ARTICLE 10

NoSQL Databases

Svetlozar Nestorov
Loyola University Chicago

Abhishek Sharma
Loyola University Chicago

Sippo Rossi
Hanken School of Economics

In this article, we will give a brief overview of NoSQL databases. The origin of the term "NoSQL" is the phrase "not only SQL." **NoSQL databases** are those that are not based on the relational model and do not use SQL as a query language. Due to this broad definition, NoSQL databases are not one specific type of database. Instead, the term encompasses a wide range of database designs with varying architectures and query languages used to interact with them. Moreover, some nonrelational databases are commonly simply referred to as NoSQL databases, while others are primarily referred to by their more specific name, such as vector databases. However, for simplicity, in this article, we will describe nonrelational databases using the term "NoSQL."

In addition to not being based on RDBMS technologies, one of the main differentiating features of a NoSQL database is its flexible and extensible data model. In relational databases, a database schema is defined in advance and captures the semantics of the elements in the database. In NoSQL databases, data does not have to be stored in structures described by an organized database schema. For example, data in many NoSQL databases is organized into simple key-value pairs. In a key-value pair, the instance of data that is stored in the database is a value that is given a key to be used for data retrieval. Whether the values are simple (e.g., a string or a number) or arbitrarily complex structures with their own semantics, the values are processed by applications outside the database system and not the database system itself.

NoSQL databases are increasingly being used due to the emergence of big data and due to web applications requiring flexibility and performance that cannot be achieved with relational databases. While the popularity of NoSQL databases has increased substantially since the early 2010s, there are no signs that they are replacing traditional relational databases. Rather, NoSQL databases have become a complementary approach to storing data in specific use cases and can be found in the back-ends of many applications and data lakes alongside relational databases.

In this article, we will primarily focus on describing this technology using two of the most common NoSQL database types, key-value stores and document stores. Moreover, this article provides practical examples from two of the most popular NoSQL database management systems, MongoDB and Redis. Later, we will briefly also outline other NoSQL database types such as graph databases, vector databases and wide-column stores. Lastly, we provide examples of the applications of NoSQL databases.

Terminology of NoSQL Versus Relational Databases

To provide some context to NoSQL databases, we will examine some of the common database terms as they are defined in most NoSQL database implementations, and compare those with the relational database definitions.

"Database," as a broad term signifying organized collection of data accompanied by metadata, is used in the same fashion in NoSQL and relational databases.

Whereas the main construct in a relational database is a "table" (relation), the equivalent construct in NoSQL databases is a "collection." In relational databases, tables first must be created, and only after that can they be populated. Creating and populating tables are two separate actions. In contrast, in NoSQL databases, the metadata and data for collections can be created at the same time.

In relational databases, values are stored in table "rows," and therefore each table contains a set of rows. In NoSQL databases, each collection contains a set of "documents." Whereas all rows of a table have the same structure, the structure of the documents in any particular collection is not required to be the same, although in most cases, the structures are similar.

In relational databases, the structural metadata of each table is represented by a "column." In NoSQL databases, the structural metadata of each collection is represented by a "field." The columns of relational tables have to be created prior to insertions of data. In contrast, each insertion of data in a collection in NoSQL can be accompanied by creation of one or more new fields for a document. Also, insertion of data in a collection does not have to use all the previously created fields (which is equivalent to a table where no columns are mandatory). The values for fields that are not used by a collection can be completely omitted in insertion statements, whereas in insertion statements for relational tables, we still must explicitly indicate that a particular row has no value in a particular column, or default values have to be specified in advance.

Every relational table must have a "primary key," which can be either a column or a set of columns. NoSQL collections have a primary key as well, but it is system-generated in most cases. In some NoSQL software, a relevant column can be used as a primary key instead of these system-generated primary keys.

NoSQL Database Examples—MongoDB

We will illustrate the data model, query language, and extensibility of NoSQL databases using a simple example of a hotel business-related database in one of the most popular and widely used NoSQL databases, MongoDB. MongoDB is a document store-oriented NoSQL database, which stores data in a JSON-like format.

Consider the following example of an online company that collects information about hotels. The data collected and presented on its website contains hotel-related information, such as rates, room types, and amenities. This company could design a relational schema and develop a relational database to capture this information. However, in this example, this company did not decide in advance what information they would collect for hotels, so they will proceed with collection of data without a schema in advance, and they will use MongoDB.

In MongoDB, such a collection of hotel documents can be built incrementally without having to modify any schema, since there is no general schema that applies to every document representing a hotel. For example, a document can at first contain the most basic hotel info, such as name, address, and numeric rating. The statements here show the creation of a single document (in a collection "hotels"), which is saved in the database:

```
db.createCollection("hotels")
db.hotels.insertOne(
    {
        name: "Metro Blu",
        address: "Chicago, IL",
        rating: 3.5
    }
)
```

Similarly, multiple documents can be inserted simultaneously as shown here:

```
db.hotels.insertMany([
    {
        name: "Experiential",
        address: "New York, NY",
        rating: 4
    },
        {
          name: "Zazu Hotel",
          address: "San Francisco, CA",
          rating: 4.5
    }
])
```

Note that even though the primary key is not mentioned for any of the three documents, MongoDB creates a unique system-generated primary key (`_id`) for each of the documents. However, if needed, MongoDB also allows users to explicitly set the primary key.

At this point the database contains a collection called hotels with three documents. All such documents can be found in the database by issuing the following query (equivalent to the `"SELECT *"` statement in SQL):

```
db.hotels.find()
```

Note that the schema for the hotels collection and the documents in it did not have to be declared or defined. In order to add a new hotel with an unknown rating, a new document without the rating field can be created as follows:

```
db.hotels.insertOne(
    {
        name: "Solace",
        address: "San Francisco, CA"
    }
)
```

All hotels in San Francisco, California can be found by querying hotels:

```
db.hotels.find( { address : "San Francisco, CA"} )
```

This is similar to the following query in SQL:

```
SELECT * FROM Hotels WHERE address = 'San Francisco, CA';
```

If the company decides to add additional information about some hotels, the updates can be applied to each hotel document, as depicted by the following statements:

```
db.hotels.updateOne( { name:"Zazu Hotel" },
    {
        $set: {
                wifi: "free"}
    }
)
db.hotels.updateOne( { name:"Zazu Hotel" },
    {
    $set: {
            parking: 45}
    }
)
```

These statements added new fields to the *Zazu Hotel* entry, namely, the availability of Wi-Fi (*free*) and the price of parking (*$45*). There is no need to change a schema and migrate the data to the new schema as would be the case for a relational database. While this may seem trivial, consider the following scenario where similar fields need to be added to a relational database. First, this would require modifying the schema of the database using commands to add two columns to the table for parking and Wi-Fi, while specifying the data types and constraints for each. Only after this could the appropriate rows be updated and the data

added to the newly created columns. However, this introduces excessive sparsity, as all other rows where this information is not available (or needed) would now always have empty attributes. Lastly, while in small databases the effort needed for modifying tables would not be much of an issue, in large databases changes to the schema can be difficult, impractical, and more time-consuming.

Another important characteristic of MongoDB is that denormalization is native to it and the data is easily prejoined prior to processing. We will illustrate this with an example based on Figure 1 (same as the Figure 3.16 in Chapter 3 of *Database Systems: Introduction to Databases and Data Warehouses [Edition 3.0]*).

This example creates an equivalent of a prejoined (denormalized) table that contains information about employees and departments that employees report to.

EMPLOYEE

EmpID	EmpName	DeptID
1234	Becky	1
2345	Molly	2
3456	Rob	1
4567	Ted	2

DEPARTMENT

DeptID	DeptLocation
1	Suite A
2	Suite B

Figure 1 Data in a simple Employees and Departments database.

```
db.createCollection("empdept")
db.empdept.insertOne(
    {
        empid: 1234,
        empname: "Becky",
        dept: { deptid: 1,
                deptlocation: "Suite A"}
    }
)
db.empdept.insertOne(
    {
        empid: 2345,
        empname: "Molly",
        dept: { deptid: 2,
                deptlocation: "Suite B"}
    }
)
db.empdept.insertOne(
    {
        empid: 3456,
        empname: "Rob",
        dept: { deptid: 1,
                deptlocation: "Suite A"}
    }
)
db.empdept.insertOne(
    {
        empid: 4567,
        empname: "Ted",
        dept: { deptid: 2,
                deptlocation: "Suite B"}
    }
)
```

In the previous example, we created a collection "emdept" and created four documents that contain information about four employees and their departments.

The query that seeks information about each employee and their department is written as follows:

```
db.empdept.find( { empid : 1234},
    {_id:0,"empid":1,"dept.deptlocation":1}
)
```

The "_id:0" portion of the statement prevents the display of a system-generated ID. The result of this query is

```
{ "empid" : 1234, "dept" : { "deptlocation" : "Suite A" } }
```

This NoSQL query is similar to the following query in SQL:

```
SELECT empid, deptlocation FROM empdept WHERE empid = 1234;
```

The NoSQL query illustrates an example that takes advantage of the fact that the data is "prejoined," since all the necessary information about which employee belongs to which department is precoded.

NoSQL Database Examples—Redis

To illustrate the diversity of NoSQL databases, we provide a small second example but this time using Redis, another widely used NoSQL database. Redis is an in-memory key-value store type NoSQL database. The basic constructs are documents that are identifiable by their key name, and as in the previous example, these documents can have a varying number of fields. While real-world use cases would be different from the provided example, as Redis is used for specific high-speed requiring tasks, we have opted to demonstrate it with the same hotel example as used previously for the sake of clarity.

In Redis, similar to MongoDB, the hotel documents can be added to the database incrementally without having to specify any schema. The statements here show the creation of three documents in Redis:

```
HSET hotel:1 name "Metro Blu" address "Chicago, IL" rating 3.5
HSET hotel:2 name "Experiential" address "New York, NY" rating 4
HSET hotel:3 name "Zazu Hotel" address "San Francisco, CA" rating 4.5
```

As Redis is a key-value store, the key for each collection is provided as the first input, which in this case is, for example, hotel:1. After specifying the key it is possible to provide fields and their values. After inserting the data it is possible now to retrieve values by specifying the key and optionally field(s). We can, for instance, retrieve the name of the first hotel with the following command, which would return the value *"Metro Blu"*:

```
HGET hotel:1 name
```

This is similar to the following query in SQL:

```
SELECT name FROM Hotels WHERE id = 1
```

To retrieve the values of all fields of a key, similarly, it would be possible to use the command here:

```
HGETALL hotel:1
```

Which would return

```
1) "name"
2) "Metro Blu"
3) "address"
4) "Chicago, IL"
5) "rating"
6) "3.5"
```

This is similar to the following query in SQL:

```
SELECT * FROM Hotels WHERE id = 1
```

It is also possible for keys to have different fields within them. For example, we could add a new hotel that does not yet have a rating by leaving out this field:

```
HSET hotel:4 name " Solace" address " San Francisco, CA"
```

In its simplest form, these functionalities, with a handful of other basic commands, are enough to get started with a simple Redis database, as Redis is designed to act as an efficient way to store values behind keys. However, with this setup, it is not possible to query the database without providing a specific key and fields to retrieve. In other words, without additional steps, we cannot query for, for example, all hotel names, or retrieve an entry based on some wildcard. To enable querying, we would need to define a schema, but this is outside the scope of this example.

Other NoSQL Database Types

Previously we described two common types of NoSQL databases, document stores and key-value stores, which are conceptually similar enough and easy to demonstrate alongside an equivalent approach using a relational database and SQL commands. However, some NoSQL databases are designed specifically to tackle limitations in the relational data model and thus can be distinctively different. Next, we provide further descriptions of three other common classes of NoSQL databases: graph stores, vector databases, and wide-column stores.

Graph stores, also known as graph databases, are a type of NoSQL database that is based on a graph structure consisting of a collection of nodes and edges. The nodes hold data, such as key-value pairs, as well as metadata. Edges (also known as relationships) connect nodes, also specifying a direction for the connection. Edges can also have properties and contain information such as a name for the connection. There are several benefits to the database design of a graph database. First, querying linked data based on relationships is very fast. Second, this architecture makes visualizing the schema of graph databases straightforward, as it can be visualized as a network of connected nodes (see Figure 2).

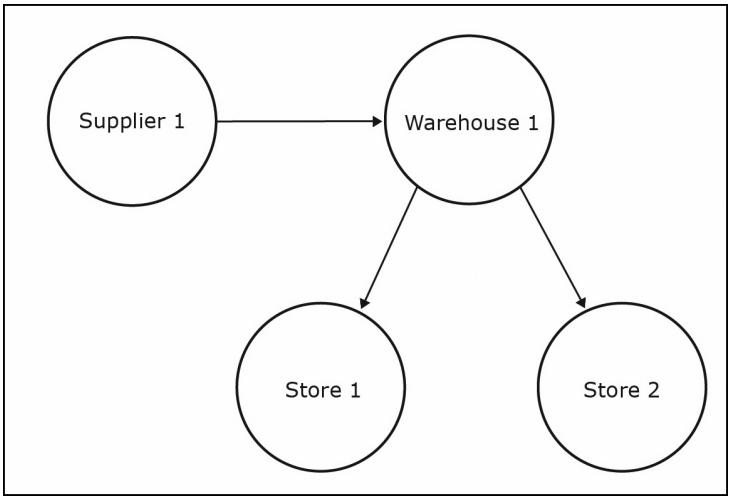

Figure 2 A graph describing the supply chain of a retailer.

As with all other types of NoSQL databases, there exists no universal design for graph database management systems. Thus, there are multiple different commercial and open-source graph database management systems in use, each with its own design and query language. One of the popular implementations of graph database management systems is Neo4j, which uses its own scripting language, Cypher.

Next, we will illustrate how a very basic graph database based on the graph shown above could be implemented with Neo4j using Cypher. First, these statements create the four nodes:

```
CREATE (store1:Store {location: 'Pomona'})
CREATE (store2:Store {location: 'Claremont'})
CREATE (warehouse1:Warehouse {location: 'LA'})
CREATE (supplier1:Supplier {location: 'West Coast'})
```

Now we have four nodes, but to be able to query them, we still need to connect them. This is done by adding edges between nodes, which can be done using the commands shown here:

```
MATCH (s:Store),(w:Warehouse)
WHERE s.location = 'Pomona' AND w.location = 'LA'
CREATE (w)-[r: SUPPLIES]->(s)
MATCH (s:Store),(w:Warehouse)
WHERE s.location = 'Claremont' AND w.location = 'LA'
CREATE (w)-[r: SUPPLIES]->(s)
MATCH (w:Warehouse),(su:Supplier)
WHERE w.location = 'LA' AND su.location = 'West Coast'
CREATE (su)-[r: SUPPLIES]->(w)
```

Now the graph that we have in the DBMS is similar to the one shown in Figure 2. One of the strengths of using a graph database is that joining data from multiple nodes is more straightforward than equivalent queries involving multiple tables in relational databases. Moreover, the queries involving multiple nodes in languages such as Cypher are shorter and more readable than those involving joining multiple tables in SQL. Consider the following example, where data on stores, warehouses, and suppliers are kept in separate tables in a relational database equivalent to the graph database. A query showing the supplier and warehouse of each store could look like this:

```
SELECT stores.storeLocation,
   warehouses.warehouseLocation,
   suppliers.supplierLocation
FROM stores
JOIN warehouses ON stores.warehouseID = warehouses.warehouseID
JOIN suppliers ON warehouses.supplierID = suppliers.supplierID;
```

In our Neo4j graph database, an equivalent query would look like this:

```
MATCH (s:Store)<-[:SUPPLIES]-(w:Warehouse)<-[:SUPPLIES]-(su:Supplier)
RETURN s.location, w.location, su.location
```

This example illustrates the difference in length and readability. Consider a more complex data set and joins involving more tables, and the differences become even more pronounced.

As could be seen from the three examples (MongoDB, Redis, and Neo4j) shown so far, NoSQL database implementations can have very little in common, as the only unifying factor is that they do not follow the relational data model. In this section, we will still briefly outline two newer NoSQL database types, which have found popularity in specific applications.

Vector databases are one of the more recent types of NoSQL databases, which have risen to prominence due to their suitability as a supporting database for generative AI models and recommendation systems. Vector databases, as the name implies, are used to store vectors, which are fixed-length lists of numbers that represent, for example, words in a numeric format. A simple vector, "A," is illustrated here:

```
A = [-0.5969982, -0.33086956, 0.32643065, -0.3570332, 0.628059]
```

Vector databases are designed to store these vectors in a manner that allows efficient querying based on similarity or distance between vectors. As a concrete example, the vector representation of the word "cat" would be placed in a vector database close to related words such as "paw" and "whisker," and the vector database management system would support rapidly locating these words and calculating the distances between them. Several of the most used NoSQL databases, such as MongoDB and Redis, have been extended so that they can be used also as vector databases. Readers can learn more about vector databases by going to MongoDB and Redis's official websites and following their tutorials for vector databases.

Wide-column stores (or column-family stores) are examples of a column-oriented NoSQL database. Conceptually, wide-column stores are similar to key-value stores and can be seen as simply a two-dimensional variant of them, where a value can contain nested key-value pairs. In wide-column stores, data is stored in a so-called column-family instead of rows. The column-families are columns of data that are related and often retrieved together, making querying them fast. An example of a popular implementation of a wide-column store is Apache Cassandra. As with vector databases, readers who want to learn more about wide-column stores can go through online tutorials such as Apache Cassandra's official website's tutorials.

Applicability of NoSQL

Before we elaborate on the applicability of NoSQL, let us briefly consider the environments where RDBMS-based databases are feasible. If the data is structured uniformly and is not massive, relational tables perform very well, while taking advantage of the simple and ubiquitous language (i.e., SQL). Similarly, if data is structured uniformly and is massive, adequate levels of performance can be achieved by using denormalized, indexed relational tables while still allowing the use of SQL. These scenarios, where RDBMS and SQL provide appropriate solutions, cover most organizational operational databases and data warehouses.

As we discussed previously, two main characteristics of a NoSQL database are the flexibility of the data model and the ease with which the data can be completely denormalized, which eliminates the need for joins. Therefore, NoSQL is appropriate for environments that can take advantage of those two characteristics. Typically, NoSQL serves a very important purpose in the storage and querying of massive amounts of data with structure that is not always uniform. Examples of environments that produce such data include social

media giants (such as Facebook, Twitter, etc.) or large online retailers (such as Amazon, Walmart, etc.). Such environments take advantage of the fact that documents of a collection do not have to share the same schema. For example, not every product in the Amazon catalog will have the same attributes. NoSQL accommodates this diversity in a very flexible manner, as illustrated previously.

A typical use case in a NoSQL setting is a search involving a small number of columns (one or several) over a large amount of data, resulting in a relatively small number of rows. Consider, for example, an Amazon.com user searching for a product based on fields such as product category, feature, and/or price. The text of the search query is matched with the search engine's index, and the most relevant items are displayed to the user in the order of the degree of relevance. The user then clicks on the item, and this leads to a read query on a NoSQL database. In online business, such retrievals must be very quick, and NoSQL databases are well suited for this need. Also, the structure of the product-related information is nonuniform and fluid, and that fits well with the use case of the schema-less architecture of NoSQL.

In such environments, the ability to quickly execute searches over massive amounts of data on relatively few columns that produce relatively few records is paramount, and the NoSQL is optimized for such requests. The architecture of NoSQL is entirely driven by such use cases, and each data set is usually only contained in what is equivalent to one table. When NoSQL databases such as MongoDB first appeared, they emerged to provide flexibility for data schema (fields added on the fly) and an alternative to join (which is inefficient with massive amounts of data). Furthermore, NoSQL databases are by design easier to scale horizontally, which allows maintaining high performance via distributed systems even with massive data sets.

As we have discussed throughout this book, for many organizations and scenarios, the ability to store data in a prestructured way is paramount, and for such cases, NoSQL is not an appropriate solution.