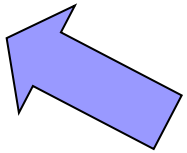
Modern Database Systems

Apache Spark

Doc. RNDr. Irena Holubova, Ph.D.
Irena.Holubova@matfyz.cuni.cz

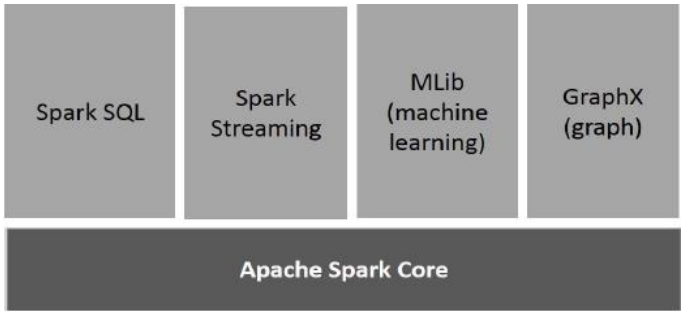
Big Data Related Technologies

- Distributed file systems
  - e.g., HDFS
- Distributed databases
  - Primarily NoSQL databases
  - And many other types
- Cloud computing
- Data analytics
  - Batch
  - Real-time
  - Stream
- …

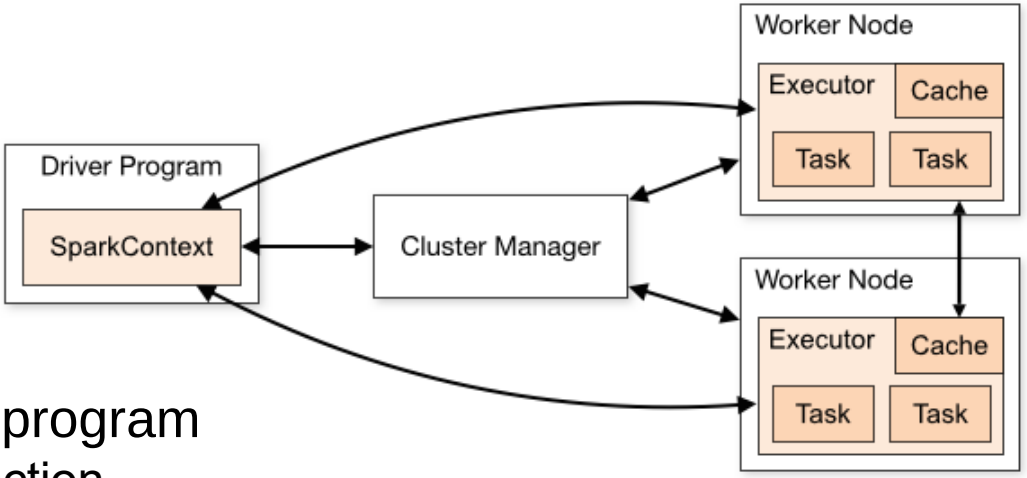


Apache Spark

- Initial release : 2014
- Unified analytics engine for large-scale data processing
  - Runs on a cluster of nodes
- Contains:
  - High-level APIs in Java, Scala, Python and R
  - Optimized engine that supports general execution graphs (DAGs)
    - MapReduce has only 2 levels
  - Higher-level tools
    - Spark SQL (SQL and structured data processing)
    - MLlib (machine learning)
    - GraphX (graph processing)
    - Spark Streaming



Spark Application



- Spark application = driver program
  - Runs the user's main function
  - Executes parallel operations on a cluster
    - Independent set of processes
    - Coordinated by SparkContext object in the driver program
- SparkContext can connect to several types of cluster managers
  - They allocate resources across applications
- When connected:
  1. Spark acquires executors on nodes in the cluster
     - Processes that run computations and store data for the application
  2. Sends the application code to the executors
     - Defined by JAR or Python files passed to SparkContext
  3. Sends tasks to the executors to run

Spark Application

- Each application gets its own executor processes which run tasks in multiple threads
  - Pros: isolating of applications
    - Scheduling + executing
  - Cons: data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system
- Driver program
  - Must listen for and accept incoming connections from its executors throughout its lifetime
  - Should be run close to the worker nodes
    - Preferably on the same local area network
  - Has a web UI
    - Displays information about running tasks, executors, and storage usage

Cluster Managers

- Spark is agnostic to the underlying cluster manager
- Cluster managers
  - Standalone – a simple cluster manager included with Spark
    - Makes it easy to set up a cluster
  - Apache Mesos – a general cluster manager
    - Can also run Hadoop MapReduce and service applications
  - Hadoop YARN – the resource manager in Hadoop 2
  - Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications

Initializing Spark

1. Build a SparkConf object
   - Contains information about application
   - appName = application name to show on the cluster UI
   - master = Spark/Mesos/YARN cluster URL or string “local” to run in local mode
2. Create a JavaSparkContext object
   - Tells Spark how to access a cluster

```
SparkConf conf =
  new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc =
  new JavaSparkContext(conf);
```

Resilient Distributed Dataset (RDD)

- Immutable collection of elements partitioned across the nodes of the cluster
  - Can be operated on in parallel
  - Can be persisted in memory
  - Automatically recover from node failures
- Ways to create RDDs:
  1. Parallelizing an existing collection in a driver program
  2. Referencing a dataset in an external storage system
     - e.g., HDFS, HBase, …
     - In general: any offering a Hadoop InputFormat

Resilient Distributed Dataset (RDD)

Parallelized Collections

- Parallelized collections are created by calling SparkContext's parallelize method
  - Elements of the collection are copied to form a distributed dataset
  - The distributed dataset (distData) can be operated on in parallel
    - See later

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

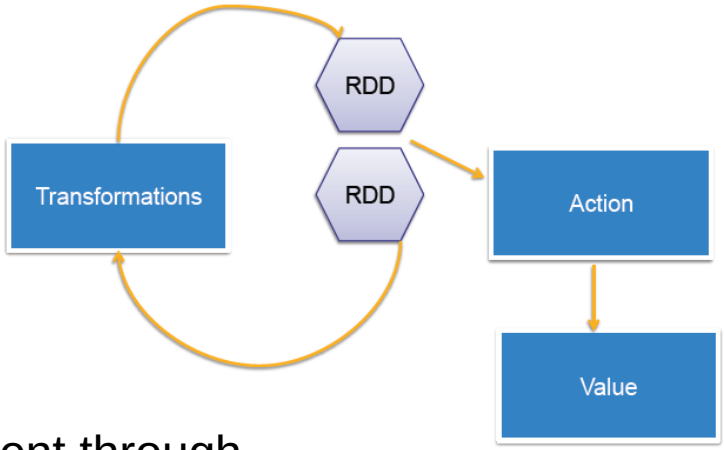
Resilient Distributed Dataset (RDD)

External Datasets

- Spark can create distributed datasets from any storage source supported by Hadoop
  - Local file system, HDFS, Cassandra, HBase, …
- Supports text files, SequenceFiles, and any other Hadoop InputFormat
- Example:
  - Text file RDDs can be created using SparkContext's textFile method
    - Takes an URI for the file (local, HDFS, …)
    - Reads it as a collection of lines
    - Optional argument: number of partitions of the file
      - Default: one partition for each block of the file (128MB by default in HDFS)
  - Once created, distFile can be acted on by dataset operations

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

RDD Operations



1. Transformations = create (lazily)
   a new dataset from an existing one
   - e.g., map = passes each dataset element through
     a function and returns a new RDD representing the results
2. Actions = return a value to the driver program after running a computation on the dataset
   - e.g., reduce = aggregates all the elements of the RDD using some function and returns the final result to the driver program

- By default: each transformed RDD may be recomputed each time we run an action on it
  - We may also persist an RDD in memory using the persist (or cache) method
    - Much faster access the next time we query it
  - There is also support for persisting RDDs on disk or replicated across multiple nodes

Transformations

- **map**(func) Returns a new distributed dataset formed by passing each element of the source through a function func.
- **union**(otherDataset) Returns a new dataset that contains the union of the elements in the source dataset and the argument.
  - **intersection, distinct**
- **filter**(func) Returns a new dataset formed by selecting those elements of the source on which func returns true.
- **reduceByKey**(func, [numPartitions]) 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 func, which must be of type (V,V) => V. The number of reduce tasks is configurable through an optional second argument.
- **sortByKey**([ascending], [numPartitions]) 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.
- …

Actions

- **reduce**(func) Aggregates the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **count**() Returns the number of elements in the dataset.
- **first**() Returns the first element of the dataset.
- **take**(n) Returns an array with the first n elements of the dataset.
- **takeOrdered**(n, [ordering]) Returns the first n elements of the RDD using either their natural order or a custom comparator.
- …

Shuffle Operations

- Certain operations trigger a shuffle = mechanism for re-distributing data so that it's grouped differently across partitions
- Involves copying data across executors and machines
  - Complex and costly operation
- Example: reduceByKey
  - Generates a new RDD where all values for a single key are combined into a tuple
    - The key and the result of executing a reduce function against all values associated with that key
  - Problem: not all values for a single key necessarily reside on the same partition or the same machine
  - Shuffle: Spark reads from all partitions to find all the values for all keys, and then brings together values across partitions to compute the final result for each key

Simple Spark Example

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

1. Defines a base RDD from an external file
   - Not loaded in memory or otherwise acted on, due to laziness
   - lines is merely a pointer to the file
2. Defines lineLengths as the result of a map transformation
   - Not immediately computed, due to laziness
3. Runs reduce = action
   - Spark breaks the computation into tasks to run on separate machines
   - Each machine runs both its part of the map and a local reduction, returning its answer to the driver program

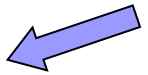
Passing Functions to Spark

- In Java, functions are represented by classes implementing the interfaces in the org.apache.spark.api.java.function package
- Two ways to create a function:
  1. Use lambda expressions to concisely define an implementation
  2. Implement Function interface and pass an instance of it to Spark

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map (
  new Function<String, Integer>() {
    public Integer call(String s) { return s.length(); }
  } );
int totalLength = lineLengths.reduce (
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
  } );
```

Closures

```
int counter = 0;
JavaRDD<Integer> rdd =
  sc.parallelize(data);
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```



1. Spark computes the task's closure
   - Variables and methods visible for the executor to perform computations
2. Spark breaks up the processing of RDD operations into tasks, each executed by an executor
   - The closure is serialized and sent to each executor
3. The variables within the closure sent to each executor are copies
   - The value of counter will still be zero

- Note: In local mode, in some circumstances the foreach function will actually execute within the same JVM as the driver and will reference the same original counter, and may actually update it
- Solution: two limited types of shared variables:
  - Broadcast variables
  - Accumulators

Broadcast Variables

- Allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
  - e.g., to give every node a copy of a large input dataset in an efficient manner
- Created from a variable v by calling SparkContext.broadcast(v)
  - Its value can be accessed by calling the value method.

```
Broadcast<int[]> broadcastVar =  sc.broadcast
  (new int[] {1, 2, 3});

broadcastVar.value();
// returns [1, 2, 3]
```

Accumulators

- Only “added” to through an associative and commutative operation
- Can be used to implement counters (as in MapReduce) or sums
- A numeric accumulator can be created by calling SparkContext.longAccumulator() or doubleAccumulator()
- Tasks running on a cluster can then add to it using the add method.
- Only the driver program can read the accumulator’s value, using its value method.

```
LongAccumulator accum = jsc.sc().longAccumulator();

sc.parallelize(Arrays.asList(1, 2, 3, 4))
  .foreach(x -> accum.add(x));

accum.value();
// returns 10
```

Spark SQL

- Spark module for structured data processing
- More information about the structure of both the data and the computation being performed
  - Internally, Spark SQL uses this extra information to perform extra optimizations
- Interact with Spark SQL: SQL, Dataset API, ...

RDD vs. DataFrame vs. Dataset

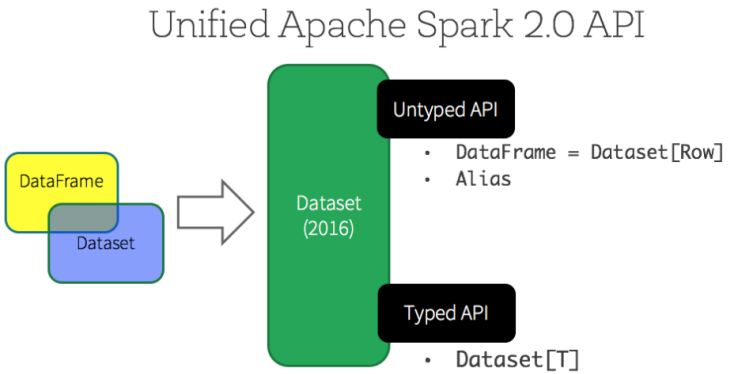
- RDD = primary API in Spark since its inception
  - Since Spark 1.0
  - Internally each final computation is still done on RDDs
- DataFrame = data organized into named columns
  - Since Spark 1.3
  - Distributed collection of data, which is organized into named columns
    - Designed to make data processing easier
      - Higher level of abstraction
      - Similar to a table in a relational database or a data frame in R/Python
  - Can be constructed from: structured data files, tables in Hive, external databases, existing RDDs , …
  - API: Scala, Java, Python, R

RDD vs. DataFrame vs. Dataset

- Dataset = a distributed collection of data
  - Since Spark 1.6
  - Provides the benefits of
    - RDDs - strong typing, ability to use powerful lambda functions
    - Spark SQL - optimized execution engine
      - i.e. DataFrame processing
  - Can be constructed from: JVM objects
  - API: Scala, Java

RDD vs. DataFrame vs. Dataset

- Since Spark 2.0: unification of DataFrame and Dataset
- Two distinct APIs:
  - Untyped API
    - Conceptually: DataFrame ~ collection of generic objects `Dataset<Row>`, where a **`Row`** is a generic untyped JVM object
  - Strongly-typed API
    - Conceptually: Dataset ~ collection `Dataset<T>` of strongly-typed JVM objects, dictated by a case class `T`
      - Defined in Scala or a class in Java



Basic Examples

```
SparkSession spark = SparkSession.builder().
  appName("Java Spark SQL basic example").
  config("spark.some.config.option", "some-value").getOrCreate();

Dataset<Row> df =
  spark.read().json("examples/src/main/resources/people.json");

df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

df.printSchema();
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
```

DataFrame - untyped

4 lines (3 sloc) | 73 Bytes

```
1  {"name":"Michael"}
2  {"name":"Andy", "age":30}
3  {"name":"Justin", "age":19}
```

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

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

// Select people older than 21
df.filter(col("age").gt(21)).show();
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+
```

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

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

- Temporary views are session-scoped
  - Disappear if the session that creates it terminates
- Global temporary view = a temporary view shared among all sessions
  - Keeps alive until the Spark application terminates
  - Tied to a system preserved database `global_temp`
    - `df.createGlobalTempView("people");`
    - Must use the qualified name to refer it
      - e.g. `SELECT * FROM global_temp.people`

Creating Datasets

- Similar to RDDs
- Use a specialized Encoder to serialize the objects for processing or transmitting over the network
- Encoders
  - Convert a JVM object of type T to and from the internal Spark SQL representation
  - Are code generated dynamically
  - Use a format that allows to perform many operations
    - E.g., filtering, sorting and hashing without deserializing

```
public static class Person implements Serializable {
  private String name;
  private int age;

  public String getName() { return name; }
  public void   setName(String name) { this.name = name; }
  public int    getAge() { return age; }
  public void   setAge(int age) { this.age = age; }
}

// Encoders are created for Java beans
Encoder<Person> personEncoder = Encoders.bean(Person.class);

// DataFrames can be converted to a Dataset by providing a class.
// Mapping based on name
String path = "examples/src/main/resources/people.json";
Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);

peopleDS.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Interoperating with RDDs

- Converting existing RDDs into Datasets:
  1. Usage of reflection to infer the schema of an RDD
     - Leads to more concise code
  2. Through a programmatic interface that allows to construct a schema and then apply it to an RDD
     - More verbose
     - Allows to construct Datasets when the columns and their types are not known until runtime, i.e. we cannot create a class beforehand

```
// Create an RDD of Person objects from a text file
JavaRDD<Person> peopleRDD = spark.read()
  .textFile("examples/src/main/resources/people.txt")
  .javaRDD()
  .map(line -> {
    String[] parts = line.split(",");
    Person person = new Person();
    person.setName(parts[0]);
    person.setAge(Integer.parseInt(parts[1].trim()));
    return person;
  });

// Apply a schema to an RDD of JavaBeans to get a DataFrame
Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);

// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row> teenagersDF =
  spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19");
```

```
// Create an RDD
JavaRDD<String> peopleRDD = spark.sparkContext()
  .textFile("examples/src/main/resources/people.txt", 1).toJavaRDD();

// Convert records of the RDD (people) to Rows
JavaRDD<Row> rowRDD = peopleRDD.map((Function<String, Row>) record -> {
  String[] attributes = record.split(",");
  return RowFactory.create(attributes[0], attributes[1].trim());
});

// The schema is encoded in a string
String schemaString = "name age";

// Generate the schema based on the string of schema
List<StructField> fields = new ArrayList<>();
for (String fieldName : schemaString.split(" ")) {
  StructField field = DataTypes.createStructField(fieldName,
                      DataTypes.StringType, true);
  fields.add(field);
}
StructType schema = DataTypes.createStructType(fields);

// Apply the schema to the RDD
Dataset<Row> peopleDataFrame = spark.createDataFrame(rowRDD, schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");

// SQL can be run over a temporary view created using DataFrames
Dataset<Row> results = spark.sql("SELECT name FROM people");
```

References

- Spark Overview https://spark.apache.org/docs/latest/index.html
- Apache Spark Examples https://spark.apache.org/examples.html
- Mastering Apache Spark 2.3.2 https://jaceklaskowski.gitbooks.io/mastering-apache-spark/
- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html