Lecture 10

NoSQL and MongoDB

NoSQL

- Not only SQL
- MongoDB
- MongoDB Node.js Driver
- MongoDB Shell (mongosh)

What is NoSQL?

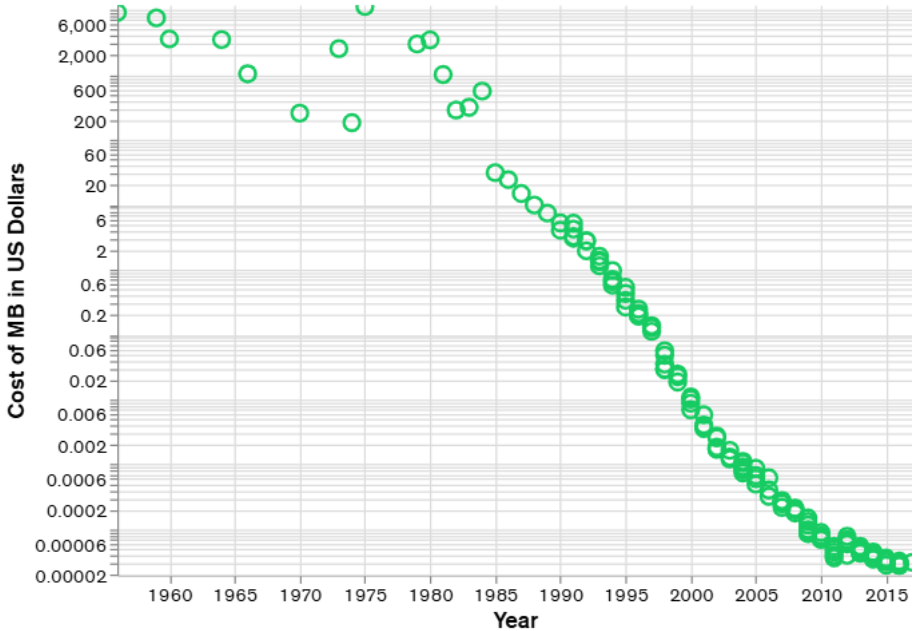
- **NoSQL** databases (AKA "not only SQL") store data differently from relational tables.
- **NoSQL** databases come in a variety of types based on their data model. The main types are document , key-value, wide-column, and graph.
- They provide flexible schemas and scale easily with large amounts of big data and high user loads.

History of NoSQL databases

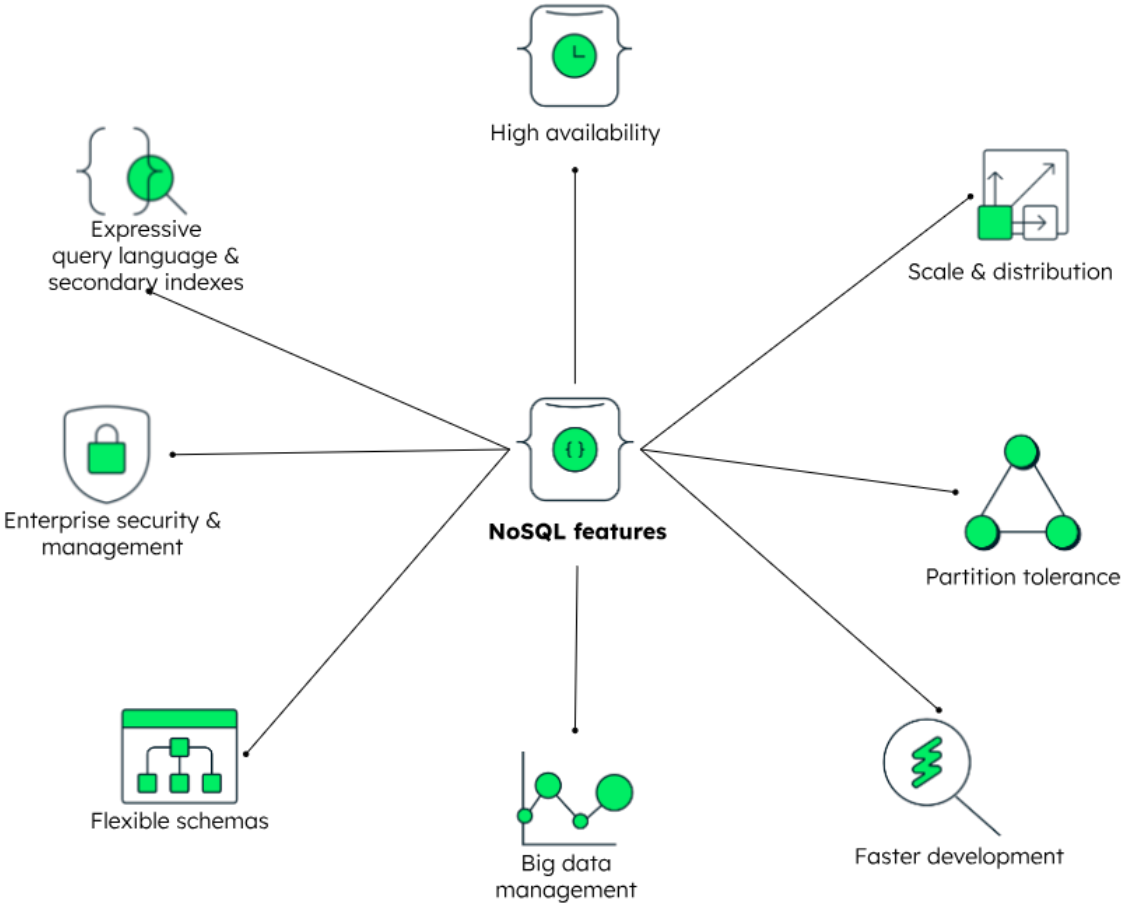
- **NoSQL databases** emerged in the late 2000s.
- The decreased cost of storage removes the need to create a complex, difficult-to-manage data model in order to avoid data duplication.
- NoSQL databases optimized for developer productivity.

History of NoSQL databases

- As storage costs rapidly decreased, the amount of data that applications needed to store and query increased.
- This data came in all shapes and sizes — structured, semi-structured, and unstructured — and defining the schema in advance became nearly impossible.

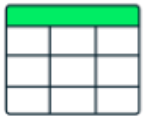


NoSQL database features



Relational database vs NoSQL database

RDBMS vs NoSQL (Document)



Relational Database

MongoDB

User table

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies table

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

```
{
   "_id": 1,
   "first_name": "Leslie",
   "last_name": "Yepp",
   "cell": "8125552344",
   "city": "Pawnee",
   "hobbies": ["scrapbooking", "eating
waffles", "working"]
}
```

- No need for joins
- No need for data normalization

When should NoSQL be used?

When deciding which database to use, decision-makers typically find one or more of the following factors that lead them to select a NoSQL database:

- ✓ Fast-paced Agile development
- ✓ Storage of structured and semi-structured data
- ✓ Huge volumes of data
- ✓ Requirements for scale-out architecture
- ✓ Modern application paradigms like microservices and real-time streaming

Document-oriented databases

- A **document-oriented database** stores data in documents similar to JSON (JavaScript Object Notation) objects.
- The values can typically be a variety of types, including things like strings, numbers, booleans, arrays, or even other objects.
- A document database offers a flexible data model, much suited for semi-structured and typically unstructured data sets.
- They also support nested structures, making it easy to represent complex relationships or hierarchical data.

View of data stored

- Document-oriented databases

```
{
  "_id": "12345",
  "name": "foo bar",
  "email": "foo@bar.com",
  "address": {
    "street": "123 foo street",
    "city": "some city",
    "state": "some state",
    "zip": "123456"
  },
  "hobbies": ["music", "guitar",
"reading"]
}
```

- Key-value databases

Key: user:12345
Value: {"name": "foo bar", "email": "foo@bar.com", "designation": "software developer"}

Wide-column stores

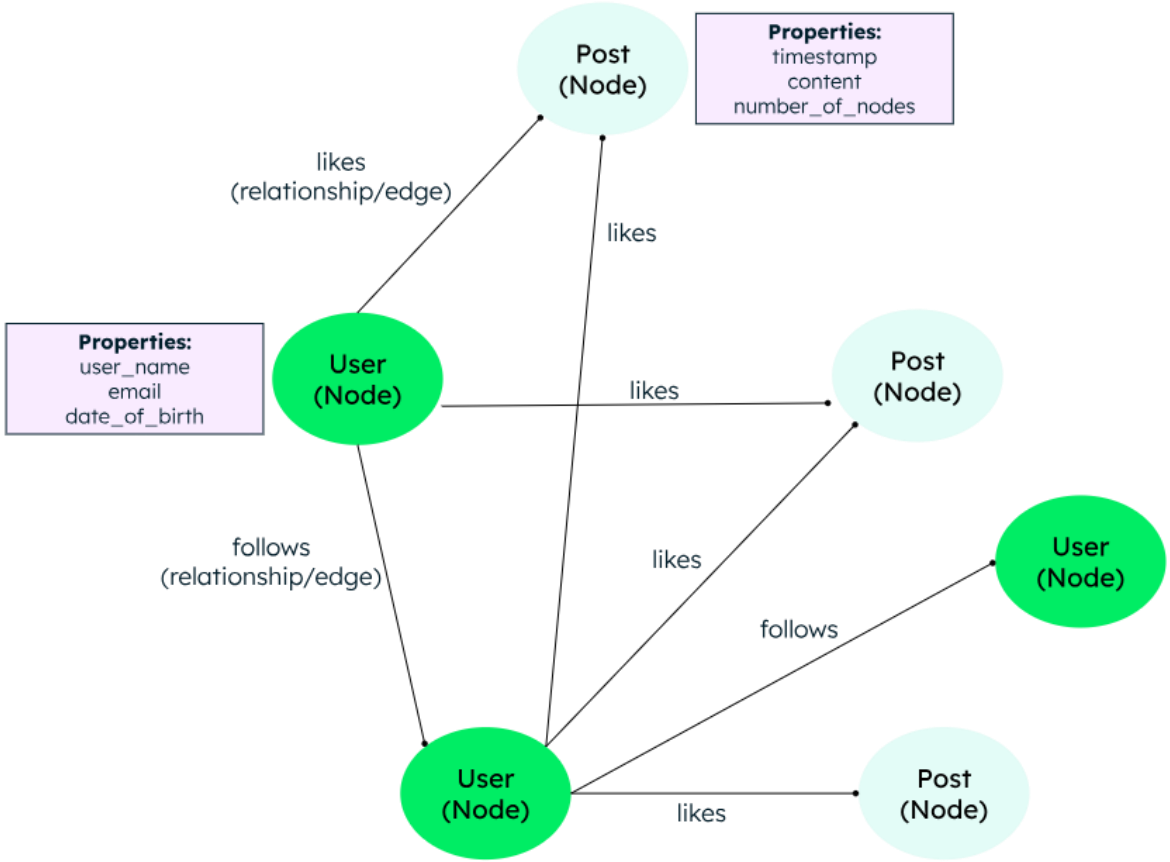
- Wide-column stores store data in tables, rows, and dynamic columns.
- The data is stored in tables. However, unlike traditional SQL databases, wide-column stores are flexible, where different rows can have different sets of columns.

name	id	email	dob	city
Foo bar	12345	foo@bar.com		Some city
Carn Yale	34521	bar@foo.com	12-05-1972	

Graph databases

- A graph database stores data in the form of nodes and edges.
- Nodes typically store information about people, places, and things (like nouns), while edges store information about the relationships between the nodes.

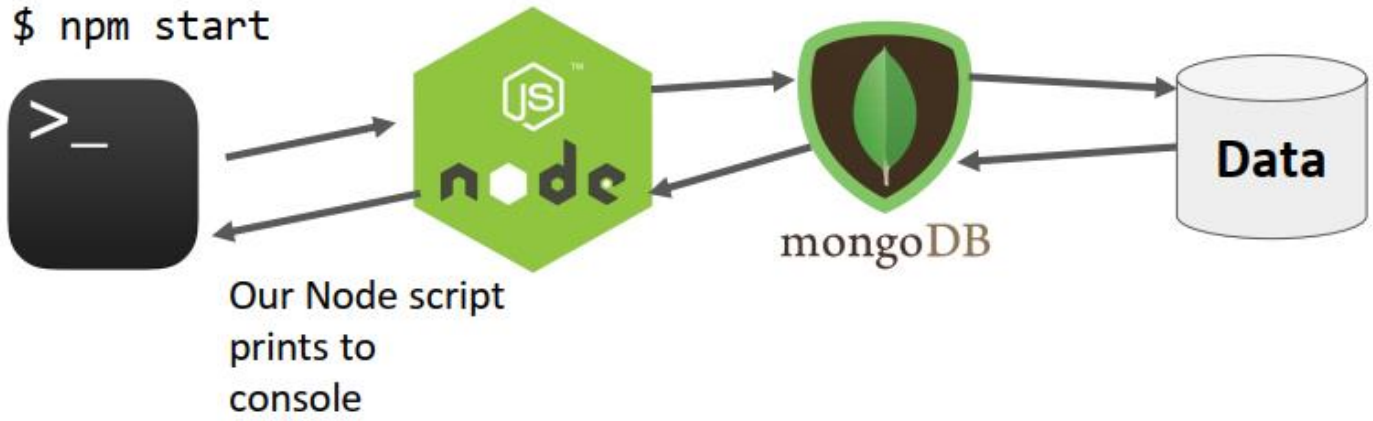
Graph databases



Mongo JS scripts

Mongo JS scripts

Before we start manipulating MongoDB from the server, let's just write some JavaScript files that will query MongoDB.



No web servers are involved yet!

NodeJS Driver

To read and write to the MongoDB database from Node we'll be using the `mongodb` library.

We will install via npm:

```
$ npm install --save mongodb
```

Checking mongoDB node.js driver version:

```
$ npm list mongodb
```

```
lect_demo@ D:\Dạy\HANU\F2024_WPR\Week10\lect_demo
├── mongodb@6.9.0
└─┬ mongoose@8.7.1
  └── mongodb@6.9.0 deduped
```

On the MongoDB website, this library is called the "MongoDB NodeJS Driver"

NodeJS Driver

The official **MongoDB Node.js driver** allows Node.js applications to connect to MongoDB and work with data. The driver features an asynchronous API which allows you to interact with MongoDB using either Promises or traditional callbacks.

Features

- Connection Guide: connect to a MongoDB instance or replica set
- Authentication: configure authentication and log a user in
- CRUD Operations: read and write data to MongoDB
- Promises and Callbacks: access return values using asynchronous Javascript
- Indexes: create and design indexes to make your queries efficient
- Collations: apply language-specific sorting rules to your query results
- Logging: configure the driver to log MongoDB operations
- Monitoring: configure the driver to monitor MongoDB server events

`mongodb` objects

The `mongodb` Node library provides objects to manipulate the database, collections, and documents:

- Db: Database; can get collections using this object

- Collection: Can get/insert/delete documents from this collection via calls like `insertOne, find, etc.`

- Documents are not special classes; they are just JavaScript objects

More Document Database

- A document database (also known as a document-oriented database or a document store) is a database that stores information in documents.
- Document databases offer a variety of advantages, including:
- An intuitive data model that is fast and easy for developers to work with
- A flexible schema that allows for the data model to evolve as application needs change
- The ability to horizontally scale out
- Because of these advantages, document databases are general-purpose databases that can be used in a variety of use cases and industries.

Getting a Db object

(Old Way)

You can get a reference to the database object by using the `MongoClient.connect(url, callback)` function:

- `url` is the connection string for the MongoDB server
- `callback` is the function invoked when connected
- `database` parameter: the Db object

```
const { MongoClient } = require('mongodb');
const DATABASE_NAME = 'eng-dict';
const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;
let db = null;
MongoClient.connect(MONGO_URL, function(err, client) {
    db = client.db();
});
```

Connection string

```
const DATABASE_NAME = 'eng-dict';

const MONGO_URL = `mongodb://localhost:27017/${DATABASE_NAME}`;
```

- The URL is to a MongoDB server, not a web URL, which is why it begins with `mongodb://` and not `http://`
- The MongoDB server is running on our local machine, which is why we use `localhost`
- The end of the connection string specifies the database name we want to use.
    - ✓ If a database of that name doesn't already exist, it will be created the first time we write to it.

MongoDB Connection string format

Getting a Db object

(New Way)

MongoDB Driver version 4.10+ no longer supports **error-first callbacks**. Instead, use Promises for handling asynchronous operations.

- `url`: Connection string for MongoDB server.
- `client`: A MongoClient instance returned by **`await`**.
- `db`: Use **`client.db(databaseName)`** to get the **`Db`** object.

```
const client = await MongoClient.connect(url);
const db = client.db(databaseName);
```

Getting a Db object

- The `MongoClient.connect` function returns a Promise so it's also possible to use this function with the `.then/.catch` style:

```
let db = null;
function onConnected(err, client){
    db = client.db()
}
MongoClient.connect(MONGO_URL).then(onConnected)
```

Using a collection

What is a Collection?

- ✓ A collection in MongoDB is similar to a table in relational databases.
- ✓ It stores documents (similar to rows) in a NoSQL format, typically as JSON objects

Accessing a Collection

- ✓ To interact with a collection, you first need to retrieve it from the database using the `db.collection()` method.

Using a collection

```
async function useCollection() {
    const client = await MongoClient.connect(MONGO_URL);
    const db = client.db(DATABASE_NAME);
    const coll = db.collection('users');

}
useCollection();
```

`const coll = db.collection(`***`collectionName`***`);`

- Obtains the collection object named ***`collectionName`*** and stores it in `coll`
- You do not have to create the collection before using it
  (It will be created the first time we write to it)
- This function is **synchronous**

collection.insertOne

```
const result = await collection.insertOne(doc);
```

- Adds one item to the collection
- `doc` is a JavaScript object representing the key-value pairs to add to the collection
- Returns a `Promise` that resolves to a result object when the insertion has completed
    - `result.insertedId` will contain the id of the object that was created

collection.insertOne example

```
async function insertUserAsync(name,age){
    const newUser = { name: name, age: age };
    const result =  await collection.insertOne(newUser);
    console.log(`Document id: ${result.insertedId}`);

}
```

We will be using the Promise + `async/await` versions of all the MongoDB asynchronous functions, as it will help us avoid callback hell.

collection.findOne

```
const doc = await collection.findOne(query);
```

- Finds the first item in the collection that matches the query
- `query` is a JS object representing which fields to match on
- Returns a Promise that resolves to a document object when `findOne` has completed
  - `doc` will be the JS object, so you can access a field via `doc.fieldName`, e.g. `doc._id`
  - If nothing is found, `doc` will be `null`

collection.findOne

```
async function findUser() {
        const query = { name: 'John Doe' };
        const user = await collection.findOne(query);

        if (user) {
            console.log('User found:', user);
        } else {
            console.log('No user found with the given criteria');
        }
}
```

collection.find

```
const cursor = await collection.find(query);
```

- Returns a Cursor to pointing to the first entry of a set of documents matching the query

- You can use `hasNext` and `next` to iterate through the list:

```
async function findAllUsers() {
    const query = {};
    const cursor = await collection.find(query);

    while (await cursor.hasNext()) {
      const user = await cursor.next();
      console.log("User:", user);
    }
}
```

(This is an example of something that is **a lot** easier to do with `async/await`)

collection.find().toArray()

```
const cursor = await collection.find(query);
const list = await cursor.toArray();
```

- `Cursor` also has a `toArray()` function that converts the results to an array.

```
async function findAllUsers() {
        const query = {};
        const users = await collection.find(query).toArray();
        console.log('Users:', users);
}
```

collection.updateOne

```
await collection.updateOne(query, newEntry);
```

- Replaces **the first item** matching `query` with `newEntry`

(**Note:** This is the simplest version of update. There are more complex versions of update that we will address later.)

```
async function updateUser(userId, newData) {
    const query = { _id: userId };
    const result = await collection.updateOne(
        query, { $set: newData }
    );
    console.log(`Matched ${result.matchedCount} document(s)
        and modified ${result.modifiedCount} document(s)`);
}
```

collection.updateMany

```
await collection.updateMany(query, newEntry);
```

- Replaces **all items** matching `query` with `newEntry`.
- To update multiple documents, you can use the `updateMany()` method.

```
async function updateUser(query, newData) {
    const result = await collection.updateMany(
        query, { $set: newData }
    );
    console.log(`Matched ${result.matchedCount} document(s)
        and modified ${result.modifiedCount} document(s)`);
}
```

"Upsert" with `collection.updateOne/Many`

MongoDB also supports **upsert**, which is:

- Update the entry if it already exists
- Insert the entry if it doesn't already exist

```
// Define upsert option
const params = { upsert: true };
```

The `upsert` option can be used with `updateOne()` or `updateMany()` to achieve this functionality.

"Upsert" with collection.updateOne/Many

```
async function upsertEntry(query, newEntry) {
    // Define upsert option
    const params = { upsert: true };
    // Perform upsert operation
    const result = await collection.updateOne(
        query, { $set: newEntry }, params
    );
    if (result.upsertedCount > 0) {
        console.log(`Inserted a new document with
            id ${result.upsertedId._id}`);

    } else {
        console.log(`Updated ${result.matchedCount} document(s)
            and modified ${result.modifiedCount} document(s)`);
    }
}
```

collection.deleteOne

```
const result = await collection.deleteOne(query);
```

- Deletes the first the item matching `query`.
- `result.deletedCount` gives the number of docs deleted (will be 1 if successful, or 0 if no match).

```
const result = await collection.deleteOne(query);
console.log(`${result.deletedCount} document(s) deleted.`);
```

collection.deleteOne/Many

```
const result = await collection.deleteOne(query);
```

- Deletes the first the item matching `query`.
- `result.deletedCount` gives the number of docs deleted.

```
const result = await collection.deleteMany(query);
console.log(`${result.deletedCount} document(s) deleted.`);
```

Advanced queries

MongoDB has a more powerful querying syntax that was not covered in this lecture.

For more complex queries, check out:

- Querying
  - Query selectors and projection operators

    Example: `db.collection('inventory').find({ qty: { $lt: 30 } });`
- Updating
  - Update operators
  - For example, the `$set` operator:

```
db.collection('words').updateOne(
    { word: searchWord },
    { $set: { definition: newDefinition }}
);
```

MongoDB Atlas

- **MongoDB Atlas** is a cloud-based database service created by the MongoDB team.

- It simplifies database deployment and management, offering flexibility for building robust and high-performance global applications.

- Atlas allows you to choose from multiple cloud providers, making it easier to manage databases across different cloud environments.

MongoDB Shell (mongosh)

- The MongoDB Shell (mongosh) is a JavaScript and Node.js REPL (Read Eval Print Loop) environment designed for interacting with MongoDB deployments, whether hosted on MongoDB Atlas, locally, or on a remote server.

- It allows you to test queries and interact directly with the data in your MongoDB database, providing a useful tool for exploring and managing your database.

Access MongoDB From Your Shell

1

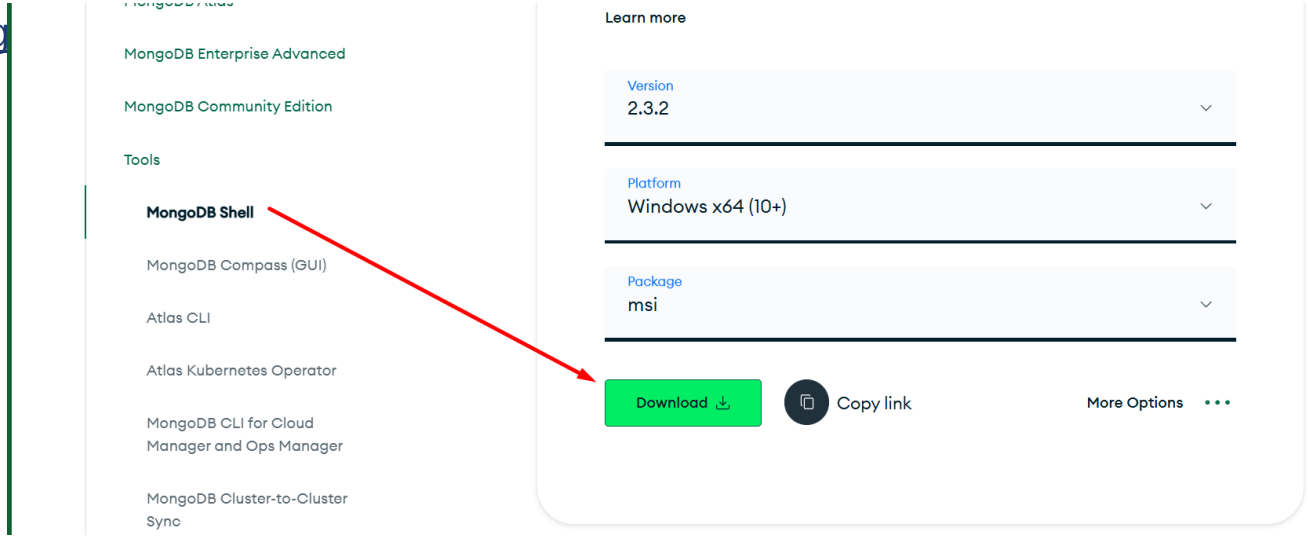
Find Your Connection String

The connection string varies depending on the type of deployment you're connecting to.

Learn how to find your connection string for Atlas.

Or connect to a self-hosted deployment.

Download mong

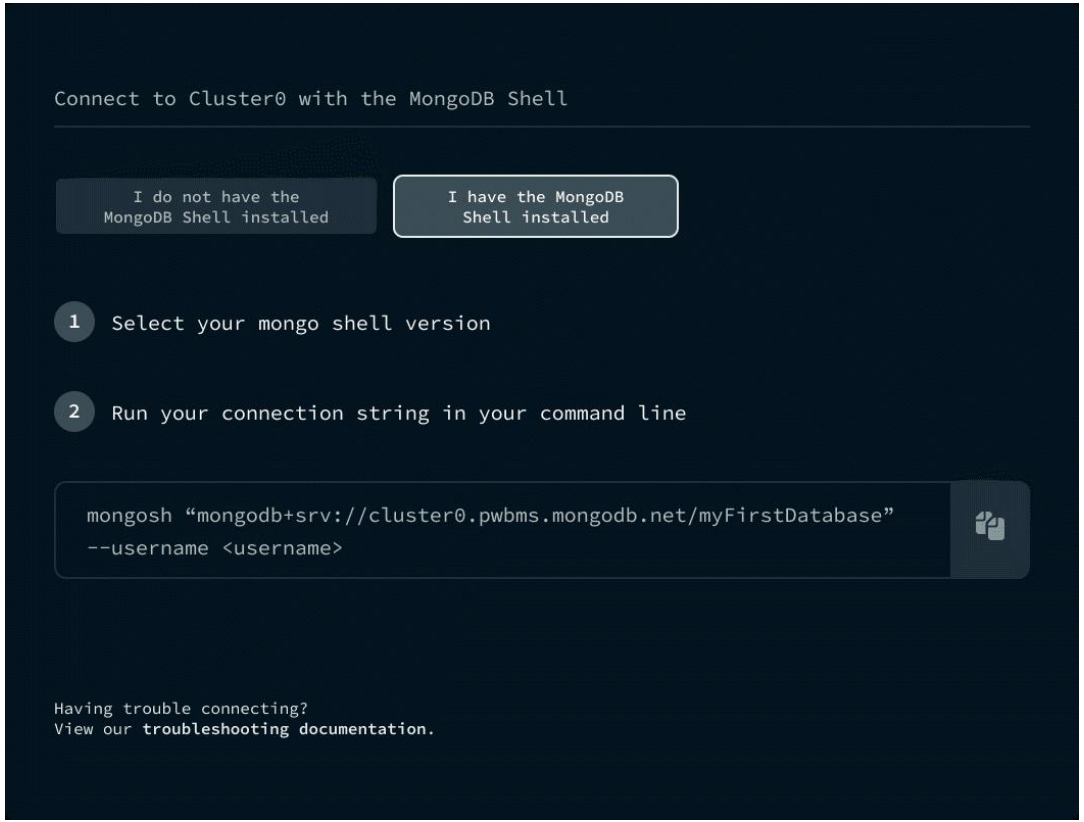


Access MongoDB From Your Shell

1

Find Your Connection String

Connection string for Atlas.

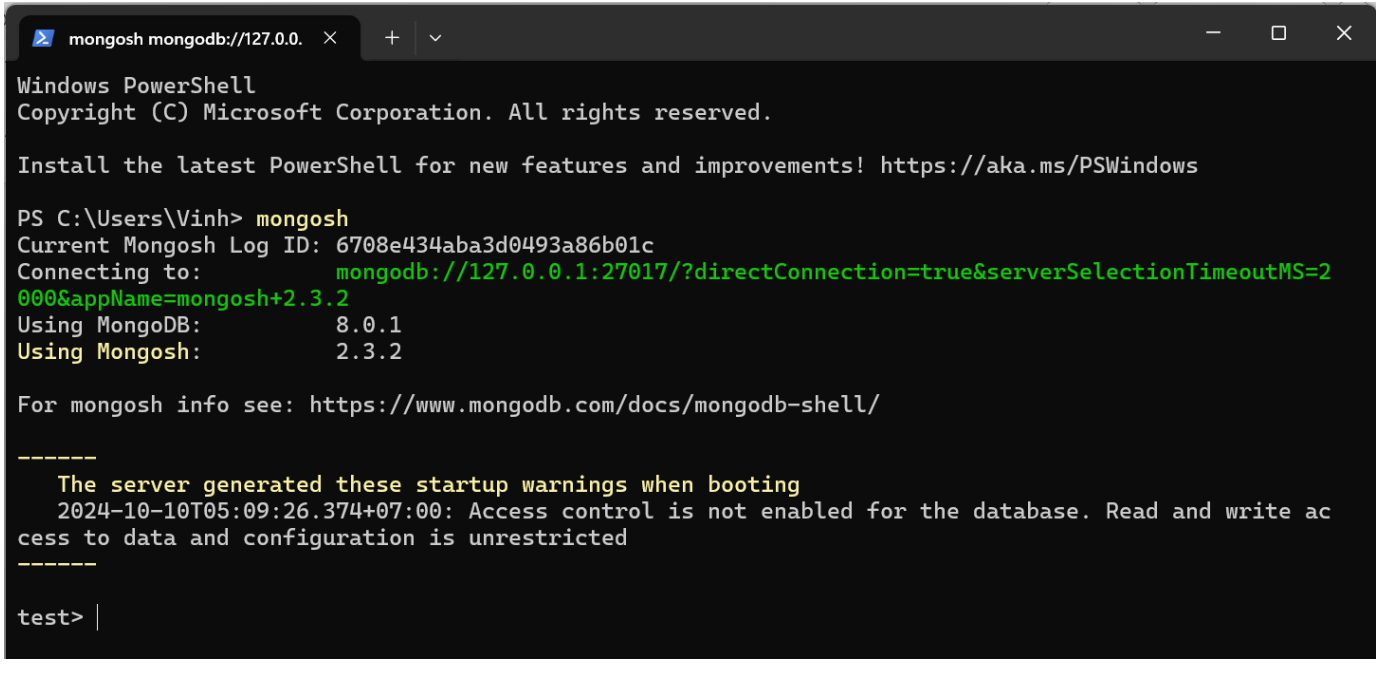


Access MongoDB From Your Shell

2

Connect to MongoDB

For self-hosted MongoDB (local or remote deployments): The connection string often looks like **mongodb://localhost:27017** for local databases or may include the IP address and port for a remote host.



Access MongoDB From Your Shell

3

Interact with Your Data

- Use your chosen connection type to view your data, import documents, and run queries.
- **`show dbs:`** Lists all databases.

```
test> show dbs
admin      40.00 KiB
config     48.00 KiB
eng-dict   72.00 KiB
local      40.00 KiB
```

Basic MongoDB Commands Overview

- **`show dbs:`** Lists all databases.
- **`use <db_name>:`** Switch to (or create) a database. *(It will be created the first time we write to it)*
- **`show collections:`** Lists all collections in the current database.
- **`db.<collection_name>.find():`** Query all documents in a collection.
- **`db.dropDatabase():`** Delete database

```
test> show dbs
admin     40.00 KiB
config    72.00 KiB
eng-dict  80.00 KiB
local     40.00 KiB
users      8.00 KiB
test> use eng-dict
switched to db eng-dict
eng-dict> show collections
users
words
eng-dict> db.words.find()
[
  {
    _id: ObjectId('6708f32ab30a22e2c386b01d'),
    word: 'dog',
    definition: 'friend'
  },
  {
    _id: ObjectId('6708f334b30a22e2c386b01e'),
    word: 'cat',
    definition: 'boss'
  },
  {
    _id: ObjectId('6708f334b30a22e2c386b01f'),
    word: 'bird',
    definition: 'flyer'
  }
]
eng-dict>
```

MongoSH: Working with Collections

- Create a collection: Collections are created automatically when you insert data.

```
db.createCollection('words')
```

- Inserting data into a collection -- Insert One Document:

```
db.words.insertOne({ word: 'dog', definition: 'friend' })
```

- Insert Multiple Documents:

```
db.words.insertMany([
    { word: 'cat', definition: 'boss' },
    { word: 'bird', definition: 'flyer' }
])
```

MongoSH: Working with Collections

- Query All Documents:

```
db.words.find().pretty()
```

- `pretty()` formats the result for better readability.

- Query Specific Document:

```
db.words.findOne({ word: 'dog' })
```

MongoSH: Working with Collections

- Update One Document:

```
db.words.updateOne(
    { word: 'dog' },
    { $set: { definition: 'woof woof' } }
)
```

- Update Multiple Documents:

```
db.words.updateMany(
    {},
    { $set: { definition: 'empty: to-update' } }
)
```

MongoSH: Working with Collections

- Delete One Document:

```
db.words.deleteOne({ word: 'dog' })
```

- Delete Multiple Documents:

```
db.words.deleteMany({ word: { $exists: true } })
```

- Delete a Collection:

```
db.words.drop()
```

MongoDB Shell Summary

- MongoDB's CLI provides a simple way to interact with databases.
- Key operations include:
    - ✓ creating databases
    - ✓ managing collections
    - ✓ inserting
    - ✓ querying
    - ✓ updating
    - ✓ deleting