Databases

What are databases?

First, what are databases for?

Storing data in your application (in memory) has the obvious shortcoming that, whatever the technology you're using, your data dies when your server stops. Some programming languages and/or frameworks take it even further by being stateless, which, in the case of an HTTP server, means your data dies at the end of an HTTP request. Whether the technology you're using is stateless or stateful, you will need to persist your data somewhere. That's what databases are for.

Then, why not store your data in flat files, as you did in the "Relational databases, done wrong" project? A solid database is expected to be **acid**, which means it guarantees:

- **A**tomicity: transactions are atomic, which means if a transaction fails, the result will be like it never happened.
- **C**onsistency: you can define rules for your data, and expect that the data abides by the rules, or else the transaction fails.
- **I**solation: run two operations at the same time, and you can expect that the result is as though they were ran one after the other. That's not the case with the JSON file storage you built: if 2 insert operations are done at the same time, the later one will fetch an outdated collection of users because the earlier one is not finished yet, and therefore overwrite the file without the change that the earlier operation made, totally ignoring that it ever happened.
- **D**urability: unplug your server at any time, boot it back up, and it didn't lose any data.

Also, a solid database will provide strong performance (because I/O is your bottleneck and databases are I/O, so their performance makes a whole lot more of a difference than the performance of your application's code) and scalability (inserting one user in a collection of 5 users should take about the same time as inserting one user in a collection of 5 billion users).

ACID is a cool acronym! CRUD is another cool one

You will definitely run into the concept of "CRUD" operations. It's just a fancy way to refer to the 4 operations that can be performed on the data itself:

Help

- **C**reate some data;

- **R**ead some data;
- **U**pdate some data;
- **D**estroy some data.

Obviously, a database should allow all four. Yes, that's it.

2+ kinds of databases

When people talk about databases, they're usually referring to **relational databases** (such as PostgreSQL, MySQL, Oracle, …); but there are many other kinds of databases used in the industry, which are globally referred to as **"NoSQL" databases**, even though they can be very different from each other, and serve very various purposes. Also, the name "NoSQL" comes from **SQL**, which is the name of the syntax used to give orders (CRUD operations, creating and deleting tables, …) to a relational databases; however, some non-relational databases, which are referred to as "NoSQL" give the option to use the SQL syntax. Therefore, the term "NoSQL" is quite controversial to refer to non-relational databases, but it is still widely used.

"NoSQL" (non-relational) databases have known a boost in popularity, over the last decade or so, so much that there was a point, a few years ago, where people were wondering if they were to replace relational databases entirely. But years later, the market has now solidified, NoSQL databases' market share doesn't progress much anymore and is now quite steady. The result: many NoSQL databases have made it into solid maturity, and are used in some very ambitious projects (as well as small ones), but relational databases are still by far the most used in projects, and are not going anywhere after all.

Therefore: it is crucial for a software engineer to know very well how relational databases work, because the odds are very strong that you will encounter them in your career; but it is also very important to get acquainted with the most popular types of NoSQL databases, because the odds that you run into them, however kinda smaller, are pretty strong too.

SQL

In order to work with relational databases, you will need to get familiar with SQL syntax. A lot of developers will acknowledge that they find the SQL syntax unpleasantly hard to use, which has some outcomes:

- Engineers that are comfortable with SQL are very respected in the industry, even more so in this age where data has gotten so valuable. To be honest, the fact that I aced the SQL challenge on my Apple interview is probably a huge reason for me to have gotten the job; it turns out the initial role was a lot about manipulating data.
- The fear of SQL explains a lot why non-relational databases got called "NoSQL", a bit like if it was a statement, a complain. Non-relational databases push a lot the button of not having to use SQL.
- Modern full-fledged frameworks contain tools that are called ORMs, and one of their roles is to abstract away SQL queries (which is good for day-to-day ease of use, but can turn out very dangerous (/rltoken/yjvRunSOVKlEuCyzfLY6EQ)). We'll cover ORMs more later, but it's worth noting that you do find back-end engineers in the industry who work with relational databases, but never write a line of SQL, which makes them a lot less valuable on a project.

- For a beginner, keep in mind that SQL's syntax is a bit hard to wrap your head around, so maybe you should follow a tutorial first. Please don't try to memorize the SQL syntax. I've used SQL extensively in very advanced cases, on systems with hundreds of millions of records, and I still go on Google each time I need to compose a SQL query.

Some terminology around relational databases

One good thing about relational databases is that whether they're PostgreSQL, MySQL, Oracle, or other, they've managed to be pretty consistent across brands. Therefore, not only are their versions of SQL pretty decently similar (at least for CRUD operations), but the terminology they're using are mostly the same.

Say you need to store users. To do that, you create a **table** that is called "users".

Your users have 3 pieces of information to store: their "id", their "login", and their "password". Those are called **columns**, and they all have **types**, like **integer** for the "id", **varchar(32)** for "login" (a string of variable length, but maximum 32), and **char(32)** (a string of exactly 32 characters, which is the case for all text encrypted with the md5 algorithm, for instance). The available types may vary heavily from one database "brand" to the other.

Now, let's add a user in the database with SQL:

```
INSERT INTO users (login, password) VALUES ('rudy', '0123456789012345678901234567890
1');
```

This adds a **row** in the table (sometimes also refered to as a **record**, or more rarely, a **tuple**).

Why are they called "relational" databases?

Historically, the initial reason was that tables used to be called "relations" (they gather a lot of datas that are "related" to each other, since they follow the same structure). However, tables are now tables, and the term "relation" has now been recycled for another use.

A **relation** as used today is something that ties two records together, most often across different tables. For instance, say you have a blog, and you have 2 tables:

- posts, with the fields id, title and body
- comments, with the fields id and body

In both tables, the "id" fields are **primary keys**, because they uniquely identify the row that they belong to (if you say "give me the post of id 4", you're sure to be getting only one post).

But how do you know that a given comment is attached to a given post. Well, you add a post*id field to the comments table, containing the id of the post you with to attach it to. The post*id field is called a **foreign key**, uniquely identifying another's table primary key.

Now that you have that, you can easily identify, from a comment, which post it is attached to; but you can also easily identify, from a post, which comments are attached to it. Just fetch the comments whose post_id field contain the id of the post you had in mind. The fact that you can do that is what is called a relation.

Once you have your relation, you can do pretty advanced things. For instance, you can join tables together while querying them, which will allow you to search for "the comments whose posts were published within the last month", for instance (well, provided the posts table has a published_at column of type date, for instance).

Note: you can have a relation between rows of the same table, for instance, a user that is the "sponsor" of another one, a comment that is a "reply" of another one, …

Some more terminology around relational databases

Indexes

Say you want to get all of the comments that are attached to the post of ID 12:

```
SELECT * FROM comments WHERE post_id=12;
```

If you have millions or billions of comments, having your database extract the comments that match this condition can be amazingly time-consuming. Therefore, you can add an **index** on the comments table, that applies to the post_id column. This will "precompute" every possible SELECT query with WHERE conditions on this column, which will update themselves every time you modify data, so that those calls are ready to respond very quickly.

Let's complicate things a bit, and say you want to optimize this query:

```
SELECT * FROM comments WHERE post_id=12 AND published=1;
```

Your index on the post_id column might not help much on that query. However, for that query, you can absolutely define an index on multiple column (in this case, the columns post_id and published).

Setting indexes properly is a known quick win to improve performance of relational databases on queries that are performed very often and take a long time to respond (so-called **slow queries**). I can quote at least a dozen occurrences in my career where setting up an index properly boosted a database's performance with minimal effort, the most notable of which allowed us to boost a data migration that was taking ~48 hours, to suddenly complete in about 3 hours.

Joins

You can join tables together that have relations between each other, so that you can operate on data across those tables. For instance, I want the titles of all posts that have published comments.

```
SELECT posts.title FROM posts JOIN comments ON posts.id = comments.post_id WHERE commen
ts.published=1;
```

(/)

(Note: each post on that query will appear as many times as it has comments, but let's focus on the join for now.)

Performance is dramatically better if you manage to get the database to do most of the work, as opposed to your application, because the database knows most about your data and how to handle it most efficiently. Joins are amazing wins for that, because the other way to get it done is to perform many separate SQL queries, and manipulate that data in your code, which is very inefficient.

Note: you can join tables together across many relations. The largest join in my career was 7-fold, in a database at Apple that contained information about localization projects.

A NoSQL kind of database: document-based databases

One particularly popular type of NoSQL database is document-oriented databases, such as MongoDB or CouchDB. One reason they're popular is because their learning curve is very smooth, and they feel natural to use: you just send them JSON documents, much like we've done in the "Relational databases done wrong" project, and they make it right when you need to fetch them back. You don't need your JSON documents to have specific fields of specific types, just send whatever JSON you want; the technical word for this is that they are *schemaless*.

One caveat is that they're much, much harder to scale than relational databases (the data being more "formatted" in relational databases makes it easier and faster to work with).

Another caveat is that there is some comfort in having the database enforcing a schema (proper columns of proper types, …); if the database doesn't do it, you can expect that some JSON documents in the collection are not of the schema you expect, and then you have to enforce schema in your code, which means more work. As a result, some document-based databases offer ways to enforce some schema, but I don't believe many developers use it, because it defeats the purpose of having schemaless storage.

Just as relational databases, document-based databases offer a variety of extra features to tune your usage of the data: indexes, joins, … sometimes even relations!

Document-based databases will be covered towards the end of year 1.

Another NoSQL kind of database: key-value stores

Some applications may need very large key-value storage, which you may think of as the persistence of a single huge "dictionary" structure (the same structure that Ruby calls "hash", Python calls "dict", PHP calls "associative array", Objective C and Swift calls "dictionary", …). An obvious need for that is around caching (if

you don't understand why, we'll cover this when we talk about caching). Cassandre, memcached and Redis are popular key-value stores.

As your collection of key-values grows, you may need pretty advanced ways to organize them (and expire them, for instance), so, obviously, each key-value storage solution comes with more advanced tools than just the usual CRUD operations.

At the intersection of NoSQL and relational

As mentioned before, NoSQL databases sometimes get closer to relational databases by allowing to be queried using the SQL syntax (like Cassandra and Hypertable); but databases are getting closer also the other way around, as relational databases themselves have started offering some document-based storage.

A mature example of that is PostgreSQL's "hstore" type, which allows to store JSON data in PostgreSQL, in a way that is queriable. Most recently, this has allowed PostgreSQL to have a certain leg up against their competition of open-source relational databases, because MySQL hasn't been able to ship a similar feature yet, although they're expected too (MySQL development has dramatically slowed down now that they belong to Oracle, which is a direct closed-source competitor; a few years ago, most MySQL contributors went ahead to create another open-source database called MariaDB, which never really became mainstream, so maybe there won't ever be document-based storage in MySQL, actually).

What NoSQL storage do I need?

NoSQL databases address all kinds of requirements, and therefore the ways they work are dramatically different. Here's a really accurate map of the various solutions: http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

Note: in year 1, your main project must be done using a relational database, and we'll cover document-oriented databases (probably MongoDB) and key-value stores (probably Redis) towards the end of the year.