I am a software engineer who focuses on databases. I was recently asked what I like about my job. This is my attempt to answer that question.

What do I like about my job as a software engineer who focuses on databases?

The shortest answer is that, for my work, *I get to interact with the truth-- to discover it, to reason about it, and to defend it*. The truth can be approached from many different angles, and people in very different careers could also describe their work as interacting with the truth. To be honest, I posed this question to some of my coworkers (who have the same job as me), and none of them gave this answer. But I hope in the next few minutes to explain what I do and to give a sense of what it means to me to interact with the truth as a software engineer who focuses on databases.

Before we go any further, I'd like to clarify some terms and provide some background context.

First of all, what is a database?
The term *database* can really refer to any structured data storage. There are three words there: structured, data, and storage, and we should think for a minute about each one. One: What is *data* (By the way I use the word "data" as a singular mass noun even though the Latin word is plural.), two: how is data *stored*, and three: what kind of *structure* are we talking about?

So, what is data. Data is any information really.
Your personal data probably includes

- your email (both the actual text of your emails and your account settings),
- your online text messages and photos,
- your bank account's contents and transaction history.

Businesses and governments store

- Financial data,
- Maps and weather data,
- Inventory and sales data,

Everything.

Okay so that's what data is; how do we store it?
We store it physically. All the world's digital data has to be stored somewhere *physically*. You're probably familiar with storing your data on devices like the flash drive inside your laptop, or an external hard drive or a floppy disk. The data is physically there on the device, just like paper files in a filing cabinet but in a much more compact format.

In the last decade there's been a big trend of storing everything *in the cloud*, which just means that you pay someone else to store your data on *their* physical devices, and when you wanna access some portion of your data, you download just that portion from them over the internet.

Data that's stored in the cloud is stored physically in a data center somewhere. A data center is a big building that's filled with thousands of racks of computers. Picture a Home Depot where

every aisle is just rack after rack of computers. For the computers themselves, you can picture boxes about the size of a closed laptop, with no keyboard or screen. Just a computing box with some wires that plug into the rack.

Globally, there are four main cloud providers who own and operate most of the world's data centers:

1. Amazon, whose cloud is called Amazon Web Services or AWS;
2. Microsoft, whose cloud is called Azure,
3. Google, whose cloud is called Google Cloud Platform or GCP, and
4. Alibaba, a Chinese company whose business is mainly in China.

Data centers require a lot of electricity. They also generate a lot of heat, so they typically use a lot of water for cooling. Therefore they're typically built in places with access to cheap electricity and water.

If you have a gmail account, the text of your emails might be sitting on some computer on a rack in a data center in Iowa or Oregon. Perhaps both Iowa *and* Oregan actually: Google keeps several copies of your data so you can still access it even if one data center goes down.

If a business stores its data "on-prem" (meaning on-premises) then that means the business operates its own mini version of one of these data centers, rather than renting the resources from one of the cloud providers. The trend in the past decade and a half has been that fewer and fewer companies store their data on-prem, and more and more companies migrate their data to be stored by one of the cloud providers instead. This makes sense: If you have a company that sells tractors or amusement park tickets, you probably don't want to also be in the business of maintaining your own mini data center; better to just pay Amazon to do that for you.

Okay hopefully by now I've driven home the point that all stored data is stored *physically*. But none of this yet has anything to do with databases. This is where the third word, *structured*, comes in. By structured, we mean that the data is physically laid out in some helpful way. We call the mechanism that provides this structure a *database*.

Much of the work of a database-focused software engineer is an attempt to answer the question: How can we cleverly organize our stored data in ways that make it efficient to access? This question applies just as easily to the organization of a paper filing cabinet as it does to the organization of a digital device. Maybe the paper files in the filing cabinet are organized into manila folders, and the folders are sorted alphabetically. If a more efficient organization system is out there, then it's always existed, we just have to discover it.

This is where the "reasoning about the truth" comes in. Some organization strategies are provably more efficient than others. Some strategies are even *optimal*: We can mathematically show that -- if you're trying to optimize for certain access patterns -- there's no more efficient way to lay out your data. Anyone across all time, in any culture, on any planet, who has

attempted to organize data, or even organize anything for that matter, has faced the same questions, and has run into the same mathematical and physical limits.

To demonstrate what I mean, let's revisit that example of alphabetizing some files. Putting things in alphabetical order might sound simple, but it's actually extremely powerful, and it's really used in practice.

Take the example of gmail. When you open your gmail inbox, you expect to see emails that were sent to you, that belong to your account. It would be a severe problem if you opened up your gmail inbox and it showed you emails that were meant for some other user. You also expect your inbox to load very quickly when you go to gmail.com.
So, when you open up your gmail inbox in your web browser, how exactly does gmail go into its database of all its emails, and quickly find the ones that were meant for your account? It's no different from if gmail stored all its emails in a big filing cabinet. If you knew that you were gonna have to retrieve a bunch of emails for a single user, you'd probably want the filing cabinet sorted by user. You'd also probably alphabetize the cabinet by email address. This would be much much more efficient than if all the emails were just stuffed haphazardly into the filing cabinet, and every time someone opened gmail.com you had to look through every email to find the ones that were sent to their address. Gmail probably does really truly physically store all its emails in Google's data centers *sorted by user*. Gmail probably translates your email address into a user ID number and orders all the users by ID number, but this is essentially equivalent to alphabetizing by email address.

So how can we quantify this efficiency boost that sorting gives us? Can we prove, is it a universal mathematical truth, that sorting a cabinet allows more efficient access to the files in the cabinet?

Imagine that you have a massive filing cabinet of all the Gmail emails, and for the sake of this example we'll say all the emails for a user are put into a manila folder, one folder per user. So when a user goes to gmail.com, all we have to do is find the right folder for that user.

Let's start by saying the folders aren't sorted; they're just randomly organized. And let's say the user example@gmail.com (that's e-x-a-m-p-l-e at gmail dot com) opens their inbox. What strategy would you take to find their folder in this unsorted cabinet? (We would call any strategy an *algorithm*.) There's no better available algorithm than to open the cabinet, start at the beginning, and look through every folder until you find the folder you're looking for. Maybe you get lucky and it's towards the beginning, or maybe you get unlucky and it's towards the end. Unless you get lucky, it's gonna be a very slow process, and even if you get lucky with "example", you probably won't get lucky again with whoever the next user is. And importantly, the more users we have, the longer this algorithm takes: a cabinet with twice as many folders will take twice as long to look through. This relationship between the size of the dataset, in this case the number of users or number of folders in the cabinet, and the time or number of steps required to perform the algorithm, in this case looking through every user or folder, is called the algorithm's *runtime complexity*. When we assess the efficiency of an algorithm, the main thing

we consider is the algorithm's runtime complexity. For this algorithm, where doubling the number of users means doubling the amount of time to find any one user's folder, we would say this algorithm has a *linear* runtime complexity, because there's a linear relationship between the number of users and the time required to find the right folder. The engineers building gmail really don't want to use an algorithm like this. Gmail launched in 2004 and now has over a billion users. If the time to load your inbox doubled every time the number of users doubled, Gmail would be totally unusable right now.

Now let's consider the case where we've sorted the folders alphabetically by email address. What's our algorithm? We know we're looking for example@gmail.com. So let's peek at some folder in the middle of the cabinet, let's say it happens to be michael@gmail.com. E (for "example") is before M (for "michael"), so we know that the folder we're looking for is somewhere in the half of the cabinet before michael. We're never gonna consider any of the folders that came after michael.
Then we peek at another folder in the middle of that first half, let's say it happens to b diana@gmail.com. We overshot, E is after D. But again, since we're never gonna consider any of the folders that were earlier than diana, we can discard half of the folders we had been considering, or one quarter of the total folders. Yes I'm just explaining how to find something that's alphabetized and you already know how to use a dictionary. I won't belabor this. But you get the idea: Each time you peek at a folder, you narrow down the remaining number of folders to look through by half, until you find the right folder.
So how would we describe the runtime complexity of this algorithm? It turns out that finding something in a sorted list has *logarithmic* runtime complexity, meaning the number of steps required to find what you're looking for grows with the logarithm of the number of total items in the list, or total number of folders in our filing cabinet.
Logarithm. Taking the logarithm of a number is essentially equivalent to asking how many digits are in the number. You can see why a Gmail engineer would be more satisfied with a logarithmic runtime complexity: As the number of users doubled from 500 million to a billion, the number of digits in the number of users only increased from nine to ten. And you can see how doubling the number of folders in our cabinet only increased the number of steps to find the example@gmail.com folder by one: When there are a billion folders in the cabinet, as soon as you've done the first peek at the middle folder, you then only have to consider either the 500 million folders to the left or the 500 million folders to the right. One single step and you've cut the problem in half. Gmail can vastly increase its number of users and the effect on gmail.com load times will be imperceptible.
Now, you probably caught that "how many digits are in this number" isn't the formal definition of a logarithm, and you're right, but there *is* a rigorous mathematical definition for evaluating runtime complexities, and in this case it actually is equivalent.

I hope this example demonstrates that something as simple as keeping your files alphabetized can dramatically improve efficiency in a way that can be proven with mathematical analysis.

You may have noticed that I glossed over something: When you go to gmail.com, it doesn't just dump your entire email history. Rather it shows your fifty newest inbound emails that aren't

archived or trashed. So there's a little more to this than just "sort the emails alphabetically by user". There's some organization of the emails *within* each user's account. In this case, the solution is again analogous to a paper filing system. The designer of the emails database would create an index, just like there's an index at the back of a textbook. In the back of a textbook, the index has topics sorted alphabetically, and next to each topic there's a page number or list of page numbers where you can find more discussion of that topic within the textbook. We've already established that you can find a topic very quickly if the topics are sorted alphabetically, so the index is a very efficient way of finding exactly where to look in the textbook for the topic that interests you. In the emails database, every email would be given a unique ID, and within each user's account, the emails would be sorted by ID. Then for each user there would be an index listing the timestamp of every non-archived, non-trashed inbound email, (those are the ones that should appear in the inbox,) and it would be sorted in reverse chronological order by the timestamp of the email. Next to each entry in the index, there would be the ID number of the email that arrived at that timestamp. Using this index, Gmail can quickly find the fifty newest non-archived, non-trashed emails for each user. Without the index, the time required to select the fifty emails to show in a user's inbox would grow linearly with the number of emails the user had overall, whereas with the index, it's logarithmic. One of the important roles of a software engineer focusing on storage is to identify ahead of time what an application's data access patterns will be, and to be sure that appropriate indices are in place to support those access patterns.

To recap, part of a database-focused software engineer's job is to design clever physical layouts that make our data efficient to access, and to prove that the layouts we design yield efficiency gains. These efficiency gains are just a mathematical possibility until we design a system that unlocks them. In this sense, this part of our job is to *discover* the truth.

The other big part of the job that I wanna talk about is how to guarantee the *correctness* of data in a database. There's little value to a database if the data in it isn't correct-- if it's not the truth. In this sense, part of our job is to *defend* the truth.

Here's a scenario:
Imagine that you have a bank account with $100 in it, and you go to withdraw $80. What your bank's software is probably doing is this: First, check in the bank's database for how much money is in your account: $100. Next, compare that amount to the amount you're trying to withdraw: $80. If you're trying to withdraw more than the amount in your account, like say in this case if you had requested $120 from the ATM, then the ATM should show you some error and not complete the transaction. If you're trying to withdraw an amount that's less than is in your account, which in this case, you are, because $80 is less than $100, then the ATM should give you the $80 in cash, and the software should update the bank's database to reflect that your account now has $20 in it.

But what happens if you enlist the help of an accomplice? Let's go back to there being $100 in the account. Say you and your accomplice both go to separate ATMs at the same time and both try to withdraw $80 from your account. If you time it just right, and both press enter at the same

time, what could happen? Both ATMs initiate this process in parallel in the bank's software: Both check the database for the amount in the account, $100. Both say 80 is less than 100, proceed. Both ATMs emit $80 cash, and then both processes update the database to show that the new balance in your account is $20. You just got $160 in cash, and you still have $20 in your bank account! The bank cannot allow this to happen. So what can they do?

The name for this category of problem is *concurrency control*: How can concurrent operations be guaranteed to get correct results? How can we guarantee that concurrent operations won't violate the truth that the database represents?

One simple approach would be for the bank's database to only allow a single operation at a time. Concurrent operations won't run into correctness issues if there *are* no concurrent operations. We call this approach *serialization*: All the operations are serialized, meaning executed one after another, never in parallel. But this would only work for a tiny bank with a few customers. Real banks may have millions of customers with thousands of transactions in flight at a time. There's not enough time in the day to process all of these transactions sequentially. Correctness is *most* important, but throughput is important too.

So what might be a better solution? Maybe we could say each user is only allowed to have one operation in flight at a time, but we allow operations for different users to proceed in parallel. This would allow much greater throughput than the first approach, and for a bank this may be good enough. But there may be cases where this throughput isn't good enough: Maybe the user is the IRS, and they have thousands of accounts, and they need to allow concurrent operations on those accounts as long as the specific deposits don't conflict with each other. This approach also runs into questions like how to handle transfers from one user to another. Those transfers may not fit this model.

There are many options, and different databases adopt different approaches. It's up to the software engineers to pick an approach that meets the needs of the specific application. Ultimately, every approach boils down to identifying operations that might conflict, and forcing them to execute serially instead of in parallel, either by identifying the conflict at the start of the process and only letting one operation proceed at a time, or by identifying the conflict at the end of the process and forcing all but one of the operations to restart themselves.

Part of the job of the software engineer is to design data access patterns that minimize the number of database operations that would need to be serialized, to allow maximum throughput to the database, while still guaranteeing that the data in the database represents the truth-- thus *defending* the truth.

To wrap up:
What I like about my job is that I get to reason about the truth, both to discover it and to defend it. I'm very lucky to get paid to think about puzzles that involve theory, math, and provable correctness. These puzzles offer me a connection to others throughout time and space who

have thought about them too. And from the stressful, contradictory world of subjective human experience, they offer me an escape.