Distributed Data Management

Summer Semester 2017

TU Kaiserslautern

Prof. Dr.-Ing. Sebastian Michel

Databases and Information Systems Group (AG DBIS)

http://dbis.informatik.uni-kl.de/

PIG AND HIVE

TWO HIGHER-LEVEL APPROACHES TO PROCESS DATA WITH MAPREDUCE

MapReduce

- Remember slides on pros and cons of MapReduce, particularly criticism (too low level, …)
- We have seen how to code joins in MR
- How to filter (grep!), group by, …

- Now: brief look at higher-level ”tools” on top of MapReduce
- Why? Claim: MapReduce too low level for “normal” users (developers) + large effort for ad-hoc queries.

Pig & Pig Latin

- High-level tool for expressing data analysis programs, originated from Yahoo (now at Apache)
- Compiler transforms query into sequence of MapReduce jobs
- Data Flow language, **Pig Latin** (not really something like SQL)
- http://pig.apache.org

Gates et al. Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. PVLDB 2(2): 1414-1425 (2009)

Relation Pig and Hadoop

Pig Latin Commands:

```
A = LOAD 'input' AS (x, y, z);
B = FILTER A BY x > 5;
STORE B INTO 'output';
```

Parsing, logical optimization.

Creation of MapReduce jobs
+ running them.

Hadoop MapReduce

Example

Input, e.g., using Shell:

grunt> ….

Commands like:

```
A = LOAD 'input' AS (x, y, z);
B = FILTER A BY x > 5;
STORE B INTO 'output';
```

Pig operates directly over files (and other sources, if specified by user defined functions (UDFs)).

(Nested) Data Model

- **Atom:**
  - int, double, chararray, etc.
  - E.g., ‘Distributed Data Management’, ‘Michel’
- **Tuple:**
  - sequence of fields (any types) (…, …, …, …)
  - E.g., (‘Distributed Data Management’, 2017, {(1,2,3)})
- **Bag:**
  - collection of tuples (multiset, i.e., can have duplicates)
  - E.g., {(‘DDM17’, ‘Infosys17’)}
- **Map:**
  - Mapping of keys to values
  - E.g., {‘Michel’ => {‘DDM17’}, ‘Deßloch’=>{‘Infosys17’}}

Violates First Normal Form of traditional RDMBS

Pig Latin: Example: Joins

- A

  (2,Tie)

  (4,Coat)

  (3,Hat)

  (1,Scarf)

- B

  (Joe,2)

  (Hank,4)

  (Ali,0)

  (Eve,3)

  (Hank,2)

A = LOAD ……; B = LOAD …..

C=Join A BY $0, B BY $1

Also support for OUTER JOINS

Data with Associated Schema

PEOPLE = LOAD 'hdfs:///user/hduser/testjoin/people.txt' **as (id: int, name: chararray);**

PARTS= LOAD 'hdfs:///user/hduser/testjoin/parts.txt' **as (name: chararray, partsid: int);**

Pig Latin: Commands (Subset)

- LOAD, STORE, DUMP
- FILTER
- FLATTEN
- FOR EACH
- GENERATE
- (CO)GROUP
- CROSS
- JOIN
- ORDER BY
- LIMIT

PLUS: Built-in and user-defined functions (UDFs)

http://wiki.apache.org/pig/PigLatin

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110

Example: Word Count

```
    //LOAD input file from HDFS
A = LOAD 'hdfs:///user/hduser/gutenberg' AS (line : chararray);
    //Parse input lines into words
B = FOREACH A GENERATE FLATTEN(TOKENIZE(line)) as term;
    //Remove whitespace-only words
C = FILTER B BY term MATCHES '\\w+';
    //Group by term
D = GROUP C BY term;
    //and count for each group (i.e., for a term) its occurrences
E = FOREACH D GENERATE group, COUNT($1) as frequency;
    //ORDER by frequency of occurrence
F = ORDER E BY frequency ASC;
```

Example: Word Count (Cont'd)

Output:

.....

....

(which,2475)

(it,2553)

(that,2715)

(a,3813)

(is,4178)

(to,5070)

(in,5236)

(and,7666)

(of,10394)

(the,20592)

2013-05-15 10:02:21,062 [main] INFO org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MultiQueryOptimizer - MR plan size after optimization: 3

.....

Counters:

Total records written : 17875

Total bytes written : 178274

…

Job DAG:

job_201305031236_0051 -> job_201305031236_0052,

job_201305031236_0052 -> job_201305031236_0053,

job_201305031236_0053

Logically, multiple connected MapReduce jobs form a DAG*

*) DAG = Directed Acyclic Graph

(CO)GROUP Example

- Consider the following data (as CSV input)

owners.csv

```
adam,cat
adam,dog
alex,fish
alice,cat
steve,dog
```

- And the following PIG script

```
owners = LOAD 'owners.csv'
   USING PigStorage(',')
   AS (owner:chararray,animal:chararray);

grouped = COGROUP owners BY animal;
DUMP grouped;
```

This returns a list of animals. For each animal, Pig groups the matching rows into bags

group	owners
cat	{(adam,cat),(alice,cat)}
dog	{(adam,dog),(steve,dog)}
fish	{(alex,fish)}

http://joshualande.com/cogroup-in-pig/

(CO)GROUP of Two Tables

owners.csv

```
adam,cat
adam,dog
alex,fish
alice,cat
steve,dog
```

pets.csv

```
nemo,fish
fido,dog
rex,dog
paws,cat
wiskers,cat
```

```
owners = LOAD 'owners.csv'
   USING PigStorage(',')
   AS (owner:chararray,animal:chararray);

pets = LOAD 'pets.csv'
   USING PigStorage(',')
   AS (name:chararray,animal:chararray);

grouped = COGROUP owners BY animal,
                  pets BY animal;
DUMP grouped;
```

What is the difference to a Join?

group	owners	pets
cat	{(adam,cat),(alice,cat)}	{(paws,cat),(wiskers,cat)}
dog	{(adam,dog),(steve,dog)}	{(fido,dog),(rex,dog)}
fish	{(alex,fish)}	{(nemo,fish)}

User Defined Functions in PIG

- Can write your custom UDF in Java and directly use it in PIG
- Here for a simple “eval” function:

```
REGISTER myudfs.jar;
A = LOAD 'student_data' AS
      (name:chararray, age: int, gpa: float);
B = FOREACH A GENERATE
                  myudfs.UPPER(name);
……
```

https://wiki.apache.org/pig/UDFManual

```
public class UPPER extends EvalFunc<String>
{
  public String exec(Tuple input) throws
IOException {
     if (input == null || input.size() == 0)
        return null;
      try{
        String str = (String)input.get(0);
       return str.toUpperCase();
      }catch(Exception e){
        throw new IOException("Caught
exception processing input row ", e);
      }
  }
}
```

Optimizations

- **Logical Optimization**:
  - Filter as early as possible
  - Eliminate unnecessary information (project)
  - …
- **Multiple MapReduce jobs** (in general, not only here in Pig) give possibilities to optimize execution order.
- Considering DAG dependencies!
- Reusing stored outputs of previous

Pig vs. Native MapReduce

Two sides of the coin (generally). Statement from Twitter engineer in 2009.

"…typically a **Pig script is 5% of the code** of native map/reduce **written in about 5% of the time**. "

"**However**, queries typically take **between 110-150% the time to execute** that a native map/reduce job would have taken."

http://blog.tonybain.com/tony_bain/2009/11/analytics-at-twitter.html

Pig Latin vs. SQL

- **Pig Latin** is a data flow programming language
  - user specified operation(s) put together to achieve task (imperative)
- **SQL** is declarative
  - user specifies what the result should be, not how it is implemented

Pig vs. RDBMS

- **RDBMS:**
  - tables with predefined schema
  - support of transactions and indices
  - aim at fast response time
- **Pig:**
  - schema at runtime (even optional)
  - any source (by applying user defined functions)
  - no loading/indexing of data as pre-processing: data is loaded at execution time (usually from HDFS)
  - like MapReduce (well, Pig is build on top of MR): aim at throughput, not super fast short queries

Hive

- For structured data
- On top of Hadoop (like Pig) and, hence, HDFS

- "**RDBMS for big data**"
- Query language is similar to SQL (declarative) (not a data flow language as Pig Latin)

- Originated from Facebook's effort to analyze their data.
- Now, an Apache Project

Hive QL

SELECT year, MAX(temperature)

FROM records

WHERE temperature != 9999 AND …..

GROUP BY year;

No full support of SQL-92 standard.

Note: There are various other projects (specifically at Apache) for big data management for various purposes. Have a closer look if you are interested!

Literature

- Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, Utkarsh Srivastava: Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. PVLDB 2(2): 1414-1425 (2009)

- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110

- http://pig.apache.org
- http://wiki.apache.org/pig/PigLatin

- http://hive.apache.org/

Summary MapReduce

- **Programming paradigm** and **infrastructure** for **processing large amounts of data in a batch fashion.**
- **Two functions, map and reduce** describe how data is processed and aggregated.
- Have seen multiple **application scenarios and corresponding algorithms**.
- MR jobs can be connected to workflows
- **PIG** is one way to automatically translate higher-level instructions to MR jobs

SPARK

http://spark.apache.org/

Apache Spark

- Apache Spark is a(-nother) **distributed, "Big Data" processing framework**.
- Everything you can do in Hadoop, you can also do in Spark.
- In contrast to Hadoop, its computation paradigm is not MapReduce, but a **multi-stage, in-memory model** based on **Resilient Distributed Datasets** (RDDs).
- It is written in **Scala**, running on a **Java Virtual Machine**.
- Spark supports different languages for application development: Java, Scala, Python, R, and SQL.
- Originally from AMPLab (UC Berkeley, 2009) . Since 2013, Apache Software Foundation.

Spark Main Features

- Spark is **expected to gradually replace Hadoop** for its improved flexibility and performance (**reported to be up to 100 times faster than Hadoop for certain algorithms**).

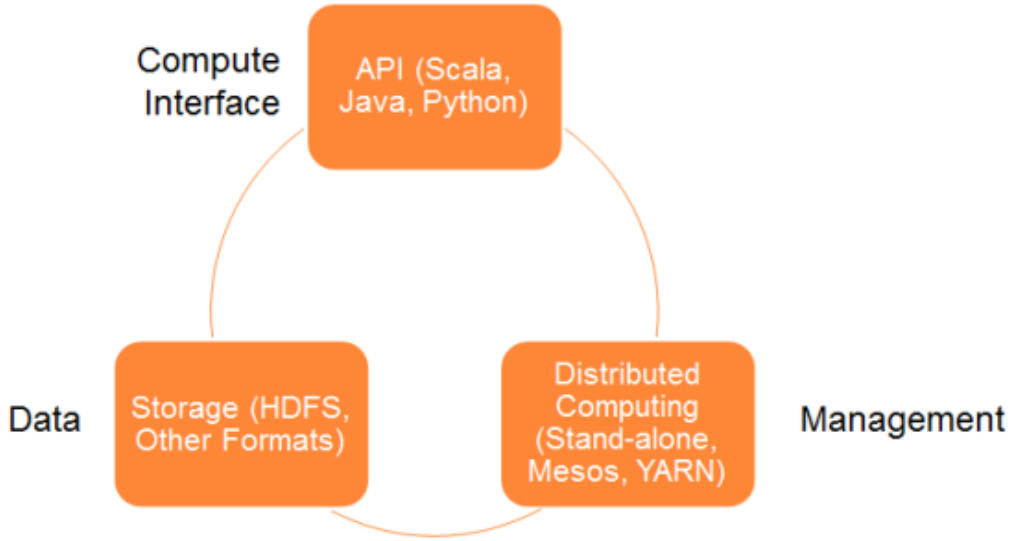
- The Spark project integrates an ecosystem providing both **low-level API's and higher-level libraries** (SQL, streaming, machine learning, graph processing, etc.).

- Although a Hadoop installation is not needed, Spark can use Hadoop's distributed storage and management components (HDFS and YARN for Hadoop v2.x).

- **Spark is known to scale well to clusters of several thousands of nodes**, with up to petabytes in terms of data size.

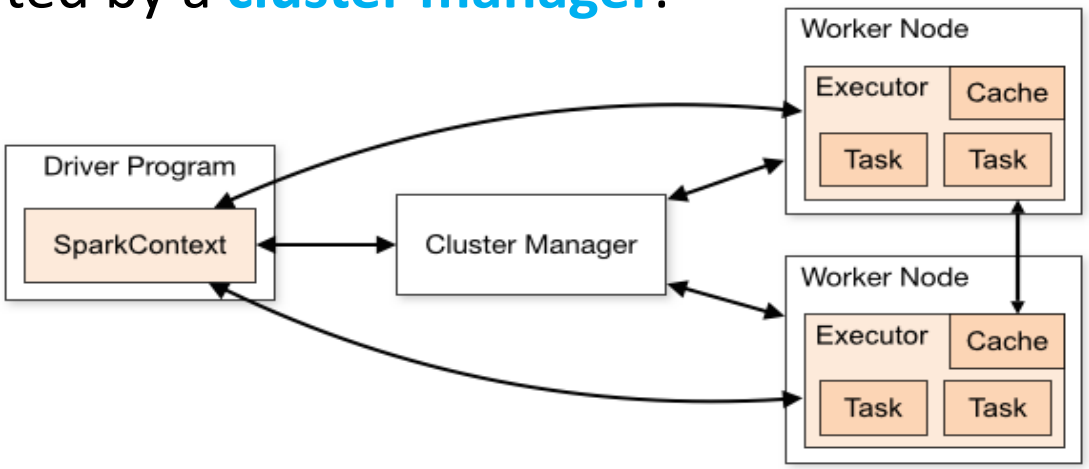
Spark Architecture

- An API with core features and extended modules for data manipulation in different languages: Java, Scala, Python, R.
- Spark supports any Hadoop-ready storage source: local file system, HDFS, HBase, Amazon S3, Cassandra, ...
- Although Spark provides a standalone cluster manager, both Apache YARN and Apache Mesos are also supported.



Cluster Manager

- Spark runs as a set of independent processes on a cluster that is coordinated by a **cluster manager**.

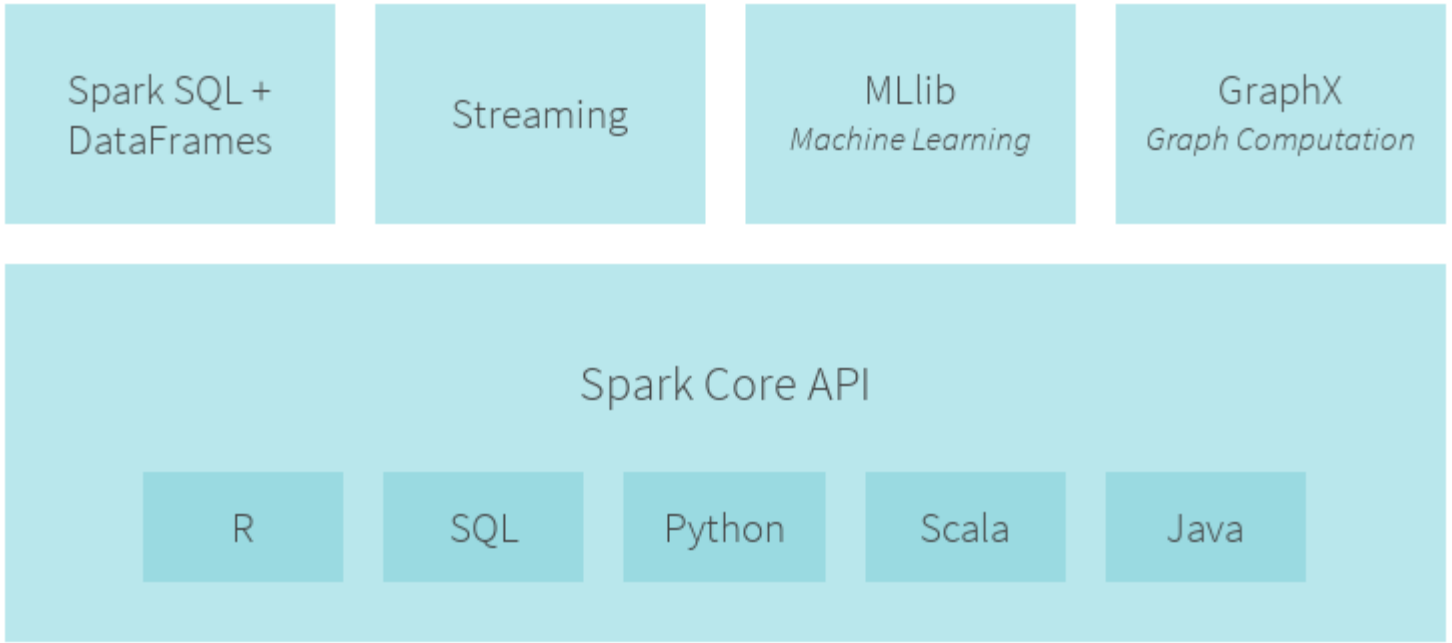


- The **SparkContext** object in the *driver program* of Spark requests the needed resources from the cluster manager, by acquiring *executors* in each machine.

- The application code is sent to the *executors*, which perform the *tasks* assigned by the **SparkContext** object in multiple threads.

Main Building Blocks

- The **Spark Core API** provides the general execution engine on top of which all other functionality is built upon.
- Four higher-level components (in the "Spark ecosystem"): **Spark SQL** (formerly "Shark"), **Streaming**, **MLlib** and **GraphX**.



Spark Core API

- The **Spark Core API** provides the main functionalities: distributed task scheduling and dispatching, basic I/O libraries and interfaces.
- Its basic programming abstraction is the **Resilient Distributed Dataset (RDD)** (**distributed, in-memory data structures, similar to a large in-memory cache**).
- Unlike Hadoop/MapReduce, Spark allows for the development of complex, **multi-step computation pipelines** using a directed acyclic graph (DAG) pattern.
- It supports **in-memory data sharing across DAGs**, allowing different jobs to work on the same data and in parallel.

Resilient Distributed Dataset (RDD)

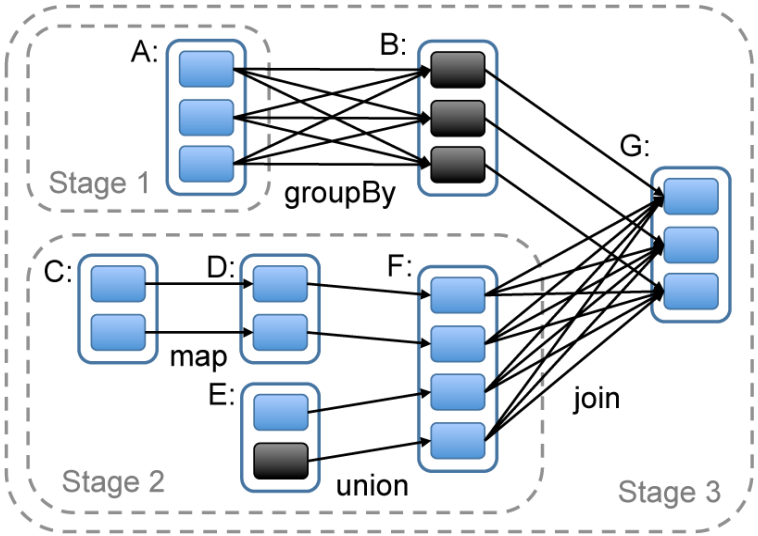
- A **Resilient Distributed Dataset** (RDD) is an immutable collection of data items which are partitioned and distributed (i.e., "sharded") across the cluster nodes' memory, thus allowing for (1) **parallel processing** and (2) **reuse across multiple processing steps** in a computation pipeline.
- They can be **created by referencing a dataset from an external storage system**, **or by explicitly distributing a collection of objects in the driver program**.
- The Spark Core **API provides primitives to operate on the RDDs**.
- They may consist of simple data structures (maps, arrays) or contain more complex Java, Scala, or Python objects.

RDD Operations

- The Spark Core API offers **two types of operations on RDDs**:

- **Transformations**, which each generate a new RDD from an existing one: map(), union(), filter(), join(), cartesian(), …

- **Actions**, which return a result that is computed from an existing RDD: reduce(), count(), first(), collect(), saveAsTextFile(), …



Example: WordCount in Spark w/ Java

```
public final class JavaWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) throws Exception {
    SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount");
    JavaSparkContext ctx = new JavaSparkContext(sparkConf);

    JavaRDD<String> lines = ctx.textFile("hdfs://entire_internet.txt", 1);

    JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

     public Iterable<String> call(String s) {
         return Arrays.asList(SPACE.split(s));}});

    JavaPairRDD<String, Integer> ones = words.mapToPair(new PairFunction<String,
         String, Integer>() {

         public Tuple2<String, Integer> call(String s) {
        return new Tuple2<String, Integer>(s, 1);}});

    JavaPairRDD<String, Integer> counts = ones.reduceByKey(new Function2<Integer,
         Integer, Integer>() {

         public Integer call(Integer i1, Integer i2) {
           return i1 + i2;}});

    counts.saveAsTextFile("hdfs://result.txt");
    ctx.stop();
  }}
```

Example: WordCount in Spark w/ Scala

```
val textFile = spark.textFile("hdfs://entire_internet.txt")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://result.txt")
```

Example JavaRDD of TPC-H Data

```
JavaRDD<Customer> customers = context.textFile("/data/customer.tbl").map(
    new Function<String, Customer>() {
      @Override
      public Customer call(String line) {

        StringTokenizer tok = new StringTokenizer(line, "|");
        Customer customer = new Customer();
        customer.setC_custkey(tok.nextToken());
        customer.setC_name(tok.nextToken());
        customer.setC_address(tok.nextToken());
        customer.setC_nationkey(tok.nextToken());
        customer.setC_phone(tok.nextToken());
        customer.setC_acctbal(tok.nextToken());
        customer.setC_mktsegment(tok.nextToken());
        customer.setC_comment(tok.nextToken());
        return customer;
      }
    });
```

Example (Cont‘d)

```
Function<Customer, Boolean> mkt_filter = new Function<Customer,Boolean>(){
      public Boolean call(Customer c) {
            return c.getC_mktsegment().equals("AUTOMOBILE");
      }
   };

JavaRDD<Customer> customers_mktsegment = customers.filter(mkt_filter);
```

Via Spark Shell

```
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-1494493404108
).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

```
val file = sc.textFile("/tmp/data")
val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
counts.saveAsTextFile("/tmp/wordcount")
```

DataFrames

- A DataFrame is a *Dataset* organized into named columns. It is conceptually equivalent to a table in a relational database.

```
JavaRDD<Customer> customers = …..

SQLContext sqlContext = new SQLContext(context);
DataFrame dfCustomers = sqlContext.createDataFrame(customers, Customer.class);
dfCustomers.registerTempTable("customers");

DataFrame queryResults = sqlContext.sql(“SELECT …. FROM …. WHERE….“)
```

http://spark.apache.org/docs/latest/sql-programming-guide.html

RDD Persistence

- By default, an RDD is computed when it is needed.
- If two methods need the same RDD, it is computed twice.
- **Persisting an RDD** avoids that:
  - myrdd.**persist**(…) aka. Myrdd.cache()
- "Caching is a key tool for iterative algorithms and fast interactive use."

https://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

RDD Partitioning

- **The initial RDD partitions depend on the input format of the data source**. For example, large files loaded from HDFS will have a 64 MB partition size.
- RDDs of key/value pairs (**JavaPairRDD** objects) can be automatically partitioned on their key by using *hashing*- or *sorting-based partitioning* methods.

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[id, name]("hdfs://users.tsv")
  .partitionBy(new HashPartitioner(100)) // Create 100 partitions
  .cache()
```

- Every transformation generates a new RDD; it is possible to cache only the partitioned RDD and keep the input RDD (which then becomes obsolete) in non-persistent mode.

Fault Tolerance

- **Fault Tolerance** achieved through **Lineage** Tracking and
- Rebuilding of lost RDDs based on that.

- Lineage means:
  - List of **dependencies** on parent RDDs is tracked
  - And how RDD is computed (which transformation/action)