Illinois Institute of Technology

Distributed Recommenders

Fall 2010

Distributed Recommenders

- Distributed Approaches are needed when:
  - Dataset does not fit into memory
  - Need for processing exceeds what can be provided with a sequential algorithm
- Traditionally distributed data mining algorithms were very time consuming to implement
- Map-Reduce framework can reduce complexity
- Mahout uses the Hadoop Map-Reduce framework
  - Slope-one (already implemented in Mahout)
  - Distributed Nearest Neighbor
    - User-Based
    - Item-Based

Overview of Hadoop

- Hadoop is an Apache project comprised of a distributed filesystem (HDFS) and a MapReduce [1] engine
- Hadoop enables applications to more easily distribute large computations across a cluster of Hadoop nodes
- HDFS breaks files into chunks, which are stored across nodes. This is needed since a typical data file used on a Hadoop cluster is larger than a single disk on the cluster.
- Generally multiple copies of each chunk are kept across different nodes for redundancy and efficiency
- A job tracker executes MapReduce jobs against data stored in HDFS

MapReduce

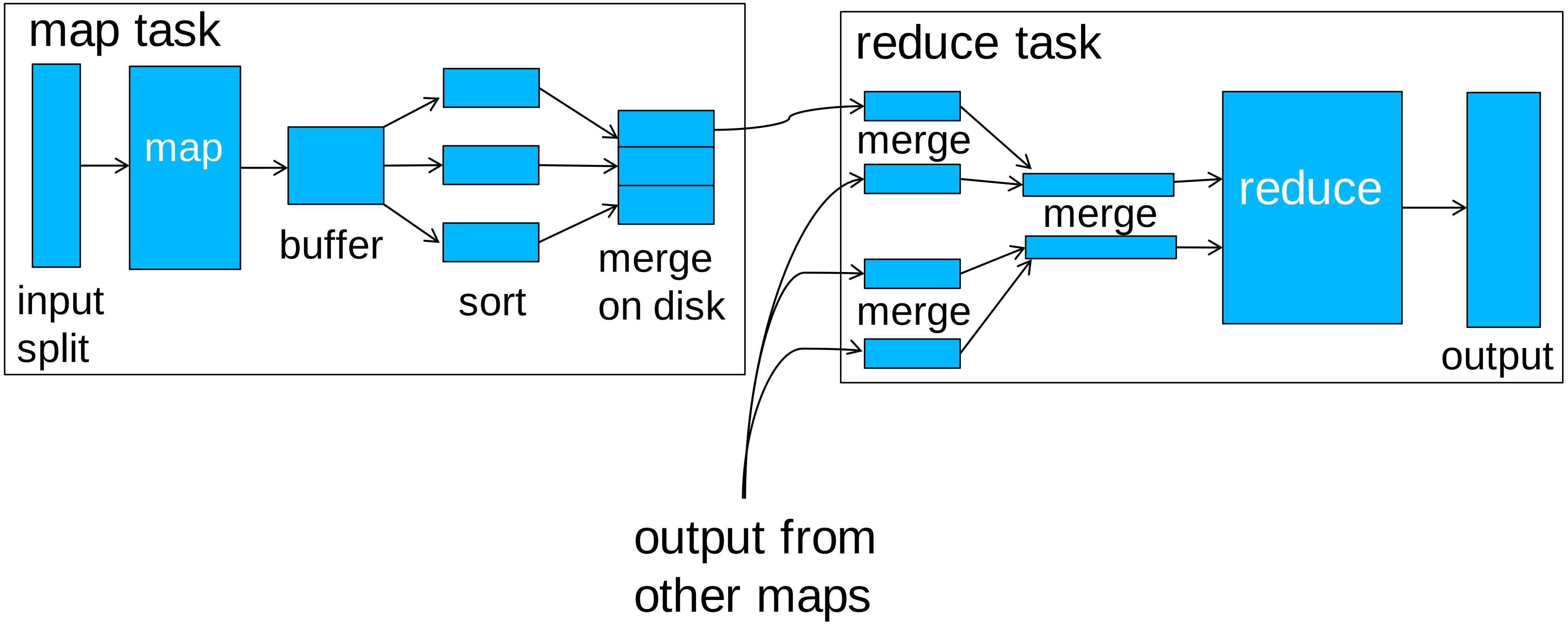
- MapReduce is a software framework for distributing computations across a cluster of computers
- Mappers and reducers are processes on the node that perform map and reduce steps, respectively
- Both steps take a key-value pair as input
  - Map: $(K_1, V_1) \rightarrow \text{list}(K_2, V_2)$
  - Reduce: $(K_2, \text{list}(V_2)) \rightarrow \text{list}\ (K_3, V_3)$

MapReduce

- Map step
  - A key and value are given to the mapper
  - The mapper performs an operation on its input and returns a key and value in a different domain
- The mappers' output is then sorted and the keys combined, so that each key is unique and paired with a list of the values output for that key
- Reduce step
  - A key and list of values are given to the reducer
  - The reducer transforms its input into a key and final value

MapReduce

- Map and reduce tasks consist of multiple steps



Components

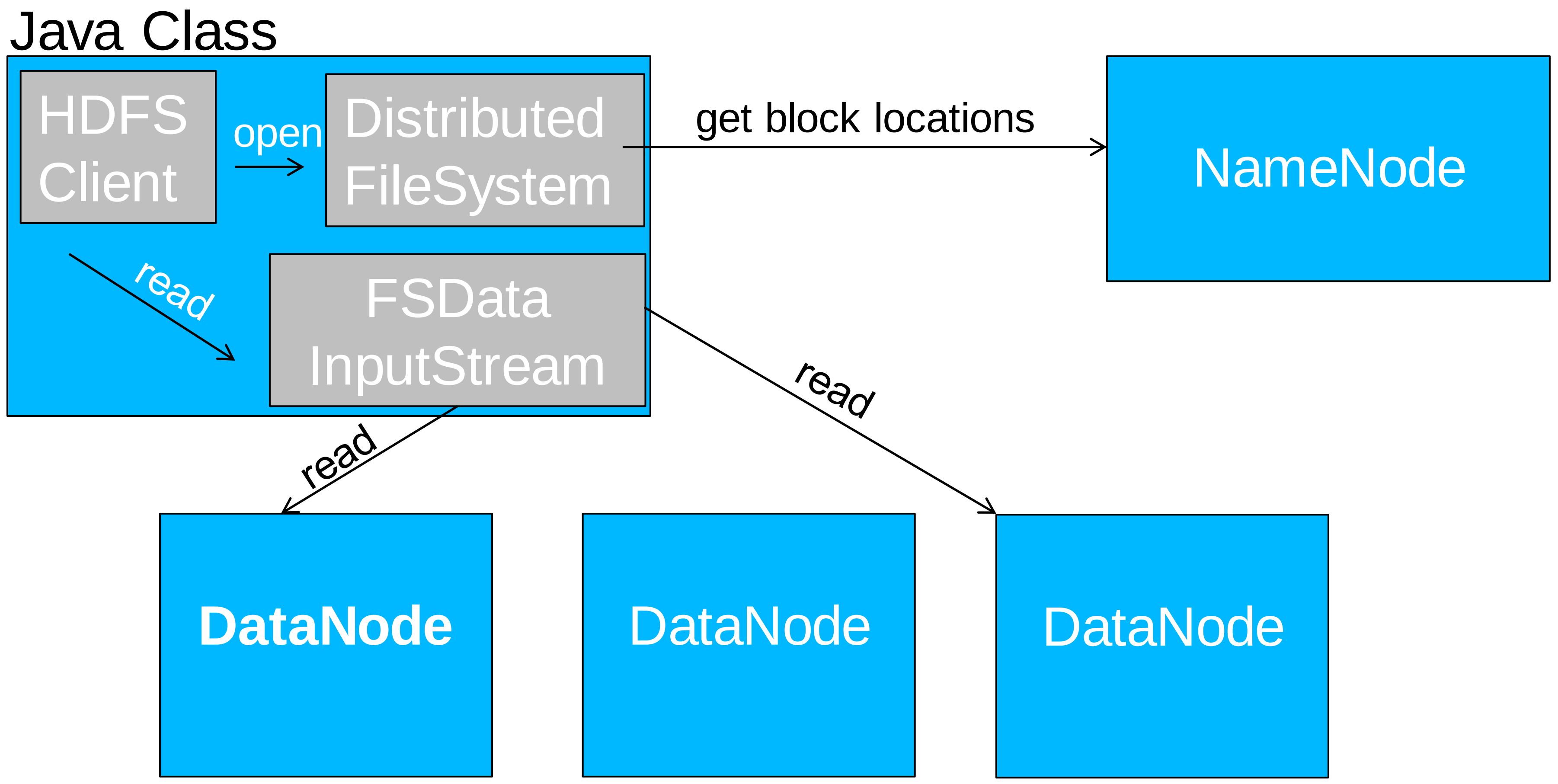
- Hadoop is comprised of several processes
  - Job Tracker
  - Task Tracker(s)
  - Name Node, Secondary Name Node
  - Data Node(s)
- The Job and Task Trackers handle MapReduce jobs
- The Name Nodes and Data Nodes provide the HDFS file system used by MapReduce jobs

Components

- The Name Node keeps track of file locations in the cluster
- The Job Tracker accepts MapReduce jobs and gives map and reduce tasks to nodes running a Tack Tracker
- Keeping multiple copies of files in the cluster allows the Job Tracker to better schedule tasks on nodes already containing the data needed for their map and reduce tasks
- Hadoop can be made aware of node location, so that tasks can be run on nodes close to the data required by the task

HDFS

- Example HDFS request on a 3 DataNode cluster
- DistributedFileSystem receives block locations from the NameNode, which are read from DataNodes by InputStream



MapReduce Example

Illinois Institute of Technology

- Given temperature readings from multiple weather stations over a 100 year period, with one file per station per year.
- Problem: Find the highest temperature for each year across all stations.
- This problem can be run on a cluster with map reduce.
- Two steps:
  - Map: Read a record and output the year and the temperature
  - Shuffle and Sort (done for you by the Map Reduce framework) puts all year information together
  - Reduce: Read all temperatures for a year and output the highest

MapReduce Example

- Mappers take data file line numbers paired with the corresponding lines
- Each mapper will parse the lines it receives and return a date-temperature pair indicating the temperature encountered
- Mapper algorithm
  - Read input lines
  - Find and store the year and temperature
  - Output a year-temperature pair

- Input: xxxxxxxxxxMM/DD/YYYYxxxxxxxxxxxxTTTT

- Output: YYYY   TTTT

MapReduce Example

- The Hadoop Job Tracker will collect the output of the various mappers, combining common keys and forming a list from their values
- This allows each reducer to work on all entries for a key
- Input:

1900	28.3
1900	29.1
1901	29
1901	29.4

- Output:

1900	28.3, 29.1
1901	29, 29.4

MapReduce Example

- Reducers receive year-temperature list pairs as input
- Each reducer computes the highest temperature in its input list paired with its year. Different reduces can work on different years at the same time.
- Reducer Algorithm
  - Read input temperature list
  - Find maximum temperature in the list
  - Return year-maximum temperature pair
- Each reducer finds the highest temperature for a given year
- The maximum number of reducers is limited by the number of years
- Input:

1900	28.3, 29.1

- Output:

1900	29.1

Map Example

```
private static final int MISSING = 9999; // temp to indicate no value

public void map(LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {

  String line = value.toString();
  String year = line.substring(15, 19); // format-specific offsets
  int airTemperature;
  airTemperature = Integer.parseInt(line.substring(87, 92));
  if (airTemperature != MISSING) {
    output.collect(new Text(year), new IntWritable(airTemperature));
  }
}
```

Reduce Example

Illinois Institute of Technology

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

  int maxValue = Integer.MIN_VALUE;
  while (values.hasNext()) {
    maxValue = Math.max(maxValue, values.next().get());
  }
  output.collect(key, new IntWritable(maxValue));
}
```

Mahout MapReduce – Slope One

Illinois Institute of Technology

- Mahout provides a parallelized implementation of Slope One's preprocessing step using Hadoop and MapReduce
- This requires two separate MapReduce jobs
  - Transform preferences into item-item pair differences
  - Transform lists of differences into average differences

Slope One Preprocessing Algorithm

```
for every item i
    for every other item j
        for every user u expressing a preference for both i and j
            add the difference in u's preference for i and j to an average
```

Mahout MapReduce – Slope One Distributed Algorithm

Illinois Institute of Technology

Sequential Slope One Preprocessing Algorithm
Step 1: Compute all preferences for a user

Read the input file, lets call it File A of the form:
[user, item, preference]
with n entries. Build a map of user → items for each user.

Step 2: For each user, compute the difference for each item pair and store the difference associated with each pair.

Compile lists of the form: <item1, item2> → (d1, d2, ….dj) where j is the number of times item1 has been obtained with item2.

Step 3: For each item pair compute its average difference from its difference list

Step 4: Output the results into a final result file: D <item1, item2, avg diff>

Mahout MapReduce – Slope One Distributed Algorithm

Distributed Slope One Preprocessing Algorithm
Step 1: Compute all preferences for a user

Read a portion of the input file, lets call it File A of the form:
[user, item, preference]
with n entries. Send n/p entries to each processor and build a map of user → items for each user.

Merge all p lists of user → items into a single File B of the form:
user → ((item1, pref1), (item2, pref2), … (itemk, prefk))

Step 2: Read File B in parallel and each processor now computes File C which is of the form: <item1, item2> → (d1, d2, ….dj) where j is the number of times item1 has been obtained with item2.

Step 3: Read File C in parallel and make each processor compute the average difference for (c/p) item pairs, where c is the total number of item pairs.

Step 4: Merge the results into a final result file: D <item1, item2, avg diff>

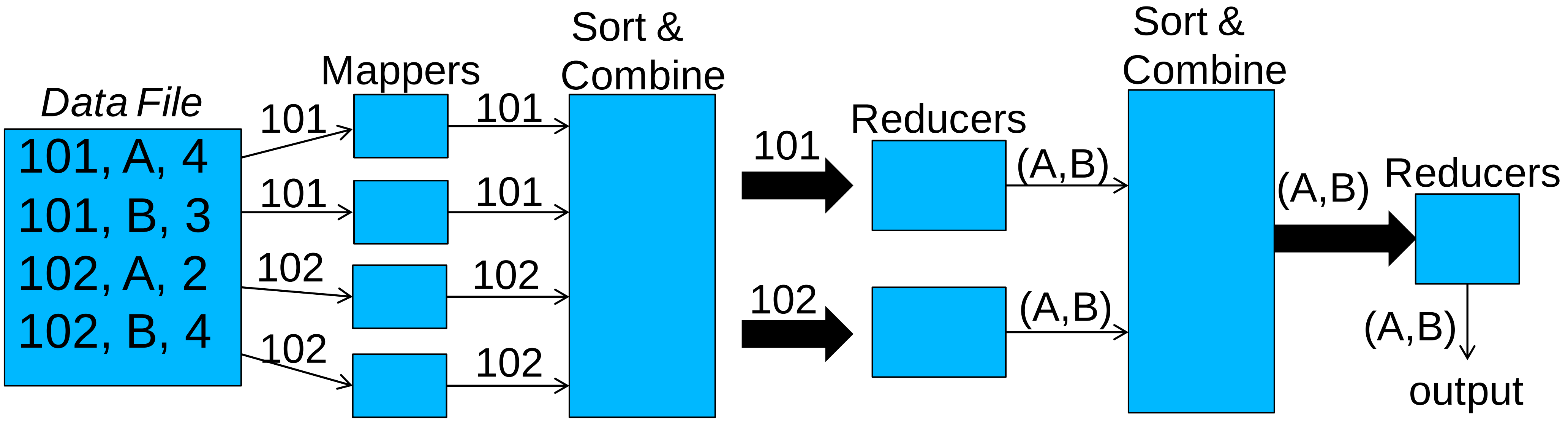
Slope One Performance

- Step 1: Compute all preferences for a user
  - With N users and I items, this will take
  - Distributed over p processors this is $\frac{}{}$
  - In the worst case every user has every item
- Step 2: Find pairs for all N users
  - $(\ )\quad \frac{}{(\quad)}$
  - $(\ )\quad \frac{}{(\quad)}\quad \frac{(\quad)(\quad)}{(\quad)}\quad \frac{(\quad)}{}\quad (\quad)$
  - We do this for each user, so it will take
  - With p processors this takes $\frac{}{}$
  - Sequentially we can compute running averages online
- Step 3: Combine pairs and compute averages (distributed only)
  - This is the same as step 2, because we use every pair
  - $(\qquad)$ or, when distributed, $\frac{}{}$

Mahout MapReduce – Slope One

Illinois Institute of Technology

- The map and reduce steps can be distributed across multiple mappers and reducers
- Each box represents a separate process, each of which can be running on the same or separate Hadoop nodes



Mahout MapReduce – Slope One

- Mappers read lines from the input data file and return userID-(itemID,preferenceValue) pairs
- This output is collected by Hadoop, sorted by user id, and the values of common keys combined

Data File

//user,item,pref
101, A, 4
101, B, 3
102, A, 2
102, B, 4

Mapper Output

Key (id)	*Value (item, pref)*
101	(A,4)
101	(B,3)
102	(A,2)
102	(B,4)

Hadoop Output

Key (id)	*Value (list of item, pref)*
101	(A,4), (B,3)
102	(A,2), (B,4)

Mahout MapReduce – Slope One

- Reducers are passed a userID and the user’s items and preferences from the mapper
- The reducers find the difference between each pair of userID’s items and return a (itemA,itemB)-difference pair for each

Reducer Input

Key (id)	***Value (item, pref)***
101	(A,4), (B,3)
102	(A,2), (B,4)

Reducer Output

Key (itemA, itemB)	***Value (difference)***
(A,B)	4-3= 1
(A,B)	2-4= -2

Mahout MapReduce – Slope One

- A new MapReduce job is run on the reducers' output
- An identity mapper is used, skipping the map step
- This output is collected and sorted, so that each (itemA,itemB) pair is associated with a list of differences between the items
- Reducers take this as input and return (itemA,itemB)-averageDifference pairs

Reducer Input

Key (itemA, itemB)	***Value (list of differences)***
(A,B)	1, -2

Reducer Output

Key (itemA, itemB)	***Value (average difference)***
(A,B)	____

Slope One Performance

- Step 1: Compute all preferences for a user
  - With N users and I items, this will take
  - Distributed over p processors this is $\frac{\ }{\ }$
  - In the worst case every user has every item
- Step 2: Find pairs for all N users
  - $(\ )\quad \frac{\ }{(\quad)}$
  - $(\ )\quad \frac{\ }{(\quad)}\quad \frac{(\quad)(\quad)}{(\quad)}\quad \frac{(\quad)}{\ }\quad (\quad)$
  - We do this for each user, so it will take
  - With p processors this takes $\frac{\ }{\ }$
  - Sequentially we can compute running averages online
- Step 3: Combine pairs and compute averages (distributed only)
  - This is the same as step 2, because we use every pair
  - $(\qquad)$ or, when distributed, $\frac{\ }{\ }$

References

[1] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. In Proceedings Sixth Symposium on Operating System Design and Implementation (OSDI'04), 2004.

Illinois Institute of Technology

Configuring Hadoop

- Hadoop can be downloaded as Hadoop Common from its website http://hadoop.apache.org
- Hadoop v0.21 is the preferred version
- Extract hadoop-0.21.0.tar.gz after downloading
- The resulting hadoop-0.21.0 directory is your Hadoop installation and will be referred to as HADOOP_INSTALL by the Hadoop documentation
- Install the Java 6 JDK if needed and note its installation directory for use configuring Hadoop
  - Something like C:\jdk1.6_02 on Windows
  - /usr/lib/jvm/java-6-sun on Ubuntu Linux

Configuring Hadoop

- Some files in HADOOP_INSTALL/conf need to be edited to configure Hadoop to run as a single-node cluster
- In conf/hadoop-env.sh, JAVA_HOME should be set to the JDK's installation directory
- Two variables need to be set to specify a hostname:port for the name node and job tracker
  - With a single node hostname can be localhost
  - conf/mapred-site.xml's mapreduce.jobtracker.address should be set to host:9002
  - conf/core-site.xml's fs.default.name is hostname:9001
  - For example:

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9001</value>
</property>
```

Configuring Hadoop – Passwordless SSH

Illinois Institute of Technology

- Hadoop uses SSH to start Hadoop components on various nodes in the cluster
- We only have one node, but the startup scripts will still SSH to localhost in order to run component startup commands
- OpenSSH‘s sshd needs to be installed
  - If using Cygwin, make sure the openssh packages are installed and run ssh-host-config
  - If using Ubuntu Linux, install openssh-server
- Now run ssh-keygen to generate a public and private key for use with SSH's public key authentication
- When prompted use a blank passphrase
- Now you should be able to SSH to your local machine with no password using the command: ssh localhost

Starting Hadoop

Illinois Institute of Technology

- After Hadoop is configured, it can be started by running bin/start-all.sh from the HADOOP_INSTALL directory
- It can be stopped with bin/stop-all.sh
- A HADOOP_INSTALL/logs directory will be created
  - Try looking at the .log files inside if something goes wrong
- Try writing a file to HDFS as /test
  - bin/hadoop dfs –put <filename> /test
- You should be able to view it in HDFS along with other files already created by Hadoop
  - bin/hadoop dfs –ls /
- You can also copy it from HDFS to the local filesystem
  - bin/hadoop dfs –get /test test-file-from-HDFS