Contents

1 Introduction

- The questions we ask of each database:
  - *What type of DB is this?* Relational etc.
  - *What was the motivation?* Who developed and why.
  - *How do you talk to it?* Through shell, programming, protocols?
  - *Why is it unique?*
  - Performance/scalability.
- Large genres of DBs:

  Relational Two-dimensional tables from set theory, queries are in Structured Query Language (SQL), based on relational set theory. PostgreSQL.

  Key-Value Literally a hash. Riak + Redis.

  Columnar Instead of storing rows of a table together, store columns together. Easy to build sparse attributes. HBase.

  Document Stores hashes of basically anything (JSON). MongoDB + CouchDB.

  Graph Stores nodes and relationships between nodes, can traverse along relationships effectively. Neo4J.
- Best practice is obviously to use multiple DBs for diff use cases.

Glossary:

Relational Based on relational albegra, not based on relations between tables.

CRUD Create Read Update Delete. Every other operation is a higher order composition of these.

REST REpresentational State Transfer, guideline for mapping CRUD resources to URLs.

MapReduce Not a new concept, but succinctly captured by *It is faster to send the algorithm to the data than the data to the algorithm.* (*Note: Generally best when reducing in MapReduce to reduce to the same schema as mapped values to allow chaining reduces.*)

CAP Theorem When a network partition error occurs and messages are lost in a *partitioned* network, you can only guarantee *consistency* (across partitions) or *availability* to further incoming requests. More precisely, *at any given moment in time you cannot be consistent, available and partition tolerant.*

2 PostgreSQL

- Roots in 1970s, supported SQL by 1996. Relational. Archetypally stable/reliable.
- Relations = `TABLE`s, attributes = `COLUMN`s, tuples = `ROW`s.
- On `INSERT`, can specify `RETURNING` to get any automatically populated values e.g. `SERIAL` primary keys.
- Joins

Inner Join Join two columns from two tables on equality of those columns.

Outer Join Join two columns from two tables and for at least one of the two tables always return even if the lookup in the other table fails.

- *Indexing* helps avoid full table scans. PostgreSQL automatically indexes the primary key and all `UNIQUE` attributes. Can do either hash or btree indexes. Also when specify `FOREIGN KEY`, index target table.
- Can `INSERT INTO` values that are `SELECT`ed from another table! Handy to prevent hardcoding primary keys everywhere.
- Aggregate functions allow post-processing, e.g. `count()`. Can `GROUP_BY` aggregate functions and can also filter on aggregated values with `HAVING` the same way `SELECT` filters with `WHERE`.
- Can use `PARTITION_BY` to not collapse rows within each group. Use case is when `SELECT`ing over a field not used in an aggregate query and so can get conflicting values when also `GROUP_BY`ing.
- Transactions + ACID compliance:
  - Transactions ensure that every command of a set is executed else none.
  - ACID—Atomic (all or nothing), Consistent (never stuck in inconsistent state e.g. nonexistent foreign keys), Isolated (transactions do not interfere), Durable (committed transactions will always endure even if server crashes).
- Can store procedures (`FUNCTION`s) that are loaded by the database side. Obviously faster than postprocessing returned data from the db but higher maintenance cost.
- Can specify `TRIGGER`s that hit these stored procedures
- Lifecycle of a SQL command: parsed into query tree, modify ased off rules (and views, which are a specific type of rule), hit query planner, executed and return.
- Can specify custom rules e.g. how to interpret certain operations on a view.
- Can fuzzy string search using `LIKE` and `ILIKE`, can regex or even Levenstein (edit distance), trigram. All have corresponding indexes that can be built, all plugable using PostgreSQL-exclusive packages.
- The `cube` seems neat, you can define feature vectors for each row and tell PostgreSQL how to measure distances between feature vectors and query on said distance.
- Apparently does not scale well horizontally b/c partitioning is difficult for relational databases. But is very good for normalized data, extremely reliable w/ transactions + ACID compliance.

3 Riak

- Riak is a highly distributed, highly available key-value DB that is built for a web interface, notably to be `cURL`ed.
- All nodes are equal, very easy to join nodes `riak-admin join`.
- Insert by `curl -X PUT`, delete by `curl -X DELETE`.
- Key format is `<server>/<bucket>/<key>` and the key can be auto-generated on insertion.

- Can `Link` values to metadata labels. Say that the label `contains` the value. Is one-way pointer.
- Can query on links, called *link walking*, with `GET` to `<server>/<link bucket>/<link>`. Can specify to `keep` each step in the link walking.
- Can also tag with various other metadata, `X-Riak-Meta-*`. Can also specify MIME type to store images etc. that are linked to existing entries.
- Can execute commands by posting JSON bodies to endpoints e.g. `/mapred`. Endpoints often take a function body in plaintext, so can generally point to a bucket + key instead. Stored procedures!
- Riak supports filtering keys prior to map reduce.
- The *Riak Ring* is key to consistency and durability;
  - All nodes are peers, growing and shrinking the cluster is trivial.
  - Riak uses a 160-bit *ring* hashing keys to determine which Riak servers store the values for which key.
  - Riak then partitions the ring and each server claims partitions sequentially on startup.
  - Riak accomplishes redundancy by hashing each key to `N` nodes, considering a write successful when `W` nodes have completed the write, and then you can specify reading from `R` nodes.
- Writes in Riak are by default not durable and not written to disk before acknowledgement.
- Riak by default 204s when writing to a server that is not yet up, and a neighboring node will buffer the write! Careful for *cascading failure* when the neighboring nodes fail consecutively due to overload.
- Riak handles concurrent writes by effectively tagging each update w/ a commit history, and conflict resolution must be performed manually by specifying which `Vclock`s an update overwrites.
  - The commit history, called the *vector clock*, is pruned as more updates to a value occur, configurable per bucket.
- Can specify pre/post commit hooks, notably validators and/or computed values.
- Search wih Apache Solr interface, `/solr`.

4 HBase

- Apache HBase is made for very big data, on the order of GBs (EN:?? That's really small.).
- HBase also uses buckets of data it calls *tables*, and *cells* that appear at the intersection of *rows* and *columns*, but is not an RDBMS at all!
- Built on Hadoop, strong for analytics since many features e.g. versioning, compression and old data purge make it an appetizing prospect for data analytics.
- XML configuration, by default uses temporary directory to store data (`hbase.rootdir`), JRuby shell.
- Tables in HBase are just big maps of maps. Each key maps to a *row* of data, which consists of maps from keys/*columns* into uninterrupted arrays of bytes. Columns' full names consist of two parts, a *column family* name and a *column qualifier*, often concatenated with a colon.

- Use `put <table>, <key>, [<column>, <value>, ...]` to insert, and `get <table>, <key>, [<projections>]` to query.
- All entries are timestamped and chill around, baked in versioning! `put` and `get` can accept timestamps. Default 3 versions, can set to store `ALL_VERSIONS`.
- Interesting case study, Facebook's message index table:
  - Row keys are user IDs.
  - Timestamps are used as messageIDs
  - Column qualifiers are the individual words of messages.
  - Allows for fast searching of messages, just by looking at all timestamps of a given word, then querying along the row for all matches to each timestamp! Wow!
  - Since messages are immutable, no reason to use versioning, clever overloading of timestamp!
- Can make schema changes with `alter`, but must `disable` the table. Internally, creates a new column family and copies all data over, so extremely expensive!
- No formal schemas! Do not enforce certain column qualifiers to exist, allows unknown column qualifiers.
- Column families allow for different keys to be configured with different performance parameters.
- HBase operations are all atomic at the row level.
- Script in Ruby for larger operations e.g. XML streaming imports.
- Inbuild support for compression, Gzip, and Bloom filters for seeing whether a row key or a row + column exists w/o making a disk call!
- Scalability by assigning regions to row keys, servers own regions.
  - Write-ahead log (WAL) buffers edit operations, which are batch persisted to disk.
  - Track where regions are assigned with the `.META` table, and assignments are done by the *master* node, which can also be a region server.
  - Regions allow for built-in distributed processing, operating over regions in parallel.
- Generally keep column families per table down, colocate data that is often fetched together but otherwise more tables is preferred.
- Can use Thrift API to access remote HBase clusters.
- One notable weakness is that HBase is designed for scale, so using any fewer than five nodes is not recommended by the HBase community. Also does not have support for indexed columns, so have to be very clever about it!

5 MongoDB

- MongoDB is highly versatile, document database and schemaless (as of the writing, by now you can specify validators).
- Powerful for being able to do ad hoc queries. Stores arbitrary JSON documents.
- Databases contain collections (similar to *buckets*).
- Auto-generated autoincrementing `_id` consisting of timestamp, client machine ID, client process ID and a 3-byte incremented counter.
- Purely JS, so syntax for operators `{ $op: val }` is a bit verbose, but can build queries like JS objects!
- `$elemMatch` is amazing, lets specify multiple criteria for a query on a nested field in Mongo.
- Can pass code into most mongo functions, and especially into `$where` blocks!
- Indexing is done in a B-tree. `db.coll.ensureIndex()`.
- Can also `db.coll.[distinct, group]()`.
- Turns out that most of the magic we execute on Mongo are actually commands sent to the server and executed on the server side, aliased via the `$cmd` collection. Can replicate with `db.eval`.
- Map reduce is also doable, can store the results to a collection as a materialized view or print inline.
- Replica sets + sharding provide durability + scale.
  - Replica sets are master-slave, writes and reads only through master by default.
  - Elections when master fails, can drop data.
  - So writes can drop when master fails; solve by requiring majority of nodes written before considered written.
- Multi-master is difficult to resolve conflicts, Mongo simply disallows, vs. Riak where we did manual conflict resolution.
- Use a `mongos` to connect to a `mongoconfig` config server and track sharding configuration. Can `mongo` into a `mongos` instance and use normally, sharding is handled behind the scenes!
- Can use `{ location: "2d" } ensureIndex`es, then can use the `$near` operator to get an indexed query *near* a point in 2D space!

6 CouchDB

- JSON + REST document-oriented database, robust under network failure, scales to all sizes. No ad-hoc querying though vs. Mongo.
- Meant to be able to be deployed anywhere and extremely robust: the only way to shut it down is to kill the process! Append-only storage model.

- Behind a REST interface like Riak.
- From `<server>/_utils`, can create DBs and manually insert documensts.
- Automagically assigned `_id` field, just like Mongo.
- All fields that begin with an underscore have special meaning, in particular `_id` and `_rev`.
- To update/delete an existing document, need both `_id` and `_rev`, which is just an autoincrementing int with a uuid after. Can't update any revision but the latest. Prevents conflicts!
- Behind REST interface, `<host>/<db>/<doc._id>`
- `GET` to fetch objects, `POST` to create, `PUT` to update, `DELETE` to delete.
- Views consist of a mapper and reducer that generate an ordered list of key-value pairs. Key is whatever the mapper emits and value is whatever is aggregated behind the key (often a list of `doc._id`'s). Ordered by whatever key mapper emits. Built in view `_all_docs`.
- Can `POST` to `_temp_view` with a body containing mapper and reducer! Can also persist a view as a *design document*.
- CouchDB recursively calls the reduce function until no duplicate keys remain! To this end, reduce functions take `(key, values, rereduce)` where the third parameter is a boolean describing whether it's a rereduce. Best practice dictates we should never need this function...
- Changes API makes it easy to keep up to date on data changes.
  - `cURL` can just hit `<host>/<db>/_changes` for paginated view of changes to the db. Polling.
  - Can also specify `?feed=longpoll` to leave connection open for a while, can implement a listener on any new data on the connection (example in Node in book).
  - Can also specify `?feed=continuous` instead for continuous stream!
- Replication:
  - Supports multi-master a.k.a. master-master replication!
  - Inserting a new master only copies inserts, not deletions/updates.
  - If two masters get different updates, then replication starts, CouchDB just deterministically picks one to win and stores the other as a conflicted version, accessible with `?conflicts=true`.

7 Neo4J

- Graph database! Means inherently schemaless.
- Terminology:

  Node Vertex between edges that may hold data as a set of key-values, *properties*. Each node contains by default property `{id}`.

  Relationship Joins two nodes, labeled with a label, *type*.

Visualization Profile Can basically strformat a label for each node in the visualization, store as a profile.

- We will learn to use Neo4j via Gremlin, a language built for graph traversal written in Groovy but basically a DSL.
- Gremlin loads the graph as `g`, can do `g.V` which returns `v[i]` enumerated, or `g.E` which returns

  `e[i][<source node id>-<type>-><dest node id>].`

  Index into verticies `g.v(index)` (note, lower case for method). Can also filter via `g.V.filter{it.prop==’val’}`.
- Retrieved verticies have member functions
  - `map()` retrieves the full map of the node.
  - `inE(), bothE(), outE()` retrieves list of incoming and outgoing edges.
- Retrieved edges have member functions
  - `inV()` retrieves the incoming vertex for any edge.
- Note that all these functions will automatically map if called by multiple results! e.g. `<vertex>.in()` gets all `<vertex>.inE().outV()`. Formally, operations are a series of *pipes*, so a *pipeline* can operate on any number of inputs/outputs.
- Sample query that can be built is

```
v = g.v(0)
v.out(’prop’).in(’prop’).filter{ !it.equals(v) }
```

  which we can read to look at all other nodes that have edge `prop` to the same node that `v` has edge `prop` to.
- To turn a pipeline into a single element, call `next()` on it. Can also `>> N` to get the first N entries of a pipeline.
- Can write looping pipelines with `loop(){<predicate>}`, where we can use `it.loops` in the predicate for how many times the loop has executed.
- More useful commands `dedup(), paths(), groupCount(), each()`, some more Groovy built-ins like `collect(), inject()` mapreduce, define functions like any sensible language.
- CRUD done by calling `save()` on edges/verticies (not pipelines, so use `next()`!).
- Also has a REST interface! Not gonna note this, pretty redundant functionality.
- Can build indexes, query on them.
- Neo4J is ACID compliant! Transactions yay.
- High Availability mode is eventually consistent. Slave writes are enabled though, just propagates to master. Solutions for per-session consistency are to bind a session to a single server. In any case, strongest with read-dominated access patterns.
- Shipped with easy-to-use backup tool, can do incremental backups!

8 Redis

- Super fast key-value store db, but supports beyond key-value store to advanced data structures, heavily speed-optimized. Also commonly used for publish-subscribe systems. Only 20k lines of source code, a very simple project w/ simple interface.

- Built in `redis-benchmark` allows testing of conf; obviously very perf-emphasis from start!

- `SET <key> <value>` and then `GET <key>` from the `redis-cli`

- `MSET <key1> <value1> <key2> ...` and `MGET <key1> <key2> ...`.

- `MULTI ... EXEC` blocks form transactions, or more precisely, it queues the operations. Rather than SQL, where we had to rollback transactions, `DISCARD` simply empties the current `MULTI` queue.

- Redis natively supports complex data structures with *extremely* large sizes, 2^{32}:

  Hashes Commands generally prefixed with `H`. *Cannot nest*, unlike Mongo/Couch!

  List Commands prefixed with `L, R` depending on which side of the list they act, so can act as both queues and stacks.

  Blocking List Commands prefixed with `B`, block an operation until the *next* value can be returned. Publish-subscribe!

  Set Commands prefixed with `S`, can add to different sets and perform set operations on them and can even store the results into a new key!

  Sorted Sets Prefixed with `Z`, sets are stored in sorted order by their key. Think priority queue! Built in operations to update the priority of each entry as well as most of the previous set operations.

- Redis provides built in expiry, since it's commonly used for caching. Can use `EXPIRE` to set expiry for existing entries or use `SETEX` (and relatives presumably) to set with expiry. Can use `TTL` to check time remaining, and can undo timeout with `PERSIST`.

- Can `telnet` in, or can *pipeline* with `nc` netcat, send multiple commands to all be executed with a single api call.

- `SUBSCRIBE` can subscribe to a particular key, called a *channel* for pubsub purposes. Producers `PUBLISH` to the channel instead and get response of how many subscribers received the message.

- Redis has a few persistence options, the most basic being no persistence at all (purely in memory!). Redis also by default only saves occassionally, get `LASTSAVE` to get last saved time. Can alter `save` fields to say "when there are N writes then save within M seconds."

- Can provide more persistence via an `appendonly` log of all writes that can be re-read on server crash. Can also set how often to append, generally `appendfsync everysec` is used in `redis.conf`.

- Interestingly, Redis is not natively built to be secure: `requirepass` conf and `AUTH` command access plain text passwords w/o debounce!

- Instead, provide command-level security via obscurity by renaming commands like `FLUSHALL` to a random key instead!

- Also provides master-slave replication! Seems like no secondary reads/writes, just for backup.

- For distributed dbs, client is responsible for computing which distributed server a key lives on.
- Use Bloom filters with built in `SETBIT, GETBIT` commands to do really fast membership!