

Chapter 1

Scalable Distributed Systems

HPC and mainframes.

- Pros: Single solution, powerful hardware, reliable design
- Cons: £££, failure models (single point of failure), vendor lock-in, adaptability, no incremental scalability

Distributed Computing. Want to achieve linear scalability, but synchronization, communication (**overheads**) prevent it. Ideally: perfect partition of data and compute (e.g. distributed client/server model on **cheap commodity hardware** (best £/resource); accounts for failures and manages the network speed bottleneck in software (parallel algorithms and systems).

Data centers Usually DC will have non-commodity network, but it will still be the slowest part in memory hierarchy. Focus on **scaling-out** not up by buying cheap hardware in bulk therefore achieve economy of scale. Elasticity: Resources on-demand (illusion of infinite resource).

1.1 Properties of DS

- Scalability: By aggregation of many resources
 - Compute
 - Storage: distributed file systems
 - Memory: cluster memory (in-memory key/value stores, caches)
 - Bandwidth: DC networks, CDN
- Location transparency from the user's perspective (black box API)
- High availability: Mask hardware and software failures
- Composition of independent services

1.1.1 Answering a google search request

1. Load-balancer routes request to lightly-loaded Google Web Server (GWS)
2. GWS routes search to one Index Server for each shard through load-balancer
3. Results are aggregated (IDs for matching documents), ie ordered by relevance
4. GWS sends appropriate IDs to Doc Servers to retrieve URL, title, summary
5. Results are aggregated (+ad, spell), producing search result page

1.2 Design principles

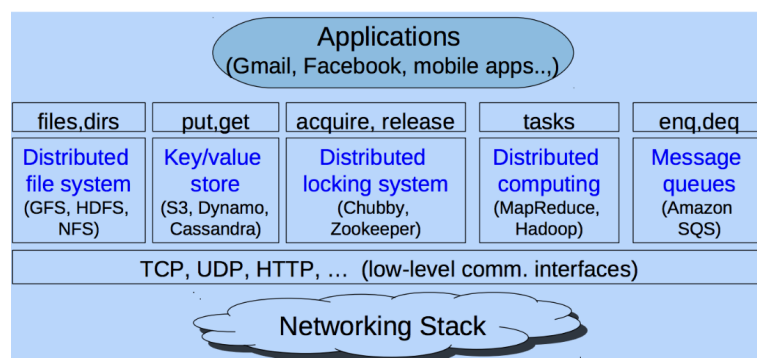
Types of scalable systems

1. Online systems (<100 ms) (OLTP)
2. Batch processing systems (>1 hours) (OLAP)
3. Nearline systems (<1 secs or mins)

Distributed Computing Challenges

- **Scalability:** Independent parallel processing of sub-requests/tasks. More servers permits serving more concurrent requests.
- **Fault tolerance:** Mask and recover from HW/SW failure. Replication data and service.
- **High availability**
- **Consistency:** */neq* services but = results.
- **Performance:** **Predictable** low-latency processing with high throughput

Abstractions : Each layer should itself be scalable, network-aware and fault-tolerant.



Design principles

- **Stateless services:** Avoid data inconsistency. Separation of data + meta-data. If state needed, use leases to expire state gracefully.
- **Caching:** Want low latency.
- **Partition/aggregation pattern** (see google search above)
- **Consistency models:** Need replication for reliability, availability and low latency access. Most apps use a mix of strongly consistent and inconsistent operations. Weaker better for low latency but harder to reason about. Decide what matters (e.g. order of posts in LinkedIn news feed? Access from multiple devices?).
- **Efficient failure recovery:** Failure is very common, but full redundancy too expensive → use failure recovery (and try to reduce the cost of failure recovery)
 - Replication: Need to replicate data and service (consistency issues)
 - Recomputation: Use stateless protocols, form data lineage for compute jobs

Chapter 2

Data Center and cloud

Cloud . Large pool of easily usable **virtualized** computing resources, development **platforms** and various **services** and application (metered service over a network).

- | | |
|---------------------------------------|---|
| + Speed – services provided on demand | – Dependency on network and internet connectivity |
| + Global scale and elasticity | |
| + Productivity | – Security and Privacy |
| + Performance and Security | – Cost of migration |
| + Customizability | – Cost and risk of vendor lock-in |

Types of cloud

- **Public:** Cloud vendors offer their computing resources over the Internet
- **Private:** infrastructure used exclusively by a single business. Services on private network
- **Hybrid:** data and applications are shared between the above. Easier for optimizing existing infrastructure, security and compliance

Cloud Service Models . google app engine

- **IaaS:** IT infrastructure – servers and VMs, storage, networks, firewall and security
- **PaaS:** Environment for applications – servers, storage, network, OS, middleware, databases
- **FaaS:** Set-up, capacity planning, and server management
- **SaaS:** Software over the Internet. Entirely managed by cloud provider (updates, patches)

Data centers Physical facility that enterprises use to house computing and infrastructure in a variety of networked formats.

- PCSS: Power, Cooling, Shelter, Security
- $TCO = CapEx + OpEx$ (capital + operations)
- Power Usage Effectiveness: $PUE = E_{used}/E_{given}$
- 3% electricity, 2% Greenhouse. Monthly costs = \$3,530,920
- Reduce costs: raise temp, reuse heat, choose location (arctic, dam), reduce conversion of E
- Energy consumption not proportional to load. Need Virtualisation + resource pooling to consolidate service on fewer servers. 6-15% utilisation to 30% (virtualised).

Improving Resource Utilisation

- Hyperscale system management software: DC as a warehouse-scale computer. Software managed pooled resources that include compute, network, and storage.
- Dynamic resource allocation: Virtualisation isn't enough, need to dynamically allocate CPU resources across servers and racks, allowing admins to quickly migrate resources to address the shifting demand (2x to 6x better).
- Networking: Software Defined Networking (SDN). Improve internal machine-to-machine communication with custom protocols.

Rack scale computing Made of Compute (& accelerators), Storage (hot / warm / cold disks), Networking (interconnect, SDN).

- evolution: physical aggregation (shared power, cooling, rack-management) → : fabric integration (fast rack-wide interconnect) → resource disaggregation (pooled compute, storage, memory resources)
- Heterogeneous Computing Resources across the Rack: Accelerators, Co-processors, Intelligent storage, Intelligent (active) memory, Smart NICs, In-network data processing

Chapter 3

Bigtable

Systems used Processing with Sawzall and MapReduce

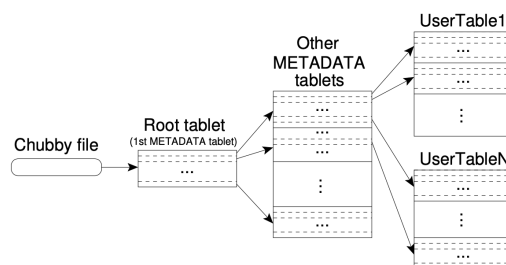
- Scheduler (Google WorkQueue): (failure detection, membership)?
- GFS: Master (metadata) & Chunk servers: (read and writing large chunks of data)
- Chubby: Coarse-grained locks with small amount of data in a lock. 5 nodes Paxos

Data model Distributed map: Atomic read or write in a single row key

- `<Row, Column, Timestamp>`, value is uninterpreted byte array
- `ColumnFamily:qualifier`: Same CF = Same AC & Same compression
- API: Lookup, insert, delete (+scan)
- Settings: In memory SSTs, data lexicographically sorted (locality control)

SSTable Immutable, sorted file of key-value pairs arranged in blocks. Index of block ranges for single disk seek. Can be shared by multiple tablets (split).

Tablets Non overlapping range of rows of a table (multiple SSTables). Unit of distribution & load balancing of each table. Client caches location of tablet. Tablet history found in METADATA table.



Server Each tablet lives at only one server. Master assigns multiple tablets per server, and monitors load and server fault (chubby lock). Tablet server splits tablets that get too big. Shared server-wide commit log & group commit.

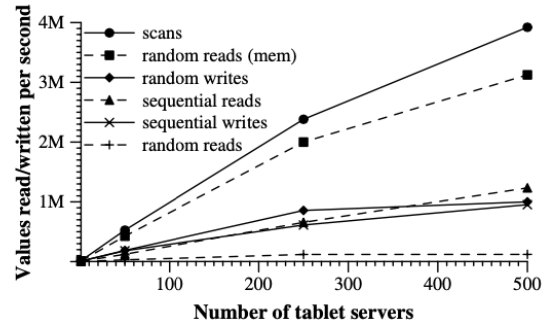
Editing/Reading a Table Writes go in GFS REDO commit log, then applied to (sorted) memtable (COW rows for concurrency). Reads applied to merged view of SSTables & memtable. R&W continue during tablet split or merge. RPC is checksummed and authorization happens on tablet server.

Compaction

- **Minor compaction:** Memtable into an SSTable (Reduce memory usage, Reduce log traffic on restart).
- **Major compaction:** Merging SSTables to one SSTable (No deletion records, only live data)
- **Merging compaction:** Reduce number of SSTables (+policy “keep only N versions”)

Locality groups . Group CFs into SSTables under shared compression scheme. Can use bloom filters to avoid fetching SSTables.

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843



Chapter 4

Dynamo

Chapter 5

Spanner: Globally distributed database

Spanner is a scalable, multi-version, globally-distributed, synchronously-replicated database.

External Consistency/Linearizability If transaction T1 commits before another transaction T2 starts, then T1's commit timestamp is smaller than T2's

5.1 Issues addressed

- Wanted strong consistency (which via CAP theorem meant less availability)
- Wanted ACID transactions
- Wanted schema
- Spanner is an example of NewSQL (SQL like model, but scalability and performance)
- Built for F1 (important app for Google). Was MySQL cluster, had big problems of resharding (done manually) and schema migration.
- Wanted easy geodistribution (coping with whole datacentre failure)
- Wanted to automate the process of replication

5.2 Main Ideas

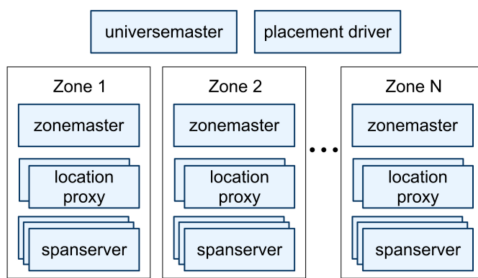
- TrueTime API
- General-purpose transactions (ACID). Support different transaction types which are optimised e.g. non- blocking reads
 - Write transactions guarantees 'external consistency' (strong form of consistency)
 - * Writes buffered.
 - * Write locks acquired at commit time (when Paxos prepare is done).
 - * But reducing availability for this (CAP) - block to wait out the uncertainty.
 - * Writes must initiate the Paxos protocol at the leader.
 - Reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date.
 - Read-Write - requires locks
 - Read-Only - lock free
 - * Requires declaration before start of transaction
 - * Reads information that is up-to-date

- Snapshot-Read - read information from past by specifying timestamp or bound
 - * Lock free (non-blocking)
 - * Globally consistent
 - * Use specific timestamp from past or timestamp bound so that data until that point will be read
- Atomic schema changes
- Temporal database. Transaction serialization via global timestamps
- Schematised, semi-relational (tabular) data model
 - SQL-like query interface
- Data is versioned. Each version is automatically timestamped at commit time - while locks are held
- Data chunks - directory/bucket
 - Unit of data movement and for defining replication properties
 - Split into fragments
 - Set of contiguous keys that share a common prefix.
 - Locality encoded in to it - pick keys to get better locality
- Shards data across many sets of Paxos groups
 - Want multiple groups to be able to partition data so you have smaller set of nodes to run Paxos on so its quicker.
 - Also, with multiple paxos groups they can have different ways of replication (nr replicas, location, etc).
- Layering of how transactions are executed.
 - If transaction can be done in one Paxos group then just run on that group.
 - When the transaction involves multiple paxos groups, then use the top layer (which uses strict two phase commit). This means users can do cross-row transactions
- Automatic resharding, rebalancing, failure response
- Globally distributed for high availability and geographic locality.

5.2.1 TrueTime API

- Global timestamps
- Exposes uncertainty in time and guarantees a bound on it
- Timestamp as a range
- Truetime timestamp getting is local, but these timestamps can be globally ordered (by reasoning about clock uncertainty/clock skew and waiting out the uncertainty to avoid overlapping timestamps)
- TrueTime has masters that globally synchronize and decide on the global bound.
- If network partition you lose the synchronization between the truetime masters. So must assume the clock skew increases over time, so over time the wait time will increase to where the system is very slow. (smaller clock uncertainty, larger throughput rate)
- Uses GPS and atomic clocks to get accurate time.

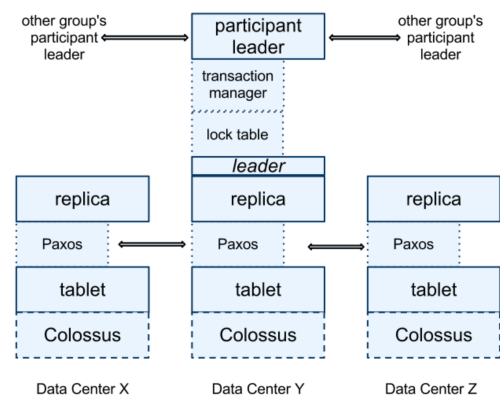
5.3 Additional points



- Universe = Spanner deployment
- Zone = unit of administrative deployment; unit of physical isolation
- Zonemaster = assigns data to spanservers
- Spanservers = serve data to clients
- Location proxies = used by clients to locate the spanservers assigned to serve their data.
- Universe master = console that displays status info about all zones for interactive debugging
- Placement driver = handles automated movement of data across zones
- Tablet: data structure. Bag of key-value mappings (key, timestamp) to string. A container that may encapsulate multiple partitions of the row space, hence its possible to colocate mutiple directories that are frequently accessed together. Tablets are stored on top of the Colossus distributed file system.
- Paxos state machine on each tablet.
- Long-lived leaders with time-based leader leases. If network partition happens, Paxos waits for 10 secs to get new leader.
- Lock table at each replica that is a leader. Contains state for 2-phase locking.
- Transaction manager = used to support distributed transactions
- Participant leader = built on top of transaction manager; used when transactions involve multiple Paxos groups.

As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

Shorter lease times would reduce the effect of server deaths on availability, but would require gretaeer amounts of lease-renewal network traffic.



Chapter 6

Zookeeper: Wait-free coordination for large scale systems

Coordination examples: Group membership • Leader election • Dynamic Configuration • Status monitoring • Queuing • Critical sections

What is Zookeeper: Highly available, scalable, distributed, configuration, consensus, group membership, leader election, naming, and coordination service.

6.1 Main contributions

- Coordination kernel – Wait-free coordination
- Coordination recipes – Build higher primitives
- Experience with coordination – Some application use ZooKeeper

6.2 Zookeeper properties

- Simplified file system type API
- No partial reads/writes
- Ordered updates and strong persistence guarantees
- Conditional updates (version)
- Watches for data changes
- Ephemeral nodes
- Generated file names
- No renames

6.3 Zookeeper guarantees

- Linearisable writes – Writes serialisable + respect precedence
- FIFO client order
 - Clients never detect old data
 - Clients get notified of change to watched data within bounded time
 - All requests from client processed in order
 - All results received by client consistent with results received by other clients

6.4 Zookeeper model

- Znode
 - In-memory data node
 - Hierarchical namespace
 - Manipulated through the Zookeeper API
 - Types: Regular/Ephemeral (can be automatically removed by system)
 - Flags: Sequential
 - Designed to store only meta-data or configuration, not general data storage
 - Can store information such as timestamps or version counters to track updates. Support conditional writes based on timestamp/version.
- Watch mechanism
 - Get notification upon update to data
 - One time triggers
- Client sessions
 - Session = connection to server from client
 - Timeout mechanism

Chapter 7

Map reduce

Data flow model. Share data through stable storage - replication, I/O, serialization costs.

Chapter 8

(Spark) Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing

8.1 Motivation

They want to leverage distributed memory to make it more efficient to reuse intermediate results across multiple computations. Iterative MapReduce runtimes etc perform data sharing implicitly for the pattern of computation they support, and do not provide a general that the user can employ to share data of their choice among operations of their choice. Wanted to allow big data analysis but with:

- More complex, multi-stage applications that reuse intermediate results (e.g. iterative ml, graph processing)
- More interactive ad-hoc queries (interactive data mining where a user runs many ad-hoc queries on the same data)

Hence needed efficient primitives for data sharing (in MapReduce the only way to share data across jobs is stable storage - slow due to replication and disk I/O, but necessary for fault tolerance). Goal: fault-tolerant and efficient distributed memory abstraction. In-memory data sharing.

With fine-grained updates to mutable state, one can only provide fault tolerance through replicating the data across machines or logging updates across machines. These are expensive for data-intensive workloads cause they require copying large amounts of data over the cluster network and they incur substantial storage overhead.

8.2 Resilient Distributed Datasets(RDDs)

- Distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
- Enables efficient data reuse. Lets users explicitly persist intermediate results in memory, control their partitioning to optimize data placement and manipulate them using a rich set of operators.
- Store data lineage instead of data.
- Recompute data based on lineage. Used in fault recovery - much quicker and less expensive than using replication. Only the lost partitions of an RDD need to be recomputed upon failure and can be recomputed in parallel on different nodes.
- Partitioned across nodes
- Immutable to simplify lineage tracking

- Can only be built (created, written) through coarse-grained, deterministic transformations (map, filter, join etc). This restricts RDDs to apps that perform bulk writes, but allows for more efficient fault-tolerance.
- Note that reads can be coarse- or fine-grained.
- Checkpointing to disk to avoid unbounded lineage
- Can efficiently express many parallel algos (these apply the same operator to many items). Unify many programming models and support new apps too.
- Best for batch workloads (coarse granularity, memory bandwidth levels of write throughput(so quite high))
- Straggler mitigation: Since RDDs are immutable the system can run backup copies of slow tasks (like in MapReduce). Hard to do with distributed shared memory cause two copies of a task would access the same memory locations and interfere with each others' updates.

8.3 Representing RDDs

- partitions
- dependencies on parent RDDs
- function for computing the dataset based on its parents
- metadata about its partitioning scheme - hash/range partitioned
- preferredLocations(p) - list nodes where partition p can be accessed faster due to data locality

Dependencies are narrow(each partition of parent is used by at most one partition of the child RDD) or wide(multiple partitions may depend on it). Narrow good for pipelined execution on one node and less recomputing needed upon recovery.

8.4 Spark programming interface

- DryadLINQ-like API in Scala
- Usable interactively from scala interpreter
- Provides:
 - RDDs
 - Operations on RDDs: transformations (build new RDDs), actions (compute and output results to app/storage system)
 - Control each RDDs partitioning (layout across nodes) and persistence (storage in RAM, on disk, replicating across machines etc.). Can co-partition (e.g. hash both on the joining field) RDDs that are e.g. repeatedly joined to avoid shuffles. Persistent RDDs can be storage in-memory as deserialized Java objects, in-memory as serialized data or on-disk.

8.5 Spark scheduler

In bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance. Creates DAG of stages to execute (narrow, wide dependencies, shuffle stages). Then launches tasks to compute missing partitions from each stage until it has computed the target RDD.

Bibliography