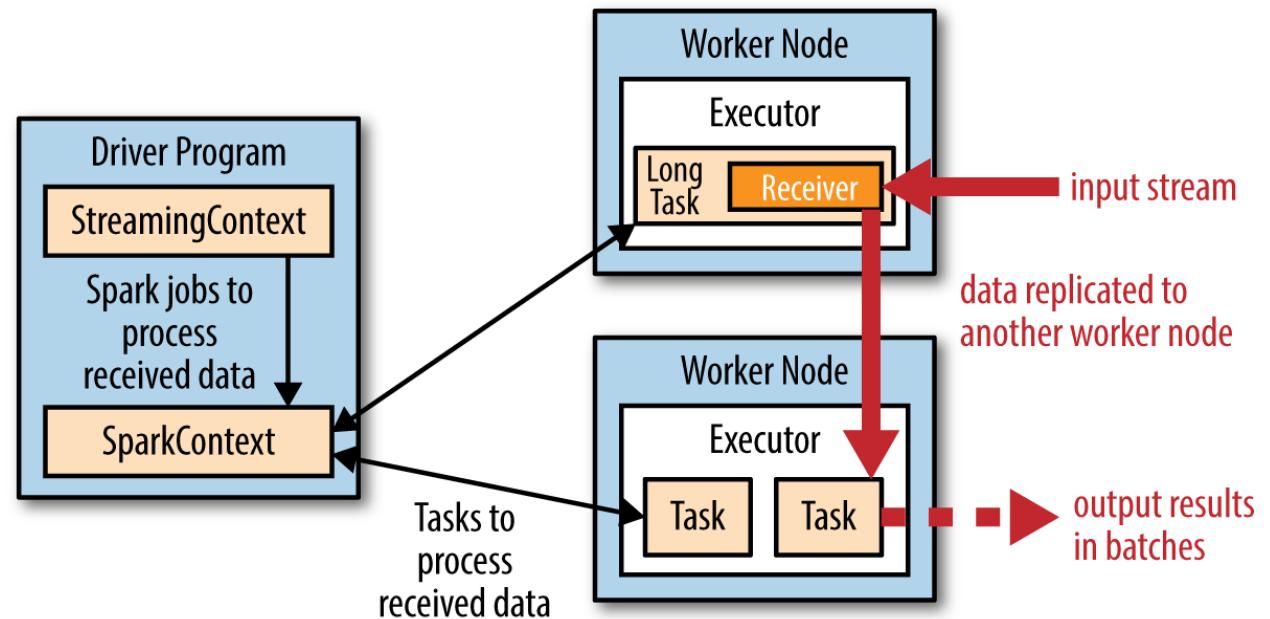
- Mini-batch computation
- Main abstraction:
DStream (Discretized Stream)
- A DStream is just a sequence of RDDs



How it works (2/2)



- **Receivers** are tasks that collect data from input and save them as RDDs
- By default the input data is replicated across another executor
 - Single worker failure tolerated
- Data stored in memory as it happens with cached RDDs
- **Checkpointing** available





Spark Streaming

1. How it works
2. **Features and API**
3. Kafka integration
4. Streaming engines comparison

DStream



- Dstream creation
 - From external input sources
 - Applying transformation to other DStreams
- Many of RDD transformations equivalent are available
- Transformation on DStreams can be grouped as
 - Stateless
 - Stateful
- Output operations available
 - Similar to RDD actions
 - They run periodically on each time step

Stateless transformations



- Batches are independent (i.e. transformations are applied on single batches separately without any knowledge about previous data)
- Common RDD transformations available
 - map(), flatMap()
 - filter()
 - repartition()
 - groupByKey(), reduceByKey()
- Multiple Dstreams can be combined
 - Join(), cogroup(), leftOuterJoin(), ...
 - Union()
- Operation transform() available to provide arbitrary RDD-to-RDD function to act on the DStream

Simple streaming application(1/2)



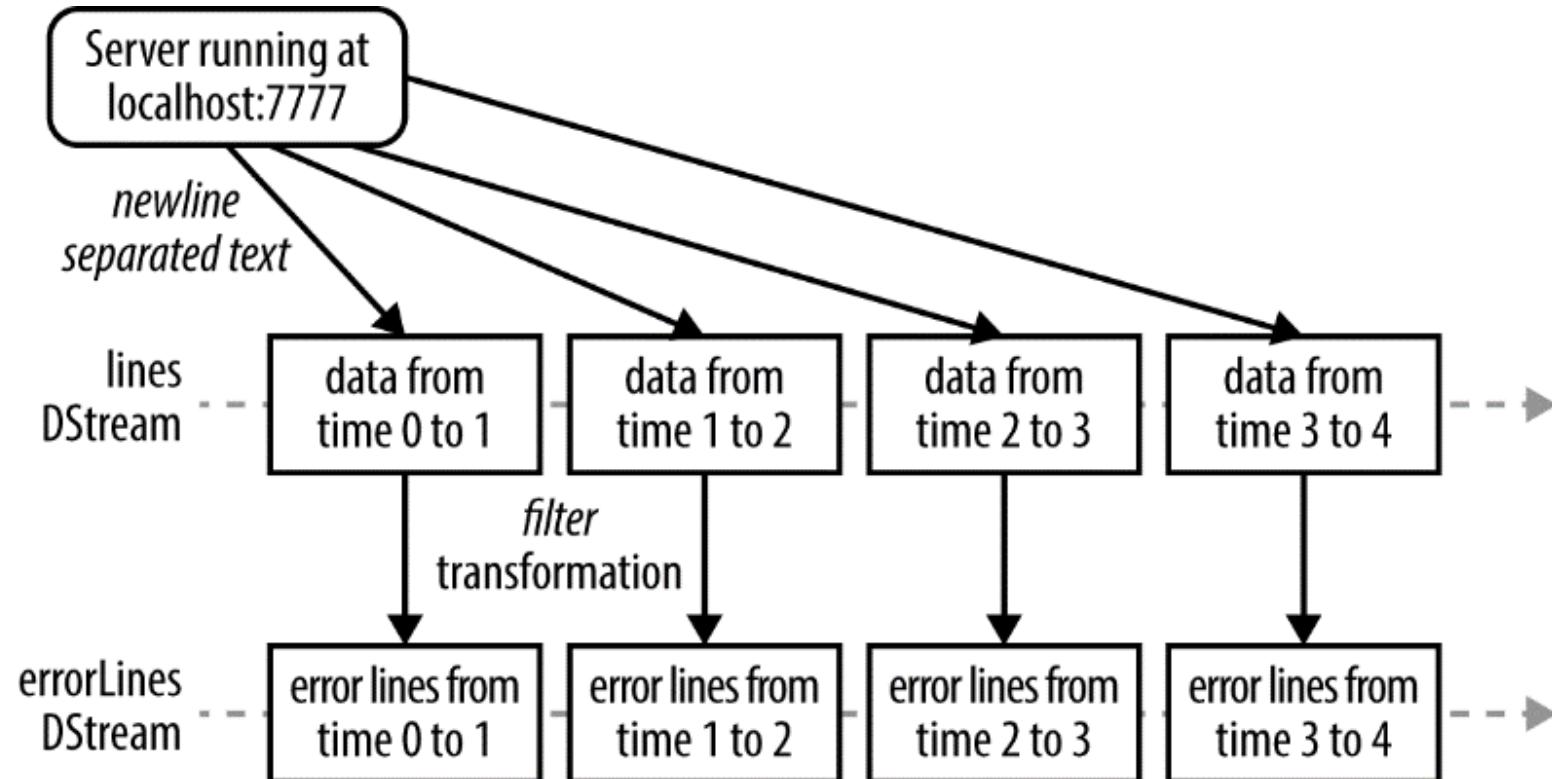
```
//Scala
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds

//Create a StreamingContext with a 2-second batch size
val ssc = new StreamingContext(conf, Seconds(2))
/* Create a DStream using data received after connecting
   to port 7777 on the local machine */
val lines = ssc.socketTextStream("localhost", 7777)
// Filter DStream lines looking for "error" lines
val errorLines = lines.filter(_.toLowerCase.contains("error"))

errorLines.print()

// Start our streaming context and wait the job to finish
ssc.start()
ssc.awaitTermination()
```

Simple streaming application(2/2)



Stateful transformations

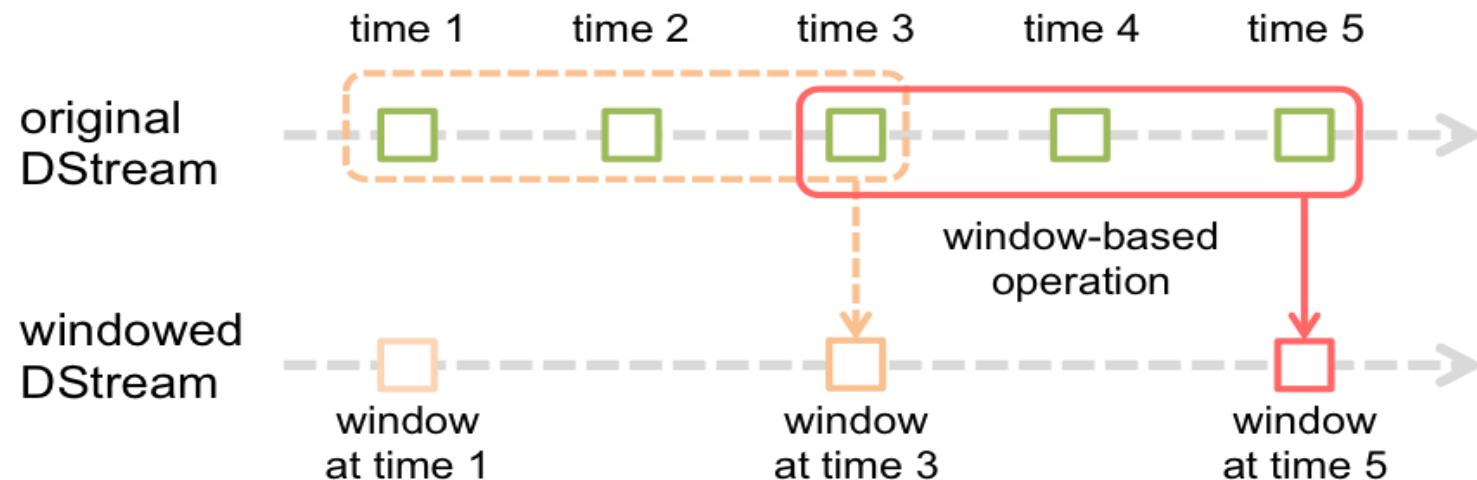


- Operations on DStreams that track data across time
- Two main categories
 - Windowed operations
 - `updateStateByKey()`
- Fault Tolerance achieved through Checkpointing

Windowed transformations



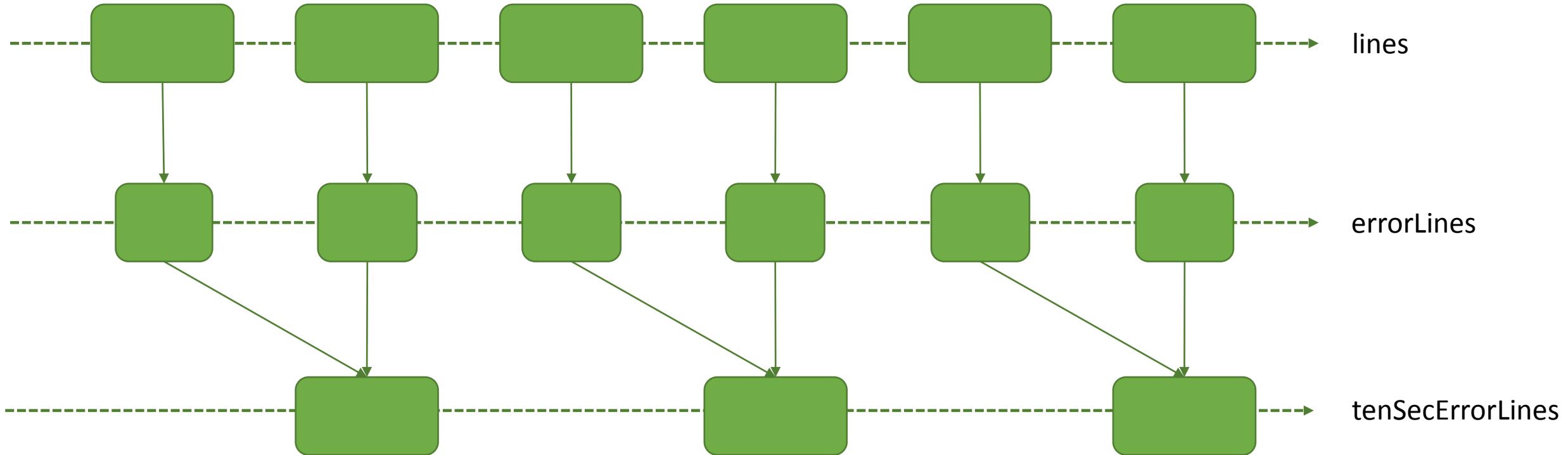
- Combine results from multiple batches
- Act over a sliding window of time periods
- Two main parameters:
 - Window Duration
 - Sliding Duration



Window() function ex.1



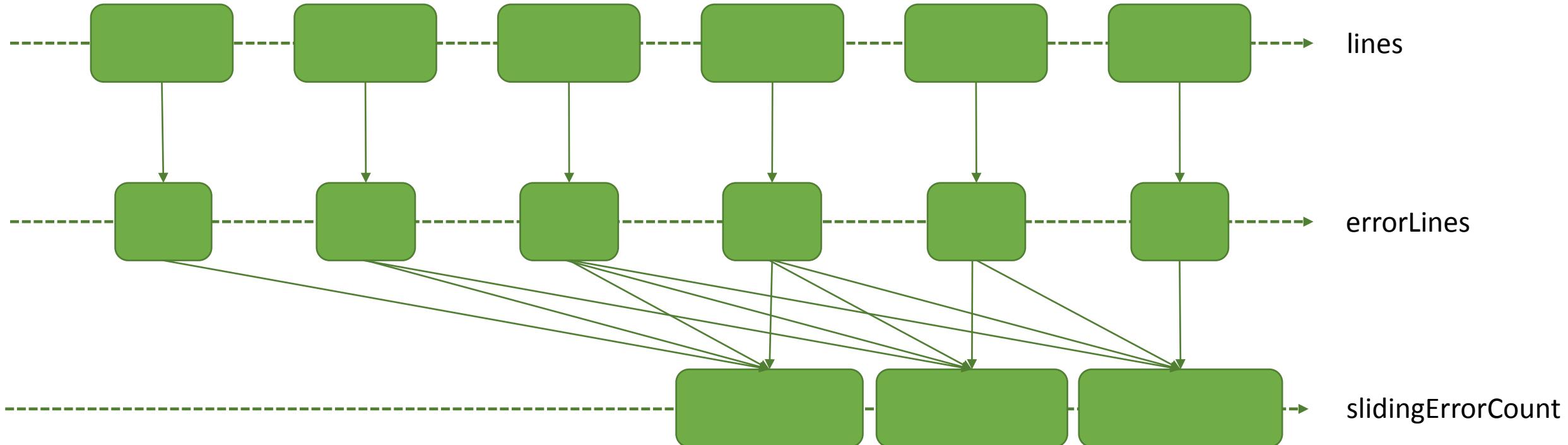
```
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("localhost", 7777)
val errorLines = lines.filter(_.toLowerCase.contains("error"))
val tenSecErrorLines = errorLines.window(Seconds(10), Seconds(10))
val errorCount = tenSecErrorLines.count()
```



Window() function ex.2



```
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("localhost", 7777)
val errorLines = lines.filter(_.toLowerCase.contains("error"))
val slidingErrorCount = errorLines.window(Seconds(20), Seconds(5))
val errorCount = slidingErrorCount.count()
```



Windowed operations

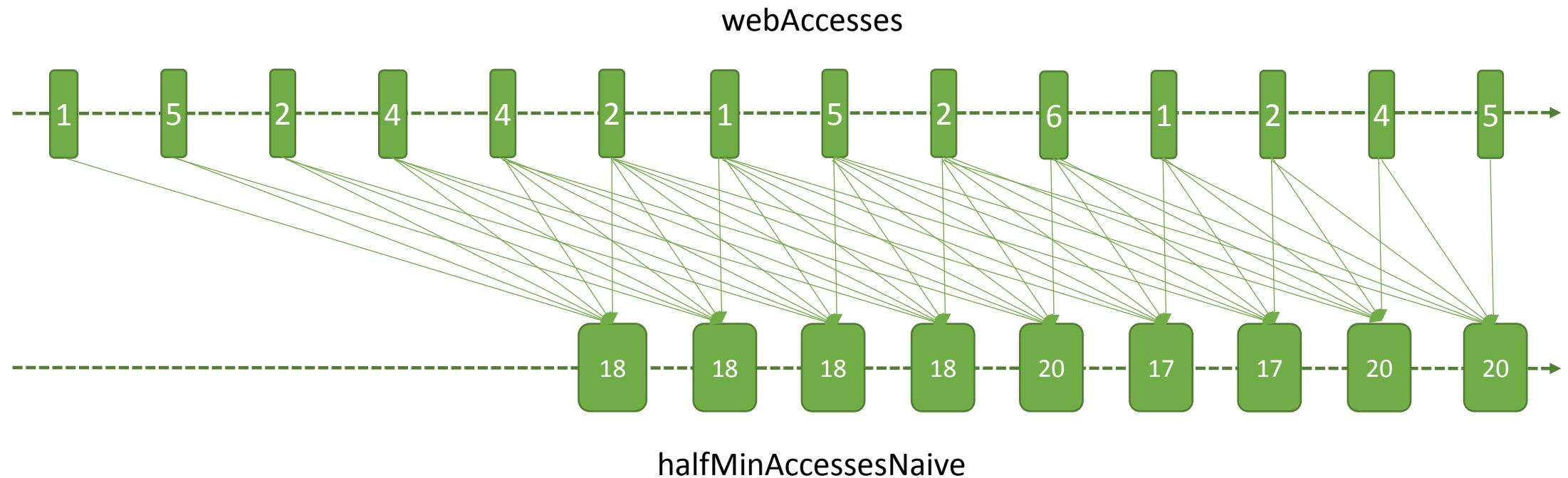


- Two possibilities to work on windows
 - `window()` -> DStream operation
 - Windowed operations
- Windowed operations are introduced for efficiency and convenience
- Examples: `reduceByWindow()`, `reduceByKeyAndWindows()`, `countByWindow()`, ...

reduceByWindow() example (1/2)



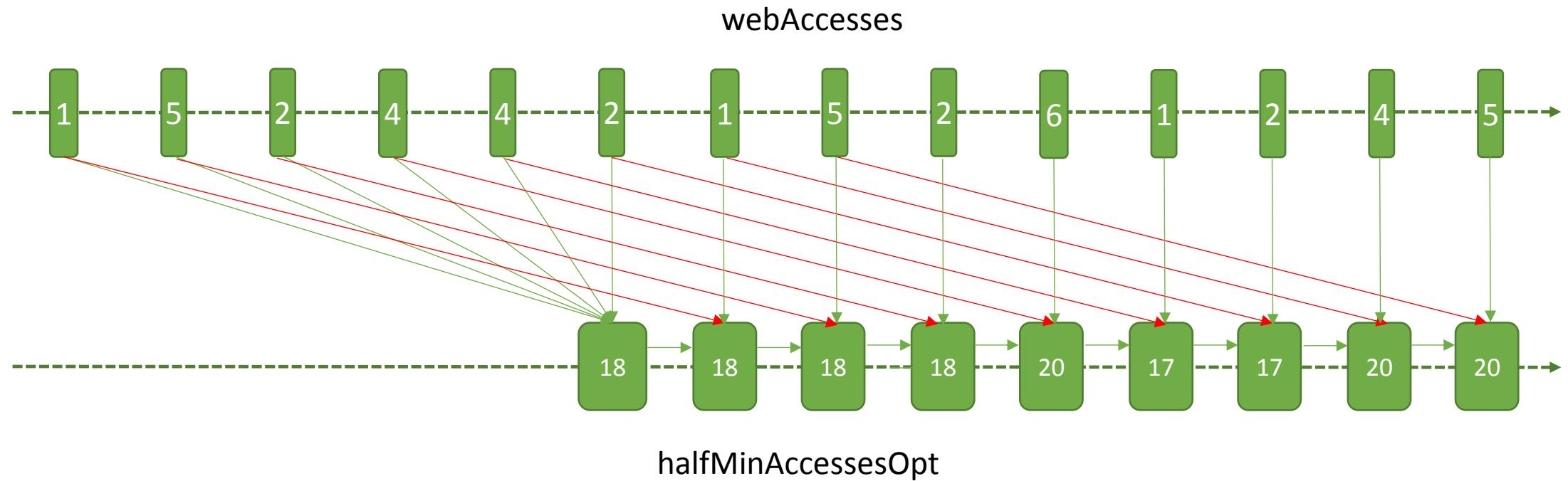
```
// Suppose to have a DStream webAccesses with the accesses in the last 5 seconds
val halfMinAccessesNaive = webAccesses.reduceByWindow(
    { (x, y) => x+y },
    Seconds(30),
    Seconds(5)
)
```



reduceByWindow() example (2/2)



```
// Suppose to have a DStream webAccesses with the accesses in the last 5 seconds
val halfMinAccessesOpt = webAccesses.reduceByWindow(
    { (x, y) => x+y },
    { (x, y) => x-y }
    Seconds(30),
    Seconds(5)
)
```



UpdateStateByKey transformation



- Helpful to maintain state across the batches in a DStream
- A state variable of Key/Value pairs is provided
- A function $update(events, oldState) \Rightarrow newState$ has to be provided (acts on a single Key)
 - events: list of events arrived in the current batch
 - oldState: optional state object
 - newState: optional new state
- The result of a `updateStateByKey()` is a DStream that contains a RDD of (key,state) pairs in each time step

UpdateStateByKey example



```
// Suppose to have a DStream with access logs of a web sites with the response codes
def updateRunningSum(values: Seq[Long], state: Option[Long]) = {
  Some(state.getOrElse(0L) + values.size)
}

val responseCodeDStream = accessLogsDStream.map(log => (log.getResponseCode(), 1L))
val responseCodeCountDStream = responseCodeDStream.updateStateByKey(updateRunningSum _)
```

Output operations



- If no output operation is applied on a DStream and any of its descendants the DStream will not be evaluated (similarly to RDD without final action)
- Common output Operations:
 - *print()*
 - *saveAsTextFile(output_dir, «txt»)*
 - *saveAsHadoopFiles[SequenceFileOutputFormat[Text, LongWritable]]*
- *foreachRDD()* is a generic output operation to run arbitrary computations on the RDDs of the Dstream (e.g., it can be used to write each partition of a RDD to a database)

Checkpointing



- It is possible to checkpoint enough information to a fault-tolerant storage system such that the application can recover from failures.
- **Metadata Checkpointing**
 - Enables the recovery from driver failure
 - Metadata includes: Configuration, DStream operations, Incomplete batches
- **Data Checkpointing**
 - Put a bound on the length of the dependency chain
 - Must be provided when stateful transformations are used

Configure checkpointing (1/2)



- A checkpointing directory has to be defined on the reliable system

```
streamingContext.checkpoint(checkpointDirectory)
```

- To recovery from driver failures the following application behavior is required
 - When the program is being started for the first time, it will create a new StreamingContext, set up all the streams and then call start().
 - When the program is being restarted after failure, it will re-create a StreamingContext from the checkpoint data in the checkpoint directory.

Configure checkpointing (2/2)



```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...) // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory) // set checkpoint directory
    ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _)

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

How to upgrade a 24/7 application



There are two possible mechanisms

- The upgraded Spark Streaming application is started and run in parallel to the existing application. Once the new one (receiving the same data as the old one) has been warmed up and is ready for prime time, the old one can be brought down.
- The existing application is shutdown gracefully. Then the upgraded application can be started, which will start processing from the same point where the earlier application left off.
 - This can be done only with input sources that support source-side buffering (like Kafka, and Flume)
- Checkpointing from previous version run cannot be done
 - Change checkpoint directory or clean up the old one



Spark Streaming

1. How it works
2. Features and API
- 3. Kafka integration**
4. Streaming engines comparison

Kafka integration



- [Apache Kafka](#) is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service.
- There are two approach to kafka integration in Spark Streaming
 - Receiver-based approach
 - Direct approach (experimental)

Receiver-based approach



- Use a Receiver to receive the data.
- The Receiver is implemented using the Kafka high-level consumer API.
- The data received from Kafka through a Receiver is stored in Spark executors, and then jobs launched by Spark Streaming processes the data.
- To ensure zero-data loss, you have to additionally enable Write Ahead Logs in Spark Streaming (introduced in Spark 1.2). This synchronously saves all the received Kafka data into write ahead logs on a distributed file system (e.g HDFS), so that all the data can be recovered on failure.

Receiver-based approach



- Linking
 - groupId = org.apache.spark
 - artifactId = spark-streaming-kafka_2.10
 - version = 1.6.3
- Programming

```
import org.apache.spark.streaming.kafka._

val kafkaStream = KafkaUtils.createStream(streamingContext,
  [ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume])
```



Spark Streaming

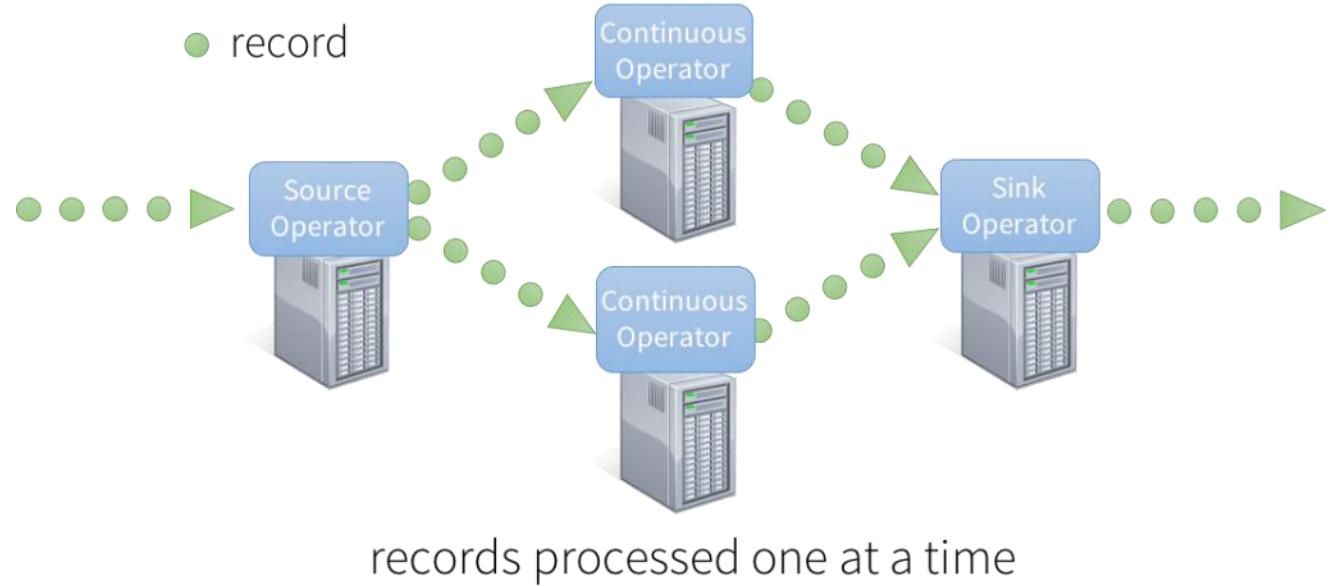
1. How it works
2. Features and API
3. Kafka integration
- 4. Streaming engines comparison**

Traditional stream processing



- 1-by-1 record processing
- Statically workload allocation
- Fully dedicated to streaming

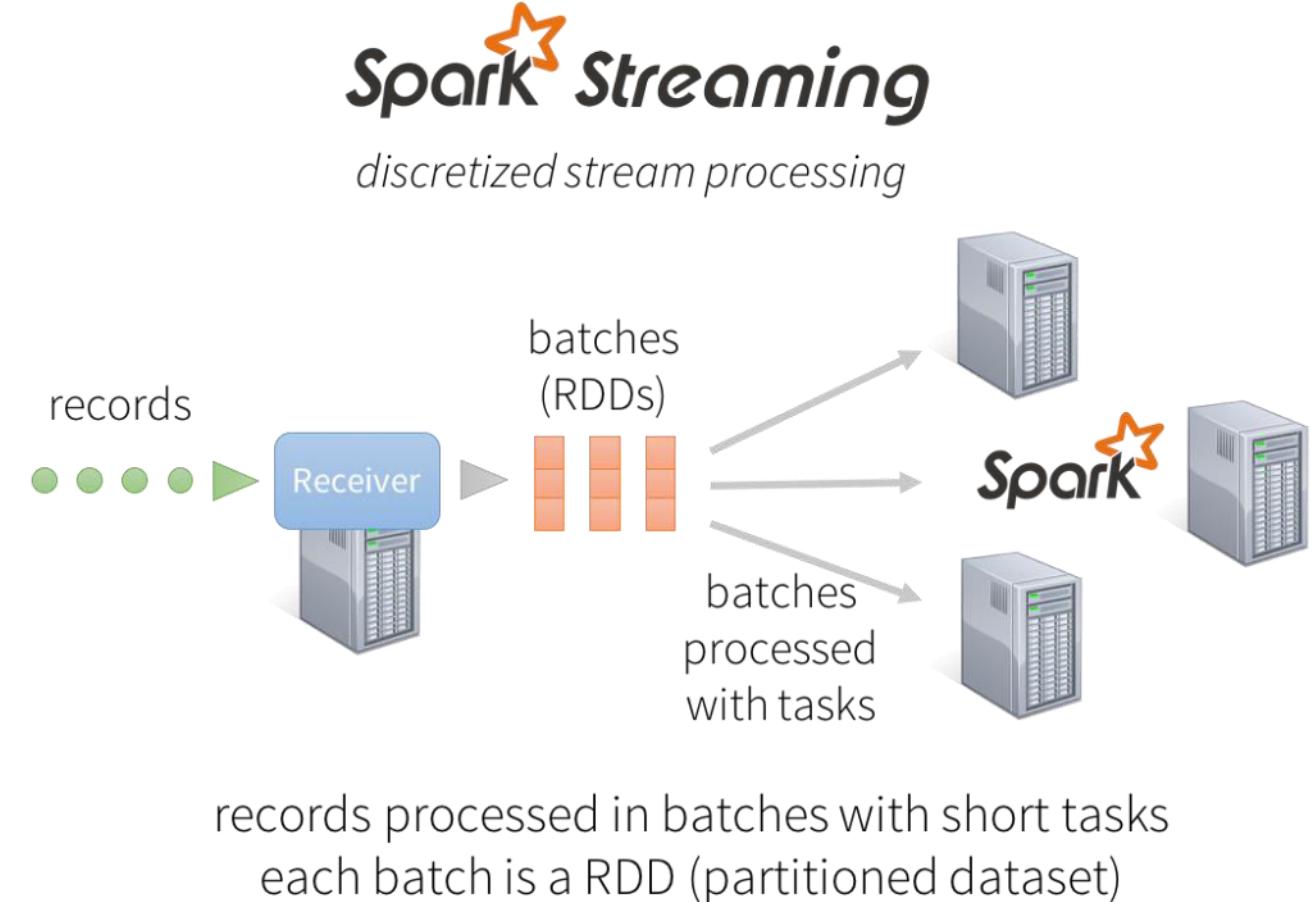
Traditional stream processing systems
continuous operator model



Spark Streaming processing



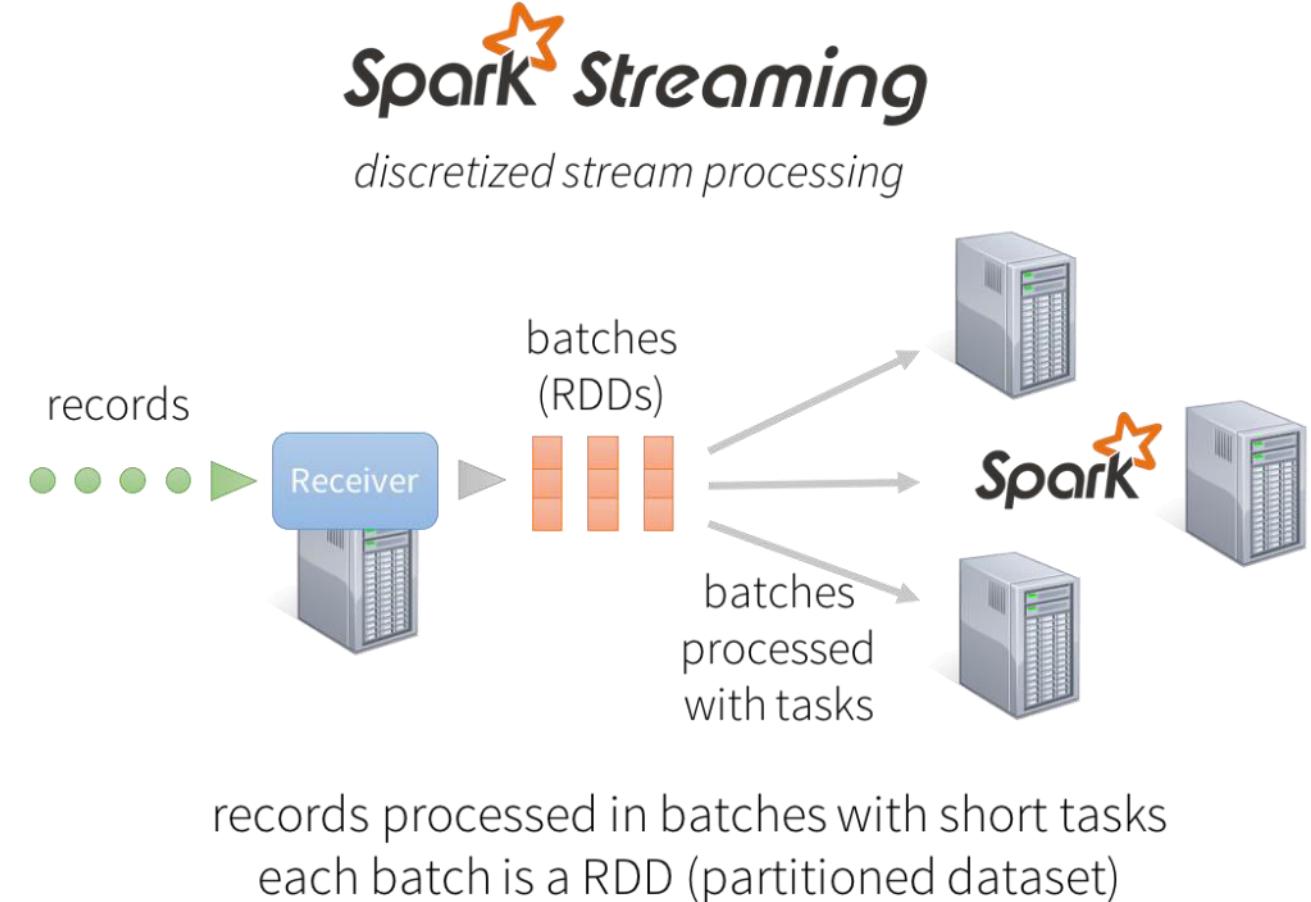
- Mini-batch processing
- Dynamic workload allocation
 - Data locality
 - Available resources
- Unification with batch computation and workloads



Spark Streaming processing



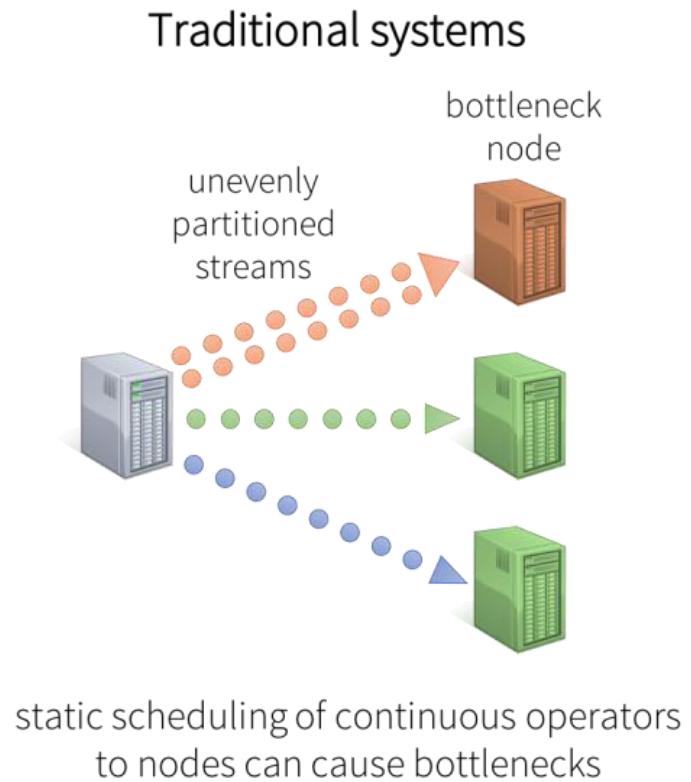
- Mini-batch processing
- Dynamic workload allocation
 - Data locality
 - Available resources
- Unification with batch computation and workloads



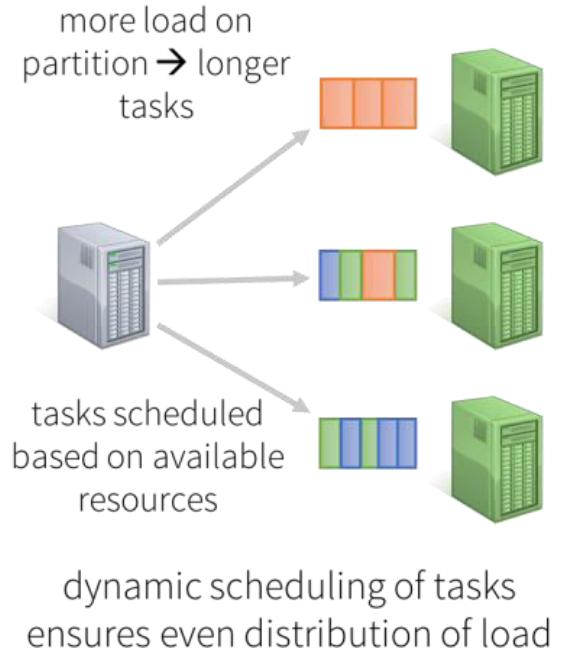
Dynamic load balancing



- In traditional system higher possibility to have bottlenecks
- In Spark Streaming natural load balancing across workers



Spark Streaming

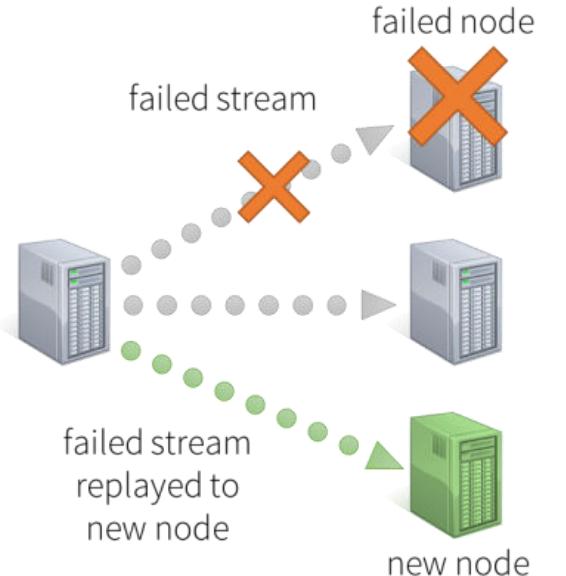


Failure and struggle recovery



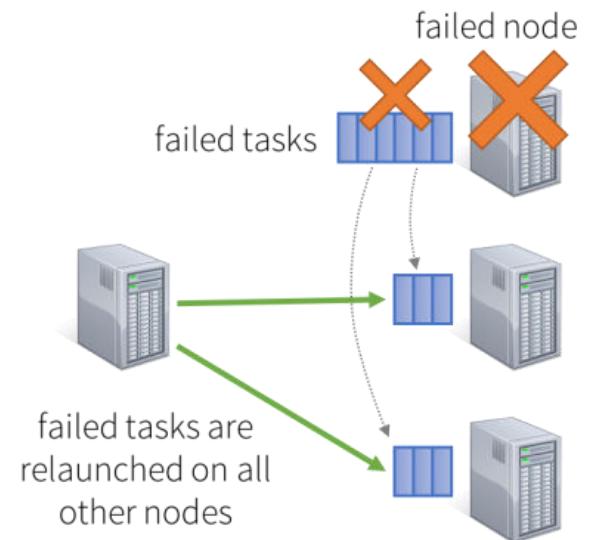
- In traditional system dedicated node for the recovery
- In Spark Streaming tasks recovered throughout multiple nodes

Traditional systems



slower recovery by using single node
for recomputations

Spark Streaming



faster recovery by using multiple
nodes for recomputations

Unification of different workloads



- Batch + Streaming workload combination example

```
// Create data set from Hadoop file
val dataset = sparkContext.hadoopFile("file")
// Join each batch in stream with the dataset
kafkaDStream.transform { batchRDD =>
  batchRDD.join(dataset).filter(...)
}
```

Main engines comparison



	Spark Streaming	Storm	Flink
Current version	1.6.1	1.0.0	1.0.2
Category	ESP	ESP/CEP	ESP/CEP
Event size	micro-batch	single	single
Available since [incubator since]	Feb 2014 [2013]	Sep 2014 [Sep 2013]	Dec 2014 [Mar 2014]
Contributors	838	207	159
Main backers	AMPLab Databricks	Backtype Twitter	dataArtisans
Delivery guarantees	exactly once at least once [with non-fault-tolerant sources]	at least once	exactly once
State management	checkpoints	record acknowledgement	distributed snapshots
Fault tolerance	yes	yes	yes
Out-of-order processing	no	yes	yes
Event prioritization	programmable	programmable time-based	programmable time-based count-based
Windowing	time-based	time-based count-based	count-based
Back-pressure	yes	yes	yes
Primary abstraction	DStream	Tuple	DataStream
Data flow	application	topology	streaming dataflow
Latency	medium	very low	low [configurable]
Resource management	YARN Mesos	YARN Mesos	YARN
Auto-scaling	yes	no	no
In-flight modifications	no	yes [for resources]	no
API	declarative	compositional	declarative
Primarily written in	Scala	Clojure Scala	Java
API languages	Scala Java Python	Java Clojure Python Ruby	Java Scala Python
Notable users	Kelkoo Localytics AsiaInfo Opentable Fairadata Guavus	Yahoo! Spotify Groupon Flipboard The Weather Channel Alibaba Baidu Yelp WebMD	King Otto Group

Ian Hellstrom analysis
(March 2016)

Hint: Check Structured Streaming (1/2)



- Streaming Transparency enhanced
- Continuous aggregation functions
- Handling of late data
- Integration with batch computation programmatically transparent

```
//Batch ETL with DataFrames  
input = spark.read  
    .format("json")  
    .load("source-path")  
  
result = input  
    .select("name", "surname")  
    .where("age > 15")  
  
result.write  
    .format("parquet")  
    .save("destination-path")
```

From batch to streaming computation



```
//Streaming ETL with DataFrames  
input = spark.readStream  
    .format("json")  
    .load("source-path")  
  
result = input  
    .select("name", "surname")  
    .where("age > 15")  
  
result.writeStream  
    .format("parquet")  
    .start("destination-path")
```

Hint: Check Structured Streaming (2/2)



Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✓	✓	✗	✗	✗	✗
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✓	some	some	some	✗	✗
Interactive Queries	✓	✓	✗	✗	✗	✗
Joins with Static Data	✓	✓	✗	✗	✗	✗

Check <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
Can be a little bit «unbalanced» (by Databricks!)