Data-intensive Computing: Massive Data Processing

DIC Systems

- Google MapReduce
  - Yahoo Hadoop/PIG
  - Data parallel computing
- IBM Research System S
  - InfosphereStream product
  - Continuous data stream processing
- Microsoft Dryad/Dryad LINQ
  - DAG processing
  - Some SQL query support

The Building Blocks of Google Infrastructure

- Distributed file systems: GFS
- Distributed storage: BigTable
- Job scheduler: the workqueue
- Parallel computation: MapReduce
- Distributed lock server: chubby

MapReduce

- A parallel programming model and an associated implementation for processing and generating large data sets.
- A user specified **map** function processes a key/value pair to generate a set of intermediate key/value pairs.
- A user specified **reduce** function merges all intermediate values associated with the same intermediate key.
- Programs written in this functional style are automatically distributed and executed on a large cluster of commodity machines.

Motivation

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

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

Map/Reduce

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

Map in Lisp (Scheme)

Unary operator

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

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

Binary operator

- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1
  - 30
- (reduce + (map square (map – l$_1$ l$_2$))))

Map/Reduce at 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

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)”

Count, Illustrated

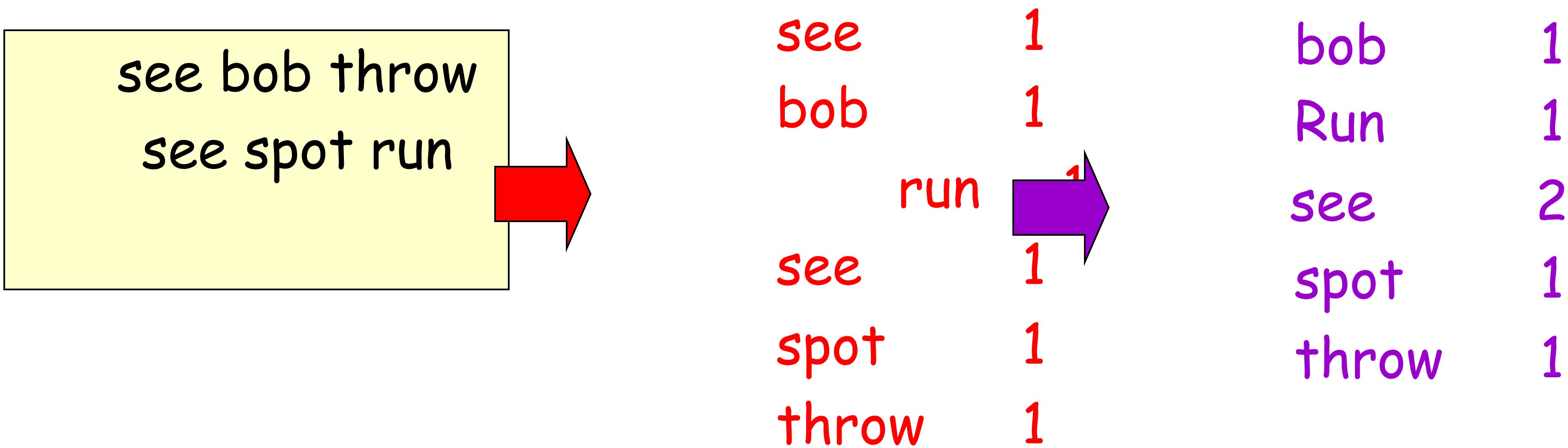
map(key=url, val=contents):

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)”



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

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

Example: Count word occurrences

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator
        intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Hadoop Code - Map

```
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)

                                            // key is empty, value is the line

    throws IOException {

    String line = value.toString();

    StringTokenizer itr = new StringTokenizer(line);

    while (itr.hasMoreTokens()) {

     word.set(itr.nextToken());

     output.collect(word, one);

    }

  }

}
```

Hadoop Code - Reduce

```
public static class ReduceClass extends MapReduceBase  implements
Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(
           Text key,
           Iterator<IntWritable> values,
           OutputCollector<Text, IntWritable> output,
           Reporter reporter)
     throws IOException {
                      // key is word, values is a list of 1's
           int sum = 0;
           while (values.hasNext()) {
             sum += values.next().get();
           }
           output.collect(key, new IntWritable(sum));
  }
}
```

15

Hadoop Code - Driver

```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
           throws Exception {
  // The job. WordCount contains MapClass and Reduce.
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName(”mywordcount");
  // The keys are words
  (strings) conf.setOutputKeyClass(Text.class);
  // The values are counts (ints)
  conf.setOutputValueClass(IntWritable.class);
  conf.setMapperClass(MapClass.class);
  conf.setReducerClass(ReduceClass.class);
  FileInputFormat.addInputPath(
           conf, newPath(inputPath));
  FileOutputFormat.setOutputPath(
           conf, new Path(outputPath));
  JobClient.runJob(conf);
}
```


16

Programming MapReduce

Externally: For user

1. Write a Map program (short), write a Reduce program (short)
2. Specify number of Maps and Reduces (parallelism level)
3. Submit job; wait for result
4. Need to know very little about parallel/distributed programming!

Internally: For the Paradigm and Scheduler

1. Parallelize Map
2. Transfer data from Map to Reduce (**shuffle data**)
3. Parallelize Reduce
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output

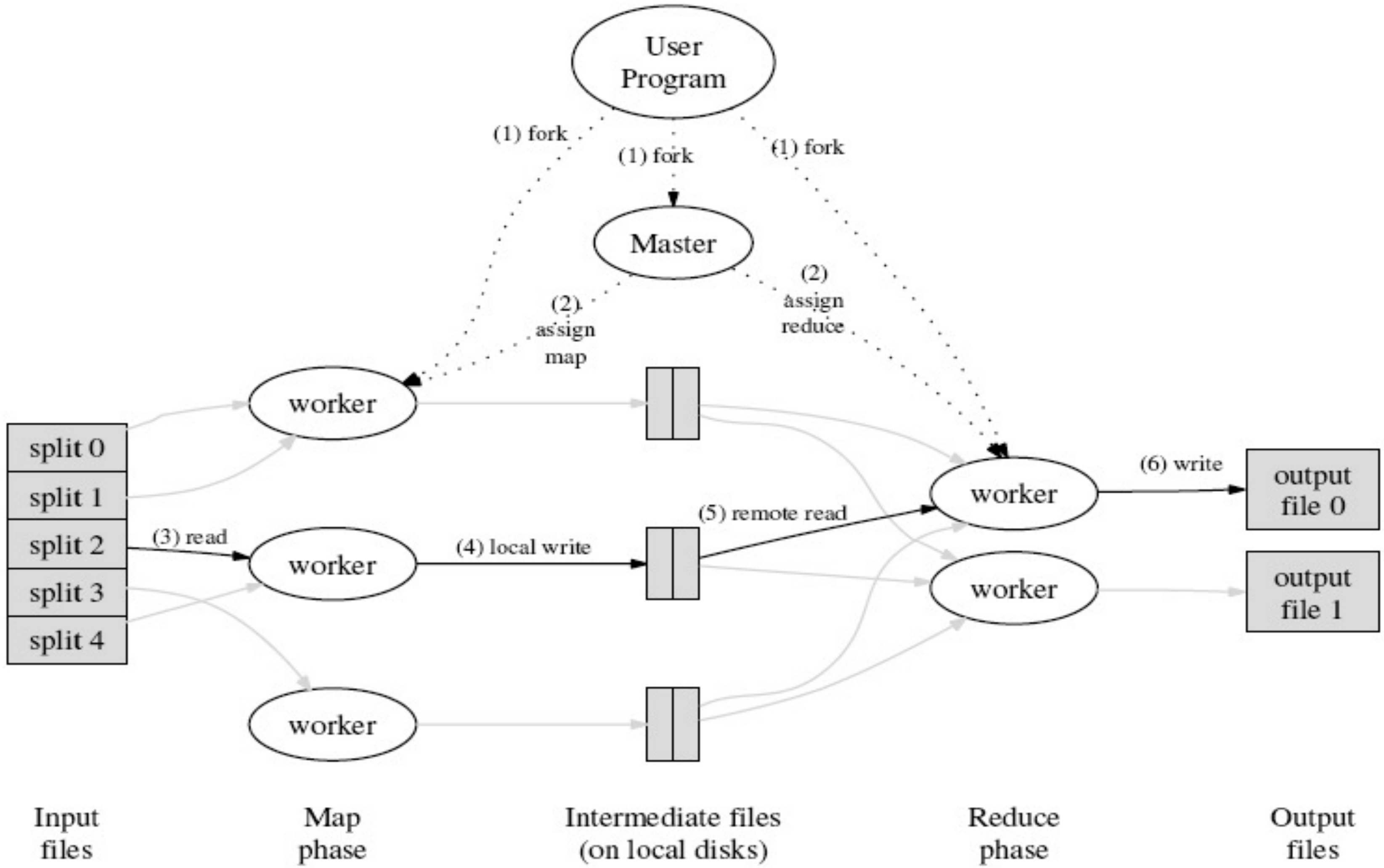
17

(Ensure that no Reduce starts before all Maps are finished. That is, ensure the ***barrier*** between the Map phase and Reduce phase)

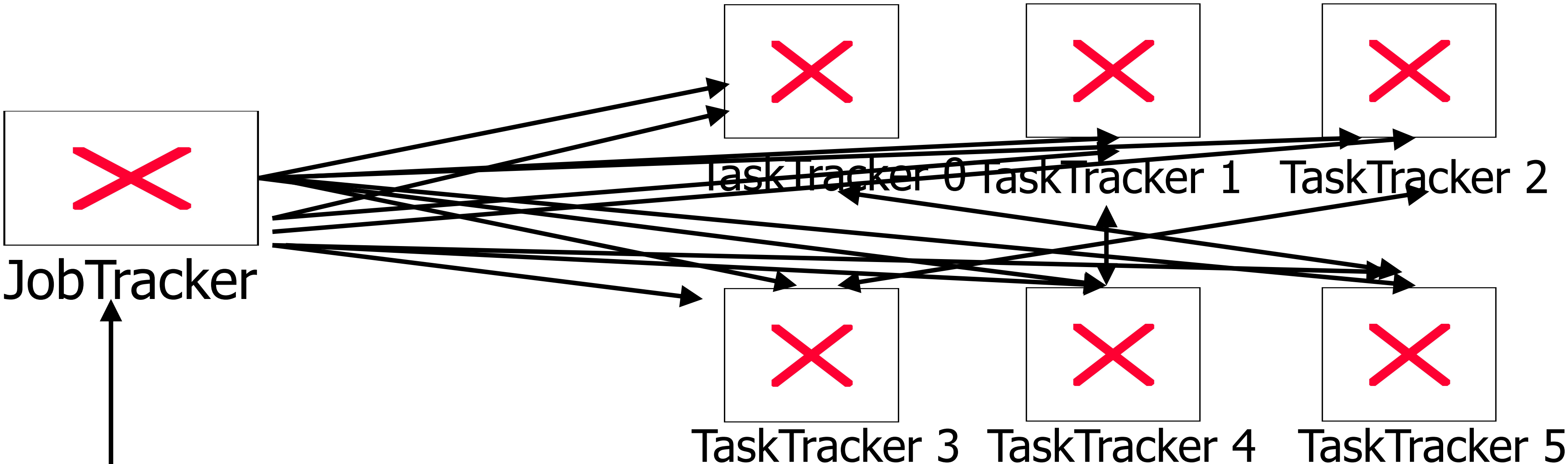
MapReduce Runtime System

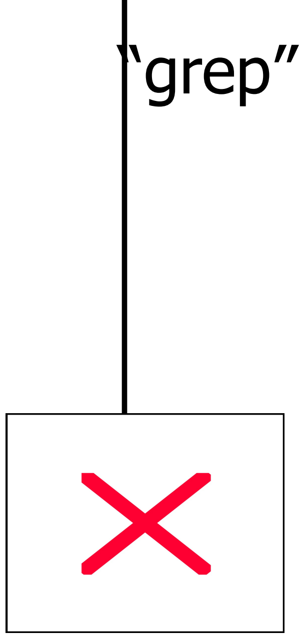
- How is this distributed?
  - Partition input key/value pairs into chunks, run map() tasks in parallel
  - After all map()s are complete, consolidate all emitted values for each unique emitted key
  - Partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

Distributed Execution



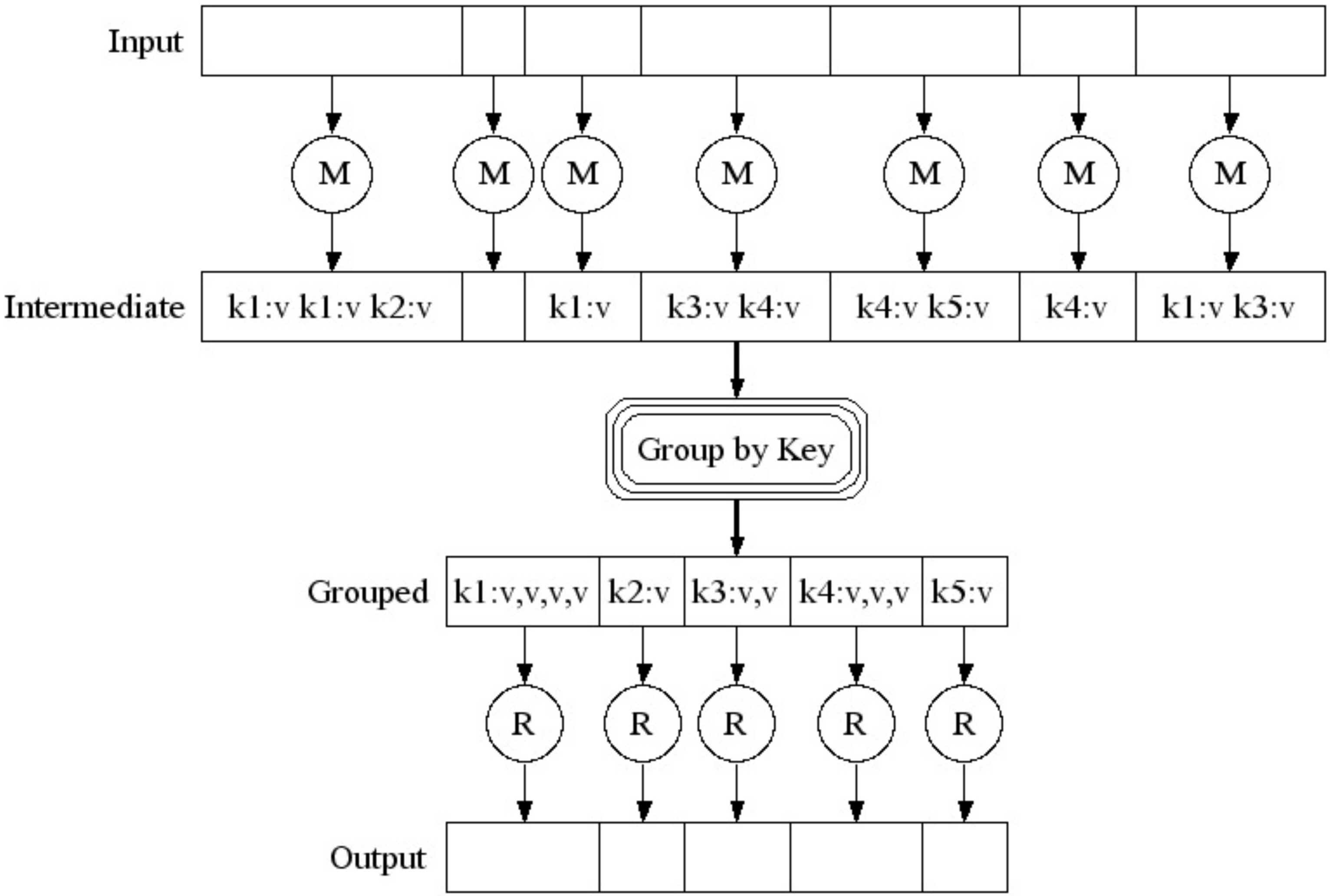
Job Processing



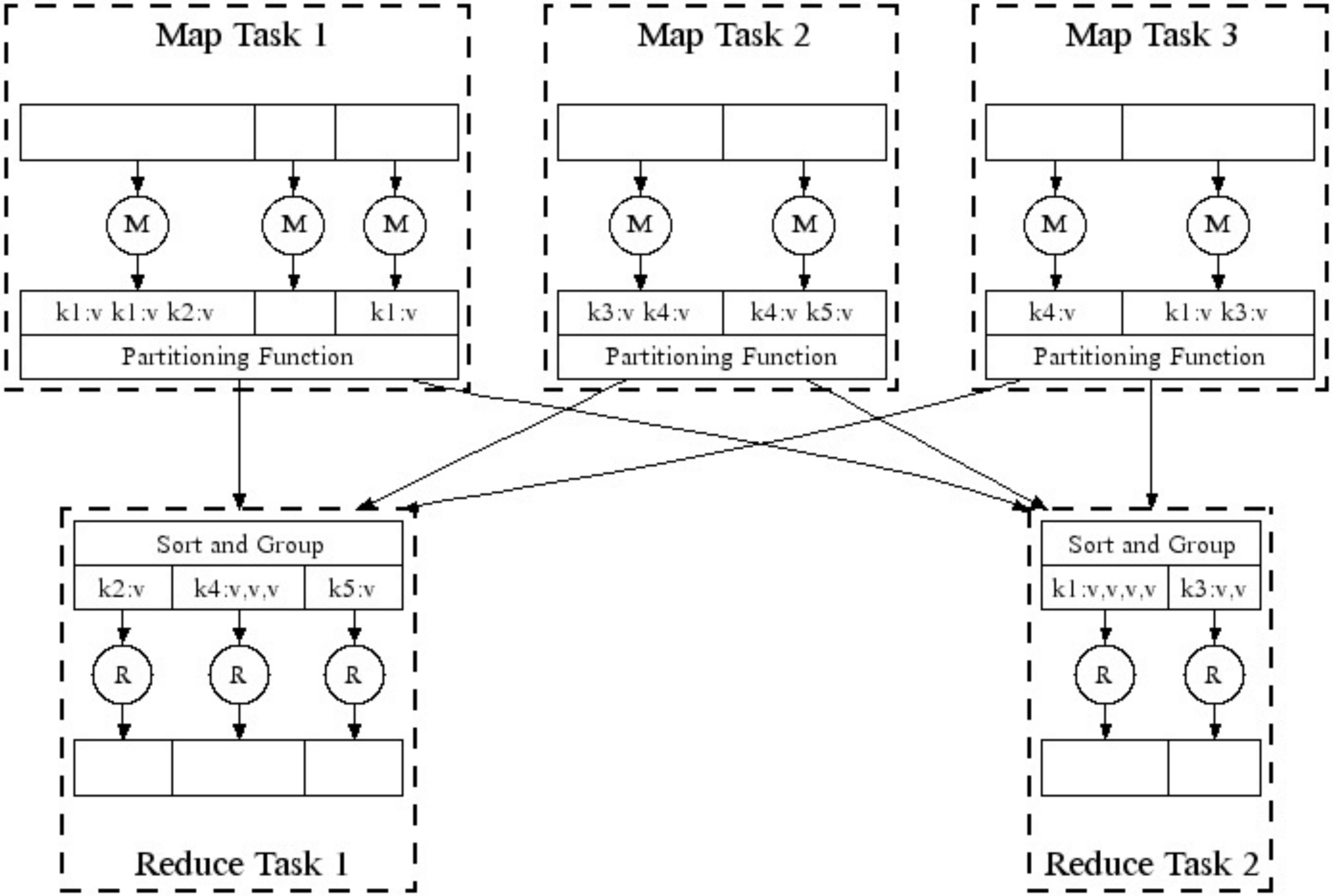


1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to trackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to another MapReduce call

Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Fault Tolerance

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks (why?)
- Re-execute in progress *reduce* tasks (why?)
- Task completion committed through master

Robust: lost 1600/1800 machines once → finished ok

Speculative Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

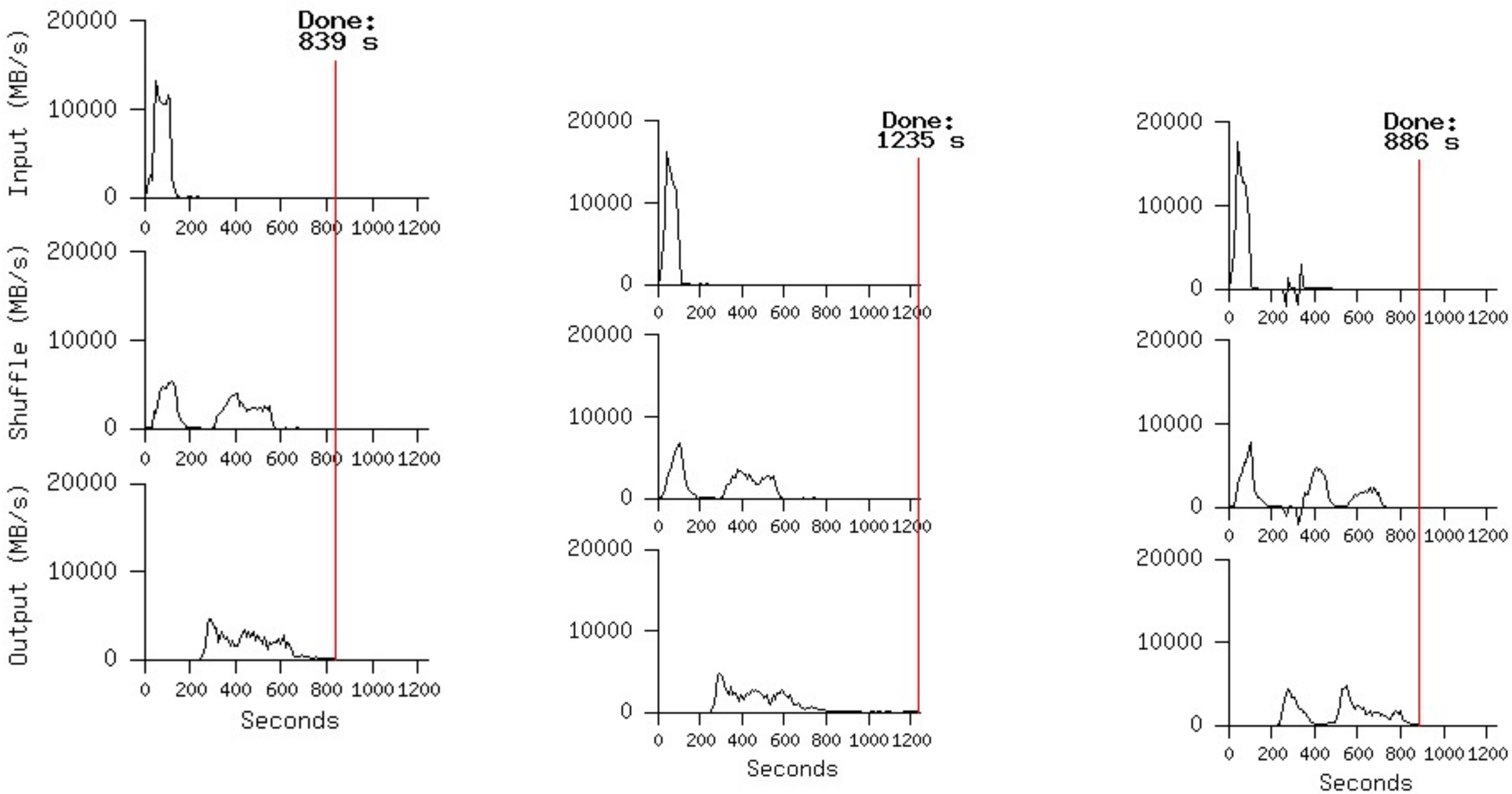
Solution: Speculative execution

- Near end of phase, spawn backup tasks
- Whichever one finishes first "wins"

Dramatically shortens job completion time

MR_Sort

Normal | **No backup tasks** | **200 processes killed**



- Backup tasks reduce job completion time a lot!
  - System deals well with failures

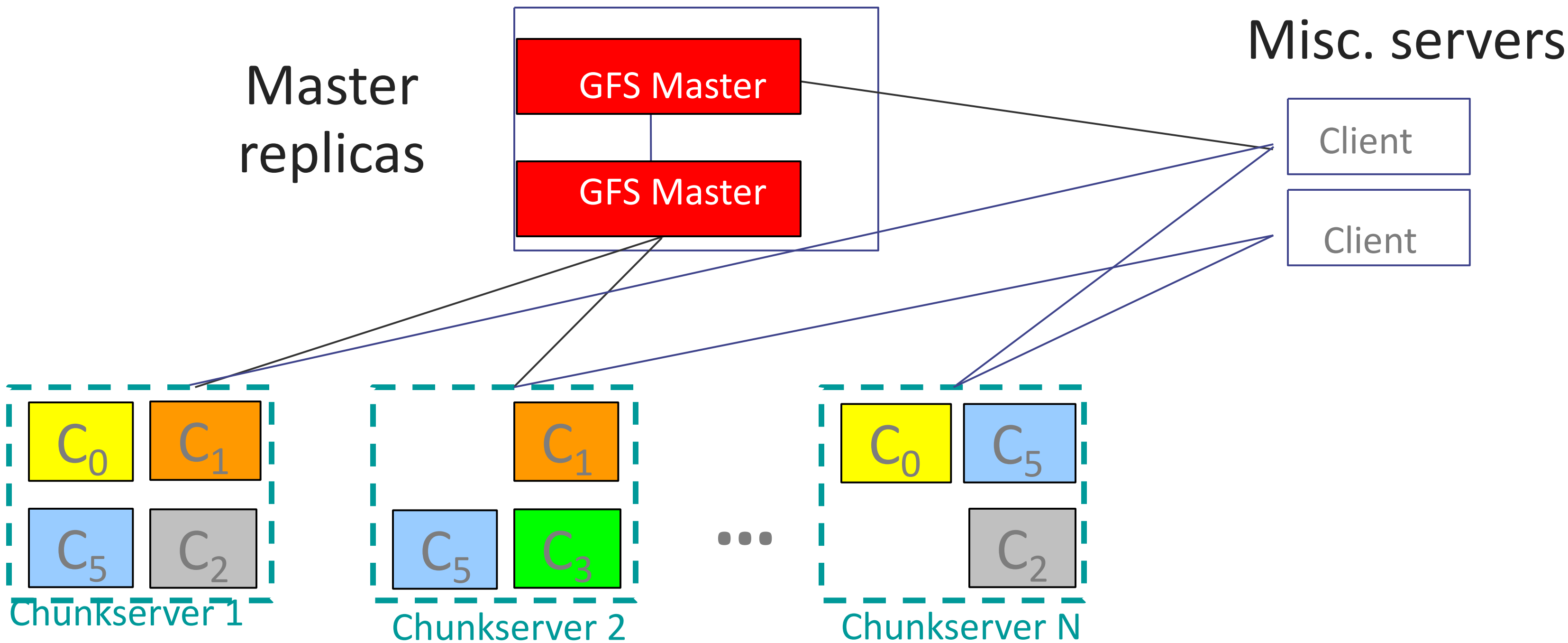
MapReduce Summary

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

GFS: The Google File System

- *Reliable* distributed storage system for *petabyte scale* filesystems.
- Data kept in 64-megabyte “chunks” stored on disks spread across thousands of machines
- Each chunk replicated, usually 3 times, on different machines so that GFS can recover seamlessly from disk or machine failure.
- A GFS cluster consists of a single *master server*, multiple *chunkservers*, and is accessed by multiple *clients*.

GFS: The Google File System



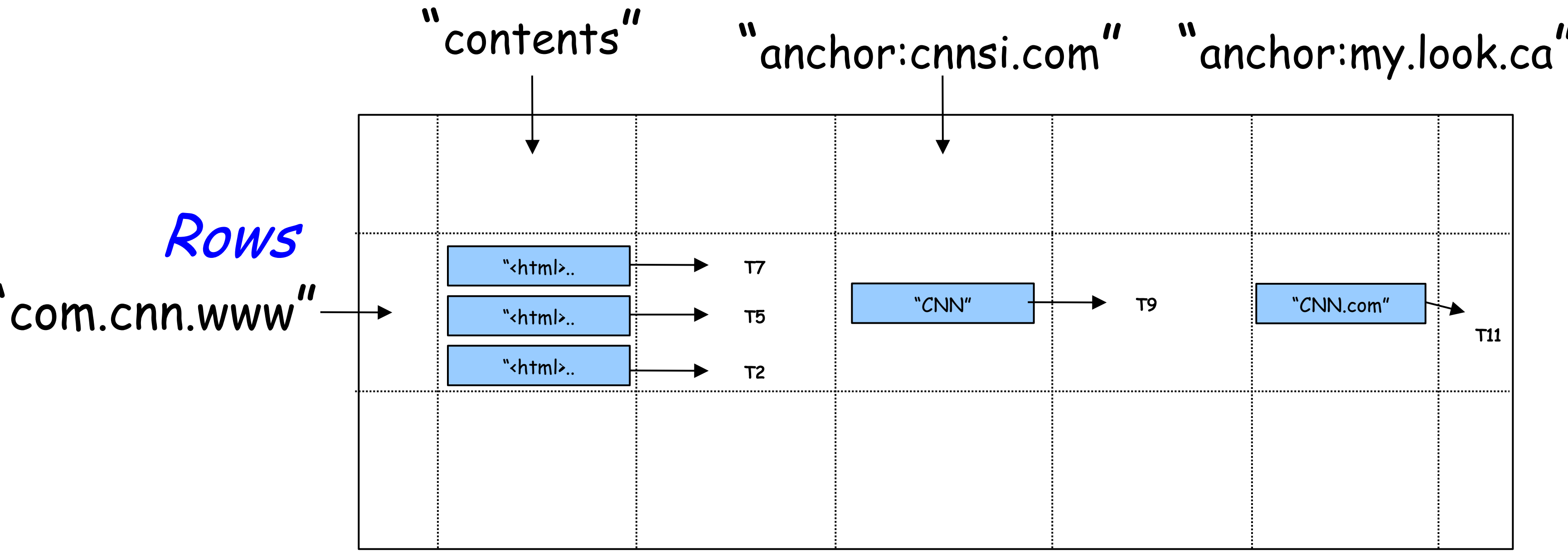
- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety

BigTable

- A distributed storage system for managing structured data
  - Designed to scale to a very large size: petabytes of data across thousands of commodity servers.
- Built on top of GFS
- Used by more than 60 Google products and projects
  - Google Earth, Google Finance, Orkut, …

Basic Data Model

- Triple (row, column, timestamp) -> keys for lookup, insert, and delete API
- Arbitrary “columns” on a row-by-row basis
  - Column “family:qualifier”: Family is heavyweight, qualifier lightweight
  - Column-oriented physical store: rows are sparse!



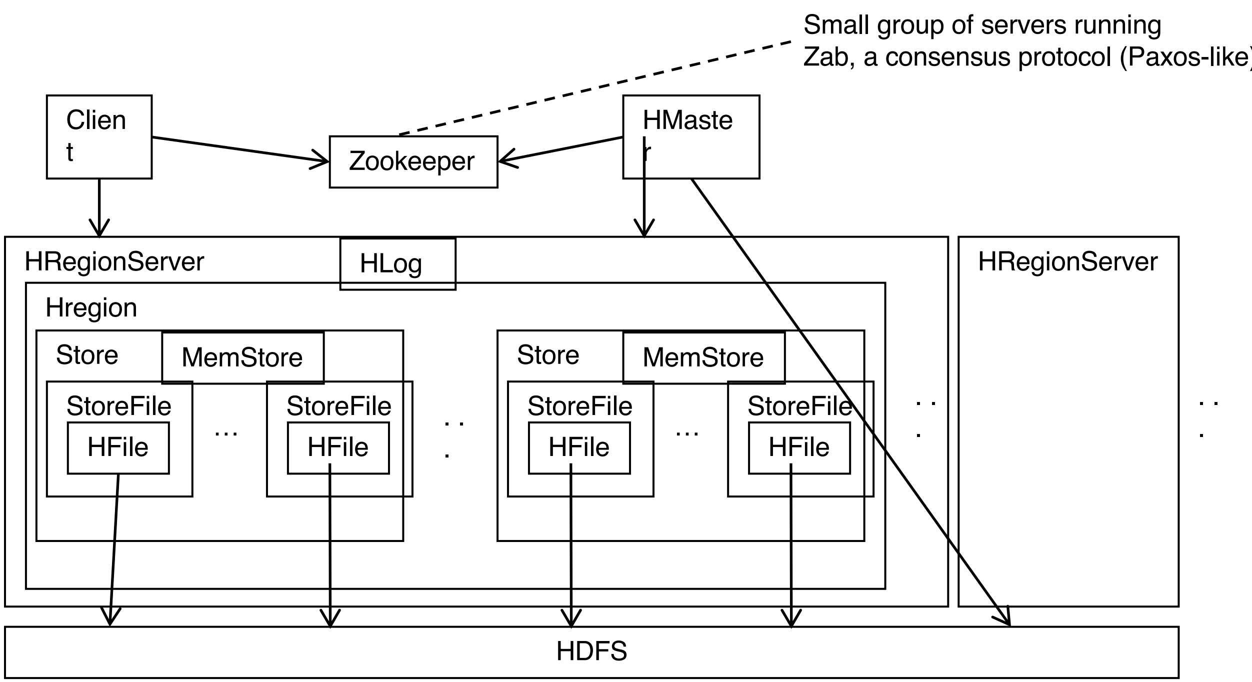
Rows

- Name is an arbitrary string.
  - Access to data in a row is atomic.
  - Row creation is implicit upon storing data.
  - Transactions within a row
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines.
- Does not support relational model
  - No table wide integrity constants
  - No multirow transactions

HBase

- Google's BigTable was first "blob-based" storage system
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- Facebook uses HBase internally
- API functions
  - Get/Put(row)
  - Scan(row range, filter) – range queries
  - MultiPut

HBase Architecture



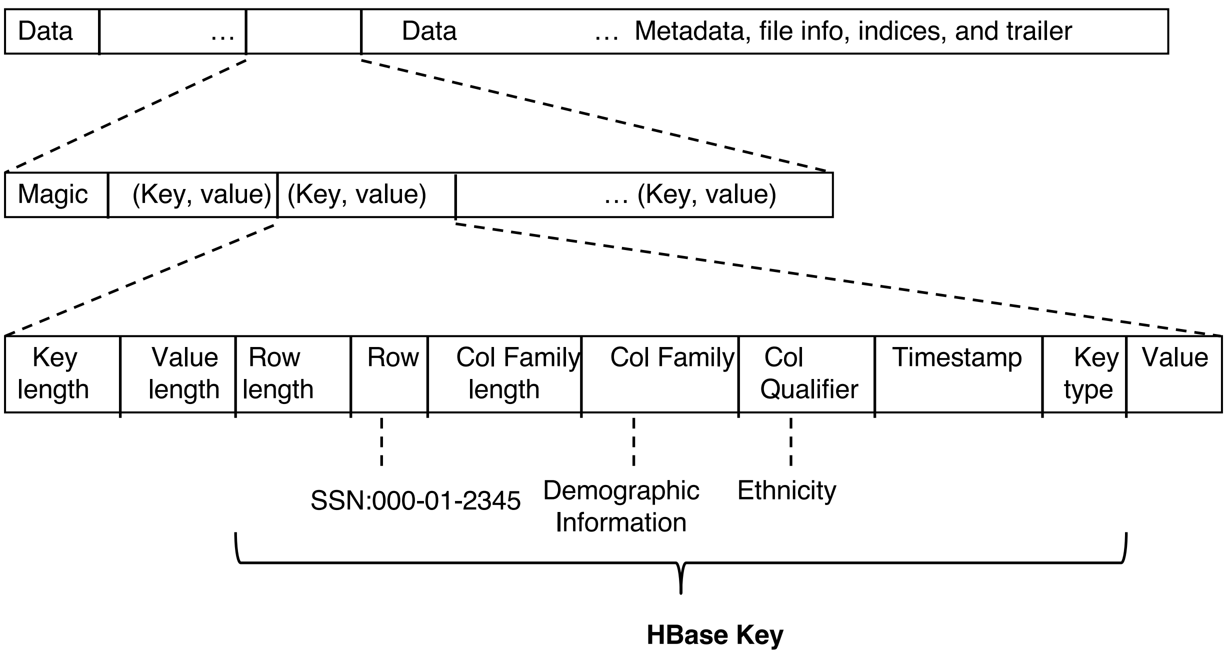
HBase Storage hierarchy

- HBase Table
  - Split it into multiple regions: replicated across servers
    - ColumnFamily = subset of columns with similar query patterns
    - One Store per combination of ColumnFamily + region
      - Memstore for each Store: in-memory updates to Store; flushed to disk when full
        - StoreFiles for each store for each region: where the data lives
          - HFile

- HFile
  - SSTable from Google's BigTable

34

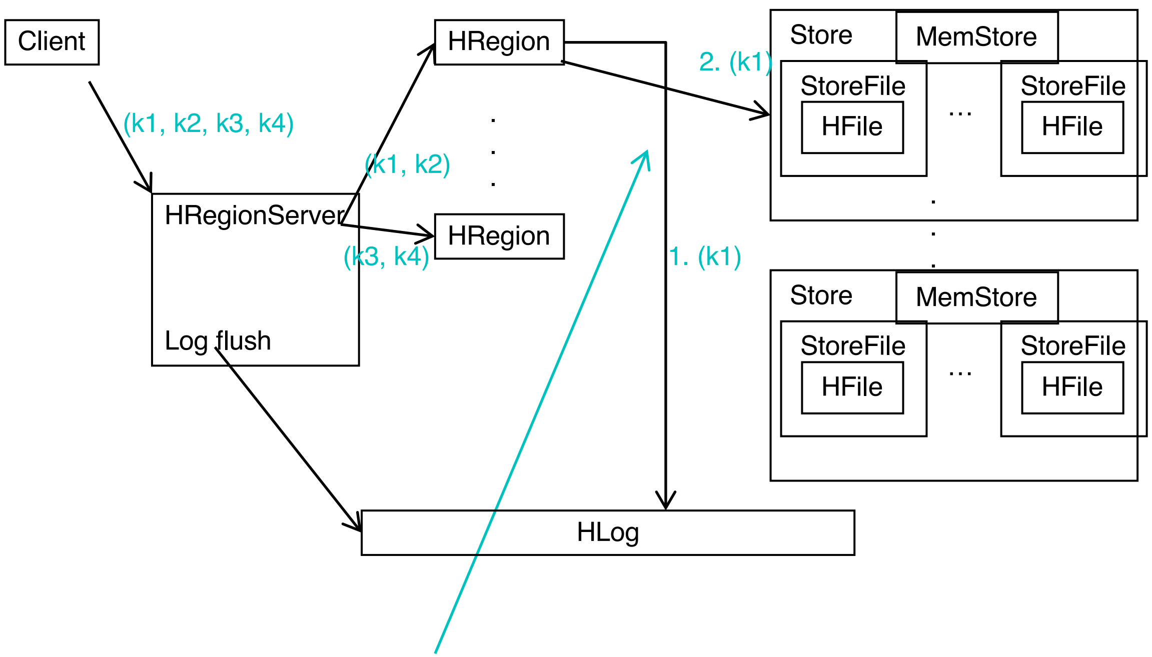
HFile



Data
…
Data
… Metadata, file info, indices, and trailer
Magic
(Key, value)
(Key, value)
… (Key, value)
Key length
Value length
Row length
Row
Col Family length
Col Family
Col Qualifier
Timestamp
Key type
Value
SSN:000-01-2345
Demographic Information
Ethnicity
HBase Key

3
5

Strong Consistency: HBase Write-Ahead Log



Write to HLog before writing to MemStore
Helps recover from failure by replaying Hlog.

3
6

Log Replay

- After recovery from failure, or upon bootup (HRegionServer/HMaster)
  - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
  - Replay: add edits to the MemStore

37

Cross-Datacenter Replication

- Single “Master” cluster
- Other “Slave” clusters replicate the same tables
- Master cluster synchronously sends HLogs over to slave clusters
- Coordination among clusters is via Zookeeper
- Zookeeper can be used like a file system to store control information

1. */hbase/replication/state*
2. */hbase/replication/peers/<peer cluster number>*
3. */hbase/replication/rs/<hlog>*

38