Google File System

Web and scalability

The web:

- How big is the Web right now? No one knows.
- Number of pages that are crawled:
  - 100,000 pages in 1994
  - 8 million pages in 2005
- Crawlable pages might be 4x
- Deep web might be 100x

Why search service (such as Google) must be scalable:

- Indexing, document, advertisement, trends (google books, analytics, etc.)
- Google service is used beyond the search interface at google.com
- In 2000, Google was handling 5.5 million searches per day
- In 2004, estimated 250 million searches per day
- Hence, server must be scalable and available all the time

Google Principles

Scalable and available.

Need many machines:

- If you use high-end machines, cost is not scalable
- Some people joke they were "smart but poor", which actually lead to novel solution
- **If we focus on the availability and reliability of *each* component of the systems (e.g. high-end disks, reliable memory, etc.), the cost is too high**
- Accept that component failures are the norm rather than the exception
    - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies

Use commodity components (i.e. cheap, non-high-end machines):

- **Build availability *around* these cheap machines/components**
    - It is okay for hardware to fail
    - We will buy more hardware
    - And write a software that manages these failures (e.g. by doing replication, fail-over, etc.)
- Cheap machines/elements scale both operationally and economically
    - No need to fix broken computers (otherwise, major overhead)
- Dedicated computers
    - In 2000, 2500 computers are dedicated for search
    - In 2004, estimated 15,000 computers
    - In 2007, (Google and Yahoo combined) 250,000+ computers

Designs:

- Make internal architecture transparent
    - Users do not need to worry about fault-tolerance, replication, load balancing, failure management, file placement, management of large files, etc.
    - **Solution: Google File System**
- A nice programming environment
    - Number of programmer grows, difficult maintain the quality of software (need common framework)
    - Want to be able to run jobs on hundreds of machines
    - **Solution: MapReduce**

Google File System

<u>Billion of objects, huge files:</u>

- Google has lots of data. A typical file is a GB-file. Cannot fit in traditional file system. Hence, need to build another file system on top of a file system
- Why create a new file system? Why not used NFS/AFS?
    - They are a company.
    - They have their own workload (search-type workload), and can tune their own systems.
    - Lessons learned: if cannot find anything suitable out there for your workload, it's better to create one.
- Implications:
    - Their interface is not POSIX compliant, e.g. not like open(path, flags).
    - They can add additional operations (e.g. snapshot, record append)

<u>Architecture of GFS cluster:</u>

- Basically, another file system on top of local file system
    - Client sees a typical file system tree
    - GFS client and server cooperates to find where the file is located in the cluster
- A file is comprised of fixed-size chunks
    - (Analogy in file system: a file is comprised of disk blocks)
    - (A chunk is basically a file in the local file system)
    - 1 chunk is 64-MB
    - A chunk is named with 64-bit unique global IDs, e.g. 0x2ef0-….)
    - A chunk is 3-way mirrored across chunkservers
    - Why do we need to have "chunk"?
        - If we store a GFS file as a local file, each file can be arbitrary long (e.g. some are as big as 100-GB, some might be small)
        - Managing replicas of arbitrary size might be cumbersome
        - Better, we need a **unit** of replication. Hence, a chunk is introduced.
        - Does this mean we have lots of internal fragmentation since the chunk size is too big? Yes, but no need to worry. Disks are cheap, and they buy lots of disks. Hence, there is no space constraint (unlike when we talk about memory management – memory is scarce, hence internal fragmentation is bad)
        - In summary, *use another level of indirection* to solve some computing problem
        - In the old case we have: bits → blocks → local file
        - In GFS: bits → blocks → local files (chunks) → a GFS file
    - Why large chunk-size?
        - A chunk is a unit of allocation, replication, etc., hence the server needs to know where a chunk is located
        - Assume, 64 bytes to describe a 64-MB chunk (1 / 1M space overhead).
        - But if 64 bytes to describe a 64-KB chunk (1 / 1K space overhead)

        - Remember there are millions of chunks in the server (hence, less space overhead is better)
        - Clients don't need much interaction with masters to gather access to a lot of data
- **A single master and multiple (hundreds) chunkservers per master**
    - Each server is a Linux machine (connected with tens of disks)
- The largest GFS clusters:
    - 1000+ storage nodes.
    - 300+ terabytes of disk storage.

    - Master maintains metadata
        - Basic: file and chunk namespaces, file-to-chunk mappings, locations of a chunk's replicas
        - Misc: Namespace, access control, file-to-chunk mappings, garbage collection, chunk migration.
    - Chunkservers respond to data requests
    - Open(fileA) → master; then master tells client the chunkservers for chunks in file A; then client will only interact with chunkservers
    - **Why the old concept of servers that server both metadata and data is not preferred in this case?**
        - Lots of data transfer
            - If a server serves both metadata and data, the server's memory might be loaded with data, and hence the metadata in cache is swapped out to disk
            - The implication, many metadata operations will be slow
        - Data transfers occupy network heavily
            - Hence, metadata operations might be slow, buried in the data transfers
            - If we have different servers for metadata and data, the network link that goes to the metadata server can be a different link or a dedicated link
    - **GFS master server only serves metadata operations**
        - Since only keep 64 bytes to describe a chunk, almost all chunk metadata can be stored in the GFS cache!
        - If I have a 64-GB memory at the server, at most I can keep 1-G (1 million) chunk metadata in the cache!

<u>Google workload:</u>
- Basically: hundreds of web-crawling applications
- Reads: small random reads and large streaming reads
- Writes:
    - Many files are written once, and read sequentially (e.g. statistics about web pages, or search results, etc.)
    - Random writes are non-existent. Hence, most modifications are **appends** (add more information to current search results, etc.)

- **Why random writes are not fully supported?**
  - First, it's cumbersome. For example, let's say I have a file that contains the result of a previous search. The file has two records:
    - www.page1.com → www.my.blogspot.com
    - www.page2.com → www.my.blogspot.com
  - The file basically says that there are two pages that have link to my blog.
  - Let's say I want to run the search again tonight, and turns out that page2 no longer has the link, but there is a new page, page3, that has a link to my blog:
    - www.page1.com → www.my.blogspot.com
    - www.page3.com → www.my.blogspot.com
  - *The classic way is just to go to the old file, delete the old record (page2), and insert a new record (page3). This is cumbersome for distributed computing!*
  - This is cumbersome, because the program might be run in parallel by many machines. Remember that in real life, the program crawls to million of pages (wait until we see MapReduce in action to understand what we mean by "run in parallel by many machines"). *Hence, to update (random write) the file properly we need locking!* And locking is just way too complex.
  - Hence, a better way is to just *delete the old file*, create a new file where this program (run on more than one machines) can append new records to the file "atomically" (see next bullets below).

Record append:

- GFS is famous with its "record append" operation.
  - Google's data is constantly updated with large sequential writes
  - Hence, optimize case for record append rather than random write
  - What happens to old data (e.g. old search results, old pages)?
    - Deleted, and space can be reused
    - If need more, buy more disks, disks are cheap!

Atomic record append:

- **How to have two clients (or even more) append records to a file atomically?**
  - Model 1:
    - client gets the metadata from the server (then client knows the file size, e.g. 100 bytes), then each client says append(100, mydata); what will happen?
  - Model 2:
    - need distributed locking. Client 1 and client 2 try to get a lock from the server to append the file. When a client finishes appending the file, the server is notified, client 2 is updated, e.g. current file size is 110 bytes). What's bad thing?
  - Model 3:
    - Client specifies the data to write to GFS client (e.g. how big);

        - GFS client contacts the master, and master chooses and returns the offset it writes to and appends the data to each replica **at least once**
        - No need for a distributed lock manager, GFS server chooses the offset, not the client
        - For example:
            - client 1 says I want to append 10 bytes, client 2 says I want to append 5 bytes.
            - GFS client for client 1 sends request to the server specifying the intent to append 10 bytes. The master server grants this request by returning offset 100, and increasing the file size to 110 bytes. Then GFS client can proceed by calling record_append(file, offset=100, data).
            - GFS client for client 2 sends request to the server specifying the intent to append 5 bytes. The server grants this request by returning offset 110, and increasing the file size to 115 bytes. Then, GFS client can proceed by calling record_Append(file, offset=110, data)
            - Why the master server returns offset 110 to the 2nd client? Because the master server already "granted" 10 bytes to GFS client 1 (hence, the file size has increased to 110 bytes although the last 10 bytes might have not been written).
        - What's the property of this record_append? Idempotent operation!
            - Since client already gets the offset (like in NFS), if the record_append fails, client can simply retry the operation again and again.

<u>Server: stateless</u>

- No states about clients
- In fact, even no caching at client, why?
    - o Because most program only cares about the output
    - o The output (e.g. search result is produced by the chunkservers, which also act as worker machines)
    - o In other words, client typically only says "please run this job (a search query)"
    - o More clear when we talk about MapReduce
- If client wants to get the latest up-to-date result, rerun the program

- Use heartbeat messages to monitor servers
- Why poll/on-demand? Rather than coordinate?
    - o On-demand wins when changes (failures) are often

<u>Fault-management:</u>

- Single-master, bad?
    - o But there also exists shadow-master
    - o If master is down. Read is sent to the shadow server
    - o Writes pause for 30-60 secs while new master and chunkservers synchronize
- How to know a server or a disk is dead?

    - Heartbeat message
    - What happens if a server/disk is dead? Regenerate data
        - Kill a chunk server with ~0.5 TB data. (all chunks were restored in around 30 minutes)
        - Killed two (duplicate) chunk servers
            - Since data was down to one copy, replication was high priority.
            - All chunks had at least 2 copies in a couple of minutes.
- **Why 3 copies?**
    - Why not 1? Obvious, no backup.
    - Why not 2?
        - **Data can be lost silently (latent sector fault: where one block of a disk becomes unavailable, e.g. scratched)**
        - Heartbeat is **only** for detecting when machine or disk is **down**
        - Latent sector faults do not mean that the whole disk is down
        - How to detect a latent sector fault? Only when you read the block.
        - When do you read the block? When application needs the data, or when the server runs a "scrubbing" utility that reads all blocks on the disk (however this is run very rarely).
        - Hence if only keep 2 copies, there is a window of vulnerability (between when *the 1st copy is lost silently* and when *this 1st copy is detected to be dead* (i.e. when this copy is read by some operations)).
        - If the 2nd copy is also lost in this window of vulnerability, there is no way to recover the data
        - By increasing the number of replicas to 3, we reduce the window of vulnerability
- **No single point of failure**, i.e. which machines to store to replicas?
    - If put in the same disk, a disk can break
    - If put in the same machine, the machine can crash
    - If put in the same rack, the power cable attached to the rack can be unplugged accidentally
    - If put in the same room, the cooling system can malfunction
    - If put in the same building, there could be fire or city power is down

<u>Other optimizations:</u>

- Pooling and Balancing
    - Put new replicas on chunkservers with below avg. disk utilization
    - Rebalances replicas periodically
- Deletion
    - Stale data: unfinished replicated creation, lost deletion messages, etc., chunks that was thought to be lost (due to down time)
    - Storage capacity is reclaimed slowly (lazy deletion)
    - Eager deletion uses resources, hence can impact on server performance
    - Better to let old chunks hang around for a while and reclaims all of them at the same time in a background operation (e.g. during the night, when servers are not too busy)

- Utilizing replicas
    - 3 replicas means 3 machines storing the data
    - When performing data transfer, fetch data from machine that is less utilized both in terms of CPU and I/O usage (another load balancing strategy)

Conclusion:

- Proprietary systems: design your system to match your workload …
- Google not good enough for general data center workloads.
- GFS is suitable for data-center for search-workload