Comments

- Programming model might appear very limited
- But, map and reduce can do anything with their input
  - Could implement a Turing machine inside…
  - …which could compute anything, but…
  - …would not result in a good parallel implementation.
- Challenge: find best MapReduce implementation for a given problem

Basic MapReduce Program Design

- Tasks that can be performed independently on a data object, large number of them: Map
- Tasks that require combining of multiple data objects: Reduce
- Sometimes it is easier to start program design with Map, sometimes with Reduce
- Select keys and values such that the right objects end up together in the same Reduce invocation
- Might have to partition a complex task into multiple MapReduce sub-tasks

Choosing M and R

- M = number of map tasks, R = number of reduce tasks
- Larger M, R: creates smaller tasks, enabling easier load balancing and faster recovery (many small tasks from failed machine)
- Limitation: O(M+R) scheduling decisions and O(M·R) in-memory state at master
  - Very small tasks not worth the startup cost
- Recommendation:
  - Choose M so that split size is approximately 64 MB
  - Choose R a small multiple of the number of workers; alternatively choose R a little smaller than #workers to finish reduce phase in one "wave"

Grep

- Find all lines matching some pattern
- No need to combine anything
  - Reduce is not needed, i.e., just identity function
- Map takes line and outputs it if it matches the pattern
- Map could also take an entire document and emit all matching lines
  - Not a good idea if there is a single large document, but works well if there are many documents

Reverse Web-Link Graph

- For each URL, find all pages (URLs) pointing to it (incoming links)
- Problem: Web page has only outgoing links
- Need all (anySource, P) links for each page P
  - Suggests Reduce with P as the key, source as value
- Map: for page *source*, create all (*target*, *source*) pairs for each link to a *target* found in page
- Reduce: since *target* is key, will receive all sources pointing to that target

Inverted Index

- For each word, create list of documents (document IDs) containing it
- Same as reverse Web-link graph problem
  - "Source URL" is now "document ID"
  - "Target URL" is now "word"
- Can augment this to create list of (document ID, position) pairs for each word
  - Map emits (word, (document ID, position)) while parsing a document

Distributed Sorting

- Can Map do pre-sorting and Reduce the merging?
  - Use *set* of input records as Map input
  - Map pre-sorts it and single reducer merges them
  - Does not scale!
- We need to get multiple reducers involved
  - What should we use as the intermediate key?

28

Distributed Sorting, Revisited

- Quicksort-style partitioning
- For simplicity, consider case with 2 machines
  - Goal: each machine sorts about half of the data
- Assuming we can find the median record, assign all smaller records to machine 1, all others to machine 2
- Sort locally on each machine, then “concatenate” output

29

Partitioning Sort in MapReduce

- Consider 2 reducers for simplicity
- Run MapReduce job to find approximate median of data
  - Hadoop also offers InputSampler
    - Writes the keys that define the partitions, to be used by TotalOrderPartitioner
    - Runs on client and downloads input data splits, hence only useful if data is sampled from few splits, i.e., splits themselves should contain random data samples
- Map outputs (sortKey, record) for an input record
- All sortKey < median are assigned to reduce task 1, all others to reduce task 2, using a partitioner
- Reduce sorts its assigned set of records

30

Partitioning Sort in MapReduce

- MapReduce has class Partitioner<KEY, VALUE>
  - Method int getPartition(KEY key, VALUE value, int numPartitions) allows assigning keys to partitions
- Example for numPartitions = 2
  - Partition 1 gets all numbers less than median
  - Partition 2 gets all larger numbers
- What about concatenating the output?
  - Not necessary, except for many small files (big files are broken up anyway)
- Generalizes obviously to more reducers

31

MapReduce and Key Sorting

- MapReduce environment guarantees that for each reduce task the assigned set of intermediate keys is processed in key order
  - After receiving all (key2, val2) pairs from mappers, reducer sorts them by key2, then calls Reduce on each (key2, list(val2)) group in order
- Can leverage this guarantee for partitioning sort
  - Reduce simply emits the records unchanged
  - No need for user sort code in Reduce function!

32

```
package org.apache.hadoop.examples;
import java.io.IOException; import java.net.URI; import java.util.*;
import org.apache.hadoop.conf.Configuration; import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache; import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable; import org.apache.hadoop.io.Writable; import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.mapred.lib.IdentityMapper; import org.apache.hadoop.mapred.lib.IdentityReducer;
import org.apache.hadoop.mapred.lib.InputSampler; import org.apache.hadoop.mapred.lib.TotalOrderPartitioner;
import org.apache.hadoop.util.Tool; import org.apache.hadoop.util.ToolRunner;
/**
 * This is the trivial map/reduce program that does absolutely nothing
 * other than use the framework to fragment and sort the input values.
 *
 * To run: bin/hadoop jar build/hadoop-examples.jar sort
 *          [-m <i>maps</i>] [-r <i>reduces</i>]
 *          [-inFormat <i>input format class</i>]
 *          [-outFormat <i>output format class</i>]
 *          [-outKey <i>output key class</i>]
 *          [-outValue <i>output value class</i>]
 *          [-totalOrder <i>pcnt</i> <i>num samples</i> <i>max splits</i>]
 *          <i>in-dir</i> <i>out-dir</i>
 */
public class Sort<K,V> extends Configured implements Tool {
  private RunningJob jobResult = null;

  static int printUsage() {
    System.out.println("sort [-m <maps>] [-r <reduces>] " +
                "[-inFormat <input format class>] " +
                "[-outFormat <output format class>] " +
                "[-outKey <output key class>] " +
                "[-outValue <output value class>] " +
                "[-totalOrder <pcnt> <num samples> <max splits>] " +
                "<input> <output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
}
```

Sort Code in Hadoop 1.0.3 Distribution;
part 1: boilerplate code

33

```
/**
 * The main driver for sort program.
 * Invoke this method to submit the map/reduce job.
 * @throws IOException When there is communication problems with the
 *                 job tracker.
 */
public int run(String[] args) throws Exception {

  JobConf jobConf = new JobConf(getConf(), Sort.class);
  jobConf.setJobName("sorter");

  jobConf.setMapperClass(IdentityMapper.class);
  jobConf.setReducerClass(IdentityReducer.class);

  JobClient client = new JobClient(jobConf);
  ClusterStatus cluster = client.getClusterStatus();
  int num_reduces = (int) (cluster.getMaxReduceTasks() * 0.9);
  String sort_reduces = jobConf.get("test.sort.reduces_per_host");
  if (sort_reduces != null) {
    num_reduces = cluster.getTaskTrackers() * Integer.parseInt(sort_reduces);
  }
  Class<? extends InputFormat> inputFormatClass = SequenceFileInputFormat.class;
  Class<? extends OutputFormat> outputFormatClass = SequenceFileOutputFormat.class;
  Class<? extends WritableComparable> outputKeyClass = BytesWritable.class;
  Class<? extends Writable> outputValueClass = BytesWritable.class;
  List<String> otherArgs = new ArrayList<String>();
  InputSampler.Sampler<K,V> sampler = null;
```

Sort Code in Hadoop 1.0.3 Distribution; part 2: Map and Reduce definition


```
for(int i=0; i < args.length; ++i) {
  try {
    if ("-m".equals(args[i])) {
      jobConf.setNumMapTasks(Integer.parseInt(args[++i]));
    } else if ("-r".equals(args[i])) {
      num_reduces = Integer.parseInt(args[++i]);
    } else if ("-inFormat".equals(args[i])) {
      inputFormatClass =
        Class.forName(args[++i]).asSubclass(InputFormat.class);
    } else if ("-outFormat".equals(args[i])) {
      outputFormatClass =
        Class.forName(args[++i]).asSubclass(OutputFormat.class);
    } else if ("-outKey".equals(args[i])) {
      outputKeyClass =
        Class.forName(args[++i]).asSubclass(WritableComparable.class);
    } else if ("-outValue".equals(args[i])) {
      outputValueClass =
        Class.forName(args[++i]).asSubclass(Writable.class);
    } else if ("-totalOrder".equals(args[i])) {
      double pcnt = Double.parseDouble(args[++i]);
      int numSamples = Integer.parseInt(args[++i]);
      int maxSplits = Integer.parseInt(args[++i]);
      if (0 >= maxSplits) maxSplits = Integer.MAX_VALUE;
      sampler =
        new InputSampler.RandomSampler<K,V>(pcnt, numSamples, maxSplits);
    } else {
      otherArgs.add(args[i]);
    }
  } catch (NumberFormatException except) {
    System.out.println("ERROR: Integer expected instead of " + args[i]);
    return printUsage();
  } catch (ArrayIndexOutOfBoundsException except) {
    System.out.println("ERROR: Required parameter missing from " + args[i-1]);
    return printUsage(); // exits
  }
}
```

Sort Code in Hadoop 1.0.3 Distribution; part 3: more boilerplate code


```
  // Set user-supplied (possibly default) job configs
  jobConf.setNumReduceTasks(num_reduces);
  jobConf.setInputFormat(inputFormatClass);
  jobConf.setOutputFormat(outputFormatClass);
  jobConf.setOutputKeyClass(outputKeyClass);
  jobConf.setOutputValueClass(outputValueClass);

  // Make sure there are exactly 2 parameters left.
  if (otherArgs.size() != 2) {
    System.out.println("ERROR: Wrong number of parameters: " + otherArgs.size() + " instead of 2.");
    return printUsage();
  }
  FileInputFormat.setInputPaths(jobConf, otherArgs.get(0));
  FileOutputFormat.setOutputPath(jobConf, new Path(otherArgs.get(1)));

  if (sampler != null) {
    System.out.println("Sampling input to effect total-order sort...");
    jobConf.setPartitionerClass(TotalOrderPartitioner.class);
    Path inputDir = FileInputFormat.getInputPaths(jobConf)[0];
    inputDir = inputDir.makeQualified(inputDir.getFileSystem(jobConf));
    Path partitionFile = new Path(inputDir, "_sortPartitioning");
    TotalOrderPartitioner.setPartitionFile(jobConf, partitionFile);
    InputSampler.<K,V>writePartitionFile(jobConf, sampler);
    URI partitionUri = new URI(partitionFile.toString() + "#" + "_sortPartitioning");
    DistributedCache.addCacheFile(partitionUri, jobConf);
    DistributedCache.createSymlink(jobConf);
  }

  System.out.println("Running on " + cluster.getTaskTrackers() + " nodes to sort from " + FileInputFormat.getInputPaths(jobConf)[0] +
    " into " + FileOutputFormat.getOutputPath(jobConf) + " with " + num_reduces + " reduces.");
  Date startTime = new Date(); System.out.println("Job started: " + startTime);
  jobResult = JobClient.runJob(jobConf);
  Date end_time = new Date(); System.out.println("Job ended: " + end_time);
  System.out.println("The job took " + (end_time.getTime() - startTime.getTime()) /1000 + " seconds.");
  return 0;
}
```

Sort Code in Hadoop 1.0.3 Distribution; part 4: job settings, sampling to get pivot elements, run job command

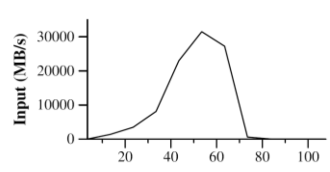

```
public static void main(String[] args) throws Exception {
  int res = ToolRunner.run(new Configuration(), new Sort(), args);
  System.exit(res);
}

/**
 * Get the last job that was run using this instance.
 * @return the results of the last job that was run
 */
public RunningJob getResult() {
  return jobResult;
}
}
```

Sort Code in Hadoop 1.0.3 Distribution; part 5: main function

Google Paper Experiments

- 1800 machine cluster
  - 2 GHz Xeon, 4 GB memory, two 160 GB IDE disks, gigabit Ethernet link
  - Less than 1 msec roundtrip time
- Grep workload
  - Scan 10^{10} 100-byte records, search for rare 3-character pattern, occurring in 92,337 records
  - M=15,000 (64 MB splits), R=1

Grep Progress Over Time



- Rate at which input is scanned as more mappers are added
- Drops as tasks finish, done after 80 sec
- 1 min startup overhead beforehand
  - Propagation of program to workers
  - Delays due to distributed file system for opening input files and getting information for locality optimization

Sort

- Sort 10^{10} 100-byte records (~1 TB of data)
- Less than 50 lines user code
- M=15,000 (64 MB splits), R=4000
- Use key distribution information for intelligent partitioning
- Entire computation takes 891 sec
  - 1283 sec without backup task optimization (few slow machines delay completion)
  - 933 sec if 200 out of 1746 workers are killed several minutes into computation

MapReduce at Google (2004)

- Machine learning algorithms, clustering
- Data extraction for reports of popular queries
- Extraction of page properties, e.g., geographical location
- Graph computations
- Google indexing system for Web search (>20 TB of data)
  - Sequence of 5-10 MapReduce operations
  - Smaller simpler code: from 3800 LOC to 700 LOC for one computation phase
  - Easier to change code
  - Easier to operate, because MapReduce library takes care of failures
  - Easy to improve performance by adding more machines

Summary

- Programming model that hides details of parallelization, fault tolerance, locality optimization, and load balancing
- Simple model, but fits many common problems
  - User writes Map and Reduce function
  - Can also provide combine and partition functions
- Implementation on cluster scales to 1000s of machines
- Open source implementation, Hadoop, is available

MapReduce relies heavily on the underlying distributed file system. Let’s take a closer look to see how it works.

The Distributed File System

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003

Motivation

- Abstraction of a single global file system greatly simplifies programming in MapReduce
- MapReduce job just reads from a file and writes output back to a file (or multiple files)
- Frees programmer from worrying about messy details
  - How many chunks to create and where to store them
  - Replicating chunks and dealing with failures
  - Coordinating concurrent file access at low level
  - Keeping track of the chunks

Google File System (GFS)

- GFS in 2003: 1000s of storage nodes, 300 TB disk space, heavily accessed by 100s of clients
- Goals: performance, scalability, reliability, availability
- Differences compared to other file systems
  - Frequent component failures
  - Huge files (multi-GB or even TB common)
  - Workload properties
    - Design system to make important operations efficient

46

Data and Workload Properties

- Modest number of large files
  - Few million files, most 100 MB+
  - Manage multi-GB files efficiently
- Reads: large streaming (1 MB+) or small random (few KBs)
- Many large sequential append writes, few small writes at arbitrary positions
- Concurrent append operations
  - E.g., Producer-consumer queues or many-way merging
- High sustained bandwidth more important than low latency
  - Bulk data processing

47

File System Interface

- Like typical file system interface
  - Files organized in directories
  - Operations: create, delete, open, close, read, write
- Special operations
  - Snapshot: creates copy of file or directory tree at low cost
  - Record append: concurrent append guaranteeing atomicity of each individual client's append

48

Architecture Overview

- 1 master, multiple chunkservers, many clients
  - All are commodity Linux machines
- Files divided into fixed-size chunks
  - Stored on chunkservers' local disks as Linux files
  - Replicated on multiple chunkservers
- Master maintains all file system metadata: namespace, access control info, mapping from files to chunks, chunk locations

49

Why a Single Master?

- Simplifies design
- Master can make decisions with global knowledge
- Potential problems:
  - Can become bottleneck
    - Avoid file reads and writes through master
  - Single point of failure
    - Ensure quick recovery

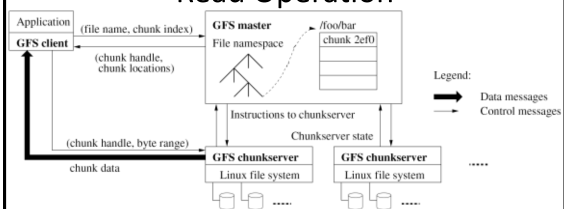
50

High-Level Functionality

- Master controls system-wide activities like chunk lease management, garbage collection, chunk migration
- Master communicates with chunkservers through HeartBeat messages to give instructions and collect state
- Clients get metadata from master, but access files directly through chunkservers
- No GFS-level file caching
  - Little benefit for streaming access or large working set
  - No cache coherence issues
  - On chunkserver, standard Linux file caching is sufficient

51

Read Operation



- Client: from (file, offset), compute chunk index, then get chunk locations from master
  - Client buffers location info for some time
- Client requests data from nearby chunkserver
  - Future requests use cached location info
- Optimization: batch requests for multiple chunks into single request

Chunk Size

- 64 MB, stored as Linux file on a chunkserver
- Advantages of large chunk size
  - Fewer interactions with master (recall: large sequential reads and writes)
  - Smaller chunk location information
    - Smaller metadata at master, might even fit in main memory
    - Can be cached at client even for TB-size working sets
- Disadvantage: fewer chunks => fewer options for load balancing
  - Fixable with higher replication factor
  - Address hotspots by letting clients read from other clients

Practical Considerations

- Number of chunks is limited by master's memory size
  - Only 64 bytes metadata per 64 MB chunk; most chunks full
  - Less than 64 bytes namespace data per file
- Chunk location information at master is not persistent
  - Master polls chunkservers at startup, then updates info because it controls chunk placement
  - Eliminates problem of keeping master and chunkservers in sync (frequent chunkserver failures, restarts)

Consistency Model

- GFS uses a relaxed consistency model
- File namespace updates are atomic (e.g., file creation)
  - Only handled by master, using locking
  - Operations log defines global total order
- State of file region after update
  - Consistent: all clients will always see the same data, regardless which chunk replica they access
  - Defined: consistent and reflecting the entire update

Relaxed Consistency

- GFS guarantees that after a sequence of successful updates, the updated file region is defined and contains the data of the last update
  - Applies updates to all chunk replica in same order
  - Uses chunk version numbers to detect stale replica (when chunk server was down during update)
- Stale replica are never involved in an update or given to clients asking the master for chunk locations
- But, client might read from stale replica when it uses cached chunk location data
  - Not all clients read the same data
  - Can address this problem for append-only updates

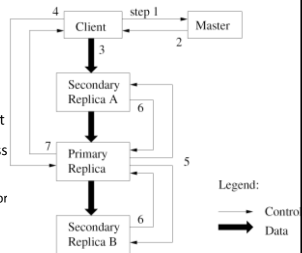
Leases, Update Order

- Leases used for consistent update order across replicas
  - Master grants lease to one replica (primary)
  - Primary picks serial update order
  - Other replicas follow this order
- Lease has initial timeout of 60 sec, but primary can request extensions from master
  - Piggybacked on HeartBeat messages
  - Master can revoke lease (e.g., to rename file)
  - If no communication with primary, then master grants new lease after old one expires

Updating a Chunk

1. Who has lease?
2. Identity of primary and secondary replicas
3. Push data to all replicas
4. After receiving all acks, send write request to primary who assigns it a serial number
5. Primary forwards write request to all other replicas
6. Secondaries ack update success
7. Primary replies to client
    1. Also reports errors
    2. Client retries steps 3-7 on error

- Large writes broken down into chunks



58

Data Flow

- Decoupled from control flow for efficient network use
- Data pipelined linearly along chain of chunkservers
    - Full outbound bandwidth for fastest transfer (instead of dividing it in non-linear topology)
    - Avoids network bottlenecks by forwarding to “next closest” destination machine
    - Minimizes latency: once chunkserver receives data, it starts forwarding immediately
        - Switched network with full-duplex links
        - Sending does not reduce receive rate
        - 1 MB distributable in 80 msec

59

Namespace Management

- Want to support concurrent master operations
- Solution: locks on regions of namespace for proper serialization
    - Read-write lock for each node in namespace tree
        - Operations lock all nodes on path to accessed node
            - For operation on /d1/d2/leaf, acquire read locks on /d1 and /d1/d2, and appropriate read or write lock on /d1/d2/leaf
        - File creation: read-lock on parent directory
    - Concurrent updates in same directory possible, e.g., multiple file creations
    - Locks acquired in consistent total order to prevent deadlocks
        - First ordered by level in namespace tree, then lexicographically within same level

60

Replica Placement

- Goals: scalability, reliability, availability
- Difficult problem
    - 100s of chunkservers spread across many machine racks, accessed from 100s of clients from the same or different racks
    - Communication may cross network switch(es)
    - Bandwidth into or out of a rack may be less than aggregate bandwidth of all the machines within the rack
- Spread replicas across racks
    - Good: fault tolerance, reads benefit from aggregate bandwidth of multiple racks
    - Bad: writes flow through multiple racks
- Master can move replicas or create/delete them to react to system changes and failures

61

Lazy Garbage Collection

- File deletion immediately logged by master, but file only renamed to hidden name
    - Removed later during regular scan of file system namespace
    - Batch-style process amortizes cost and is run when master load is low
- Orphaned chunks identified during regular scan of chunk namespace
- Chunkservers report their chunks to master in HeartBeat messages
- Master replies with identities of chunks it does not know
    - Chunkserver can delete them
- Simple and reliable: lost deletion messages (from master) and failures during chunk creation no problem
- Disadvantage: difficult to finetune space usage when storage is tight, e.g., after frequent creation/deletion of temp files
    - Solution: use different policies in different parts of namespace

62

Stale Replicas

- Occur when chunkserver misses updates while it is down
- Master maintains chunk version number
    - Before granting new lease on chunk, master increases its version number
    - Informs all up-to-date replicas of new number
        - Master and replicas keep version number in persistent state
    - This happens before client is notified and hence before it can start updating the chunk
- When chunkservers report their chunks, they include version numbers
    - Older than on master: garbage collect it
    - Newer than on master: master must have failed after granting lease; master takes higher version to be up-to-date
- Master also includes version number in reply to client and chunkserver during update-process related communication

63

Achieving High Availability

- Master and chunkservers can restore state and start in seconds
- Chunk replication
- Master replication, i.e., operation log and checkpoints
- But: only one master process
  - Can restart almost immediately
  - Permanent failure: monitoring infrastructure outside GFS starts new master with replicated operation log (clients use DNS alias)
- Shadow masters for read-only access
  - May lag behind primary by fraction of a sec

Experiments

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

- Chunkserver metadata mostly checksums for 64 KB blocks
  - Individual servers have 50-100 MB of metadata
  - Reading this from disk during recovery is fast

Results

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

- Clusters had been up for 1 week at time of measurement
- A's network configuration has max read rate of 750 MB/s
  - Actually reached sustained rate of 580 MB/s
- B's peak rate is 1300 MB/s, but applications never used more than 380 MB/s
- Master not a bottleneck, despite large number of ops sent to it

Summary

- GFS supports large-scale data processing workloads on commodity hardware
- Component failures treated as norm, not exception
  - Constant monitoring, replicating of crucial data
  - Relaxed consistency model
  - Fast, automatic recovery
- Optimized for huge files, appends, large sequential reads
- High aggregate throughput for concurrent readers and writers
  - Separation of file system control (through master) from data transfer (between chunkservers and clients)