An Analysis of NoSQL Databases: Uses, Implementation, and Architecture

Harrison Groll

I. INTRODUCTION

As a database administrator, one of the first questions that is often asked is what should my back-end look like. Two significant architectures include the use of a relational database, taking advantage of the structured query language (SQL), as well as a non-relational or Not-only structured query language (NoSQL) methodology. While SQL remains the de facto school of thought for many database implementations, it is necessary to take a good look into its non-relational counterpart. Indeed, NoSQL has a plethora of real life implementations for back-end design and, as we will discuss, is the favored architecture in many use cases where SQL is either not preferred or is unable to handle the data in question. The following paper will provide a thorough analysis of the NoSQL methodology by examining its uses, implementation, and architecture. This will include an in depth comparison between the SQL and NoSQL design. Lastly, real-world examples will be provided throughout to buttress claims of the two database deigns as well as to solidify abstract concepts into everyday architecture design problems.

II. THE (NON) RELATIONAL ARCHITECTURES

A. SQL and The Relational Approach

Developed in 1970 by IBM, SQL has been able to persist many decades as one of the most widely used back-end design implementations. Utilizing a relational design, SQL allows for a concrete system to be built upon relations that exist within a database. These relations, in turn, allow for normalization at many levels. For example, suppose a back-end engineer is implementing a relational database for a typical college. Within the college, students exist in the database with attributes including but not limited to name, date of birth, social security number, and so on (in the form of a Student table). Moreover, suppose that within the school, each Student can enroll in one or more Courses (Course table). Suppose Student named "John Smith" enrolls in Course named "English 1". How might this be presented in a database? A non-relational design might require mass amounts of repetitiveness with respect to each record. Suppose that each of the Students information is repeated at least twice in the database, one for the Student Table and another to record each Student in a Course. With this design, in the Course table, Student John Smith would be required to have all of his profile information repeated in order to include all given attributes (name, date of birth, etc). One could easily imagine the sheer number of duplicate data that would quickly fill up computer memory. Moreover, to make matters worse, imagine the complexities that would arise if one record were to be updated! Suppose John Smith has to update his birthday on file from "04/11/1966" to "04/12/1966". Each record that exists with John Smith's information would have to be updated as well (linear growth that yields X number of updates). This design is not only grossly inefficient, but also prone to many issues. A simple mistake (i.e. forgetting to update one of X records of John Smith) will result in inconsistency of data within the database at hand, moreover, it need not be said that a database holding inconsistent data is essentially meaningless. This instead can all be avoided by adopting best practises of a relational database design. With this, we need only to have our data exist in one place in the database by establishing predefined relationships within our database. For example, instead of having again all our data in a Course table, we can simply have a reference in our Course table with a foreign key that would be rooted to a particular student in the Student table (see fig. 1).

TABLE I
COURSE TABLE

courseID	courseName	studentID
101	"Newtonian Mechanics"	359
102	"English 1"	349

TABLE II
STUDENT TABLE

studentID	studentName	dob
359	"Sarah Greene"	10021998
349	"John Smith"	04121966

Fig. 1. Through the relational model, we need not repeat records in the database, but rather a "reference" or foreign key that implies a relation to another record in the database. Here, it is clear that student 349 is enrolled in course 102 and that student 349 is John Smith.

By implementing a relational design, we immediately reduce any and all data replication. Any updates in our Student table, for example, will automatically effect all references in other tables. Additionally, queries written in SQL for relational databases are fairly simple and easy to learn. By understanding the database, it is not difficult to come up with easy to read statements in order to get the data that you are looking for.

SELECT *
FROM Student s, Course, c
WHERE s.id = c.id
AND s.name = ”John Smith”; (1)

It logically follows then that SQL through normalization serves to better data integrity and data consistency so, it begs the question: why not always use the relational model. As we will see in the next section, there exist certain situations in which the relational model is inefficient and/or impossible to utilize.

B. NoSQL and the Non-Relational Approach

All seems well with our current relational model, no? We have managed to eliminate just about all our data redundancy, whilst upholding data integrity, memory, and, as a result, computational run-time. So why not always use the relational model. As we have mentioned before, the relational model requires that we establish a predefined relationship among our data in order to bear the fruits of the relational model. What if however, the data at hand exists in a way where unlikely/impossible that we could predict any sort of concrete relation? On that same token, imagine if our database was implemented not for a Student and a Course (see previous section), but rather for a Student in one of many Courses in one of many national Colleges. The data at hand could be too complex and overwhelming to implement a seemingly infinite number of relations. Moreover, it would be impossible to know the computational logic that exists for each and every College in the country and queuing a specific Student could be computationally expensive. Enter the non-relational model. NoSQL grew in popularity among back-end designers where it would otherwise be difficult or impossible to map out the ”business logic” of the data at hand. Similarly, imagine also a scenario where one could not accurately predict the data that would be collected through a database administrator or application programming interface (API). In the previous section we explored the case of a Student in a Course. With this example it is easy to imagine a particular data type that would be declared (i.e. date of birth could be type dateTime, SSN could be INT, etc), however, but what if the data at hand was not as black and white. Let us put forth an example of a cooking website where users can enter their favorite recipe to share with others. Imagine the data types that we would need to implement. Perhaps all measurements could be recorded in cups, although it wouldn't make much sense to record 2 table spoons in terms of cups, no? We could instead have a table in our database that has rows of different measurements serving as foreign keys to a Measure table (i.e. cups, teaspoons, ounces, etc); however, what if there are custom measurements that we couldn't possible account for. Specifically, what if a recipe asks for a ”sprinkle”, ”dash”, or ”touch” of salt. It would indeed be difficult to implement such a system that could cater to every lingo. While this is just one example, it is easy to foresee potential pitfalls that could arise due to unpredictable data types. By implementing a NoSQL database, we could leave this question open-ended by taking advantage of a user-defined data-type. By leaving the relational model by the door, we equip ourselves with the tools we need to face head on with large semi-structured and unstructured data. Moreover, this serves doubly well when the data at hand is prone to constant changing and large, complex relations simply won't do.

III. Architecture

Now that we have the basics down, let us explore in detail the inner mechanism of the NoSQL implementation as well as compare it to the relational model. On a macro-level, there some of the more common NoSQL implementations include: key value, document, wide column, as well as graph. Each of these types has their respective gains and losses, and each rises as the superior model under their optimal case(s). Similar to a hash-map or dictionary, the key-value database is regarded as the simplest of the NoSQL implementations. Each key in the database is associated a a corresponding value, allowing for very quick look-up speeds when working with large data sets. This can prove to be useful in situations ranging from those that include a product and price on implementation to a comment/user relationship in a database. Moreover, examples common key-value database management systems include Redis, Aerostrike, and Oracle NoSQL Database.

$$349- > ”JohnSmith”$$

Next, building upon the key-value design, the document model acts as a collection of key-value pairs and is able to store a more complex series of data (i.e. lists, arrays, etc). This proves to be particularly useful with data transferring throughout the web (JSON, XML, etc):

$$TheDocumentModel$$

id: 32,
name: ”Salt”,
amount: ”dash”,
quantity: 2

With this implantation, it is easy to see just how flexible our database could become when dealing with measurements in our recipe database (see section II.B NoSQL and the Non-Relational Approach). Indeed, we could create a value for ”quantity” and ”measurement” within our database, allowing it to hold the values of anything our users might throw at it [quantity: 3, measurement: ”dash”]. Additionally, one significant difference between the object and key-value database is that the former takes advantage of meta-data, thus making it easy to query an object within our database. If, for example, we wanted to query our database to find a recipe that contains salt, the object model could complete this with relative ease. One might consider using this implementation in situations

that include but are by no means limited to back end sites for user accounts, service requests for a particular office, and/or fun facts associated with characters in Star Wars serving via API. Popular implementations include ArangoDB, BaseX, and Cloudant.

Next, we have the wide column implementation. Although it in many ways may seem the most similar to a typical relational database through its terminology (there exists tables and columns), it important to note that it is not the same thing and its differences will be discussed. Unlike the standard relational model, the wide columns design does NOT take advantage of normalization (its data is denormailzed) and its columns are not fixed. This allows for more flexibility with our data in that we do not require relations (it goes without saying that we do of course do not get those same benefits of the relational model). Additionally, unlike the relational model, we can allow for our values to be complex data structures, again building upon the flexibility of the model. MongoDB, Azure Table Storage (ATS), and Riak all work to utilize a common wide column database. Overall, wide column is ideally designed for large volumes of data, read and write performance, and high availability.

A fourth implementation of NoSQL related to the use of graphs. These databases use graph structures for semantic queries with edges, nodes, and properties in order to properly represent and store the data at hand. Neo4j and InfiniteGraph are both popular outlets for implementing a graph database. Graphs can be used for their flexibility, performance, and agility. As such, graphs are commonly used in situations that involve social networks, recommendation, and penalization.

IV. Closing Remarks

This paper was opened with a question regarding the proper back-end design to use. Hopefully upon reading this paper, the nuances between the different possible implementations have been made clear. Overall, relational databases shine when the data at hand is unlikely to change (and the relations between them for that matter). One the other side of the isle, NoSQL databases are used most often in cases of large, complex data sets that is largely non/semi-structured. While the relational model is rigid, well defined, and largely structured, the non-relational model is dynamic, varied and, in many cases, unstructured. Scalability with the relational model is met with larger (more expensive) servers that reinforces data consistency, although is vulnerable in that it has a single point of failure. Scalability with the non-relational model is met with more (quantity of) servers (less expensive), resilient, and balanced requests. It is for these reasons that the relational and non-relation model is said to scale vertically and horizontally, respectively. Through studying these implementations it is clear that there is no objective 'winner' with respect to database design. Instead, we must acknowledge that no one design is perfect and that there will always be trade-offs; however, by taking the time to study the pros and cons of each design, we can make informed decisions that reflect the data at hand as well as the computational logic expectations of use.

R

References

[1] InfoWorld- Why you should use a graph database https://www.infoworld.com/article/3251829/nosql/why-you-should-use-a-graph-database.html

[2] inLearning- Advanced NoSQL for Data Science https://www.linkedin.com/learning/advanced-nosql-for-data-science/advantages-of-nosql-databases

[3] Neo4j- Graph Databases for Beginners: Why Graph Technology Is the Future https://neo4j.com/blog/why-graph-databases-are-the-future/

[4] 3PilarGlobal- EXPLORING THE DIFFERENT TYPES OF NOSQL DATABASES PART II https://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases