Normalizing Data

What we will cover...

1. Where data comes from
2. Operational database goals
3. ACID
4. 3NF

Goal

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

Context

There are two main uses of databases we want to discuss:

1. Operational Databases (this lecture)
2. Analytics Databases (future lecture)

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.

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 (gets partially-modified data!), 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.

ACID by example

Let's consider some Python code to make this more concrete.

Imagine you are modeling a bank. You might create an `Account` class to model the state of each user's account.

```python
class Account():
    def withdraw(self, amt):
        self.funds -= amt

    def deposit(self, amt):
        self.funds += amt

alice = Account()
bob = Account()
```

ACID by example

Now imagine you want Alice to send 100 moneys to Bob.

Maybe your code would look like this.

```
alice = Account()
bob = Account()


alice.withdraw(100)
bob.deposit(100)
```

ACID by example

Now imagine you want Alice to send 100 moneys to Bob.

Maybe your code would look like this.

But something breaks! Maybe you had a bug. Maybe there was a power outage. But Bob doesn't get the money.

What's wrong?

```
alice = Account()
bob = Account()


alice.withdraw(100)
bob.deposit(100) # fails!
```

ACID by example

The solution is to wrap all the changes up in a `transaction`, where a transaction, if it succeeds, moves the database from one valid state to another valid state.

```
alice = Account()
bob = Account()

def transaction(alice, bob, amt):
    alice.withdraw(amt)
    bob.deposit(amt)

previous_state = save_states(alice, bob)
try:
    transaction(alice, bob, 100)
except: # but on steroids
    restore_states(alice, bob, previous_state)
```

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 (not indirectly).

Normalizing Examples

Let's define this.

Wide Data

id	name	follows_1	follows_2	follows_3
1	foo	873	738	3098
2	bar	983	999	348
3	baz	2789	389	987

Long Data

userid	name	follows
1	foo	873
1	foo	738
1	foo	3098
2	bar	983
2	bar	999
2	bar	348

Primary Key (simple)

id	userid	name	follows
1	1	foo	873
2	1	foo	738
3	1	foo	3098
4	2	bar	983
5	2	bar	999
6	2	bar	348

Remove Dependencies (simple PK)

userid	name
1	foo
2	bar

id	userid	follows
1	1	873
2	1	738
3	1	3098
4	2	983
5	2	999
6	2	348

Remove Dependencies (composite PK)

userid	name
1	foo
2	bar

userid	follows
1	873
1	738
1	3098
2	983
2	999
2	348

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!

Review

1. Where data comes from
2. Operational database goals
3. ACID
4. 3NF