# Operational Databases

Nandan Rao

September, 2019

## Goal

Understand the good-old-fashioned relational database (SQL): why it is the way it is.

## Where data comes from

- Humans
- Machines

## Human Data

Some data is carefully collected by humans, entered into excel, and provided to you in a CSV so that you can analyze it.

## Machine Data

A lot of data, however, is created by software applications.

This data isn’t made to be analyzed. It’s created by the software application for its own purpose.

For software, by software.

## Applications

Software applications really just:

1. Record data
2. Mutate data
3. Display data

## Application Databases

Applications use databases to help them with those tasks.

There are many different types of databases they might use to help them. Often, they use many databases at the same time!

For now, let's assume we can only pick one database for our application. What qualities would we want from that database?

## Qualities one might want in a database

- Safe place for data to live
- Fast to record, mutate, and get my data
- I don’t mess anything up when doing any of the above operations
- Single source of truth (consistency)

These are not easy to obtain at the same time.

## Safety

If I want my data to be safe, I might put it on the most durable thing we have: harddrives.

But harddrives are slow.

So it’s hard to store data on a harddrive and make it fast to work with. This is a fundamental problem with databases.

## Fast Harddrive Access

To make it faster to work with data on a harddrive, I need to minimize how much data I have to read or write to the disk for any given operation.

Indices help solve that problem!

## Concurrency

No matter how fast I manage to get everything working, it won’t be fast enough for “web scale”.

Web scale means that many many users might be using my application at once, which means I might have to do many many operations in any given second.

Disks are simple too slow to handle all these operations one-at-a-time. Thus, we need concurrency.

## Dealing with concurrency

Performing a lot of operations at once gets tricky: how do we not mess anything up?

What if someone is trying to mutate some data and someone else is trying to read it, at the same time?

What if two operations come in that can’t both be fulfilled? Money transfer in a bank?

## Dealing with failures

Failures happen.

What happens if power goes out while I’m in the middle of updating some data, and I leave the data in such an “invalid” state. My application will break! Everything will break!

## ACID

- Atomicity. Operations are grouped into atomic “transactions” which either all fail or all succeed.
- Consistency. Transactions convert data from one valid state to another.
- Isolated. No transaction affects any other transaction (it can be concurrent, but must act as though it were serial).
- Durable. All transactions, once finished, are safely stored.

## Why ACID?

It should be clear that ACID is helpful in this model of the "operational database" - a database that is used by software to record, mutate, and display data to users.

This is a paradigm where there are *many* users at once and generally they record/mutate very few things at a given time. They read few things and what they read must be in perfect shape. Data is sacred and must be kept in perfect shape at all times, so that the software application itself to rely on the data.

## The Dominance of Relational Systems

Relational databases were invented in the early 70's to solve a lot of problems of basic systems including:

- A declarative language for accessing data (what evolved into SQL) that abstracts from underlying data structures.
- A "workable abstraction" of the data into "normalized tables."

ACID, too, became a core feature of these relational systems. These systems completed dominated the world of operational databases for many years and are still dominate today!

## 3NF for Humans

Third Normal Form (3NF) is what is commonly referred to as “normalized” data. It has some fancy terms if you look it up, but it’s actually simple:

- Data should be in “long” format (each column one type, each row one value).
- Each table has a “primary key”. This is a unique identifier and can be a composite of one or more columns.
- All columns in a given table should “depend” directly on the primary key. If they don’t there is redundancy, and data can be moved to separate tables (“normalized”).

## Normalizing Examples

Let’s look at some examples.

## Normalized Data

Normalized data has two advantages:

1. Saves space.
2. Makes things easy to update and avoid invalid states, everything exists in only one place!

ACID + Normalization – why they work well together.

## SQL

SQL as a language has proven very effective, even when the underlying system is not a relational database and not normalized, it provides a single language to do many things we want to do with data!

It’s important to remember, even though we often refer to these traditional, relational, ACID databases as “SQL” databases, that SQL is nothing more than a language that they happen to use!

## Other databases!

Because relational, ACID databases were dominate for so long, when people started using other databases they become known as “NoSQL” databases. This is terrible terminology, but it’s important to know. In common parlance, then:

- **SQL** refers to the type of database we discussed above (MySQL, Oracle, Postgres, MSSQL Server)
- **NoSQL** refers to *everything* else.

These terms are abused to no end. We will try and talk about specific technologies.

## Key-Value Stores

A KV store is in many ways the simplest form of database. It’s basically just a Python dictionary. Retrieving a single value by its key is extremely fast. Everything else is slow.

These are useful for many different applications, a cache being one of the most common.

Redis is a popular in-memory KV store, and RocksDB is a popular embedded disk-backed KV store.

## Document Stores

Some databases throw away the idea of tables completely and think of data as “documents.”

Often this means JSON.

JSON is very flexible, infinitely nestable, and schemaless, allowing these databases to store things in a format that is often *natural* for the data and *flexible*. Flexibility, however, is a double-edged sword!

MongoDB is a popular document stores used as an operational database.

## Search

Some databases are specialized in searching. They put a lot of CPU, memory, and disk space into creating optimized indices for retrieving data quickly based on complex queries (full text, numeric, geospatial).

The natural downside is that can’t write data quickly!

Elasticsearch is a popular search database (it’s also a document store).

## Wide Column

Some databases still use tables, columns, and rows, but add more flexibility to the table format: allowing, for example, rows to have different columns! Or nested columns, or lists inside columns, etc.

Cassandra is a popular example!

## Graph

Some databases take an entirely different structure. A graph database can be especially interesting to data scientists. This database stores its data entirely based on their relationships with each other: a graph!

Neo4j is the most popular example.

## Not ACID

Some NoSQL databases throw away ACID requirements. Many throw away normalization requirements. By getting rid of these constraints, they easily excel in other areas. For example, they might scale better (distributed and ACID and speed is tough to get all at once).

We'll talk about distributed storage a lot more in the following weeks!