Why NOSQL?

If you’ve built web applications in recent years, you’ve probably used a relational database as the primary data store. If you’re familiar with SQL, you might appreciate the usefulness of a well-normalized data model, the necessity of transactions, and the assurances provided by a durable storage engine. Simply put, the relational database is mature and well-known. When developers start advocating alternative datastores, questions about the viability and utility of these new technologies arise. Are these new datastores replacements for relational database systems? Who’s using them in production, and why? What trade-offs are involved in moving to a nonrelational database? The answers to those questions rest on the answer to this one: why are developers interested in MongoDB?

MongoDB is a database management system designed to rapidly develop web applications

and internet infrastructure. The data model and persistence strategies are

built for high read-and-write throughput and the ability to scale easily with automatic

failover. Whether an application requires just one database node or dozens of them,

MongoDB can provide surprisingly good performance. If you’ve experienced difficulties

scaling relational databases, this may be great news. But not everyone needs to

operate at scale. Maybe all you’ve ever needed is a single database server. Why would

you use MongoDB?

Perhaps the biggest reason developers use MongoDB isn’t because of its scaling strategy, but because of its intuitive data model. MongoDB stores its information in documents rather than rows. What’s a document? Here’s an example:

{

\_id: 10,

username: 'peter',

email: 'pbbakkum@gmail.com'

}

This is a pretty simple document; it’s storing a few fields of information about a user. What’s the advantage of this model? Consider the case where you’d like to store multiple emails for each user. In the relational world, you might create a separate table of email addresses and the users to which they’re associated. MongoDB gives you another way to store these:

{

\_id: 10,

username: 'peter',

email: [

'pbbakkum@gmail.com',

'pbb7c@virginia.edu'

]

}

And just like that, you’ve created an array of email addresses and solved your problem. As a developer, you’ll find it extremely useful to be able to store a structured document like this in your database without worrying about fitting a schema or adding more tables when your data changes.

MongoDB’s document format is based on JSON, a popular scheme for storing arbitrary data structures. JSON is an acronym for JavaScript Object Notation. As you just saw, JSON structures consist of keys and values, and they can nest arbitrarily deep. They’re analogous to the dictionaries and hash maps of other programming languages. A document-based data model can represent rich, hierarchical data structures. It’s often possible to do without the multitable joins common to relational databases.

For example, suppose you’re modeling products for an e-commerce site. With a fully normalized relational data model, the information for any one product might be divided among dozens of tables. If you want to get a product representation from the database shell, you’ll need to write a SQL query full of joins. With a document model, by contrast, most of a product’s information can be represented within a single document. When you open the MongoDB JavaScript shell, you can easily get a comprehensible representation of your product with all its information hierarchically organized in a JSON-like structure. You can also query for it and manipulate it. MongoDB’s query capabilities are designed specifically for manipulating structured documents, so users switching from relational databases experience a similar level of query power. In addition, most developers now work with object-oriented languages, and they want a data store that better maps to objects. With MongoDB, an object defined in the programming language can often be persisted as is, removing some of the complexity of object mappers. If you’re experienced with relational databases, it can be helpful to approach MongoDB from the perspective of transitioning your existing skills into this new database. If the distinction between a tabular and object representation of data is new to you, you probably have a lot of questions. You’ll learn the history of MongoDB and take a tour of the database’s main features. Next, you’ll explore some alternative database solutions in the NoSQL2 category and see how MongoDB fits in. Finally, you’ll learn where MongoDB works best and where an alternative datastore might be preferable given some of MongoDB’s limitations. MongoDB has been criticized on several fronts, sometimes fairly and sometimes unfairly. Our view is that it’s a tool in the developer’s toolbox, like any other database, and developers should know its limitations and strengths. Some workloads demand relational joins and different memory management than MongoDB provides. On the other hand, the document-based model fits particularly well with some workloads, and the lack of a schema means that MongoDB can be one of the best tools for quickly developing and iterating on an application.

MongoDB’s key features

A database is defined in large part by its data model. In this section, you’ll look at the document data model, and then you’ll see the features of MongoDB that allow you to operate effectively on that model. This section also explores operations, focusing on MongoDB’s flavour of replication and its strategy for scaling horizontally.

1.2.1 Document data model

MongoDB’s data model is document-oriented. If you’re not familiar with documents in the context of databases, the concept can be most easily demonstrated by an example. A JSON document needs double quotes everywhere except for numeric values. The following listing shows the JavaScript version of a JSON document where double quotes aren’t necessary.

{

\_id: ObjectID('4bd9e8e17cefd644108961bb'),

title: 'Adventures in Databases',

url: 'http://example.com/databases.txt',

author: 'Rutul',

vote\_count: 20,

tags: ['databases', 'mongodb', 'indexing'],

image: {

url: 'http://example.com/db.jpg',

caption: 'A database.',

type: 'jpg',

size: 75381,

data: 'Binary'

},

Listing 1.1 A document representing an entry on a social news site

\_id field, primary key, Tags stored as array of b strings, Images attribute pointing to c another document.

This listing shows a JSON document representing an article on a social news site (think Reddit or Twitter). As you can see, a document is essentially a set of property names and their values. The values can be simple data types, such as strings, numbers, and dates. But these values can also be arrays and even other JSON documents c. These latter constructs permit documents to represent a variety of rich data structures. You’ll see that the sample document has a property, tags B, which stores the article’s tags in an array. But even more interesting is the comments property d, which is an array of comment documents. Internally, MongoDB stores documents in a format called Binary JSON, or BSON. BSON has a similar structure but is intended for storing many documents. When you query MongoDB and get results back, these will be translated into an easy-to-read data structure.

The MongoDB shell uses JavaScript and gets documents in JSON, which is what we’ll use for most of our examples. Where relational databases have tables, MongoDB has collections. In other words,

MySQL (a popular relational database) keeps its data in tables of rows, while MongoDB keeps its data in collections of documents, which you can think of as a group of documents. Collections are an important concept in MongoDB. The data in a collection is stored to disk, and most queries require you to specify which collection you’d like to target. Let’s take a moment to compare MongoDB collections to a standard relational database representation of the same data. Figure 1.1 shows a likely relational analog.

Because tables are essentially flat, representing the various one-to-many relationships in your post document requires multiple tables. You start with a posts table containing the core information for each post. Then you create three other tables, each of which

includes a field, post\_id, referencing the original post. The technique of separating an

object’s data into multiple tables like this is known as normalization. A normalized data

set, among other things, ensures that each unit of data is represented in one place only.

But strict normalization isn’t without its costs. Notably, some assembly is required.

To display the post you just referenced, you’ll need to perform a join between the post

and comments tables. Ultimately, the question of whether strict normalization is

required depends on the kind of data you’re modeling, and chapter 4 will have much

more to say about the topic. What’s important to note here is that a document-oriented

data model naturally represents data in an aggregate form, allowing you to work with

an object holistically: all the data representing a post, from comments to tags, can be

fitted into a single database object.