What is a Database?

This title is abstract because this idea is rather abstract. The question surfaced recently while working on the digital library at pariyatti.org: How should we store the data? Documents? CMS? Wiki? Graph wiki? RDBMS? Graph database? And just what is “the data”?

Summary

In summary, it seems apparent that a user-editable graph database like Wikibase is ultimately the direction a huge digital library needs to go. But this requires the buy-in of librarians and many thousands of person-hours to build the library. On the other end of the spectrum, we have totally disorganized data: files and other documents (a YouTube video or Spotify podcast isn’t exactly a *file*). In the middle, we have the other options. They sort of form a spectrum:

Intelligent Graph (Graph wiki) => Graph DB => RDBMS => Wiki => CMS => Dumb Documents

The question of a custom system, built with its intentions tailored toward the immediate users rather than an ivory-tower vision of a complete digital library, boils down to only two of these options: Graph DB and RDBMS. For the purposes of this document it is assumed the reader understands that “NoSQL” and document databases are just poorly-implemented relational databases with undocumented schemas and that no one in their right mind would consider them as persistence for a digital library of any sort.

See: https://martinfowler.com/articles/schemaless

Graph Databases

I won’t bother explaining what a graph db is or why it’s a good fit for this problem space. If you can imagine a mathematical graph (directed graphs, DAGs, and the like), you can imagine a graph database. Concepts, categories, and other ontological components are easily captured this way, just like you might draw them on paper.

Obviously graph databases are much more complex than a digital version of a paper graph. But the concepts map one-to-one.

Relational Databases (RDBMSes)

The question leading to the creation of this document was this: Can't we create all the same relationships and entities we can create in a graph db inside an RDBMS? The answer is yes, you can. But there is a follow-on question: Should we? The answer to that question is no. Here's why.

One Thing Only

A relational database is very good at one thing and one thing only: saving data to disk ("inserts" and "updates") and reading that data from disk ("queries"). Okay, that's two things but they are two sides of the same coin. Over the years, this (rather amazing) skill has been conflated with all sorts of other light and magic: dynamic behaviour like triggers, APIs, networking, plugins, and even adopting some of the features which make document databases so appealing to some (like JSON storage).

These are not the key features of a relational database. More often than not, they have been added because commercial database vendors want to sell you a product and products with more features convince more CTOs (and DBAs). We'll take a small detour here to explain why these extra features are generally terrible ideas. Take triggers. A trigger is a sort of dynamic callback which is fired when an event occurs within the database. Sometimes this is useful. I can ask the database to do something it's good at, automatically. More often than not, however, triggers become *tiny programs*. This is bad. Triggers are usually managed by these aforementioned DBAs and very few DBAs I've met have the self-discipline to put their triggers (or any other tiny programs written in SQL) into version control. In a world where the database is everything, it also becomes your version control system. Databases do not make good version control systems.

Now that we've mentioned SQL we can take up that example. When "web services" were all the rage 20 years ago, both database vendors and their customers were tempted to create fancy new ad-hoc APIs on top of databases. (I'm not referring to ActiveRecord or DataMapper pattern implementations here.) This bad habit lasted for over a decade and I have rewritten entire systems where SQL was re-implemented as an object-oriented network protocol (which was, inevitably, translated into SQL). SQL is pretty good and — as far as relational databases are concerned — I haven't seen a better generic API built directly on top of a database yet.

Let's go back to what relational databases (for the rest of this section I'll just lazily refer to them as "databases") are good at: Putting data on disk and reading it off of disk. Why is this such an "amazing skill", you're asking? Don't files do that? Indeed they do and if you just need to save

something to disk and read it off of disk, with no complications, you don't need a database — you need a file. But in a system of any real complexity, there will be other considerations: How many processes/threads are talking to disk at once? How many of those processes are writing? How many are reading? Is there a potential for conflict? How do we notify a process that a write or read has failed? What happens when the computer running the database crashes? A good database solves these problems for you. Database reads and writes are not only consistent but they are *fast*. The database takes care of caching, transactions, conflicts, callbacks to the database client, replication, failover, and all sorts of other insanely helpful tools when building a large complex system that your customers expect will never fail. Ultimately, all these tools boil down to one thing: data on a disk.

What is "relational"? (By example)

There are two very important downstream effects of databases being so good at talking to hard disks: (1) Tables and (2) SQL. "Tables" is an oversimplification of the entire notion of a "schema" but if you imagine building a database from scratch you can see how we get here. Let's say I'm a chai walla and I let my customers carry a tab. First, I want to store a list of my **customers** on disk. It makes sense for this data to take the shape of the ledger I track my customers with in real life, with rows and columns. So I do that. But keeping a column in this table for "account balance" doesn't seem as effective as it could be. Even on paper, I'm able to track my customers' transaction history: They bought 8 chai and 6 biscuits (for coworkers, obviously) on a Friday, then a single chai on Saturday, and then they cleared their tab on Monday. If that customer goes on a business trip for 9 months and comes back wondering about his tab, the tab should keep track of the details. Especially on a computer… that's why we have computers: precision. So I create a *second* table to represent **account transactions**, but how can I quickly glance at the summary? The first question is "what's on my tab?" not "what's my transaction history?" — those sorts of details are only useful to settle a dispute. This **account summary** is best represented by a *view* — a dynamic table that the database updates in the background to keep track of these aggregates without calculating them every time. Last but not least, it would be helpful to keep track of the relationships between our little tables with something more unique than the customer's name (God forbid). So we create relationships between the tables based on a meaningless identifier (maybe an integer or a UUID). A **customer** has many **account transactions** but each **account transaction** will only belong to one customer.

Those are my *tables*. As my chai business expands and I open more locations, each location becomes a client of this database. SQL lets all my locations ask generic questions of the database like "what's on my tab?" or "when did this customer last visit one of my shops?" and it can do this because of the relationships between the tables. Even though SQL looks very much like English, it's actually based on something called a *Relational Algebra* which means it's both beautiful and systematic — but also not meant for me, as a chai vendor. SQL is for programmers and analysts to talk to the database via some user interface they provide me and my staff. When complicated situations arise, I won't be bothered with the inner workings of the

database because the programmers will shield me from the math of it all. For example, if my customer Prateek is buying a chai from one location while I look up his balance at another location, the balance I see is not corrupt or convoluted by the fact that there is a read operation happening at the same time as a write operation. Similarly, if Prateek gives his "Preferred Customer" card to a friend and they both happen to purchase a chai at once, in two separate locations, nothing will go wrong. Behind the scenes, the database will resolve any potential conflicts. Even if it can't, one transaction will go through and the other location will get a simple error message: "please try again." My employee tries again and the transaction goes through.

See: https://en.wikipedia.org/wiki/Relational_algebra

There is a large body of literature on the topic. One of the best books available on the subject of transactions is the biblical *Transaction Processing*. We don't need to go into any more detail here, though, and we won't.

See: "Transaction Processing" — https://dl.acm.org/doi/book/10.5555/573304

The important take-away here is that databases *rarely fail*. At least, they rarely fail to save something to disk or take something off of disk. The two sides of this coin can be further broken down into four operations, lovingly summarized as *crud*:

Create: Make some new data
Read: Look up some existing
Update: Change some existing data
Delete: Destroy some existing data

The sorts of complications mentioned above happen most often with the last two. Updating or deleting data is a *mutation*. And if you plan a little bit, you can treat *time* as a first-class citizen of your database: Every table becomes a log of immutable transactions, bound to the event in time when they occurred. Things are created or read but never updated or destroyed. Few databases enforce such a model so it's up to your programmers to do so. But it's a good model, even if it's rarely used.

Even if you subscribe to all four CRUD operations, though, a database is still an amazing tool that will rarely fail you. We've been building these tools for decades and they're really good at this simple task of saving data to a disk... as long as you want your data described by the *Relational Algebra*.

Uh-Oh

We don't always want a Relational Algebra. The Relational Algebra defines semantics which are specific to this tabular view of the world. While this is a great way to make lists of things, it's not a very good way to track other entities. The specific example in question is that of a digital library. This digital library will hold very different shapes of information. Some information can be kept directly (like PDFs and MP3s) but some will point to more abstract digital artefacts (YouTube videos and podcasts) and others will point to real-world objects (like a printed book, via its ISBN or a one-time event via its location and time). These things are all possible to track in a database. We might create tables for **files** (or something more specific like **pdfs** and **mp3s** or something in between like **documents** and **audio**). We can also create tables to track other digital artefacts, as long as it is possible to point to them by a URL. The **books** table might include other documents like **transcripts** and **excerpts** or perhaps these would have their own tables. It's up to us.

But how should we capture more nuanced semantics and metadata in our database? In our specific library example, we have **topics**, **collections**, **locations**, **authors** (who are also the subjects of the books and videos in question), and document details like whether or not something is a derivative work and which sub-section of the original document it pertains to if it is. Add to this the complexities of editions, different translations, different copyrights (and copyright laws), and our database is quickly becoming untenable. A *Relational Algebra* prefers nouns it can list.

If the above examples aren't convincing, we can go deeper down the rabbit hole. A **topic** might include subtopics: **emotions** contain **anger**, **sloth**, **fear**, and so forth. But there are different kinds of anger. Are we talking about anger management? Anger in the workplace? Anger in the family? Abstract anger? Collective anger in social media? We should have the toolkit to discuss and describe all these things in meaningful ways. This becomes harder and harder with a (relational) database.

Often, systems with such complex relationships do evolve in a relational environment. The consequence is almost always a dense web of many-to-many relationships and/or join tables which serve to provide the system maximum flexibility under the relational model. For anyone who has ever worked on such a system, this is the breaking point: the system must be rewritten because the relationships are no longer meaningful and there is no value in the constraints a database allows us to place on our entities. The relational semantics have broken down.

Uh-Oh, Uh-Oh

Sometimes such a system will be rewritten… but refuse to divorce itself from Relational Algebras and SQL-based databases. Instead, the programmers will attempt to lift themselves one layer of abstraction above the direct implementation. No longer will the system discuss mundane categories like **books** and **authors** and **emotions**. Instead, tables will begin to

describe a higher-level taxonomy with confusing (but list-worthy) nouns like **entity**, **state**, **process**, **category**, **class**, **group**, **type**, **rank**, **weight**, and so forth. At some point, such a system may even make the fatal decision to encode *relationships themselves* into a table.

Knowing what we know about the strengths of relational databases, this is obviously a mistake.

The Relational Algebra is a beautiful system, carefully engineered over the decades to solve precisely the category of problem it was always intended to solve. But that beauty and efficiency quickly breaks down when that category of problems is no longer meaningful. If every problem can be shoehorned into relational data storage by completely altering the level of abstraction, the database no longer knows how to do its job. For instance, a database is *very good* at quickly looking up a customer by name precisely because it knows where the customers are: the **customers** table. If the system interprets the world too abstractly and a “customer” is now just a special type of “entity” stored in the **entities** table, all the power of a relational database is lost. Add to this confusion the distinct possibility that such a system will inevitably attempt to reinvent the wheel by defining **relationships** as tabular data instead of the relational model itself, and the entire thing comes crashing down. The database can no longer impose logical constraints, it can no longer resolve conflicts easily, it can no longer find and create data quickly, and it can no longer create small, trustworthy transactions.

It will still work, mind you.

As I said, relational databases are amazing tools and the software industry has done wonders with them. Systems which survive for decades under a great deal of commercial and academic scrutiny become very solid. The world-wide web, Linux, the JVM, and PostgreSQL are good examples of this and they will all put up with a great deal of abuse and misuse before they actually break. That doesn’t mean we should abuse them.