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. We can scale them up and we can scale them out, as we have discussed in the distributed databases section. 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.

Motivation. The primary motivation for wanting to go to NoSQL would likely 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).

An upside of NoSQL databases is that they scale horizontally quite well. Unlike in SQL there is no need to do some magic to get a single server as big and fast as possible or to manually scale it across multiple servers. The MongoDB (one of the NoSQL vendors) documentation will tell you that one of the major advantages of the NoSQL approach is auto-sharding: no need to set up different database instances, get them to talk to each other, and the application probably needs be modified to work with multiple database locations. NoSQL, however, allows automatic sharding: they natively and automatically spread data across an arbitrary number of servers.

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). Users do both reads and updates: they look at products on Amazon and products can be purchased and can also be sold out. Doing something like NoSQL might let you do it and have higher speed and higher availability, but there are costs.

Tradeoffs. What we want to do is ultimately limited by 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 nodes are updated, we get availability; if systems require locking all nodes before allowing a read, we get consistency [Mes13].

Let us consider two quick examples in the context of booking a flight. If we choose to prioritize consistency and there is network partitioning, then we would refuse to allow any updates like seat selection until such time as all servers are in agreement and the seat can be locked on all nodes. If we choose availability, then in the event of partition we allow the person to choose the seat anyway, which may be based on stale data. In that case, you might have let two people choose the same seat and there will be a need to sort this out later. For something light booking flights we might think it sensible to choose consistency over availability, because there is only one seat 32D on the airplane and it isn't exactly the same as any other seat. What if instead it was online shopping and you want to buy a book? One copy of the book is as good as any other, isn't it?

To give a real-life example of prioritizing availability over consistency, consider an online shopping scenario that really happened to me in November 2017. I was looking to buy a shirt and sweater, both of which were listed on the website as being available in the colours and sizes that I wanted. After suffering through the insufferable checkout process I received an e-mail telling me that I had successfully placed the order and the items would be shipped. Then, a while later I received another e-mail telling me they don't actually have the items and can't ship them and they're really sorry. And their website still showed the items as available. This is slightly embarrassing for the company, but it does happen.

For the most part, NoSQL databases do not provide ACID transactions. What you get is sometimes called BASE: Basically Available, Soft State, Eventually Consistent [Way11]. The first two points are fairly obvious, but the new part in that acronym is *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. Even so, that's not really a problem because the information will get to you eventually, even if you are constantly trying to catch up, you will never be THAT far behind, and also, it is not super important whether or not you learn whether your friend Terry ate that artisan heirloom radicchio.

As a developer it can feel like freedom to no longer have to worry about designing database tables and worrying about adding columns and thinking about foreign keys. The rules are just slowing you down, making you do a bunch of boring stuff, forcing you to talk with a database administrator about adding a column here or changing a data representation... Although there are probably some rules in organizations that you can do without, there are reasons for some of them that are valid and should probably be observed.

Structure helps in a few ways. It standardizes data representation where possible. If there is an address table in the database and different developers want to use it in different scenarios, they can re-use the already existing structure and you don't get three different implementations, all subtly different ("zipcode" vs "postalcode" vs "postcode"...). Moreover, if you make some changes in a SQL database to a table you will need to think about the migration strategy: if a data type will be shorter or a new field will be added or a structure changed then some way of converting the data from the old format to the new is needed. And this problem can be solved in NoSQL, of course, by doing it in the application: when you load a record, update it to the newest format, perhaps? Or write something to crawl through the data to update all the elements that need changing? In a relational database you're kind of forced to think about it because the alter-table statements may make it obvious that default values are needed.

Some of the NoSQL databases get speed by deleting some features that were very important throughout the course. There may be no joins. Really, no joins at all. And why would there be if there are no tables and therefore no good ways to relate two data elements together? If there are no joins then related data has to be stored together, namely in de-normalized schemas. This is exactly what it sounds like: it's a schema that is not in one of the normal

forms that we discussed. On the contrary, data is duplicated and things that should probably be separated are kept together. Because it's fast, you see; joins are slow so let's not have joins!

Thing is, though, joins enforce consistency, so we might lose consistency in our database as a result of this. Duplicate data, or multiple entries of the same data that are all slightly different (Jane Doe, Jane R Doe...) any one of which you might get back in a particular query. For some things you may not care all that much – if it is something like keeping track of how many megabytes of data the user has used this month then "close enough is good enough"... Maybe.

Some NoSQL implementations do allow joins but they don't work quite like your typical SQL join with join attributes defined in a query... You can, if you want, write your own sort of join, where you do some sequential lookup: first look up record x which contains a way to find related record y... This might be acceptable for finding an individual record, but to do so for 10 000 records is tedious. So perhaps you need to write your own join? At least we learned how those algorithms work...

But then, why are you trying to reinvent the wheel? Relational database vendors have poured a lot of time and money and effort into optimizing every part of the database engine, including joins and fetches and the like. Even sometimes strange things like trying to time your code execution with the position of the read arm of the hard disk drive to micro-optimize a read or write operation [Way12].

Another way that NoSQL might work is it might not have transactions at all. Things happen, or don't, or might be halfway completed. It's up to you as the application developer to try to do things atomically or have some sort of failure recovery (rollback) mechanism. We'll consider, later, a case study about the Oracle NoSQL product that has something resembling a transaction.

Types of NoSQL Databases

As you might already know, the variants of SQL that different databases speak can cause some problems: you can't necessarily take a query written for MySQL and use it in a Postgres database. It might be necessary to convert it because keywords are a little different.

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 a few things for now: key-value databases, document databases, column databases, and graph databases.

Nevertheless, NoSQL tools do not have any sort of standardized language between them. Every vendor has their own idea about what is best and how things should work. In some cases, the queries are expressed as functions or they can be written in a query language that has some similarities to SQL queries. As a result of this, the costs of switching between tools can be high; significantly higher than it would for switching between SQL database providers.

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.

Another thing that we don't get in this situation is aggregation or other "magic" the database can do for you. Remember that you can ask a relational database to sum a certain thing for you: how much income was generated in July of 2016? This is a select with aggregation and we can load just the data we need and crush it down to a single number (385 000) and transmit that result. In a key-value store, no such functionality exists. You can query the data that meets your certain criteria, but you have to do your own addition in application code. That itself isn't necessarily bad but your performance might be very slow if you have to load a large amount of data from the database and send it to the application. That communication cost can be killer depending on the volume of data, the connection type, connection speed, and so on.

Column Databases. Column databases are pretty much like key-value pairs with a slight bit of structure tacked on to the value part of it. In this case, a value has three components: name, the content ("value"), and a timestamp. The name is really the key; obviously, the value and timestamp are stored under that key. A column is then a logical view that encompasses, key, value, and timestamp all together. The timestamp is used if we ever need to differentiate between different versions of the data and in case of recovery.

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 or convert the output of a select statement back into XML in the future. While there can be some tools that do that for you automatically so that you don't have to write the conversions manually, the work done is nonzero.

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 they 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. This is, interestingly, a case where both things are used: there is a relational database that is used for most data, but documents are put into the non-relational document database. The choice is not binary.

Graph Databases. In a graph database, a graph structure is used: there are nodes, edges, and properties. Nodes and edges are first-class entities. What we would normally consider an entity in a relational database is modelled as a node. What is modelled as a relationship in the relational database is an edge. Nodes can have properties, and so can edges. This approach might even seem more natural for certain use cases than the standard relational model approach where everything is converted, eventually, into a table (or attributes within a table).

This supports a lot more ideas about joining, relationships, and constraints, while of course avoiding a lot of the structure imposed by the standard relational database In fact, we have something that looks like the document store model – there are only two kinds of first-class elements (instead of everything being a table) – but we don't necessarily give up the idea of formal relationships between entities [Cox17].

Example: Oracle NoSQL

You have probably heard of Oracle, you know, those guys who make the crazy expensive, enterprise-grade relational databases. They made a NoSQL database, and, well, it's their way of doing things: intended for the enterprise. It is fair to say that the Oracle approach is somewhere between the standard relational database approach and the NoSQL no-rules, speed-is-everything, forget-safety kind of approach.

The Oracle NoSQL database provides something like transactions that are a practical approximation of ACID compliance. Instead of a simple key-value structure, Oracle divides the key into major and minor parts; the major is an object identifier and the minor parts are the "fields" in the record. So instead of just having the value as an impenetrable blob, it has an object with multiple fields and the names of those fields are now just called minor keys. So if a user has some unique ID and then associated things like addresses, the major key is the user ID and the minor keys are the address elements [Way11].

In short you get an ACID "promise" when all writes in a group are attached to the same major key. This makes it suitable for certain types of work: updating a user's personal information and storing it, for example. It is not suitable, however, for a standard bank example of transferring money from account A to B. Because the transaction hits two separate accounts, under two separate major keys, there is the distinct possibility that the transfer will not work as planned and moving \$50 from A to B does not result in the total sum of money remaining the same after the transfer. And while banks probably enjoy the thought of taking \$50 out of your account and not giving it to someone else, there are kind of sort of laws against that.

This promise is fulfilled by the implementation of their NoSQL database: one master machine is guaranteed to hold all minor keys associated with a major key [Way11]. That's actually super convenient because it means that all these minor attributes are on the same node and we can therefore get consistent behaviour through the use of local locks and it all works. But no such guarantee is provided for when the major keys are different because they could be located on different nodes. They might be on the same node and things might work, but there are no promises.

Oracle NoSQL offers both kinds of scaling, replication and sharding. With sharding, your data is spread out and you have less contention (faster writes). If you have replication you get faster reads and higher availability of data (and reliability in some sense). Of course, the more replication you have the longer it can take to write some data, because you do have to get all the systems to agree.

But even that is configurable: you can tell the system what "durability policy" you want to have. Choose if writing successfully to one node enough, or if a simple majority is sufficient, or if all nodes need to agree on the write. Values are associated with a version number which you can use in writing your own sort of replication and durability policy if you so desire [Way11].

Conclusion

NoSQL has lots of advantages and disadvantages when compared to standard relational databases. If used in the right situation, it can speed operations and provide a lot of scalability and availability. If used in the wrong situation it either wrecks your application/company, causes endless headaches, or forces you to reinvent the wheel and implement a lot of features that relational databases come with out of the box. Start with the idea of using a relational database, and consider carefully what you would be losing if you went to a NoSQL approach. If it's the right call for this need, then, by all means. You have the power and the choice is yours. Choose wisely.

References

[Cox17] Graham Cox. Introduction to graph databases, 2017. Online; accessed 30-November-2017. URL: https://www.compose.com/articles/introduction-to-graph-databases/.

[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.

[Way11] Peter Wayner. First look: Oracle NoSQL Database, 2011. Online; accessed 30-November-2017. URL: https://www.infoworld.com/article/2621199/database/first-look--oracle-nosql-database.html.

[Way12] Peter Wayner. 7 hard truths about the NoSQL revolution, 2012. Online; accessed 29-November-2017. URL: https://www.infoworld.com/article/2617405/nosql/7-hard-truths-about-the-nosql-revolution.html.