NDBI040: PRACTICAL CLASS 3

APACHE SPARK

(RECOMMENDED) REQUIREMENTS

- Database concepts
- Java or Python or Scala programming

- Java 8 JDK or newer installed
- Maven
- NetBeans IDE (or another IDE of yours choice)
- macOS / Linux command line or PuTTy / WinSCP on Windows

SERVER ACCESS

CONNECT TO NOSQL SERVER

- `ssh` on macOS / Linux
- `PuTTy` on Windows

- nosql.ms.mff.cuni.cz:42222
- Login and password send by e-mail
- Change your initial password (if not yet changed) by `passwd`

TRANSFER FILES

- `scp` on macOS / Linux
- `WinSCP` on Windows

APACHE SPARK

- Cluster computing technology
- Based on Hadoop MapReduce, extends the model
- Supports in-memory cluster computing

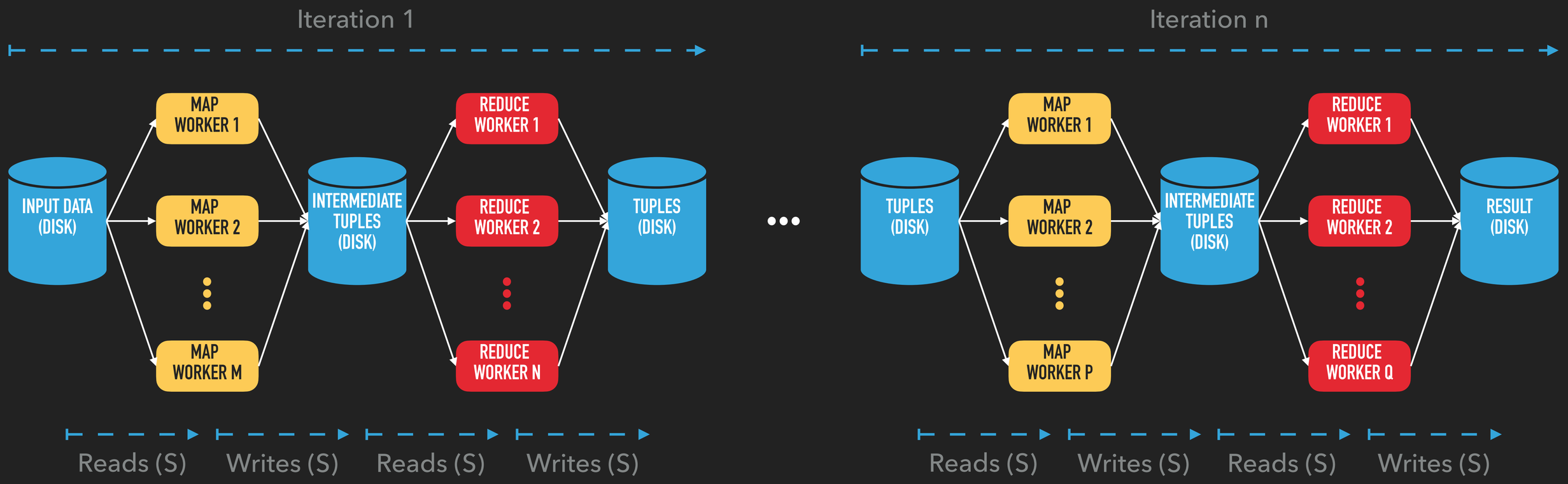
- Multi-language support: Java, Scala, Python, R
- Supports: MapReduce, SQL queries, Streaming data, Machine learning, Graph algorithms

MAPREDUCE VS. APACHE SPARK

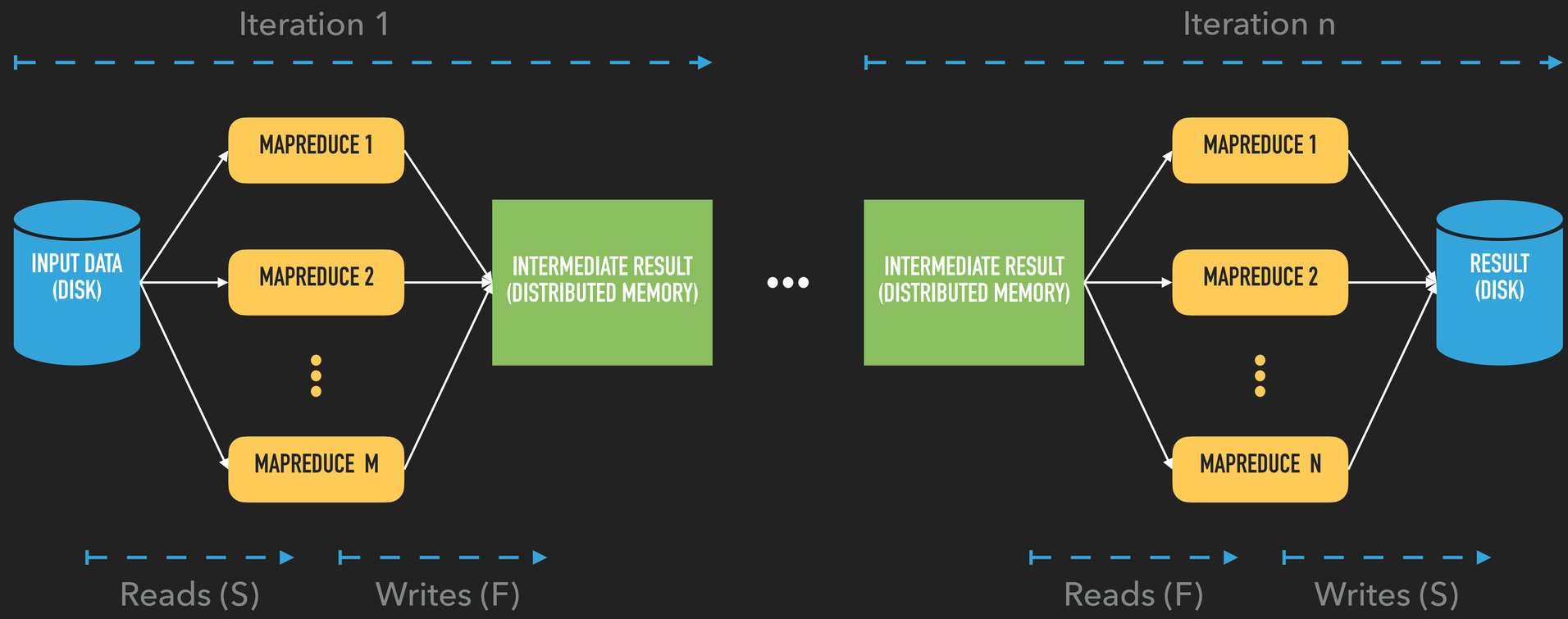
- MapReduce
  - You have to write data to an external storage system before reusing data between computations
  - Data sharing is slow(er) due to replication, serialization and disk IO

- Apache Spark
  - Resilient Distributed Dataset (RDD) supports in-memory processing (much faster than writing on disk)

MAPREDUCE VS. APACHE SPARK



MAPREDUCE VS. APACHE SPARK



INITIALIZING SPARK

- First, build an instance of `SparkConf` that contains information about your application
  - `appName` application name to show on the cluster UI
  - `master` Spark/Mesos/YARN cluster URL, e.g. spark://196.168.234.219:7070/, or string `"local"` to run in local mode
- Next, create an instance of `SparkContext` that tells Spark how to access a cluster

SPARK SHELL

- You have already access to instance of SparkContext, e.g. available as `sc`

(JAVA) APPLICATION CODE

- Create an instance of 1) `SparkConf` and 2) `JavaSparkContext`

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

RESILIENT DISTRIBUTED DATASET (RDD)

- Immutable distributed collection of objects
- RDD is divided into logical partitions and distributed across the nodes on the cluster
- Can be operated on in parallel
- Can be persisted in memory

WAYS TO CREATE RDD

- Parallelizing an existing collection in a driver program
  - `JavaRDD<Integer> distData = context.parallelize(Arrays.asList(1, 2, 3, 4, 5));`

- Referencing a dataset in an external storage system (e.g. HDFS, HBase, ...)
  - `JavaRDD<String> lines = context.textFile("path_to_file");`

RDD OPERATIONS: TRANSFORMATIONS

- map(func)
- filter(func)
- flatMap(func)
- mapPartitions(func)
- mapPartitionsWithIndex(func)
- sample(withReplacement, fraction, seed)
- union(otherDataset)
- intersection(otherDataset)
- distinct([numTasks])
- groupByKey([numTasks])
- reduceByKey(func, [numTasks])
- aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
- sortByKey([ascending], [numTasks])
- join(otherDataset, [numTasks])
- cogroup(otherDataset, [numTasks])
- cartesian(otherDataset)
- pipe(command, [envVars])
- coalesce(numPartitions)
- repartition(numPartitions)
- repartitionAndSortWithinPartitions(partitioner)

RDD OPERATIONS: ACTIONS

- No transformation is performed until an action is executed

- reduce(func)
- collect()
- count()
- first()
- take(n)
- takeSample(withReplacement, num, [seed])
- takeOrdered(n, [ordering])
- saveAsTextFile(path)
- saveAsSequenceFile(path)
- saveAsObjectFile(path)
- countByKey()
- foreach(func)

PASSING FUNCTIONS TO SPARK

BY LAMBDA EXPRESSION

- Use lambda expression

```
data.reduceByKey((a, b) -> a + b);
```

BY IMPLEMENTING INTERFACE FUNCTION

- In Java, functions are represented by classes implementing interface `Function[2,3,4]<IN[,IN[,IN[,IN]]], OUT>` from package `org.apache.spark.api.java.function`
- Pass an instance of implemented class (either as an anonymous inner class or a named one) to Spark

```
data.reduceByKey(new Function2<Integer, Integer, Integer>() {
  @Override
  public Integer call(Integer a, Integer b) throws Exception {
    return a + b;
  }
});
```

EXERCISE 1: WORDCOUNT (SOLVED)

CREATE YOUR WORKING DIRECTORY

- `cd ~`
- `mkdir -p mySpark/WordCount`
- `cd mySpark/WordCount`

COPY THE SAMPLE INPUT DATA

- `cp /home/NDBI040/mapreduce/input1/movies.txt ~/mySpark/WordCount`

OPEN SPARK SHELL (SCALA ENVIRONMENT)

- `spark-shell`

READ THE INPUT FILE USING SCALA API AND CREATE RDD

- `scala> val data = sc.textFile("movies.txt")`

EXERCISE 1: WORDCOUNT (SOLVED)

EXECUTE WORDCOUNT TRANSFORMATION

- i.e. split each line into words, map each word into `(word, 1)` pair, and reduce those keys
- `scala> val result = data.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_+_)`

APPLY THE ACTION

- i.e. store all the transformations result into a text file.
  - `scala> result.saveAsTextFile("output")`

QUIT SPARK SHELL

- `scala> :quit`

CHECK THE OUTPUT

- cat output/part-00000

EXERCISE 2: JAVA WORDCOUNT (SOLVED)

- Download project `NDBI040-wordCountSpark` from the practical class website
- Build the project in order to create jar file
  - Check folder `NDBI040-wordCountSpark/target/` to see the build target
- Deploy the task on NoSQL server
  - Use either `scp` or `WinSCP` to copy `ndbi040-wordCountSpark-1.0.jar` into folder `~/mySpark/WordCount`
- Run the task
  - `spark-submit --class WordCount --master local ndbi040-wordCountSpark-1.0.jar movies.txt output2`
- Check the output

EXERCISE 3: PI ESTIMATION

- Estimate the value of π by Monte Carlo method
  - https://en.wikipedia.org/wiki/Monte_Carlo_method
- Pick 1 << 17 (2^16) random points in the unit square ((0,0) to (1,1)) and see how many fall in the unit circle
- The fraction should be π / 4
  - Parallelize input collection
  - `Math.random()` returns random double value
  - You only need functions `map` and `reduce`
- Implement, compile, deploy and submit the task

SPARK SQL

- Spark module for structured data processing
- Spark SQL data structures (DataFrame, Dataset) provide information about the structure of the data and the computation
- Supports execution of SQL queries
- Supports reading data from an existing database (Hive, MySQL, ...)

- The entry point is the `SparkSession` class
- `SparkSession spark = SparkSession.builder().appName("AppName").getOrCreate();`

DATAFRAME, DATASET

DATAFRAME

- Distributed collection of data, which is organized into named columns
- Conceptually equivalent to a table in a relational database
- Can be constructed from structured data files, external databases, existing RDDs, ...

```
Dataset<Row> dataFrame = spark.read().json("actors.json");
```

DATASET

- Distributed collection of data
- Can be constructed from strongly-typed JVM objects and manipulated using transformations
- Ability to use lambda functions

```
Dataset<Person> dataset = spark.read().json("actors.json").as(actorEncoder);
```

EXERCISE 4

- Find all actors who are younger than Ivan Trojan
  - Download project `ndbi040-actorsSpark` and data file `actors.json`
  - `Actor` class is already implemented
  - Create instance of `Encoder<Actor>`, i.e. `actorEncoder`
  - Read `Dataset<Actor>` from input json file, i.e. use `spark.read().json("path").as(actorEncoder)`
  - Call appropriate SQL query or use dataset methods `filter(...)`, `select(...)` and exploit function `col("column")`
  - Print result (only firstName and lastName) on output by calling dataset method `show()`

REFERENCES

- Apache Spark
  - https://spark.apache.org
- Quick Start
  - http://spark.apache.org/docs/latest/quick-start.html
- RDD Programming Guide
  - https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds
- Spark SQL, DataFrames and Datasets Guide
  - https://spark.apache.org/docs/latest/sql-getting-started.html
- Submitting Applications
  - https://spark.apache.org/docs/latest/submitting-applications.html
- Additional Spark Examples
  - https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples