LINUX VOICE TUTORIAL

CODE NINJA: NOSQL

BEN EVERARD

When data gets big, get NoSQL – it'll future-proof your project and enhance your job prospects too!

WHY DO THIS?

- Understand how huge databases handle billions of transactions.
- Gain the flexibility of not having a schema.
- Use the trendiest database in town.

Before we take a look at NoSQL databases, let's first consider databases in general. Broadly speaking, a database is anything that can store and retrieve data. Most of the common databases use Structured Query Language (SQL) to access and manipulate this data. SQL databases are in the relational class of database. In relational databases, everything is stored in tables, and there are links between these tables known as keys. Each table has a series of columns, and each column has a data type associated with it.

As a quick example, a shop may have a database with tables for customers, orders, invoices, and stock. If you needed to know what items a particular customer had bought, you'd need to link the relevant rows from all the tables to build up a picture of what was going on. SQL makes this linking of the tables very easy. Splitting data up this way means that data isn't duplicated, and so can be easily updated. For example, in this example, a customer's address can be stored in one table and automatically linked to all orders. This means you can easily find out the current address for a customer on an old order.

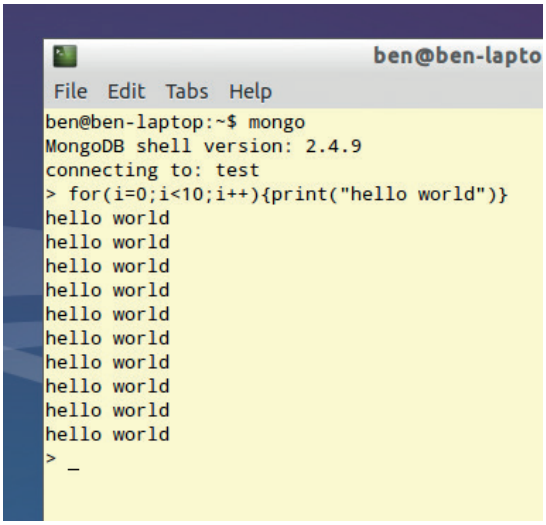
"There are some areas in which relational databases are struggling to keep up."

Relational databases have served the computing world well for a few decades; however, the computing world is changing and there are some areas in which these old-fashioned databases are struggling to keep up.

- **Size** Once a database gets too big to store on a single machine, it becomes complex to store it in a relational database.
- **Performance** At very high transactional levels, the overhead of linking tables together can slow down the database.
- **Flexibility** Changing the table structure can be a complex procedure.

At this point, we should say that the above points are only relevant in the most extreme cases. Relational databases can be very big, very fast (and a little flexible). Unless you're trying to push the boundaries of what your hardware can do, a relational database will probably serve your needs well.

However, if you're starting a new company and hope to be the next Google or Facebook, how do you ensure that your database will scale to a few billion users? The answer is NoSQL. Technically speaking,



MongoDB's JavaScript Shell enables you to create simple programs that interact with the database.

NoSQL can refer to any database that doesn't use SQL, but it's generally used to refer to schema-less databases. These basically consist of one big pot into which you can put any data you want regardless of its format. These are sometimes known as document stores.

Introducing MongoDB

We'll look at one of the most popular of the document stores, *MongoDB*. In this database, data is stored in JSON-style documents. Each document can be put in the store regardless of what format it's in.

To try this out, you'll first need to install *MongoDB*. In Ubuntu and derivitives, it's in a package called **mongodb**, so you can grab it with:

```
sudo apt-get install mongodb
```

Once that's finished, you can run it with:

```
mongo
```

This will drop you into the *MongoDB* shell. It uses a stripped-down version of JavaScript that you can use to build software, but we won't deal too much with that. As a quick example, we'll add and retrieve a couple of items in completely different formats:

```
db.test.save({writer: "Ben", title: "Code Ninja: MongoDB"})
db.test.save({issue: "11", mag: "Linux Voice"})
db.test.find();
```

This automatically creates a database called **test**, then puts two entries in it. The final line retrieves them from the database.

You can also grab particular entries by adding parameters to the **find** function. For example:

```
db.test.find({writer: "Ben"});
```

As you can see, the fact that it is schema-less means you can store and retrieve any information. This means that if your requirements change, you can just put different information in. On the other hand, it means that you can't always be sure what format the data coming out will be. This can have advantages for all sorts of projects. It means you can just start coding your hobby project, and not have to worry about the overhead of changing the design should you with to, and it also means that a multi-million dollar Internet of Things project won't be rendered obsolete in six months when a new device comes out and needs to store different data.

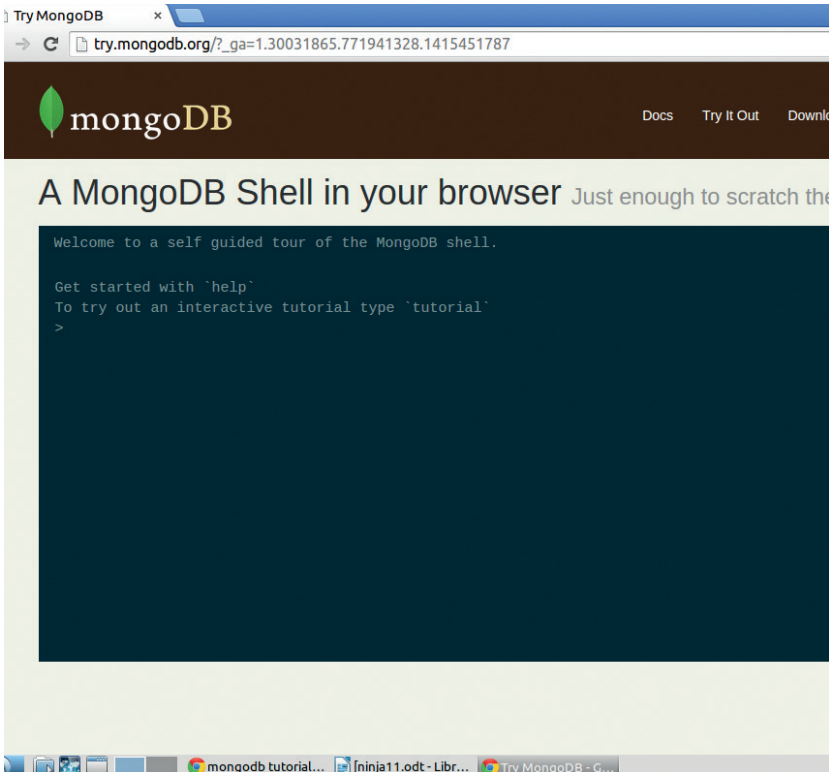
Linking documents

The concept of linking tables isn't completely gone. Even though there are no tables, you can still reference other bits of data in *MongoDB*. Unlike in relational databases, *MongoDB* gives you a choice. You can do this by linking documents or embedding them. Linking documents works in a fairly similar way to linking tables in relational databases. You just include the document index of one document in another. Embedding is different. When you embed one document in another, you make a copy of the first document inside the second one. This means that the database takes up more space, and it also means that if you update the first document, the second document won't get updates as well. However, it also means that any queries returning the document will be quicker because they only have to find a single document in the store. The actual time it takes to perform the query isn't usually critical on its own, however, it means that you can perform more queries per second on the database server than you otherwise could. If you're serving billions of people who each do hundreds of queries a day, this can make a huge difference.

```
mag = db.test.findOne({issue: "11"});
db.test.save({writer: "Ben", issue: mag, title: "DistroHopper"});
db.test.save({writer: "Ben", issue_id: mag._id, title:
"DistroHopper"});
```

This will create two different documents for DistroHopper, one which is linked with a reference (a manual reference in *MongoDB* speak), and the other is embedded. You can see the differences between them by finding them:

```
> db.test.find();
{ "_id" : ObjectId("545e4058035f3795183110d2"), "issue" : "11",
"mag" : "Linux Voice" }
{ "_id" : ObjectId("545e44b9e4d4d3f214c587f5"), "writer" :
"Ben", "issue_id" : ObjectId("545e4058035f3795183110d2"),
"title" : "DistroHopper" }
{ "_id" : ObjectId("545e44e2e4d4d3f214c587f7"), "writer" :
"Ben", "issue" : { "_id" : ObjectId("545e4058035f379518311
0d2"), "issue" : "11", "mag" : "Linux Voice" }, "title" :
"DistroHopper" }
```



You don't have to install *MongoDB* to try it out. There's a web-based version (with a tutorial) available at http://try.mongodb.org.

Not only can *MongoDB* perform a query faster on a particular piece of hardware, but it can also spread the load across hardware better. Typically, getting better performance out of an SQL database means buying a faster computer to run it on. This is known as scaling up. You can get better performance out of a NoSQL database by running it across more computers (scaling out). This is too complex a topic to get into in just two pages, but briefly, scaling out makes it easier to manage your database as load increases (especially as it scales up to huge transaction volumes). Again, this is something most users never need to worry about because a good server is powerful enough to run a large *MySQL* database.

Any readers well versed in SQL are probably reeling at some aspects of *MongoDB*, like embedding documents. For anyone indoctrinated with the importance of normalising data, this looks like a terrible violation of all things that are important in a database. In many ways, it is. However, in return for violating these important principals, you get speed and scalability. It's not a tradeoff that always makes sense, but there are occasions when this flexibility can be important.

Perhaps the most compelling reason to learn NoSQL though is the job market. Currently **jobs.com** lists NoSQL as the second best trending skill in the jobs market. What's more, if you're still near the start of your career, you won't be competing for jobs with anyone with huge amounts of experience. There are very few people with more than three years NoSQL experience, so it's relatively easy to enter the field.