

Tutorial 14

What is NoSQL?

NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables.

NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph.

They provide flexible schemas and scale easily with large amounts of data and high user loads.

NoSQL database features

Each NoSQL database has its own unique features. At a high level, many NoSQL databases have the following features:

Flexible schemas

Horizontal scaling

Fast queries due to the data model

Ease of use for developers

Why NoSQL?

NoSQL databases are used in nearly every industry

. Use cases range from the highly critical (e.g., storing financial data and healthcare records) to the more fun and frivolous (e.g., storing IoT readings from a smart kitty litter box).

What is a Document Database?

A document database (also known as a document-oriented database or a document store) is a database that stores information in documents.



```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Document databases are considered to be non-relational (or NoSQL) databases. Instead of storing data in fixed rows and columns, document databases use flexible documents.

What are documents?

A document is a record in a document database. A document typically stores information about one object and any of its related metadata.

Documents store data in field-value pairs. The values can be a variety of types and structures, including strings, numbers, dates, arrays, or objects. Documents can be stored in formats like JSON, BSON, and XML.

Below is a JSON document that stores information about a user named Tom.

```
{
  "_id": 1,
  "first_name": "Tom",
  "email": "tom@example.com",
  "cell": "765-555-5555",
  "likes": [
    "fashion",
    "spas",
    "shopping"
  ],
  "businesses": [
    {
      "name": "Entertainment 1000",
      "partner": "Jean",
      "status": "Bankrupt",
      "date_founded": {
        "$date": "2012-05-19T04:00:00Z"
      }
    },
    {
      "name": "Swag for Tweens",
      "date_founded": {
        "$date": "2012-11-01T04:00:00Z"
      }
    }
  ]
}
```

Collections

A collection is a group of documents. Collections typically store documents that have similar contents.

Not all documents in a collection are required to have the same fields, because document databases have a flexible schema

Continuing with the example above, the document with information about Tom could be stored in a collection named users. More documents could be added to the users collection in order to store information about other users. For example, the document below that stores information about Donna could be added to the users collection.

```
{
  "_id": 2,
  "first_name": "Donna",
  "email": "donna@example.com",
  "spouse": "Joe",
  "likes": [
    "spas",
    "shopping",
    "live tweeting"
  ],
  "businesses": [
    {
      "name": "Castle Realty",
      "status": "Thriving",
      "date_founded": {
        "$date": "2013-11-21T04:00:00Z"
      }
    }
  ]
}
```

Note that the document for Donna does not contain the same fields as the document for Tom.

CRUD operations

Document databases typically have an API or query language that allows developers to execute the CRUD (create, read, update, and delete) operations.

Create: Documents can be created in the database. Each document has a unique identifier.

Read: Documents can be read from the database. The API or query language allows developers to query for documents using their unique identifiers or field values. Indexes can be added to the database in order to increase read performance.

Update: Existing documents can be updated — either in whole or in part.

Delete: Documents can be deleted from the database.

MongoDB

MongoDB is a general-purpose document database. Other examples of document databases include CouchDB and Firebase.

In MongoDB, the first field in every document is named `_id`. The `_id` field serves as a unique identifier for the document.

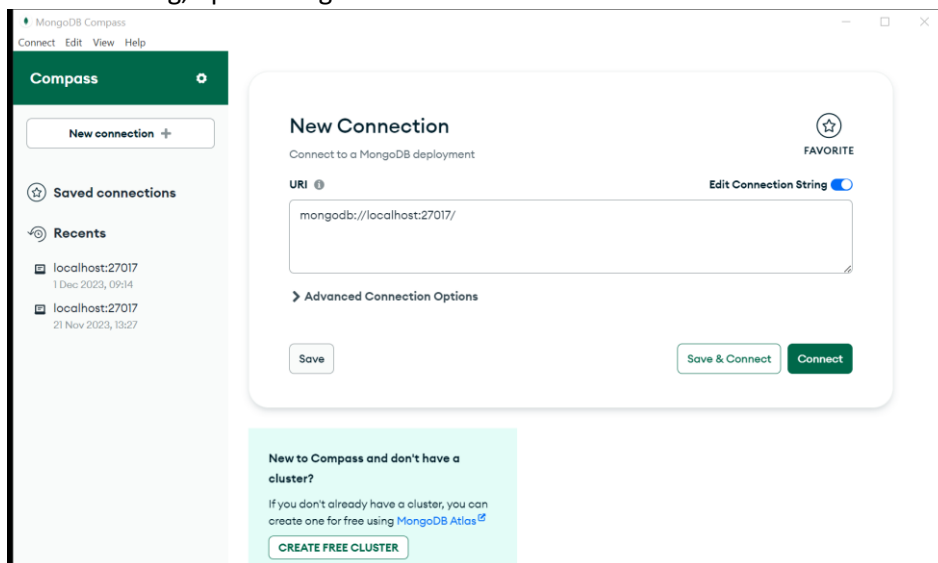
MongoDB stores data in BSON (Binary JSON) documents. The most obvious difference between a document database and a relational database is the way data is modeled. Document databases typically model data using flexible JSON-like documents with field-value pairs. Relational databases typically model data using rigid tables with fixed rows and columns.

MongoDB – Installation and Using MongoDB Compass

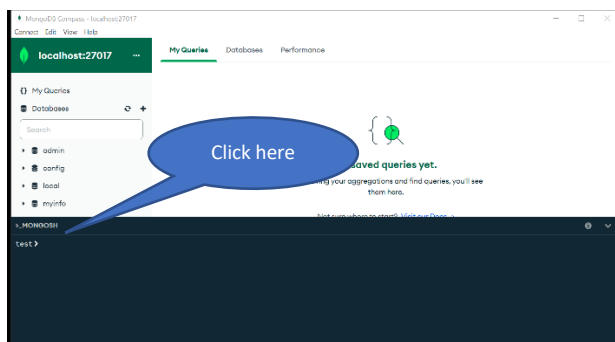
[Download and Install Compass — MongoDB Compass](https://www.mongodb.com/docs/compass/current/install/)
<https://www.mongodb.com/docs/compass/current/install/>

Alternatively you can use online editor : [Create a new MongoDB program - myCompiler - myCompiler](https://www.mycompiler.io/new/mongodb) - <https://www.mycompiler.io/new/mongodb>

After installing, open MongoDB and connect localhost:27017



Click > `_MONGOSH` to view the terminal



In the terminal, type the following commands (also available as text file);

For clearing the terminal window press Ctrl L

```
-- Show Current Databases
show dbs

-- Drop the Unicorns Database
use unicorns
db.dropDatabase

use unicorns

=====
=====
-- CREATE, INSERT & DELETE

db.unicorns.insertOne({name: 'Aurora',
gender: 'f', weight: 450})

db.unicorns.find()

db.unicorns.getIndexes()

-- Insert a totally different document into unicorn
db.unicorns.insert({name: 'Leto',
gender: 'm',
home: 'Arrakeen',
worm: false})

-- Remove what we've put so far in the unicorns collection via:
db.unicorns.remove({}).
db.unicorns.removeMany({})

-- Let's set up some data to play with
db.unicorns.insert({name: 'Horny',
dob: new Date(1992,2,13,7,47),
loves: ['carrot','papaya'],
weight: 600,
gender: 'm',
vampires: 63});
db.unicorns.insert({name: 'Aurora',
dob: new Date(1991, 0, 24, 13, 0),
loves: ['carrot', 'grape'],
weight: 450,
gender: 'f',
vampires: 43});
db.unicorns.insert({name: 'Unicrom',
dob: new Date(1973, 1, 9, 22, 10),
loves: ['energon', 'redbull'],
weight: 984,
gender: 'm',
vampires: 182});
db.unicorns.insert({name: 'Rooooooodles',
dob: new Date(1979, 7, 18, 18, 44),
loves: ['apple'],
weight: 575,
gender: 'm',
vampires: 99});
db.unicorns.insert({name: 'Solnara',
dob: new Date(1985, 6, 4, 2, 1),
```

```

    loves:['apple', 'carrot',
    'chocolate'],
    weight:550,
    gender:'f',
    vampires:80));
db.unicorns.insert({name:'Ayna',
dob: new Date(1998, 2, 7, 8, 30),
loves: ['strawberry', 'lemon'],
weight: 733,
gender: 'f',
vampires: 40});
db.unicorns.insert({name:'Kenny',
dob: new Date(1997, 6, 1, 10, 42),
loves: ['grape', 'lemon'],
weight: 690,
gender: 'm',
vampires: 39});
db.unicorns.insert({name: 'Raleigh',
dob: new Date(2005, 4, 3, 0, 57),
loves: ['apple', 'sugar'],
weight: 421,
gender: 'm',
vampires: 2});
db.unicorns.insert({name: 'Leia',
dob: new Date(2001, 9, 8, 14, 53),
loves: ['apple', 'watermelon'],
weight: 601,
gender: 'f',
vampires: 33});
db.unicorns.insert({name: 'Pilot',
dob: new Date(1997, 2, 1, 5, 3),
loves: ['apple', 'watermelon'],
weight: 650,
gender: 'm',
vampires: 54});
db.unicorns.insert({name: 'Nimue',
dob: new Date(1999, 11, 20, 16, 15),
loves: ['grape', 'carrot'],
weight: 540,
gender: 'f'});
db.unicorns.insert({name: 'Dunx',
dob: new Date(1976, 6, 18, 18, 18),
loves: ['grape', 'watermelon'],
weight: 704,
gender: 'm',
vampires: 165});

```

```

-- Now that we have data, we can master selectors. {field: value} is used
-- to find any documents where field is equal to value.
-- {field1: value1, field2: value2} is how we do an and statement.
-- The special $lt, $lte, $gt, $gte and $ne are used for less than,
-- less than or equal, greater than, greater than or equal and
-- not equal operations.
-- For example, to get all male unicorns
-- that weigh more than 700 pounds, we could do:

```

```

db.unicorns.find({gender: 'm',
weight: {$gt: 700}})
-- OR
db.unicorns.find({gender: {$ne: 'f'},
weight: {$gte: 701}})

```

```

-- The $exists operator is used for matching the presence or absence
-- of a field, for example

db.unicorns.find({
vampires: {$exists: false}})

-- The '$in' operator is used for matching one of several values that
-- we pass as an array, for example:

db.unicorns.find({
loves: {$in:['apple','orange']}})

-- This returns any unicorn who loves 'apple' or 'orange'

-- If we want to OR rather than AND several conditions on different fields,
-- we use the $or operator and assign to it an array of selectors we want
or'd

db.unicorns.find({gender: 'f',
$or: [{loves: 'apple'},
{weight: {$lt: 500}}]})

-- The above will return all female unicorns which either love apples or
-- weigh less than 500 pounds.

-- The ObjectId which MongoDB generated for our _id field can be selected
like so:
db.unicorns.find(
{_id: ObjectId("TheObjectId")})

db.unicorns.find(
{_id: ObjectId("635fbb191a899f4f86a4ca4d")})

-- We've seen how these selectors can be used with the find command.
-- They can also be used with the remove command

=====

-- UPDATING
-- In its simplest form, update takes two parameters: the selector (where)
to use
-- and what updates to apply to fields. If Rooooooodles had gained a bit of
weight,
-- you might expect that we should execute:

db.unicorns.find({name: 'Rooooooodles'})

db.unicorns.updateMany({name: 'Rooooooodles'},
{$set: {weight: 590}})

-- The $inc operator is used to increment a field by a certain
-- positive or negative amount. If Pilot was incorrectly awarded a
-- couple vampire kills, we could correct the mistake by executing:

db.unicorns.update({name: 'Pilot'},
{$inc: {vampires: -2}})

db.unicorns.find({name: 'Pilot'})

```

```

-- If Aurora suddenly developed a sweet tooth, we could add a value to
-- her loves field via the $push operator:

db.unicorns.update({name: 'Aurora'},
{$push: {loves: 'sugar'}})

db.unicorns.find({name: 'Aurora'})

=====
=====

-- UPSERTS
-- An upsert updates the document if found or inserts it if not.

-- With the upsert option omitted (or set to false), executing the
following
-- won't do anything:

db.hits.update({page: 'unicorns'},
{$inc: {hits: 1}});

db.hits.find();

-- However, if we add the upsert option, the results are quite different

db.hits.update({page: 'unicorns'},
{$inc: {hits: 1}}, {upsert:true});

db.hits.find();

-- Since no documents exists with a field page equal to unicorns,
-- a new document is inserted. If we execute it a second
-- time, the existing document is updated and hits is incremented to 2.

db.hits.update({page: 'unicorns'},
{$inc: {hits: 1}}, {upsert:true});
db.hits.find();

-- The final surprise update has to offer is that, by default, it'll update
-- a single document. So far, for the examples we've looked at, this might
-- seem logical. However, if you executed something like:

db.unicorns.update({},
{$set: {vaccinated: true }});
db.unicorns.find({vaccinated: true});

db.unicorns.find({vaccinated: true});

-- You might expect to find all of your precious unicorns to be vaccinated.
-- To get the behavior you desire, the multi option must be set to true:

db.unicorns.update({},
{$set: {vaccinated: true }},
{multi:true});

db.unicorns.find({vaccinated: true});

=====
=====

```

```

-- FIELD SELECTION AND CURSORS

-- For example, we can get all of the unicorns' names without
-- getting back other fields by executing:

db.unicorns.find({}, {name: 1});

-- By default, the _id field is always returned.
-- We can explicitly exclude it by specifying {name:1, _id: 0}.

db.unicorns.find({}, {name:1, _id: 0});

-- ORDERING
-- We specify the fields we want to sort on as a JSON document,
-- using 1 for ascending and -1 for descending. For example:

//heaviest unicorns first
db.unicorns.find().sort({weight: -1})

//by unicorn name then vampire kills:
db.unicorns.find().sort({name: 1,
vampires: -1})

-- PAGING
-- Paging results can be accomplished via the limit and skip cursor
methods.
-- To get the second and third heaviest unicorn, we could do:

db.unicorns.find()
.sort({weight: -1})
.limit(2)
.skip(1)

-- count is actually a cursor method. Drivers which don't provide such a
-- shortcut need to be executed like this:

db.unicorns.find({vampires: {$gt: 50}})
.count()

=====

-- NO JOINS in MongoDB
-- Create an Employee

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d730"),
name: 'Leto'})

-- Now let's add a couple employees and set their manager as Leto

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d731"),
name: 'Duncan',
manager: ObjectId(
"4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d732"),
name: 'Moneo',
manager: ObjectId(
"4d85c7039ab0fd70a117d730")});

```

```

-- Of course, to find all of Leto's employees, one simply executes:

db.employees.find({manager: ObjectId(
"4d85c7039ab0fd70a117d730"}})

-- Arrays and Embedded Documents

-- As a simple example, if an employee could have two
-- managers, we could simply store these in an array:

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d733"),
name: 'Siona',
manager: [ObjectId(
"4d85c7039ab0fd70a117d730"),
ObjectId(
"4d85c7039ab0fd70a117d732")] })

-- Besides arrays, MongoDB also supports embedded documents.
-- Go ahead and try inserting a document with a nested document, such as:

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d734"),
name: 'Ghanima',
family: {mother: 'Chani',
father: 'Paul',
brother: ObjectId(
"4d85c7039ab0fd70a117d730"}}))

-- Embedded documents can be queried using a dot-notation:

db.employees.find({
'family.mother': 'Chani'})

-- Combining the two concepts, we can even embed arrays of documents:

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d735"),
name: 'Chani',
family: [ {relation:'mother',name: 'Chani'},
{relation:'father',name: 'Paul'},
{relation:'brother', name: 'Duncan'}}])

db.employees.find({})

-- Does not work!!!!!!!!!!
db.employees.find({
'family.relation' {'brother.name' :['Duncan']}})

```