INFSCI 2750: Cloud Computing

Lecture 3: MapReduce

Dr. Balaji Palanisamy
Associate Professor
School of Computing and Information
University of Pittsburgh
bpalan@pitt.edu

Slides Courtesy: Prof. Bina Ramamurthy, University of Buffalo
Prof. Keke Chen, Wright State University
Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007

Outline

- Goals
- Programming model
- Examples
- Working mechanism
- Using hadoop mapreduce

Goals

- Understand the mapreduce programming model
  - Learn from some simple examples
- Understand how it works with the GFS (or HDFS)

Background

- Processing large datasets
- Computations are conceptually straightforward
- Parallel and distributed solutions are required
  - Data are distributed
  - Parallelized algorithms
  - Failures are norm
  - Easiness of programming

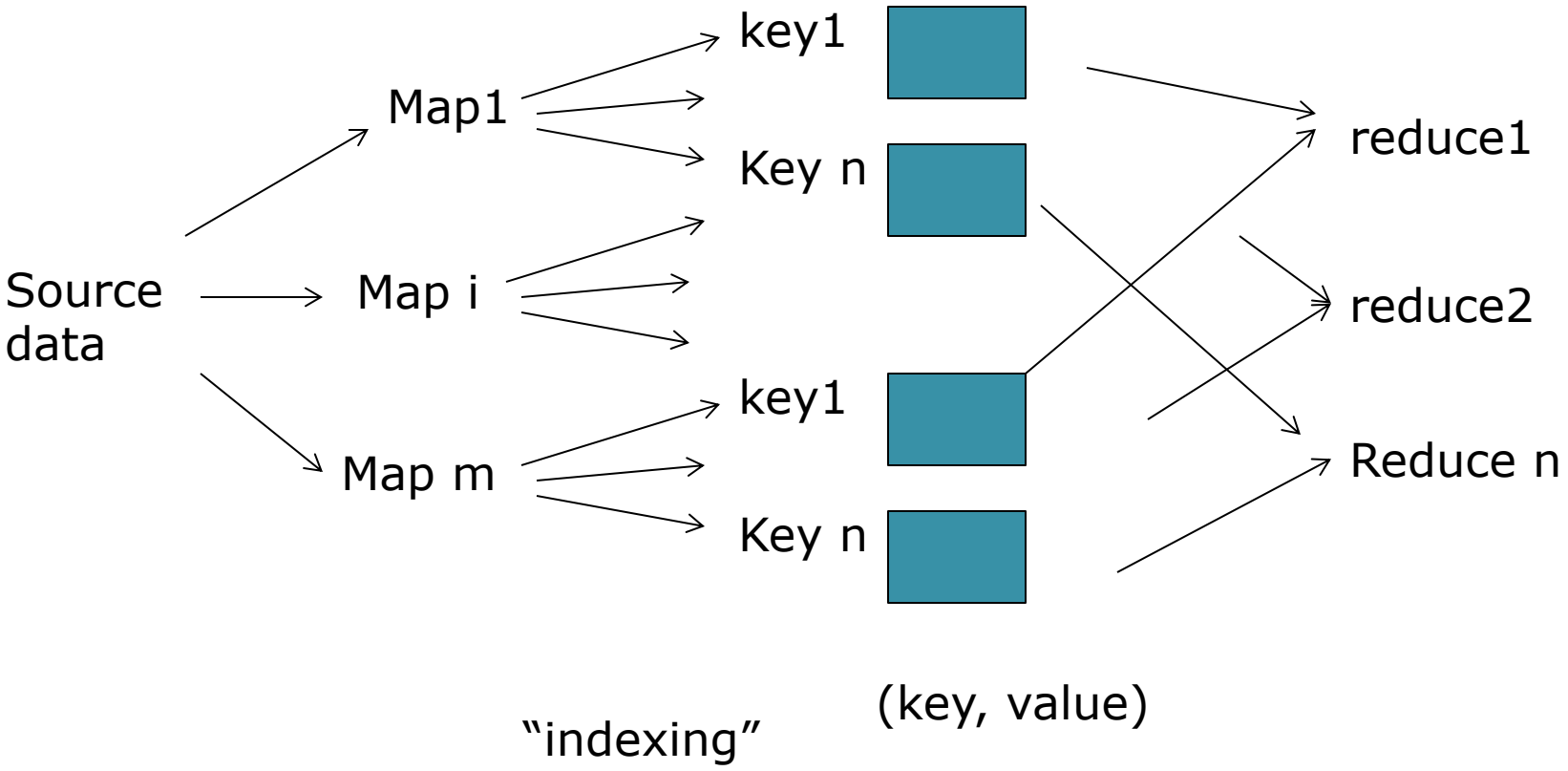
Design ideas

- Simple and powerful interface for programming
  - Application developers do not need to care about data management, failure handling, and algorithms coordinating distributed servers.

Mapreduce programming model

- Long history in programming language
  - Commonly used in functional programming (starting from 1930's, lambda calculus)
- Map function and reduce function
  - Applications need to encode the logic in these two functions
  - Complicated jobs might be implemented with multiple MapReduce programs

Basic ideas


key1
Map1
Key n
reduce1
Source data
Map i
reduce2
key1
Map m
Reduce n
Key n
(key, value)
"indexing"

- Example: document indexing
  - Map
    - Input: documents (DocID, document),
    - Output: (word, (DocID, position))
    - Break down documents to words
  - Reduce
    - Input: list of (word, (DocID, position)) with the same word
    - Output: (word, list of (DocID, position))

Map function

- How it works
  - Input data: tuples (e.g., lines in a text file)
  - Apply user-defined function to process data by keys
  - Output (key, value) tuples
    - The definition of the output keys is normally different from the input
- Under the hood:
  - The data file is split and sent to different distributed maps (that the user does not know)
  - Results are grouped by key and stored to the local linux file system of the map

Reduce function

- How it works
  - Group mappers' output (key, value) tuples by key
  - Apply a user defined function to process each group of tuples
  - Output: typically, (key, aggregates)

- Under the hood
  - Each reduce handles a number of keys
  - Reduce pulls the results of assigned keys from maps' results
  - Each reduce generates one result file in the GFS (or HDFS)

Summary of the ideas

- Mapper generates some kind of index for the original data
- Reducer apply group/aggregate based on that index
- Flexibility
  - Developers are free to generate all kinds of different indices based on the original data
  - Thus, many different types jobs can be done based on this simple framework

Example: grep

- Find keywords from a set of files
- Use Maps only
  - Input: (file, keyword)
  - Output (list of positions)

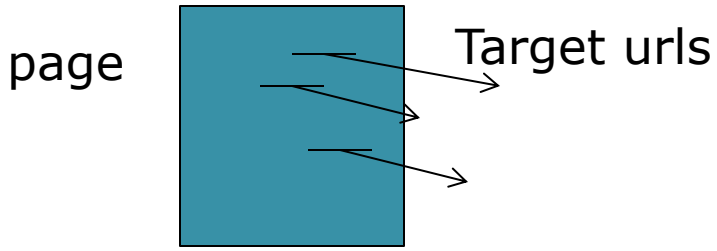
Example: count URL access frequency

- Work on the log of web page requests
  - (session ID, URL)…
- Map
  - Input: URLs
  - Output: (URL, 1)
- Reduce
  - Input (URL, 1)
  - Output (URL, counts)

Note: example 1's workload is on maps, while example 2's workload is on reduces

Example: reverse web-link graph

- Each source page has links to target pages, find out (target, list (sources))



- Map
  - Input (src URL, page content)
  - Output (tgt URL, src URL)
- Reduce
  - Output (tgt URL, list(src URL))

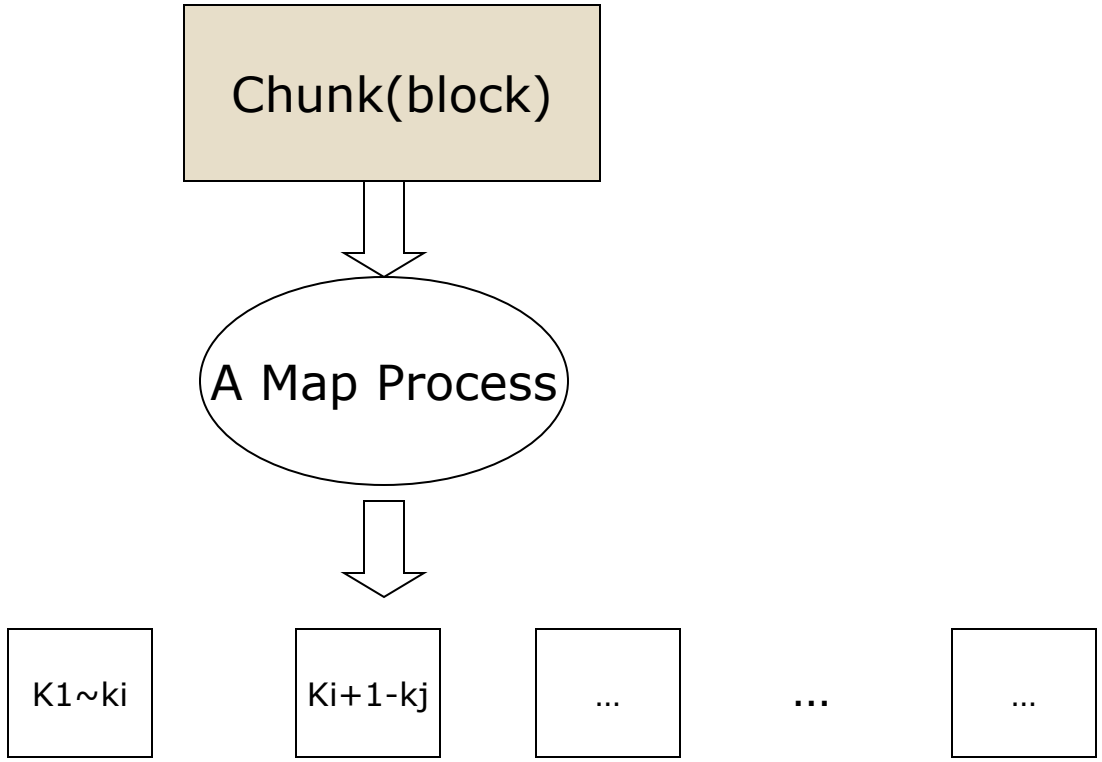
More examples

- Can implement more complicated algorithms
  - Sort
  - PageRank
  - Join tables
  - matrix computation, machine learning and data mining algorithms, e.g., the Mahout library

Implementation

- Based on GFS/HDFS
  - Holds all assumptions that GFS/HDFS holds
  - Main tasks: handle job scheduling and failures
- Assume there are
  - M map processes
  - R reduce processes

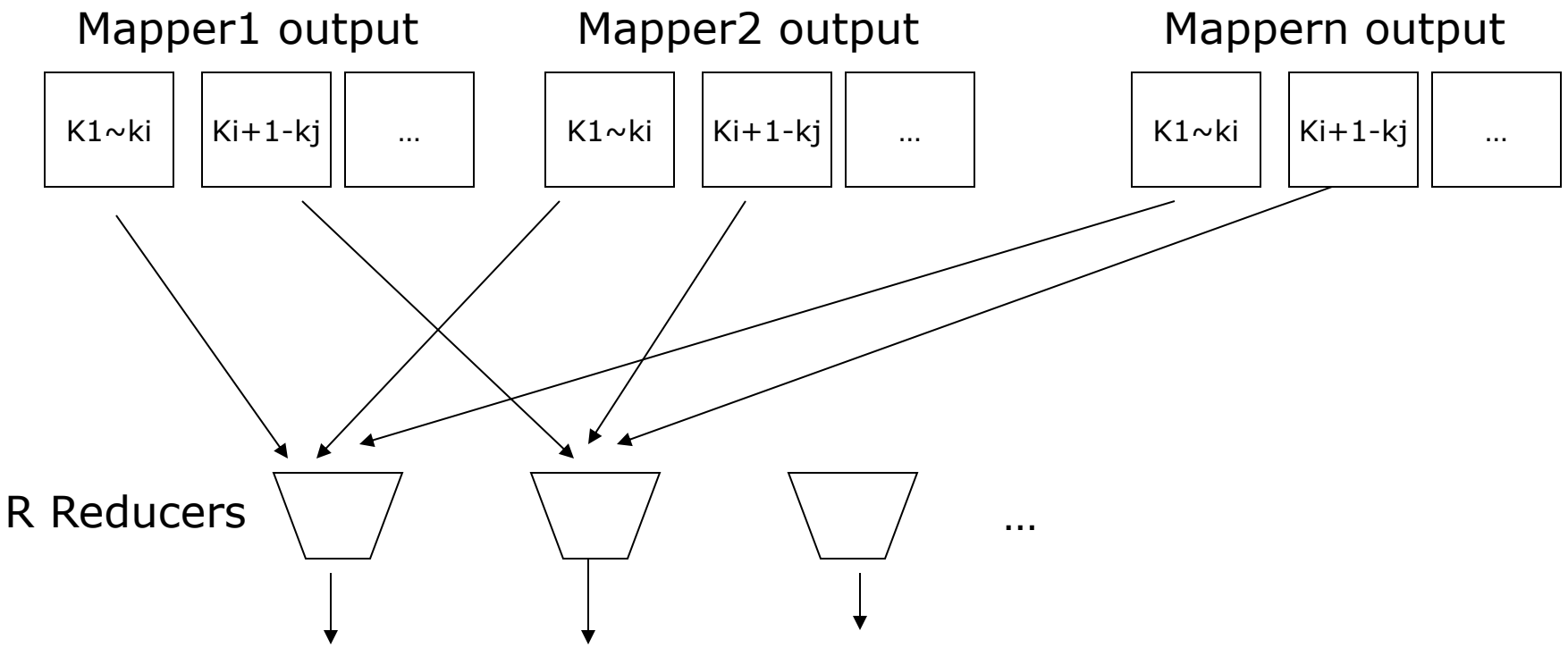
Map implementation



R parts in map's local storage

Map processes are allocated to be close to the chunks as possible
One node can run a number of map processes. It depends on the setting.

Reducer implementation



- R final output files stored in the user designated directory

Fault Tolerance

- Worker failure
  - Map/reduce fails → reassign to other workers
  - Node failure → redo all tasks in other nodes
    - Chunk replicas make it possible
- Master failure
  - Log/checkpoint
  - Master process and GFS master server

In Real World

- Applied to various domains in Google
  - Machine learning
  - Clustering
  - reports
  - Web page processing
  - indexing
  - Graph computation
  - …
- Mahout library
- Research projects

- Hadoop streaming example
- Hadoop java API
  - Framework
  - important APIs

Using hadoop mapreduce

- Programming with java library
  - http://hadoop.apache.org/common/docs/current/mapred_tutorial.html
  - Help you understand controls at detailed level
- More convenient: hadoop-streaming
  - Allow you to plug in map/reduce functions
  - Map/reduce functions can be scripts/exe/unix commands

Wordcount

- Problem: counting frequencies of words for a large document collection.
- Implement mapper and reducer respectively, using python
  - Some good python tutorials at http://wiki.python.org/

Mapper.py

```
import sys


for line in sys.stdin:
    line = line.strip()
  words = line.split()
  for word in words:
      print '%s\t1' % (word)
```

Reducer.py

```
import sys

word2count={}

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
            count = int(count)
            word2count[word] = word2count.get(word, 0)+ count
    except ValueError:
            pass

for word in word2count:
            print '%s\t%s'% (word, word2count[word])
```

Running wordcount

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -mapper "python mapper.py" \
    -reducer "python reducer.py" \
    -input text -output output2 \
    -file /localpath/mapper.py -file /localpath/reducer.py
```

Running wordcount

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -mapper "python mapper.py" \
    -reducer "python reducer.py" \
    -input text -output output2 \
    -file mapper.py -file reducer.py \
    -jobconf mapred.reduce.tasks=2 \
    -jobconf mapred.map.tasks=4
```

- If mapper/reducer takes files as parameters

```
hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -mapper "python mapper.py" \
    -reducer "python reducer.py myfile" \
    -input text -output output2 \
    -file /localpath/mapper.py -file /localpath/reducer.py
    -file /localpath/myfile
```

Web interfaces

- Tracking mapreduce jobs http://localhost:50030/
- http://localhost:50060/ - web UI for task tracker(s)
- http://localhost:50070/ - web UI for HDFS name node(s)

  - lynx http://localhost:50030/

Hadoop Java APIs

- hadoop.apache.org/common/docs/current/api/
- benefits
  - Jave code is more efficient than streaming
  - More parameters for control and tuning
  - Better for iterative MR programs

Important base classes

- Mapper<keyIn, valueIn, keyOut, valueOut>
  - Function map(**Object**, **Writable**, Context)
- Reducer<keyIn, valueIn, keyOut, valueOut>
  - Function reduce(**WritableComparable**, **Iterator**, Context)
- Combiner
- Partitioner

The framework

```
public class Wordcount{
        public static class MapClass extends
        Mapper<Object, Text, Text, LongWritable> {
            public void setup(Mapper.Context context){…}
           public void map(Object key, Text value,  Context context)
                    throws IOException {…}
        }

        public static class ReduceClass
        Reducer<Text, LongWritable, Text, LongWritable> {
            public void setup(Reducer.Context context){…}
            public void reduce(Text key, Iterator<LongWritable>
        values, Context context) throws IOException{…}
        }

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

The wordcount example in java

- http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#Example%3A+WordCount+v1.0
- Old/New framework
  - Old framework for version prior to 0.20

Mapper of wordcount

```
public static class WCMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

WordCount Reducer

```
 public static class WCReducer
       extends Reducer<Text,IntWritable,Text,IntWritable>
{
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable>
                       values, Context context
                      ) throws IOException,
                          InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

How to run your app

1. Compile to jar file
2. Command line

   hadoop jar your_jar your_parameters

   - Normally you need to pass in
     - Number of reducers
     - Input files
     - Output directory
     - Any other application specific parameters

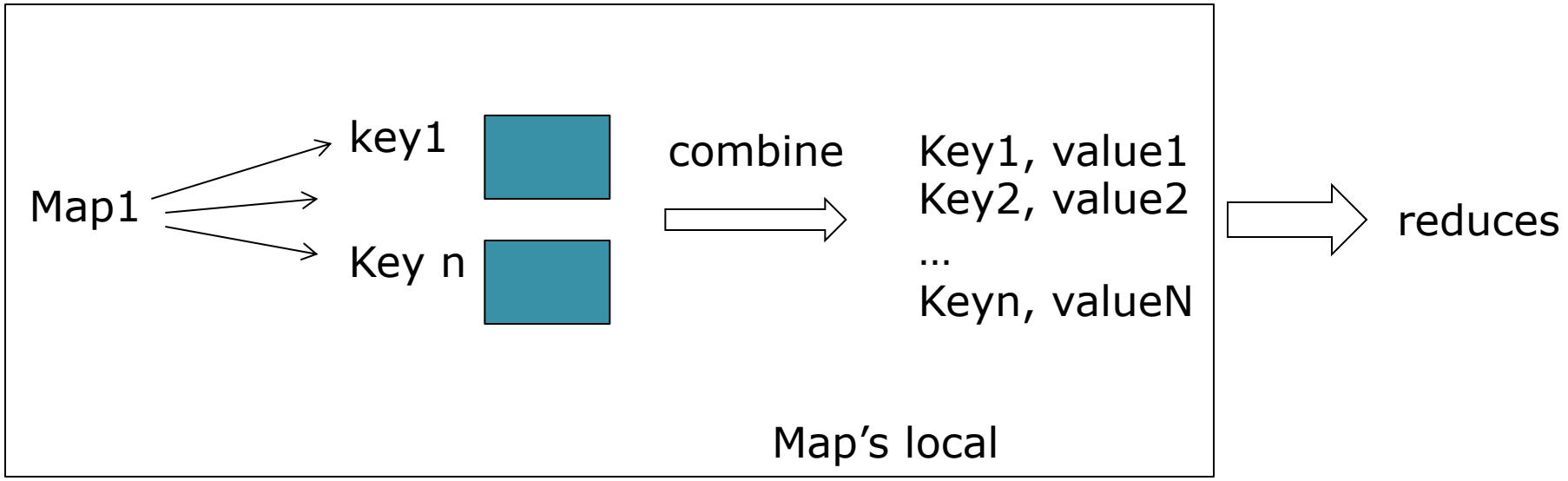
Access Files in HDFS?

Example: In map function

```
Public void setup(Mapper.Context context){
    Configuration conf = context.getConfiguration();
    string filename = conf.get("configfile"); // "configfile" parameter set
in main()

    Path p = new Path(filename);  // Path is used for opening the file.

    FileSystem fs = FileSystem.get(conf);//determines local or HDFS

    FSInputStream file = fs.open(p);

    while (file.available() > 0){
          …
    }
    file.close();

  }
```

Combiner

- Apply reduce function to the intermediate results locally after the map generates the result



Partitioner

- If map's output will generate N keys (N>R, R:# of reduces)
  - By default, N keys are randomly distributed to R reduces
  - You can use partitioner to define how the keys are distributed to the reduces.

More basic examples

- Check the code at /usr/local/hadoop/src/examples

- Check the binary with the command

  Hadoop jar /usr/local/hadoop/hadoop*example*.jar

Using virtual machines for course projects

- Preparation
  - Download putty.exe (windows)
    - http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
  - Use ssh directly on mac and linux

accounts

- Each student gets a VM
  - Access the vm using the IP address (for example)
    - Hostname: IP address
  - Root password

HDFS commands

- http://hadoop.apache.org/common/docs/r0.20.0/hdfs_shell.html
  - Try each listed command
- Examples:
  - hadoop fs –ls HDFS_directory
  - hadoop fs –put local_file HDFS_file
  - hadoop fs –get HDFS_file local_file
  - Other: -cat, -mkdir, -rm, -rmr

Test run mapreduce

1. Upload a text file
2. hadoop jar /usr/local/hadoop/hadoop*examples*.jar wordcount your_hdfs_files output_dir
3. hadoop fs –ls output_dir

Debates on mapreduce

- Most opinions are from the database camp
- Database techniques for distributed data store
  - Parallel databases
- traditional database approaches
  - Scheme
  - SQL

DB researchers' initial response to MapReduce

- "Mapreduce: a giant step backword"
  - A sub-optimal implementation, in that it uses brute force instead of indexing
  - Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
  - Missing most of the features that are routinely included in current DBMS
  - Incompatible with all of the tools DBMS users have come to depend on

- A giant step backward in the programming paradigm for large-scale data intensive applications
  - Schema
  - Separating schema from code
  - High-level language
- Responses
  - MR handles large data having no schema
  - Takes time to clean large data and pump into a DB
  - There are high-level languages developed: pig, hive, etc
  - Some problems that SQL cannot handle
  - Unix style programming (pipelined processing) is used by many users

- A sub-optimal implementation, in that it uses brute force instead of indexing
  - No index
  - Data skew : some reducers take longer time
  - High cost in reduce stage: disk I/O
- Responses
  - Google's experience has shown it can scale well
  - Index is not possible if the data has no schema
  - Mapreduce is used to generate web index
  - Writing back to disk increases fault tolerance

- Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
  - Hash-based join
  - Teradata

- Responses
  - The authors do not claim it is novel.
  - many users are already using similar ideas in their own distributed solutions
  - Mapreduce serves as a well developed library

- Missing most of the features that are routinely included in current DBMS
  - Bulk loader, transactions, views, integrity constraints …
- Responses
  - Mapreduce is not a DBMS, designed for different purposes
  - In practice, it does not prevent engineers implementing solutions quickly
  - Engineers usually take more time to learn DBMS
  - DBMS does not scale to the level of mapreduce applications

- Incompatible with all of the tools DBMS users have come to depend on
- Responses
  - Again, it is not DBMS
  - DBMS systems and tools have become obstacles to data analytics ☺

Some important problems

- Experimental study on scalability
- High-level language

Experimental study

- Sigmod09 "A comparison of approaches to large scale data analysis"
  - Compare parallel SQL DBMS (anonymous DBMS-X and Vertica) and mapreduce (hadoop)
  - Tasks
    - Grep
    - Typical DB tasks: selection, aggregation, join, UDF

Grep task

- 2 settings: 535M/node, 1TB/cluster

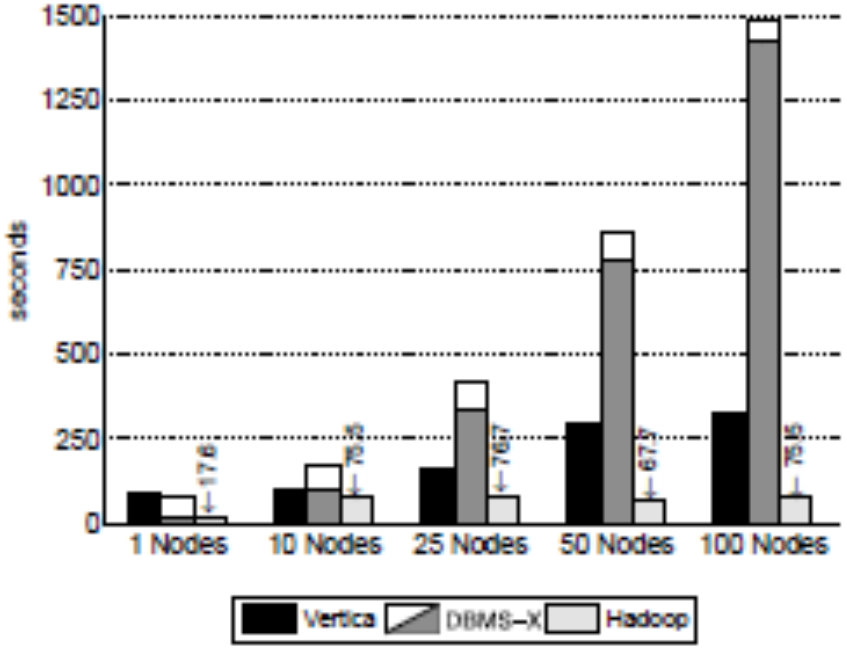


Figure 1: Load Times – Grep Task Data Set (535MB/node)

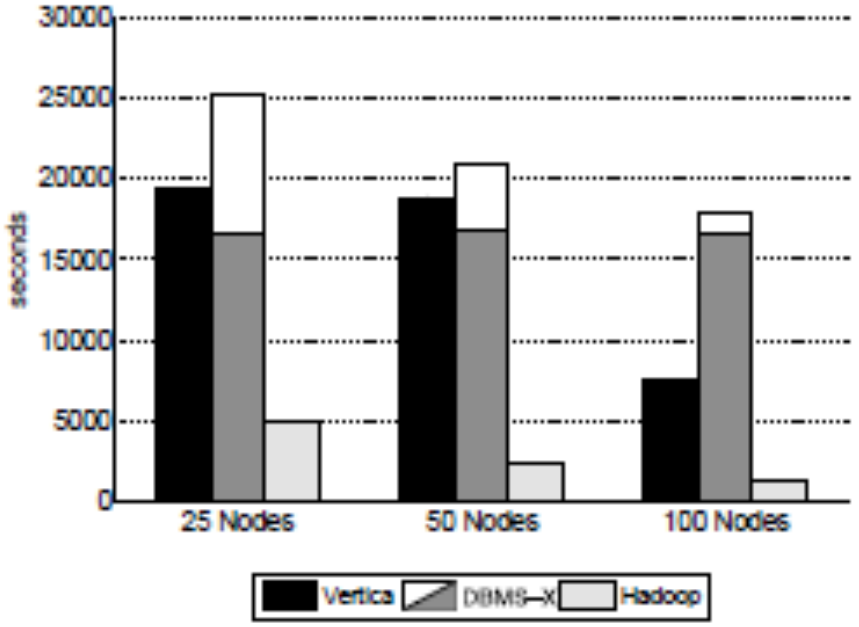


Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

Hadoop is much faster in loading data

Grep: task execution

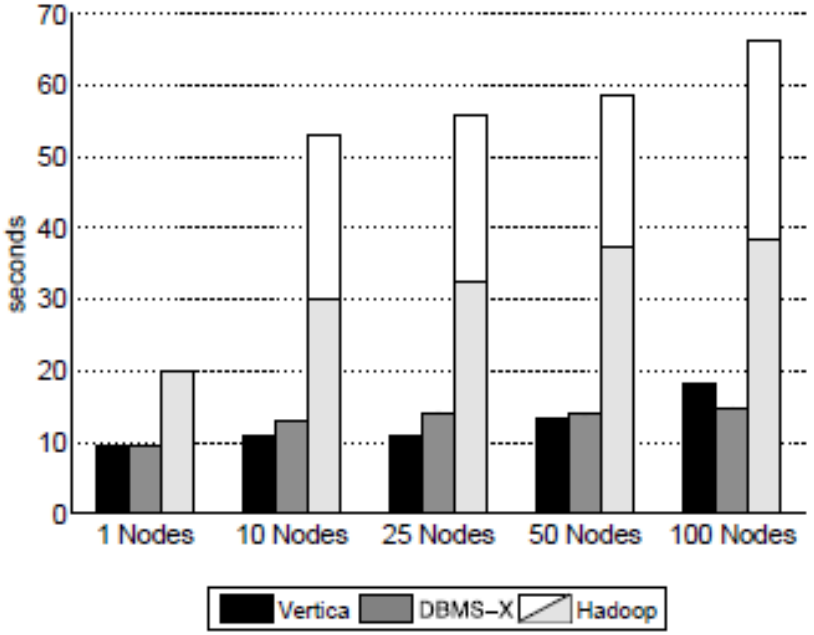


Figure 4: Grep Task Results – 535MB/node Data Set

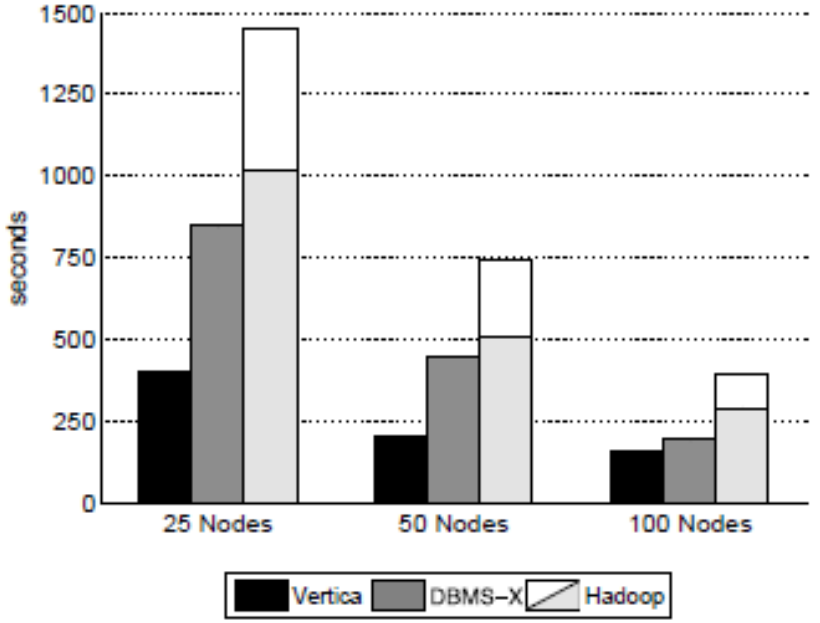


Figure 5: Grep Task Results – 1TB/cluster Data Set

Hadoop is the slowest…

Aggregation task

- Select sourceIP, sum(adRevenue) from Uservisits group by source IP

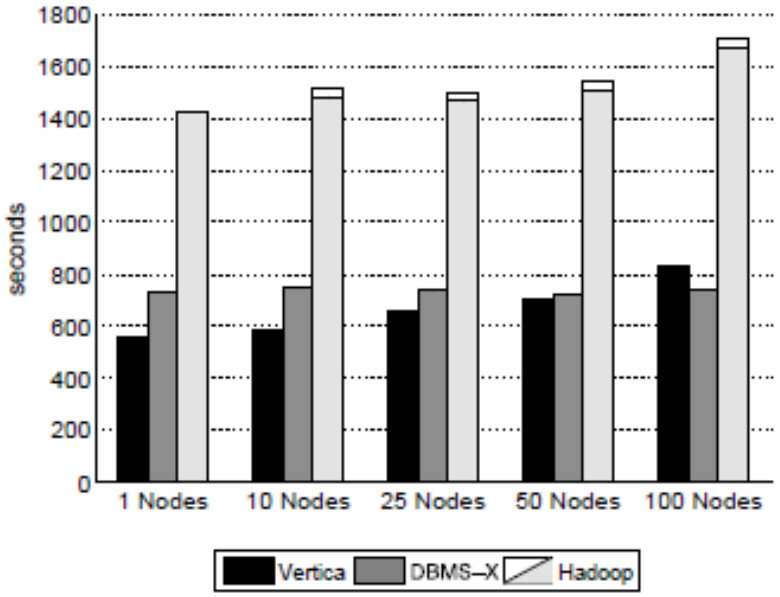


Figure 7: Aggregation Task Results (2.5 million Groups)

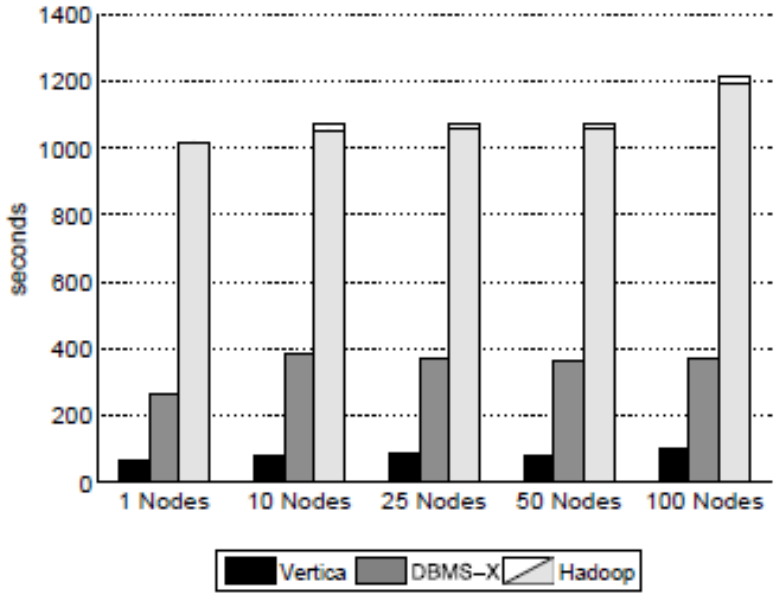


Figure 8: Aggregation Task Results (2,000 Groups)

Join task

- Mapreduce takes 3 phases to do it

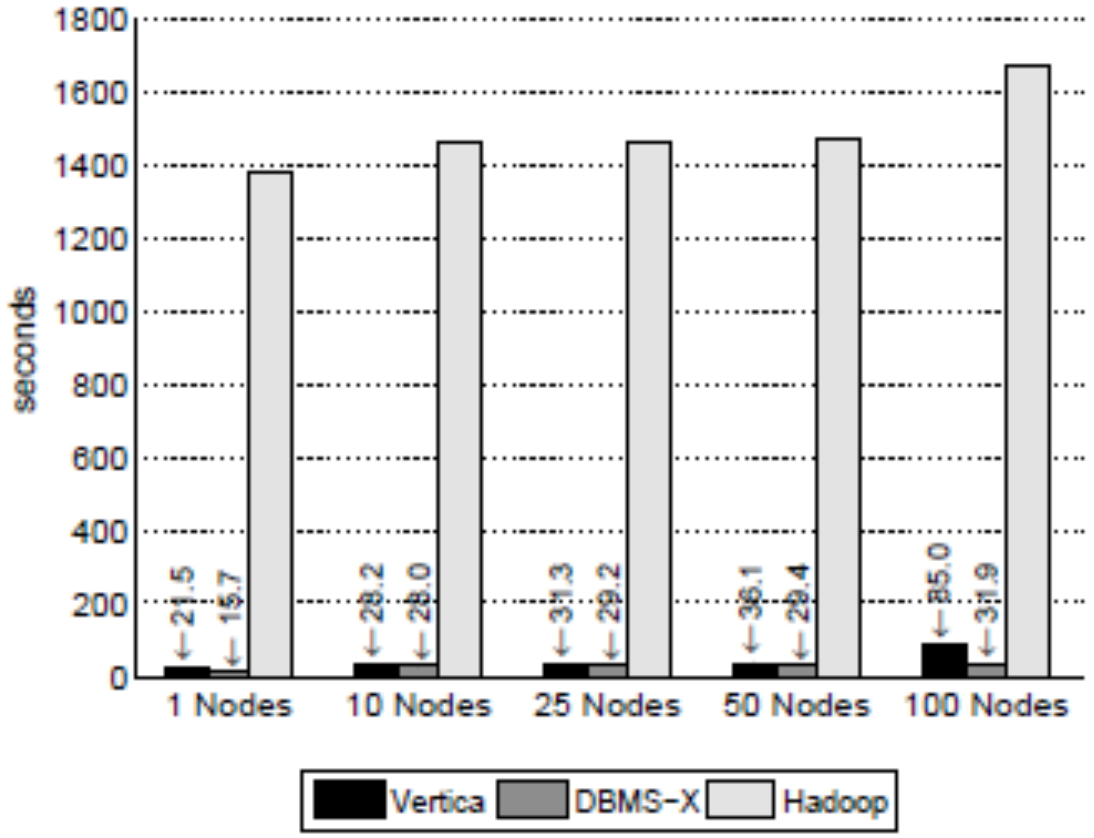


Figure 9: Join Task Results

Discussion

- System level
  - Easy to install/configure MR, more challenging to install parallel DBMSs
  - Available tools for performance tuning for parallel DBMS
  - MR takes more time in task start-up
  - Compression does not help MR
  - MR has short loading time, while DBMSs take some time to reorg the input data
  - MR has better fault tolerance