MapReduce, HBase, Pig and Hive

University of California, Berkeley

School of Information

IS 257: Database Management

History of the World, Part 1

- Relational Databases – mainstay of business
- Web-based applications caused spikes
  - Especially true for public-facing e-Commerce sites
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application (ie. Ehcache)

Scaling Up

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as ‘scaling out’ or ‘horizontal scaling’
- Different approaches include:
  - Master-slave
  - Sharding

Scaling RDBMS – Master/Slave

- Master-Slave
  - All writes are written to the master. All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated down
  - Large data sets can pose problems as master needs to duplicate data to slaves

Scaling RDBMS - Sharding

- Partition or sharding
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

Other ways to scale RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINs, thereby reducing query time
  - This involves de-normalizing data
- In-memory databases (like VoltDB)

NoSQL

- NoSQL databases adopted these approaches to scaling, but lacked ACID transaction and SQL
- At the same time, many Web-based services needed to deal with Big Data (the Three V's we looked at last time) and created custom approaches to do this
- In particular, MapReduce…

MapReduce and Hadoop

- MapReduce developed at Google
- MapReduce implemented in Nutch
  - Doug Cutting at Yahoo!
  - Became Hadoop (named for Doug's child's stuffed elephant toy)

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don't want hassle of ***managing*** things

- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

From "MapReduce…" by Dan Weld

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

From "MapReduce…" by Dan Weld

Map in Lisp (Scheme)

- (map ***f list [list$_2$ list$_3$ …]***)

Unary operator

- (map square '(1 2 3 4))
  - (1 4 9 16)

Binary operator

- (reduce + '(1 4 9 16))

)

  - 30
- (reduce + (map square (map – l_1 l_2))))

From "MapReduce…" by Dan Weld

Map/Reduce ala Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs

- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

From “MapReduce…” by Dan Weld

Programming model

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
- map (in_key, in_value) -> list(out_key, intermediate_value)
  - Processes input key/value pair
  - Produces set of intermediate pairs
- reduce (out_key, list(intermediate_value)) -> list(out_value)
  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

count words in docs

– Input consists of (url, contents) pairs

– map(key=url, val=contents):
  - For each word *w* in contents, emit (w, "1")

– reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

From "MapReduce…" by Dan Weld

Count, Illustrated

map(key=url, val=contents):

For each word *w* in contents, emit (w, "1")

reduce(key=word, values=uniq_counts):

Sum all "1"s in values list

Emit result "(word, sum)"

see bob throw
see spot run

see	1
bob	1
run	1
see	1
spot	1
throw	1

bob	1
run	1
see	2
spot	1
throw	1

From "MapReduce…" by Dan Weld

Example

- Page 1: the weather is good
- Page 2: today is good
- Page 3: good weather is good.

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

Map output

- Worker 1:
  - (the 1), (weather 1), (is 1), (good 1).
- Worker 2:
  - (today 1), (is 1), (good 1).
- Worker 3:
  - (good 1), (weather 1), (is 1), (good 1).

From “MapReduce: Simplified data Processing… ”, Jeffrey Dean and Sanjay Ghemawat

Reduce Input

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 1), (is 1), (is 1)
- Worker 3:
  - (weather 1), (weather 1)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 1), (good 1), (good 1), (good 1)

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

Reduce Output

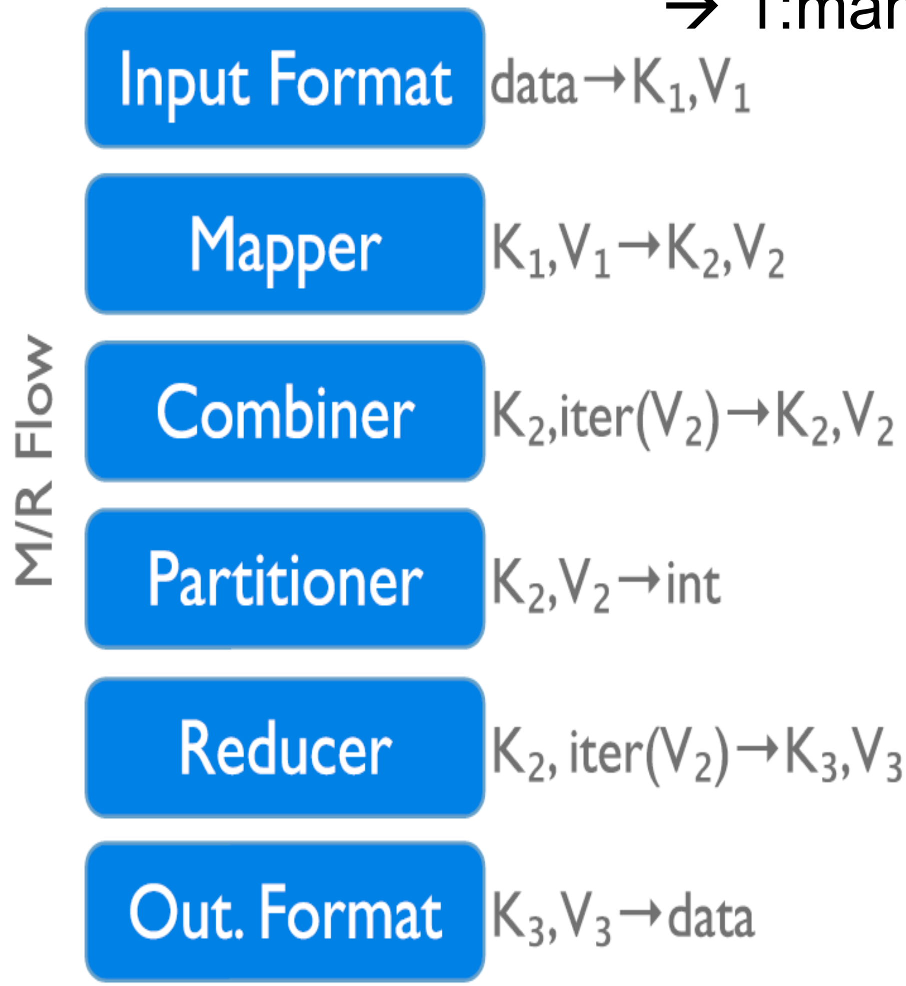
- Worker 1:
  - (the 1)
- Worker 2:
  - (is 3)
- Worker 3:
  - (weather 2)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 4)

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

Data Flow in a MapReduce Program in Hadoop

- InputFormat
- Map function
- Partitioner
- Sorting & Merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat

→ 1:many



Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, "1")
- reduce(key=line, values=uniq_counts):
  - Don't do anything; just emit line

From "MapReduce…" by Dan Weld

Reverse Web-Link Graph

- Map
  - For each URL linking to target, …
  - Output <target, source> pairs
- Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, ***list*** (source)> pairs

From “MapReduce…” by Dan Weld

MapReduce in Hadoop (1)

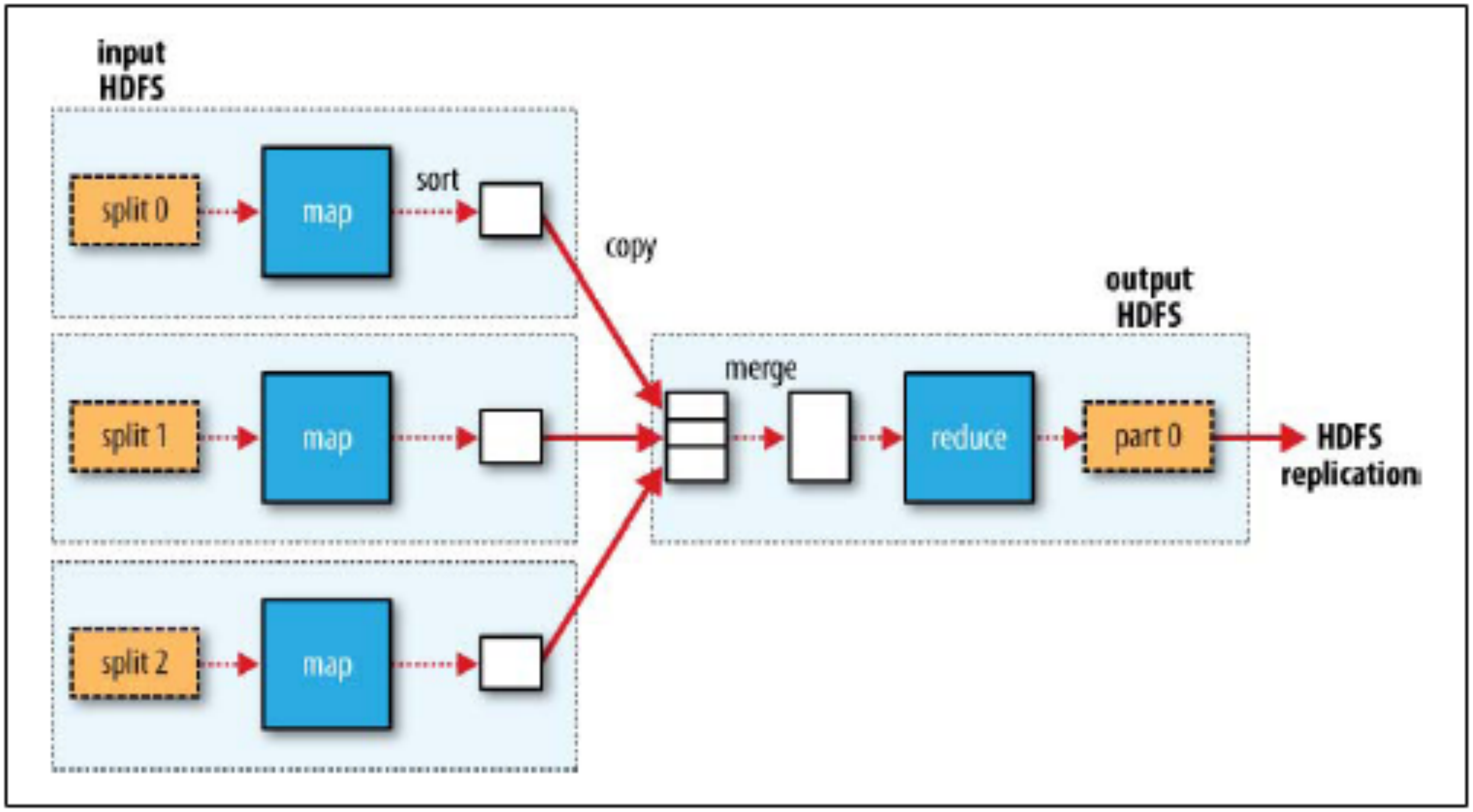


Figure 2-2. MapReduce data flow with a single reduce task

MapReduce in Hadoop (2)

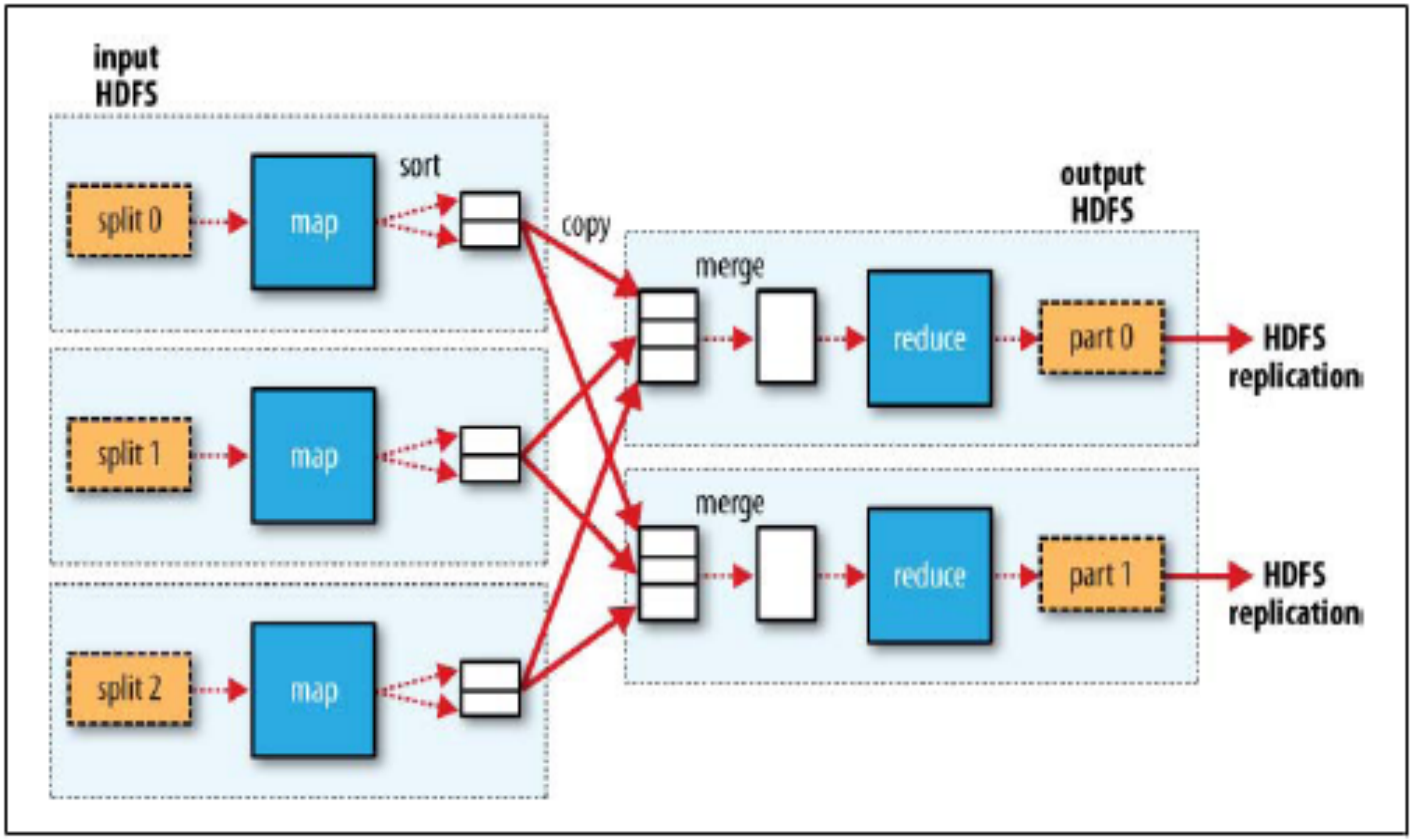


Figure 2-3. MapReduce data flow with multiple reduce tasks

MapReduce in Hadoop (3)

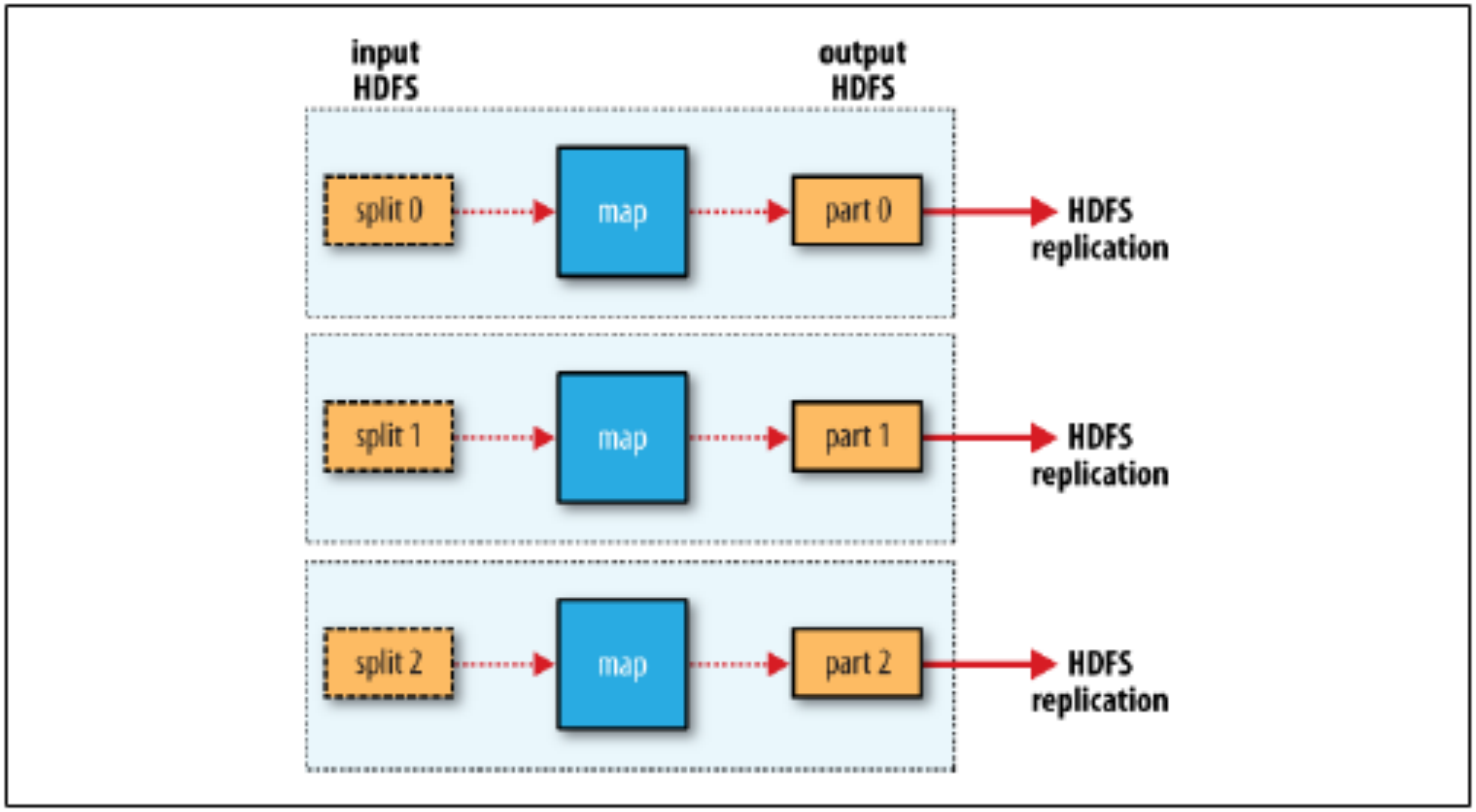


Figure 2-4. MapReduce data flow with no reduce tasks

Fault tolerance

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master
- Master failure:
  - Could handle, but don't yet (master failure unlikely)

From “MapReduce: Simplified data Processing… ”, Jeffrey Dean and Sanjay Ghemawat

Refinement

- Different partitioning functions.
- Combiner function.
- Different input/output types.
- Skipping bad records.
- Local execution.
- Status info.
- Counters.

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

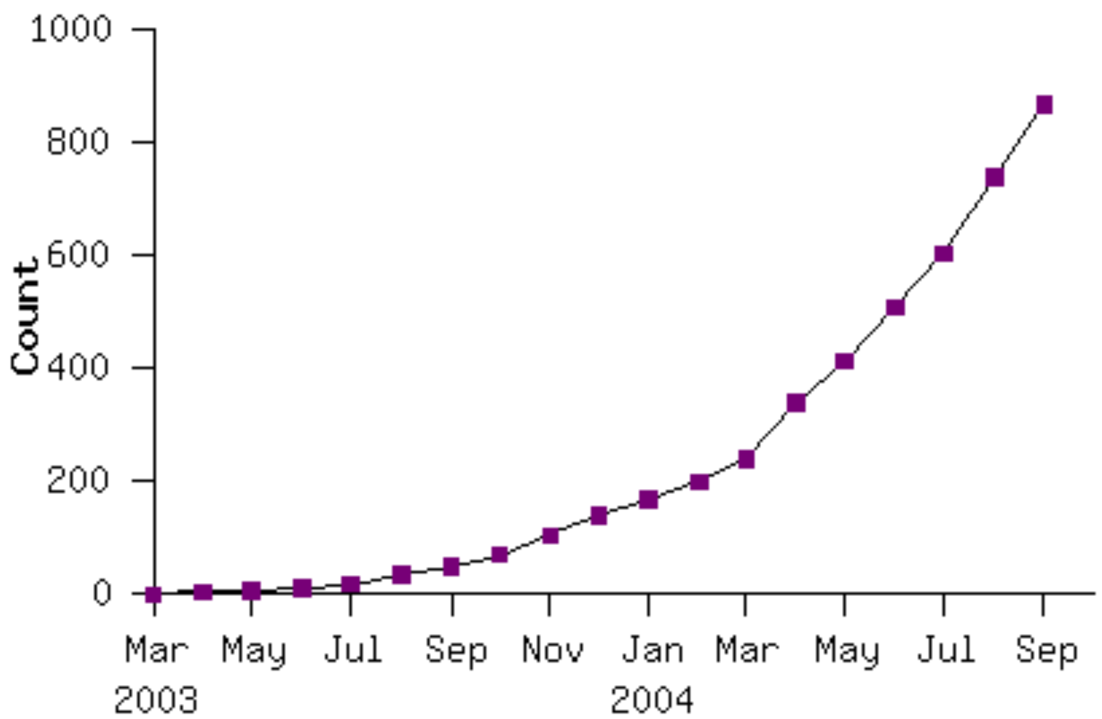
Performance

- Scan 10^10 100-byte records to extract records matching a rare pattern (92K matching records) : 150 seconds.

- Sort 10^10 100-byte records (modeled after TeraSort benchmark) : normal 839 seconds.

From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

More and more mapreduce



From "MapReduce: Simplified data Processing… ", Jeffrey Dean and Sanjay Ghemawat

Conclusion

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Fun to use: focus on problem, let library deal w/ messy details

From “MapReduce: Simplified data Processing… ”, Jeffrey Dean and Sanjay Ghemawat

But – Raw Hadoop means code

- Most people don't want to write code if they don't have to
- Various tools layered on top of Hadoop give different, and more familiar, interfaces
- Hbase – intended to be a NoSQL database abstraction for Hadoop
- Hive and it's SQL-like language

Hadoop Components

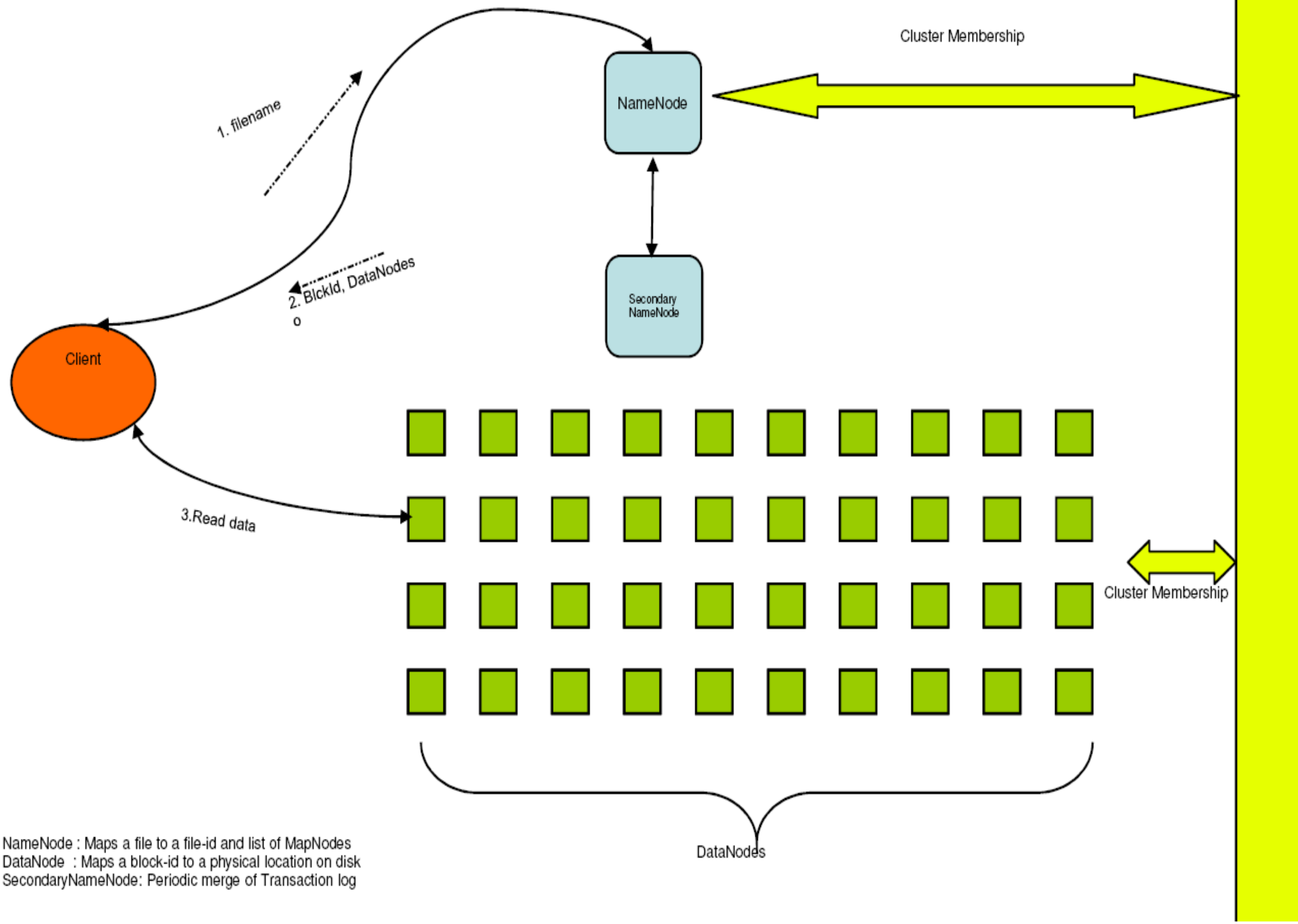
- **Hadoop Distributed File System (HDFS)**
- **Hadoop Map-Reduce**
- **Contributes**
  – Hadoop Streaming
  – Pig / JAQL / Hive
  – HBase
  – Hama / Mahout

Hadoop Distributed File System

Goals of HDFS

- **Very Large Distributed File System**
  - 10K nodes, 100 million files, 10 PB
- **Convenient Cluster Management**
  - Load balancing
  - Node failures
  - Cluster expansion
- **Optimized for Batch Processing**
  - Allow move computation to data
  - Maximize throughput

HDFS Architecture

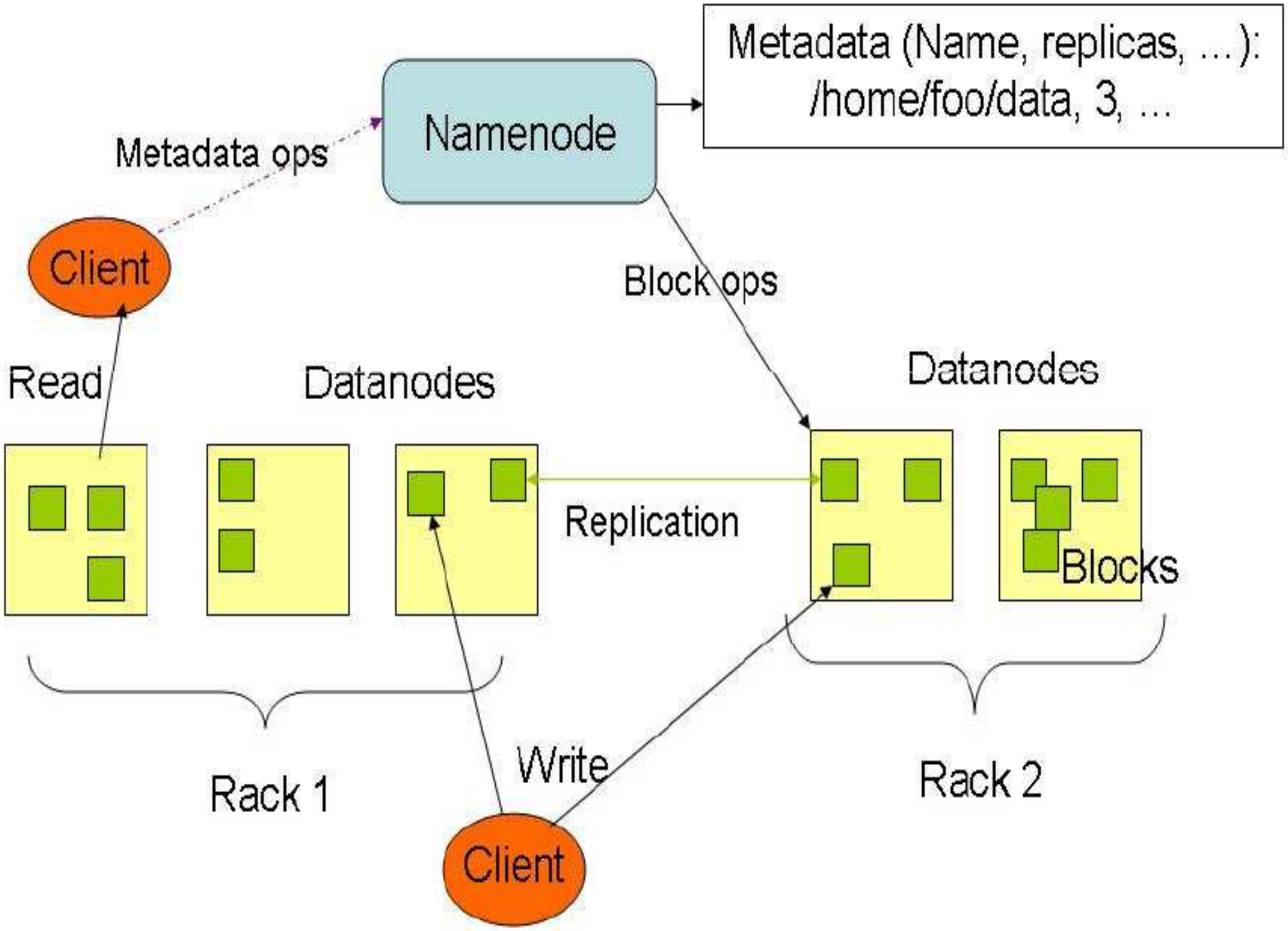


NameNode : Maps a file to a file-id and list of MapNodes
DataNode : Maps a block-id to a physical location on disk
SecondaryNameNode: Periodic merge of Transaction log

HDFS Details

- **Data Coherency**
  - Write-once-read-many access model
  - Client can only append to existing files
- **Files are broken up into blocks**
  - Typically 128 MB block size
  - *Each block replicated on multiple DataNodes*
- **Intelligent Client**
  - Client can find location of blocks
  - Client accesses data directly from DataNode

HDFS Architecture



HDFS User Interface

- **Java API**
- **Command Line**
  - –hadoop dfs -mkdir /foodir
  - –hadoop dfs -cat /foodir/myfile.txt
  - –hadoop dfs -rm /foodir myfile.txt
  - –hadoop dfsadmin -report
  - –hadoop dfsadmin -decommission datanodename
- **Web Interface**
  - –http://host:port/dfshealth.jsp

HDFS

- Very large-scale distributed storage with automatic backup (replication)
- Processing can run at each node also
  - Bring the computation to the data instead of vice-versa
- Underlies all of the other Hadoop “menagie” of programs

PIG – A data-flow language for MapReduce

MapReduce too complex?

- Restrict programming model
  - Only two phases
  - Job chain for long data flow
- Put the logic at the right phase
  - In MR programmers are responsible for this
- Too many lines of code even for simple logic
  - How many lines do you have for word count?

Pig…

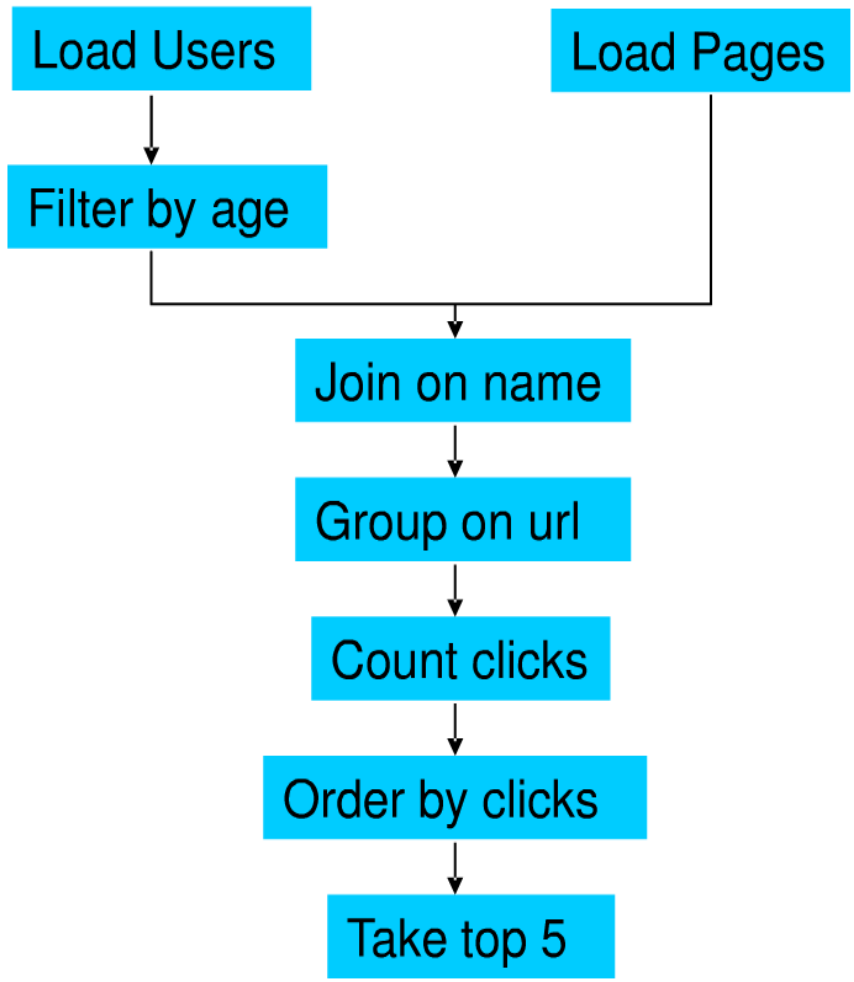
- High level dataflow language (Pig Latin)
  - Much simpler than Java
  - Simplify the data processing
- Put the operations at the apropriate phases (map, shuffle, etc.)
- Chains multiple MapReduce jobs
- Similar to relational algebra, but on files instead of relations

Pig Latin

- Data flow language
  - User specifies a sequence of operations to process data
  - More control on the processing, compared with declarative language
- Various data types are supported
- ”Schema”s are supported
- User-defined functions are supported

Motivation by Example

- Suppose we have user data in one file, website data in another file.
- We need to find the top 5 most visited pages by users aged 18-25



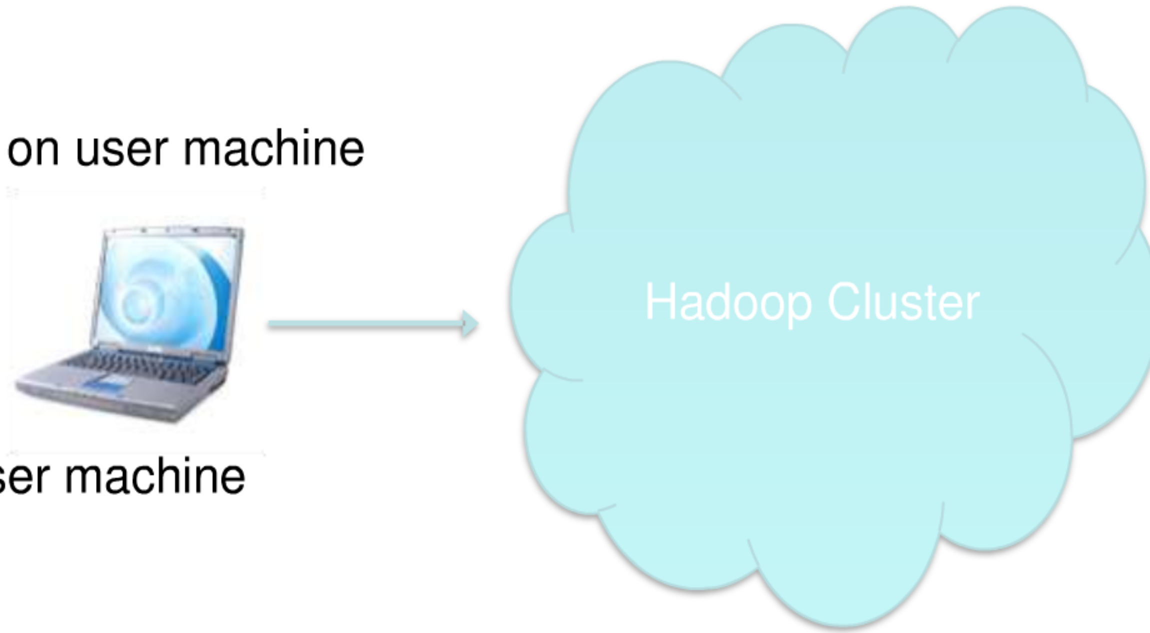
In Pig Latin

```
Users = load‘users’as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load ‘pages’ as (user, url);
Jnd = joinFltrdby name, Pages by user;
Grpd = groupJndbyurl;
Smmd = foreachGrpdgenerate group,
COUNT(Jnd) as clicks;
Srtd = orderSmmdby clicks desc;
Top5 = limitSrtd 5;
store Top5 into‘top5sites’;
```

Pig runs over Hadoop

Job executes on cluster

Pig resides on user machine



User machine

No need to install anything extra on your Hadoop cluster.

How Pig is used in Industry

- At Yahoo!, 70% MapReduce jobs are written in Pig
- Used to
  - Process web log
  - Build user behavior models
  - Process images
  - Data mining
- Also used by Twitter, LinkedIn, Ebay, AOL, etc.

MapReduce vs. Pig

- MaxTemperature

Year	Temperature	Air Quality	…
1998	87	2	…
1983	93	4	..
2008	90	3	…
2001	89	5	…
1965	97	4	…

Table1

```
SELECT Year,
MAX(Temperature)
FROM  Table1
WHERE AirQuality = 0|1|4|5|9
GROUPBY Year
```

In MapReduce

```java
// cc MaxTemperatureMapper Mapper for maximum temperature example
// vv MaxTemperatureMapper
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class MaxTemperatureMapper extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

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

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't
      airTemperature = Integer.parseInt(line.substrin
    } else {
      airTemperature = Integer.parseInt(line.substrin
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches(
      output.collect(new Text(year), new IntWritable(
    }
  }
}
// ^^ MaxTemperatureMapper
```

```java
// cc MaxTemperatureReducer Reducer for maximum temperature example
// cc MaxTemperatureReducer2 Reducer for maximum temperature example
// vv MaxTemperatureReducer
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

// vv MaxTemperatureReducer2
public class MaxTemperatureReducer extends Ma
  implements Reducer<Text, IntWritable, Text,

  public void reduce(Text key, Iterator<IntWr
      OutputCollector<Text, IntWritable> outp
      throws IOException {

    int maxValue = Integer.MIN_VALUE;
    while (values.hasNext()) {
      maxValue = Math.max(maxValue, values.ne
    }
    output.collect(key, new IntWritable(maxVa
  }
}
// ^^ MaxTemperatureReducer2
// ^^ MaxTemperatureReducer
```

```java
// cc MaxTemperature Application to find the maximum temperature in the weather dataset
// vv MaxTemperature
import java.io.IOException;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;

public class MaxTemperature {

  public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    JobConf conf = new JobConf(MaxTemperature.class);
    conf.setJobName("Max temperature");

    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setMapperClass(MaxTemperatureMapper.class);
    conf.setReducerClass(MaxTemperatureReducer.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    JobClient.runJob(conf);
  }
}
// ^^ MaxTemperature
```

In Pig

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

Wait a mimute

- How to map the data to records
  - By default, one line → one record
  - User can customize the loading process
- How to identify attributes and map them to schema?
  - Delimiters to separate different attributes
  - By default tabs are used, but it can be customized

MapReduce vs. Pig cont.

- Join in MapReduce
  - Various algorithms. None of them are easy to implement in MapReduce
  - Multi-way join more complicated

MapReduce vs. Pig cont.

- Join in Pig
  - Various algorithms already available.
  - Some of them are generic to support multi-way join
  - Simple to integrate into workflow…

```
A = LOAD 'input/join/A’;
B = LOAD 'input/join/B’;
C = JOIN A BY $0, B BY $1;
DUMP C;
```

Hive - SQL on top of Hadoop

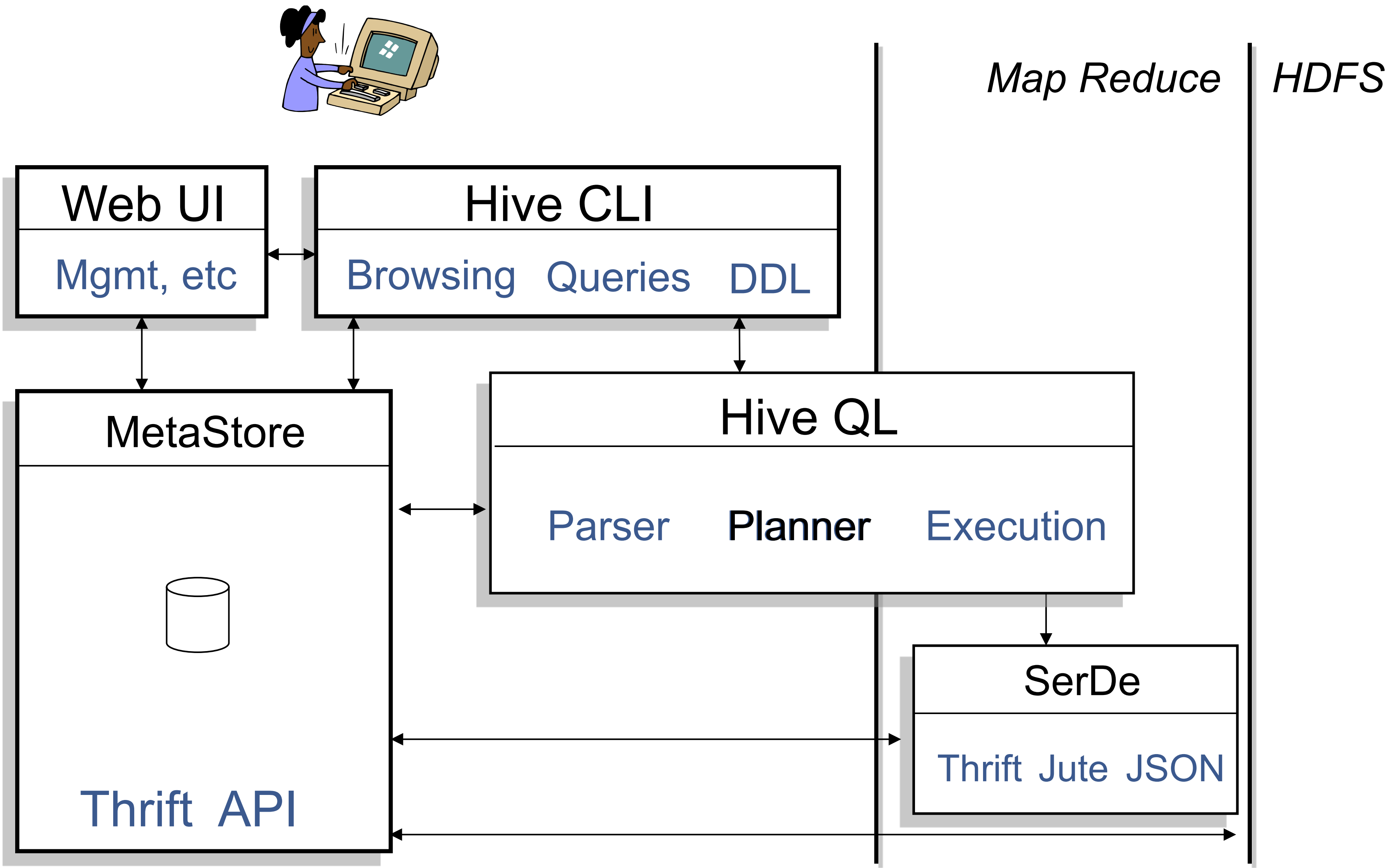
Map-Reduce and SQL

- **Map-Reduce is scalable**
  - SQL has a huge user base
  - SQL is easy to code
- **Solution: Combine SQL and Map-Reduce**
  - Hive on top of Hadoop (open source)
  - Aster Data (proprietary)
  - Green Plum (proprietary)

Hive

- **A database/data warehouse on top of Hadoop**
  – Rich data types (structs, lists and maps)
  – Efficient implementations of SQL filters, joins and group-by's on top of mapreduce
- **Allow users to access Hadoop data without using Hadoop**
- **Link:**
  – **http://svn.apache.org/repos/asf/hadoop/hive/trunk/**

Hive Architecture



Hive QL – Join

- SQL:

```
INSERT INTO TABLE pv_users
SELECT pv.pageid, u.age
FROM page_view pv JOIN user u ON (pv.userid = u.userid);
```

page_view

pageid	**userid**	time
1	**111**	9:08:01
2	**111**	9:08:13
1	**222**	9:08:14

X

user

userid	age	gender
111	25	female
222	32	male

=

pv_users

pageid	age
1	25
2	25
1	32

Hive QL – Join in Map Reduce

page_view

pageid	**userid**	time
1	**111**	9:08:01
2	**111**	9:08:13
1	**222**	9:08:14

user

user id	age	gender
111	25	female
222	32	male

Map

key	value
111	<**1**,1>
111	<**1**,2>
222	<**1**,1>

key	value
111	<**2**,25>
222	<**2**,32>

Shuffle Sort

key	value
111	<**1**,1>
111	<**1**,2>
111	<**2**,25>

key	value
222	<**1**,1>
222	<**2**,32>

Reduce

pv_users

page id	age
1	25
2	25

page id	age
1	32

Hive QL – Group By

- SQL:
  - INSERT INTO TABLE pageid_age_sum
    SELECT pageid, age, count(1)
    FROM pv_users
    GROUP BY pageid, age;

pv_users

pageid	age
1	25
2	25
1	32
2	25

pageid_age_sum

pageid	age	Count
1	25	1
2	25	2
1	32	1

Hive QL – Group By in Map Reduce

pv_users

pageid	age
1	25
2	25

key	value
<1,25>	1
<2,25>	1

Map

pageid	age
1	32
2	25

key	value
<1,32>	1
<2,25>	1

Shuffle Sort

key	value
<1,25>	1
<1,32>	1

key	value
<2,25>	1
<2,25>	1

Reduce

pageid_age_sum

pageid	age	Count
1	25	1
1	32	1

pageid	age	Count
2	25	2

Beyond Hadoop – Spark

Spark

- One problem with Hadoop/MapReduce is that it is fundamental batch oriented, and everything goes through a read/write on HDFS for every step in a dataflow
- Spark was developed to leverage the main memory of distributed clusters and to, whenever possible, use only memory-to-memory data movement (with other optimizations
- Can give up to 100fold speedup over MR

Spark

- Developed at the AMP lab here at Berkeley
- Open source version available from Apache
- DataBrick was founded to commercialize Spark
- Related software includes a very-high-speed Database – SparkDB
- Next time we will hear a talk (recorded) from Michael Franklin about BDAS & Spark