MapReduce in Spark

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Bachelor studies, seventh semester
Academic year 2018/19 (winter semester)

Review of the previous lectures

- Mining of massive datasets.
- Evolution of database systems.
- Dimensional modeling.
- ETL and OLAP systems.
- Processing of massive datasets I
  - Physical storage and data access
  - Materialization, denormalization and summarization
- Processing of massive datasets II
  - Data partitioning
  - MapReduce:
    - The overall idea of the MapReduce paradigm.
    - WordCount and matrix-vector multiplication.
  - Spark: MapReduce in practice.

Outline

Outline

Algorithms in MapReduce

- How to implement fundamental algorithms in MapReduce?
  - Relational-Algebra Operations.
  - Matrix multiplication.

Outline

Relational-algebra operations

Example (Relation **Links**)

From	To
`url1`	`url2`
`url1`	`url3`
`url2`	`url3`
`url2`	`url4`
...	...

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)
- Operations:
  - Selection
  - Projection
  - Union, intersection, and difference
  - Natural join
  - Grouping and aggregation

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)
- Operations:
  - Selection
  - Projection
  - Union, intersection, and difference
  - Natural join
  - Grouping and aggregation
- Notation:
  - R, S - relation
  - t, t' - a tuple
  - $\mathcal{C}$ - a condition of selection
  - A, B, C - subset of attributes
  - a, b, c - attribute values for a given subset of attributes

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$
- **Map**:

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**:

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

Selection

- **Operation**: $\text{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

	Input	Output
map	`<k1, t>`	`list(<t, t>)`
reduce	`(<t, list(t)>)`	`list(<t, t>)`

Projection

- **Operation**: $\text{Project}_A(R)$

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**:

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**:

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Projection

- **Operation**: $\mathrm{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

	Input	Output
map	`<k1, t>`	`list(<t', t'>)`
reduce	`(<t', list(t',...,t')>)`	`list(<t', t'>)`

Union

- **Operation**: $\mathrm{Union}(R, S)$

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**:

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

Union

- **Operation**: $\mathrm{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

	Input	Output
map	`<k1, t)>`	`list(<t, t>)`
reduce	`(<t, list(t)>)` or `(<t, list(t,t)>)`	`list(<t, t>)`

Intersection

- **Operation**: $\mathrm{Intersection}(R, S)$

Intersection

- **Operation**: $\mathrm{Intersection}(R, S)$
- **Map**:

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

Intersection

- **Operation**: $\mathrm{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

	Input	Output
map	`<k1, t)>`	`list(<t, t>)`
reduce	`(<t, list(t)>)` or	`list(<t, t>)` if
	`(<t, list(t,t)>)`	`(<t, list(t,t)>)`

Minus

- **Operation**: $\mathrm{Minus}(R, S)$

Minus

- **Operation**: $\mathrm{Minus}(R, S)$
- **Map**:

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**:

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**: For each key t, do the following.
  1. If the associated value list is $[\texttt{name}(R)]$, then produce (t, t).
  2. If the associated value list is anything else, which could only be $[\texttt{name}(R), \texttt{name}(S)]$, $[\texttt{name}(S), \texttt{name}(R)]$, or $[\texttt{name}(S)]$, produce nothing.

Minus

- **Operation**: $\mathrm{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \mathtt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \mathtt{name}(S))$.
- **Reduce**: For each key t, do the following.
  1. If the associated value list is $[\mathtt{name}(R)]$, then produce (t, t).
  2. If the associated value list is anything else, which could only be $[\mathtt{name}(R), \mathtt{name}(S)]$, $[\mathtt{name}(S), \mathtt{name}(R)]$, or $[\mathtt{name}(S)]$, produce nothing.

	Input	Output
map	<k1, (t, R)> or	list(<t, R>) or
	<k1, (t, S)> or	list(<t, S>)
reduce	(<t, list(R)>) or	list(<t, t>) if
	(<t, list(S)>) or	(<t, list(R)>)
	(<t, list(R,S)>) or	
	(<t, list(S,R)>)	

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**:

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**:

Natural Join

- **Operation**: $\text{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**: Each key value b will be associated with a list of pairs that are either of the form $(\texttt{name}(R), a)$ or $(\texttt{name}(S), c)$. Construct all pairs consisting of one with first component $\texttt{name}(R)$ and the other with first component S, say $(\texttt{name}(R), a)$ and $(\texttt{name}(S), c)$. The output for key b is a list $(b, (a1, b, c1))$, $(b, (a2, b, c2))$, $\ldots$.

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\mathtt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\mathtt{name}(S), c))$.
- **Reduce**: Each key value b will be associated with a list of pairs that are either of the form $(\mathtt{name}(R), a)$ or $(\mathtt{name}(S), c)$. Construct all pairs consisting of one with first component $\mathtt{name}(R)$ and the other with first component S, say $(\mathtt{name}(R), a)$ and $(\mathtt{name}(S), c)$. The output for key b is a list $(b, (a1, b, c1))$, $(b, (a2, b, c2))$, $\ldots$.

	Input	Output
map	`<k1, (t, R)>` or	`list(<b, (a, R)>)` or
	`<k1, (t, S)>` or	`list(<b, (c, S)>)`
reduce	`<b, list((a1, R), ..., (c1,S), ...)>`	`list(<b, (ai, b, cj)>)`

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta, A, B)}(R)$

Grouping and Aggregation

- **Operation**: $\mathrm{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A,B,C)$ by attributes A and aggregate values of B by using function θ.
- **Map**:

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**:

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A,B,C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a,b,c) produce the key-value pair (a,b).
- **Reduce**: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a,x), where x is the result of applying θ to the list. For example, if θ is `SUM`, then $x = b_1 + b_2 + \ldots + b_n$, and if θ is `MAX`, then x is the largest of $b_1, b_2, \ldots, b_n$.

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is `SUM`, then $x = b_1 + b_2 + \ldots + b_n$, and if θ is `MAX`, then x is the largest of $b_1, b_2, \ldots, b_n$.

	Input	Output
map	`<k1, t>`	`list(<a, b>)`
reduce	`<a, list((b1, b2, ...)>`	`list(<a, f(b1, b2, ...)>)`

Outline

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

  $$P = MN$$

  is the matrix P with element p_{ik} in row i and column k, where:

  $$pik =$$

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

$$P = MN$$

  is the matrix P with element p_{ik} in row i and column k, where:

$$pik = \sum_{j} m_{ij} n_{jk}$$

Matrix Multiplication

- We can think of a matrix M and N as a relation with three attributes: the row number, the column number, and the value in that row and column, i.e.,:

$$M(I, J, V) \quad \text{and} \quad N(J, K, W)$$

with the following tuples, respectively:

$$(i, j, m_{ij}) \quad \text{and} \quad (j, k, n_{jk}).$$

- In case of sparsity of M and N, this relational representation is very efficient in terms of space.
- The product MN is almost a natural join followed by grouping and aggregation.

Matrix Multiplication

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

$$(j, (M, i, m_{ij})) \,.$$

  Analogously, send each matrix element n_{jk} to the key value pair:

$$(j, (N, k, n_{jk})) \,.$$

- **Reduce**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

$$(j, (M, i, m_{ij}))\,.$$

Analogously, send each matrix element n_{jk} to the key value pair:

$$(j, (N, k, n_{jk}))\,.$$

- **Reduce**: For each key j, examine its list of associated values. For each value that comes from M, say (M, i, m_{ij}), and each value that comes from N, say (N, k, n_{jk}), produce the tuple

$$(i, k, v = m_{ij}n_{jk}),$$

The output of the Reduce function is a key j paired with the list of all the tuples of this form that we get from j:

$$(j, [(i_1, k_1, v_1), (i_2, k_2, v_2), \ldots, (i_p, k_p, v_p)])\,.$$

Matrix Multiplication

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$((i_1, k_1), v_1), ((i_2, k_2), v_2), \ldots, ((i_p, k_p), v_p).$$

- **Reduce**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$((i_1, k_1), v_1), ((i_2, k_2), v_2), \ldots, ((i_p, k_p), v_p).$$

- **Reduce**: For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair

$$((i, k), v),$$

where v is the value of the element in row i and column k of the matrix

$$P = MN.$$

Matrix Multiplication with One Map-Reduce Step

- **Map**:

Matrix Multiplication with One Map-Reduce Step

- **Map**: For each element m_{ij} of M, produce a key-value pair

$$((i,k),(M,j,m_{ij}))\,,$$

for $k = 1, 2, \ldots$, up to the number of columns of N.
Also, for each element n_{jk} of N, produce a key-value pair

$$((i,k),(N,j,n_{jk}))\,,$$

for $i = 1, 2, \ldots$, up to the number of rows of M.

Matrix Multiplication with One Map-Reduce Step

- **Reduce**:

Matrix Multiplication with One Map-Reduce Step

- **Reduce**: Each key (i, k) will have an associated list with all the values

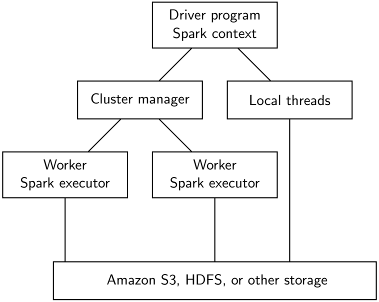
$$(M, j, m_{ij}) \quad \text{and} \quad (N, j, n_{jk}),$$

  for all possible values of j. We connect the two values on the list that have the same value of j, for each j:
  - We sort by j the values that begin with M and sort by j the values that begin with N, in separate lists,
  - The jth values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied,
  - Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

Outline

Programming in Spark

- Spark uses in-memory storage for storing immediate results, while Hadoop stores data on disk.
- A Spark program consists of two parts:
  - A driver program: runs on *your* machine.
  - Worker programs: run on cluster nodes or in local threads.
- A Spark program first creates a `SparkContext` object that tells how to access a cluster



Programming in Spark

- Three types of APIs:
  - RDD: an immutable collection of elements partitioned across the nodes of the cluster
  - Dataset: a strongly-typed, distributed and immutable collection of data that can benefit of the optimized execution engine.
  - Dataframe: an immutable distributed collection of data organized into named columns (implemented as Dataset of type `Row`).

Resilient Distributed Datasets

- RDDs are immutable, distributed, lazy, and compile-time type-safe based on Scala collections API
- They track lineage information to efficiently recompute lost data
- Enable operations on collection of elements in parallel
- Construction of RDD:
  - Parallelization of an existing collection in the driver program,
  - By transforming an existing RDDs,
  - From files in HDFS or any other storage system.
- The number of partitions is to be set by a programmer.

Programming in Spark

- Two types of operations:
  - Transformations: create a new dataset from an existing one in the lazy manner (do not run computations on data immediately).
  - Actions: return a value to the driver program after running a computation on the dataset or storing the results to the file system.

Transformations

- Transformations are recipes for creating a result
- 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 and slow workers
- Examples:
  - `map`, `flatMap`
  - `filter`
  - `distinct`
  - `union`, `intersection`
  - `join`, `cartesian`
  - `reduceByKey`, `groupByKey`, `sortByKey` ($\Leftarrow$ MapReduce-style operations working on pair RDDs)

Actions

- Actions cause Spark to execute recipe to transform input data.
- Examples:
  - `reduce`,
  - `collect`
  - `count`
  - `first`, `take(1)`, `take(n)`
  - `saveAsTextFile`, `saveAsSequenceFile`
  - `countByKey`
  - `foreach(func)`

Caching of results

- 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 that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).
- To persist RDD use the `persist()` or `cache()` methods on it.

```
val textFile = sc.textFile("~/data/all-shakespeare.txt")
textFile.count()
textFile.count()
textFile.cache()
textFile.count()
textFile.count()
```

Lifecycle of Spark Program

- Create RDDs from external data or parallelize a collection in your driver program,
- Lazily transform them into new RDDs,
- Cache some RDDs for reuse.
- Perform actions to execute parallel computation and produce results.

Closure

- Spark automatically creates closures for:
  - Functions that run on RDDs at workers
  - Any global variables used by those workers
- One closure per worker
  - Sent for every task
  - No communication between workers
  - Changes to global variables at workers are not sent to driver

Shared Variables

- Broadcast Variables
  - Efficiently send large, read-only value to all workers
  - Saved at workers for use in one or more Spark operations
  - Like sending a large, read-only lookup table to all the nodes
- Example:

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]]
    = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Shared Variables

- Accumulators
  - Aggregate values from workers back to driver
  - Only driver can access value of accumulator
  - For tasks, accumulators are write-only
  - Use to count errors seen in RDD across workers
- Example:

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator
    (id: 0, name: Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.
    add(x))
...

scala> accum.value
res2: Long = 10
```

Let us check some code

- Word count I:

```
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = (textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _))
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Let us check some code

- Matrix-vector multiplication:

```
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.
    split(","); (t(0).trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t =
    line.split(","); (t(0).trim.toInt, t(1).trim.toInt, t(2).
    trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.
    value(j-1))}.reduceByKey(_ + _)
v.toDF.orderBy("_1").show
```

Dataframes and Datasets

- Alternative for RDDs
- Rather *What* than *How* programming style $\Rightarrow$ More optimizations possible
- Datasets are strongly typed, but Dataframes not.
- They use the `SqlContext`.
- SQL-like queries: either from Scala or SQL.

Dataframes and Datasets

- A sample code:

```
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

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

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

- One can also use SQL directly:

```
df.createOrReplaceTempView("people")

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

Dataframes and Datasets

- Creating dataframes and datasets:

```scala
case class Person(name: String, age: Long)

val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]

val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// Create an RDD of Person objects from a text file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
```

Dataframes and Datasets

- Data sources:
  - The default data source is `parquet`, which is highly efficient columnar format.
  - Other data sources are also supported like `json`, databases via `jdbc`, `hive` databases, and many others.

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")

val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.
    json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

- For file-based data source, it is also possible to bucket and sort or partition the output.

```
peopleDF
  .write
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("people_partitioned_bucketed")
```

Let us check some code

- Dataframes and Datasets:

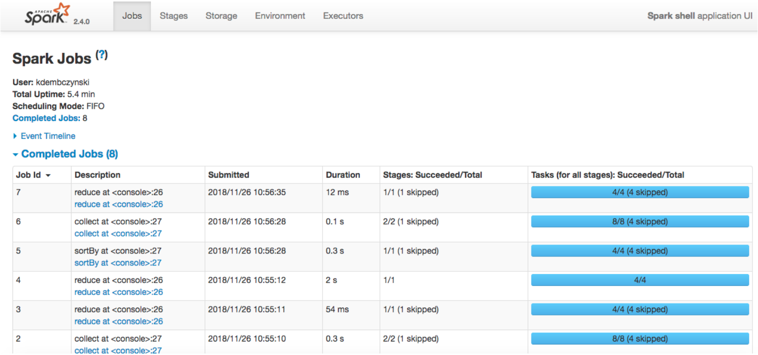
```
val songs = spark.read.
                   option("delimiter", ",").
                   csv("songs").
                   toDF("song_id", "track_long_id", "song_long_id", "artist", "song"
     )

val facts = spark.read.
                  option("delimiter", ",").
                  csv("facts").
                  toDF("id", "user_id", "song_id", "date_id")

facts.groupBy("song_id").
      count.
      join(songs, facts("song_id")===songs("song_id")).
      select("song", "count").
      orderBy(desc("count")).
      show(10)
```

Monitoring Spark

- Every SparkContext launches a web UI, by default on port 4040.
- It displays useful information about the application:
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Environmental information
  - Information about the running executors
- To access the interface, you can open in your web browser the following page:

  `http://<driver-node>:4040` (e.g. `http://localhost:4040`)
- If multiple SparkContexts are running on the same host, they will bind to successive ports beginning with 4040 (4041, 4042, etc).

Monitoring Spark


Spark 2.4.0 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Spark Jobs (?)

User: kdembczynski
Total Uptime: 5.4 min
Scheduling Mode: FIFO
Completed Jobs: 8

▸ Event Timeline

▾ **Completed Jobs (8)**

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	reduce at <console>:26 reduce at <console>:26	2018/11/26 10:56:35	12 ms	1/1 (1 skipped)	4/4 (4 skipped)
6	collect at <console>:27 collect at <console>:27	2018/11/26 10:56:28	0.1 s	2/2 (1 skipped)	8/8 (4 skipped)
5	sortBy at <console>:27 sortBy at <console>:27	2018/11/26 10:56:28	0.3 s	1/1 (1 skipped)	4/4 (4 skipped)
4	reduce at <console>:26 reduce at <console>:26	2018/11/26 10:55:12	2 s	1/1	4/4
3	reduce at <console>:26 reduce at <console>:26	2018/11/26 10:55:11	54 ms	1/1 (1 skipped)	4/4 (4 skipped)
2	collect at <console>:27 collect at <console>:27	2018/11/26 10:55:10	0.3 s	2/2 (1 skipped)	8/8 (4 skipped)

Aggregation functions

- distributive: `count()`, `sum`, `max`, `min`,
- algebraic: `ave()`, `stdev`, `var`,
- holistic: `median`, `rank`, `mode`, `distinct count`.

Outline

Summary

- Algorithms in MapReduce:
  - Relational-algebra operations.
  - Matrix multiplication.
- Programming in Spark

Bibliography

- J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014 `http://infolab.stanford.edu/~ullman/mmds.html`
- J.Lin and Ch. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010 `http://lintool.github.com/MapReduceAlgorithms/`
- Ch. Lam. *Hadoop in Action*. Manning Publications Co., 2011
- `https://spark.apache.org/docs/`
- Anthony D. Joseph. Introduction to Big Data with Apache Spark, 2016