Processing of massive data: MapReduce

2. Hadoop

DISTRIBUTED
SYSTEMS
LABORATORY

MapReduce Implementations

- Google were the first that applied MapReduce for big data analysis
  - Their idea was introduced in their seminal paper "*MapReduce: Simplified Data Processing on Large Clusters*"
  - Also, of course, they were the first to develop a MapReduce framework
  - Google has provided lots of information about how their MapReduce implementation and related technologies work, *but have not released their software*
- Using Google's seminal work, others have implemented their own MapReduce frameworks

MapReduce Implementations (2)

- Disco, by Nokia (http://discoproject.org/, based on Python)
- Skynet, by Geni (http://skynet.rubyforge.org/ , based on Ruby)
- Some companies offer data analysis services based on their own MapReduce platforms (Aster Data, Greenplum)
- **Hadoop** (http://hadoop.apache.org/) is the most popular open-source MapReduce framework
  - Amazon's Elastic MapReduce offers a ready-to-use Hadoop-based MapReduce service on top of their EC2 cloud.
  - It is not a new MapReduce implementation, and it does not provide extra analytical services either

About Hadoop

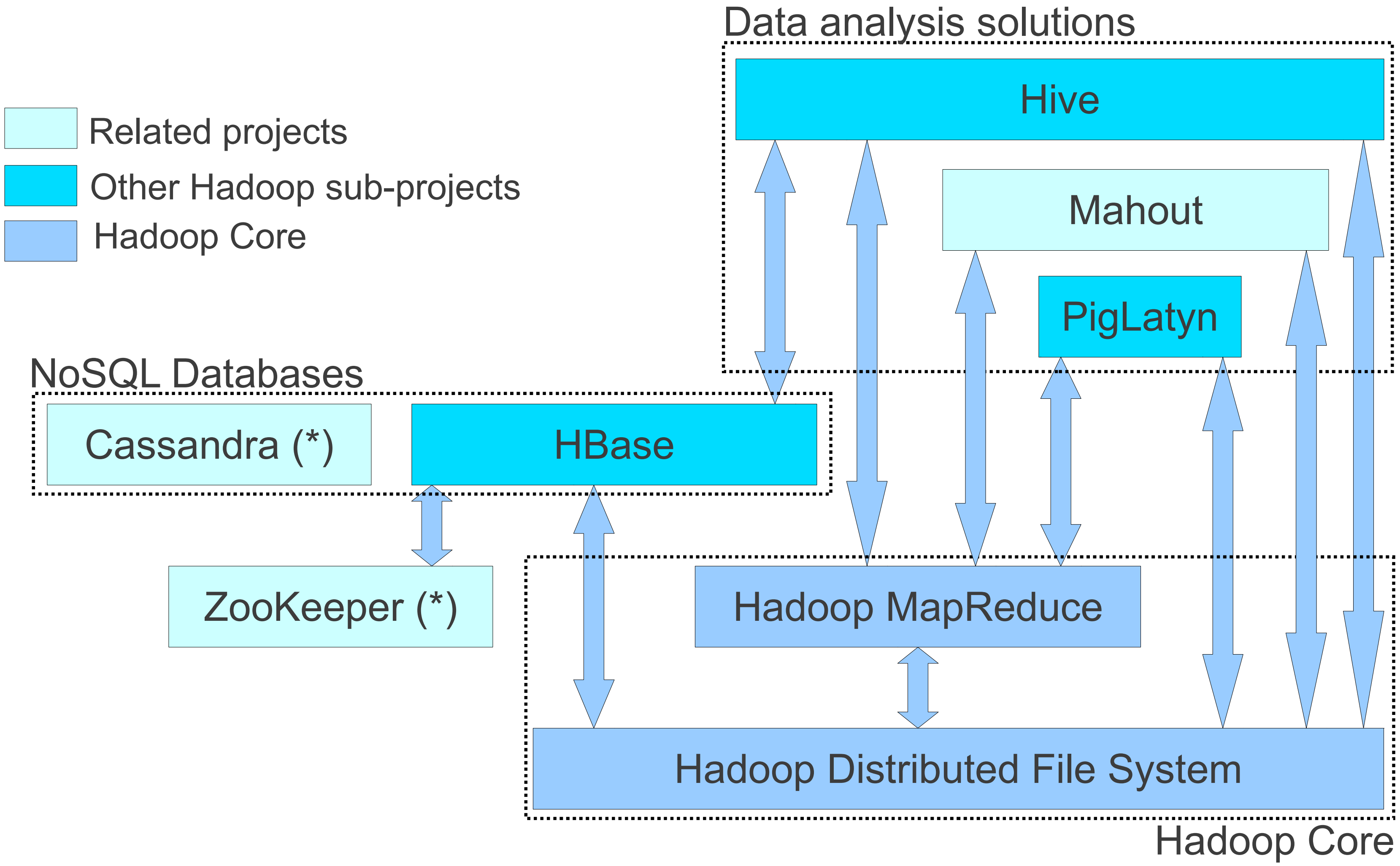
- Hadoop is a project of the Apache Software Foundation (ASF)
- They develop software for the distributed processing of large data sets
- Several software projects are part of Hadoop, or are related with it:
    - “Core”: Hadoop Common, Hadoop Distributed File System, Hadoop MapReduce
    - Related: HBase, PigLatyn, Cassandra, Zookeeper...

Hadoop Core Projects

- Hadoop Common: common functionality used by the rest of Hadoop projects (serialization, RPC...)
- Hadoop Distributed File System (HDFS), based on Google File System (GFD), provides a distributed and fault-tolerant storage solution
- Hadoop MapReduce, implementation of MapReduce

(the three projects are distributed toghether)

Hadoop Software Ecosystem



- ▢ Related projects
- ■ Other Hadoop sub-projects
- ▨ Hadoop Core

Data analysis solutions

Hive

Mahout

PigLatyn

NoSQL Databases

Cassandra (*)

HBase

ZooKeeper (*)

Hadoop MapReduce

Hadoop Distributed File System

Hadoop Core

(*) ZooKeeper and Cassandra are related with Hadoop because they have similar funcionality, or are used by, Hadoop sub-projects. But they do not depend on Hadoop's sw

Hadoop Distributed File System

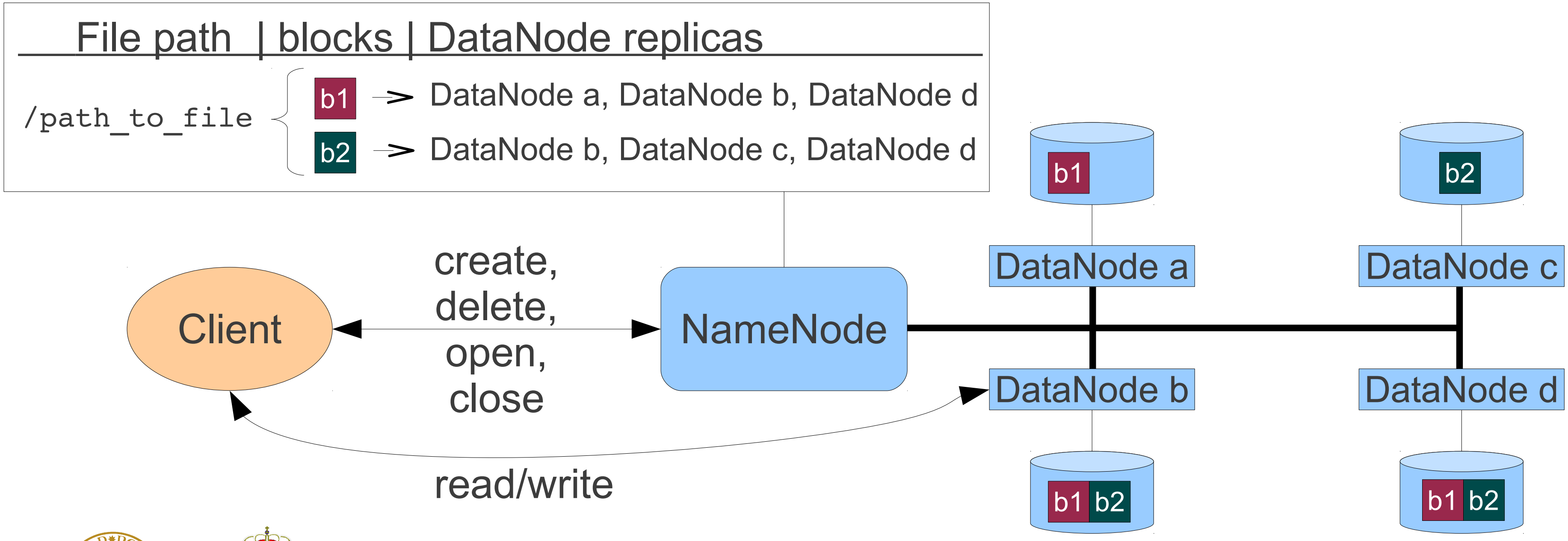
- Before focusing on Hadoop MapReduce we must understand how HDFS works
- HDFS is based on Google File System (GFS), which was created to meet Google's need for a distributed file system (DFS)
  - It pursued the typical goals of a DFS: performance, availability, reliability, scalability
  - But also it was created taken into account Google's environment features and apps needs
  - HDFS was built following the same premises and architecture

HDFS Design Features

- HDFS design assumes that:
    - Failures are the norm (not the exception)
        - Constant monitoring, error detection, fault tolerance and automatic recovery are required
    - Files will be big (TBs), and contain billions of app objects
        - Small files are possible, but not a goal for optimization
    - Once written and closed, files are only read and often only sequentially
        - Batch-like analysis applications are the target
    - Only one writer at any time
    - High data access bandwidth is preferred to low latency for individual operations

HDFS Architecture

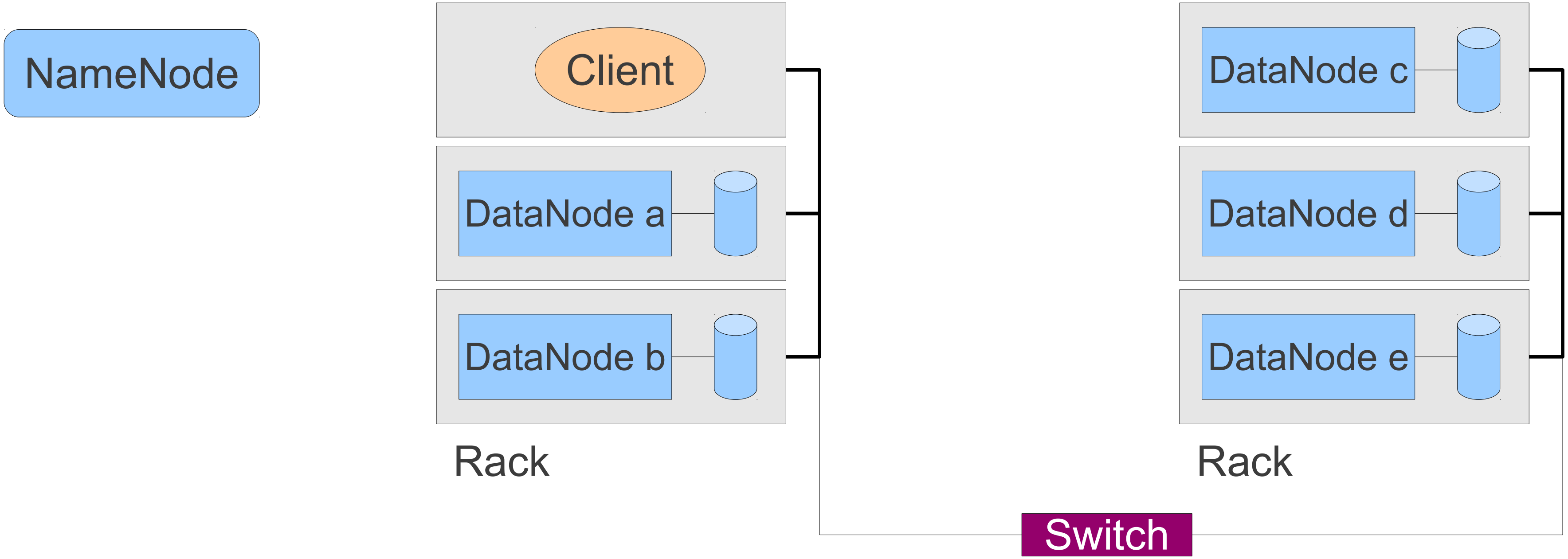
- HDFS files are split into **blocks**
- Two types of nodes:
  - ***NameNode***, unique, manages the namespace: maps file paths and names with blocks and their locations
  - ***DataNode***, keep data blocks and serves r/w requests from clients. Also, it decides where each data block replica is stored



File path	blocks	DataNode replicas
`/path_to_file`	b1	DataNode a, DataNode b, DataNode d
	b2	DataNode b, DataNode c, DataNode d

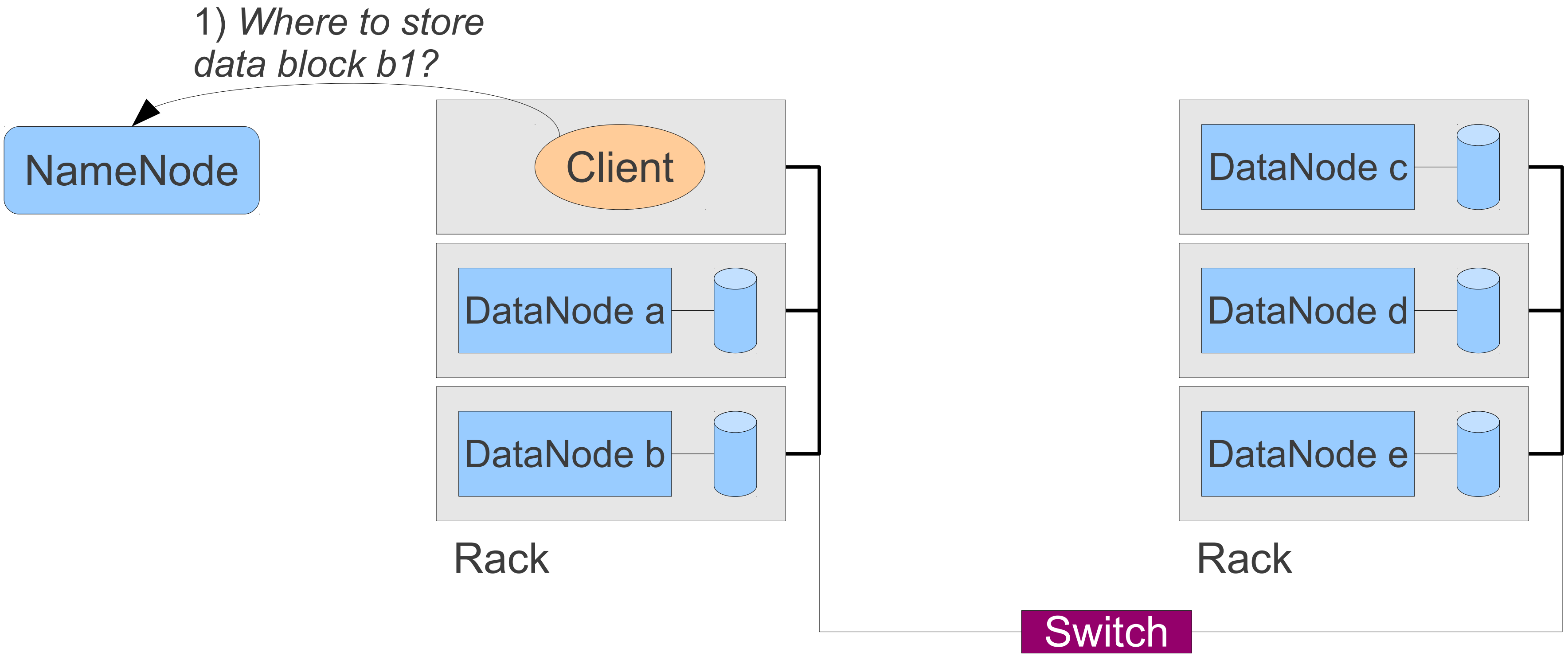
HDFS Replicas Placement

- Where to store each block replica is decided by the NameNode
  - Bandwidth among nodes in the same rack is greater than inter-racks
  - Default: 3 copies per block, one on local rack and the two others on a remote rack



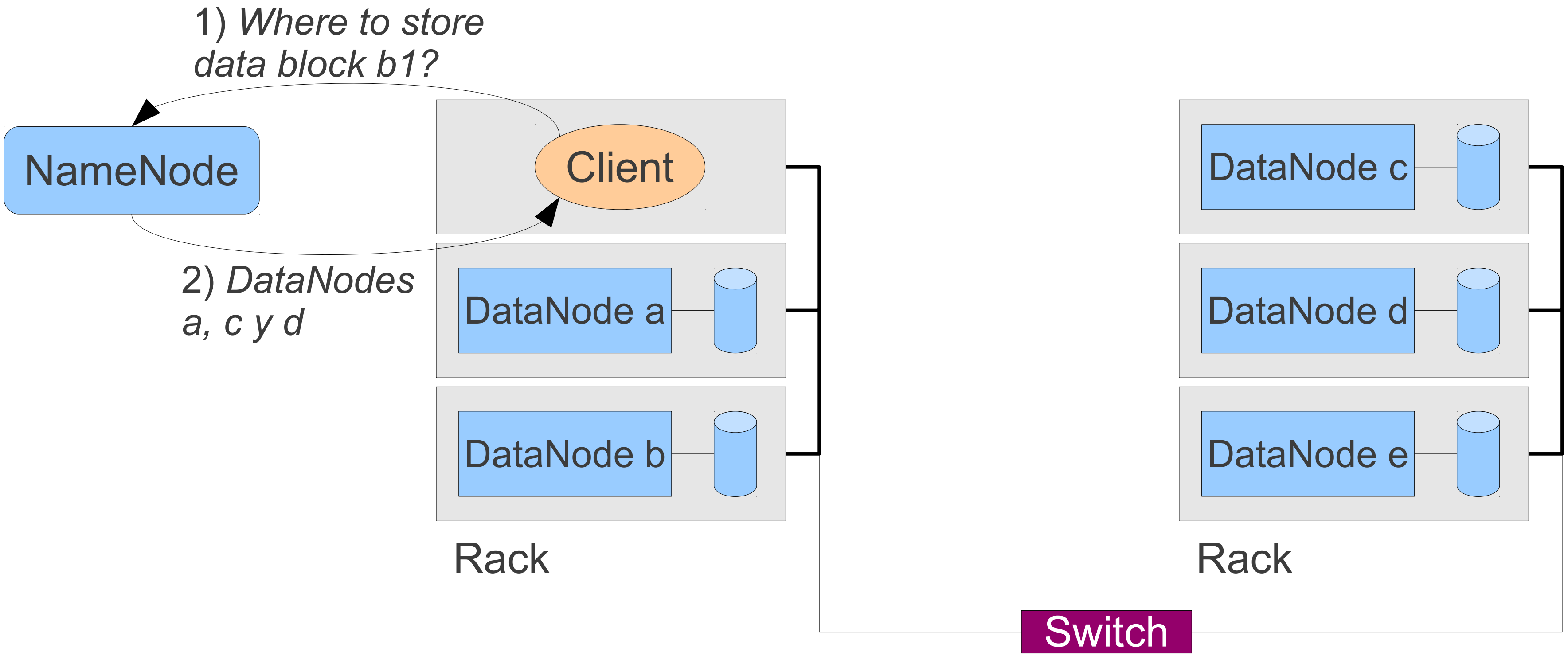
HDFS Replicas Placement

- Where to store each block replica is decided by the NameNode
  - Bandwidth among nodes in the same rack is greater than inter-racks
  - Default: 3 copies per block, one on local rack and the two others on a remote rack



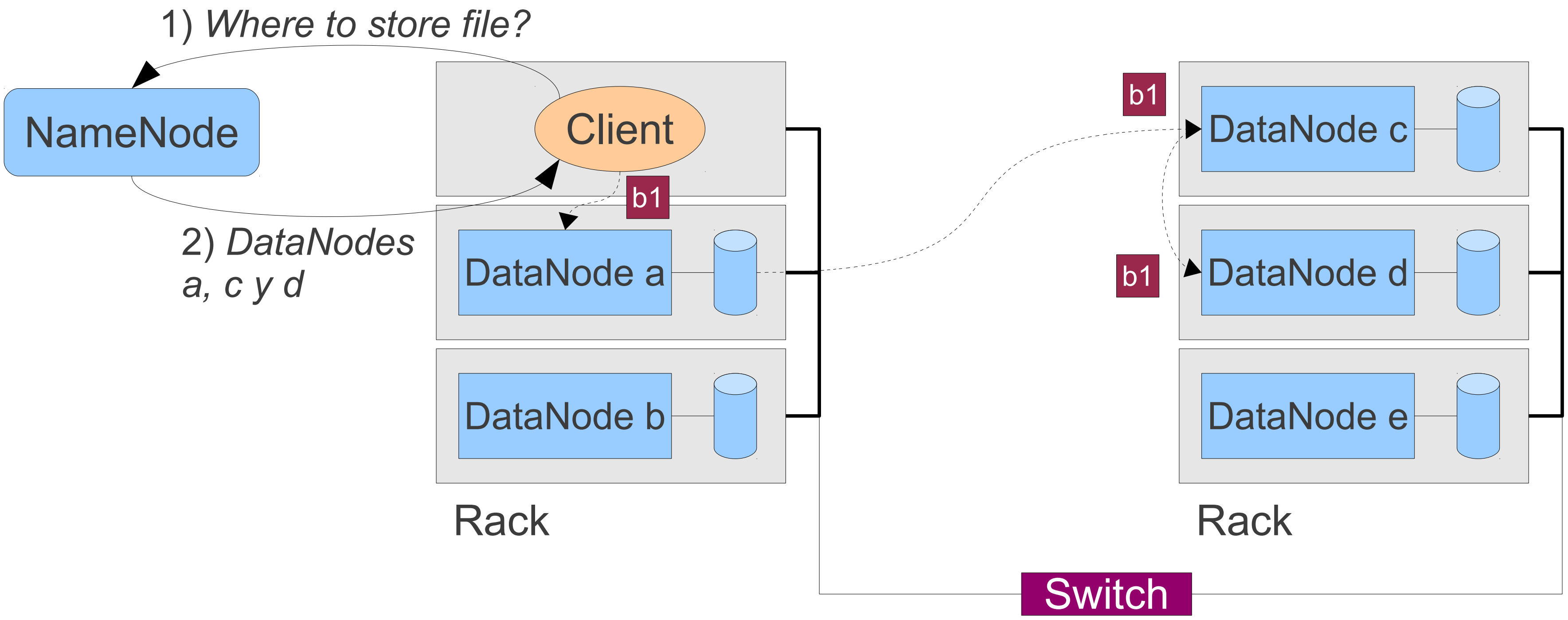
HDFS Replicas Placement

- Where to store each block replica is decided by the NameNode
    - Bandwidth among nodes in the same rack is greater than inter-racks
    - Default: 3 copies per block, one on local rack and the two others on a remote rack



HDFS Replicas Placement

- Where to store each block replica is decided by the NameNode
  - Bandwidth among nodes in the same rack is greater than inter-racks
  - Default: 3 copies per block, one on local rack and the two others on a remote rack



HDFS Replicas Placement (2)

- This placement police balances:
  - Availability: three nodes, or two racks must fail to make the block unavailable
  - Lower writing times: data moves only once through the switch
  - High read throughput: readers will access to closer blocks; by this policy is easy that the reader is in the same node (or at least the same rack) that the block

HDFS Consistency Considerations

- HDFS keeps several replicas of each data block
- Given that:
  - Once written and closed, files are only read
  - There will be only one writer at any time
- Then consistency among replicas is greatly simplified
  - GFS also makes some assumptions that simplify consistency management
  - However it is not as restrictive as HDFS: Concurrent writers are allowed
    - Potential inconsistencies must be addressed at app. level

HDFS NameNode

- Keeps the state of the HDFS, using
  - An in-memory structure that stores the filesystem metadata
  - A local file with a copy of that metadata at boot time (`FsImage`)
  - A local file that logs all HDFS file operations (`EditLog`)
- At boot time the NameNode:
  - Reads `FsImage` to get the HDFS state
  - Applies all changes recorded in `EditLog`
  - Saves the new state to the `FsImage` file (checkpoint)
  - Truncates the `EditLog` file
- At run time the NameNode:
  - Stores all file operations on the `EditLog`

HDFS Secondary NameNode

- Optionally, a secondary NameNode can be used
- It is used to merge the `FsImage` and `EditLog` of the NameNode to create a new `FsImage`
  - This prevents the `EditLog` to grow without bonds
  - It is a demanding process that demands time and would force the NameNode to be offline
  - Once the new `FsImage` is built, is sent back to the NameNode
- The Secondary NameNode **does not** replace the primary NameNode
  - Its copy of the `FsImage` could be used in case of failure, but it would lack the changes registered on the `EditLog`

HDFS Data Location Awareness

- Clients can query the NameNode about where data replicas are located (DataNodes URLs)
- This information can be used to “move code instead of data”
  - Data can be in the order of TBs
  - Moving data to the node where the processing sw is located would be inefficient
  - Moving code to the host(s) where data is stored is faster and demands less resources

Data Correctness

- HDFS checks the correctness of data using **checksums**
- The last node in the pipeline verifies that the checksum of the received block is correct
- Also, each DataNode periodically checks all its blocks checksum
- Corrupted ones are reported to the NameNode, who will replace them with the block from other replica

Other HDFS Features/APIs

- APIs for Compression (zip, bzip...)
- Easy conversion to binary formats when reading/writing
- `SequenceFile`, used to store sequential data as binary records in the form key-value
- `MapFile`, is a `SequenceFile` that allows random access by key (i.e. it is a 'permanente' map)

Interacting with HDFS

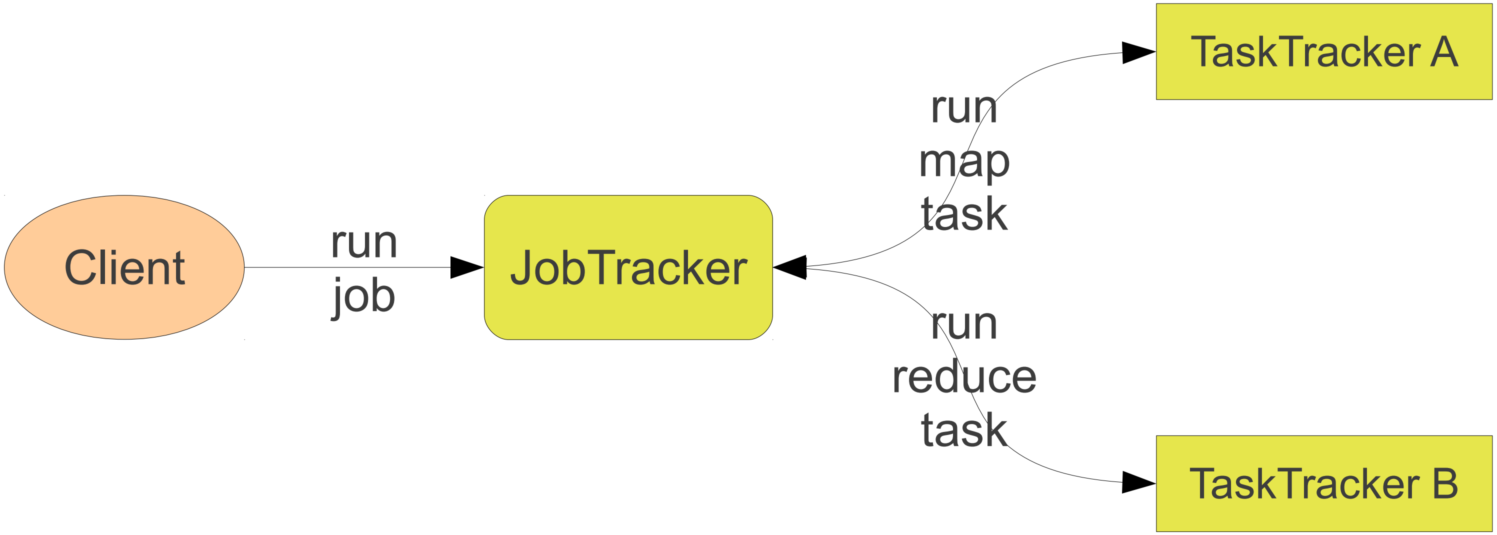
- HDFS offers a Java API based on `FileSystem`
  - Java apps can access HDFS as a typical file system
  - There is a C API with a similar interface
- It is possible to access to HDFS from the shell using the `hdfs` application
- HDFS can be accessed remotely through a RPC interface (based on Apache Thrift middleware)
- WebDAV, FTP, HTTP

Hadoop MapReduce

- Once we have seen how HDFS works, we can focus on Hadoop MapReduce
- It is a distributed implementation of the MapReduce abstraction
- It can work on top of the local file system
- But it is intended to work on top of HDFS

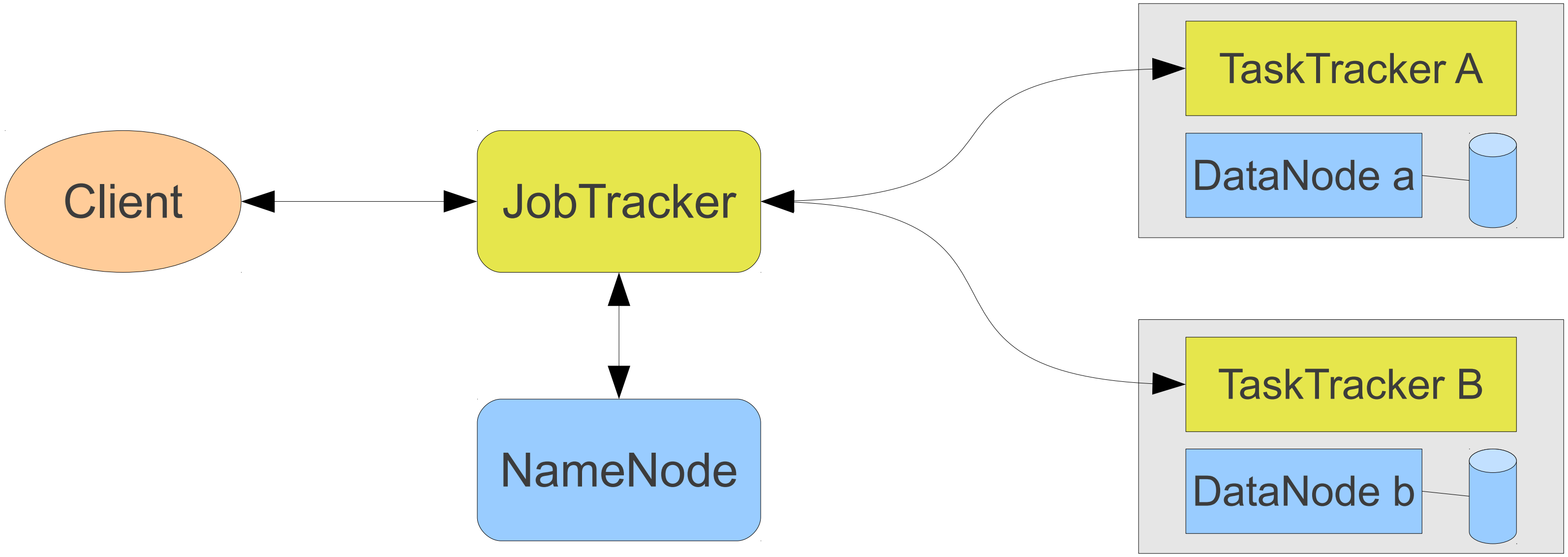
Hadoop MapReduce Architecture

- A MapReduce processing is called a **Job**, each Job is split into **Tasks**, where each task executes *map* or *reduce* over data
- Two types of nodes:
  - ***JobTracker***, unique, manages the Jobs and monitors the TaskTrackers
  - ***TaskTracker***, executes Map and Reduce functions as commanded by the JobTracker



Hadoop MapReduce Deployment

- Usually HDFS and MapReduce will run in the same datacenter
    - The job will be programmed so input data and results are stored in HDFS
- Each physical host can run one (or more) DataNodes and one (or more) TaskTrackers
    - This way, data locality can be exploited



Hadoop Code Example

- Example of MapReduce programming
- Problem
  - The input is a file that contains links between web pages as follows:

```
http://host1/path1 -> http://host2/path2
http://host1/path3 -> http://host2/path4
...
```

  - The output is a file that for each web page gives the number of outgoing and incoming connections:

```
http://host1/path1 2 4
http://host2/path2 0 1
...
```

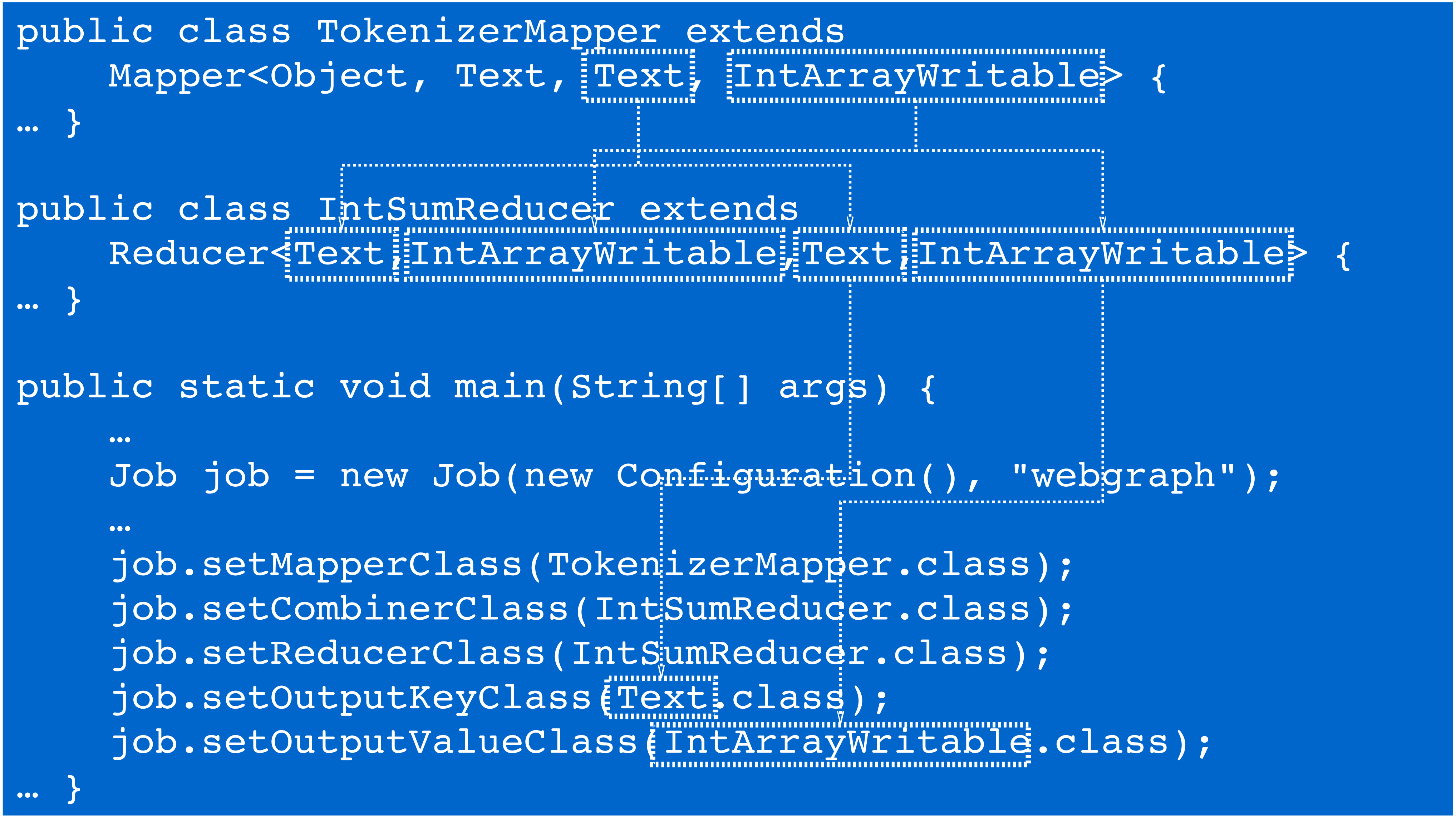
Hadoop Code Example, main

```
public class TokenizerMapper extends
    Mapper<Object, Text, Text, IntArrayWritable> {
… }

public class IntSumReducer extends
    Reducer<Text,IntArrayWritable,Text,IntArrayWritable> {
… }

public static void main(String[] args) {
    …
    Job job = new Job(new Configuration(), "webgraph");
    …
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntArrayWritable.class);
… }
```

Hadoop Code Example, main



```
public class TokenizerMapper extends
    Mapper<Object, Text, Text, IntArrayWritable> {
… }

public class IntSumReducer extends
    Reducer<Text, IntArrayWritable, Text, IntArrayWritable> {
… }

public static void main(String[] args) {
    …
    Job job = new Job(new Configuration(), "webgraph");
    …
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntArrayWritable.class);
… }
```

Hadoop Code Example, main

```
public class TokenizerMapper extends
    Mapper<Object, Text, Text, IntArrayWritable> {
… }

public class IntSumReducer extends
    Reducer<Text,IntArrayWritable,Text,IntArrayWritable> {
… }

public static void main(String[] args) {
    …
    Job job = new Job(new Configuration(), "webgraph");
    …
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntArrayWritable.class);
… }
```

Hadoop Code Example, main

```
public class TokenizerMapper extends
    Mapper<Object, Text, Text, IntArrayWritable> {
… }

public class IntSumReducer extends
    Reducer<Text,IntArrayWritable,Text,IntArrayWritable> {
… }

public static void main(String[] args) {
    …
    Job job = new Job(new Configuration(), "webgraph");
    …
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntArrayWritable.class);
… }
```

Hadoop Code Example, Mapper

```
public class TokenizerMapper extends Mapper<Object, Text, Text, IntArrayWritable> {

  private final static IntWritable one = new IntWritable(1);
  private final static IntWritable cero = new IntWritable(0);
  private final static IntArrayWritable origin =
      new IntArrayWritable(new IntWritable[]{one, cero});
  private final static IntArrayWritable destination =
      new IntArrayWritable(new IntWritable[]{cero, one});
  private Text word = new Text();
  …
}
```

Hadoop Code Example, Mapper (2)

```
public class TokenizerMapper extends Mapper<Object, Text, Text, IntArrayWritable> {
  …
  public void map(Object key, Text value, Context context) throws IOException,
                                                                  InterruptedException {
    BufferedReader reader = new BufferedReader(new StringReader(value.toString()));
    while(true) {
      String line = reader.readLine();
      if(line == null) {
        reader.close();
        break;
      }
      line = line.trim();
      if(line.isEmpty())
        continue;
      String[] urls = line.split(WebGraph.URLS_SEPARATOR);
      if(urls.length != 2) {
        context.setStatus("Malformed link found: " + value.toString());
        return;
      }
      String urlOrigin = urls[0]; String urlDest = urls[1];
      word.set(urlOrigin); context.write(word, origin);
      word.set(urlDest); context.write(word, destination);
    }
  }
}
```

Hadoop Code Example, Reducer

```
public class IntSumReducer extends Reducer<Text,IntArrayWritable,Text,IntArrayWritable>{

  private IntArrayWritable result = new IntArrayWritable();

  public void reduce(Text key, Iterable<IntArrayWritable> values, Context context)
      throws IOException, InterruptedException {
    int sumLinksOrig = 0;
    int sumLinksDest = 0;
    for(IntArrayWritable val: values) {
      Writable[] intArray = val.get();
      sumLinksOrig += ((IntWritable)intArray[0]).get();
      sumLinksDest += ((IntWritable)intArray[1]).get();
    }
    result.set(new IntWritable[]{new IntWritable(sumLinksOrig),
                                 new IntWritable(sumLinksDest)});
    context.write(key,result);

  }
}
```

Bibliography

- (Paper) "*The Google File System*". Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. SOSP 2003
- (Book) "Hadoop: The Definitive Guide" (2nd edition). Tom White. Yahoo Press, 2010

(Example with Hadoop)