Google Technology Round-Up

Philip Snowberger

Department of Computer Science and Engineering
University of Notre Dame

2007-10-30

Outline

Introduction

The Google Filesystem

Chubby

Bigtable

MapReduce

Summary

Google technologies

- Google Filesystem (GFS)
  - Fault-tolerant, high-performance cluster filesystem
  - Re-examined assumptions about filesystems
- Chubby
  - Highly-available distributed locking service
- Bigtable
  - Distributed storage service
  - Serves semi-structured key/value pairs
  - Meets both performance and availability goals for various workloads
- MapReduce
  - Programming model+implementation
  - Leverages GFS and Bigtable for performance wins

Outline

Introduction

The Google Filesystem

Chubby

Bigtable

MapReduce

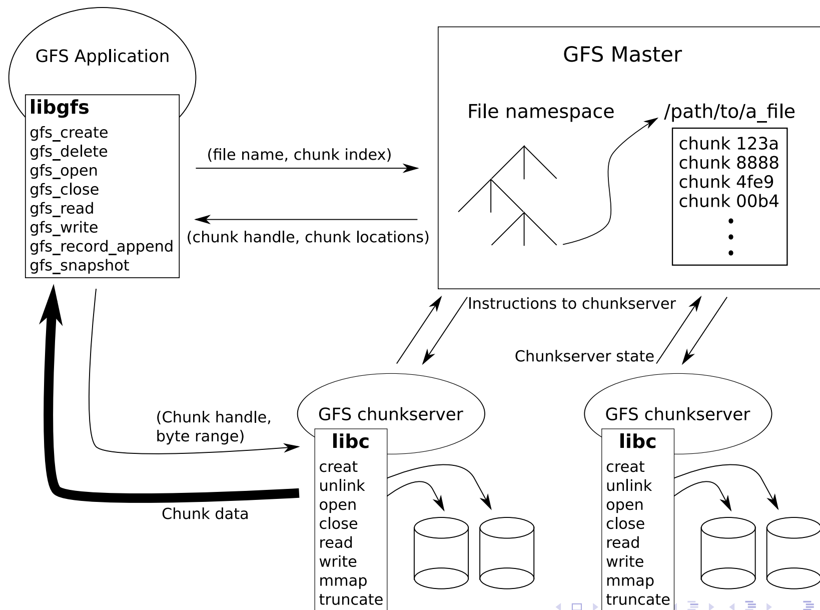
Summary

GFS: Challenging assumptions

- Hardware is unreliable; systems should be built to robustly withstand all kinds of faults and errors
- Traditional models of filesystem usage did not fit Google's use case
  - lots of multi-GB files instead of millions of multi-KB files
  - the most common access patterns were: multiple clients doing short appends, a single client doing one extremely long append, and multiple clients doing sequential reads of entire files
- Traditional consistency models didn't quite fit
- Sustained throughput was more important than latency

Out of the ashes...

- GFS monitors itself to detect and react to component failures
- GFS is optimized for sequential reads, small random reads, and sequential *record appends*
  - Multiple clients can append records simultaneously and GFS guarantees that each record will get written *at least* once
- Small random writes are supported, but not optimized
- The POSIX filesystem API isn't supported, but many similar operations are:
  - *create*, *delete*, *open*, *close*, *read*, *write*
  - *snapshot*, *record append*


GFS Application
libgfs
gfs_create
gfs_delete
gfs_open
gfs_close
gfs_read
gfs_write
gfs_record_append
gfs_snapshot
(file name, chunk index)
(chunk handle, chunk locations)
GFS Master
File namespace
/path/to/a_file
chunk 123a
chunk 8888
chunk 4fe9
chunk 00b4
Instructions to chunkserver
Chunkserver state
(Chunk handle,
byte range)
Chunk data
GFS chunkserver
libc
creat
unlink
open
close
read
write
mmap
truncate
GFS chunkserver
libc
creat
unlink
open
close
read
write
mmap
truncate

GFS architecture

- A GFS cluster has a single master and multiple *chunk servers*
- Files are divided into fixed-size *chunks*
  - Chunks are replicated across multiple chunk servers
- The master controls all the metadata such as the namespace, the locations of each chunk, and so on
- Neither the clients nor the chunk servers cache the chunks
- The chunk size is commonly fixed at 64 MB, and each chunk is stored as a regular file on a regular Linux filesystem

Single master

- The single-master constraint simplifies consistency
- All metadata operations go through the master
- All mutations are written to an operation log that is replicated to multiple other servers before the mutation 'succeeds'
- The master is responsible for maintaining replication levels for chunks, as well as rebalancing chunks when new chunk servers are added
  - *Thought experiment: should pre-existing chunks be rebalanced to the new chunk server, or should the master just let it fill up with chunk writes?*

Multiple chunk servers

- Replicas for any given chunk are spread out as much as possible to minimize the chance of a hardware failure or network partition making any chunk unavailable
- Read operations can therefore use the aggregate bandwidth available to all the chunk servers holding replicas for the chunk being read
- Each chunk server (in Google's clusters) has a single gigabit link; with good separation between replicas, many such links can be used to serve client requests

Outline

Introduction

The Google Filesystem

Chubby

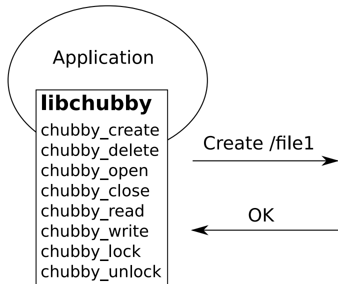
Bigtable

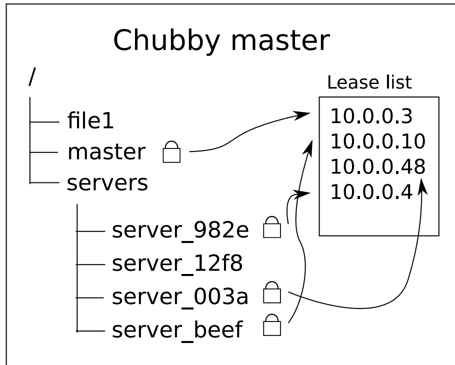
MapReduce

Summary

Chubby: a distributed locking service

- Chubby is a Google project that offers a distributed locking service
- A Chubby service exports a filesystem-like namespace; clients can lock any file or directory, or populate/query the contents of files and directories
- When a client loses access to the Chubby service (e.g. by a network partition), it loses its session lease after the lease expiration time
- When a client loses its session lease, it loses any Chubby locks and open Chubby file handles it may have had


Application
libchubby
chubby_create
chubby_delete
chubby_open
chubby_close
chubby_read
chubby_write
chubby_lock
chubby_unlock
Create /file1
OK


Chubby master
/
file1
master
servers
server_982e
server_12f8
server_003a
server_beef
Lease list
10.0.0.3
10.0.0.10
10.0.0.48
10.0.0.4

Outline

Introduction

The Google Filesystem

Chubby

Bigtable

MapReduce

Summary

Bigtable: A faster, better database

- Bigtable is a distributed storage system for quickly storing and accessing semi-structured data
- Bigtable is scalable in terms of
  - capacity
  - throughput and latency
- Rather than relational data, Bigtable stores key/value pairs:

  key (row:string, column:string, time:int64)
  value any arbitrary byte string
- Bigtable uses GFS and Chubby, we'll soon see how

So what does "semi-structured data" mean?

- A Bigtable consists of a sequence of rows, each of which has a variable number of columns
- Row keys are arbitrary byte strings $\leq$ 64 kB, whereas column keys are arbitrary strings of the form *family:qualifier*
- These "column families", such as "anchor:" and "contents:", are grouped together on disk, are the basic unit of access control in Bigtable

Key			Value
row key	column key	timestamp	
"com.cnn.www"	"contents:"	t3	"<html> ..."
	"contents:"	t5	"<html> ..."
	"anchor:yahoo.com"	*t8*	"CNN"
	"anchor:news.google.com"	*t9*	"CNN.com"
"com.irisheyes.www"	"contents:"	t7	"<html> ..."
	"anchor:ndnation.com"	*t18*	"ND Nation"
"com.ndnation.www"	"contents:"	t11	"<html> ..."
	"anchor:irisheyes.com"	*t22*	"Irisheyes.com"

Benefits of grouping columns families together

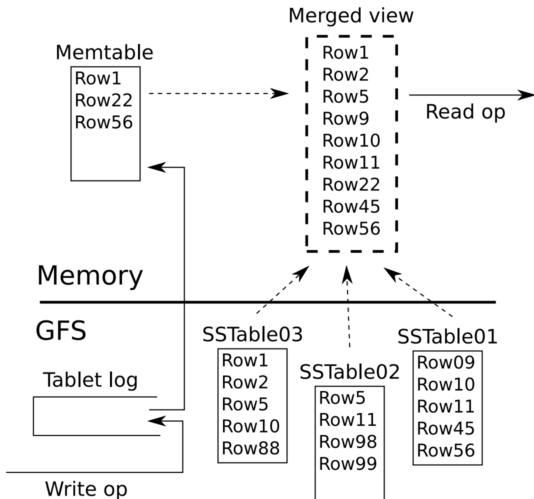
- Each column family can be pegged into memory or not, and have different compression algorithms turned on or off
  - "img:" family is probably already compressed (it's images)
  - "contents:" family compresses well (it's blobs of HTML)
- Applications quite often scan through an entire row space, reading out the contents of a single column family
  - For instance, a WWW link mapping program doesn't care about the contents of web pages, just the anchors linking them to one another
- If the column family is stored sequentially on disk, this operation is fast

Bigtable data structures

- Rows are stored lexicographically sorted by the row keys
- Row ranges, called "tablets", are the unit of distribution
- There is a single master server and enough tablet servers that each tablet server holds about 10-1000 tablets
- Tablets are serialized onto disk into a format called SSTable, which provides a persistent, ordered immutable key/value map

What other Google platform services does Bigtable use?

- Bigtable stores many of its data structures in GFS:
  - When clients send a row mutation request, the tablet server first logs the request to GFS before playing the log into its in-memory cache
  - To reduce memory requirements of the tablet server, when the in-memory cache gets too large, the tablet server will serialize it into an SSTable and write it to GFS, and then create a new, empty in-memory cache
  - These SSTables are referred to upon read requests, since the tablet server must create a merged view of the in-memory cache and all the SSTables to service the read request
  - To keep down the complexity of this merge, tablet servers will periodically merge SSTables together (this is fast, but why?)


Merged view
Memtable
Row1
Row22
Row56
Row1
Row2
Row5
Row9
Row10
Row11
Row22
Row45
Row56
Read op
Memory
GFS
SSTable03
Row1
Row2
Row5
Row10
Row88
SSTable02
Row5
Row11
Row98
Row99
SSTable01
Row09
Row10
Row11
Row45
Row56
Tablet log
Write op

What other Google platform services does Bigtable use?

- Bigtable relies on Chubby for locking
  - Bigtable ensures that there can only be one master at any time by making the tablet master acquire an exclusive lock on a uniquely-named file in Chubby as its first action
  - Tablet servers create a file in a specific Chubby directory and acquire an exclusive lock on it; the tablet master periodically checks this directory for tablet server additions
  - When a tablet server loses its Chubby session lease, it loses its lock on the above file; the master server notices this and starts up a new tablet server to replace the old one; when the old tablet server realizes it has lost its Chubby lock, it terminates

Outline

Introduction

The Google Filesystem

Chubby

Bigtable

MapReduce

Summary

MapReduce: turning computation on its head(s)

- MapReduce is a programming model and library implementation for parallel programming on clusters
- The programmer writes at least a *map* and a *reduce* function (additional functions can improve performance in some cases)
- The MapReduce implementation spawns input-consuming "mappers" on cluster nodes, and when they're finished, it spawns "reducers" on cluster nodes to produce the output
- One of the key features of MapReduce is that it operates on input that is already spread throughout a cluster in some way
  - GFS and Bigtable are two ways to realize this data dispersal
  - MapReduce was originally written with GFS support but a wrapper was recently written to make it work with Bigtable too

The MapReduce programming model

input	key/value pairs of type (k1,v1)
Map	(k1,v1) $\longrightarrow$ list(k2,v2)
Reduce	(k2,list(v2)) $\longrightarrow$ list(v2)
output	list(v2)

Example: counting words

input key: name of document
value: contents of document

Map for each word in value:
    EmitIntermediate(word, "1")

Reduce result = 0
for each v in values:
    result += ParseInt(v)
Emit(AsString(result))

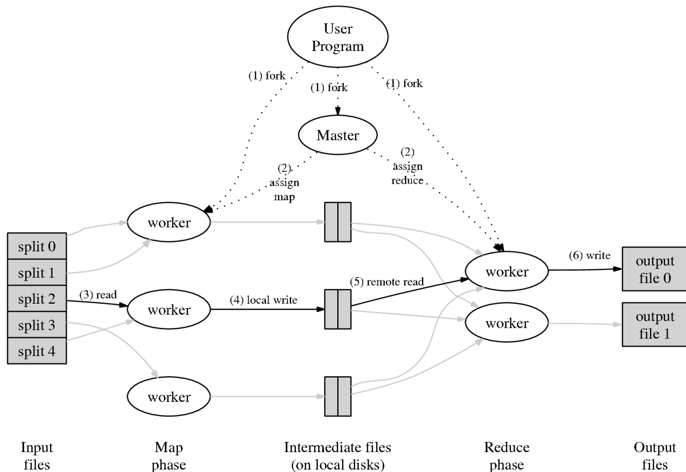
output key: word
value: number of times that key word appears in all the documents

So where does GFS/Bigtable come in?

- Those input key/value pairs need to already be distributed around the cluster
- If a GFS file contained nothing but key/value pairs in a known format, the MapReduce implementation can just split the file up along GFS chunk boundaries and distribute them to Mappers
- Likewise if all the key/value pairs were in a Bigtable, they could be parceled out perfectly evenly by making each Mapper responsible for $\frac{\text{\# of pairs}}{\text{\# of Mappers}}$ pairs

That's not all

- In addition to getting input from GFS and Bigtable, MapReduce writes the output of the Map phase directly to storage before invoking the Reduce phase.
- Finally, the results of the MapReduce computation are available as just more rows in the Bigtable or another file in GFS, chunked over the whole cluster



* adapted from "MapReduce: Simplified Data Processing on Large Clusters", Jeffrey Dean and Sanjay Ghemawat, appeared in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA,

Outline

Introduction

The Google Filesystem

Chubby

Bigtable

MapReduce

Summary

Why does any of this matter to me?

- There is a Free Software implementation of GFS and MapReduce in Java called Hadoop from the Apache Software Foundation
- There is a (non-distributed) implementation of just MapReduce written for C++ by TrollTech (makers of Qt)
- However, so far, Google hasn't released any of GFS, Chubby, Bigtable, or MapReduce to the public in the form of open-source code
- Is Google just building up the walls of their ivory tower? Are these technologies going to benefit Google exclusively in their quest to "to organize the world's information and make it universally accessible and useful"?

Conclusions

- The designs of GFS and Bigtable each discarded several traditional notions of filesystems and of databases, realizing big performance and reliability/availability wins
- MapReduce, inspired by Lisp's "map" and "reduce" functions, has opened up a new way of doing cluster-wide high performance computing, all while leveraging existing technologies
- Google has firmly established itself as a company where hairy engineering research gets done at a tremendous pace

Potential exam questions (page 1/2)

- Succinctly describe two design decisions on which the designers of GFS took the path less traveled.
- Say an application wants to write some number of records containing the same data to GFS. How could the application know if a given record was a duplicate (as allowed by GFS's "at *least* once" record appends)?
- Given the usage patterns observed by the GFS designers, when a new (empty) chunk server is added to a GFS cluster, should the master move some chunk replicas to the new server to balance cluster load, or should the master leave it alone, knowing that clients will soon fill up the new server's free space?

Potential exam questions (page 2/2)

- Think about the “merged view” that Bigtable tablet servers use to serve read requests. How could they efficiently handle a “delete row” request (*without* expunging the row from all the SSTables on disk)?
- Why is it necessary for Bigtable tablet servers to periodically merge its SSTables together? Is it efficient? How?