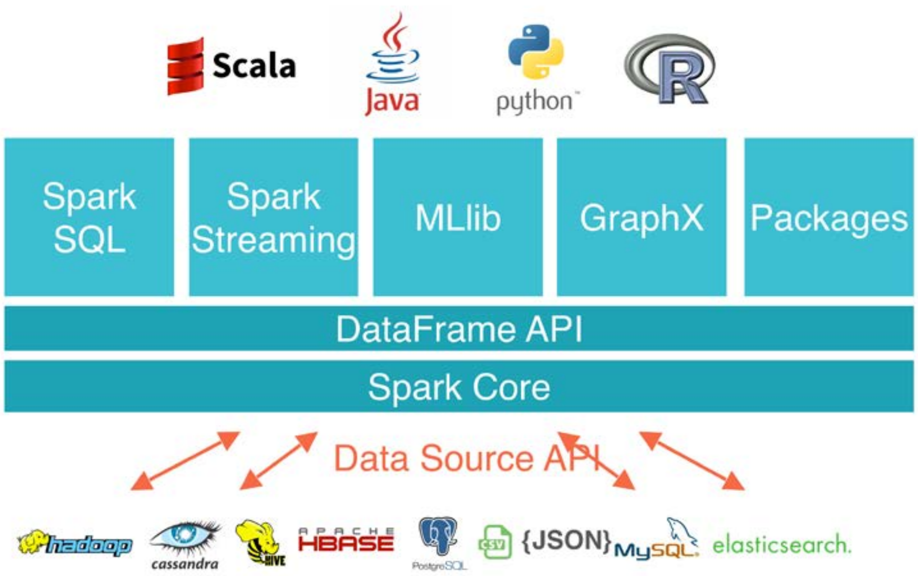
CMSC 691
High Performance Distributed Systems

Apache Spark

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Apache Spark

- Open source cluster computing framework for large scale data
- Spark core: APIs in Java, Scala, Python and R
- Higher-level tools: Spark SQL, MLlib, GraphX, Spark Streaming
- Top trending and paying technology according to Stack Overflow!

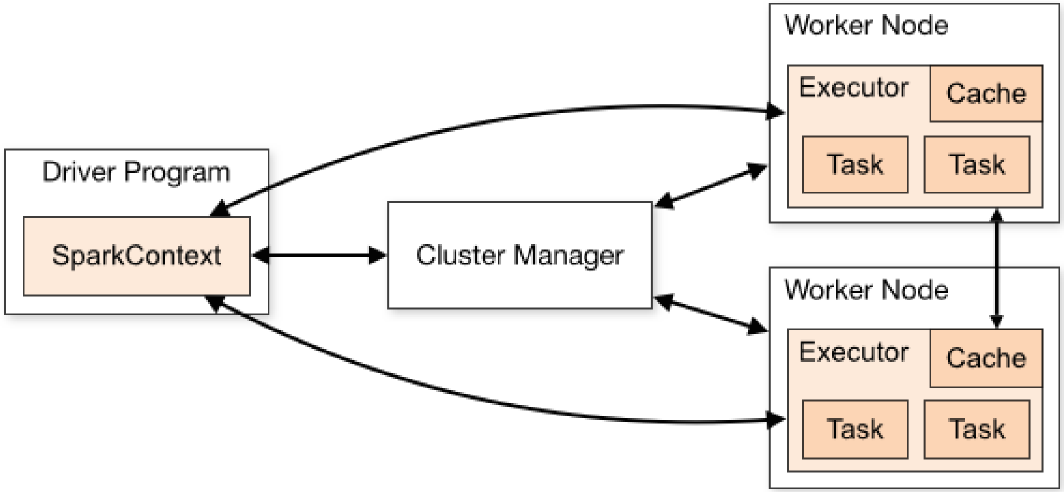


Apache Spark vs Hadoop

- Batch, interactive, and real-time programming
- Spark also integrates Hadoop and HDFS ecosystem
- High-level API: DataFrames: collections of rows with a schema
- Low-level API: Resilient distributed dataset (RDD): read-only multiset of distributed data maintained in a fault-tolerant way
- Much faster than Hadoop in iterative algorithms. Improves efficiency through **in-memory computing primitives** and general computation graphs (up to 100x faster than Hadoop!)
- Spark code size is much smaller (Scala, functional progr. lang.)

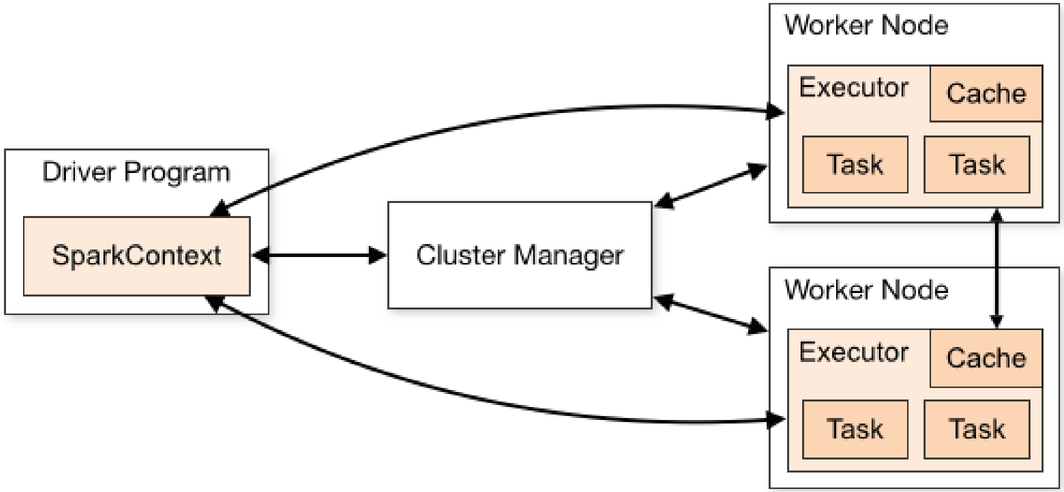
Apache Spark architecture

- SparkContext object defines your job (main program), run in the driver (master of the cluster), while actual computation is conducted in remote executors on nodes in the cluster
- Worker: any node that can run code in the cluster
- Executor: process on a worker node that runs and keeps data
- Task: unit of work that will be sent to one executor



Spark driver and processing

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

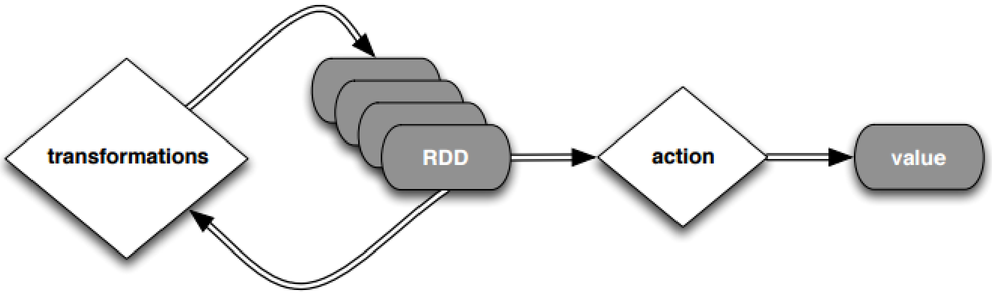


Resilient Distributed Datasets (RDD)

- Primary abstraction in Spark
- Fault-tolerant collection of data elements operated in parallel
- Two types:
    - *Parallelized collections*: parallelizing an existing collection in your driver program
    - *Referencing dataset in an external storage system*: HDFS, HBase, or any data source offering a Hadoop InputFormat
- Operations:
    - *Transformations*: create a new dataset from an existing one
    - *Actions*: return a value to the driver program after running a computation on the dataset

Resilient Distributed Datasets (RDD)

- Transformations are **lazy** (not computed immediately) but when an action is run on the transformed RDD
- Optimize the required calculations
- RDD can be persisted into storage in memory or disk



- Functional programming not procedural programming, oriented towards data transformations
- Simplified programming using lambda functions in Java 8

RDD 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

RDD 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)

Spark example (Java): transformations and actions

```
public class CountLinesWord {

    public static void main(String[] args)
    {
        String inputfile = args[0];
        SparkConf conf = new SparkConf().setAppName("Count lines with a given word");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<String> data = sc.textFile(inputfile).cache();

        long numLines1 = data.filter(new Function<String, Boolean>() {
            public Boolean call(String s) { return s.contains("sit"); }
        }).count();

        long numLines2 = data.filter(new Function<String, Boolean>() {
            public Boolean call(String s) { return s.contains("urna"); }
        }).count();

        System.out.println("Lines with sit: " + numLines1 + ", lines with urna: " + numLines2);
    }
}
```

SparkContext (job)

Read data file

Transformation

Input

Output

Action

Spark persistence

- Cache a dataset in memory across operations .cache()
- Each node stores in memory any slices of it that it computes and **reuses** them in other actions on that dataset
- Options: *memory only*, *memory and disk*, *disk only*, others
- *Memory only*: if the RDD does not fit in memory some partitions will not be cached and will be recomputed on the fly each time they're needed (default)
- *Memory and disk*: if the RDD does not fit in memory some partitions will not be cached and will be recomputed on the fly each time they're needed (default)
- Remember: Hadoop stored results for all transformations in disk

Apache Spark Installation and Eclipse project setup

- Download Spark 2.0.1 with Hadoop 2.7 from the Spark website
- Extract into a folder and edit your .bashrc to add the path
- In Eclipse, create a new Maven project for your Spark code
- Edit the pom.xml to add Spark dependencies

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.0.1</version>
</dependency>
```

- Use the sample code provided for the lecture
- Compile and build the jar package using maven
- Execute: *spark-submit --class PACKAGE.MAINCLASS --master local[4] target/PACKAGE.jar inputdata*

CMSC 691
High Performance Distributed Systems

Apache Spark

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu