An Introduction to Apache Spark

Amir H. Payberah
amir@sics.se

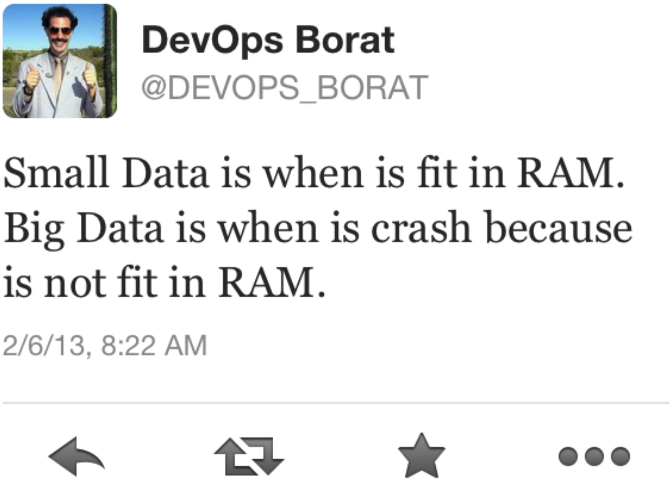
SICS Swedish ICT

Big Data

small data

big data

Big Data


DevOps Borat
@DEVOPS_BORAT
Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.
2/6/13, 8:22 AM

How To Store and Process Big Data?

Scale Up vs. Scale Out

- Scale up or scale vertically
- Scale out or scale horizontally

APACHE HBASE

mahout

GraphX

hadoop

kafka

Storm

Dato

Spark

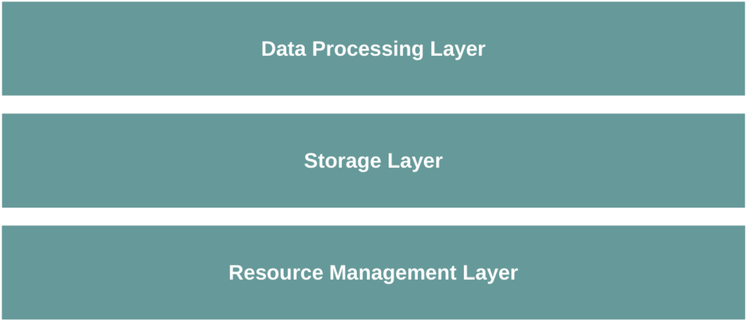
HIVE

S4 distributed stream computing platform

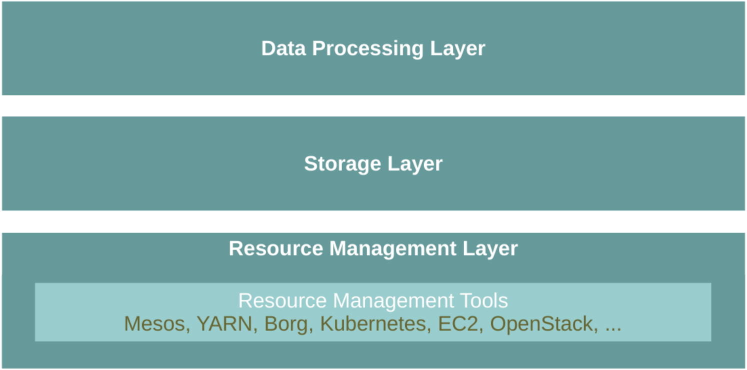
cassandra

Google Cloud Platform

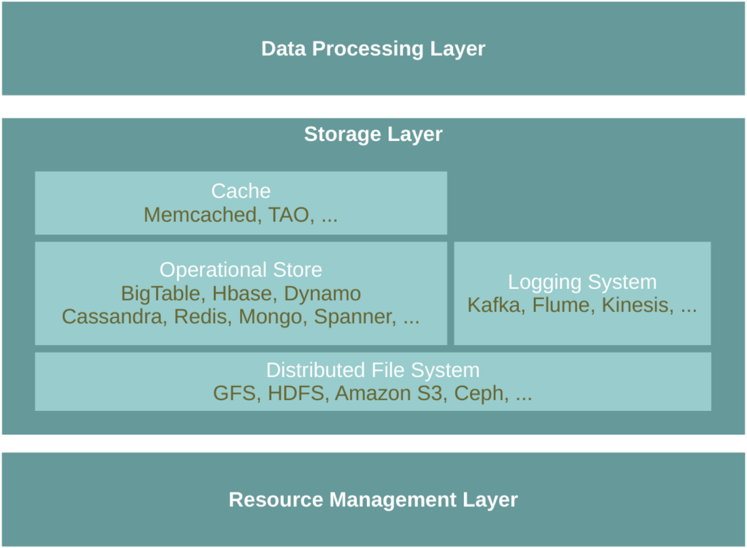
Three Main Layers: Big Data Stack



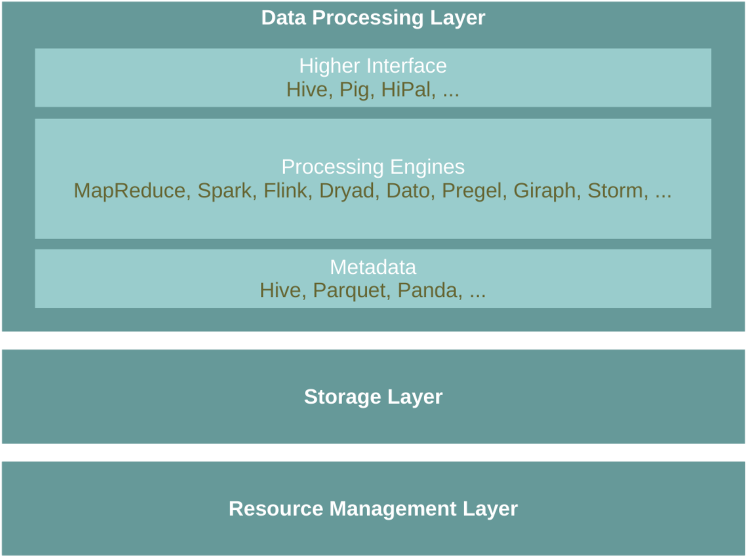
Resource Management Layer



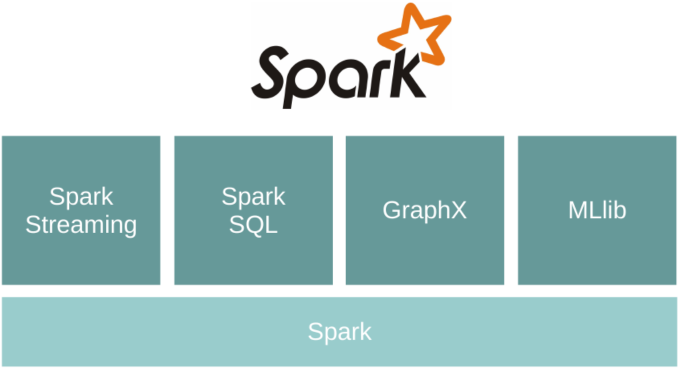
Storage Layer



Processing Layer



Spark Processing Engine



Cluster Programming Model

Warm-up Task (1/2)

- We have a huge text document.
- Count the number of times each distinct word appears in the file
- Application: analyze web server logs to find popular URLs.

Warm-up Task (2/2)

- File is too large for memory, but all $\langle \texttt{word}, \texttt{count} \rangle$ pairs fit in memory.
- `words(doc.txt) | sort | uniq -c`

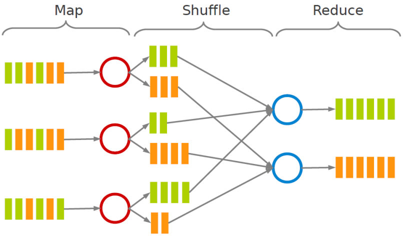
Warm-up Task in MapReduce

- `words(doc.txt) | sort | uniq -c`

Warm-up Task in MapReduce

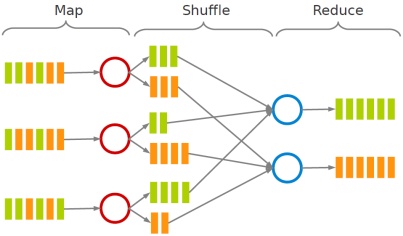
- `words(doc.txt) | sort | uniq -c`

- Sequentially read a lot of data.



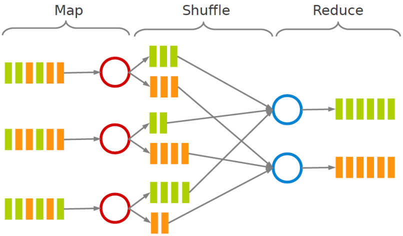
Warm-up Task in MapReduce

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.



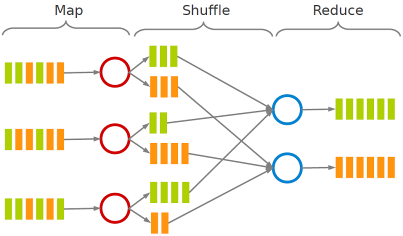
Warm-up Task in MapReduce

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.



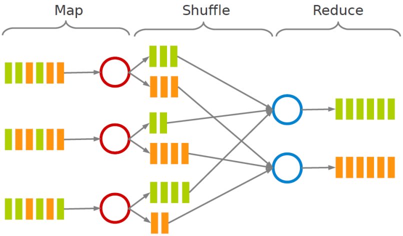
Warm-up Task in MapReduce

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.



Warm-up Task in MapReduce

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.



Example: Word Count

- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.

- The map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.
- The reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count - reduce

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.

- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Example: Word Count - map

```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

Example: Word Count - reduce

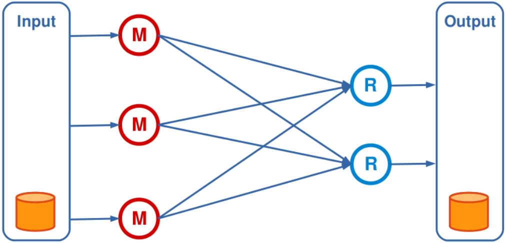
```java
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

Example: Word Count - driver

```java
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setCombinerClass(MyReduce.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```

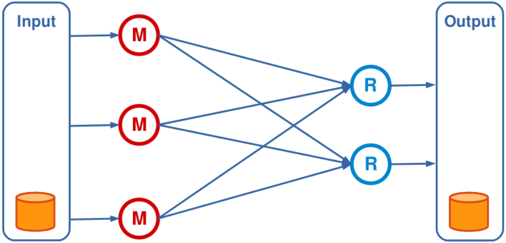
Data Flow Programming Model

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.



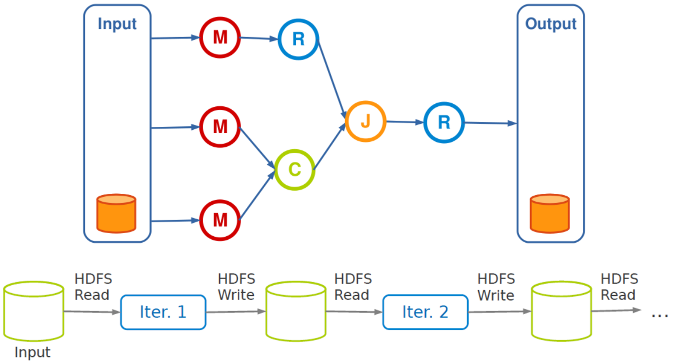
Data Flow Programming Model

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.
- MapReduce greatly simplified big data analysis on large unreliable clusters.



MapReduce Limitation

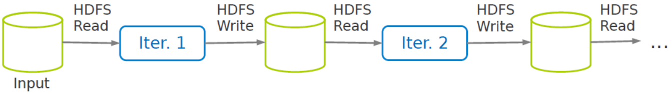
- MapReduce programming model has not been designed for complex operations, e.g., data mining.
- Very expensive (slow), i.e., always goes to disk and HDFS.



Spark (1/3)

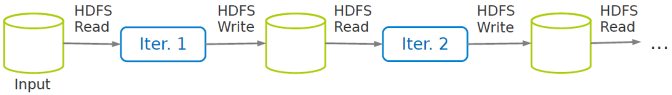
- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

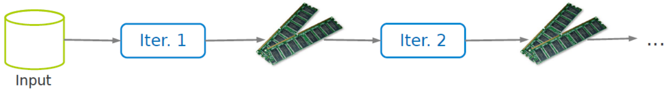
Spark (2/3)



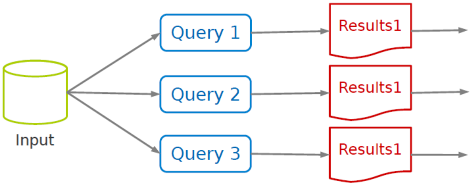
HDFS Read
Iter. 1
HDFS Write
HDFS Read
Iter. 2
HDFS Write
HDFS Read
...
Input

Spark (2/3)

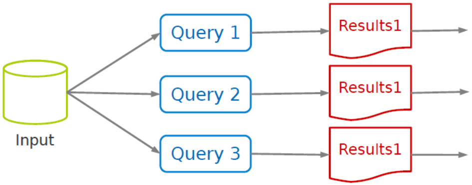

HDFS
Read
Iter. 1
HDFS
Write
HDFS
Read
Iter. 2
HDFS
Write
HDFS
Read
...
Input

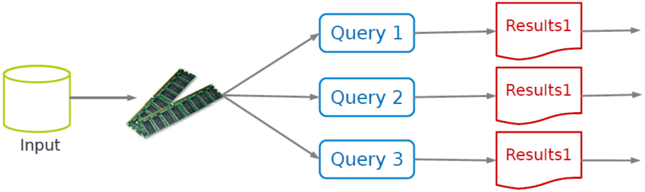

Iter. 1
Iter. 2
...
Input

Spark (3/3)



Spark (3/3)


Input
Query 1
Query 2
Query 3
Results1
Results1
Results1


Input
Query 1
Query 2
Query 3
Results1
Results1
Results1

Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
    - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/2)

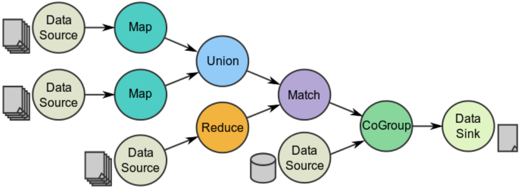
- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.
- Built through coarse grained transformations, e.g., `map`, `filter`, `join`.

Spark Programming Model

- Job description based on directed acyclic graphs (DAG).



Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

- Load text file from local FS, HDFS, or S3.

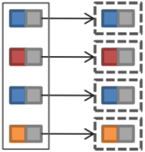
```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

RDD Higher-Order Functions

- Higher-order functions: RDDs operators.
- There are two types of RDD operators: transformations and actions.

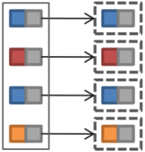
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

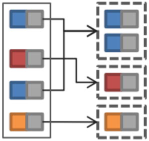
- All pairs are independently processed.



```scala
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}

// selecting those elements that func returns true.
val even = squares.filter(x => x % 2 == 0) // {4}

// mapping each element to zero or more others.
nums.flatMap(x => Range(0, x, 1)) // {0, 0, 1, 0, 1, 2}
```

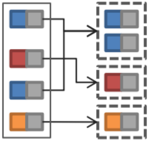
RDD Transformations - Reduce

- Pairs with identical key are grouped.
- Groups are independently processed.



RDD Transformations - Reduce

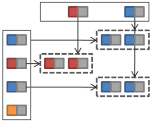
- Pairs with identical key are grouped.
- Groups are independently processed.



```
val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.reduceByKey((x, y) => x + y)
// {(cat, 3), (dog, 1)}

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}
```

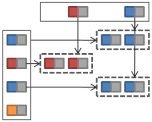
RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                                ("about.html", "3.4.5.6"),
                                ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                   ("about.html", "About")))

visits.join(pageNames)
  // ("index.html", ("1.2.3.4", "Home"))
  // ("index.html", ("1.3.3.1", "Home"))
  // ("about.html", ("3.4.5.6", "About"))
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)
or
nums.reduce(_ + _) // 6
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)
or
nums.reduce(_ + _) // 6
```

- Write the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

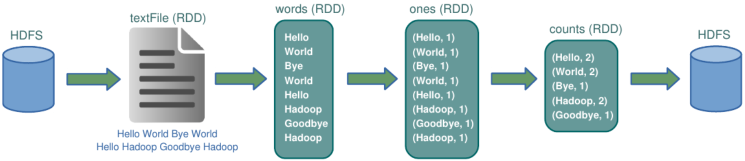
SparkContext

- Main entry point to Spark functionality.
- Available in shell as variable sc.
- In standalone programs, you should make your own.

```
val sc = new SparkContext(master, appName, [sparkHome], [jars])
```

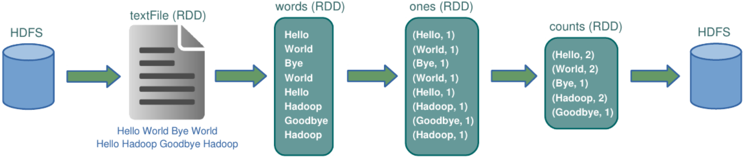
Example: Word Count

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



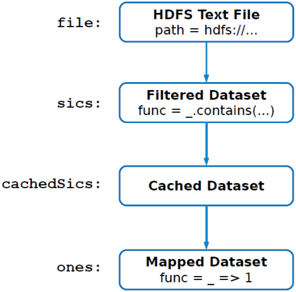
Example: Word Count

```
val textFile = sc.textFile("hdfs://...")

val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Lineage

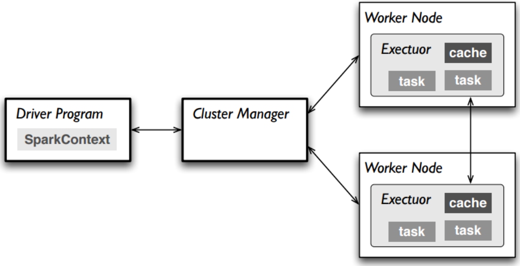
- Lineage: transformations used to build an RDD.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.




```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

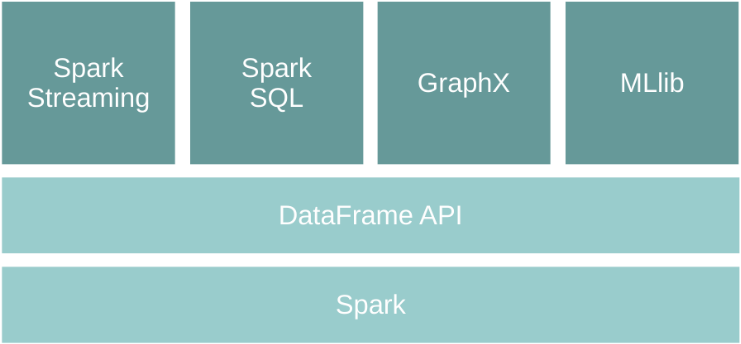
Spark Execution Plan

1. Connects to a cluster manager, which allocate resources across applications.
2. Acquires executors on cluster nodes (worker processes) to run computations and store data.
3. Sends app code to the executors.
4. Sends tasks for the executors to run.



Spark SQL

Spark SQL



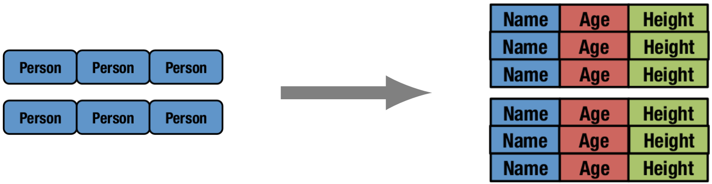
Spark Streaming
Spark SQL
GraphX
MLlib
DataFrame API
Spark

DataFrame

- A DataFrame is a distributed collection of rows with a homogeneous schema.
- It is equivalent to a table in a relational database.
- It can also be manipulated in similar ways to RDDs.
- DataFrames are lazy.

Adding Schema to RDDs

- Spark + RDD: functional transformations on partitioned collections of opaque objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating DataFrames

- The entry point into all functionality in Spark SQL is the SQLContext.

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json(...)
```

DataFrame Operations (1/2)

- Domain-specific language for structured data manipulation.

```
// Show the content of the DataFrame
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// name
// Michael
// Andy
// Justin
```

DataFrame Operations (2/2)

- Domain-specific language for structured data manipulation.

```
// Select everybody, but increment the age by 1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20

// Select people older than 21
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// Count people by age
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

Running SQL Queries Programmatically

- Running SQL queries programmatically and returns the result as a DataFrame.
- Using the `sql` function on a `SQLContext`.

```
val sqlContext = ...  // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
```

Converting RDDs into DataFrames

```scala
// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
                .map(p => Person(p(0), p(1).trim.toInt)).toDF()

people.registerTempTable("people")
```

Converting RDDs into DataFrames

```
// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
               .map(p => Person(p(0), p(1).trim.toInt)).toDF()

people.registerTempTable("people")
```

```
// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext
    .sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
teenagers.map(t => "Name: " + t.getAs[String]("name")).collect()
         .foreach(println)
```

Spark Streaming

Data Streaming

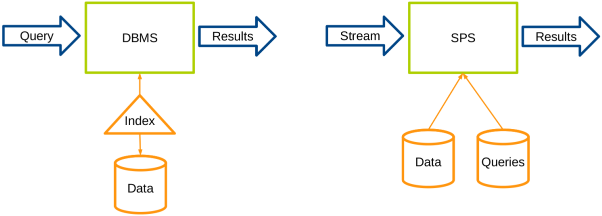
- Many applications must process large streams of live data and provide results in real-time.
  - Wireless sensor networks
  - Traffic management applications
  - Stock marketing
  - Environmental monitoring applications
  - Fraud detection tools
  - ...

Stream Processing Systems

- Stream Processing Systems (SPS): data-in-motion analytics
  - Processing information as it flows, without storing them persistently.
- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

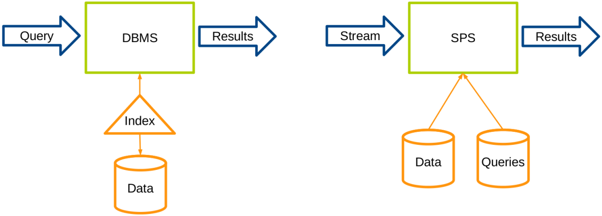
DBMS vs. SPS (1/2)

- DBMS: persistent data where updates are relatively infrequent.
- SPS: transient data that is continuously updated.



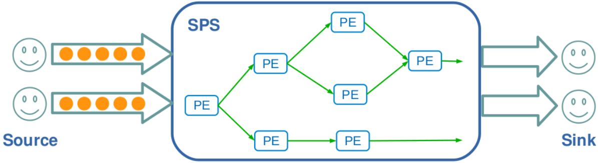
DBMS vs. SPS (2/2)

- DBMS: runs queries just once to return a complete answer.
- SPS: executes standing queries, which run continuously and provide updated answers as new data arrives.



SPS Architecture

- Data source: producer of streaming data.
- Data sink: consumer of results.
- Data stream is unbound and broken into a sequence of individual data items, called tuples.

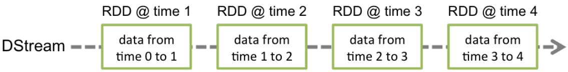


Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Treats each batch of data as RDDs and processes them using RDD operations.
  - Finally, the processed results of the RDD operations are returned in batches.

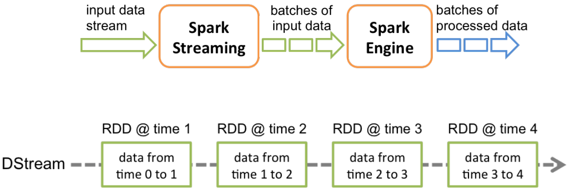
Discretized Stream Processing (DStream)

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...



Discretized Stream Processing (DStream)

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...

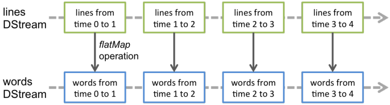


- Initializing Spark streaming

```
val scc = new StreamingContext(master, appName, batchDuration,
[sparkHome], [jars])
```

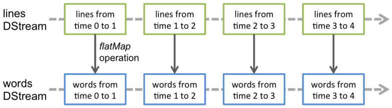
DStream API

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations: map, join, ...

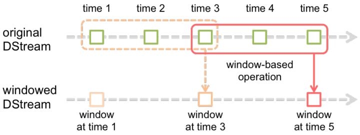


DStream API

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations: map, join, ...



  - Window operations: group all the records from a sliding window of the past time intervals into one RDD: window, reduceByAndWindow, ...

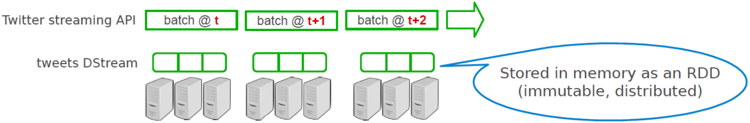


Window length: the duration of the window.
Slide interval: the interval at which the operation is performed.

Example 1 (1/3)

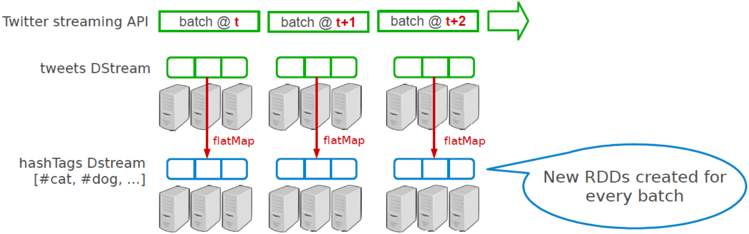
- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
```



- Get hash-tags from Twitter.

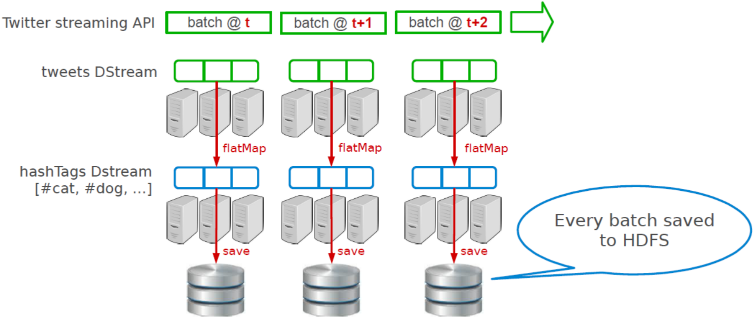
```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap(status => getTags(status))
```



Example 1 (3/3)

- Get hash-tags from Twitter.

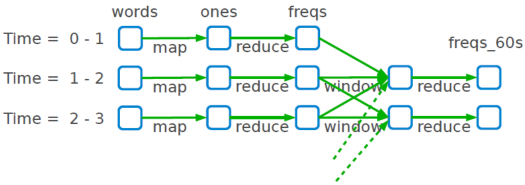
```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Example 2

- Count frequency of words received in last minute.

```
val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(args(1), args(2).toInt)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs_60s = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
```



Summary

Summary

- How to store and process big data? scale up vs. scalue out
- Cluster programming model: dataflow
- Spark: RDD (transformations and actions)
- Spark SQL: DataFrame (RDD + schema)
- Spark Streaming: DStream (sequence of RDDs)

Questions?