COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

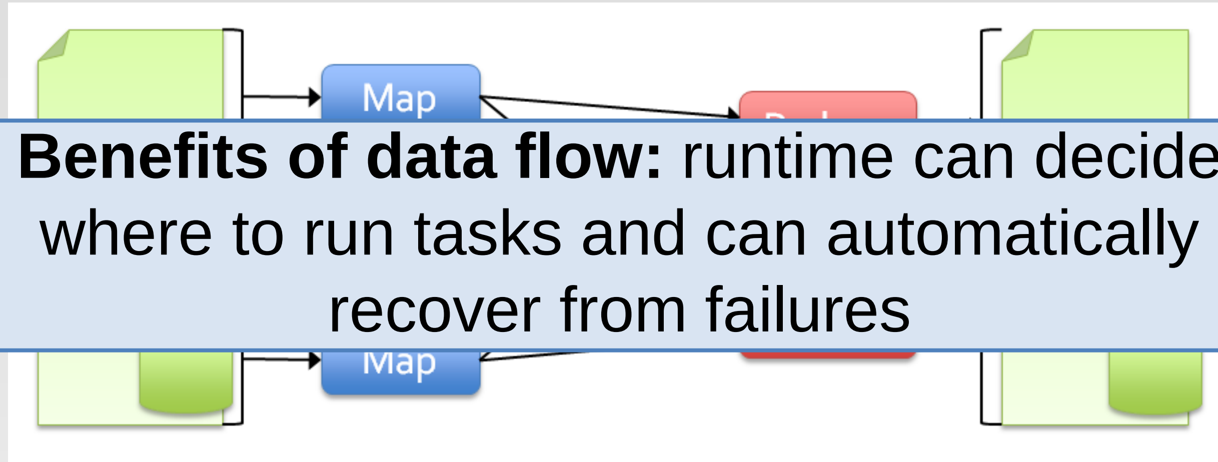
Chapter 4.1: Spark I

Part 1: Spark Introduction

Limitations of MapReduce

- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at one-pass computation.
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster **data sharing** across parallel jobs
  - One reaction: specialized models for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

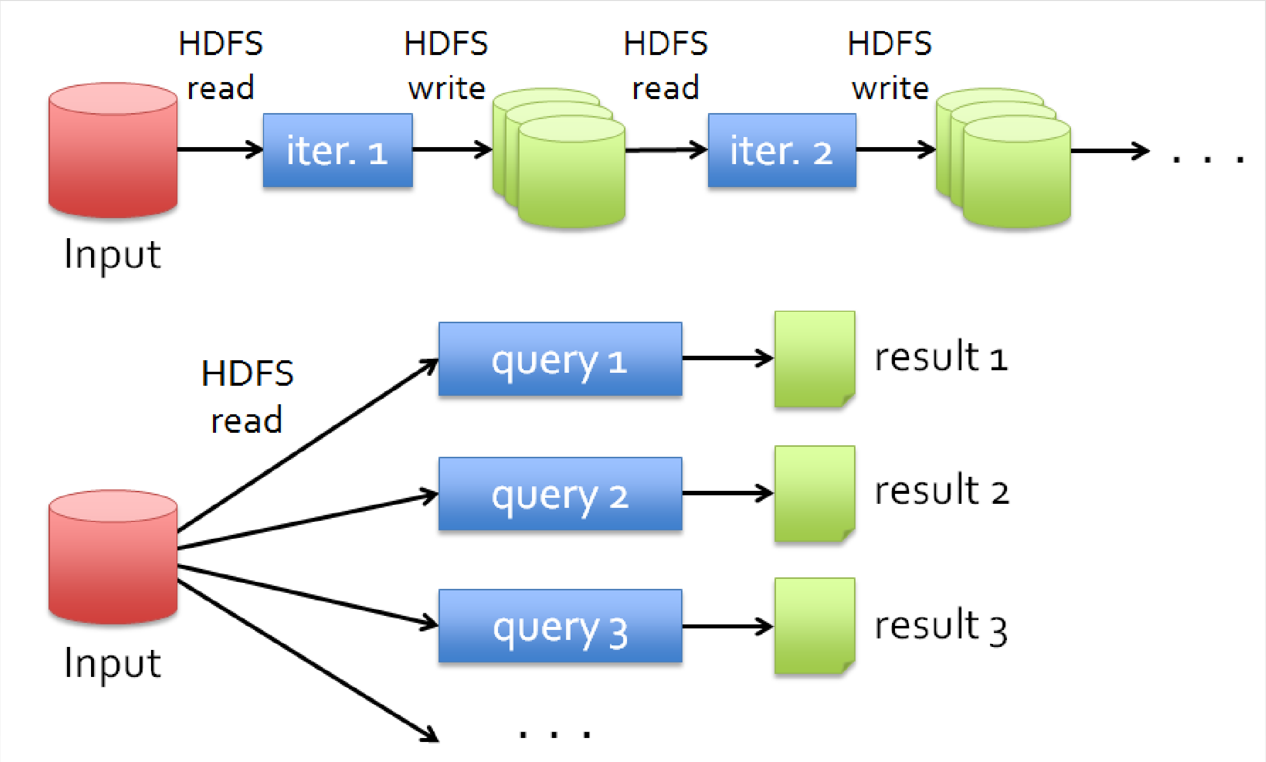
Limitations of MapReduce



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability
  - Not easy to write MapReduce programs using Java

Data Sharing in MapReduce



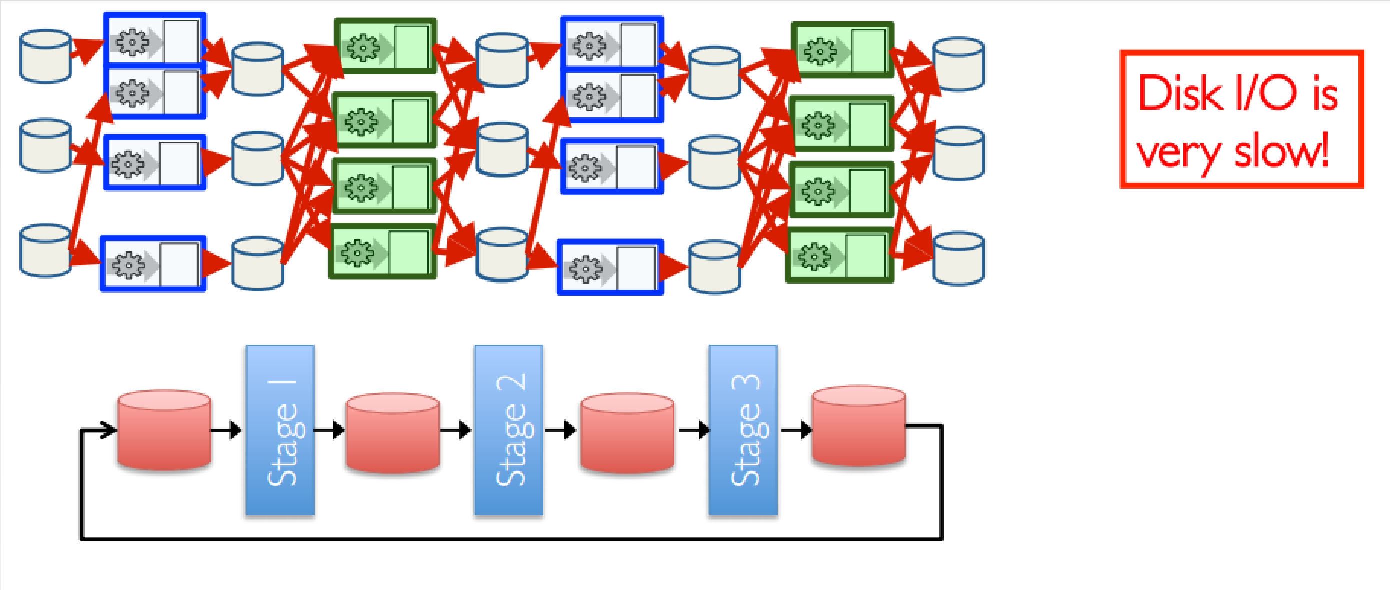
Slow due to replication, serialization, and disk IO

- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

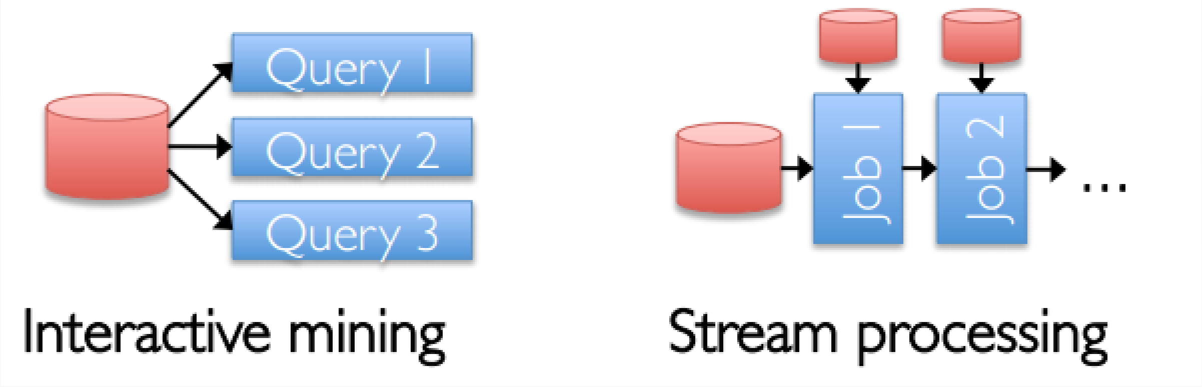
Efficient primitives for **data sharing**

Data Sharing in MapReduce

- Iterative jobs involve a lot of disk I/O for each repetition



- Interactive queries and online processing involves lots of disk I/O



Hardware for Big Data

Lots of hard drives

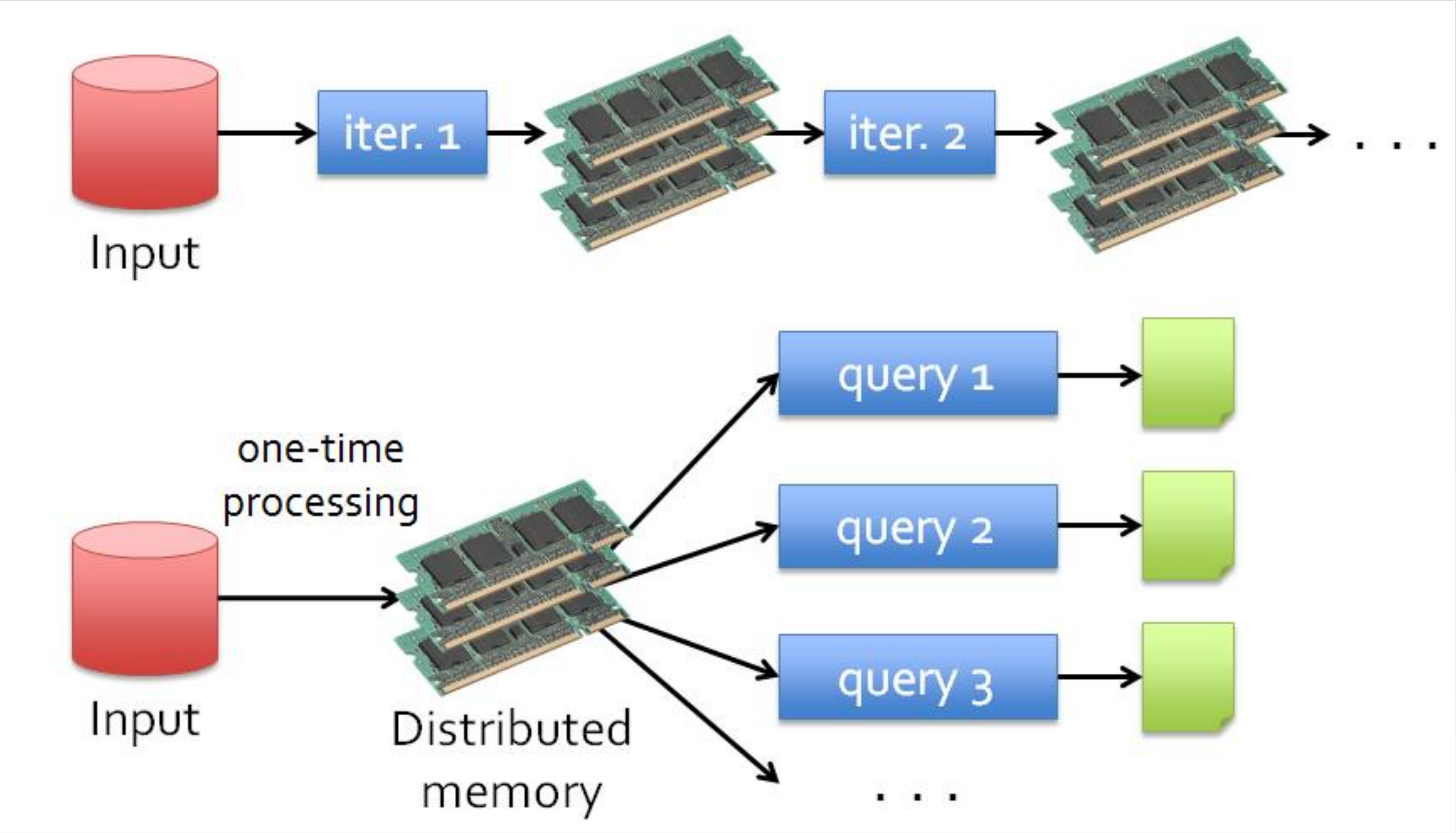
Lots of CPUs

And lots of memory!

Goals of Spark

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

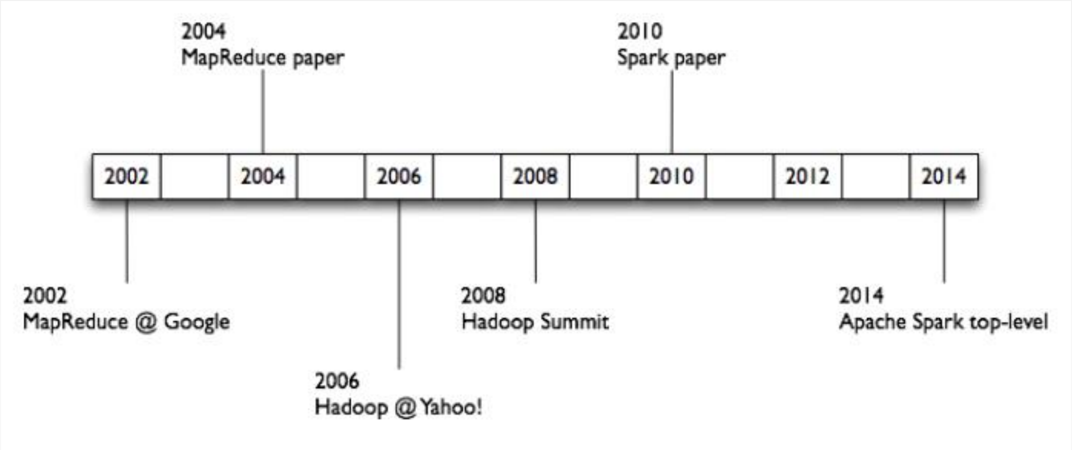
Data Sharing in Spark Using RDD



10-100× faster than network and disk

What is Spark

- One popular answer to “What’s beyond MapReduce?”
- Open-source engine for large-scale distributed data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java, Python, and R
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks



What is Spark

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - **In-memory** computing primitives
  - General computation graphs

  → Up to 100 × faster (10 × on disk)
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell

  → Often 5 × less code

What is Spark

- **Spark is not**
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management
  - Spark uses Hadoop for storage purpose only

- Spark's design philosophy centers around four key characteristics:
  - Speed
  - Ease of use
  - Modularity
  - Extensibility

Speed

- Its internal implementation benefits immensely from the performance improvement of CPUs and memory.
  - The framework is optimized to take advantage of memory, multiple cores, and the underlying Unix-based operating system

- Spark builds its query computations as a directed acyclic graph
  - Tasks can execute in parallel across workers on the cluster

- It has a physical execution engine which generates compact code for execution

Ease of Use

- Spark achieves simplicity by providing a fundamental abstraction of a simple logical data structure called a Resilient Distributed Dataset (RDD)

- Since Spark 2.x, DataFrames and Datasets APIs have been developed upon RDD

- By providing a set of ***transformations*** and ***actions*** as operations, Spark offers a simple programming model that you can use to build big data applications in familiar languages.

Modularity

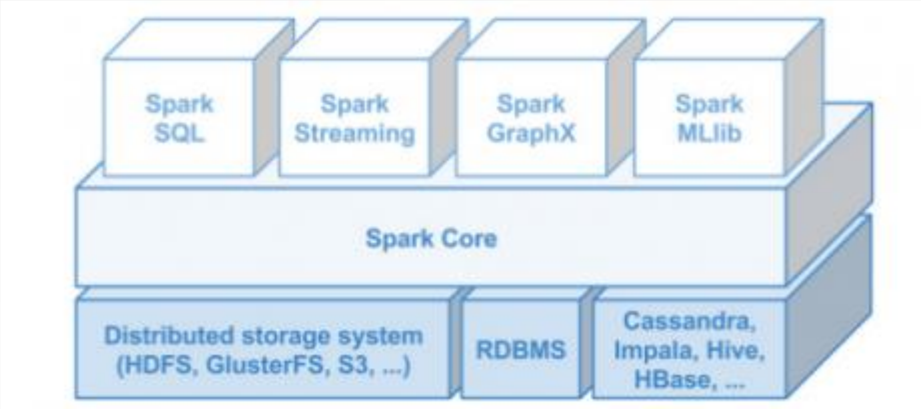
- Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R.

- Spark offers unified libraries with well-documented APIs that include the following modules as core components: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX, combining all the workloads running under one engine.

- You can write a single Spark application that can do it all—no need for distinct engines for disparate workloads, no need to learn separate APIs.

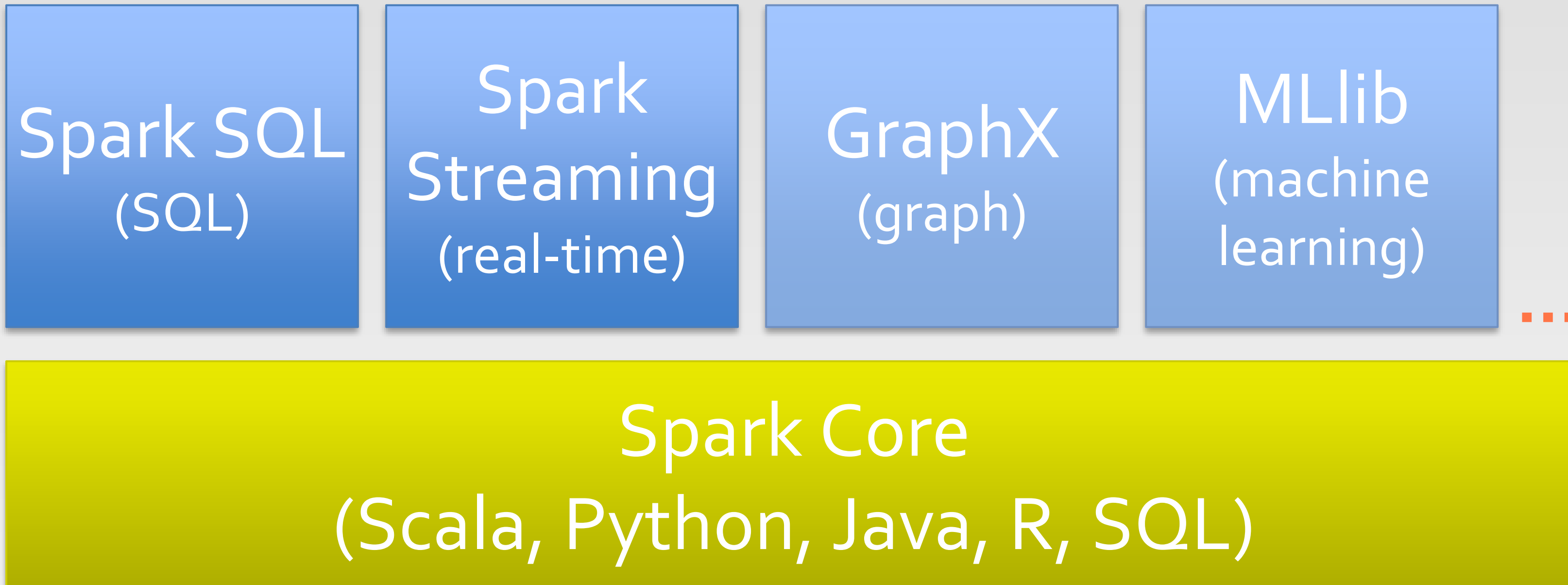
Extensibility

- Spark focuses on its fast, parallel computation engine rather than on storage.
  - You can use Spark to read data stored in myriad sources—local file systems, Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMSs, and more—and process it all in memory.
- Spark's DataFrameReaders and DataFrameWriters can also be extended to read data from other sources, such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3



What is Spark

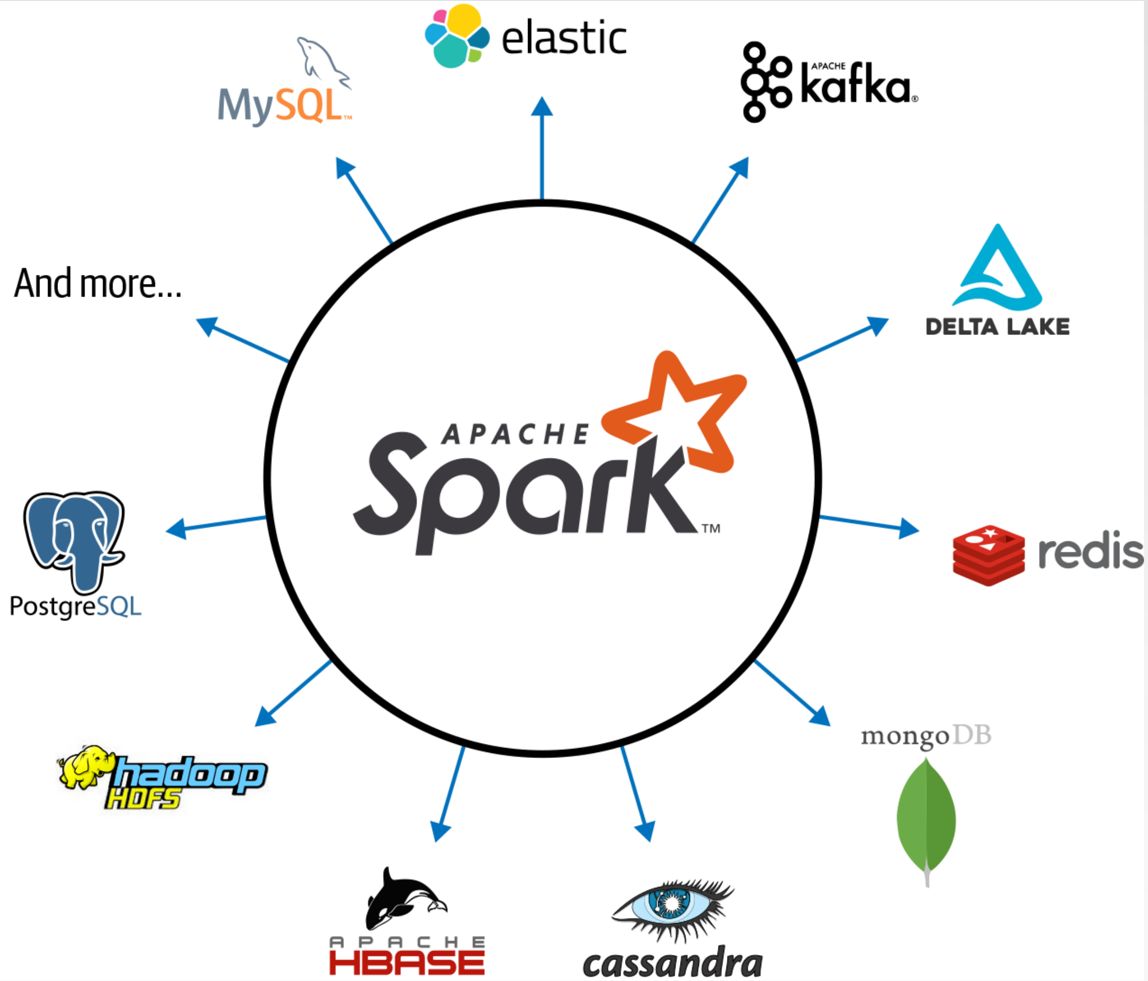
- Spark is the basis of a wide set of projects in the Berkeley Data Analytics Stack (BDAS)



  - Spark SQL (SQL on Spark)
  - Spark Streaming (stream processing)
  - GraphX (graph processing)
  - MLlib (machine learning library)

Spark's Ecosystem of Connectors

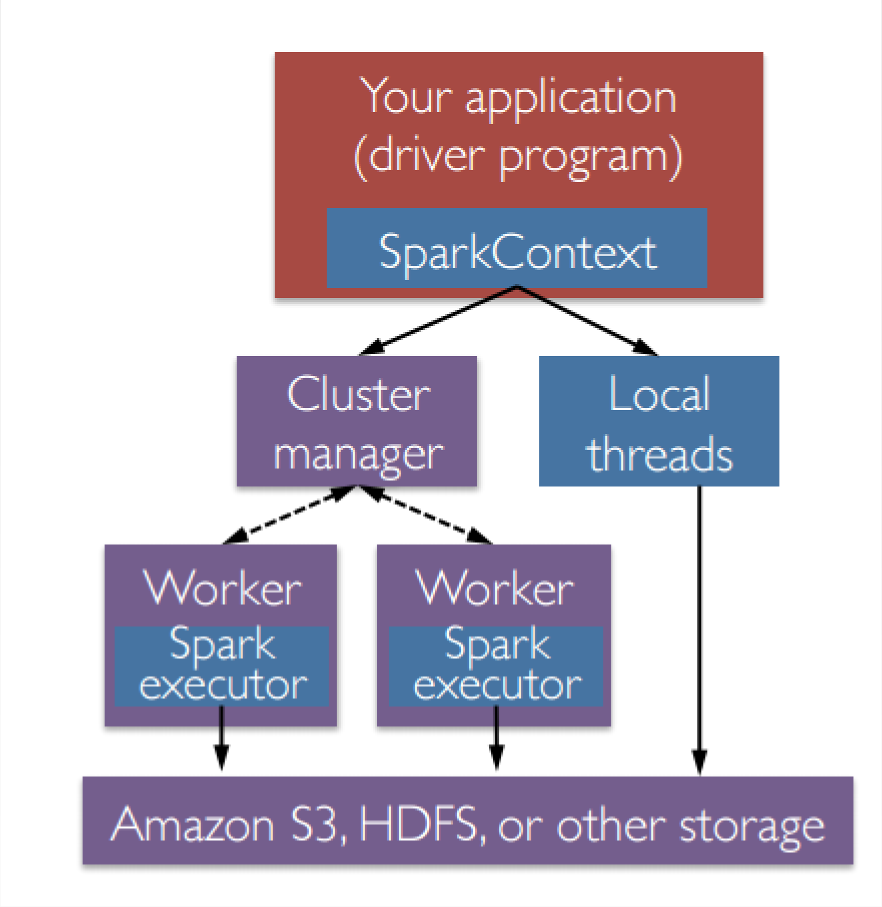
- The community of Spark developers maintains a list of third-party Spark packages as part of the growing ecosystem



Spark Ideas

- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system on top of Hadoop.
- Achieve fault-tolerance by re-execution instead of replication

Spark Workflow



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Define RDDs
  - Connect to several types of cluster managers (e.g., YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Download and Configure Spark

- Current version: 3.5.0. https://spark.apache.org/downloads.html
  - You also need to install Java first

Download Apache Spark™

1. Choose a Spark release: 3.5.0 (Sep 13 2023)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.5.0-bin-hadoop3.tgz
4. Verify this release using the 3.5.0 signatures, checksums and project release KEYS by following these procedures.

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

- After downloading the package, unpack it and then configure the path variable in file ~/.bashrc

```
export SPARK_HOME=/home/comp9313/spark
export PATH=$SPARK_HOME/bin:$PATH
```

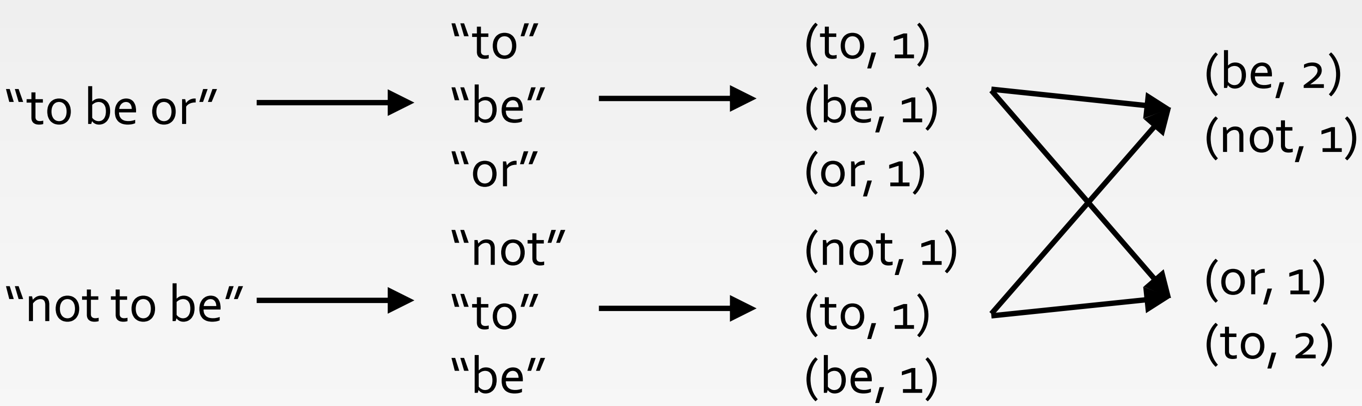
Spark Shell

- Spark comes with four widely used interpreters that act like interactive “shells” and enable ad hoc data analysis: pyspark, spark-shell, sparksql, and sparkR

```
comp9313@comp9313-VirtualBox:~$ spark-shell
2021-10-06 21:25:47,310 WARN util.Utils: Your hostname, comp9313-VirtualBox resolves to a loopback address: 127.0.1.1;
 using 10.0.2.15 instead (on interface enp0s3)
2021-10-06 21:25:47,311 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/home/comp9313/spark/jars/spark-unsafe_2.
12-3.1.2.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
2021-10-06 21:25:47,979 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using bu
iltin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-1633515956456).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 11.0.11)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Word Count in Spark

```
textfile = sc.textFile("hdfs://...", 4)

words = textfile.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b: a + b)
count.collect()
```



Python Supports Functional Programming

- First-Class Functions. Functions are treated like objects:
  - passing functions as arguments to other functions
  - returning functions as the values from other functions
  - assigning functions to variables or storing them in data structures

```
>>> plusOne = lambda x: x+1
>>> type(plusOne)
<class 'function'>
>>> plusOne(5)
6
```

```
>>> def f(plusOne):
...     return plusOne(5)
...
>>> f(plusOne)
6
```

```
>>> def f(x):
...     return lambda y:y+x
...
>>> f(5)(2)
7
```

Closures

- Closures: a function whose return value depends on the value of one or more variables declared outside this function.

```
// plusFoo can reference any values/variables in scope
foo = 1
plusFoo = lambda x: x+foo

plusFoo(5)   → 6

// Changing foo changes the return value of plusFoo
foo = 5
plusFoo(5)   → 10
```

Higher Order Functions

- Higher Order Functions
  - A function that does at least one of the following:
    - takes one or more functions as arguments
    - returns a function as its result

```
>>> def f(plusOne):
...     return plusOne(5)
...
>>> f(plusOne)
6
```

```
>>> def f(x):
...     return lambda y:y+x
...
>>> f(5)(2)
7
```

More Examples on Higher Order Functions

```
basefunc = lambda x: (lambda y: x + y)
// interpreted by:
   basefunc(x):
       sumfunc(y):
              return x+y
       return sumfunc

closure1 = basefunc(1)        closure1(5) = ?
                                          6
closure2 = basefunc(4)        closure2(5) = ?
                                          9
```

- basefunc returns a function, and closure1 and closure2 are of function type.
- While closure1 and closure2 refer to the same function basefunc, the associated environments differ, and the results are different

Part 2: RDD Introduction

RDD: Resilient Distributed Datasets

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**
  - Data residing on multiple nodes in a cluster.
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel.

RDD: Resilient Distributed Datasets

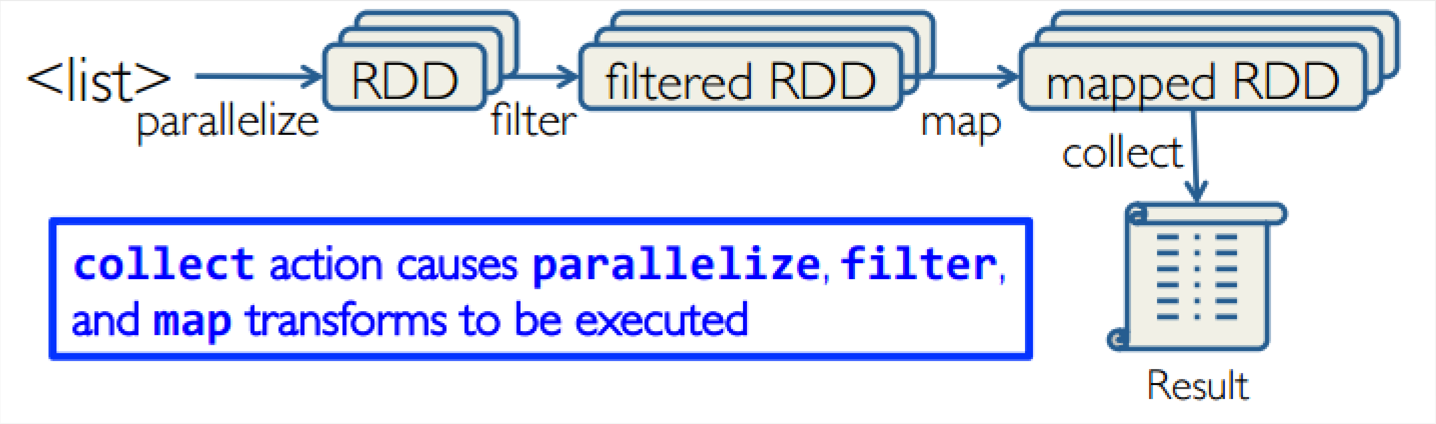
- *Resilient Distributed Datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory across cluster
  - Manipulated through parallel operators
  - Automatically recomputed on failure based on lineage
- RDDs can express many parallel algorithms, and capture many current programming models
  - Data flow models: MapReduce, SQL, …
  - Specialized models for iterative apps: Pregel, …

RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing collections (lists or arrays)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - Persistence
  - Partitioning

Creating RDDs

- From HDFS, text files, Amazon S3, Apache HBase, SequenceFiles, any other Hadoop InputFormat
- Creating an RDD from a File
  - inputfile = sc.textFile("...", 4)
    - RDD distributed in 4 partitions
    - Elements are lines of input
    - Lazy evaluation means no execution happens now

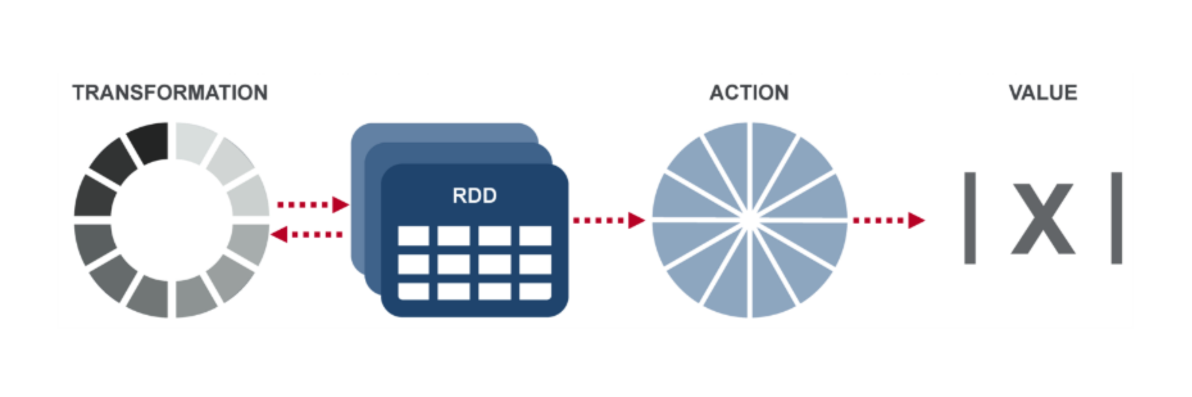
```
>>> inputfile = sc.textFile("file:///home/comp9313/pg100.txt")
>>> inputfile
file:///home/comp9313/pg100.txt MapPartitionsRDD[1] at textFile at NativeMethodA
ccessorImpl.java:0
```

- Turn a collection into an RDD
  - sc.parallelize([1, 2, 3]), creating from a Python list

Repartition and Coalesce

- Sometimes we may need to repartition the RDD, PySpark provides two ways to repartition:
  - repartition(): shuffles data from all nodes, also called full shuffle
  - coalesce(): shuffle data from minimum nodes. For example, if you have data in 4 partitions, doing coalesce(2) moves data from just 2 nodes.
  - Both of the functions take the number of partitions to repartition rdd.
  - Note that repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.
  - repartition() is used to increase or decrease the RDD partitions whereas coalesce() is used to **only** decrease the number of partitions in an efficient way.

RDD Operations



- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, join, etc.*
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

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
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

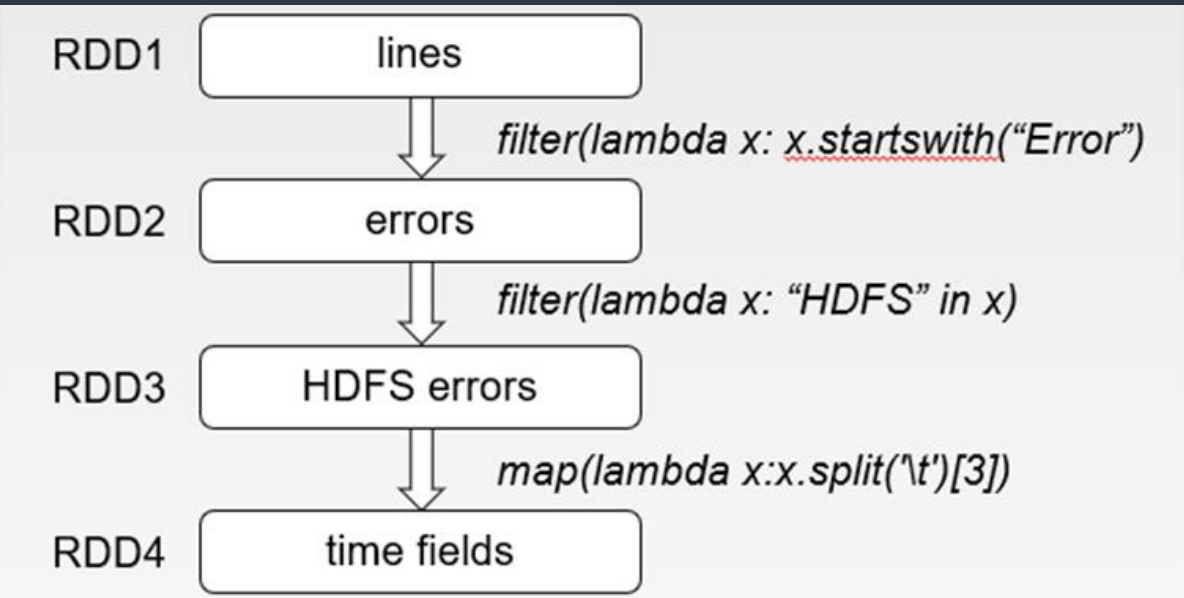
Action	Description
`reduce(`*func*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*n*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*n*`, `*key=func*`)`	return n elements ordered in ascending order or as specified by the optional key function

- Example: words.collect().foreach(println)

Example

- Web service is experiencing errors and an operators want to search terabytes of logs in the Hadoop file system to find the cause.

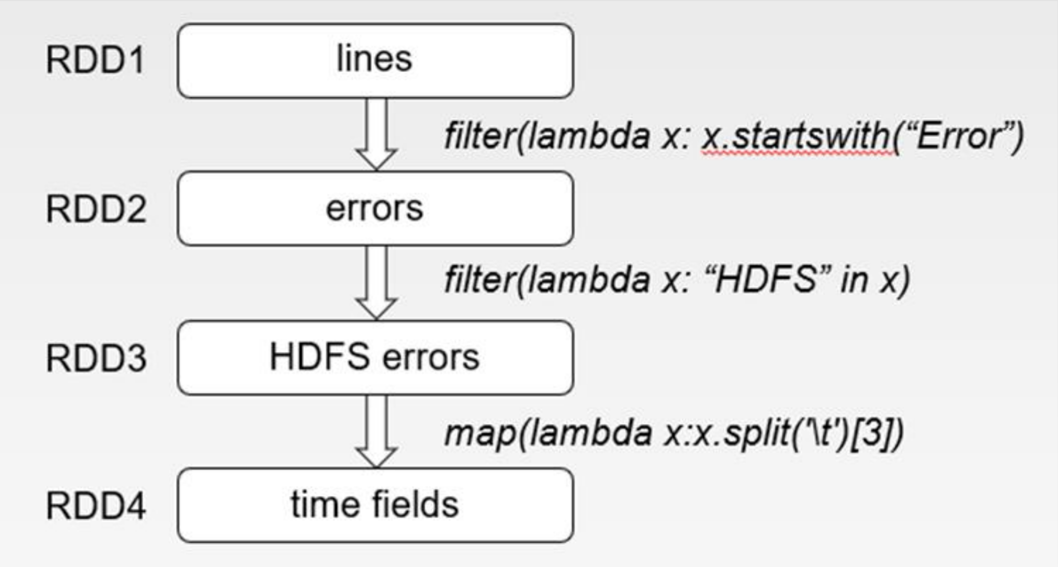
```
lines = sc.textFile("hdfs://…") //base RDD, obtained from a file on HDFS
errors = lines.filter(lambda x: x.startswith("Error")) //get messages that start
errors.persist() //persist the data in memory
errors.count()
errors.filter(lambda x: "HDFS" in x).map(lambda x:x.split('\t')[3]).collect()
```



- Line1: RDD backed by an HDFS file (base RDD lines not loaded in memory)
- Line3: Asks for errors to persist in memory (errors are in RAM)

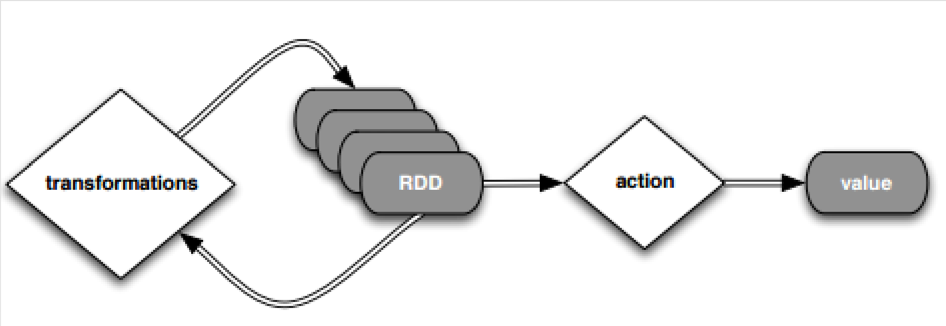
Lineage Graph

- RDDs keep track of lineage
- RDD has enough information about how it was derived from to compute its partitions from data in stable storage.



- Example:
  - If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines.
  - Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program.

Deconstructed

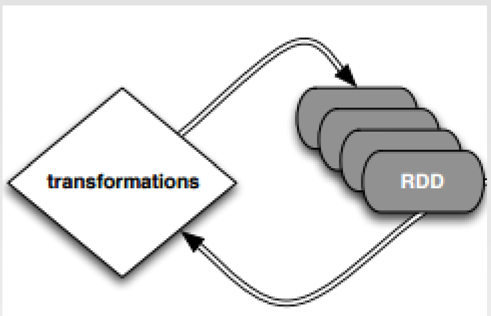



```
//base RDD
lines = sc.textFile("hdfs://...")
//Transformed RDD
errors = lines.filter(lambda x: x.startswith("Error"))
errors.persist()
errors.count()
errors.filter(lambda x: "HDFS" in x).
        map(lambda x: x.split('\t')[3]).
        collect()
```

Deconstructed

//base RDD

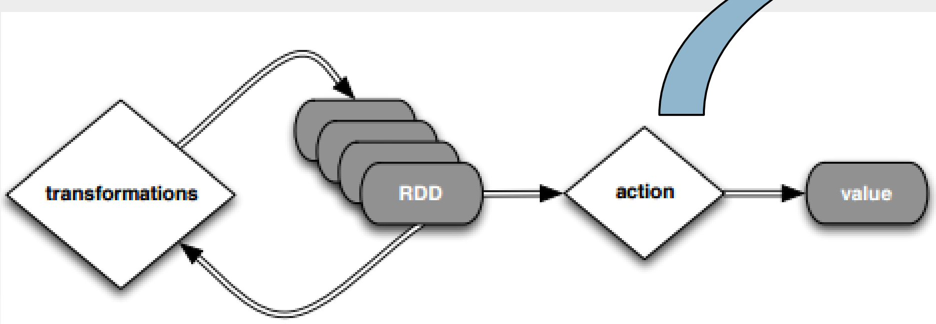
lines = sc.textFile("hdfs://...")



//Transformed RDD

errors = lines.filter(lambda x: x.startswith("Error"))

errors.persist()



errors.count()

count() causes Spark to: 1) read data; 2) sum within partitions; 3) combine sums in driver

Put transform and action together:

errors.filter(lambda x: "HDFS" in x).map(lambda x: x.split('\t')[3]).collect()

RDD Persistence: Cache/Persist

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
- When you persist an RDD, each node stores any partitions of it. You can reuse it in other actions on that dataset
- Each persisted RDD can be stored using a different *storage level,* e.g.
  - MEMORY_ONLY:
    - Store RDD as deserialized Java objects in the JVM.
    - If the RDD does not fit in memory, some partitions will not be cached and will be recomputed when they're needed.
    - This is the default level.
  - MEMORY_AND_DISK:
    - If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- cache() = persist(StorageLevel.MEMORY_ONLY)

Why Persisting RDD?

```
lines = sc.textFile("hdfs://...")
errors = lines.filter(lambda x: x.startswith("Error"))
errors.persist()
errors.count()
```

- If you do errors.count() again, the file will be loaded again and computed again.
- Persist will tell Spark to cache the data in memory, to reduce the data loading cost for further actions on the same data
- erros.persist() will do nothing. It is a lazy operation. But now the RDD says "read this file and then cache the contents". The action will trigger computation and data caching.

References

- http://spark.apache.org/docs/latest/index.html
- http://www.scala-lang.org/documentation/
- http://www.scala-lang.org/docu/files/ScalaByExample.pdf
- A Brief Intro to Scala, by Tim Underwood.
- Learning Spark. 1st and 2nd Edition

End of Chapter 4.1