Scalable distributed systems design

Online systems (< 100 ms)
Batch processing systems (> 1 hours)
Nearline systems (< 1 secs or mins)
Scalability, Fault tolerance, High availability, Consistency, Performance
Distributed file system, Key/value store, Distributed locking system, Distributed computing, Message queues, Low-level comm. Interface

Cloud computing & Data centers

Public cloud, Private cloud, Hybrid cloud
IaaS: Virtualization, Servers, Storage, Networking
PaaS: …, Runtime, Middleware, O/S
Rate-1: Basic Site Infrastructure
Rate-2: Redundant Capacity Component Site Infrastructure -> Has generators
Rate-3: Concurrently Maintainable Site Infrastructure -> Dual power feed
Rate-4: Fault-Tolerant Site Infrastructure -> 2x generators, UPSs and HVAC
Power usage effectiveness (PUE) = Total amount of energy used / energy delivered to equipment
Total facility power = covers IT systems (servers, network, storage) + other equipment (cooling, UPS, switch gear, generators, PDUs, batteries, light
Challenge: cooling data centers, Energy Proportional Computing, Managing a Data Centre and its Resources, networking at scale, Fault Tolerance of Components Cloud computing & Data centers
Costs for Running a Data Centre: TCO = CapEx + OpEx
Rack-scale computer (pre-packaged)

- Compute: standard compute, accelerators
- Storage: hot / warm / cold disks
- Networking: interconnect, software defined networking

From server-centric to resource centric design

- Past: physical aggregation -> shared power, cooling, rack-management
- Now: fabric integration -> fast rack-wide interconnect
- Future goal: resource disaggregation

Future: Heterogeneous Computing Resources across the Rack

Bigtable

Building blocks: Scheduler, Google File System, Chubby Lock Service
Google File System: master, Chunk servers (replicated on 3 machines)
Chubby Lock Service: Paxos, Coarse grain locks
<row, column, timestamp> triple for key

- Each value is uninterpreted array of bytes
- Can be serialized and deserialized separately
- Ordering on concatenation of row keys, column keys, and timestamps

Arbitrary “columns” on a row-by-row basis

- Column family:qualifier
- Family is heavyweight, qualifier lightweight
- Column-oriented physical store - rows are sparse

Bigtable vs Relational

- read/write of data under a single row key is atomic
- Immutable data similar to versioning DBs (can keep last N versions or last N days)

Components - **SSTable**

- Immutable, sorted file of key-value pairs
- Index is of block ranges, not values
- Index loaded into memory
- Lookup is single disk seek
- structure: 64k block, 64k block, …, index

Components - **Tablet**

- Unit of distribution & load balancing
- Built out of multiple SSTables
- A range of rows, cannot overlap
- A tablet can point to the middle of a SSTable

Components - **Table**

- Multiple tablets make up table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap
- Structure: Multiple tablets

Finding tablet: Chubby file -> Root tablet (1st METADATA tablet) -> Other metadata tablets -> Tablets
Components - **Tablet server**

- manage tablets, multiple tablets per server
- Each tablet lives at only one server
- SSTable is fixed size; tablets will grow in sizes
- Server splits tablets that get too big

Components - **Master server**

- Use chubby to monitor health of tablet servers & restart failed servers
- GFS replicates data

Mutations committed to commit log (in GFS)

- Then applied to in-memory version (memtable)
- Each memtable row is copy-on-write

Reads applied to merged view of SSTables & memtable

- Reads/writes continue during tablet split or merge

Deletion is just recorded: Tombstone
Minor compaction: Moving memtable to a new SSTable
Merging compaction

- SSTables will overlap in their key ranges
- Prevent SSTable segmentation
- SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live

Major compaction: A merging compaction that rewrites all SSTables (of a given key) into exactly one SSTable
Locality groups: group column families together
Throughput

- Random reads: Contention
- Random writes: Better than random reads
- Sequential reads: Faster because of locality
- Scans: No need to index exactly

Aggregate rate: Scans > random reads (mem) > random writes ≈ sequential read ≈ sequential writes > random reads

Dynamo

Availability, resolving update conflicts, Incremental scalability, symmetry, decentralization, heterogeneity
Put(key, context, object)
Get(key)
CAP: Consistency, availability and partition-tolerance
Eventual consistency: sacrifice strong consistency for availability
High Availability for writes
Sloppy Quorum and hinted handoff
Anti-entropy using Merkle trees
Gossip-based membership protocol and failure detection
Distributed hash table (DHT): Lookup(key) -> IP address
If my-id < my-successor-id < key-id: Pass down to successor
Data replicated on N hosts
Preference list: Longer than N to allow for failures
R W quorum system: R + W > N
Sloppy quorum: Can be equal or less than N
Every storage node: Request coordination, Membership & failure detection, Local persistent storage

Spanner

General-purpose transactions (ACID)
rows must have names
External Consistency: Commit order respects global wall-time order
Version management: Strict 2PL
Use read locks on all data items that are read

- Read latest version, not based on timestamp

Writes buffered, and acquire write locks at commit time (when prepare is done)

- Timestamp assigned at commit time

TrueTime: Marzullo’s algorithm
TT.now() return TTinterval: [earliest; latest]
TT.after(t) return true if t has definitely passed
TT.before(t) return true if t has definitely not arrived
earliest < TT.now() < latest, latest - earliest = 2*ϵ
(time at pick s = TT.now().latest) - (time at wait until TT.s now().earliest > s) = Commit wait
System Architecture

- Top level: Universemaster, Placement drive
- Zones: (Zone 1: Zonemaster, Location proxy, Spanservers), …

Participant leaders: Transaction manager, Lock table
Leader replicas: Paxos <-> Paxos
Tablet: Colossus
Data Chunks: Directory, smallest unit of data placement & defining replication properties
Can change schema easily: Non-blocking
Block Just before the timestamp in the future
Use transaction to update schema information
Movedir: registers that it is starting to move data & moves in background, then uses a transaction to atomically move that nominal amount & update metadata
Non-leader & leader-soft kill: doesn't affect availability
Latency: all reads < single-site commit < multi-site commit

MapReduce

map(k1,v1) -> list(k2,v2)
reduce(k2, list(v2)) -> list (v3)
Map: Processes input data & generates (key, value) pairs
Shuffle: Distributes intermediate pairs to reduce tasks
Reduce: Aggregates all values associated to each key
Intermediate results persisted to local disks
Combiner function: Partial reduce on local
Failure Recovery: restart task
Speculative Execution: attempts to locate slow tasks (stragglers) and run redundant (speculative) tasks
If task’s progress score less than (average – 0.2) & task ran for at least 1 minute -> mark as straggler
Support for iteration: Loop unrolling

Resilient Distributed Datasets

Goal: In-Memory Data Sharing
Partitioned across nodes
Immutable to simplify lineage tracking
Can only be built through coarse-grained deterministic transformations (map, filter, join, …)
Checkpointing to disk to avoid unbounded lineage
RDD Recovery: Simply redo the computation
Generality of RDDs: Data flow models, iterative models
Coarse updates, High write throughput (near memory bandwidth), Best for batch workloads
Transformations: Output RDD
map((f:T)->U), filter((f:T)->Bool), groupByKey(), reduceByKey(f:(V,V)->V), …
Actions: Don’t output RDD
collect(), reduce(f:(T,T)->T), save(path : String), …
Spark architecture: Master, workers (Block, Msgs)
Co-partitioning: Partition on the same way
Performance: Hadoop > Basic Spark > Spark (co-partitioning)

DB Storage
DBMS: Query optimization & execution -> Relational operators -> Files & access methods -> Buffer management -> Disk space management -> Storage
Disk Structure: Disk head, Sector, Tracks, Platters
Time to access a disk = Seek time + Rotational delay + Transfer time
Seek time is constant when seek distance < D
Key to lower I/O cost: reduce seek/rotation delays
"Next" block concept: blocks on same track, followed by blocks on same cylinder, followed by blocks on adjacent cylinder
Improve locality:
Non-random access (scan, index traversal)
Random access (hash join): partition to fit in TLB
Compress data: Read faster but need to decompress
Cache Sensitive Search Tree: Fit node into a L2 cache line
Nodes are numbered & stored level by level, left to right
CSB+ tree: Children of the same node stored in an array (node group)
CSB+ tree with segments: Divide child array into segments
Full CSB+ tree: CSB+ tree with pre-allocated children array
Execution time:
Search: CSS < full CSB+ ≈ CSB+ < CSB+ seg < B+
Insertion: B+ ≈ full CSB+ < CSB+ seg < CSB+ < CSS
Cache Conscious Join Method: Radix join

Solid State Storage and Databases
Flash Disks Structure: SSD -> Flash controller -> Flash packages -> Dies -> Planes -> Blocks -> Pages
Access time depends on: Device organization (internal parallelism), Software efficiency (driver), Bandwidth of flash packages
Flash Translation Layer (FTL)
- Complex device driver (firmware)
- Tunes performance and device lifetime
- The FTL performs logical-to-physical address translation, garbage collection, wear-leveling, error correction code (ECC), and bad block management

Flash: most developed, Not enough density, Problematic bulk erase size, Good access time, Bad endurance
PCM: promising competitor, Nonvolatile, Too low density, good access time (comparable to RAM), Ok endurance
HDD optimizations:
- Data structures: B-trees, bitmap indexes, column organization, compression
- Query plans (prefer sequential vs random access)
- Buffer pool, buffering policies, Write-ahead logging
- Column stores

OLTP: Many writes
OLAP: No writes, Large sequential reads
Flash-only OLTP: smaller random-to-sequential gap
Append/pack: hot dataset & cold dataset
- Updates: just mark changed page as invalid and add a new entry in hot for it
- Reclaim: reclaim space in hot when it runs out of space by moving a range old pages to cold
- Gives consistently high throughput

Flash-aided Business Intelligence (OLAP)
Freshness: in-place updates -> Query w/ updates
Performance: batch updates -> Query only + updates only
Buffer updates on Flash instead of memory: flash has larger capacity and smaller price
Materialized Sort-Merge (MaSM)
merge(Table range scan for disk, merge(Mem scan for memory, Run scan for SSD))
Logging on Flash+HDD: database -> Interface -> In-memory log buffer -> Request queue -> Workers -> Disks

Main Memory DB
OLAP (Business Intelligence): Massive amounts of data, Complex queries, Large number of tables, Long running but still somewhat interactive, Mostly read only
OLTP: Really only transactions, i.e., updates, Few tables touched, Typically generated queries
Pie chart: Latching 24%, Locking 24%, Buffer pool 24%, Recovery 24%, Useful work 4%
Solution choices:
OldSQL: Mediocre performance on New TP
NoSQL: Give up SQL and ACID for performance
NewSQL: Preserve SQL and ACID
Solve locking: timestamp order, MVCC
Solve buffer pool: Main memory
Solve latching: innovative use of B-trees, single-threading
Solve logging: built-in replication and failover
Locking: Concurrent access to tuples in the database
Latching: Concurrent access to data structure
VoltDB: Main-memory storage, Single threaded (run transaction to completion), No locking/latching, no log, use redundancy for durability
Pie chart: Locking: 5%, Useful work: 95%
OldSQL for New OLTP: Too slow, Does not scale
NoSQL for New OLTP: Lacks consistency guarantees, Low-level interface
NewSQL for New OLTP: Fast, scalable and consistent, Supports SQL
Partitions: One partition per physical CPU core
types of tables: Partitioned, Replicated
types of work: Single-Partition, Multi-Partition (has locking)
VoltDB partition structure: Work Queue, execution engine, Table Data, Index Data
single-threaded comes at a price: other transactions wait
VoltDB is built for throughput over latency
Schema Changes: modify schema and stored procedures -> build catalog -> deploy catalog

Graph databases
Pros: powerful data model (as general as RDBMS), connected data locally indexed, easy to query, scales up well
Cons: sharding
good for: Recommendations, Geospatial, Web of things, …
Translating to Neo4j
- Each entity table is represented by a label on nodes
- Each row in an entity table is a node
- Columns on those tables become node properties
- Add unique constraints for business primary keys, add indexes for frequent lookup attributes

Example code
- START a=node(*)MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
- WHERE NOT((hugo)-[:ACTED_IN]->(movie))
- SET movie.tagline = "We bury our sins here, Dave."
- RETURN a.name, d.name, count(*) AS count
- ORDER BY(count) DESC LIMIT 5;

Aggregation: count(x), min(x), max(x), avg(x), collect(x)
SET, CREATE UNIQUE, DELETE

DNA Storage
DNA: dense, durable, scarcity of silicon supply, parallelism
Complementarity of A, T, C & G: A <-> T, C <-> G
Challenges: Biological constraints, Error-prone synthesis & sequencing
Reality: Identifier, Error Correction Codes, Value
Exploit chemical processes:
- Annealing of complementary nucleotides
- Polymerase chain reaction (PCR) to replicate/amplify DNA sequences
- Loop-mediated isothermal amplification

Purposes:
- Content detection
- Content retrieval through amplification
- Solving combinatorial problems

Writing Data to DNA (1): The Unstructured Way
Dump database to a binary archive file and encode
Limitations:
- $\log_4$ (#segments) nucleotides reserved for offset
- No point queries supported
- Cannot perform near-molecule data processing

Writing Data to DNA (2): Structured Data Layout
- NSM: one row per oligo (use unique primary key)
- DSM: columnset partitioning for "large" rows
- Reduces overhead from $\log_4$(#segments) to $\log_4$(cardinality) (#segs. >> Card.)

Data Cleaning: Sequencing -> Clustering & reassembly -> Decode
Near-molecule Query Processing
Polymerase Chain Reaction (PCR): amplify oligo many times
Encoding: Table & Column ID, Primary Key, Error Correction Codes, Value
Sequence using nanopore sequencing (Oxford Nanopore)
Near-molecule Query Processing: Join
Complementarity – matching base pairs
Gel electrophoresis after PCR
Nanopore sequencing to retrieve resulting annealed oligos

Cold Storage
Hot data: SSD -> Provisioned for peak, High throughput, Low latency, High cost
Warm data: Fast disks -> High density, Low hardware cost, Low operating cost, Latency lower than tape
Cold data: Slow disks, SAN or Tape -> Low cos, High latency
Pelican vs. tape: Better performance, similar cost
Provision resource: disks, power, cooling, bandwidth
Benefits of removing unnecessary resources: High density of storage, Low hardware cost, Low operating cost (capped performance)
CSD Rack: Disk groups (Disks), CSD controller
Pelican - reorder requests in order to minimize the impact of spin up latency
- Reordering is done to amortize the group spin up latency over the set of operations.
- Power Up in Standby is used to ensure disks do not spin without the Pelican storage stack managing the constraints.
- Initialization is done on the group level.
- If there are no user requests, then all 4 groups are concurrently initialized.

Performance:
- near average FP throughput at high workload rate
- Time to first byte takes more time than FP

Pros: reduced cost & power consumption
Cons: Tight constraints – less flexible to changes
CSD: average execution time increases linearly as number of clients or group switch latency increases
HDD: constant
Common batch processes on cold data: Massive-scale Group-by / Join, [Near]-duplicate detection, Data Localization, In-place Map-Reduce
Buff-Pack: Greedy approach
Flush into the current disk group to avoid switching
Off-Pack: Write data to the wrong disk group, then transfer
T_total = T_switch + T_seek + T_read + T_write
Off-Pack is better for: a smaller buffer, a higher # disk groups, a higher throughput
Execution time against disk group switch latency or number of disk groups
Execution time against buffer size or data size
Baseline > BuffPack > OffPack > HDD Rack (No Switch)
- All decrease when buffer size increases

Multicores
OLTP: Throughput peaks first, then goes down again because of contentions
OLAP: Goes up and plateau because of memory bandwidth
data intensive applications: 50%-80% of cycles are stalls
- Problem: instruction fetch & long-latency data misses
- Instructions need more capacity
- Data misses are compulsory

Minimizing Memory Stalls
- Prefetching: light, temporal stream, software-guided
- Cache conscious: code optimizations, alternative data structures/layout, vectorized execution
- Exploiting common instructions: computation spreading

Row store: Good for OLTP, Accessing many columns
Column store: Good for analytical queries (OLAP), Accessing a few columns
Index tree in memory:
- Lookup-heavy workload: Store in preorder
- Scan-heavy workload: Store in level order

Volcano Iterator: poor data & instruction cache locality
Vectorized Execution: good data & instruction cache locality, allows exploiting SIMD
Hot instruction: reused frequently
Computation spreading - SLICC: Continue to execute on the adjacent core every cycle
L1-I misses: minimized footprint & maximizing re-use
LLC data misses: maximize cache-line utilization through cache-conscious algorithms and layout
Modern Parallelism: Instruction & data parallelism, Multithreading, Horizontal parallelism
Critical section types: Unbounded (e.g. Locking, latching), Fixed (e.g. Transaction manager), Cooperative (e.g. Logging)
Improvement: move on without releasing & acquiring locks
Unscalable components: data Access in Centralized B-tree
Physiological Partitioning (PLP): assign branches to threads
Serial Log delays: serialize at the log head, I/O delay to harden the commit record, serialize on incompatible lock
Aether Holistic Logging: early lock release, flush pipelining, consolidation array
shared-everything: stable, not optimal
Island shared-nothing: robust, middle ground
shared-nothing: fast, sensitive to workload

XML & RDBMS'
<!ELEMENT note(to,from,heading,body,message +)>
Structure-Mapping approach & Model-Mapping approach
Edge: All the edges of XML document are stored in a single table.
- Edge(Source, Target, Label, Flag, Value)

Monet: It Partitions the edge table according to all possible label paths.
- Element Node (Source, Target, Ordinal)
- Text Node (ID, Value)

XParent: Based on LabelPath, DataPath, Element and Data.
- LabelPath (ID, Len, Path)
- DataPath (Pid, Cid)
- Element (pathID, Ordinal, Did)
- Data (PathID, Did, Ordinal, Value)

XRel: XML data stored based on Path, Element, Text, and Attribute.
- Path (PathID, Pathexp)
- Element (PathID, Start, End, Ordinal)
- Text (PathID, Start, End, Value)
- Attribute (PathID, Start, End, Value)

XRel and XParent outperform Edge in complex queries.
Edge performs better when using simple queries.
Label-paths help in reducing querying time

Document Databases
BSON: Lightweight, Traversable, Efficient (decoding and encoding)
MongoDB: Document, Collection, PK: _id Field, Uniformity not Required, Index, Embedded Structure
SQL: Tuple, Table/View, PK: Any Attribute(s), Uniform Relation Schema, Index, Joins
Shell: db, show dbs, use <name>, show collections
db.<collection>.find({<field1>:<value1>,<field2>:<value2>})
db.<collection>.find({ $or:[<field1>:<value1>,<field2>:<value2>]})
db.<collection>.find({<field>: {$in [<value1>, <value2>]}})
db.<collection>.find({<field>:<value>}, {<field2>: 1})
db.<collection>.find({<field>: { $exists: true}})
db.<collection>.update({<field1>:<value1>}, {$set: {<field2>:<value2>}}, {multi:true})
db.<collection>.update({<field>:<value>}, { $unset:{<field>:1}})
db.<collection>.remove({<field>:<value>})
Embedding
- easy for the server to handle
- Embed when the "many" objects always appear with (viewed in the context of) their parents.
- De-normalization provides Data locality, and Data locality provides speed
- Need to update in multiple places

Linking
- Linking when you need more flexibility
- Link by id as a foreign key

```
db.users.ensureIndex( { score: 1 } )
db.users.getIndexes()
db.users.dropIndex( {score: 1} )
db.zips.aggregate(
    {$match: {state: "TN"}},
    {$group: {_id: "TN", pop: {$sum: "$pop"}}}
);
```
$limit, $skip, $sort
distinct()