Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

BigData, Hadoop, HDFS, MapReduce & Spark- Part 1

Introduction

Data is one of the most important assets of any organization. The scale at which data is being collected and used in organizations is growing beyond imagination. The speed at which data is being ingested, the variety of the data types in use, and the amount of data that is being processed and stored are breaking all-time records every moment. It is very common these days, even in small-scale organizations, that data is growing from gigabytes to terabytes to petabytes. For the same reason, the processing needs are also growing that ask for capability to process data at rest as well as data on the move.

Take any organization; its success depends on the decisions made by its leaders and for making sound decisions, you need the backing of good data and the information generated by processing the data. This poses a big challenge on how to process the data in a timely and cost-effective manner so that right decisions can be made. Data processing techniques have evolved since the early days of computers. Countless data processing products and frameworks came into the market and disappeared over these years. Most of these data processing products and frameworks were not general purpose in nature. Most of the organizations relied on their own bespoke applications for their data processing needs, in a silo way, or in conjunction with specific products.

What is Big data?

Big data is a phrase or methodology that describes the aquire, cure, process and store the humongous volume of data that is limited to handled by the traditional systems with in the stipulated period of time.

Big data is simply the large sets of data that businesses and other parties put together to serve specific goals and operations. Big data can include many different kinds of data in many different kinds of formats. As a rule, it's raw and unsorted until it is put through various kinds of tools and handlers.

Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time, extremely handles large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions.

Types Of Big Data:

Broadly classified into man made and machine made data such as click stream event logs, device logs, photos, videos, historical datawarehouse data, geo satellite data etc. Shared nothing architecture.

Big data sourced from web data, social media, click stream data, man-made data, machine/device data etc.

Web log click stream data eg. Online banking transactions comparison between traditional and bigdata.

Man made data eg. Sentimental analysis through twitter, fb etc. Product campaign analysis.

Machine/device data such as geospacial data from gps devices such as Telematics, truck roll for efficient technian productivity measurement.

Characteristics:

- Volume – huge volume of machine and man made data such as logs, click events, media files and historical warehouse data.

  Eg . FB, Historical data (Bank example, Airlines example) etc.

- Velocity – Speed at which data is generated, captured, processed and stored.
  Eg. Tweets, click stream events, GPS Geospatial data etc.

- Variety – not only structured, it can accommodate un structured, semi structured, media, logs etc.
  Eg. STB logs, logs generated from different devices.
- Veracity – Trustworthy of the data we receive, how much true data we receive. Data accuracy is based on the veracity of source data that we receive at the very lower granular level. **Eg.** Website click steam logs.
- Value – ROI can be derived base on utilizing the data stored and bringing business insights from it.
  Eg. Click stream logs ROI derived based on the interest shown by the customer in the webpage (positive and negative scores).
- Variable – data varies from time to time wrt size, type that is unpredictable.
  Eg. A company started with only deals (Snapdeal) then enter into retail marketing and business grown in all media.

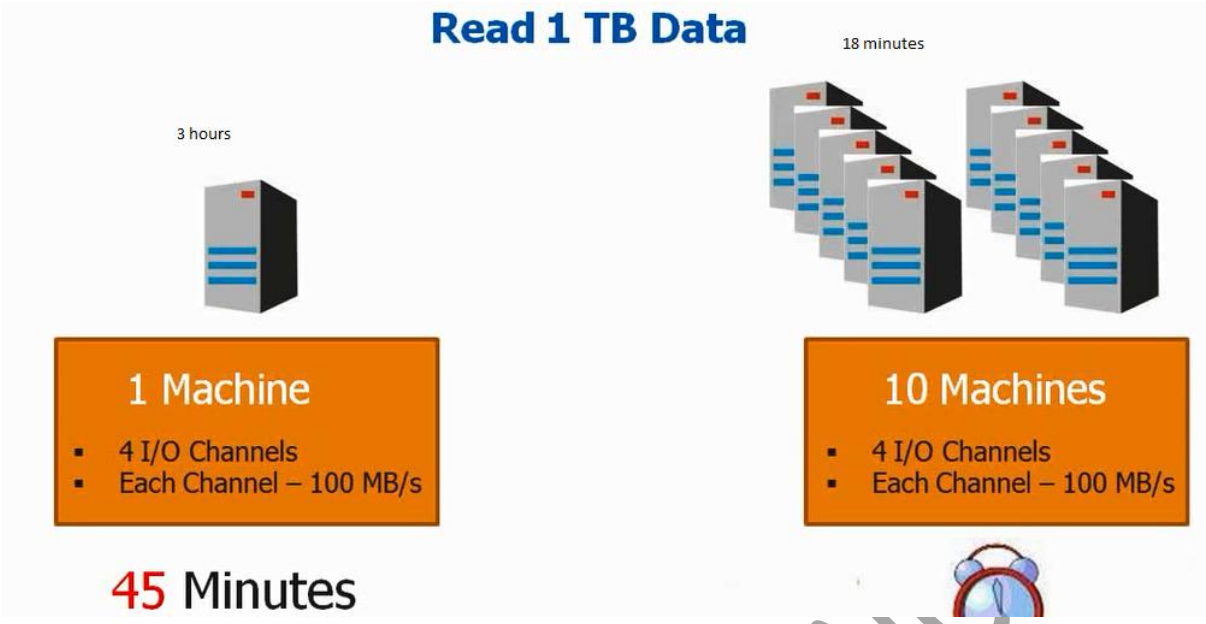
What is hadoop:

Hadoop is a open source Apache software stack that runs on cluster of commodity hardware that provides distributed storage and processing of large data sets.It is highly scalable, fault tolerant, data locality, highly distributed coordination, distributed MPP for heterogeneous data and heterogeneous software frameworks, low cost commodity hardware, high availability, highly secured, open source, resilient etc.

Hadoop Distributed File System (HDFS)

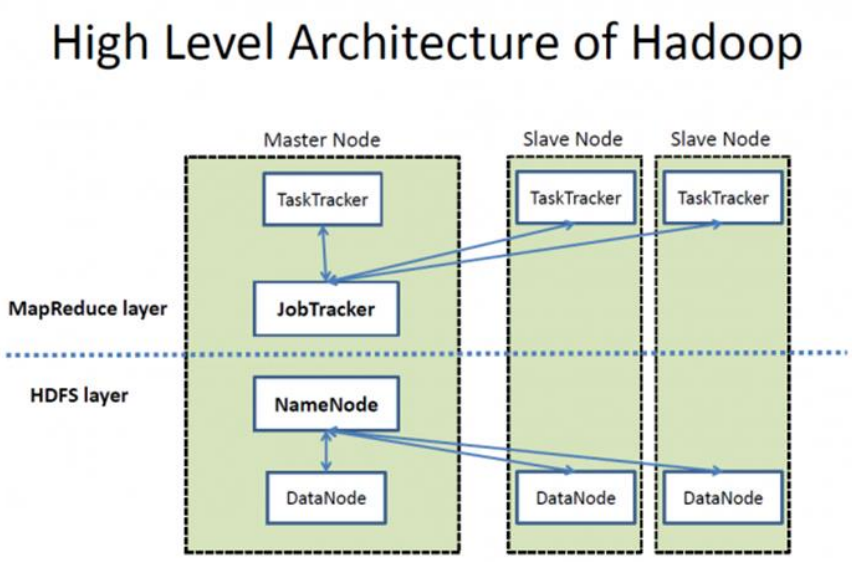
HDFS is a distributed file system that manages and provides high-throughput distributed access to data. HDFS creates multiple replicas of each data block and distributes them on computers throughout a cluster to enable resilient access hence improve fault tolerance.

Why HDFS

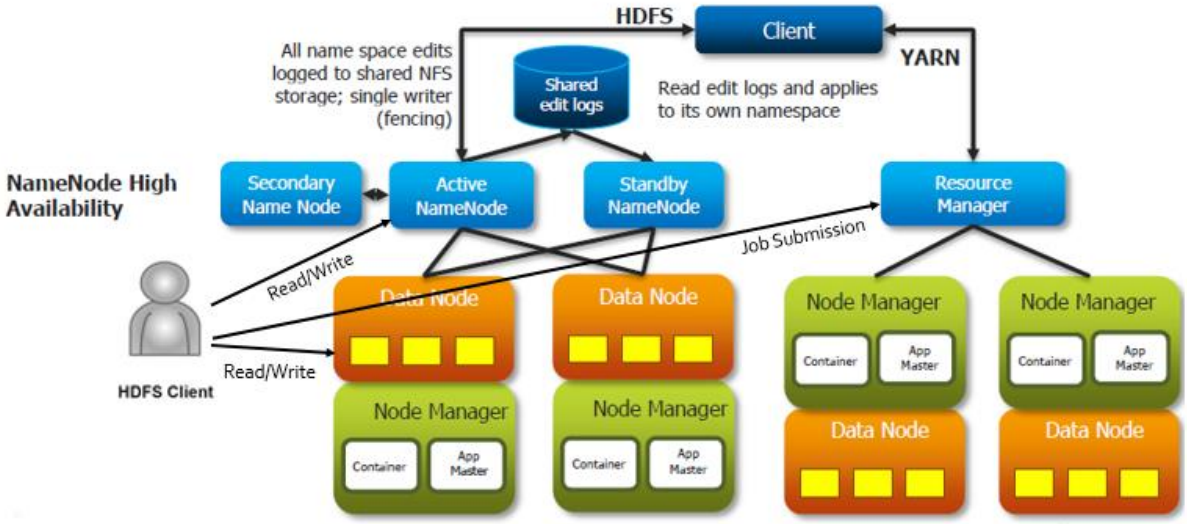


Where ever you create directories in hdfs will be displayed commonly in all nodes in the cluster as a single file system unlike creating directories in each node separately.

Architecture



HDFS & MapReduce Architecture



NameNode is the master node or DFS Master of the system. It maintains the name system (directories and files) and manages the blocks which are present on the DataNodes.

Metadata Components:

fsimage – Stores the inode details like modification time, access time, access permission, replication. **editlogs** – This keeps tracking of each and every change that is being done on HDFS. (Like adding a new file, deleting a file, moving it between folders..etc).

Any change that we do to HDFS will be tracked in edit logs, not in the fsimage. So the edit logs file will keep on growing whereas the fsimage size remains the same. This won't have any impact unless or until we restart the cluster. When we restart the cluster, the fsimage needs to get loaded in to the main memory. Since all changes are present in the edilogs not in the fsimage, hadoop will try to write editlogs changes to fsimage (this takes some time depends on the size of the editlogs file). If you have not restarted your cluster for months together and if you are about to do it, then there will be a vast down time as editlogs size would have grown like anything.

The namenode maintains the entire metadata in RAM, which helps clients receive quick responses to read requests. Therefore, it is important to run namenode from a machine that has lots of RAM at its disposal. The higher the number of files in HDFS, the higher the consumption of RAM. The namenode daemon also maintains a persistent checkpoint of the metadata in a file stored on the disk called the fsimage file.

Whenever a file is placed/deleted/updated in the cluster, an entry of this action is updated in a file called the edits logfile. After updating the edits log, the metadata present in-memory is also updated accordingly.

It is important to note that the fsimage file is not updated for every write operation.

Secondary Name Node:

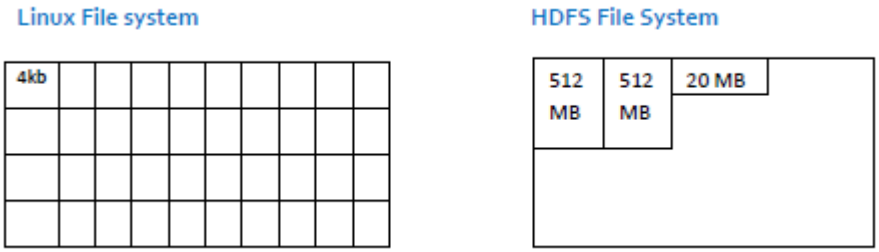
1. Secondary name node act as a backup node in the case of name node crashed and the fsimage/editlog is lost completely in namenode. With the Admin interaction the FSImage from Secondary name node can be copied to a new name node and bring the name node up and running using the fsimage copied from SNN.
2. The secondary name node is responsible for performing periodic housekeeping functions for the *NameNode*. It only creates checkpoints of the filesystem present in the *NameNode*.
3. The *Secondary NameNode* is not a failover node for the *NameNode*.

DataNode

1. Data Nodes are the slaves which are deployed on each machine and provide the actual storage.
2. They are responsible for serving read and write requests for the clients.
3. All datanodes send a heartbeat message to the namenode every 3 seconds to say that they are alive. If the namenode does not receive a heartbeat from a particular data node for 10 minutes, then it considers that data node to be dead/out of service and initiates replication of blocks which were hosted on that data node to be hosted on some other data node.
4. The data nodes can talk to each other to rebalance by move and copy data around and keep the replication high.
5. When the datanode stores a block of information, it maintains a checksum for it as well. The data nodes update the namenode with the block information periodically and before updating verify the checksums. If the checksum is incorrect for a particular block i.e. there is a ***disk level corruption for that block,*** it skips that block while reporting the block information to the namenode. In this way, namenode is aware of the disk level corruption on that datanode and takes steps accordingly.
6. **Blockreport** - The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory. When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode.

Terminologies

Daemon: process which runs in background and has no controlling terminal.
Node: A single machine in cluster
Rack: A computer rack (commonly called a rack) is a metal frame used to hold various hardware devices such as servers, hard disk drives, modems and other electronic equipment
Cluster: A cluster is a group of servers and other resources that act like a single system and enable high availability and, in some cases, load balancing and parallel processing.
Block: A block is the smallest unit of data that can be stored or retrieved from the disk. File systems deal with the data stored in blocks.



YARN – Yet Another Resource Negotiator is a resource manager that was created by separating the processing engine and resource management capabilities of MapReduce as it was implemented in Hadoop 1. YARN is often called the Data operating system of Hadoop because it is responsible for managing and monitoring workloads, maintaining a multi-tenant environment, implementing security controls, and managing high availability features of Hadoop.

Like an operating system on a server, YARN is designed to allow multiple, diverse user applications to run on a multi-tenant platform. In Hadoop 1, users had the option of writing MapReduce programs in Java, in Python, Ruby or other scripting languages using streaming, or using Pig, a data transformation language. Regardless of which method was used, all fundamentally relied on the MapReduce processing model to run.

YARN supports multiple processing models in addition to MapReduce. One of the most significant benefits of this is that we are no longer limited to working the often I/O intensive, high latency MapReduce framework. This advance means Hadoop users should be familiar with the pros and cons of the new processing models and understand when to apply them to particular use cases.

YARN Components

1. ResourceManager (RM)
2. ApplicationsManager
3. NodeManager (NM)
4. ApplicationMaster (AM)
5. Container

I) ResourceManager (RM)

The ResourceManager (RM) is the key service offered in YARN. Clients can interact with the framework using ResourceManager. ResourceManager is the master for all other daemons available in the framework.

a) Scheduler

1. The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc.
2. It is a pure scheduler since it does not monitor or track the status of the application instead it purely performs its scheduling function based the resource requirements of the applications

3. It schedules the resources depending on the resource “Container”
4. The Scheduler has the pluggable policy plug-in which is responsible for partitioning the cluster resources among the various queues, applications etc, for example a) CapacityScheduler, b) FairScheduler

b) ApplicationsManager

1. Applications Manager is responsible for accepting job-submissions.
2. Assigning the first container for executing the application specific Application-Master.
3. Provides the service for restarting the Application-Master container on failure.

II) NodeManager (NM)

1. Node-Manager is responsible for Container management.
2. Monitoring container resource usage (like cpu, memory, disk, network).
3. Reporting to the Resource-Manager/Scheduler.

III) ApplicationMaster (AM)

1. Application Master is responsible for negotiating resources with the ResourceManager and for working with the NodeManagers to start the containers.
2. Application-Master is responsible for negotiating appropriate resource containers from the Scheduler.
3. Tracking the status and monitoring progress for applications running in each containers under the Application-Master.

IV) Container

Resource Container incorporates elements such as memory, cpu, disk, network, Command line to launch the process within the container, Environment variables and Local resources necessary on the machine prior to launch, such as jars, shared-objects, auxiliary data files, Security-related tokens etc.

Hadoop Ecosystems – No Unified Vision

- Sparse Modules
- More of Batch
- Diversity of Tools/APIs
- Huge coding efforts
- Heavily I/O Bounded
- High Latency

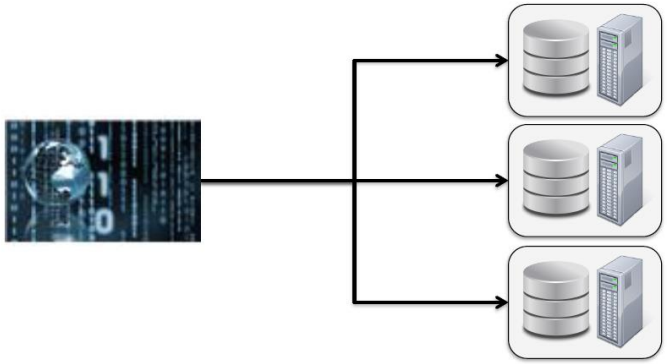
General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

Spark is a **Java Virtual Machine** (**JVM**) based distributed data processing engine that scales, and it is fast compared to many other data processing frameworks. Spark was originated at the *University of California Berkeley* and later became one of the top projects in Apache.

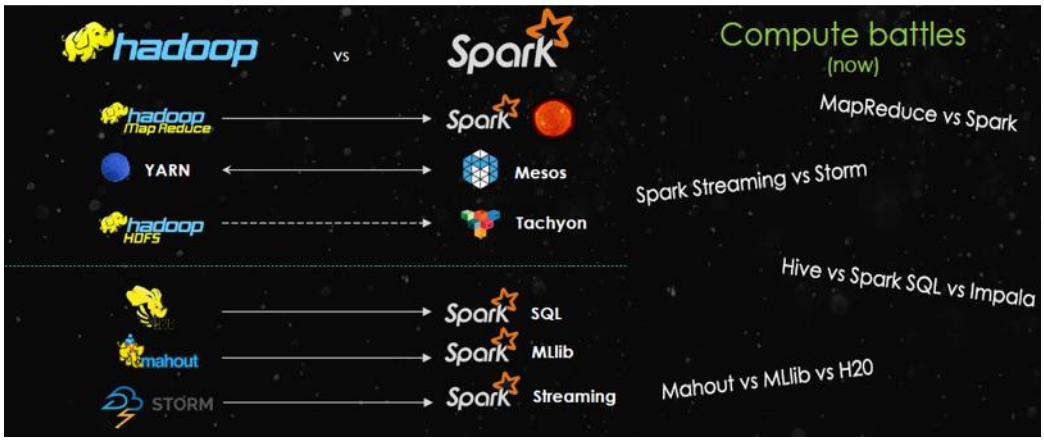
The research paper, *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*, talks about the philosophy behind the design of Spark.

Big Data Processing

- **Hadoop introduced a radical new approach based on two key concepts**
  - Distribute the data when it is stored
  - Run computation where the data is
- **Spark takes this new approach to the next level**
  - Data is distributed in memory



Spark vs Others



MapReduce Execution Model

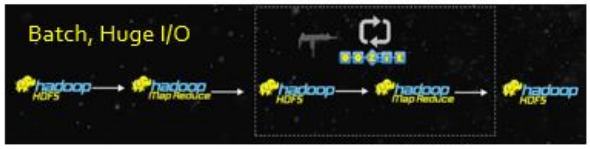
MapReduce was great for batch processing, but users quickly needed to do more:

- More complex, multi-pass algorithms
- More interactive ad-hoc queries
- More real-time stream processing

Result: many *specialized* systems for these workloads

Consider Hive as main SQL tool

- Typical Hive query is translated to 3-5 MR jobs
- Each MR would scan put data to HDD 3+ times
- Each put to HDD – write followed by read
- Sums up to 18-30 scans of data during a single Hive query

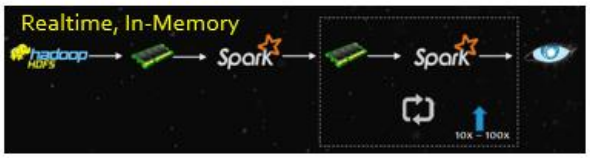


Spark Execution Model

Spark offers you

- Lazy Computations
  - Optimize the job before executing
- In-memory data caching
  - Scan HDD only once, then scan your RAM
- Efficient pipelining
  - Avoids the data hitting the HDD by all means

- Allows optimizations
- Reduce disk I/O
- Reduce shuffle I/O
- Single pass through dataset
- Parallel execution
- Task pipelining



SPARK-BASICS

Unified Big Data Processing Engine

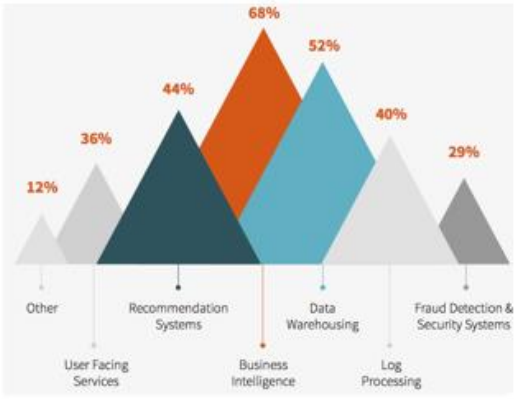
Brief Introduction to Apache Spark

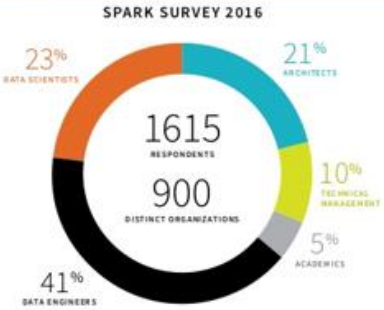
- Apache Spark™ is a fast and general engine for large-scale data processing framework for massive parallel computing (cluster)
- Harnessing power of cheap memory
- Written using Scala, Java and Python languages
- High-level APIs support in Scala, Java and Python
- Has had 36,338 commits made by 1,319 contributors representing 1,068,621 lines of code with proper comments
- Developers from 50+ companies includes UC Berkley, Cloudera, Yahoo, Databricks, Intel, Groupon etc.,
- Apache Committers from 16+ organizations

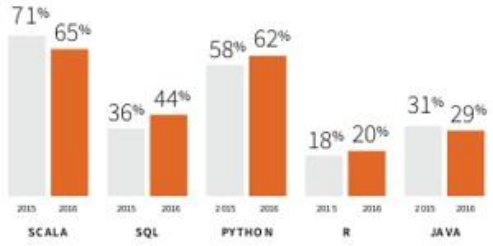
Languages

Scala	76%	Python	9%
Java	7%	9 Other	8%

Spark Opportunities & Solutions

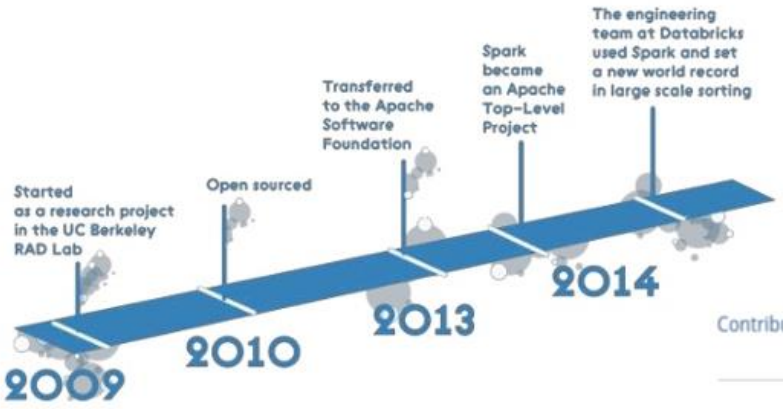


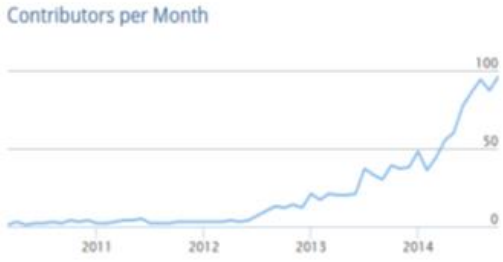




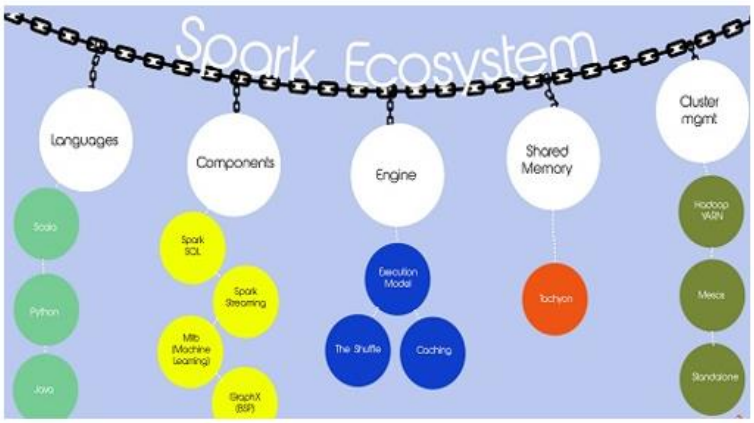
INCEPTEZ TECHNOLOGIES

Brief History

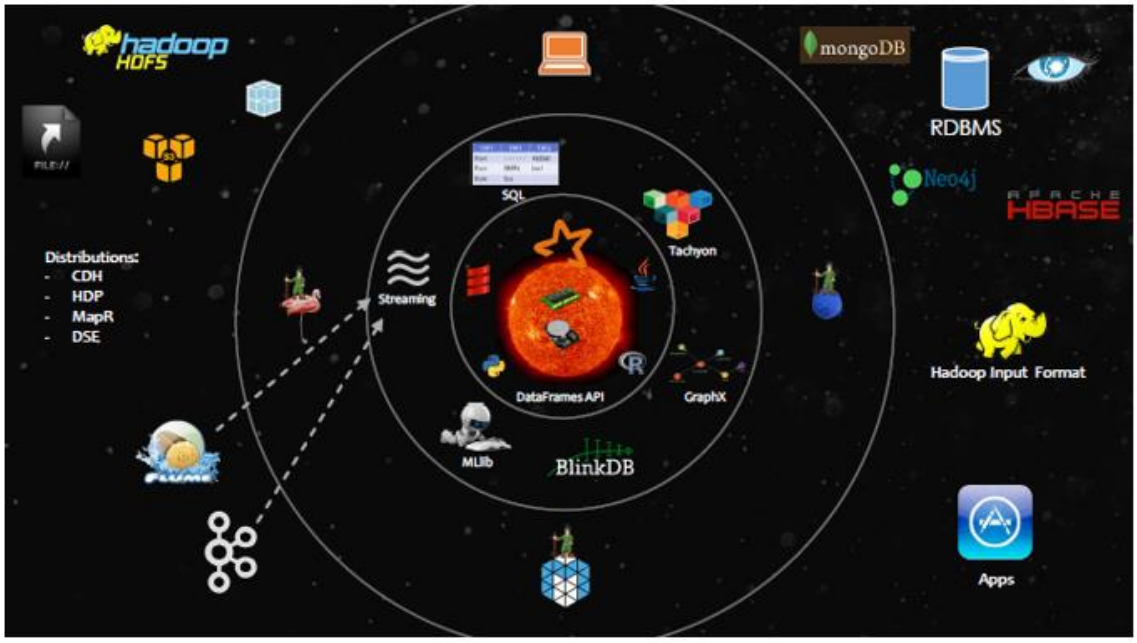




Capabilities and Ecosystems

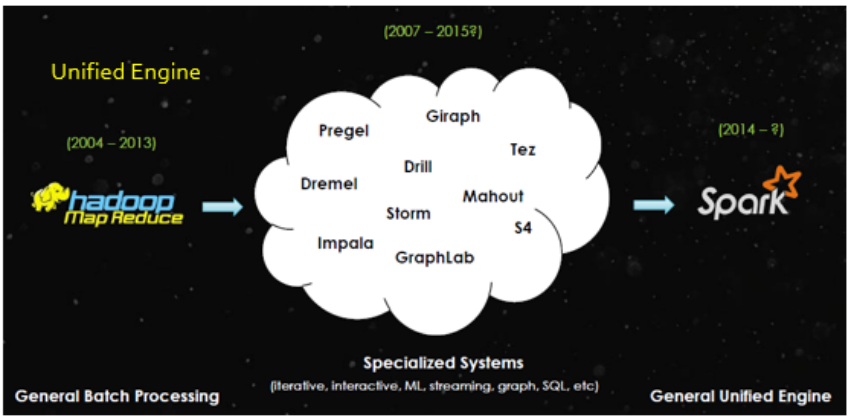


Spark Universe

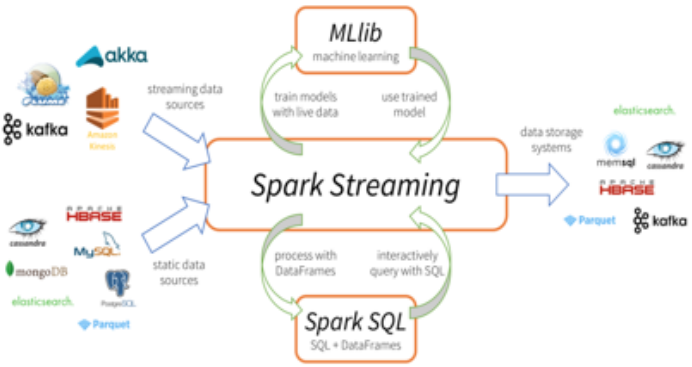


Why Spark

- Distributed data analytics Unified system
- No copying or ETL of data between systems
- Combine processing types in one program
- Code reuse
- One system to learn
- One system to maintain
- Realtime, Iterative, In memory

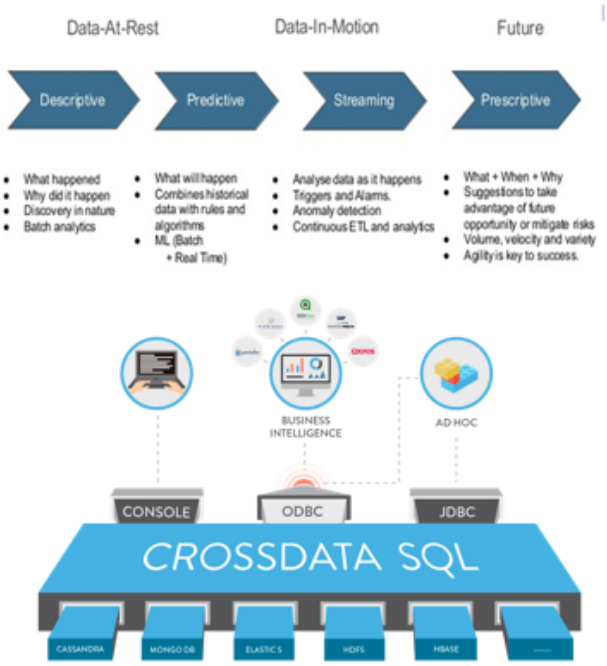


Technical Benefits

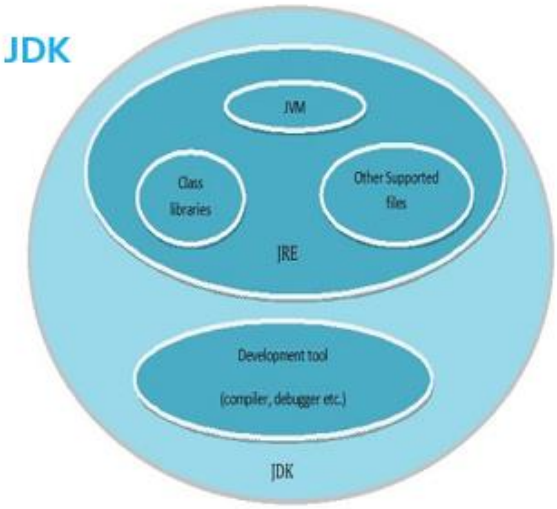


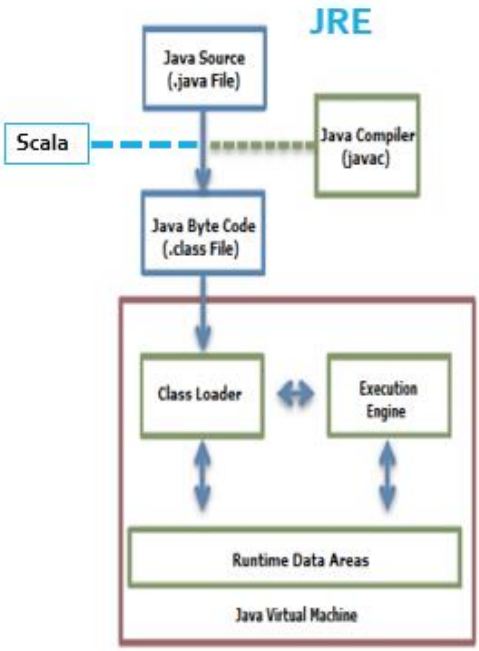
- Alternative to MapReduce
- Compatibility with HDFS, HBASE and YARN
- Spark SQL component to access structured data
- Stable API
- Support for multiple languages
- Library support

Business Benefits

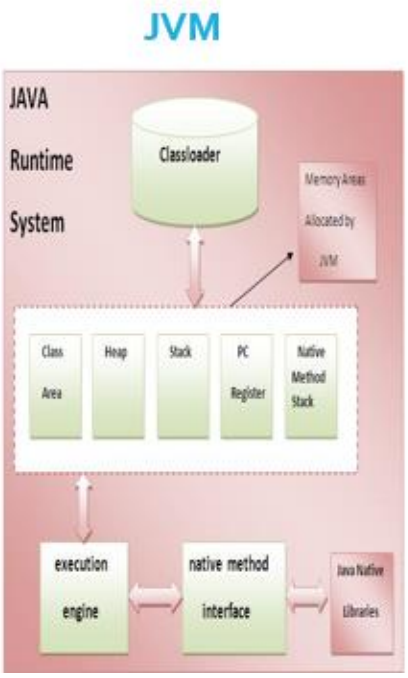


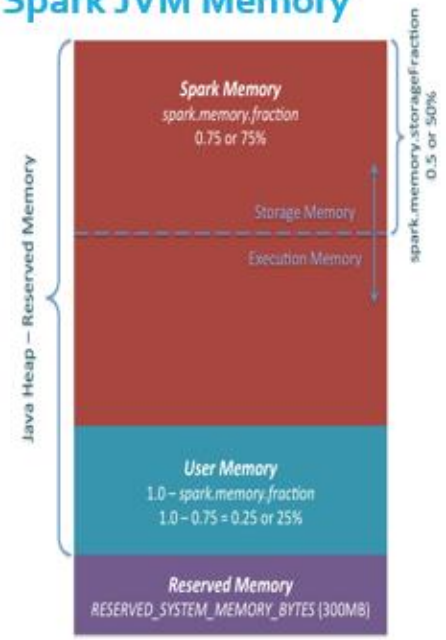
Compiler Directives





JVM – Memory Allocations





Spark Introduction

Spark is an in-memory cluster computing framework for processing and analyzing large amounts of data. It provides a simple programming interface, which enables an application developer to easily use the CPU, memory, and storage resources across a cluster of servers for processing large datasets.

Key Features

The key features of Spark include the following:

- Easy to use
- Fast
- General-purpose
- Scalable
- Fault tolerant

Spark Execution Model

Pillars of Spark

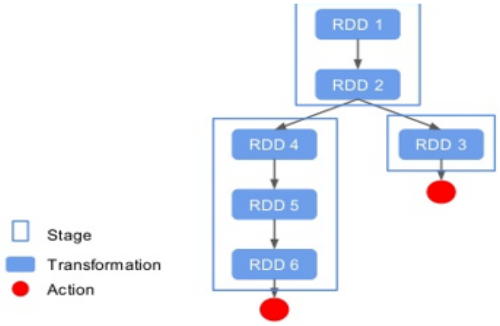
RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – stored in memory across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

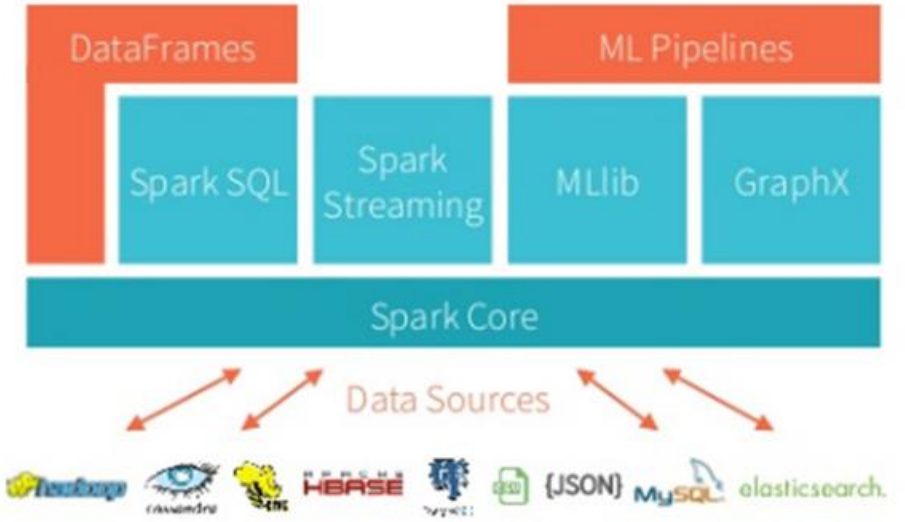
Pillars of Spark

Direct Acyclic Graph – sequence of computations performed on data

- *Node* – RDD partition
- *Edge* – transformation on top of data
- *Acyclic* – graph cannot return to the older partition
- *Direct* – transformation is an action that transitions data partition state (from A to B)



Spark Library Stack



Spark comes with a core data processing engine and a stack of libraries working on top of the core engine. It is very important to understand the concept of stacking libraries on top of the core framework. All these libraries that are making use of the services provided by the core framework support the data abstractions offered by the core framework and much more. Before Spark came onto market, there were lots of independent open source products doing what the library stack in discussion here is now doing. The biggest disadvantage with these point products was their interoperability. They don't stack together well. They were implemented in different programming languages. The programming language of choice supported by these products, and the lack of uniformity in the APIs exposed by these products, were really challenging to get one application done with two or more such products. That is the relevance of the stack of libraries that work on top of Spark. They all work together with the same programming model. This helps organizations to standardize on the data processing toolset without vendorlock-in. Spark comes with the following stack of domain-specific libraries, and the above figure gives a comprehensive picture of the whole ecosystem as seen by a developer

In any organization, structured data is still very widely used. The most ubiquitous data access mechanism with structured data is SQL. Spark SQL provides the capability to write SQL-like queries on top of the structured data abstraction called the DataFrame API. DataFrame and SQL go very well and support data coming from various sources, such as Hive, Avro, Parquet, JSON, and many more. Once the data is loaded into the Spark context, they can be operated as if they are all coming from the same source. In other words, if required, SQL-like queries can be used to join data coming from different sources, such as Hive and JSON. Another big advantage that Spark SQL and the DataFrame API bring onto the developers table is the ease of use and no need-to-know functional programming methods, which is a requirement to do programming with RDDs.

Spark Core

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed datasets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL

Spark SQL is Spark's package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics. This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0.

Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark Streaming

Spark Streaming is a Spark component that enables processing of live streams of data. Examples of data streams include logfiles generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time. Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.

MLlib

Spark comes with a library containing common machine learning (ML) functionality, called MLlib. MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.

GraphX

GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

Working with RDDs

The most important feature that Spark took from Scala is the ability to use functions as parameters to the Spark transformations and Spark actions. Quite often, the RDD in Spark behaves just like a collection object in Scala. Because of that, some of the data transformation method names of Scala collections are used in Spark RDD to do the same thing. This is a very neat approach and those who have expertise in Scala will find it very easy to program with RDDs. We will see a few important features in the following sections.

Spark RDD is immutable

There are some strong rules based on which an RDD is created. Once an RDD is created, intentionally or unintentionally, it cannot be changed. This gives another insight into the construction of an RDD. Because of that, when the nodes processing some part of an RDD die, the driver program can recreate those parts and assign the task of processing it to another node, ultimately, completing the data processing job successfully. Since the RDD is immutable, splitting a big one to smaller ones, distributing them to various worker nodes for processing, and finally compiling the results to produce the final result can be done safely without worrying about the underlying data getting changed.

Spark RDD is distributable

If Spark is run in a cluster mode where there are multiple worker nodes available to take the tasks, all these nodes will have different execution contexts. The individual tasks are distributed and run on different JVMs. All these activities of a big RDD getting divided into smaller chunks, getting distributed for processing to the worker nodes, and finally, assembling the results back, are completely hidden from the users. Spark has its own mechanism for recovering from the system faults and other forms of errors which occur during the data processing and hence this data abstraction is highly resilient.

Spark RDD lives in memory

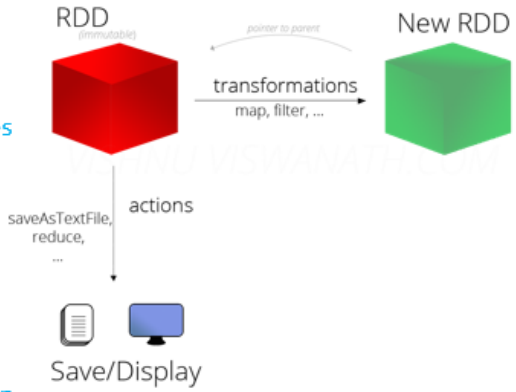
Spark does keep all the RDDs in the memory as much as it can. Only in rare situations, where Spark is running out of memory or if the data size is growing beyond the capacity, is it written to disk. Most of the processing on RDD happens in the memory, and that is the reason why Spark is able to process the data at a lightning fast speed.

Spark RDD is strongly typed

Spark RDD can be created using any supported data types. These data types can be Scala/Java supported intrinsic data types or custom created data types such as your own classes. The biggest advantage coming out of this design decision is the freedom from runtime errors. If it is going to break because of a data type issue, it will break during compile time.

All about RDDs

- Core Spark abstraction
- Lazily evaluated
- Partitions can be persisted in-memory or on-disk
- Broken up into partitions, which are distributed across nodes
- Abstraction to represent the large distributes
- Immutable, re-computable
- Fault tolerant – Via concept of Lineage
- Contains transformation history ("lineage") for data set



- Partitions – Set of data splits associated with this RDD
- Dependencies – List of parent RDDs involved in computation
- Compute – Function to compute partitions
- Preferred Locations – Place to put data
- Partitioner – How the data is split into partitions

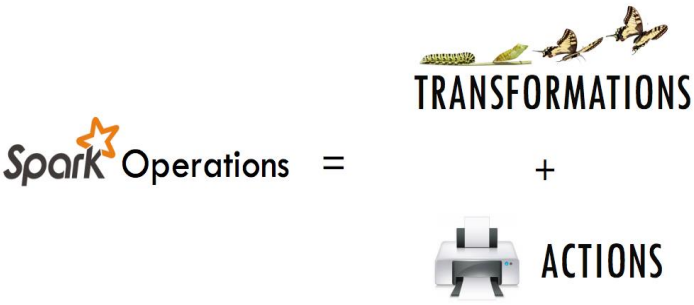
Creating RDDs

Three ways to create an RDD

- **From a file or set of files**
- **From data in memory**
- **From another RDD**

Types of RDDs

- **HadoopRDD**
- **FilteredRDD**
- **MappedRDD**
- **PairRDD**
- **ShuffledRDD**
- **UnionRDD**
- **JsonRDD**
- **SchemaRDD**
- **CassandraRDD**
- **EsSparkRDD**



RDD Operations

Spark applications process data using the methods defined in the RDD class or classes derived from it. These methods are also referred to as operations. Since Scala allows a method to be used with operator notation, the RDD methods are also sometimes referred to as *operators*. The beauty of Spark is that the same RDD methods can be used to process data ranging in size from a few bytes to several petabytes. In addition, a Spark application can use the same methods to process datasets stored on either a distributed storage system or a local file system. This flexibility allows a developer to develop, debug and test a Spark application on a single machine and deploy it on a large cluster without making any code change. RDD operations can be categorized into two types: transformation and action. A transformation creates a new RDD. An action returns a value to a driver program.

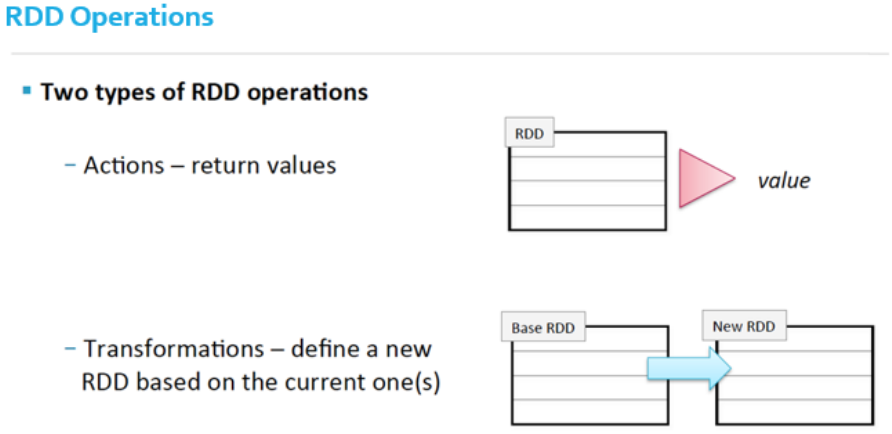
RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).

Transformations

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

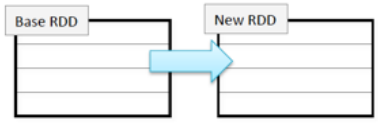
By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

A transformation method of an RDD creates a new RDD by performing a computation on the source RDD. This section discusses the commonly used RDD transformations. RDD transformations are conceptually similar to Scala collection methods. The key difference is that the Scala collection methods operate on data that can fit in the memory of a single machine, whereas RDD
Methods can operate on data distributed across a cluster of nodes. Another important difference is that RDD transformations are lazy, whereas Scala collection methods are strict. This topic is discussed in more detail later in this chapter.



RDD Operations - Transformations

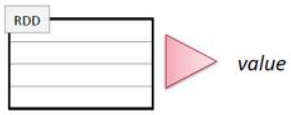
- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- **Some common transformations**
  - **`map(function)`** – creates a new RDD by performing a function on each record in the base RDD
  - **`filter(function)`** – creates a new RDD by including or excluding each record in the base RDD according to a boolean function



Actions are RDD methods that return a value to a driver program. This section discusses the commonly used RDD actions.

RDD Operations - Actions

- **Some common actions**
  - **`count()`** – return the number of elements
  - **`take(n)`** – return an array of the first *n* elements
  - **`collect()`** – return an array of all elements
  - **`saveAsTextFile(filename)`** – save to text file(s)




```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

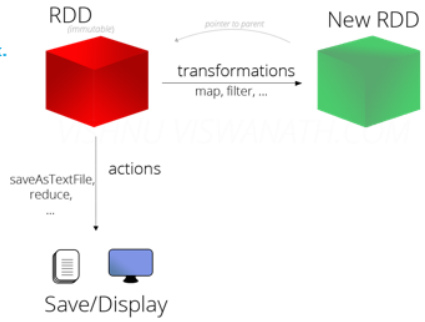
Lifecycle of a Spark Program

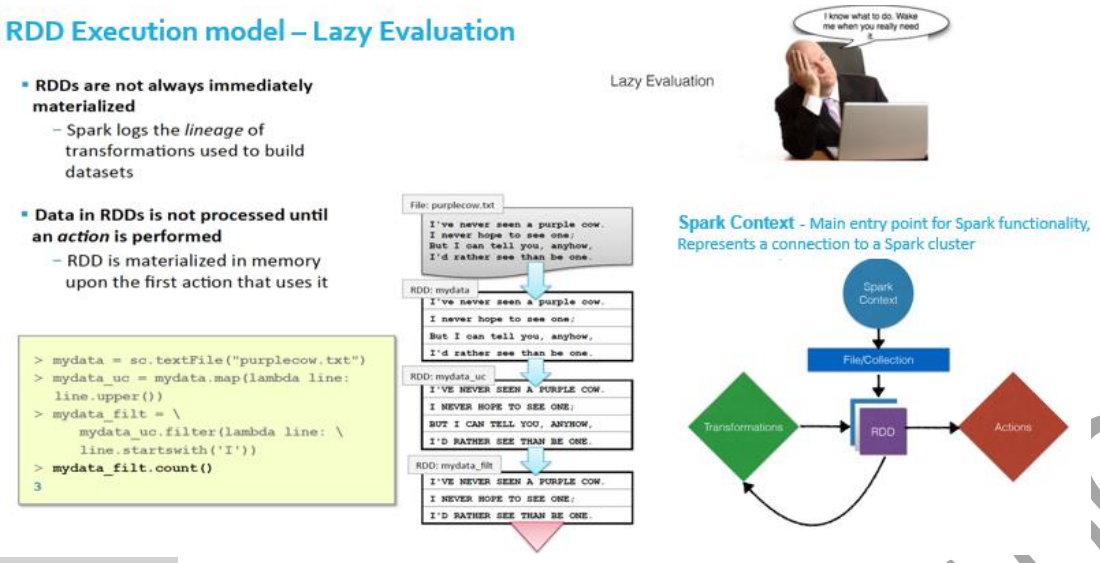
1)Create some input RDDs from external data or parallelize a collection in your driver program.

2)Lazily transform them to define new RDDs using transformations like filter()or map()

3)Ask Spark to cache()any intermediate RDDs that will need to be reused.

4)Launch actions such as count()and collect()to kick off a parallel computation, which is then optimized and executed by Spark.





Lazy Operations

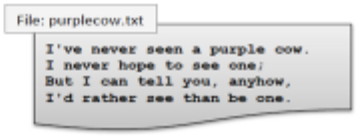
RDD creation and transformation methods are lazy operations. Spark does not immediately perform any computation when an application calls a method that return an RDD. For example, when you read a file from HDFS using textFile method of SparkContext, Spark does not immediately read the file from disk. Similarly, RDD transformations, which return a new RDD, are lazily computed. Spark just keeps track of transformations applied to an RDD.

Spark just makes a note of how an RDD was created and the transformations applied to it to create child RDDs. Thus, it maintains lineage information for each RDD. It uses this lineage information to construct or reconstruct an RDD when required.
If RDD creation and transformations are lazy operations, when does Spark actually read data and compute transformations? The next section answers this question.

Action Triggers Computation - RDD transformations are computed when an application calls an action method of an RDD or saves an RDD to a storage system. Saving an RDD to a storage system is considered as an action, even though it does not return a value to the driver program. When an application calls an RDD action method or saves an RDD, it triggers a chain reaction in Spark. At that point, Spark attempts to create the RDD whose action method was called. If that RDD was generated from a file, Spark reads that file into the memory of the worker nodes. If it is a child RDD created by a transformation of another RDD, then Spark attempts to first create the parent RDD. This process continues until Spark finds the root RDD. It then performs all the transformations required to generate the RDD whose action method was called. Finally, it performs the computations to generate the result that the action method returns to the driver program. Lazy transformations enable Spark to run RDD computations efficiently. By delaying computations
until an application needs the result of an action, Spark can optimize RDD operations. It pipelines operations and avoids unnecessary transfer of data over the network.

Lazy Execution (1)

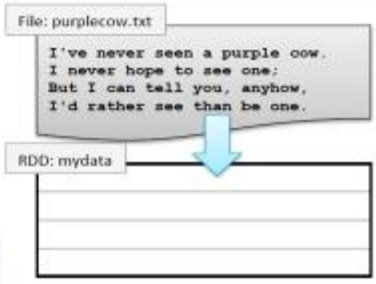
- **RDDs are not always immediately materialized**
  - Spark logs the *lineage* of transformations used to build datasets




```
>
```


Lazy Execution (2)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

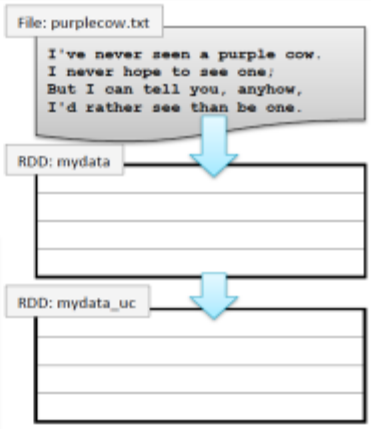



```
> mydata = sc.textFile("purplecow.txt")
```

Lazy Execution (3)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

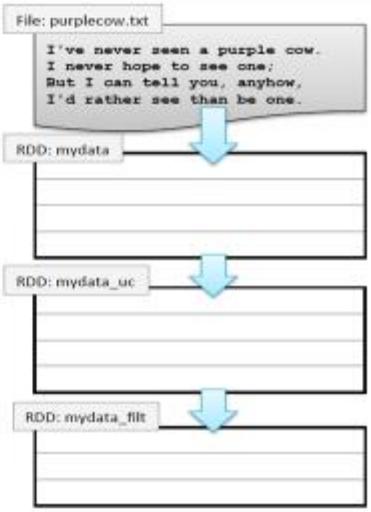
```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
```



Lazy Execution (4)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

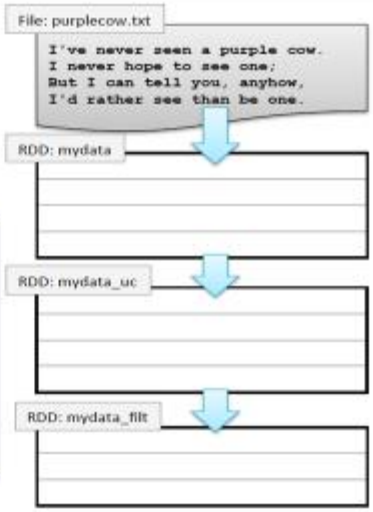
```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
```



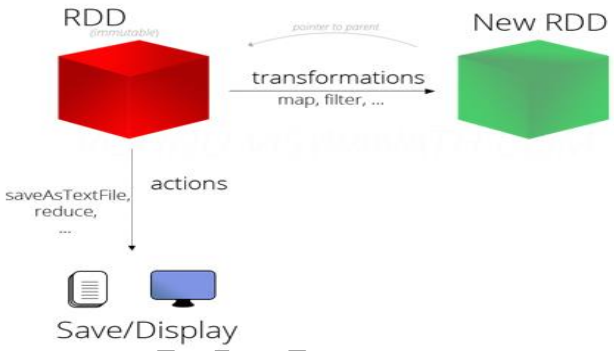
Lazy Execution (4)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
```



RDD Execution Model



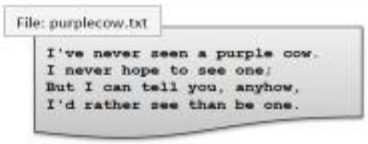
RDD Lineage

- When you call a transformation, Spark does not execute it immediately, instead it creates a **lineage**.
- A lineage keeps track of what all transformations has to be applied on that RDD, including from where it has to read the data.


```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter(line => line.contains("money"))
filtered.count()
```

Lineage Example (1)

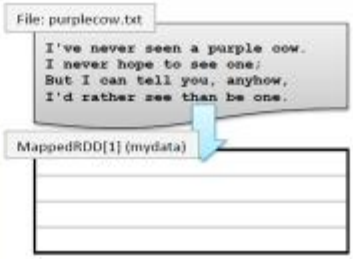
- **Each *transformation* operation creates a new *child* RDD**



Lineage Example (2)

- **Each *transformation* operation creates a new *child* RDD**

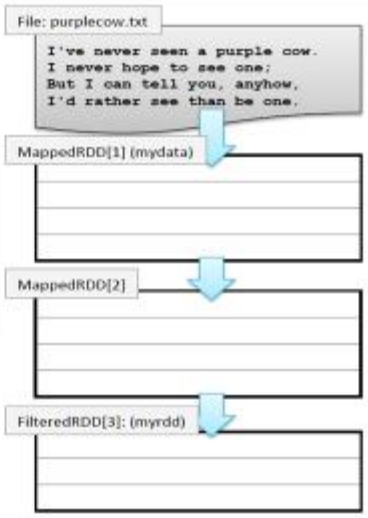
```
> mydata = sc.textFile("purplecow.txt")
```



Lineage Example (3)

- **Each *transformation* operation creates a new *child* RDD**

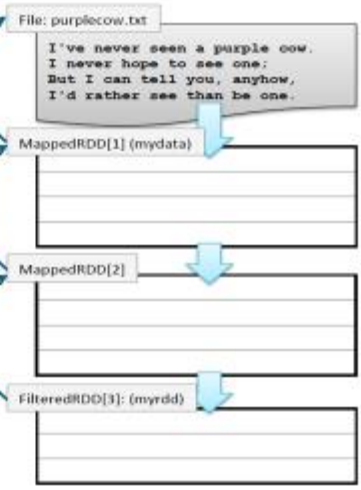
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example (4)

- **Spark keeps track of the *parent* RDD for each new RDD**
- **Child RDDs *depend on* their parents**

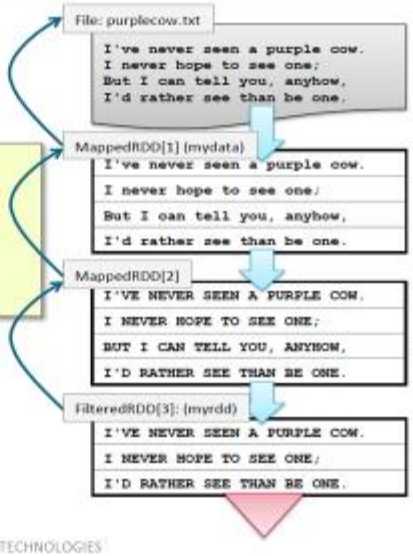
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example (5)

- ***Action*** **operations execute the parent transformations**

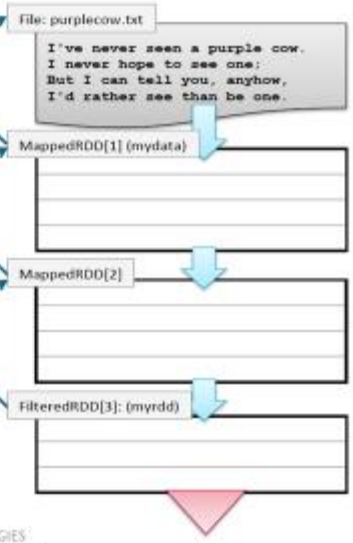
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
```



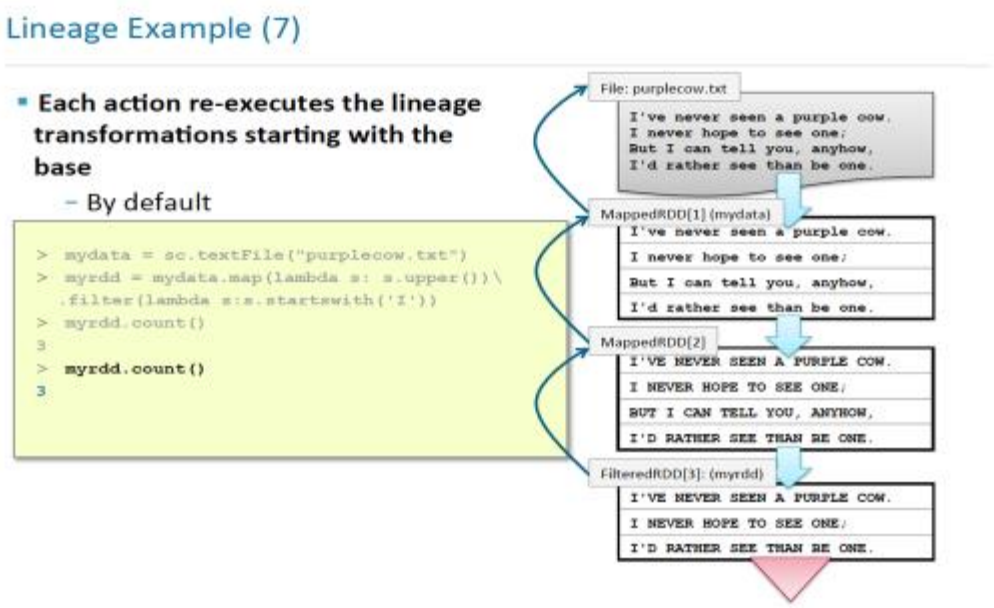
Lineage Example (6)

- **Each action re-executes the lineage transformations starting with the base**
  - By default

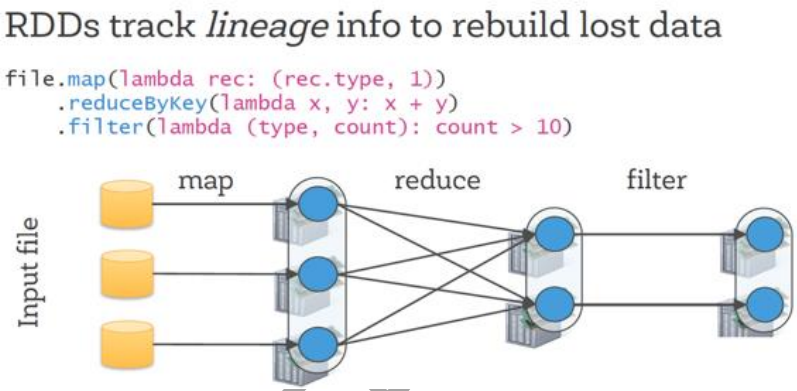
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
```



INCEPTEZ TECHNOLOGIES



Fault Tolerance



Fault tolerance is important in a distributed environment. Earlier you learned how Spark automatically moves a compute job to another node when a node fails. Spark's RDD caching mechanism is also fault tolerant. A Spark application will not crash if a node with cached RDD partitions fails. Spark automatically recreates and caches the partitions stored on the failed node on another node. Spark uses RDD lineage information to re-compute lost cached partitions.

Loading data in RDD

- **For file-based RDDS, use `SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- **Files are referenced by absolute or relative URI**
  - Absolute URI: `file:/home/training/myfile.txt`
  - Relative URI (uses default file system): `myfile.txt`

Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat. Text file RDDs can be created using SparkContext's textFile method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines.

```
scala> val distFile = sc.textFile("data.txt")
distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26
```

Once created, distFile can be acted on by dataset operations. For example, we can add up the sizes of all the lines using the map and reduceoperations as follows: distFile.map(s => s.length).reduce((a, b) => a + b).

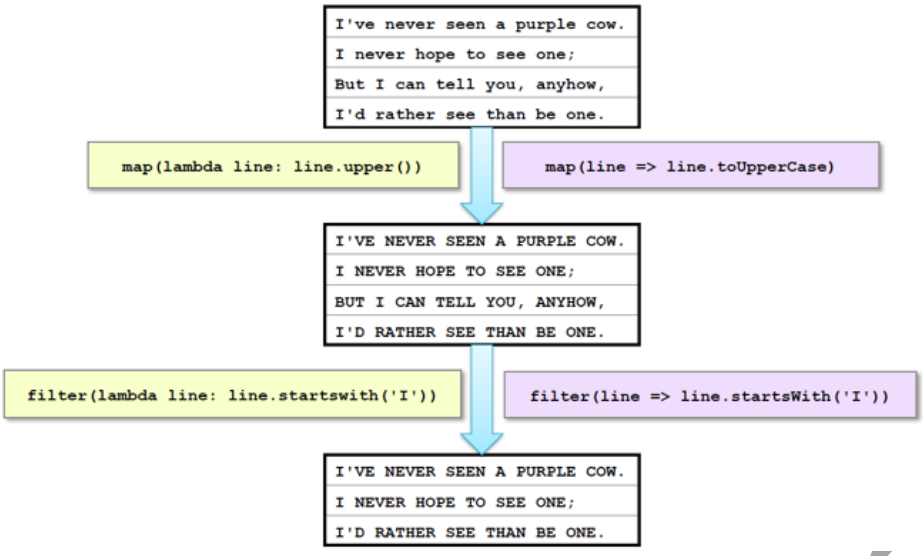
Some notes on reading files with Spark:

- If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system.
- All of Spark's file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/*.txt"), and textFile("/my/directory/*.gz").
- The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks.

Apart from text files, Spark's Scala API also supports several other data formats:

- SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file.
- For SequenceFiles, use SparkContext's sequenceFile[K, V] method where K and V are the types of key and values in the file. These should be subclasses of Hadoop's Writable interface, like IntWritable and Text. In addition, Spark allows you to specify native types for a few common Writables; for example, sequenceFile[Int, String] will automatically read IntWritables and Texts.
- For other Hadoop InputFormats, you can use the SparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source. You can also use SparkContext.newAPIHadoopRDD for InputFormats based on the "new" MapReduce API (org.apache.hadoop.mapreduce).
- RDD.saveAsObjectFile and SparkContext.objectFile support saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD.

Example: `map` and `filter` Transformations



RDD Operations

= easy = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General	Math / Statistical	Set Theory / Relational	Data Structure / I/O
• map • filter • flatMap • mapPartitions • mapPartitionsWithIndex • groupBy • sortBy	• sample • randomSplit	• union • intersection • subtract • distinct • cartesian • zip	• keyBy • zipWithIndex • zipWithUniqueID • zipPartitions • coalesce • repartition • repartitionAndSortWithinPartitions • pipe

ACTIONS

• reduce • collect • aggregate • fold • first • take • forEach • top • treeAggregate • treeReduce • forEachPartition • collectAsMap	• count • takeSample • max • min • sum • histogram • mean • variance • stdev • sampleVariance • countApprox • countApproxDistinct	• takeOrdered	• saveAsTextFile • saveAsSequenceFile • saveAsObjectFile • saveAsHadoopDataset • saveAsHadoopFile • saveAsNewAPIHadoopDataset • saveAsNewAPIHadoopFile

A Sample Application

```
val sc = new SparkContext(...)

val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

SparkContext

Resilient distributed datasets (RDDs)

Transformation

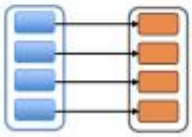
Action

Types of Transformations

vs

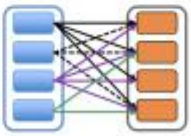
narrow

each partition of the parent RDD is used by at most one partition of the child RDD

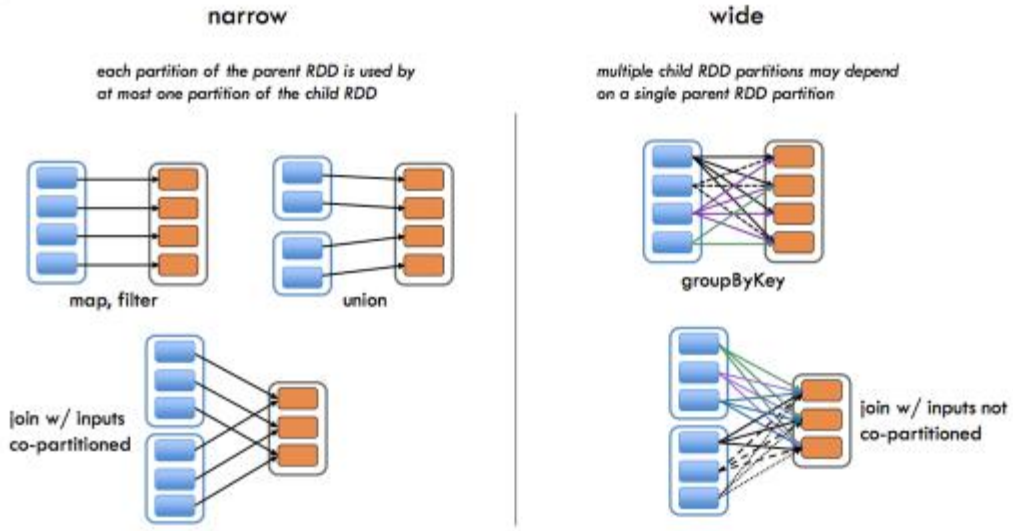


wide

multiple child RDD partitions may depend on a single parent RDD partition



Types of Transformations



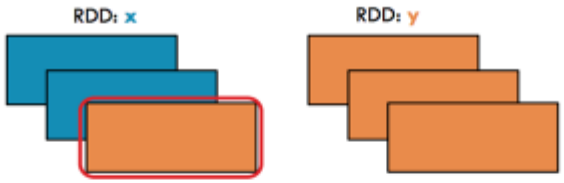
Spark Essentials : Transformations

transformation	description
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials : Transformations

transformation	description
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

MAP



map(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each element of this RDD

```python
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

```
x: ['b', 'a', 'c']
y: [('b', 1), ('a', 1), ('c', 1)]
```

```scala
val x = sc.parallelize(Array("b", "a", "c"))
val y = x.map(z => (z,1))
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```


INCEPTEZ TECHNOLOGIES

FILTER

After `filter()` has been applied...

RDD: x

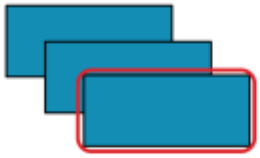
before

RDD: y

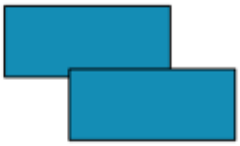
after

FILTER

RDD: x



RDD: y



`filter(f)`

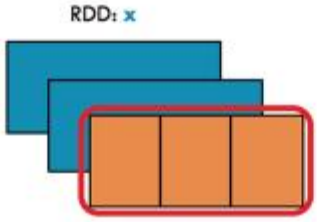
Return a new RDD containing only the elements that satisfy a predicate

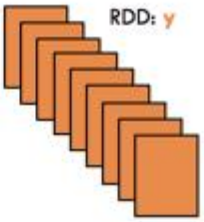
```python
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3]
y: [1, 3]
```

```scala
val x = sc.parallelize(Array(1,2,3))
val y = x.filter(n => n%2 == 1)
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```

FLATMAP





`flatMap(f, preservesPartitioning=False)`

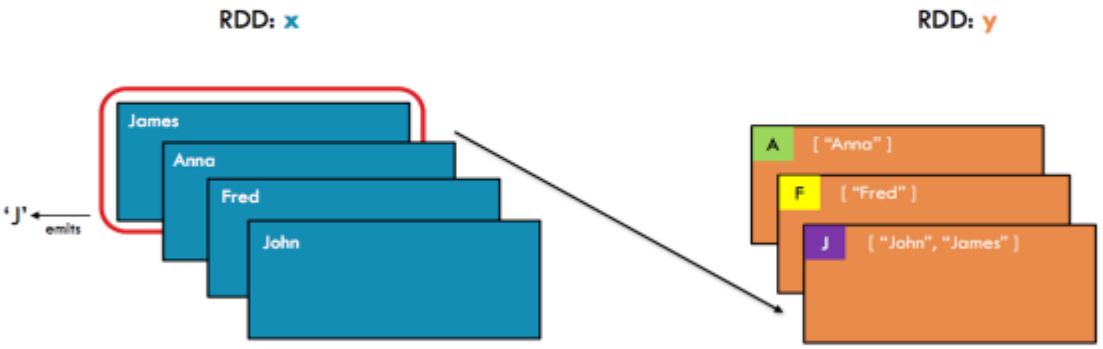
Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

```python
x = sc.parallelize([1,2,3])
y = x.flatMap(lambda x: (x, x*100, 42))
print(x.collect())
print(y.collect())
```

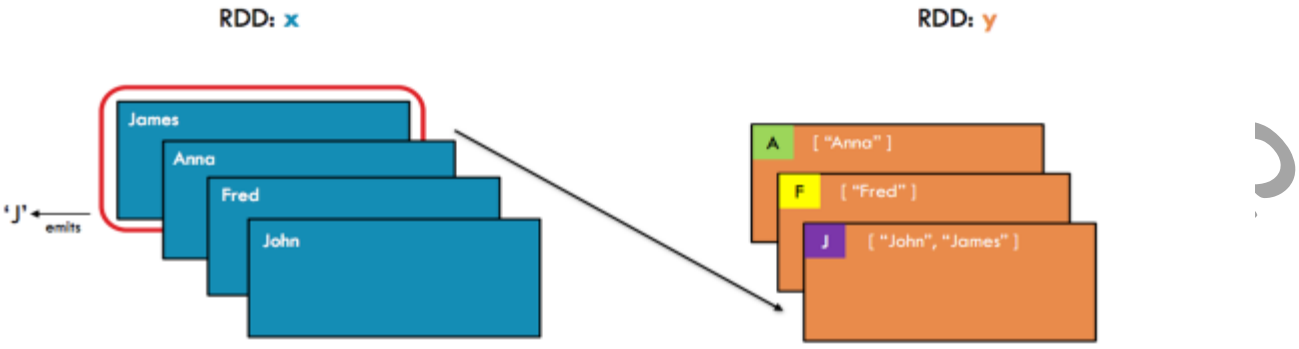
```
x: [1, 2, 3]
y: [1, 100, 42, 2, 200, 42, 3, 300, 42
```

```scala
val x = sc.parallelize(Array(1,2,3))
val y = x.flatMap(n => Array(n, n*100, 42))
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```

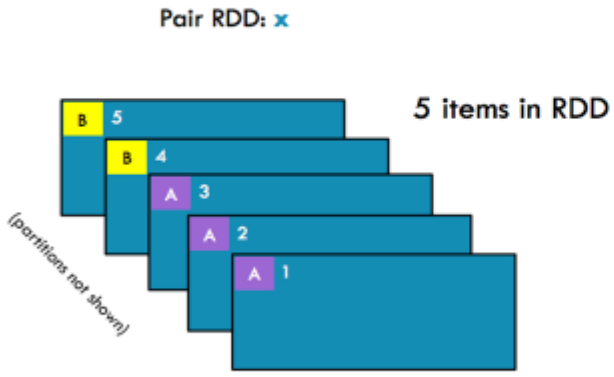

GROUPBY



GROUPBY



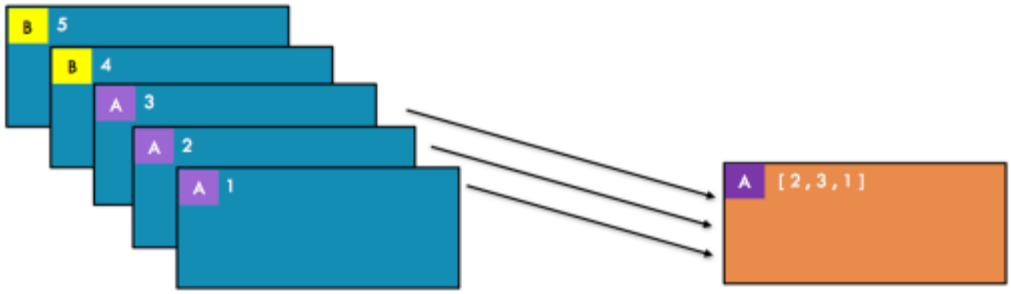
GROUPBYKEY



GROUPBYKEY

Pair RDD: x

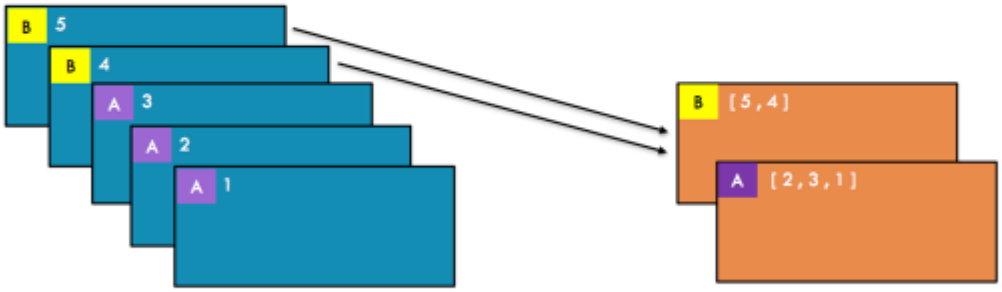
RDD: y



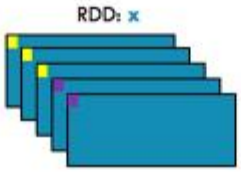
GROUPBYKEY

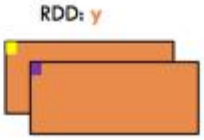
Pair RDD: x

RDD: y



GROUPBYKEY





groupByKey(*numPartitions=None*)

Group the values for each key in the original RDD. Create a new pair where the original key corresponds to this collected group of values.

```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print(list((j[0], list(j[1])) for j in y.collect()))
```

```
x: [('B', 5),('B', 4),('A', 3),('A', 2),('A', 1)]

y: [('A', [2, 3, 1]),('B',[5, 4])]
```

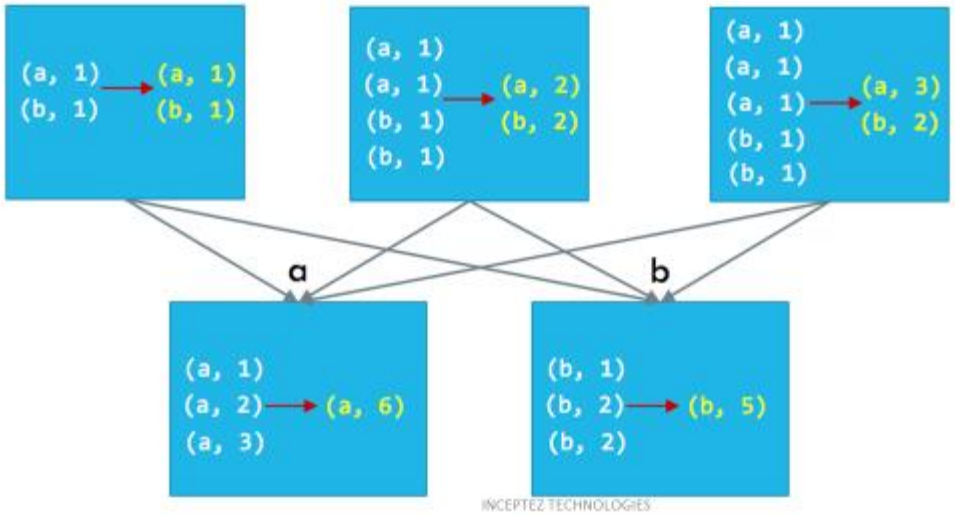
```
val x = sc.parallelize(
        Array(('B',5),('B',4),('A',3),('A',2),('A',1)))
val y = x.groupByKey()
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```


INCEPTEZ TECHNOLOGIES

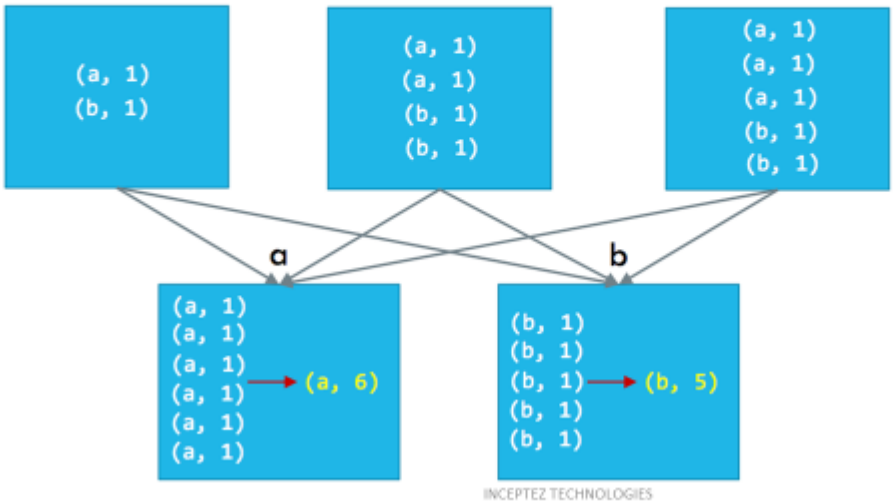
REDUCEBYKEY VS GROUPBYKEY

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

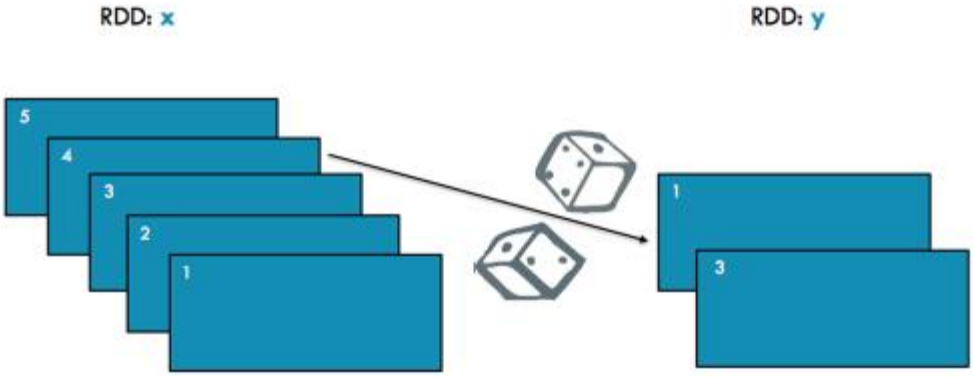
REDUCEBYKEY



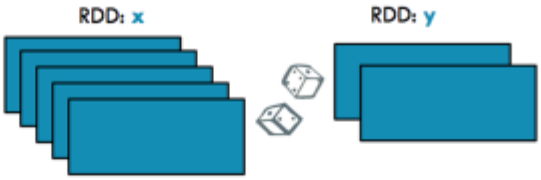
GROUPBYKEY



SAMPLE



SAMPLE



sample(*withReplacement, fraction, seed=None*)

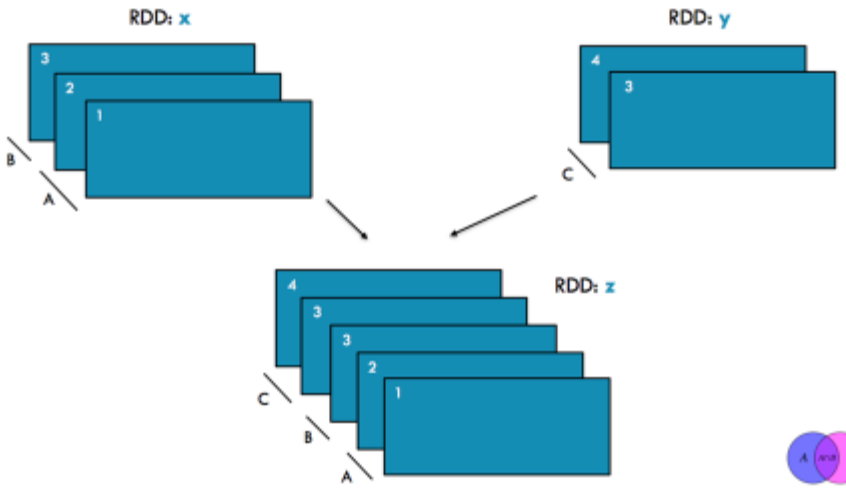
Return a new RDD containing a statistical sample of the original RDD

```
x = sc.parallelize([1, 2, 3, 4, 5])
y = x.sample(False, 0.4, 42)
print(x.collect())
print(y.collect())
```

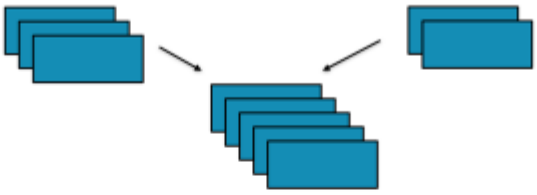
```
val x = sc.parallelize(Array(1, 2, 3, 4, 5))
val y = x.sample(false, 0.4)

// omitting seed will yield different output
println(y.collect().mkString(", "))
```

```
x: [1, 2, 3, 4, 5]

y: [1, 3]
```

UNION



UNION



Return a new RDD containing all items from two original RDDs. Duplicates are *not* culled.

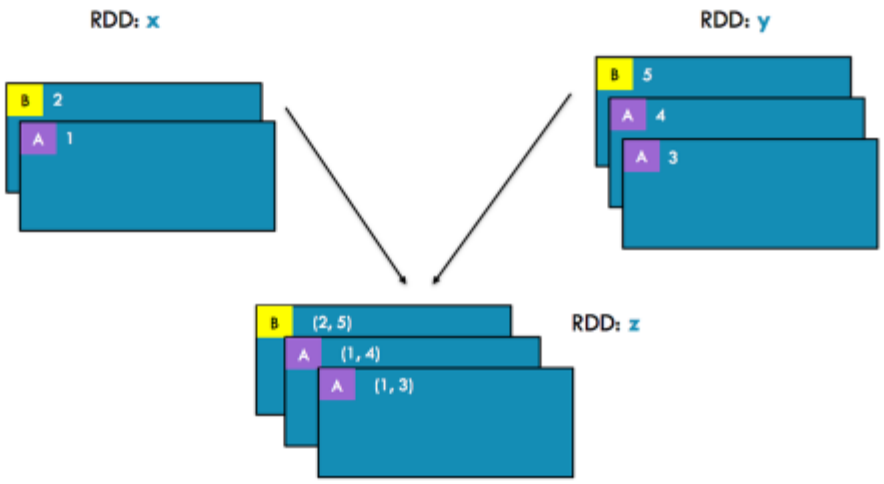
union(*otherRDD*)

```python
x = sc.parallelize([1,2,3], 2)
y = sc.parallelize([3,4], 1)
z = x.union(y)
print(z.glom().collect())
```

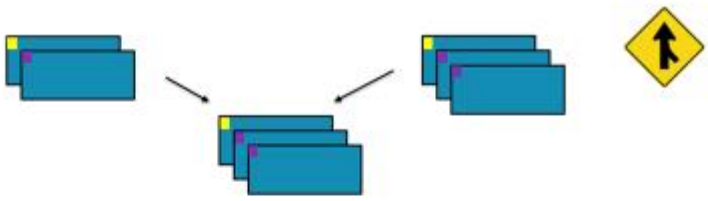
```
x: [1, 2, 3]
y: [3, 4]
z: [[1], [2, 3], [3, 4]]
```

```scala
val x = sc.parallelize(Array(1,2,3), 2)
val y = sc.parallelize(Array(3,4), 1)
val z = x.union(y)
val zOut = z.glom().collect()
```

JOIN



JOIN



Return a new RDD containing all pairs of elements having the same key in the original RDDs

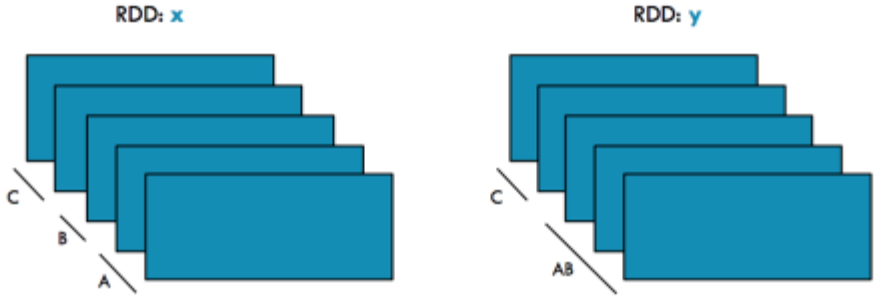
union(*otherRDD, numPartitions=None*)

```
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
print(z.collect())
```

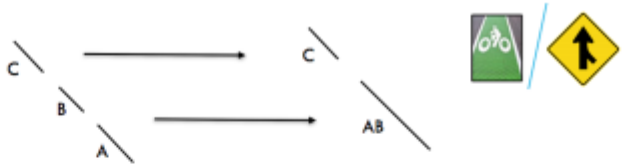
```
val x = sc.parallelize(Array(("a", 1), ("b", 2)))
val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))
val z = x.join(y)
println(z.collect().mkString(", "))
```

```
x: [("a", 1), ("b", 2)]

y: [("a", 3), ("a", 4), ("b", 5)]

z: [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]
```

COALESCE



COALESCE



Return a new RDD which is reduced to a smaller number of partitions

```
coalesce(numPartitions, shuffle=False)
```

```python
x = sc.parallelize([1, 2, 3, 4, 5], 3)
y = x.coalesce(2)
print(x.glom().collect())
print(y.glom().collect())
```

```scala
val x = sc.parallelize(Array(1, 2, 3, 4, 5), 3)
val y = x.coalesce(2)
val xOut = x.glom().collect()
val yOut = y.glom().collect()
```

```
x: [[1], [2, 3], [4, 5]]
y: [[1], [2, 3, 4, 5]]
```

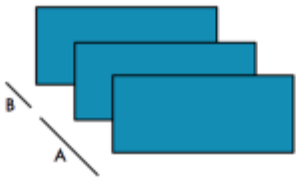
Actions

action	description
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Actions

action	description
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

COLLECT



collect()

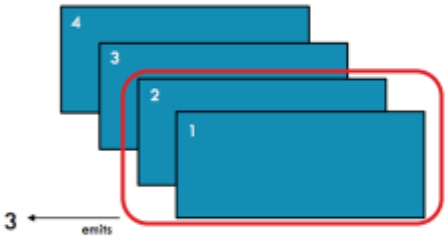
Return all items in the RDD to the driver in a single list

```python
x = sc.parallelize([1,2,3], 2)
y = x.collect()

print(x.glom().collect())
print(y)
```

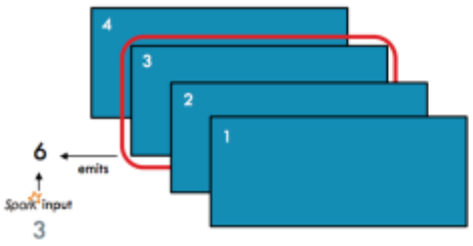
```
x: [[1], [2, 3]]
y: [1, 2, 3]
```

```scala
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.collect()

val xOut = x.glom().collect()
println(y)
```

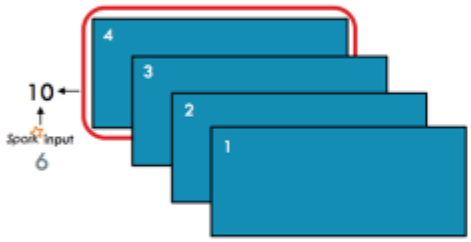

REDUCE

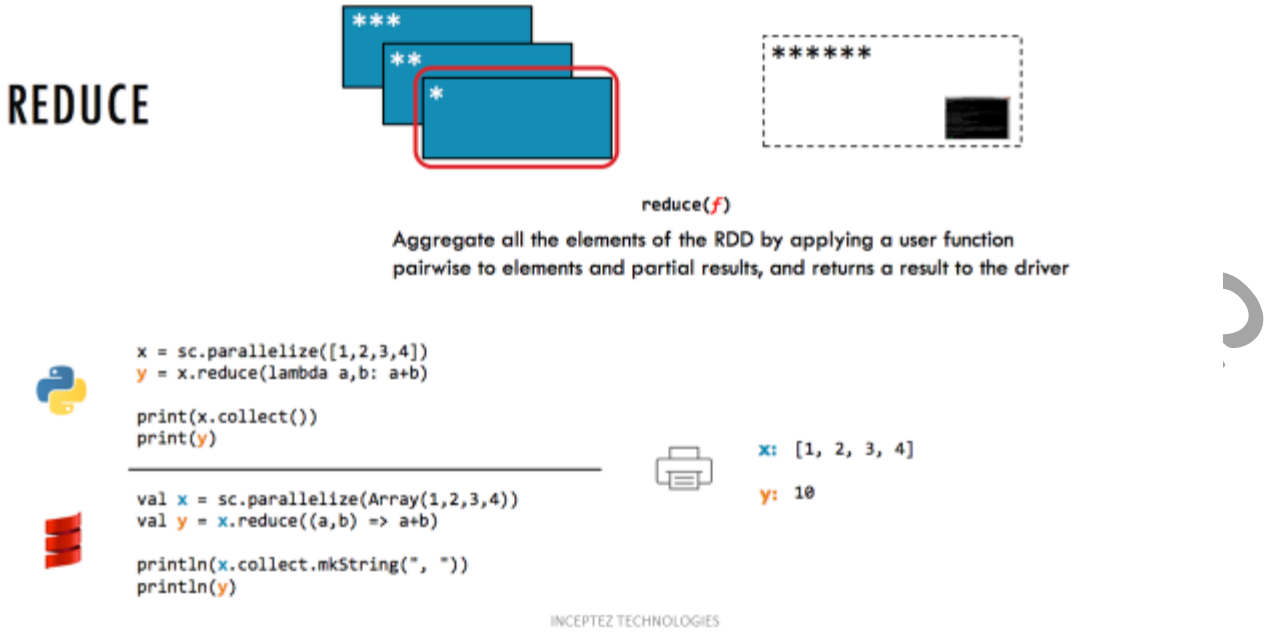


REDUCE



REDUCE





Spark Essentials

Spark uses a cluster manager to acquire cluster resources for executing a job. A cluster manager, as the name implies, manages computing resources across a cluster of worker nodes. It provides low-level scheduling of cluster resources across applications. It enables multiple applications to share cluster resources and run on the same worker nodes. Spark currently supports three cluster managers: standalone, Mesos, and YARN. Mesos and YARN allow you to run Spark and Hadoop applications simultaneously on the same worker nodes.

Cluster Resource Managers

Spark Standalone (AMPLab, Spark default)
- Suitable for a lot of production workloads
- Only Spark workloads

YARN (Hadoop)
- Allows hierarchies of resources
- Kerberos integration
- Multiple workloads from different execution frameworks
- Hive, Pig, Spark, MapReduce, etc.,

Mesos (AMPLab)
- Similar to YARN, but allows elastic allocation to disparate execution frameworks
- Coarse-grained - Single, long-running Mesos tasks runs Spark mini tasks
- Fine-grained - New Mesos task for each Spark task, Higher overhead, not good for long-running Streaming Spark jobs

Hardware Requirements

CPU Cores

Spark scales well to tens of CPU cores per machine because it performs minimal sharing between threads. You should likely provision at least 8-16 cores per machine.

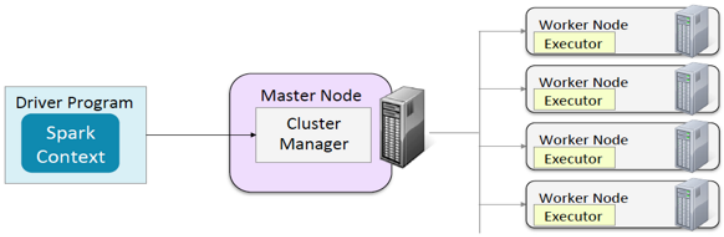
Memory
Spark runs well with anywhere from **8 GB to hundreds of gigabytes of memory per machine**. In all cases, we recommend allocating only at **most 75% of the memory for Spark**; leave the rest for the operating system and buffer cache.

Local Disks
While Spark can perform a lot of its computation in memory, it still uses local disks to store data that doesn't fit in RAM, as well as to preserve intermediate output between stages. We recommend having **4-8 disks per node, configured without RAID**

Network
When the data is in memory, a lot of Spark applications are network-bound. Using a **10 Gigabit** or higher network is the best way to make these applications faster. This is especially true for "distributed reduce" applications such as group-by, reduce-by, and SQL joins.

Spark Terminologies

- **A Spark Driver**
  - The "main" program
    - Either the Spark Shell or a Spark application
  - Creates a Spark Context configured for the cluster
  - Communicates with Cluster Manager to distribute tasks to executors



SparkContext

First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically

Other programs must use a constructor to instantiate a new `SparkContext`

Then in turn `SparkContext` gets used to create other variables

Scala:

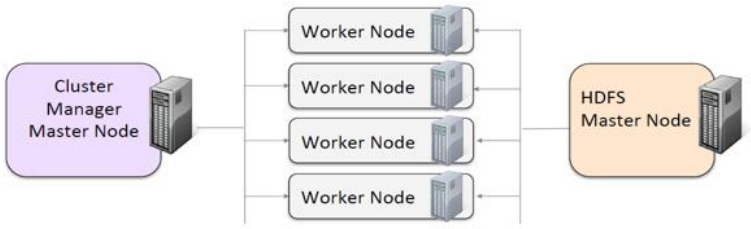
```
scala> sc
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc
<pyspark.context.SparkContext object at 0x7f7570783350>
```

Cluster Terminologies

- **A cluster is a group of computers working together**
  - Usually runs HDFS in addition to Spark Standalone, YARN, or Mesos
- **A node is an individual computer in the cluster**
  - *Master* nodes manage distribution of work and data to *worker* nodes
- **A daemon is a program running on a node**
  - Each performs different functions in the cluster



Programming Terminologies

- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
  - A driver program is an application that uses Spark as a library. It provides the data processing code that Spark executes on the worker nodes. A driver program can launch one or more jobs on a Spark cluster.
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)
- **Deploy Mode**
  - **cluster** : Driver inside the cluster
  - **client** : Driver outside of Cluster
- **Worker Node :** Node that run the application program in cluster
  - **A worker provides CPU, memory, and storage resources to a Spark application. The workers run a Spark application as distributed processes on a cluster of nodes.**
- **Executor**
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
  - Cache Memory & Swap storage for RDD lineage
  - An executor is a JVM (Java virtual machine) process that Spark creates on each worker for an application. It executes application code concurrently in multiple threads. It can also cache data in memory or disk.
  - An executor has the same lifespan as the application for which it is created. When a Spark application terminates, all the executors created for it also terminate.
- **Job**
- Consists multiple tasks, Created based on an Action
- **Stage :** Each Job is divided into a smaller set of tasks called Stages that is sequential and depend on each other
- **Task :** A unit of work that will be sent to executor.
  - A task is the smallest unit of work that Spark sends to an executor. It is executed by a thread in an executor on a worker node. Each task performs some computations to either return a result to a driver program or partition its output for shuffle.
  - Spark creates a task per data partition. An executor runs one or more tasks concurrently. The amount of parallelism is determined by the number of partitions. More partitions mean more tasks processing data in parallel.
- **Partitions:** Data unit that will be handled parallel, Same as Blocks in HDFS.

Running Spark on Cluster

The Scala and Spark shells are both command-line REPL (read-evaluate-print-loop) tools. A REPL enables interactive programming. It allows you to type an expression and have it immediately evaluated. It reads an expression, evaluates it, prints the result on the console, and waits for the next expression.

The Spark shell creates and makes an instance of the SparkContext class available as sc. The Scala prompt indicates that the Spark shell is ready to evaluate any expression you type. Since the Spark shell is based on the Scala shell, you can enter any Scala expression after the Scala> prompt.