

A case for RAID

RAID-1: Mirrored Disks

Mirrored disks are the most expensive option. Every write to a data disk is also a write to a check disk. Since these writes happen in parallel, it should be at least half as fast.

Main drawback: duplicating all disks means doubling the cost of the disks or using only halving storage capacity.

RAID-2: Hamming Code for ECC

If all data bits in a group are read or written together, there is no impact on performance. Reads of less than the group size require reading the whole group for error checking, and writes to a portion of the group require reading the whole group, modifying the data, and writing the full group, including check information.

1 parity disk can detect 1 error, but to correct an error we need enough check disks to identify the disk with the error. For 10 data disks, we need 4 check disks.

Good for supercomputers, bad for TPS.

RAID-3: Single Check Disk Per Group

Only 1 parity bit is needed to detect an error. Information on a failed disk can be reconstructed by calculating the parity of the remaining good disks and comparing bit by bit to the parity calculated for the original full group. If the check disk is the failure, read all data disks and store group parity in the replacement disk.

Drawback vs RAID-2: Soft errors can be corrected without rereading a sector. Also, the extra check information for each sector is not needed, increasing disk space.

Good for supercomputers but bad for TPS

RAID-4: Independent Reads/Writes

No longer spreads individual transfer information across several disks, keep each individual unit in a single disk. Allows parallelism for reads - the ability to do more than one IO per group at a time.

Parity calculation is much simpler: $\text{new parity} = (\text{old data} \oplus \text{new data}) \oplus \text{old parity}$. The check disk in a group must be read and written with every small write in that group, making it the bottleneck.

RAID-5: No single check disk

Distributes data and checks across all the disks - including the check disks. Because of this, RAID 5 can support multiple individual writes per group.

Best of both worlds: small read/writes perform close to the speed of a level 1 RAID and it keeps the large transfer performance and high storage capacity of RAID 3/4. RAID-5 is good for supercomputers or transaction processing.

The Google File System

GFS is a scalable distributed file system for large distributed data-intensive application. It has fault tolerance and delivers high aggregate performance to a large number of clients.

Assumptions

Component failures are common

Modest number of very large file

Reads are large streams or small & random

Large sequential writes that append data

Multiple clients concurrently append to the same file

High bandwidth is more important than low latency

New Operations

Snapshot creates a copy of a file or directory at low cost

Record append allows multiple clients to append to the same file and guarantees the atomicity of each append.

Architecture

A GFS cluster has one master and many chunkservers, and is accessed by many clients

Files are divided into 64MB chunks

Each chunk is replicated on multiple chunkservers

The master maintains all system metadata, chunk lease management, garbage collection, and chunk migration.

File data is not cached by the client or the chunkserver
Metadata is cached by the client

GFS does not have a per-directory data structure that lists all the files in that directory.

It does not support aliases for the same file (symbolic links)

GFS represents its namespace as a lookup table mapping full pathnames to metadata, stored in memory using prefix compression.

Master

Clients do not read and write files through the master, they ask the master which chunkservers it should contact, and interact with this chunkserver for subsequent operations

The master executes all namespace operations, manages chunk replicas, makes placement decisions, creates new chunks and replicas, coordinates system-wide activities to keep chunks fully replicated, to balance load, and to reclaim unused storage.

Metadata

Master stores 3 types of metadata: File/chunk namespaces, mapping from files to chunks, and locations of each chunk's replicas.

All metadata is kept in the master's memory, so master operations are fast.

Master does not keep a record of which chunkservers have a replica of a given chunk. It polls chunkservers for that information at startup. A chunkserver has the final word over what chunks it does or does not have on its own disks.

The operation log contains a historical record of critical metadata changes.

The operation log is the only persistent record of metadata and serves as a logical time line that defines the order of concurrent operations.

Files, chunks, and their versions are all uniquely identified by the logical times at which they were created.

The master recovers its state by replaying the operation log.

The master's state is checkpointed whenever the log grows too large, so it can recover by loading the latest checkpoint and replaying the log records after that.

Leases

The lease mechanism is designed to minimize management overhead at the master. Each mutation happens at all of the chunk's replicas.

Leases are used to maintain a consistent mutation order across replicas.

The global mutation order is defined first by the lease grant order chosen by the master, and then by a serial order that the machine granted the lease chooses.

Consistency model

GFS has a relaxed consistency model

File namespace mutations are atomic.

A file region is consistent if all clients will see the same data.

A file region is defined if it is consistent and all clients will see the last mutation in its entirety.

When a mutation succeeds without interference from other writers, the affected region is defined.

Concurrent successful mutations leave the region undefined but consistent.

A failed mutation makes the region inconsistent.

Mutations may be writes or record appends

A record append causes data to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing.

GFS may insert padding or record duplicates in between. These occupy inconsistent regions and are usually small.

After a sequence of successful mutations the file region is guaranteed to be defined and contain the data written by the last mutation

A chunk is lost irreversibly only if all of its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable rather than corrupted.

Atomic Record Appends

In a record append, the client specifies the data, and GFS appends it to the file at least once atomically at an offset of GFS's choosing and returns that offset to the client.

If a record append fails at any replica, the client retries the operation.

As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part.

GFS does not guarantee that all replicas are identical, just that the data is written at least once as an atomic unit.

Snapshot

The snapshot operation makes a copy of a file or directory tree almost instantaneously, using standard copy-on-write techniques.

Stale Replica detection

Chunk replicas become stale if a chunkserver fails and misses mutations to the chunk while it is down.

For each chunk, the master maintains a chunk version number to distinguish up-to-date and stale replicas

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas.

The master removes stale replicas in its garbage collection.

Before that, it considers stale replicas to not exist when it replies to client requests for chunk information.

Fault Tolerance

We cannot trust the machines, or the disks

Two strategies keep the system highly available: fast recovery and replication

Fast recovery: Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they were terminated.

Chunk replication: Each chunk is replicated on multiple chunkservers on different racks

Master replication: The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines.

Each chunkserver independently verifies the integrity of its own copy by maintaining checksums

Concurrency Control and Recovery

Concurrency control ensures users see consistent states of the database even though their operations may be interleaved by the database

Recovery ensures that the database is not corrupted as the result of a software, system, or media failure.

Transaction

A transaction is a unit of work that must commit or abort as a single atomic unit.

If it commits, all updates it performed are made permanent and visible to other transactions.

If it aborts, all of its updates are removed from the database and the database is as if the transaction had never executed.

ACID Properties

Atomicity: All operations of a transaction complete successfully, or none of them do.

Consistency: A transaction performed on a consistent database results in a consistent database.

Isolation: A transaction's behavior is not impacted by the presence of other concurrent transactions.

Durability: The effects of committed transactions survive failures.

Inconsistent Retrieval Problem

If a read happens during a transaction that temporarily leaves the database in an inconsistent state.

Serializability

The most widely accepted notion of correctness for concurrent execution of transactions. This is the property that a (possibly interleaved) execution of a group of transactions has the same effect on the database as some serial execution of those transactions.

Conflict serializability is based on a schedule (or partial ordering) of transactions.

This ordering is required to specify two types of dependencies: All operations for which an order is specified in a transaction must appear in that order, and the ordering of all conflicting operations must be specified.

Two operations conflict if they both operate on the same item and one of them is a write.

Two schedules are equivalent if they contain the same transactions and they order all conflicting operations in the same way.

A schedule is serializable iff it is equivalent to some serial schedule.

Recovery

Three types of failures must recover:

Transaction Failure: A transaction that cannot commit must rollback all of the update it made

System Failure: If the volatile memory contents are lost, all updates that have committed must be in the database, and all other updates must be removed from the database.

Media Failure: If data is lost on the non-volatile storage, then the database must be restored from an archival version and brought up to date using operation logs.

Logging

A log is a sequential file that stores records of transactions and the state of the system.

When a log record is created, it is assigned a Log Sequence Number.

When an update is made to a data item in the buffer, a log record is made for that update.

Checkpoints are taken periodically during normal operation to make recovery faster.

Physical Logging

Indicates location of modified data in the database.

If UNDO is allowed, then the value of the item prior to the update (Before image) is recorded.

If REDO is allowed, then the value of the item after the update (After image) is recorded.

Recovery actions are idempotent, they have the same effect no matter how many times they are applied.

Logical Logging

Records high-level information about operations that are performed.

REDO and UNDO must determine the set of actions required to perform the high-level operation.

Advantages: Minimizes amount of data written to the log

Disadvantage: Recovery based on logical logging is difficult to implement: Actions in a logged operation are not performed atomically.

Physiological Logging

Records are constrained to a single page, but are high-level on operations on that page.

Write Ahead Logging

All log records pertaining to an updated page are written to non-volatile storage before the page is allowed to be written to non-volatile storage

A transaction is not committed until all of its log records are written to stable storage.

Deadlocks

Deadlock avoidance: Imposes an order in which locks can be obtained, or aborts transactions that are blocked

Deadlock detection: Uses timeouts, or explicit checking.

Two phase locking

Two types of locks (Shared, and Exclusive)

A transaction is well-formed if it always holds an S or X lock on an item before reading it, and always holds an X lock before writing it.

Two phase locking forces all transactions to be well-formed and follow the rule: once a transaction has released a lock, it cannot obtain any additional locks.

Two phase locking is sufficient but not necessary to implement serializability.

Phantom problem

One transaction reads twice, another transaction adds a tuple that satisfies the query in between the reads.

The two read transactions will differ, which could not occur in a serial schedule

Isolation levels

Trades consistency for concurrency in a controlled manner.

Read uncommitted (Degree 0): No read locks obtained. Risk is seeing updates that will be rolled back, or seeing incomplete transactions.

Read committed (Degree 1): Holds read locks on individual items for short duration.

Risk is seeing non-repeatable reads. Two reads could see two different values.

Repeatable read (Degree 2): Holds read locks on individual items for long duration.

Risk is the phantom problem.

Serializable (Degree 3): Holds read locks on predicates for long durations.

Solves the phantom problem.

Hierarchical locking

Concurrent transactions can obtain locks at different granularities

A lock at a particular level locks all items underneath it.

Lock escalation: If a transaction obtains many locks on one level, it could be given a lock at the level above.

ARIES

An implementation of recovery.

Uses WAL, with STEAL/NOFORCE buffer management policy:

Pages on stable storage can be overwritten and data pages do not need to be forced to disk to commit a transaction.

Its REDO repeats history: It redoes updates for transactions that will eventually be undone. This allows it to employ physiological logging.

It uses page oriented REDO and logical UNDO.

First pass processes the log forward from the most recent checkpoint, determines dirty pages and active transactions.

Second pass repeats history by processing the log forward.

Third pass proceeds backwards from the end, removing the effects of all uncommitted transactions.

Uses fuzzy checkpoints that are extremely inexpensive.