Key Value / NOSQL

- The key value abstraction
  - key to value
- Dictionary data structure
  - distributed
- Database
  - MySQL
  - Data stored in tables
  - Schema-based, i.e., structured tables
  - Foreign keys refer to the primary key a different table
  - Queries
- Needs of today workloads
  - speed
  - avoid single point of failure
  - Low TCO (total cost of operation)
  - fewer system administrators
  - incremental scalability
  - scale out, not up
    - scale up = grow your cluster capacity by replacing with more powerful machines
      - traditional approach
      - not cost-effective, above the sweet spot
    - scale out = incrementally grow your cluster capacity by adding more COTS machines (Components off the shelf)
      - cheaper
      - over a long duration, pages in a few newer (fast) machines as pup phase out a few older machines
      - used by most companies who run datacenter and clouds today
- Key-value/NoSQL Data Model
  - NoSQL = “Not Only SQL”
  - Necessary API operations: get(key) and put(key,value)
    - get returns value; put updates value if already exists
    - and some extended operations, e.g., “CQL” in Cassandra key-value store
  - Tables
    - “Column families” in cassandra, “Table” in HBase, “Collection” in MongoDB
    - may be unstructured: may not have schemas
      - some columns may be missing from some rows
    - don’t always support joins or have foreign keys
    - can have index tables, just like RDBMs
  - Tables
    - unstructured
    - no schema imposed
    - columns missing from some rows
    - no foreign keys and joins not supported
  - Columns-Oriented Storage
    - NoSQL systems often use column-oriented storage
      - RDB<Ss store an entire row together
      - NoSQL systems typically store a column together
        - Entries within a columns are indexed and easy to locate, given a key (and vice-versa

        - Why useful?
            - range searches within a column are fast since you don’t need to fetch the entire database

- **Cassandra** (client -> coordinator -> replicas for the key (ring))
    - distributed key-value store
    - intended to run in a datacenter
    - designed at Facebook
    - ope-sourced later, today an Apache project
- How do you know which server stores the key
    - Cassandra uses a ring-based DHT but without finger tables or routing
        - Key-server mapping is called the partitioner used by the coordinator
- Data placement strategies
    - replication strategy: two options:
        - simple strategy - uses the partitioner of which there are two kinds
            - RandomPartitioner: Chord-like hash partitioning
            - ByteOrderedPartitioner: Assigns ranges of keys to servers
                - Easier for range queries (e.g. Get me all twitter users starting with [a-b])
        - network topology strategy: for multi-DC deployments
            - Two replicas per DC
            - Three replicas per DC
            - Per DC
                - First replica placed according to Partitioner
                - Then go clockwise around ring until you hit a different rack
- Snitches
    - Maps: IPs to racks and DCs. Configured in cassandra
    - Some options:
        - SimpleSnitch: Unaware of Topology
        - RackInferring: Assumes topology of network by octet of server’s IP address = x dc rack node (octets)
        - PropertyFileSnitch: uses a config file
        - EC2Snitch: uses EC2
            - EC2 Region = DC
            - Availability zone = rack
- Writes
    - need to be lock-free and fast
    - client sends write to one coordinator node in Cassandra cluster
        - Coordinator may be per-key, or per-client, or per-query
        - Per-key Coordinator ensures writes for the key are serialized
    - Coordinator uses Partitioner to send query to all replica nodes responsible for key
    - When X replicas respond, coordinator returns an acknowledgement to the client
    - Always writable: Hinted Handoff mechanism
        - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up
    - One ring per datacenter
        - per DC coordinator elected to coordinate with other DCs
        - Election done via Zookeeper, which runs a Paxos (consensus) variant
            - Paxos: elsewhere in this course
- Writes a replica Node

    - on receiving a write
        - log it in disk commit log (for failure recovery)
        - make changes to appropriate memtables
            - memtable = in memory representation of multiple key-value pairs
            - cache that can be searched by key
            - write-back cache as opposed to write-through
        - later, when memtable is full or old, flush to disk
            - data file: an ss table (sorted string table) - list of key-value pairs, sorted by key
            - index file: an ss table of (key,position in data ss table) pairs
            - and a bloom filter (for efficient search) - next slide
- Bloom filter
    - compact way of representing a set of items
    - checking for existence in set is cheap
    - some probability of false positives: an item not in set may check true as being set
    - never false negatives
    - large bit map initially all the bits are set to zero
    - key k
        - use a small set of hash functions to hash the key k
        - each has returns a number between the inclusive bit map
    - on insert, set all based bits
    - on check-if-present return true if all hashed bits set
        - false positives
    - false positive rate low
        - k = 4 hash functions
        - 100 items
        - 3200 bits
        - FP rate - .02%
- Compaction
    - data updates accumulate over time and ss_tables and logs need to be compacted
        - the process of compaction merges ss_tables i.e., be merging updates for a key
        - run periodically and locally at each server
- Deletes
    - don't delete item right away
    - add a tombstone to the log
    - when compaction encounters tombstone deletes item
- Reads: Similar to writes
    - Coordinator can contact X replicas (e.g., in same rack)
        - Coordinator sends read to replicas that have responded quickest in past
        - When X replicas respond, coordinator returns the latest-timestamped value from among those X
    - Coordinator also fetches value from other replicas
        - checks consistency in the background, initiating a read repair if any two values are different
        - This mechanism seeks to eventually bring all replicas up to date
    - a row may be split across multiple ss_tables => reads need to touch multiple ss_tables => reads slower than writes
- Membership
    - any server in cluster could be the coordinator
    - every server needs to maintain a list of all the other servers that are currently in the server

    - list needs to be updated automatically as servers join, leave, and fail
- Cluster membership - gossip-style
- Suspicion mechanisms in cassandra
    - suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
    - accrual detector: failure detector output a value (PHI) representing suspicion
    - apps set an appropriate threshold
    - PHI calculation for a member
        - Inter-arrival times for gossip messages
        - PHI(t) =
            - -log(CDF or probability(t_now - t_last)) / log 10
        - PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
    - In practice, PHI = 5 => 10-15 sec detection time
- Cassandra vs RDBMs
    - MySQL comparison
        - > 50GB data
        - writes 300 ms avg (.12 Cassandra)
        - reads 350 ms avg (15 ms)

The Mystery of X-The Cap Theorem
- Cap Theorem
    - In a distributed system you can satisfy at most 2 out of the 3 guarantees:
        - 1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
        - 2. **Availability**: the system allows operations all the time, and operations return quickly
        - 3. **Partition-tolerance:** the system continues to work in spite of network partitions
- Why is availability important
    - availability = reads/writes complete reliably and quickly
    - measurements have shown that a 500ms increase in latency for operations at amazon.com or at google.com can cause a 20% drop in revenue.
    - at amazon, each added millisecond of latency implies a $6M yearly loss
    - SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients
- Why is Consistency important
    - consistency = all nodes see same data at any time, or reads return latest written value by any client
    - when you access your bank or investment account via multiple clients, you want the updates done form one client to be visible to other clients
    - When thousands of customers are looking to book a flight, all updates from any client should be accessible by other clients
- Why is Partition-Tolerance Important
    - Partitions can happen across datacenter when the Internet gets disconnected
        - internet router outages
        - under-sea cables cut
        - DNS not working
- Cap Theorem Fallout
    - Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

    - Cassandra
        - Eventual (weak) consistency, Availability, Partition-tolerance
    - Traditional RDBMss
        - Strong consistency over availability under a partition
- CAP Tradeoff
    - Starting point for NoSQL Revolution
    - A distributed storage system can achieve at most two of C, A, and P.
    - When partition-tolerance is important, you have to choose between consistency and availability
- Eventual Consistency
    - If all writes stop (to a key), then all its values (replicas) will converge eventually
    - If writes continue, then system always tries to keep converging
        - moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up
    - may still return stale values to clients
    - but works well when there a few periods of low writes - system converges quickly
- RDBMS vs Key-value stores
    - provide ACID
        - Atomicity
        - Consistency
        - Isolation
        - Durability
- Key-value stores like Cassandra provide BASE
    - Basically Available Soft-state Eventual Consistency
    - Prefers Availability over Consistency
- Mystery of X
    - Cassandra has consistency levels
    - Client is allowed to choose a consistency level for each operation (read/write)
        - ANY: any sever (may not be replica)
            - Fastest: coordinator caches write and replies quickly to client
        - ALL: all replicas
            - Ensures strong consistency, but slowest
        - ONE: at least one replica
            - Faster than ALL but cannot tolerate a failure
        - QUORUM: quorum across all replicas in all datacenter
            - What?
- QUORUM
    - majority
        - > 50 %
    - Any two quorums intersect
        - Client 1 does a write in red quorum
        - Client 2 does read in blue quorum
    - At least one server in blue quorum returns latest write
    - Quorums faster than ALL, but still ensure strong consistency
- Quorums in Detail
    - Several key-value/NoSQL stores use quorums
- Reads
    - Client specifies value of R
    - R = read consistency level

    - Coordinator waits for R replicas to respond before sending result to client
    - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.
- Writes
    - Client specifies W (<= N)
    - W = write consistency level
    - Client writes new value to W replicas and returns (Two flavors)
        - Coordinator blocks until quorum is reached
        - Asynchronous: Just write and return
    - R = read replica count, W = write replica count
    - Two necessary conditions
        - W + R > N
        - W > N/2
    - Select values based on application
        - (W=1,R=1): very few writes and reads
        - (W=N, R=1): great for read-heavy workloads
        - (W=N/2 + 1, R = N/2 + 1); great for write-heavy workloads
        - (W=1, R = N): great for write heavy workloads with mostly one client writing per key
- Other consistency levels
    - QUORUM: quorum across all replicas in all datacenter
        - global consistency, but still fast
    - LOCAL_QUORUM: quorum in coordinator's DC
        - Faster: only waits for quorum in first DC client contacts
    - EACH_QUORUM: quorum in every DC
        - Lets each DC do its own quorum: supports hierarchical replies
- Types of consistency
    - Cassandra offers Eventual Consistency

The Consistency Spectrum

- Stronger consistency models slower reads and writes
- Eventual consistency
    - if writes to a key stop, all replicas of key will converge
    - originally from amazon's dynamo and linden's Voldemort systems
- New Consistency models
    - striving towards strong consistency
    - while still trying to maintain high availability and partition-tolerance
    - Per-key sequential: Per key, all operations have a global order
    - CRDTs (Commutative Replicated Data types): Data structures for which commutated writes give same result
        - Effectively, servers don't need to worry about consistency
- Red-blue consistency: rewrite client transactions to separate ops into red ops vs blue ops
    - blue ops can be executed (commutated) in any order across DC's
    - red ops need to be executed in the same order at each DC
- Causal consistency: reads must respect partial order based on information flow
- Strong Consistency Models
    - Linearizability: Each operation by a client is visible (or available) instantaneously to all other clients
        - Instantaneously in real time
    - Sequential Consistency

        - After the fact, find a “reasonable” ordering of the operations (can re-oder operations) that obeys sanity (consistency) at all clients, and across clients
        - “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
    - Transaction ACID properties e.g. newer key-value/NoSQL stores
        - Hyperdex
        - Spanner
        - Transaction chains

HBase

- BigTables was first “blob-based” storage system
- HBase
    - Get/Put(row)
    - Scan(row range, filter) - range queries
    - MultiPut
    - prefers consistency (over availability)
- HBase Architecture
    - Client
    - Zookeeper
        - small group of servers running Zab, a consensus protocol
- HBase Table
    - Split it into multiple regions:L replicated across severs
        - ColumnFamily = subset of columns with similar query patterns
        - One store per combination of ColumnFamily + region
            - Memstore for each Store: in-memory updates to Store; flushed to disk when full
                - StoreFiles for each store for each region where the data lives
                    - HFile
- HFile
    - SS_table from googles big table
- Strong Consistency: HBase Write-Ahead log
    - Write to HLog before writing to MemStore Helps recover from failure by replaying Hlog
- Log Replay
    - After recovery from failure, or upon bootup (HRegionServer/HMaster)
        - Replay any stale logs(use timestamps to find out where the database is w.r.t. the logs)
        - Replay: add edits to the MemStore
- Cross-Datacenter replication
    - Single “Master” cluster
    - Other “Slave” clusters replicate the same tables
    - Master cluster synchronously sends HLogs over to slave clusters
    - Coordination among clusters is via Zookeeper
    - Zookeeper can be used like a file system to store control information
- Summary
    - Traditional Databases (RDBMs) work with strong consistency, and offer ACID
    - Modern workloads don’t need such strong guarantees, but do need fast response times (availability)
    - Unfortunately, CAP theorem
    - Key-value/NoSQL systems offer BASE
        - Eventual consistency, and a variety of other consistency models striving towards strong consistency

  - Design of Cassandra and HBase

Time and Ordering

Introduction to Basics

- Why Synchronization
  - Time synchronization is required for both
    - Correctness and Fairness
- Synchronization In the Cloud
- Why is it Challenging
  - End hosts in Internet-based systems (like clouds)
    - each have their own clocks
    - unlike processors (CPUs) within one server or workstation which share a system clock
  - Processes in Internet-based systems follow an asynchronous system model
    - No bounds on
      - message delays
      - processing delays
    - Unlike multi-processor (or parallel) systems which follow a synchronous system model
- Some definitions
  - An asynchronous distributed system consists of a number of processes
  - Each process has a state (values of variables)
  - Each process takes actions to change its state, which may be an instruction or a communication action (send, receive)
  - An event is the occurrence of an action
  - Each process has a local clock - event within a process can be assigned timestamps, and thus ordered linearly
  - But - in a distributed system, we also need to know the time order of events across different processes
- Clock Skew vs. Clock Drift
  - each process (running at some end host) has its own clock
  - when comparing two clocks at two processes
    - clock skew = relative difference in clock values of two processes
      - like distance between two vehicles on a road
    - clock drift = relative difference in clock frequencies (rates) of two processes
      - like difference in speeds of two vehicles on the road
  - A non-zero clock skew implies clocks are not synchronized
  - A non-zero clock drift causes skew to increase (eventually)
- How often Synchronize?
  - Maximum Drift rate (MDR) of a clock
  - Absolute MDR is defined relative to Coordinated Universal Time (UTC). UTC is the “correct” time at any point of time
    - MDR of a process depends on the environment
  - Max drift rate between two clocks with similar MDR is 2 * MDR
  - Given a maximum acceptable skew M between any pair of clocks, need to synchronize at least once every: M / (2 * MDR) time units
    - Since time = distance / speed
- External vs internal synchronization
  - Consider a group of processes
  - External synchronization

        - each process C(i)'s clock is within a bound D of a well known clock S external to the group
        - |C(i) - S| < D at all times
        - External clock may be connected to UTC or an atomic clock
        - Cristian's algorithm, NTP
    - Internal Synchronization
        - Every pair of processes in group have clocks within bound D
        - |C(i) - C(j)| < D at all times and for all processes i,j
        - Berkeley algorithm
- External vs Internal Synchronization
    - External Synchronization with D => Internal Synchronization with 2 * D
    - Internal Synchronization does not imply External Synchronization
        - In fact, the entire system may drift away from the external clock S

Cristian's Algorithm
- Basics
    - External time synchronization
    - All processes P synchronize with a time server S
- What wrong
    - By the time response message is received at P, time has moved on
    - P's time set to t is inaccurate
    - Inaccuracy a function of message latencies
    - Since latencies unbounded in an asynchronous system, the inaccuracy cannot be bounded
- Cristian's algorithm
    - P measures the round-trip-time RTT of message exchange
        - P => S => P
    - Suppose we know the minimum P -> S latency min1
    - And the minimum S -> P latency min2
        - min1 and min2 depend on operating system overhead to buffer messages, TCP time to queue messages, etc.
    - the actual time at P, when it receives response is between [t+min2, t + RTT-min1]
    - P sets its time to halfway through this interval
        - To: t + (RTT + min2 - min1) / 2
    - Error is at most (RTT - min2 - min1) / 2
- Gotchas
    - allowed to increase the clock value but should never decrease clock value
        - may violate ordering of events within the same process
    - allowed to increase or decrease speed of clock
    - If error is too high, take multiple readings and average them

NTP
- Network time protocol
    - NTP Servers organized in a tree
    - Each client = a leaf of tree
    - each node synchronizes with its free parent
        - primary servers
        - secondary serves
        - tertiary servers
        - client

- NTP Protocol
- WHY o = (tr1 - tr2 + ts2 - ts1) / 2
    - Offset = (tr1 - tr2 + ts2 - ts1) / 2
    - Let’s calculate the error
    - Suppose real offset is oreal
        - child is ahead of parent by oreal
        - parent is ahead of child by -oreal
    - Suppose one-way latency of message 1 is L1 (L2 for Message 2)
    - No one knows L1 or L2
    - Then
        - tr1 = ts1 + L1 + oreal
        - tr2 = ts2 + L2 - oreal
    - Subtracting the second equation from the first
        - $|oreal - o| < |(L2-L1)/2| < |(L2 + L1)/2|$
        - Thus, the error is bounded by the round-trip-time
- And Yet
    - We still have a non-zero error
    - We just can’t seem to get rid of error
        - Can’t as long as message latencies are non-zero
    - can we avoid synchronizing clocks altogether, and still be able to order events?

Lamport Timestamps
- Ordering Events in a distributed system
    - To order events across processes, trying to sync clocks is one approach
    - What if we instead assigned timestamps to events that were not absolute time?
    - As long as these timestamps obey causality, that would work
        - If an event A causally happens before another event B, then timestamp(A) < timestamp(B)
        - Humans use causality all the time
- Logical Ordering
    - used in almost all distributed systems since then
    - almost all cloud computing systems use some form of logical ordering of events
    - define a logical relation Happens-Before among pairs of events
    - Happens-Before denoted as ->
    - Three rules
        - On the same process: a - > b, if time(a) < time(b) (using the local clock)
        - If p1 send m to p2: send(m) -> receive(m)
        - (Transitivity) If a -> b and b -> c then a -> c
    - Creates a partial order among events
        - Not all events related to each other via ->
- In practice: Lamport Timestamps
    - Goal: Assign logical (Lamport) timestamp to each event
    - Timestamps obey causality
    - Rules
        - Each process uses a local counter (clock) which is an integer
            - initial value of counter is zero
        - A process increments its counter when a send or an instruction happens at it. The counter is assigned to the event as its timestamp
        - A send (message) event carries its timestamp

    - For a receive (message) event the counter is updated by max(local_clock, message timestamp) + 1
- Not always Implying causality
  - concurrent events
- Concurrent Events
  - A pair of concurrent events doesn’t have a causal path from one event to another (either way, in the pair)
  - Lamport timestamps not guaranteed to be ordered or unequal for concurrent events
  - Ok, since concurrent events are not causality related!
  - Remember
    - E1 -> E2 => timestamp(E1) < timestamp (E2), BUT timestamp(E1) < timestamp(E2) => {E1 -> E2} or {E1 and E2 concurrent}

Vector Clocks
- Vector Timestamps
  - used in key-value stores like Riak
  - each process uses a vector of integer clocks
  - suppose there are N processes in the group 1…N
  - Each vector has N elements
  - Process i maintains vector V_i[1…N]
  - j’th element of vector clock at process i, V_i[j], is i’s knowledge of latest events at process j
- Assigning vector timestamps
  - incrementing vector clocks
  - on an instruction or send event at process i, it increments only its i’th element of the vector clock
  - each message carries the send-event’s vector timestamp v_message[1…N]
  - on receiving a message at process i:
    - v_i[i] = v_i[i] + 1
    - v_i[j] = max(v_message[j],v_i[j]) for j != i
- Causally-Related
  - Two events are causally related iff
    - VT_1 < VT_2
      - iff VT_1 <= VT_2 &
        - there exists j such that
          - 1 <= j <= N & VT_1[j] < VT_2[j]
- Concurrent
  - Two events VT_1 and VT_2 are concurrent iff
    - NOT (VT_1 <= VT_2) AND NOT (VT_2 <= VT_1)
    - denote this as VT_2 ||| VT_1
- Logical Timestamps: Summary
  - Lamport timestamps
    - Integer clocks assigned to events
    - Obeys causality
    - Cannot distinguish concurrent events
  - Vector timestamps
    - Obey causality
    - By using more space, can also identify concurrent events
- Time and Ordering: Summary
  - Clocks are unsynchronized in an asynchronous distributed system

  - But need to order events, across processes
  - Time synchronization
    - Cristian’s algorithm
    - NTP
    - Berkeley Algorithm
    - But error a function of round-trip-time
  - Can avoid time sync altogether by instead assigning logical timestamps to events
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
- sdf