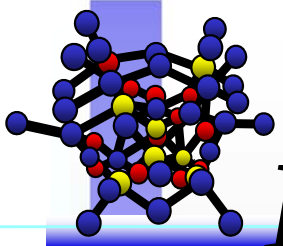


Knowledge Management and Protection Systems (KMaPS)

Corso di Laurea in Ingegneria

Part 10 – *Hadoop and Applications*

Gianni Pantaleo, Paolo Nesi

DISIT Lab http://www.disit.dinfo.unifi.it/

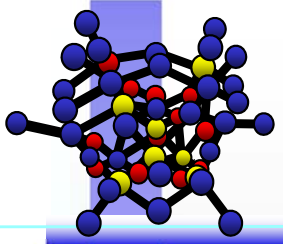
Department of Information Engineering, DINFO

University of Florence

Via S. Marta 3, 50139, Firenze, Italy

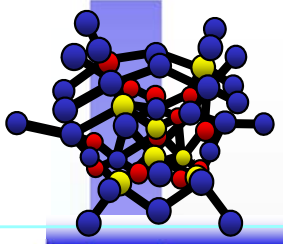
tel: +39-055-2758515, fax: +39-055-2758570

paolo.nesi@unifi.it, http://www.disit.dinfo.unifi.it/nesi



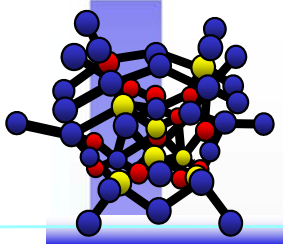
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



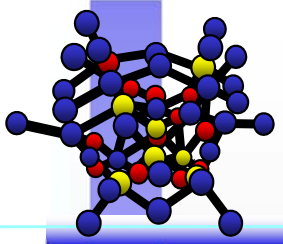
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



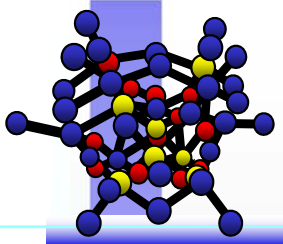
Problem Scope

- Large Scale ("*Web Scale*") data processing
  - Order of hundreds of Gigabytes to Terabytes or Petabytes
  - At this scale, input data set is not likely to even fit on a ***single*** computer's hard drive, much less in memory…
- Parallel architectures, ***multi***-machine clusters rises data failure probability.



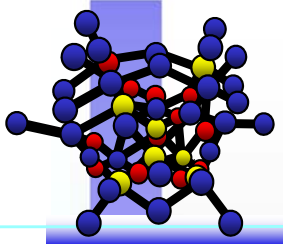
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



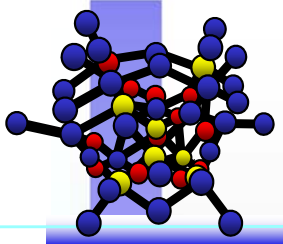
Hadoop

- Hadoop is an **open source framework** for processing, storing and analyzing massive amounts of distributed, unstructured data.
- It is designed to **scale up from a single server to thousands of machines**, with a very high degree of **fault tolerance**.
- Hadoop goal is to **scan large data set to produce results** through a **distribute** and **highly scalable** batch processing **systems**.



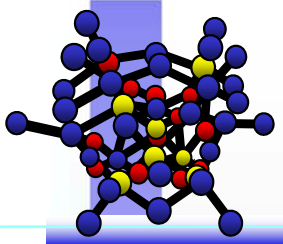
Hadoop

- Hadoop was inspired by MapReduce, a user-defined function developed by Google in early 2000s for indexing the Web.
- Hadoop is now an Open Source project of the Apache Software Foundation, where hundreds of contributors continuously improve the core technology;
- **Key Idea**: Rather than banging away at one, huge block of data with a single machine, Hadoop breaks up Big Data into multiple parts so each part can be processed and analyzed at the same time.



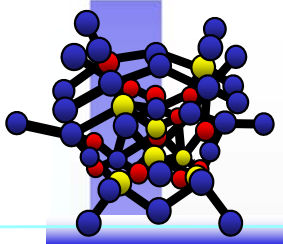
Hadoop

- Hadoop changes the economics and the dynamics of large scale computing, defining a processing solution that is:
  - **Scalable –** New nodes can be added as needed, and added without needing to change data formats, how data is loaded, how jobs are written, or the applications on top.
  - **Cost effective –** Hadoop brings massively parallel computing to commodity servers. The result is a sizeable decrease in the cost per terabyte of storage, which in turn makes it affordable to model all data.
  - **Flexible –** Hadoop is schema-less, and can absorb any type of data, structured or not, from any number of sources. Data from multiple sources can be joined and aggregated in arbitrary ways enabling deeper analyses than any one system can provide.
  - **Fault tolerant –** When you lose a node, the system redirects work to another location of the data and continues processing without missing a beat.



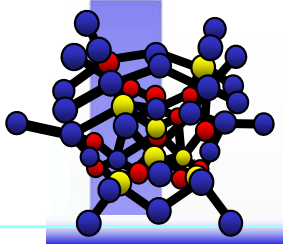
Typical problems

- Large number of nodes in the clusters
- Relevant probability of failures:
  - CPU: hundreds of thousands of nodes are present
  - Network: congestion may lead to do not provide data results and data inputs in times
  - Network: failure of apparatus
  - Memory: run out of space
  - Storage: run out of space, failure of a node, data corruption, failure of transmission
  - Clock: lack of synchronization, locked files and records not released, atomic transactions may lose connections and consistency
- Specific parallel solutions provide support for recovering from some of these failures



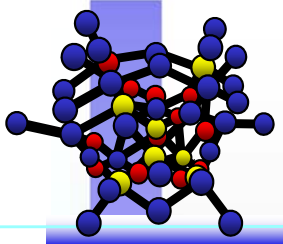
Typical problems

- Hadoop has no security model, nor safeguards against maliciously inserted data.
  - It cannot detect a man-in-the-middle attack between nodes
  - it is designed to handle very robustly:
    - hardware failure
    - data congestion issues
- Storage may be locally become full:
  - Reroute, redistribute mechanisms are needed
  - Synchronization among different nodes is needed
  - This may cause:
    - network saturation
    - Deadlock in data exchange



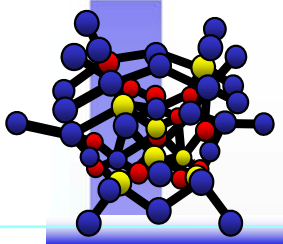
Recovering from failure

- If some node fails in providing results in time, the other nodes should do the work of the missing node
- The recovering process should be performed automatically
- The Solution may be not simple to implement in non parallel architectures, topologies, data distribution, etc.
- Limits:
  - Individual hard drives can only sustain read speeds between 60-100 MB/second
  - assuming four independent I/O channels are available to the machine, that provides 400 MB of data every second



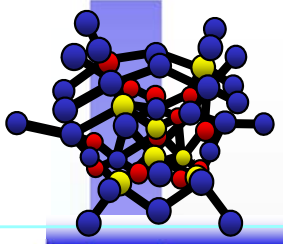
Processes vs Data

- Processes works on specific chunks of data. The allocations of processes depend on the position of the chunks they need to access,
- That is: data locality.
  - This minimize the data flow among nodes
  - Avoid communications among nodes to serve computational nodes
  - Hadoop is grounded on moving computation to the data !!!
    And **not** data to computation.



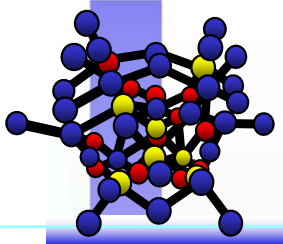
Hadoop Flat Scalability

- Hadoop on a limited amount of data on a small number of nodes may not demonstrate particularly stellar performance as the overhead involved in starting Hadoop programs is relatively high.
- parallel/distributed programming paradigms such as MPI (Message Passing Interface) may perform much better on two, four, or perhaps a dozen machines.
- specifically designed to have a very flat scalability curve
- If it is written for 10 nodes may work on thousands with small rework effort



Index

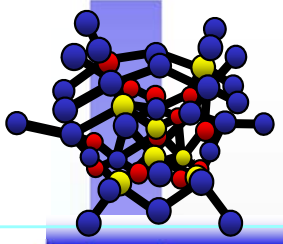
- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



Hadoop Architecture

- Apache Hadoop has a distributed master-slave architecture consisting of two main components:
  - **HDFS** Hadoop Distributed File System (HDFS) for storage

  - **MapReduce** paradigm for Computational capabilities

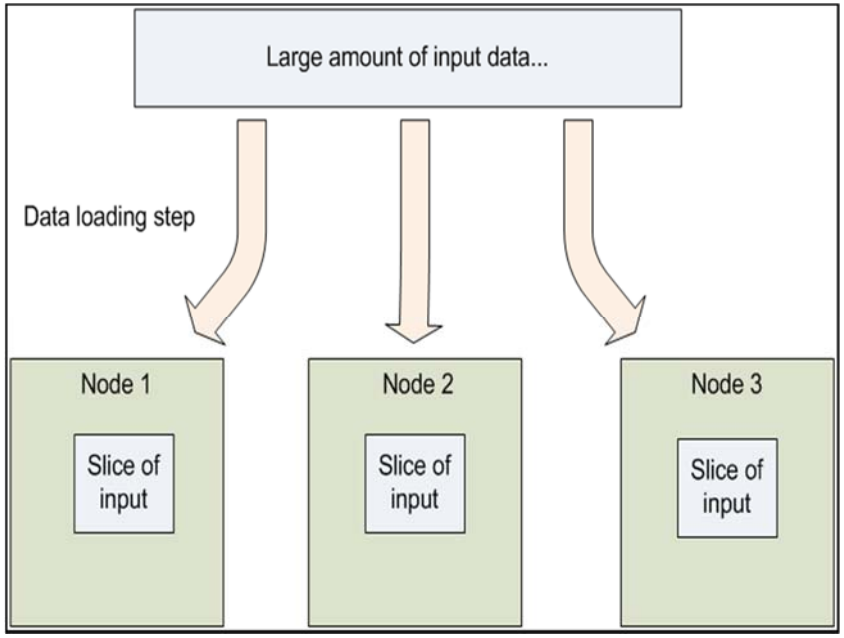


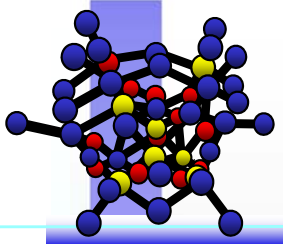
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API

Hadoop HDFS

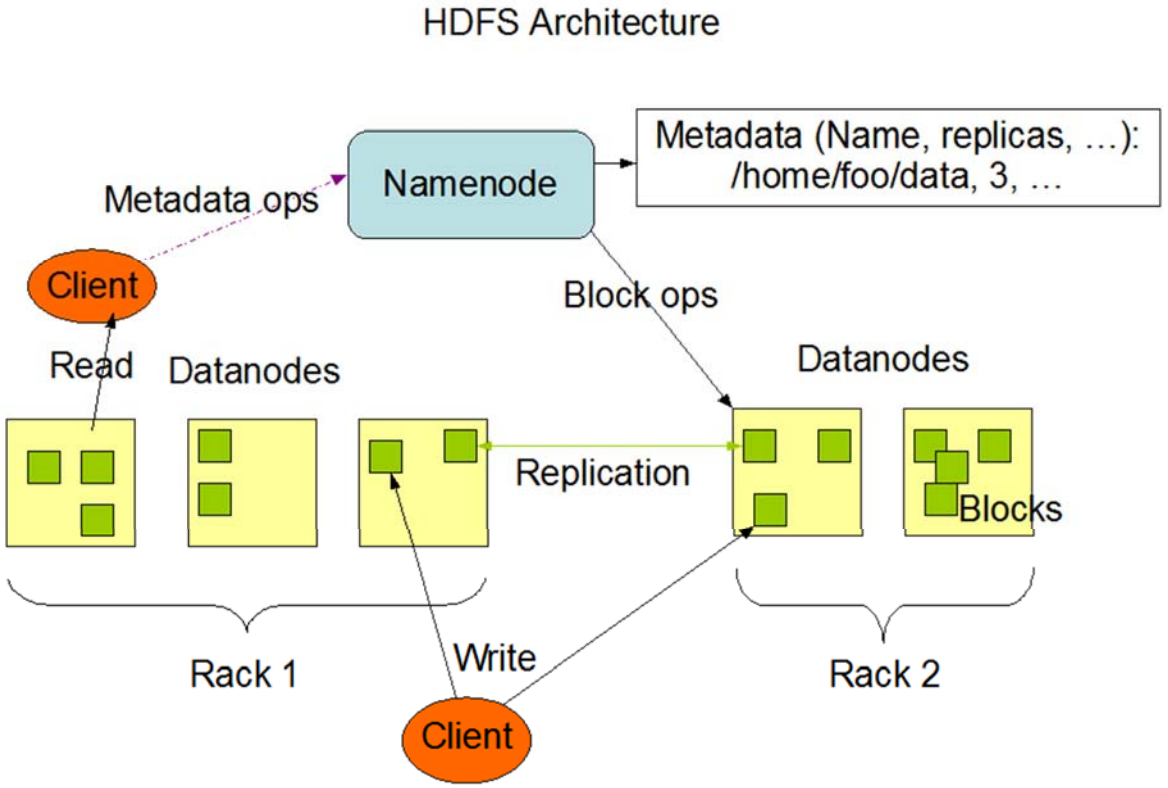
- Based on the Hadoop Distributed File System (HDFS) that distribute data files on large chunks on the different cluster nodes
- HDFS assumes nodes will fail, so it achieves reliability by ***replicating data chunks across multiple nodes***, thus supporting the failure of nodes.
  - An active process maintains the replications when failures occurs and when new data are stored.
  - Replicas are not instantly maintained aligned !!!

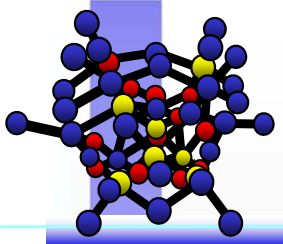




Hadoop HDFS

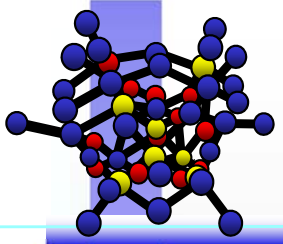
- **HDFS** file system spans all the nodes in a Hadoop cluster for data storage - default block size in **HDFS** is **64MB**
- It links together the file systems on many local nodes to make them into one big file system.





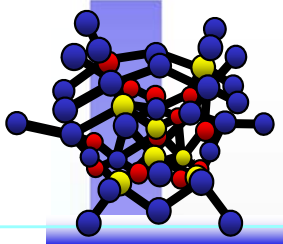
Hadoop Cluster Stack

- A Hadoop "stack" is made up of a number of components. They include:
  - **Hadoop Distributed File System** (HDFS): The default storage layer in any Hadoop cluster.
  - **Name Node**: The node in a Hadoop cluster that provides the client information on where in the cluster particular data is stored and if any nodes fail;
  - **Secondary Name Node**: A backup to the Name Node, it periodically replicates and stores data from the Name Node if it fails; it provides a sort of Name Node checkpoint management service.
  - **Job Tracker**: The node in a Hadoop cluster that initiates and coordinates MapReduce jobs and data processing tasks.
  - **Data (slave) Nodes**: Data Nodes store data and take directions to process it from the Job Tracker.



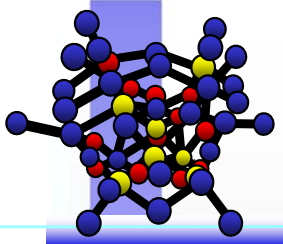
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



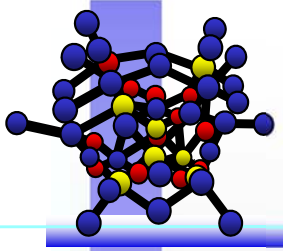
Hadoop MapReduce

- The main idea:
  - Each individual record is processed by a task in isolation from one another. Replications allow to reduce communications for: contour conditions, data overlap, etc.
- This approach ***does not allow*** any program to be executed.
- Algorithms ***have to be converted into parallel implementations*** to be executed on cluster of nodes.
- This programming model is called ***MapReduce model***

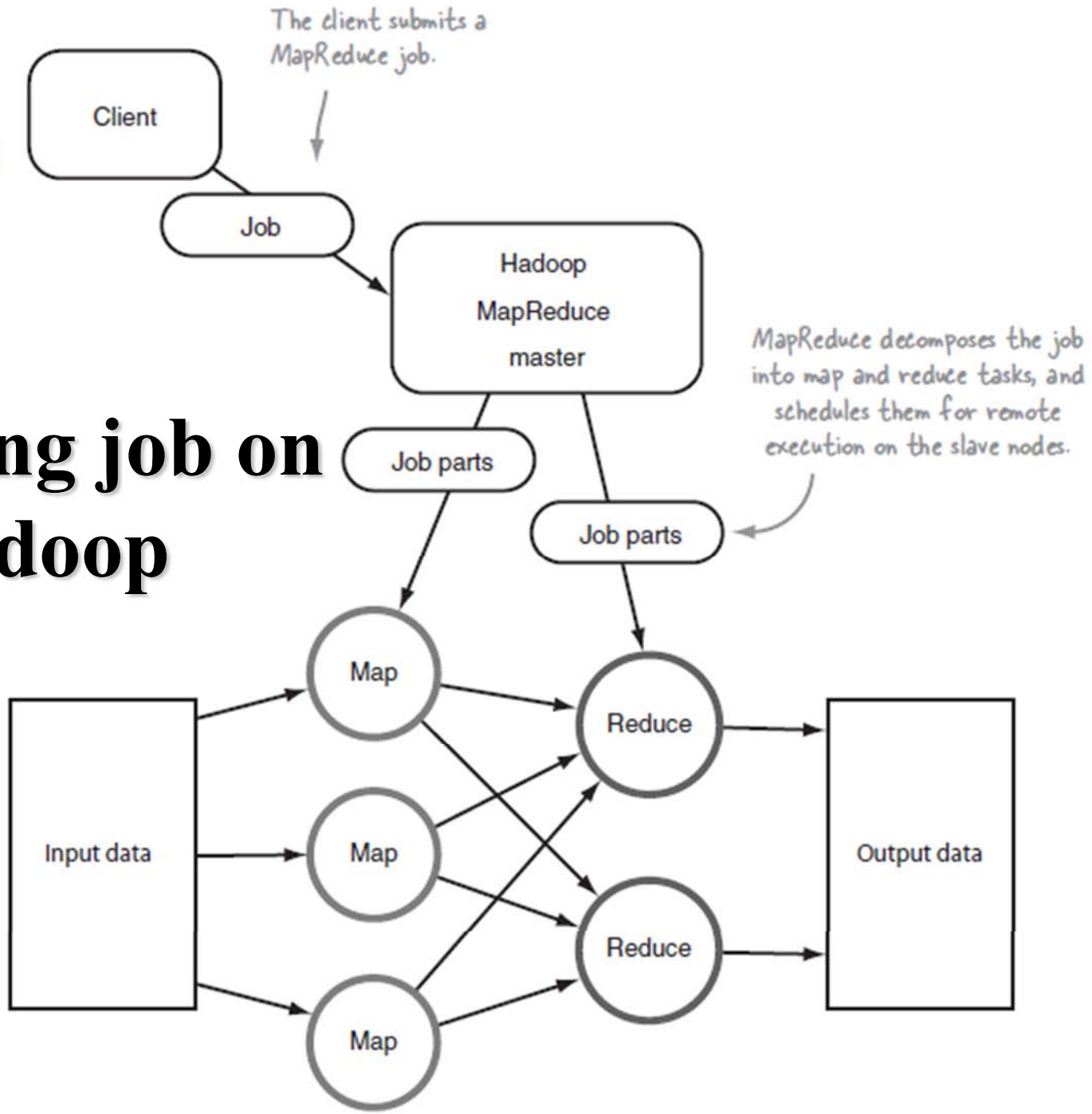


MapReduce Architecture

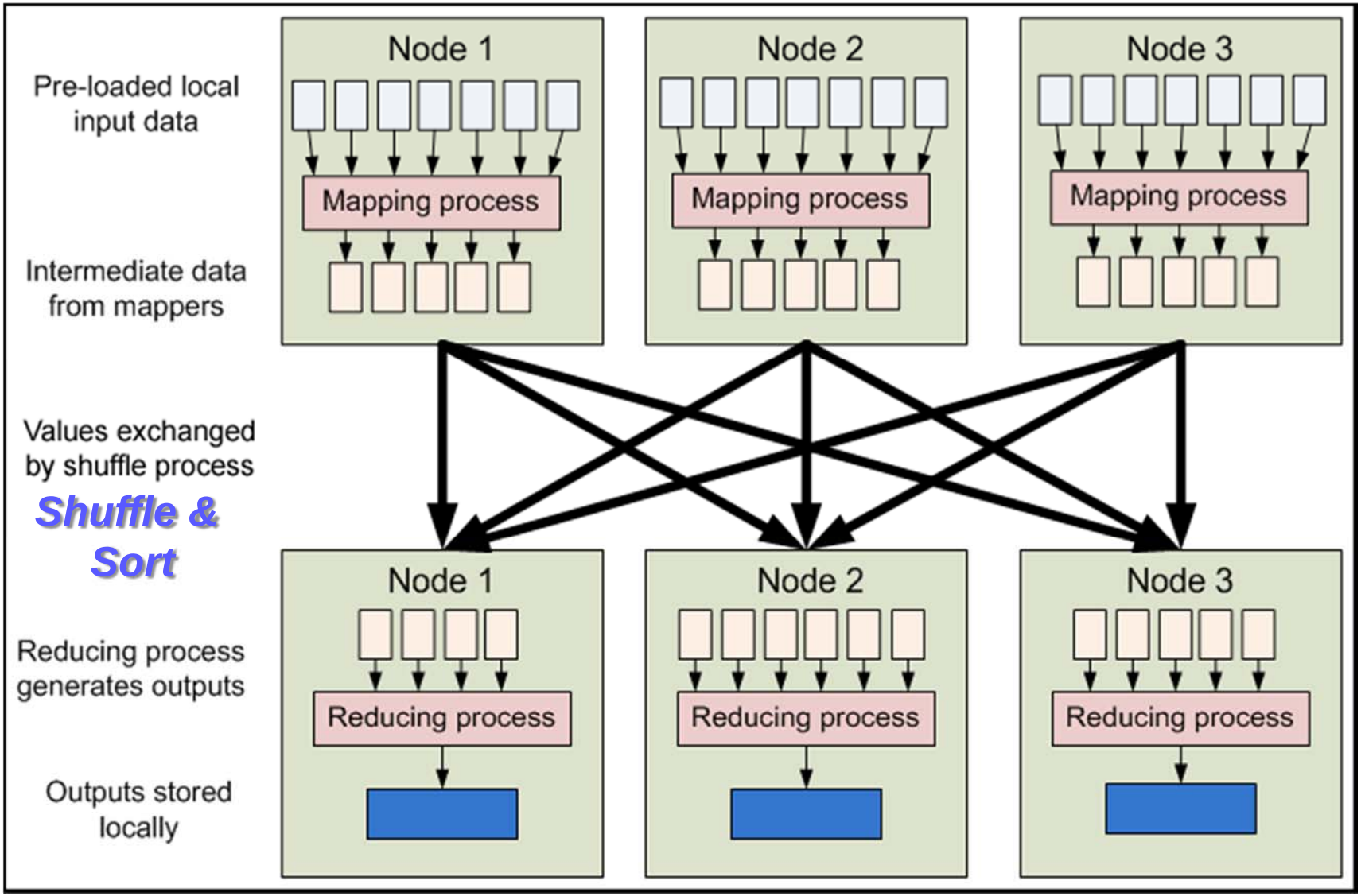
- **MapReduce** is the heart of Hadoop. It is this programming paradigm that allows for **massive scalability** across hundreds or thousands of servers in a Hadoop cluster.
- Designed to realize:
  - Large scale distributed batch processing infrastructures
  - Exploit low costs hardware from 1 to multiple cores, with low to large Mem, with low to large storage each
  - Covering huge (big data) storage that could not be covered by multi core parallel architectures
- The **Map function** in the master node takes the input, partitions it into smaller sub-problems, and distributes them to operational nodes.
  - Each operational node could do this again, creating a multi-level tree structure.
  - The operational node processes the smaller problems, and returns the response to its root node.
- In the **Reduce function**, the root/master node take the outputs/results of all the sub-problems, combining them to output the answer to the problem it is trying to solve.
- See Yahoo! Hadoop tutorial: https://developer.yahoo.com/hadoop/tutorial/index.html

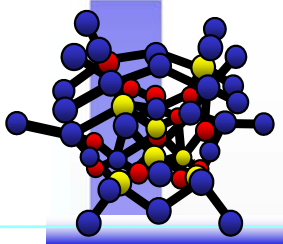


Releasing job on Hadoop



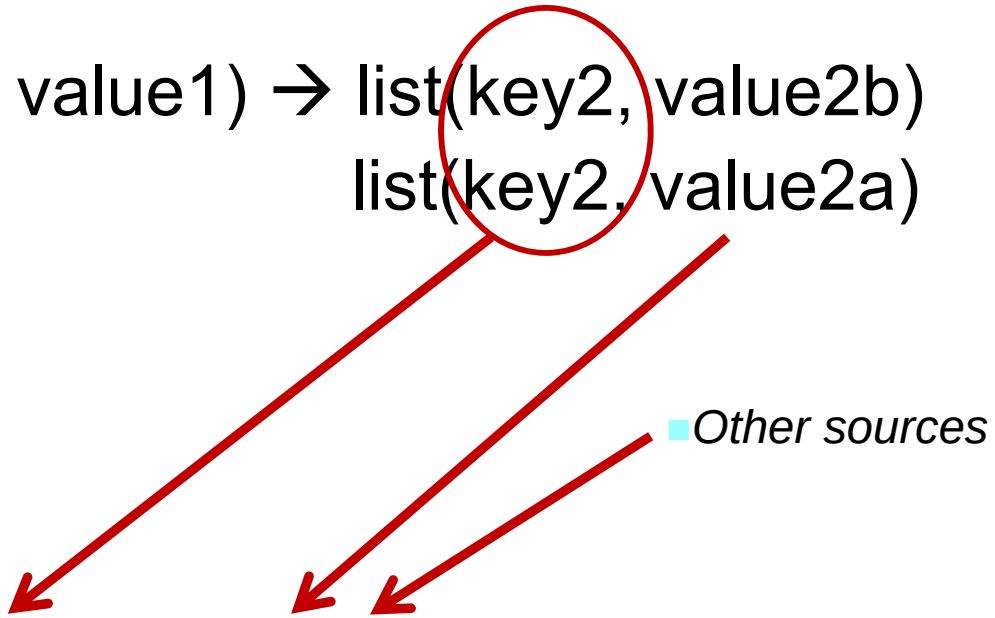
Mappers and Reducers



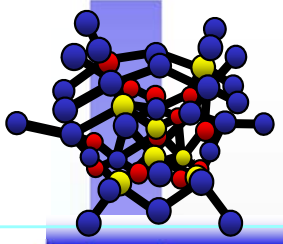


Key / Value Mapping

- Mapper
  - map(key1, value1) → list(key2, value2b)
    list(key2, value2a)

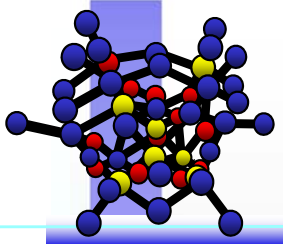


- Reducer:
  - reduce (key2, list (value2)) → list(key3, value3)
- When the mapping phase has completed, the intermediate (key, value) pairs must be exchanged (***Shuffle & Sort*** phase) among machines to send all values with the same key to a single reducer.



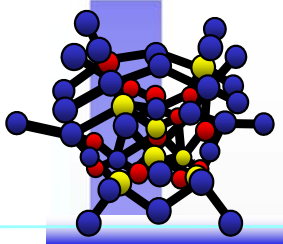
Index

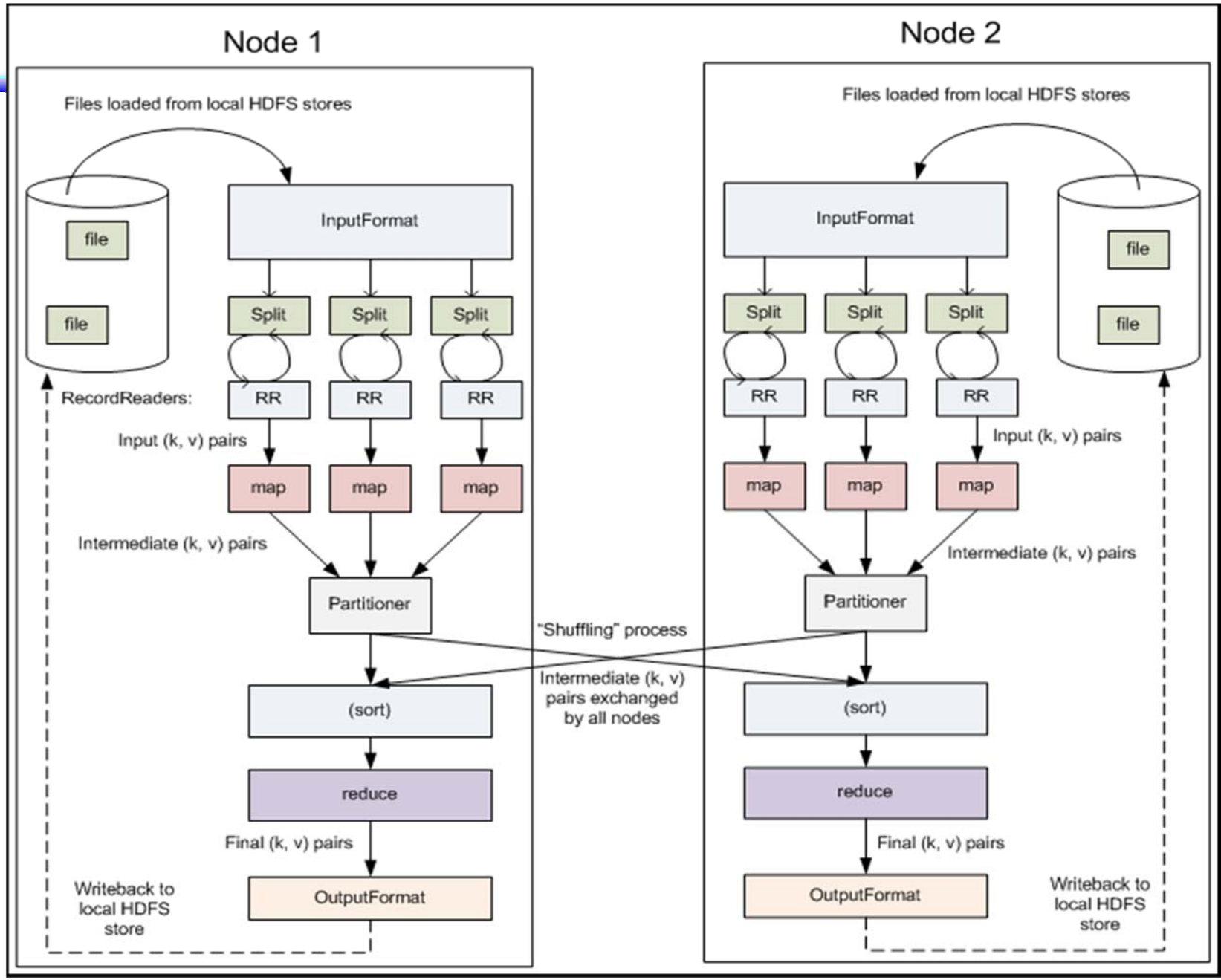
- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API

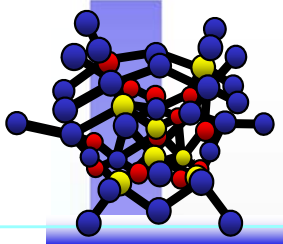


How Hadoop Works

- Programmer has to write the Mappers and the Reducers functions
- Data migration
  - *FROM*: Nodes hosting Mappers' outputs
  - *TO*: Nodes needing those data to processing Reducers
- Hadoop internally manages all of the data transfer and cluster topology issues.
- This is quite different from traditional parallel and GRID computing where the communications have to be coded (with MPI, RMI, etc.)




Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
file
file
InputFormat
InputFormat
Split
Split
Split
Split
Split
Split
RecordReaders:
RR
RR
RR
RR
RR
RR
Input (k, v) pairs
Input (k, v) pairs
map
map
map
map
map
map
Intermediate (k, v) pairs
Intermediate (k, v) pairs
Partitioner
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
(sort)
reduce
reduce
Final (k, v) pairs
Final (k, v) pairs
Writeback to local HDFS store
OutputFormat
OutputFormat
Writeback to local HDFS store

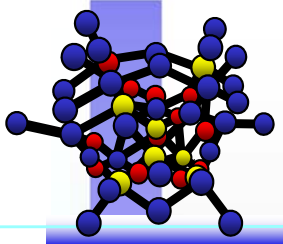


How Hadoop Works

1. A client accesses **unstructured** and **semi-structured data** from **different sources** including log/text files, internal data stores etc.

2. It **breaks the data up into "parts"**, which are then loaded into a file system made up of multiple nodes running on commodity hardware.

   - How these input files are split up and read is defined by the **InputFormat** class.
   - Selects the files or other objects that should be used for input.
   - Defines the **InputSplits** that break a file into tasks.
   - Provides a factory for **RecordReader** objects that read the file.



How Hadoop Works

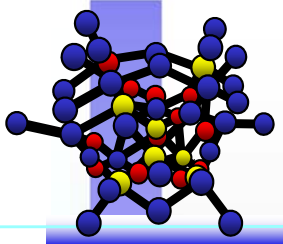
3. Each "part" is **replicated multiple times** and loaded into the file system so that **if a node fails, another node has a copy** of the data contained on the failed node.

4. A **Name Node acts as facilitator**, communicating back to the client information file metadata such as: which nodes are available, where in the cluster certain data resides, which nodes have failed etc.

5. Once the data is loaded into the cluster, it is ready to be analyzed via the MapReduce framework.

6. **Mapper**: The client submits a **Map job** - usually a query written in Java – to one of the nodes in the cluster known as the **Job Tracker**.

7. The **Job Tracker refers to the Name Node** to determine which data it needs to access to complete the job and where such data is located in the cluster.



How Hadoop Works

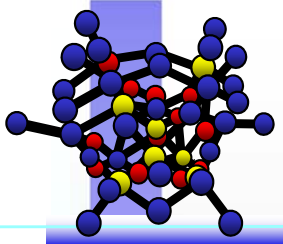
8. Once determined, the Job Tracker submits the query to the relevant nodes.

9. **Rather than bringing all the data back into a central location for processing**, processing then occurs at each node simultaneously, or in parallel. This is an essential characteristic of Hadoop**.**

10. Intermediate (*key,value*) pairs are output by the Mappers.

11. **Partition & Shuffle:** All values for the same key are reduced together regardless of which mapper is its origin. The ***Partitioner*** class determines which partition a given (key, value) pair will go to.

12. When **each node has finished processing** its given job, the intermediate (*key,value*) pairs are input to the **Reducers**.



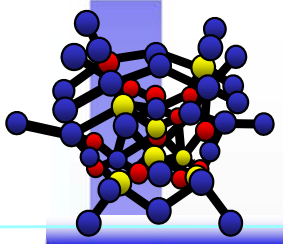
How Hadoop Works

13. **Reducer**: The **client initiates a "Reduce" job** through the Job Tracker in which **results of the map phase** stored locally on individual nodes **are aggregated to determine the "answer"** to the original query, then loaded on to another node in the cluster.

14. The (key, value) pairs provided by the Reducers are then written to HDFS output files.

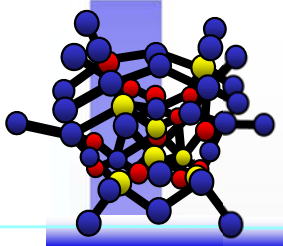
15. The client accesses these **results**, which can then be loaded into an **analytic environments** for analysis or further processing.

16. The **MapReduce** job has now been completed.



Index

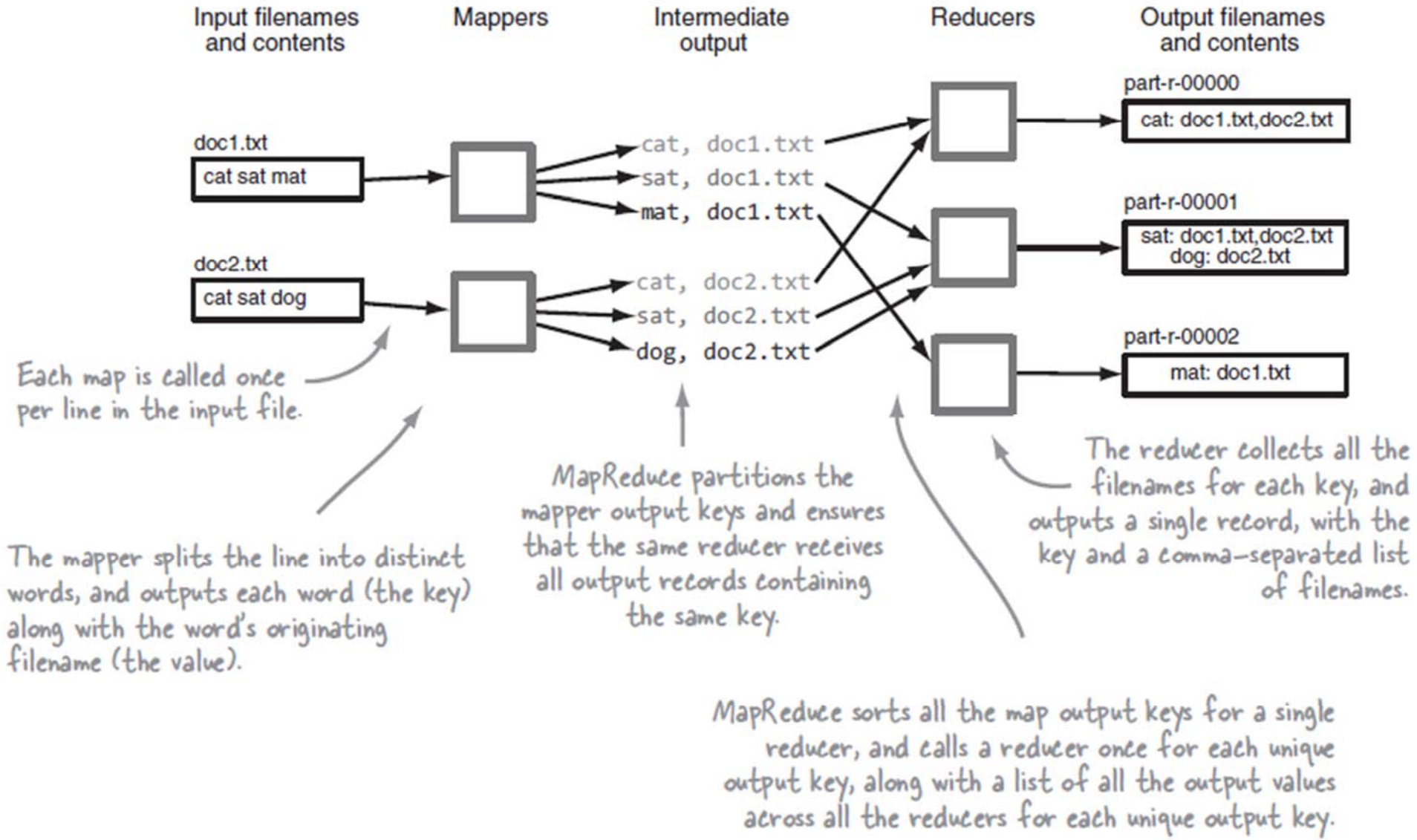
- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API

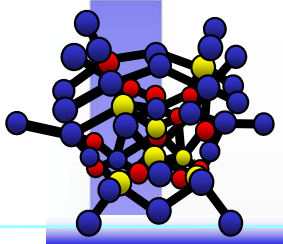


Example 1: Inverted Index Creation

- OBIETTIVO: Si abbia in input un certo numero di file di testo. Si vuole ottenere in output una lista contenente l'elenco delle singole parole presenti in ogni file testo e, per ogni singola parola, l'elenco di file che contengono quella stessa parola.
- Approccio implementativo tradizionale:
  - Implementare un metodo che raggruppi le parole comuni ai vari files di testo;
  - Eseguire tipicamente la procedura in memoria → problemi di esaurimento di memoria se si dispone di un gran numero di files o di chiavi univoche;
  - Usare un database intermedio di appoggio → non efficiente.
- Approccio distribuito tramite Hadoop:
  - Suddividere ("*Tokenize*") ciascun file testo in singole linee;
  - Produrre files intermedi contenenti una singola parola per linea;
  - Ordinare i files intermedi e raggrupparli per parola singola;

Example 1: Inverted Index Creation





Example 1: Mapper

When you extend the MapReduce mapper class you specify the key/value types for your inputs and outputs. You use the MapReduce default InputFormat for your job, which supplies keys as byte offsets into the input file, and values as each line in the file. Your map emits Text key/value pairs.

```
public static class Map
    extends Mapper<LongWritable, Text, Text, Text> {
```

A Text object to store the document ID (filename) for your input

```
  private Text documentId;

  private Text word = new Text();
```

To cut down on object creation you create a single Text object, which you'll reuse.

```
  @Override
  protected void setup(Context context) {
    String filename =
        ((FileSplit) context.getInputSplit()).getPath().getName();
    documentId = new Text(filename);
  }
```

Extract the filename from the context

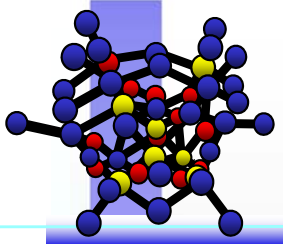
This method is called once at the start of the map and prior to the map method being called. You'll use this opportunity to store the input filename for this map.

This map method is called once per input line; map tasks are run in parallel over subsets of the input files.

```
  @Override
  protected void map(LongWritable key, Text value,
                     Context context)
      throws IOException, InterruptedException {
    for (String token :
            StringUtils.split(value.toString())) {
      word.set(token);
      context.write(word, documentId);
    }
  }
}
```

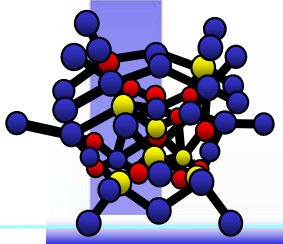
For each word your map outputs the word as the key and the document ID as the value.

Your value contains an entire line from your file. You tokenize the line using StringUtils (which is far faster than using String.split).



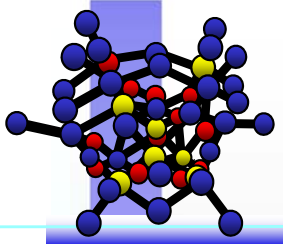
Example 1: Map Output

- Mapping output: → list(key2, value2)
  - Queste, Document1
  - sono, Document1
  - le, Document1
  - slide, Document1
  - di, Document1
  - Mario, Document1
  - Rossi, Document1
  - Le, Document2
  - slide, Document2
  - del, Document2
  - corso, Document2
  - di, Document2
  - Sistemi, Document2
  - Distribuiti, Document2



Example 1: Reduce Input

- reduce (key2, list (value2)) → **list(key3, value3)**
  - Queste, Document1
  - sono, Document1
  - le, Document1
  - slide, list (Document1, Document2)
  - di, list (Document1, Document2)
  - Mario, Document1
  - Rossi, Document1
  - Le, Document2
  - corso, Document2
  - Sistemi, Document2
  - Distribuiti, Document2



Example 1: Reducer

Much like your Map class you need to specify both the input and output key/value classes when you define your reducer.

```
public static class Reduce
    extends Reducer<Text, Text, Text, Text> {
  private Text docIds = new Text();
  public void reduce(Text key, Iterable<Text> values,
                     Context context)
      throws IOException, InterruptedException {

    HashSet<Text> uniqueDocIds = new HashSet<Text>();

    for (Text docId : values) {

      uniqueDocIds.add(new Text(docId));
    }

    docIds.set(new Text(StringUtils.join(uniqueDocIds, ",")));

    context.write(key, docIds);
  }
}
```

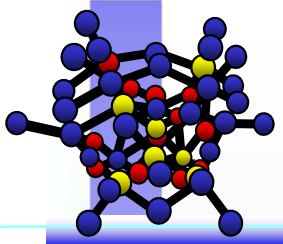
The reduce method is called once per unique map output key. The Iterable allows you to iterate over all the values that were emitted for the given key.

Keep a set of all the document IDs that you encounter for the key.

Iterate over all the DocumentIDs for the key.

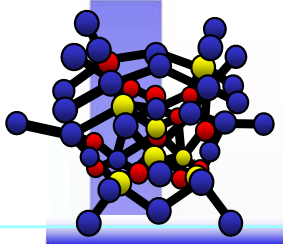
Add the document ID to your set. The reason you create a new Text object is that MapReduce reuses the Text object when iterating over the values, which means you want to create a new copy.

Your reduce outputs the word, and a CSV-separated list of document IDs that contained the word.



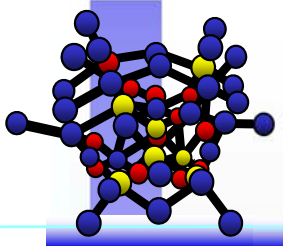
Example 1: Driver

```
public void run(String inputPath, String outputPath) throws
Exception {

    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
     // the keys are words (strings)
    conf.setOutputKeyClass(Text.class);
    // the values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(Reduce.class);
    FileInputFormat.addInputPath(conf, new Path(inputPath));
    FileOutputFormat.setOutputPath(conf, new Path(outputPath));
    JobClient.runJob(conf);

  }
```



Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API

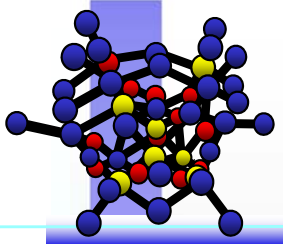


Example 2: A Simple Word Count

- Simple Word Counting program:
  - Document1.txt: `Queste sono le slide di Mario Rossi`
  - Document2.txt: `Le slide del corso di Sistemi Distribuiti`
- Expected Output:

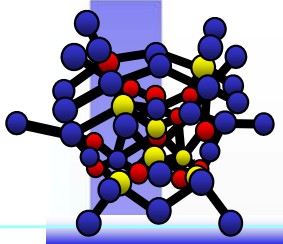
Queste	1
sono	1
le	1
slide	2
di	2
Mario	1
Rossi	1
Le	1
del	1
corso	1
Sistemi	1
Distribuiti	1

```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```



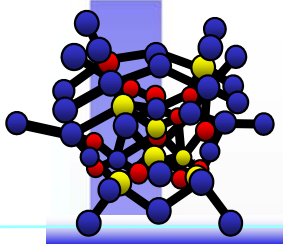
Example 2: Mapper

```
public static class MapClass extends MapReduceBase
    implements Mapper <LongWritable, Text, Text, IntWritable> {

     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();
     public void map (LongWritable key, Text value,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
          String line = value.toString();
          StringTokenizer itr = new StringTokenizer(line);
          while (itr.hasMoreTokens()) {
               word.set(itr.nextToken());
               output.collect(word, one);
          }
     }
}
```



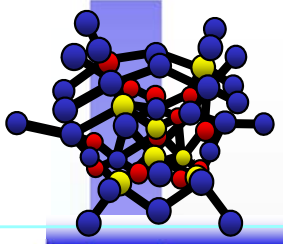
Example 2: Reducer

```
/*** A reducer class that just emits the sum of the input values. */
public static class Reduce extends MapReduceBase
    implements Reducer <Text, IntWritable, Text, IntWritable> {

    public void reduce (Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter
        reporter) throws IOException {

      int sum = 0;
      while (values.hasNext()) {
          sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));

    }

}
```



Example 2: Reducer Output

- Counting Words (*Example 2*)
  - …
  - le, 1
  - slide, 2
  - di, 2
  - …
- inverted index (*Example 1*)
  - …
  - le, Document1
  - slide, list (Document1, Document2)
  - di, list (Document1, Document2)
  - …

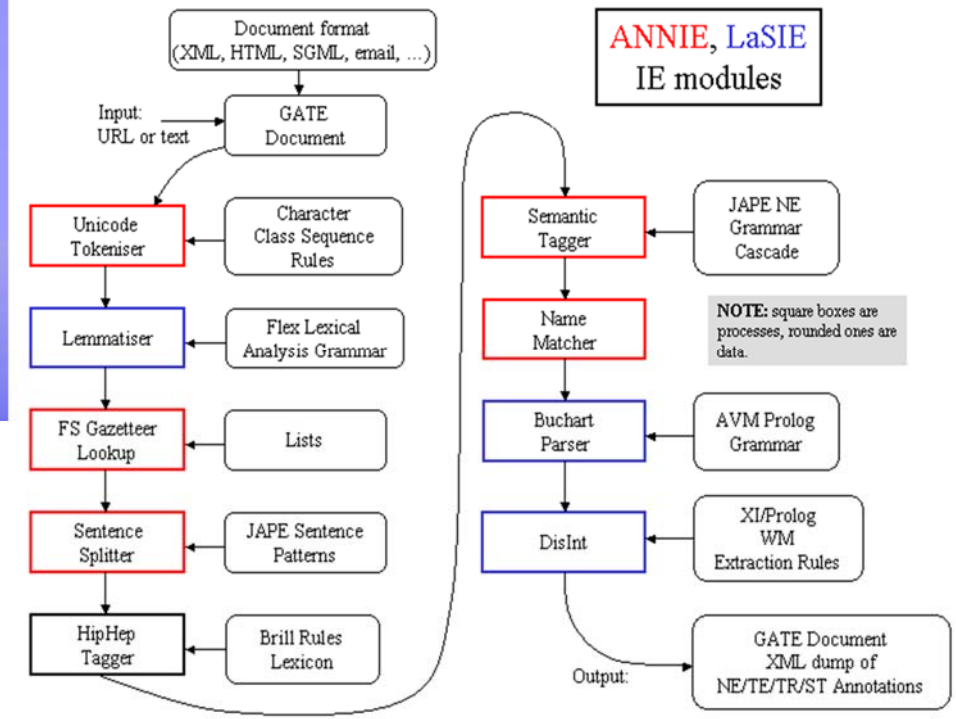


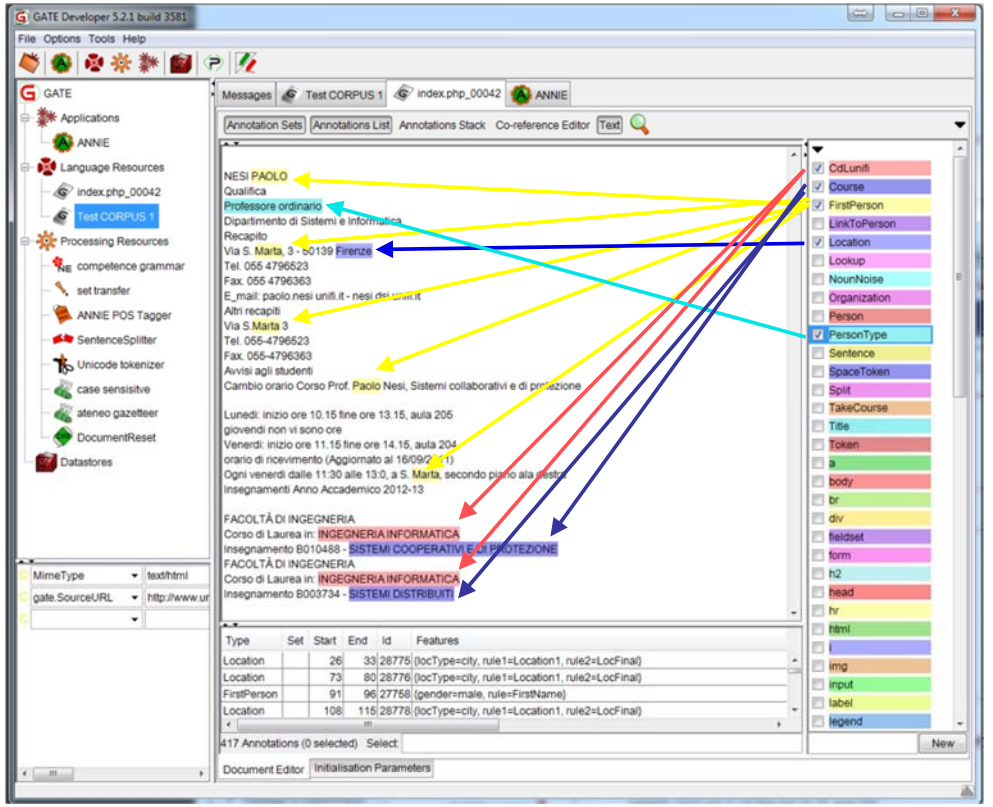
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API

Example 3: A Real Implementation Case

- **GATE – *General Architecture for Text Engineering***
  - Open Source Framework for developing and deploying software components that process human Natural Language
  - GATE supports plain text documents, XML, RTF, HTML, SGML



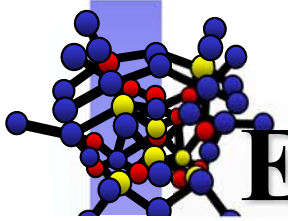


Example 3: A Real Implementation Case

- Problema: utilizzare le API di GATE per eseguire l'annotazione grammaticale (POS, Part-Of-Speech tagging) di testo in linguaggio naturale dalle pagine di un dominio Web.

- Esempio: *"Il Professor Paolo Nesi è il docente di questo corso."*

Il:	*articolo determinativo mashile singolare*
Professor:	*sostantivo comune maschile singolare*
Paolo Nesi:	*nome proprio*
è:	*verbo essere terza persona singolare indicativo presente*
il:	*articolo determinativo maschile singolare*
docente:	*sostantivo maschile singolare*
di:	*preposizione semplice*
questo:	*aggettivo dimostrativo maschile singolare*
corso:	*sostantivo comune maschile singolare*

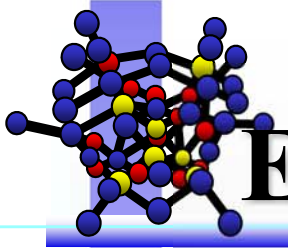


Example 3: A Real Implementation Case

GATEApplication.java : https://github.com/wpm/Hadoop-GATE/blob/master/src/main/java/wpmcn/gate/hadoop/HadoopGATE.java

```
/**
 * A wrapper for a GATE application
 * This object parses arbitrary text using a GATE application supplied to the constructor.
 */
public class GATEApplication {
   private CorpusController application;
   private Corpus corpus;

   /**
       * Initialize the GATE application
       *
       * @param gateHome path to an GATE application directory containing
       * an application.xgappin in its root
   **/
   public GATEApplication(String gateHome) throws GateException, IOException {
      Gate.runInSandbox(true);
      Gate.setGateHome(new File(gateHome));
      Gate.setPluginsHome(new File(gateHome, "plugins"));
      Gate.init();
      URL applicationURL = new URL("file:" + new Path(gateHome, "<GATE_APP>.xgapp").toString());
      application = (CorpusController) PersistenceManager.loadObjectFromUrl(applicationURL);
      corpus = Factory.newCorpus("GATE Corpus");
      application.setCorpus(corpus);
   }

   /**
     * Analyze text from web URL, returning annotations in XML @param text the
     * text to analyze
   **/
   public String xmlAnnotation(String url) throws ResourceInstantiationException,
ExecutionException {
     Document document = Factory.newDocument(new URL(url));
     annotateDocument(document);
     String xml = document.toXml();
     Factory.deleteResource(document);
     return xml;
   }
```

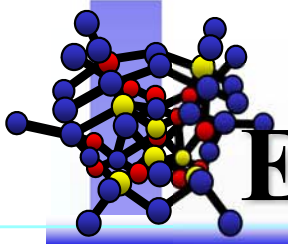
```
   @SuppressWarnings({"unchecked"})
   private Document annotateDocument(Document document) throws
ResourceInstantiationException, ExecutionException {
      corpus.add(document);
      application.execute();
      corpus.clear();
      return document;
   }

   /**
     * Free resources associated with this object. This should
     * be called before the object is deleted.
    **/
   public void close() {
      Factory.deleteResource(corpus);
      Factory.deleteResource(application);
   }


/**
      * Entry point that allows the GATE application to be
      * run outside Hadoop. The first positional argument is
      * an archived GATE application file, and the second
      * positional argument is a document. The document
      * annotations are written to standard out.
      *
**/
   static public void main(String[] args) throws Exception {
      String gateHome = args[0];
      String content = FileUtils.readFileToString(new File(args[1]));

      GATEApplication gate = new GATEApplication(gateHome);
      String annotation = gate.xmlAnnotation(content);
      System.out.println(annotation);
      gate.close();
   }
}
```



Example 3: A Real Implementation Case

HadoopGATE.java : https://github.com/wpm/Hadoop-GATE/blob/master/src/main/java/wpmcn/gate/hadoop/GATEApplication.java

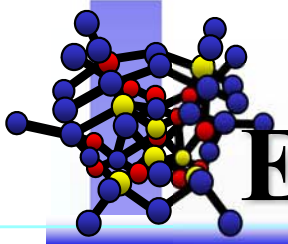
```
public class HadoopGATE {

    public static class Map extends Mapper<LongWritable, DBInputWritable, Text, Text> {
        //MapReduceBase implements Mapper<BytesWritable, Text, Text, Text> {
        //  [. . .]
    }

    public static class Reduce extends Reducer<Text, Text, Text, NullWritable> {
        // MapReduceBase implements Reducer<Text, Text, DBOutputWritable, NullWritable> {
        //  [. . .]
    }

    public static void main(String[] args) throws Exception {

        Configuration conf = new Configuration();

        // The first positional argument is a path to the archived GATE application.
        Path localGateApp = new Path(args[0]);  //da inserire path di GATE HOME

        // Copy the GATE application to a UNIQUE temporary HDFS directory.
        Path hdfsGateApp = new Path("<TMP_DIRECTORY>" + UUID.randomUUID());

        FileSystem fs = FileSystem.get(conf);
        fs.copyFromLocalFile(localGateApp, hdfsGateApp);
        DistributedCache.addCacheArchive(hdfsGateApp.toUri(), conf);

        Job job = new Job(conf);
        job.setJarByClass(KeywordExtraction.class);
        job.setJobName("POS-Tagging Task");
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(NullWritable.class);
        job.setInputFormatClass(DBInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        //Inserire Credenziali DB MySQL
        DBConfiguration.configureDB(job.getConfiguration(), "com.mysql.jdbc.Driver",
        "jdbc:mysql://<IP_NODE>:<PORT>/<DB_NAME>", // db url
        "<USER_NAME>",     // user name
        "<PASSWORD>"); //password

        DBInputFormat.setInput(job, DBInputWritable.class,
             "SELECT id, url, crawling_date FROM <DB_NAME>.<TABLE_NAME>",
             "SELECT COUNT(id) FROM <TABLE_NAME>");
        TextOutputFormat.setOutputPath(job, new Path("<OUTPUT_HADOOP_PATH>" +
        System.currentTimeMillis() + ""));

         boolean success = job.waitForCompletion(true);
         if (success){
             FileSystem.get(job.getConfiguration()).deleteOnExit(hdfsGateApp);
         }
         System.exit(success ? 0 : -1);

    }
```



Example 3: A Real Implementation Case

Map Class

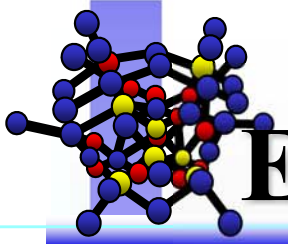
```
public static class Map extends Mapper<LongWritable, DBInputWritable, Text, Text> {
     //MapReduceBase implements Mapper<BytesWritable, Text, Text, Text> {

       public String getHost(String url){
             if(url == null || url.length() == 0) return "";
             int doubleslash = url.indexOf("//");
             if(doubleslash == -1) {
                 doubleslash = 0;
             } else {
                 doubleslash += 2;
             }

             int end = url.indexOf('/', doubleslash);
             end = end >= 0 ? end : url.length();

             int port = url.indexOf(':', doubleslash);
             end = (port > 0 && port < end) ? port : end;

             return url.substring(doubleslash, end);
       }

       /*Considerando key = URL della singola pagina, value = dominio Web a cui appartiene
       la pagina a cui punta l'URL */
       /* N.B. GATE si occupa del parsing del contenuto testuale della pagina web, ricevendo
       direttamente l'URL in ingresso. */
       public void map(LongWritable key, DBInputWritable value, Context context) throws
                 IOException, InterruptedException{
            //OutputCollector<Text, Text> output, Reporter reporter) throws IOException {

             Text domain = new Text();
             Text url = new Text();
             String s = getHost(value.getUrl());
             domain.set(s);
             url.set(value.getUrl());
             context.write(domain, url);

       }

}
```



Example 3: A Real Implementation Case

Reduce Class (1 of 2)

```
public static class Reduce extends Reducer<Text, Text, Text, NullWritable> {
     // MapReduceBase implements Reducer<Text, Text, DBOutputWritable, NullWritable> {

   private static GATEApplication gate;

   @Override
   protected void setup(Context context) throws IOException, InterruptedException {
      if (null == gate) {
           Configuration configuration = context.getConfiguration();
           Path[] localCache = DistributedCache.getLocalCacheArchives(configuration);
           try {
                gate = new GATEApplication(localCache[0].toString());
           } catch (GateException e) {
                throw new RuntimeException(e);
           }
      }
   }

   public void reduce(Text key, Iterable<Text> values, Context context) throws
   IOException, InterruptedException{
      //OutputCollector<Text, NullWritable> output) throws IOException {

      // Estrazione Keyword e Keyphrase

      // [. . .]
   }

   @Override
   protected void cleanup(Context context) throws IOException, InterruptedException {
      gate.close();
    }
}
```

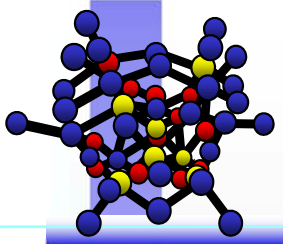


Example 3: A Real Implementation Case

Reduce Class (2 of 2)

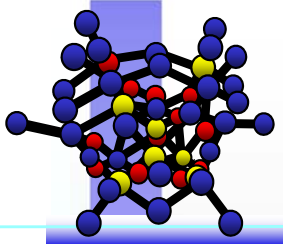
```
public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
   //OutputCollector<Text, NullWritable> output) throws IOException {

   // Estrazione Keyword e Keyphrase
   ArrayList<String[]> termsDocsArray = new ArrayList<String[]>();
   ArrayList<HashMap<String, String>> POSTermsDocsArray = new ArrayList<HashMap<String,
            String>>();
   ArrayList<String> fileNames = new ArrayList<String>();
   ArrayList<String> allTerms = new ArrayList<String>();
   ArrayList<double[]> tfidfDocsVector = new ArrayList<double[]>();

   // Vengono passati gli URL (da MySQL) a cui si trovano i documenti di testo
   for(Text v: values) {
        String fileURL = v.toString();
        String POSKeywords;
        HashMap<String, String> POSTerms = new HashMap();
        try {
             POSKeywords = gate.POSKeywordsAnnotation(fileURL);
        } catch (ResourceInstantiationException e) {
             throw new RuntimeException(e);
        } catch (ExecutionException e) {
             throw new RuntimeException(e);
        } catch (MalformedURLException ex) {
             throw new RuntimeException(ex);
        }
        String[] lines = POSKeywords.split(System.getProperty("line.separator"));
```

```
        for(int i = 0; i<lines.length; i++){
             if(lines[i].contains("KPH")){
                  String[] keyphrase = lines[i].split(" KPH");
                  allTerms.add(keyphrase[0]);
                  POSTerms.put(keyphrase[0], "KPH");
             } else {
                  String[] keyword = lines[i].split(" ");
                  allTerms.add(keyword[0]);
                  POSTerms.put(keyword[0], keyword[1]);
             }
        }
        fileNames.add(fileURL);
        Iterator<String> itr = fileNames.iterator();

        while(itr.hasNext()){
             String fileName = itr.next();
             String strout = "";
             for(int i = 0; i<allTerms.size(); i++){
                  strout = strout + "; " + allTerms.get(i);
             }
        }
   }
   Text out = new Text();
   out.set(strout);
   context.write(out, NullWritable.get());
}
```



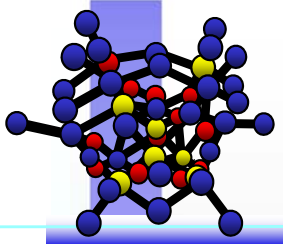
Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



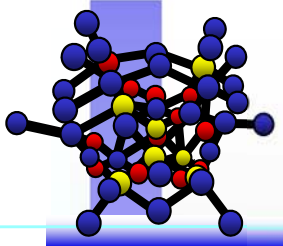
Hadoop Pros & Cons

- **Main Benefit**: it allows enterprises to process and analyze **large volumes** of unstructured and semi-structured data, in **a cost-effective and time-effective** manner.
- Hadoop clusters can scale to **petabytes** order of data; **enterprises can process** and analyze **all** relevant data (no sample data sets).
- Developers can **download** the Open Source Apache Hadoop distribution **for free** and begin experimenting with Hadoop in less than **a day**.



Hadoop Pros & Cons

- **Data Scientists** can apply an iterative approach to analysis, **continually refining** and testing queries to uncover previously unknown insights. **Inexpensive** to **get started** with Hadoop.
- **Main downside**: Hadoop and its components are immature and still developing.
- Implementing and managing Hadoop clusters and performing advanced analytics on large volumes of unstructured data, requires significant **expertise**, **skill** and **training**.

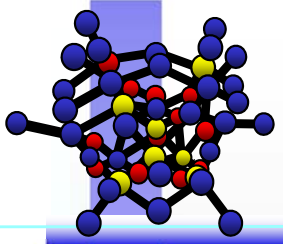


Hadoop Complementary Subprojects

HBASE

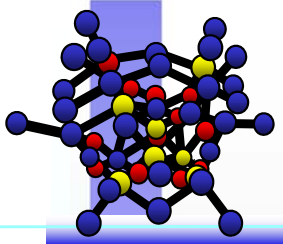
Cassandra

HIVE



Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



HBase is the Hadoop Distributed Datastore.

HBase is a NoSql Database

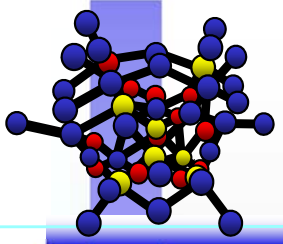
- However, it offers typical *Create*, *Read*, *Update* and *Delete* (*CRUD*) operations.

Open-source implementation of ***Google's BigTable***.

Developed as part of Apache's Hadoop project, it runs on top of **HDFS**.

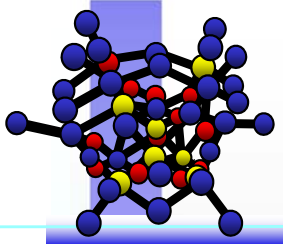
Integration with MapReduce framework.

Provides high scalability and fault-tolerant storage of very large quantities of sparse data.



Index

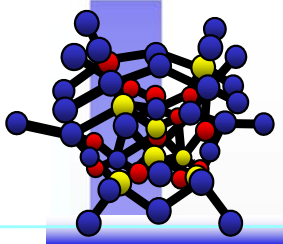
- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



HBase Data Model

- Data are stored in ***Tables.***
- ***Tables*** contains ***Rows***, which are referenced by a unique ***Key***
  - A Key is an array of bytes: it can be any kind of serialized data structured (String, int, long etc…).
  - *Rows are sorted lexicographically by key* (Compared on a binary level from left to right).

    E.G.: keys 1,3,8,10,12, 24 will get sorted as 1,10,12, 2, 24, 3, 8.
- Rows are composed of ***Columns***
  - Columns are NOT static; new columns are created at runtime.
- ***Columns*** are grouped into ***Column Families***
  - Each column family is stored as one file (HFile) in HDFS, which is basically a key-value Map.
  - Each column is referenced by `family:qualifier` (e.g. `user:first_name`).
- Data are actually stored in ***Cells,*** Identified by *Row x Column-family x Column.*



HBase Data Model

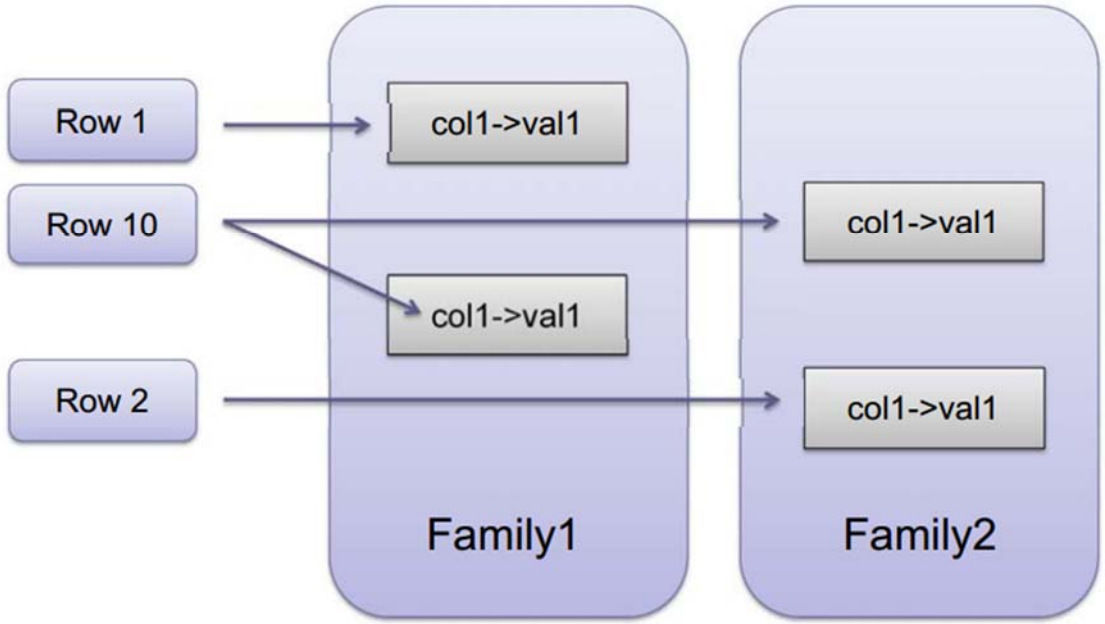
- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Rowid	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

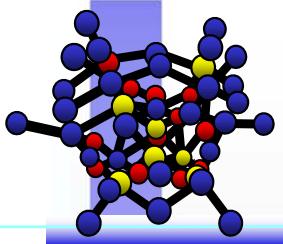
COLUMN FAMILIES

Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

HBase Data Model

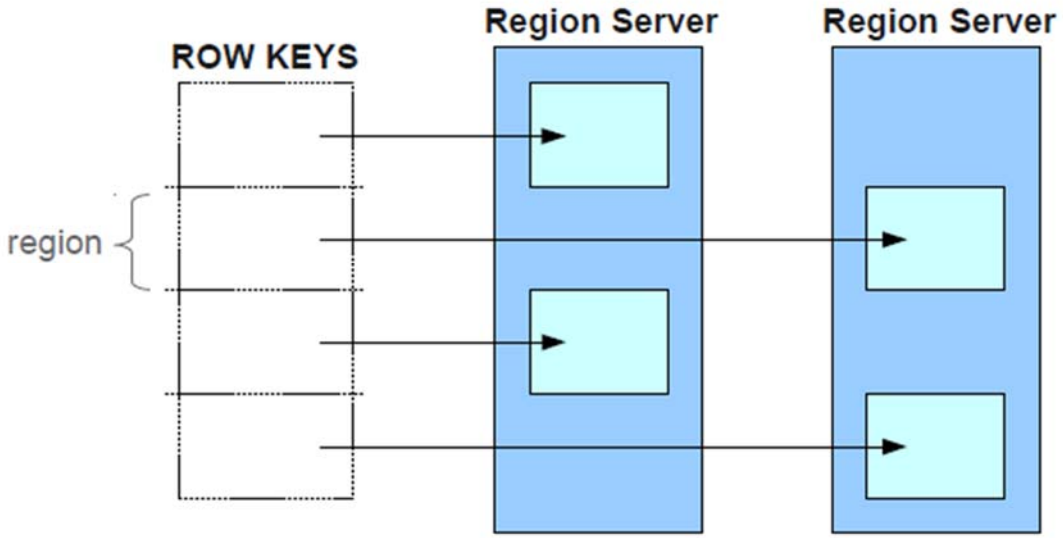


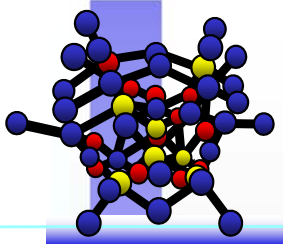
- ***Cell Values*** are versioned
  - For each cell multiple versions are kept (3 by default).
- ***Cell Value*** = *Table + RowKey + Columng_Family + Column + Timestamp.*



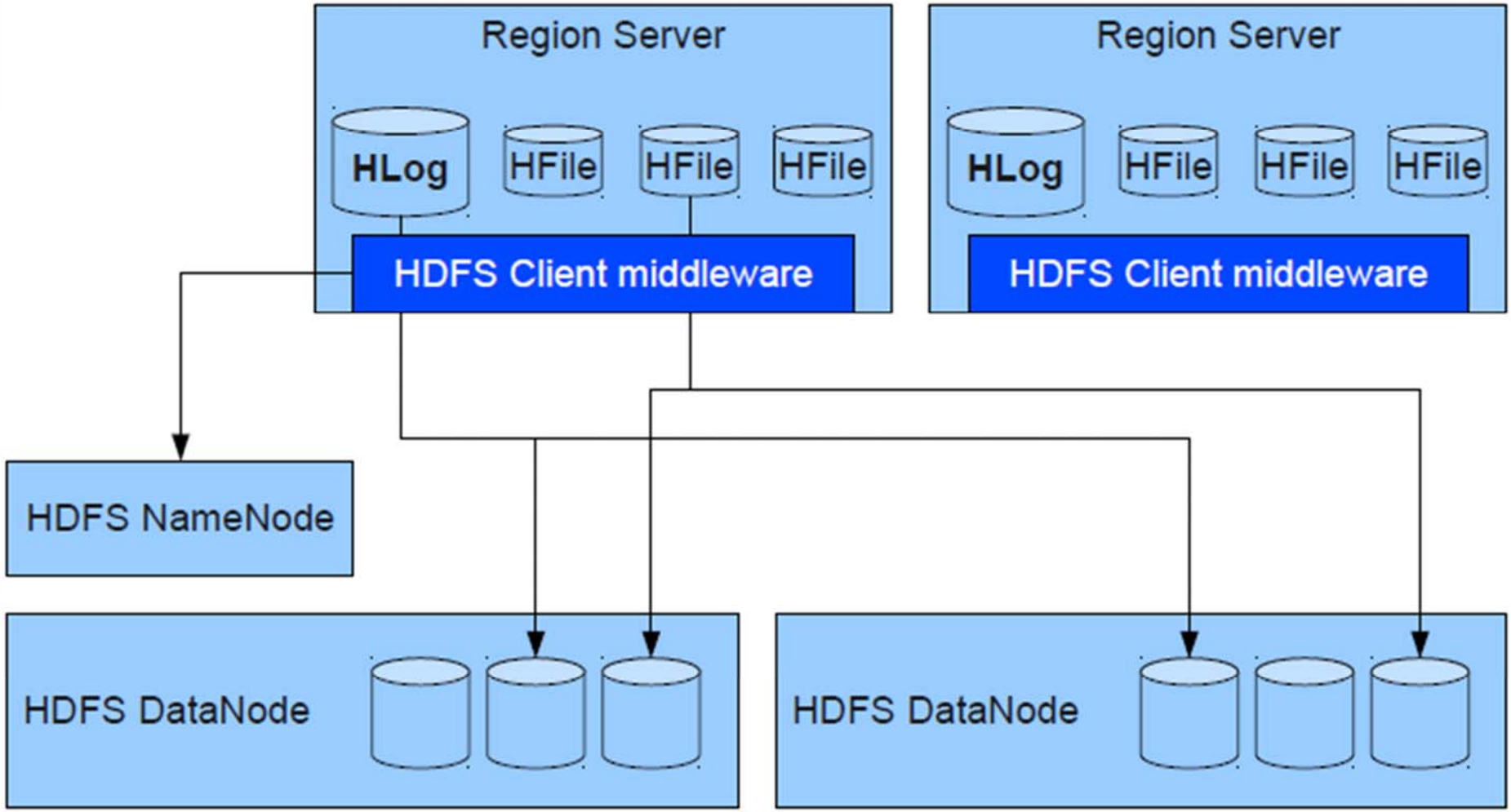
HBase Data Model

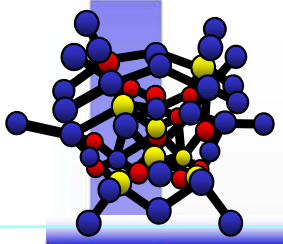
- For scalability, ***Tables*** are split into ***Regions*** by ***Key*** (*Start Key* → *Stop Key*).
- A ***Region*** is a range of rows stored together
  - Regions are dynamically split as they become too big and merged if too small.
- Each ***Region*** is assigned to a ***Region Server*** on different Cluster Nodes





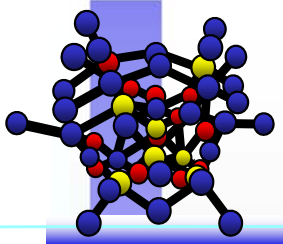
HBase over HDFS





Index

- Problem Scope
- *(Introduction to)* Hadoop
- Hadoop Architecture
  - HDFS
  - MapReduce
- How Hadoop Works
- Example 1: *Inverted Index Creation*
- Example 2: *A Simple Word Count*
- Example 3: *A Real Implementation Case*
- Hadoop Pros & Cons
- HBase
  - Data Model
  - Client API



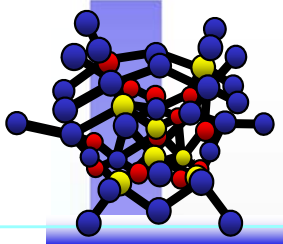
HBase Client API

- HBaseAdmin is used to:
  - Create/delete tables: `createTable(); deleteTable();`

```
HBaseAdmin hbAdm = new HbaseAdmin(HBaseConfiguration.create());
hbAdm.createTable(new HTableDescriptor("TestTable"));
```

  - Add/delete column families to tables: `addColumn(); deleteColumn();`

```
hbAdm.addColumn("TestTable", new HcolumnDescriptor("TestColFamily"));
```



HBase Client API

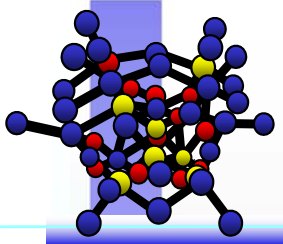
- `HTable` class is the basic data access entity:
  - Read data with `get();`

```
Get get = new Get(Bytes.toBytes("testRow"));
Result result = testTable.get(get);
for(byte[] family: result.keySet())
   for(byte qual: result.get(family).keySet())
     for(Long ts: result.get(family).get(qual).keySet())
        String val = Bytes.toString(
                        Result.get(family).get(qual).get(ts));
```

  - Write data with `put();`

```
Put put = new Put(Bytes.toBytes("testRow"));
put.add(Bytes.toBytes("testFam"),
        Bytes.toBytes("testQual"), Bytes.toBytes("value"));
testTable.put(put);
```

  - Delete data with `delete(); checkAndDelete();`



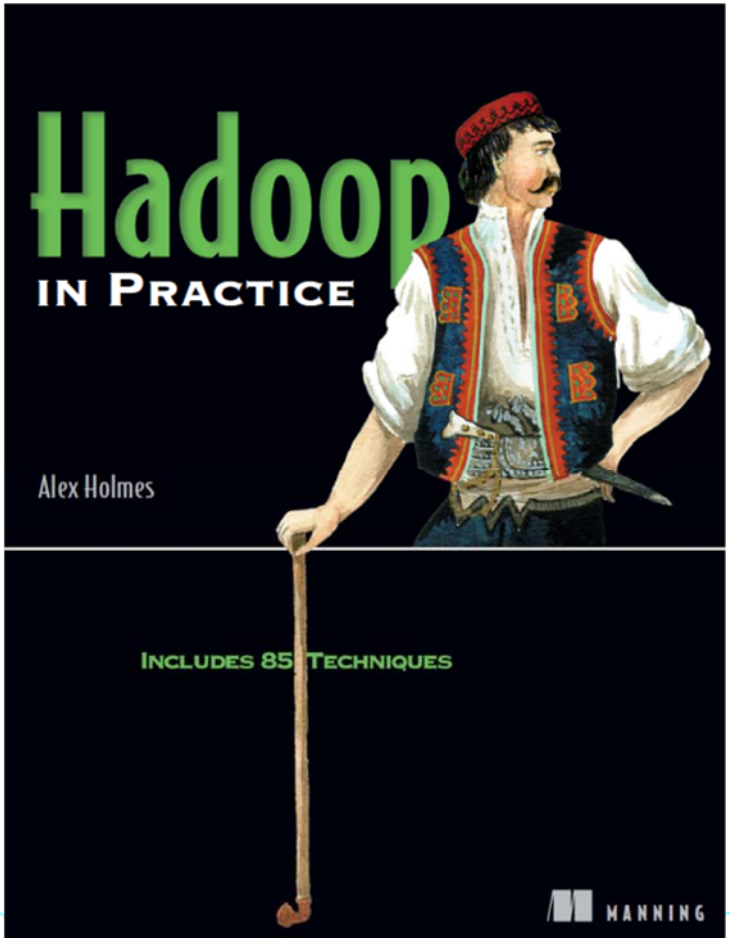
References

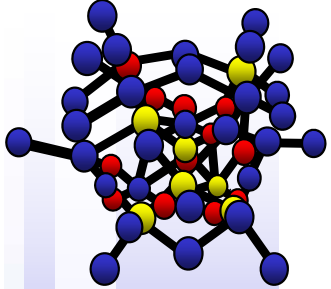
- *P. Bellini, I. Bruno, D. Cenni, P. Nesi, "Micro grids for scalable media computing and intelligence on distributed scenarious", IEEE Multimedia, Feb 2012, Vol.19, N.2, pp.69.79, IEEE Computer Soc. Press.*
- *P. Bellini, M. Di Claudio, P. Nesi, N. Rauch, "Tassonomy and Review of Big Data Solutions Navigation", in "Big Data Computing", Ed. Rajendra Akerkar, Western Norway Research Institute, Norway, Chapman and Hall/CRC press, ISBN 978-1-46-657837-1, eBook: 978-1-46-657838-8, July 2013.*
- *Yahoo! Hadoop Tutorial: https://developer.yahoo.com/hadoop/tutorial/*
- *A. Holmes, Hadoop in Practice, 2012 by Manning Publications Co. All rights reserved.*
- *N. Dimiduk, A. Khurana, HBase in Action, 2013 Manning Publications Co. All rights reserved.*

References

- Yahoo tutorial
  - https://developer.yahoo.com/hadoop/tutorial/index.html

- Book:
  - ALEX HOLMES, *Hadoop in Practice*, 2012 by Manning Publications Co. All rights reserved.





fine

fine