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.

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.