ECE356: Database Systems Winter 2018

Lecture 35 — NoSQL

Jeff Zarnett

NoSQL

NoSQL, well, originally "non SQL" or "non relational", is about data storage that isn't your typical relational database. Now it's "Not Only SQL", because some of the alternative databases can operate on SQL or SQL-like query languages (but that's sort of not what they are for). These days, NoSQL has become very popular and I sometimes have Fourth Year Design Project groups come to me and tell me they want to use NoSQL on a single user app that runs occasionally... Does that make sense?

If you ask them their motivations, such students will say something along the lines of, NoSQL is faster. As we will soon see, it can be but nothing comes for free. In any case, how much any particular app needs speed is an open question. Speed isn't everything.

In reality, the right place to start is assuming you want a relational database, that is, a typical SQL based system. There are a lot of advantages to this and they tend to scale pretty well for a typical workload. As we proceed through this discussion we'll see what the tradeoffs are and why we might be willing to give up the advantages of the relational database because we, more or less, have no choice.

The primary motivation for wanting to go to NoSQL should be scalability. That is to say, storing very large amounts of data, handling heavy workloads, and making data accessible to users wherever they are. When we say a lot of data we're talking about things like Twitter, Facebook, Uber, and other services that have astounding numbers of messages and millions if not billions of users (or, at least user accounts... there is a distinct possibility that a large fraction of Twitter and Facebook accounts are actually bots).

Let's imagine that you do have an application that really does have huge amounts of users or operates over a lot of user data, or both (e.g., Amazon). Doing something like NoSQL might let you do it and have higher speed and higher availability, but there are costs.

Let's talk about the CAP theorem, also known as Brewer's theorem, which says that there exists an iron triangle in distributed data stores. The iron triangle, you might remember from a discussion about project management, is "fast, cheap, good: pick two". The CAP theorem says that it is impossible to get more than two out of the following guarantees: Consistency, Availability, Partition Tolerance [GL02]. To define these things:

- **Consistency**: Every read receives either the most recent write, or an error.
- **Availability**: Every request gets a non-error response, with no guarantee it contains the most recent write.
- **Partition Tolerance**: The system can continue even if messages are lost between nodes.

The iron triangle is actually perhaps not the best analogy. It's not as if, at the start of the system or even during the design phase, you just choose what two items you want. No, actually, instead of 2 out of 3, it's choose either availability or consistency (maybe the descriptions gave you this hint). This is because there *will* be network failures or at least delays, meaning there is partitioning, even if it is temporary. Then it's a question of what happens: if reads can happen before all notes are updates, we get availability; if systems require locking all nodes before allowing a read, we get consistency [Mes13].

For the most part, NoSQL databases do not provide ACID transactions. Instead, they may offer what is called *eventual consistency*. This is to say that eventually, after some nontrivial period of time, the data will reach consistency. This is something you might have experienced (more or less) in applications like Facebook; if someone posts something you may not see it eventually as it could take some time for that thing to propagate its way through the network to you. Now, the problem is you might never catch up, because the content is constantly being updated. People are posting new pictures of their dinner at every moment.

So instead of working on the basis of relations with tables and keys and whatnot, how do NoSQL databases work? There's no simple, single answer to this because there are a lot of options. Remember, this is about things that are not standard relational databases. There are some options that support multiple modes, but we'll choose to focus on just two things for now: key-value databases, and document databases.

Key-Value Databases. Surely at this point you are familiar with the idea of a key-value pair: in data structures and algorithms you learned about a Map (HashMap) for example that operates on the basis of `get( key )` and `put(key, value)`.

Key-value pairs are a recommended way to store small amounts of data for things like Android applications. If you want user preferences in an app to be saved so it is remembered when you open the app again, this is one way to do it. And you can put arbitrary things in there, so you could save the configuration as XML or even just put the (serialized) data structure in there. As long as you know what it is and how to interpret it, you can get it out again. This is small-scale, obviously, but the idea has merit.

Now that idea is applied to a larger amount of data. Instead of defined tables with specific attributes, you have an arbitrary key and an arbitrary value. There may be no restrictions on the form of the key or the form of the value; they can just be bytes of any length with no specific format, rules, or limits. This allows you to store anything you like... pdf documents, Java objects, flat text, comma-separated data... anything.

This scales well, since the only operations are get (read) and put (write) and there are no complexities related to searching, foreign keys, transaction management, et cetera. Eventual consistency can be achieved when every location, sooner or later, gets the update of the data. Put operations simply overwrite old data, and data elements (values) have no formal relationships to one another. You can create some ad-hoc ones by storing under one key a list of other keys, but this is informal.

Document Databases. A document oriented store is somewhat more structured than the simple key-value pair. Instead of just treating the value in they key-value pair as an opaque series of bytes, we might analyze it and do something with it. Based on the structure of the value, as a document, it might be possible to extract some (meta)data about the document and then use that for something else (e.g., search).

We could store lots of documents in the database in formats like XML, JSON, or binary data like PDF or Word documents. The documents themselves don't need to be formatted in any specific way, follow the same format or conform to any schema. And the content is completely arbitrary. If a document contains five XML tags, it has only those and there's no need to put nulls for XML elements that don't exist in that document. And, of course, no need to convert it from XML to an insert statement.

The basic operations are generally defined as CRUD [Mar83]: **C**reate (insert, put), **R**etrieval (query, search, get), **U**pdate (edit), **D**elete (remove). These can come under other names but this acronym is well known. These could be translated to get and put operations, but CRUD is a more friendly API that makes well to use cases.

Aside from the convenience of CRUD rather than get/put, the major advantage is the analysis that allows searching or perhaps better query performance. If we wanted to search for documents with a certain element, e.g., find all documents relating to a particular Tax ID, then having done the meta-analysis we could (maybe) find them relatively quickly if we have an index. If we are willing to have a bit more structure, we could allow differentiation of the data. Suppose the documents are e-mail and we have a bit of structure. Then if we want to search for "abc@xyz.com" we could find e-mails where that address corresponds to the recipient but not the sender.

A friend and database administrator referred at one point to a document database store as a "Datengrab", German for "Data Grave". It's where documents are placed when the die. If they ever come out it's nice, but admittedly in this particular use case, if the document happens to fall off a cliff it can always be regenerated from the source.

References

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[Mar83] J. Martin. *Managing the Data Base Environment*. A James Martin book. Pearson Education, Limited, 1983. URL: `https://books.google.de/books?id=ymy4AAAAIAAJ`.

[Mes13] Lior Messinger. Better explaining the CAP Theorem, 2013. Online; accessed 28-November-2017. URL: `https://dzone.com/articles/better-explaining-cap-theorem`.