

Document Databases

Flexible, scalable, schema-less databases for storing and analyzing documents.



Outlines

- Introduction.
- *Types*
- Cloud Document Databases
- MongoDB Use Cases

Outlines

- Introduction.
- *Types*
- Cloud Document Databases
- MongoDB Use Cases

What is Document Database?

- Document databases are non-relational (or NoSQL) databases. Instead of storing data in fixed rows and columns, document databases use flexible documents.
- Document databases are the most popular alternative to tabular, relational databases.

What are 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.

Let's discover it.



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



```
{
  "_id": "5cf0029cafff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Key characteristics



- Document databases are general-purpose databases that serve a variety of use cases for both transactional and analytical applications:
 1. Single view or data hub
 2. Real-time analytics
 3. Customer data management and personalization
- 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.

Key Features



1. **Document model:** Data is stored in documents (unlike other databases that store data in structures like tables or graphs). Documents map to objects in the most popular programming languages, which allows developers to rapidly develop their applications.
2. **Flexible schema:** Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields. Note that some document databases support schema validation, so the schema can be optionally locked down.

Key Features

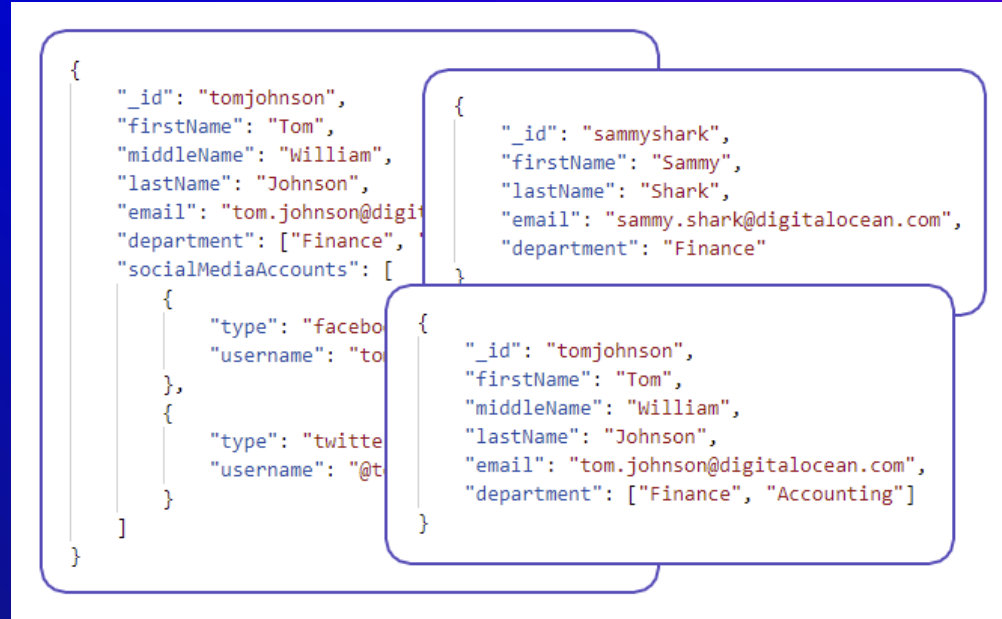


- 3. *Distributed and resilient:*** Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) and data distribution. Document databases provide resiliency through replication.
- 4. *Querying through an API or query language:*** Document databases have an API or query language that allows developers to execute the CRUD operations on the database. Developers have the ability to query for documents based on unique identifiers or field values.

What makes document databases different from relational databases?

1. *The intuitiveness of the data model:*

Documents map to the objects in code, so they are much more natural to work with. There is no need to decompose data across tables, run expensive joins, or integrate a separate Object Relational Mapping (ORM) layer. Data that are accessed together are stored together, so developers have less code to write, and end users get higher performance.



What makes document databases different from relational databases?

- 2. *The ubiquity of JSON documents:*** JSON has become an established standard for data interchange and storage. JSON documents are lightweight, language-independent, and human-readable. Documents are a superset of all other data models so developers can structure data in the way their applications need — rich objects, key-value pairs, tables, geospatial and time-series data, or the nodes and edges of a graph.
- 3. *The flexibility of the schema:*** A document's schema is dynamic and self-describing, so developers don't need to first pre-define it in the database.

Outlines

- Introduction.
- Types
- Cloud Document Databases
- MongoDB Use Cases

What is JSON?

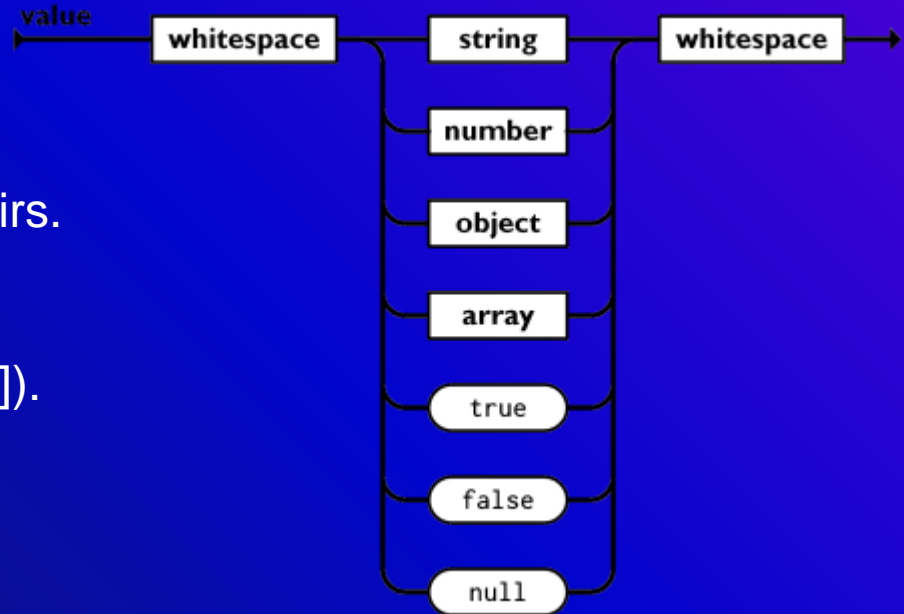


JSON (JavaScript Object Notation) is a lightweight and open standard file format that uses human-readable text for storing and transporting data. It's a data-interchange format that transmits data objects that contains key-value pairs and arrays to store data. It's easy for machines to parse and generate. JSON is based on a subset of the JavaScript programming language and was originally designed as an alternative to XML. Also, it's a language-independent format which means it supports every kind of language, framework, and library.

Basic JSON Syntax

Rules:

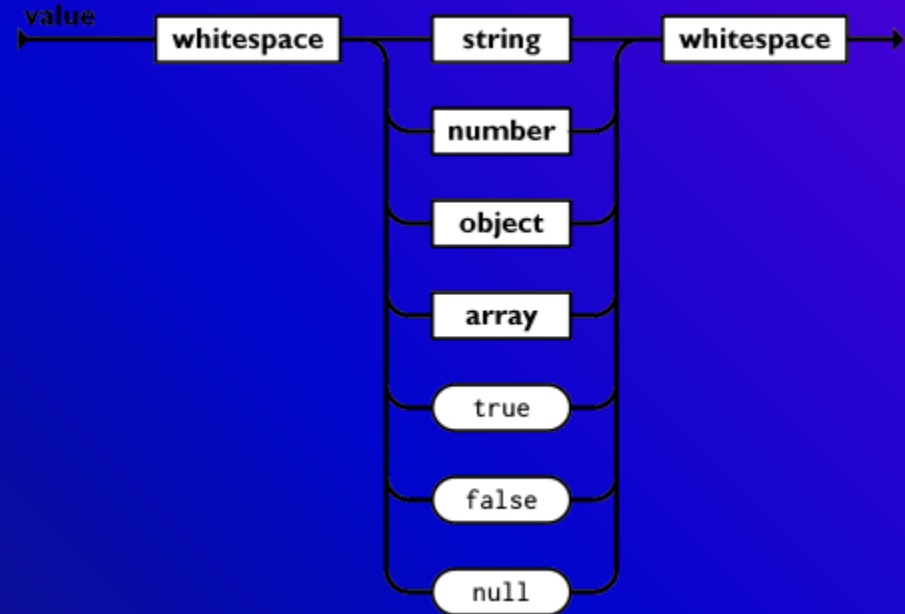
- <KEY>:<VALUE> Data is in key/value pairs.
- Data is separated by commas.
- Objects are enclosed in curly braces ({}).
- Arrays are enclosed in square brackets ([]).



Basic JSON Syntax

In JSON, values must be one of the following data types:

- ✓ a string
- ✓ a number
- ✓ an object
- ✓ an array
- ✓ a boolean
- ✓ null



Basic JSON Syntax

Rules:

- <KEY>:<VALUE> Data is in key/value pairs.
- Data is separated by commas.
- Objects are enclosed in curly braces ({}).
- Arrays are enclosed in square brackets ([]).



```
{ "employees": [  
  { "firstName": "John", "lastName": "Doe" },  
  { "firstName": "Anna", "lastName": "Smith" },  
  { "firstName": "Peter", "lastName": "Jones" }  
]}
```


What is Data Modeling?

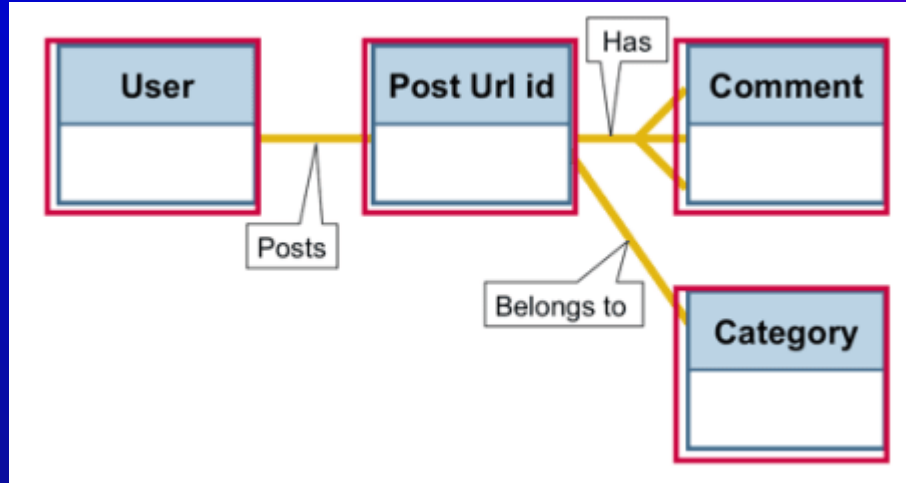
- ❑ Data Modeling is a process of generating a visual representation of an entire information system to express relations between data and structures. It revolves around analyzing and defining all different data points business collects and generates and also managing relationships between data points. The process itself is an exercise to get a better understanding and clarification of the data requirements as per business needs.
- ❑ Data Modeling enforces business rules, regulatory compliances, and government policies on the data. It defines how data will flow into and out of the Database.

JSON Modeling for Document Databases

- ❑ JSON Modeling bind controls to JavaScript object data which is generally specified in JSON format. It's a client-side model and does not support mechanisms that involve server-side paging or loading. JSON Modeling doesn't support any for sending data back to the server.
- ❑ JSON Modeling has a data model for document Databases consisting of a structured hierarchy. A document in the JSON data model may contain embedded documents as data. In JSON Modeling each document with data represent the attributes of an entity. Documents are organized into collections that are equivalent to tables in Relational Databases.

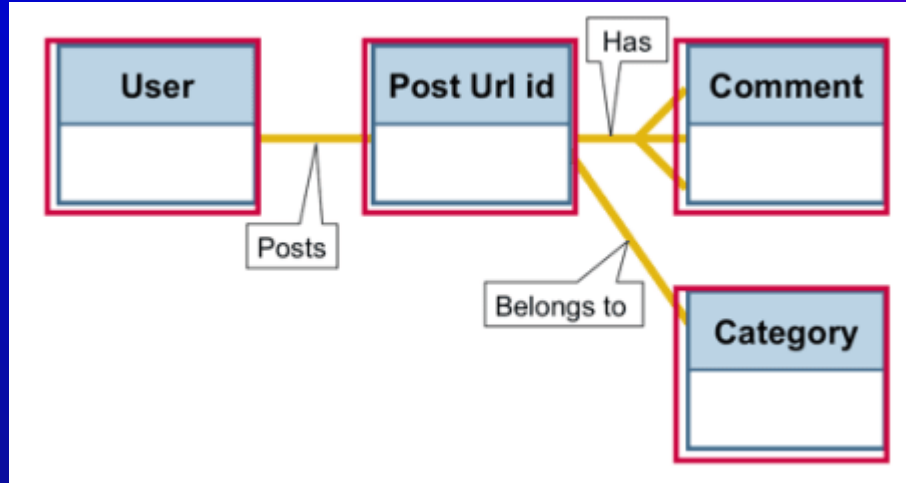
JSON Modeling Process

- ❖ In this section, you will learn how to perform JSON Modeling in NoSQL Databases using an example. It's a good practice to start with Entity-Relationship modeling to define the entities, relationships, and attributes for the application.



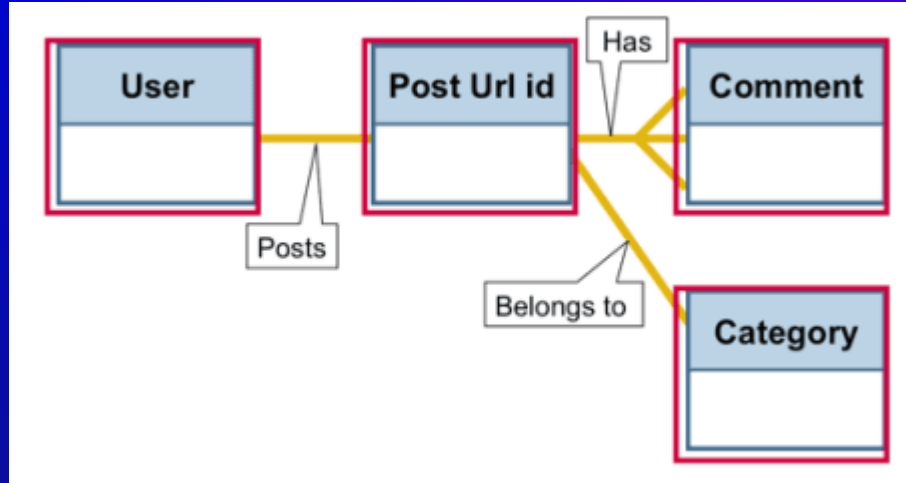
JSON Modeling Process

- ❖ Entities are the main objects, Attributes are properties of the objects and Relationships are connections between Entities. This can be one-to-one, one-to-many, many-to-many, etc. Let's take the example of social media applications. The E-R diagram for social media applications is shown.



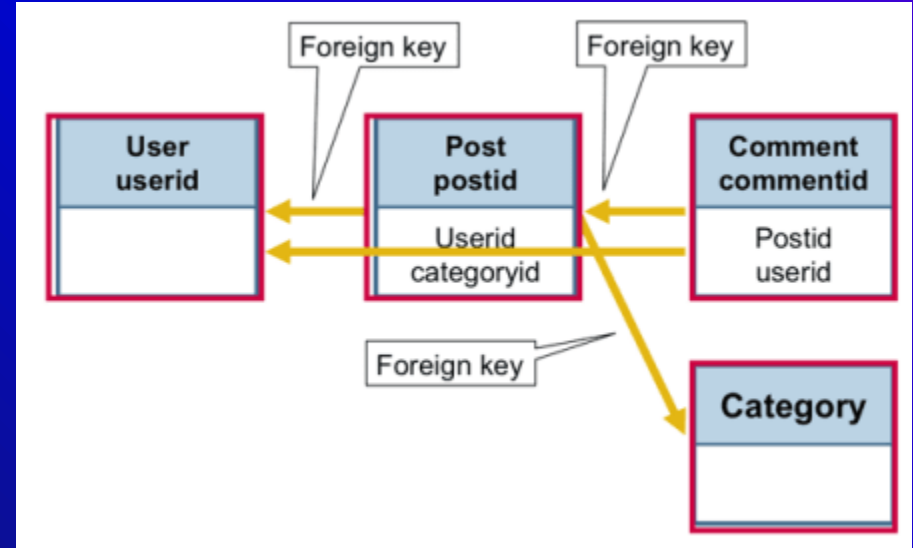
JSON Modeling Process

- ❖ In this diagram, the Entities are User, Post, and Comment. Relationships are user makes a post, a post has comments, and a post belongs to a category.



JSON Modeling Process

- ❖ Here, users are stored in the user table. The posted URL is stored in the Post table with a foreign key to the user that posted it, and a foreign key to the category for the post. Comments about a post are stored in the comments table with a foreign key to the post and a foreign key to the user that commented.

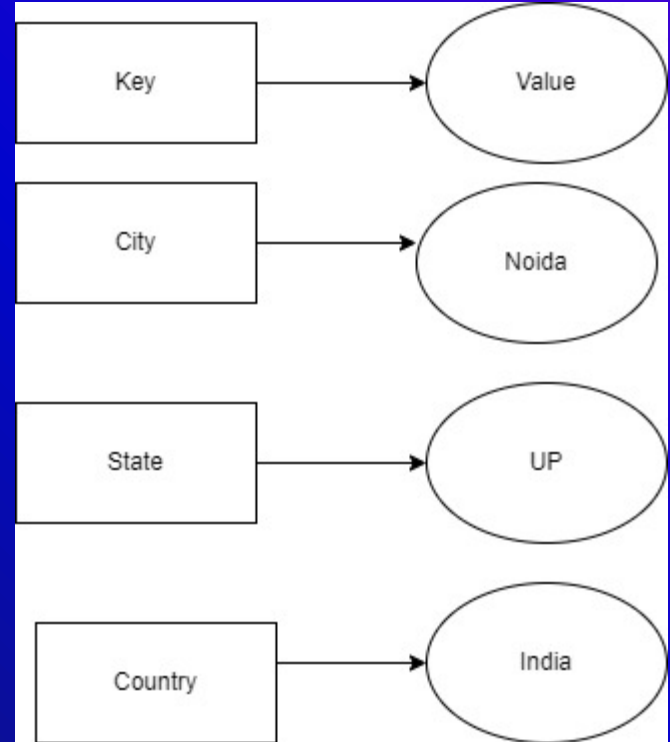


Rules for JSON Modeling

- ✓ If the Relationship is **one-to-one** or **one-to-many** then store related data as nested objects.
- ✓ If the Relationship is **many-to-one** or **many-to-many** then store related data as separate documents.
- ✓ If data reads are mostly parent fields then store children as a separate document.
- ✓ If data reads are mostly **parent + child** reads then store children as nested objects.
- ✓ If data writes are mostly either **parent or child** then store children as separate documents.
- ✓ If data writes are mostly either **parent and child** then store children as nested objects.

Key-Value Data document Model

- ❖ A key-value data model or database is also referred to as a key-value store. It is a non-relational type of database. In this, an associative array is used as a basic database in which an individual key is linked with just one value in a collection. For the values, keys are special identifiers. Any kind of entity can be valued. The collection of key-value pairs stored on separate records is called key-value databases and they do not have an already defined structure.



How do key-value databases work?

- ❖ Several easy strings or even a complicated entity are referred to as a value that is associated with a key by a key-value database, which is utilized to monitor the entity. Like in many programming paradigms, a key-value database resembles a map object or array, or dictionary, however, which is put away in a tenacious manner and controlled by a DBMS.
- ❖ An efficient and compact structure of the index is used by the key-value store to have the option to rapidly and dependably find value using its key. For example, Redis is a key-value store used to track lists, maps, heaps, and primitive types (which are simple data structures) in a constant database. Redis can uncover a very basic point of interaction to query and manipulate value types, just by supporting a predetermined number of value types, and when arranged, is prepared to do high throughput.

When to use a key-value database:

- ❖ Here are a few situations in which you can use a key-value database:-
 - ✓ User session attributes in an online app like finance or gaming, which is referred to as real-time random data access.
 - ✓ Caching mechanism for repeatedly accessing data or key-based design.
 - ✓ The application is developed on queries that are based on keys.

Advantages vs Disadvantages

❖ Advantages:

- ✓ It is very easy to use. Due to the simplicity of the database, data can accept any kind, or even different kinds when required.
- ✓ Its response time is fast due to its simplicity, given that the remaining environment near it is very much constructed and improved.
- ✓ Key-value store databases are scalable vertically as well as horizontally.
- ✓ Built-in redundancy makes this database more reliable.

❖ Disadvantages:

- As querying language is not present in key-value databases, transportation of queries from one database to a different database cannot be done.
- The key-value store database is not refined. You cannot query the database without a key.

Wide-column Data document Model

- ❖ A wide-column database is a NoSQL database that organizes data storage into flexible columns that can be spread across multiple servers or database nodes, using multi-dimensional mapping to reference data by column, row, and timestamp.

Row A	Column 1	Column 2	Column 3
	Value	Value	Value
Row B	Column 1	Column 2	Column 3
	Value	Value	Value

Wide-column Data document Model

- ❖ A wide-column database is a type of NoSQL database in which the names and format of the columns can vary across rows, even within the same table. Wide-column databases are also known as column family databases. Because data is stored in columns, queries for a particular value in a column are very fast, as the entire column can be loaded and searched quickly. Related columns can be modeled as part of the same column family.

Row A	Column 1	Column 2	Column 3
	Value	Value	Value
Row B	Column 1	Column 2	Column 3
	Value	Value	Value

How Does a Wide-column Store Database Differ from a Relational Database?

- ❖ A relational database management system (RDBMS) stores data in a table with rows that all span a number of columns. If one row needs an additional column, that column must be added to the entire table, with null or default values provided for all the other rows. If you need to query that RDBMS table for a value that isn't indexed, the table scan to locate those values will be very slow.
- ❖ Wide-column NoSQL databases still have the concept of rows, but reading or writing a row of data consists of reading or writing the individual columns. A column is only written if there's a data element for it. Each data element can be referenced by the row key, but querying for a value is optimized like querying an index in a RDBMS, rather than a slow table scan.

What's the Difference Between Wide-Column and Key-Value Store NoSQL databases?

- ❖ NoSQL databases do not store their data in related tables, but NoSQL data stores include four major implementations: Key-Value databases, Wide-column databases, Document databases, and Graph databases. We'll just look at the first two, because they are similar. Key-Value databases are the simplest model and can be thought of as a configuration file or a two-column table of keys with an associated value. Wide-column databases expand that key-value store concept across multiple columns, but only the columns that are needed for that record.

What are Wide-column Database Use Cases?

- ❖ Wide-column databases are ideal for use cases that require a large dataset that can be distributed across multiple database nodes, especially when the columns are not always the same for every row.
- Log data
- IoT (Internet of Things) sensor data
- Time-series data, such as temperature monitoring or financial trading data
- Attribute-based data, such as user preferences or equipment features
- Real-time analytics

What are Wide-column Database Examples?

- ❖ Some common wide-column store database examples include Apache Cassandra, ScyllaDB, Apache HBase, Google BigTable, and Microsoft Azure Cosmos DB. When it comes to a wide-column database, Cassandra is often mentioned first because of its pioneering work. But ScyllaDB is Cassandra rewritten in the C++ programming language, making it faster and more reliable. ScyllaDB has continued to evolve as a notable wide-column database Cassandra.



Break
“Go out and see the world”



Outlines

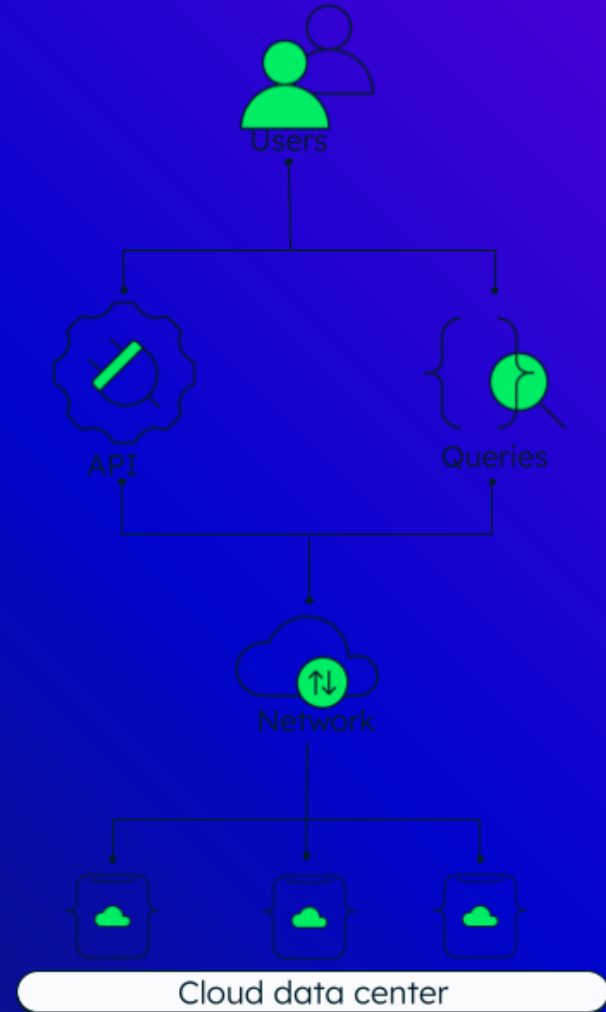
- Introduction.
- *Types*
- Cloud Document Databases
- MongoDB Use Cases

What is a Cloud Database?

- ❖ Cloud databases are just like traditional ones but require no set-up and infrastructure maintenance. Cloud databases run on a cloud computing environment.
- ❖ A cloud database is a database that is deployed in a cloud environment as opposed to an on-premise environment. The database itself can be offered as a SaaS (Software-as-a-Service) application or simply be hosted in a cloud-based virtual machine. Applications can then access all the data stored in a cloud database over a network from any device.

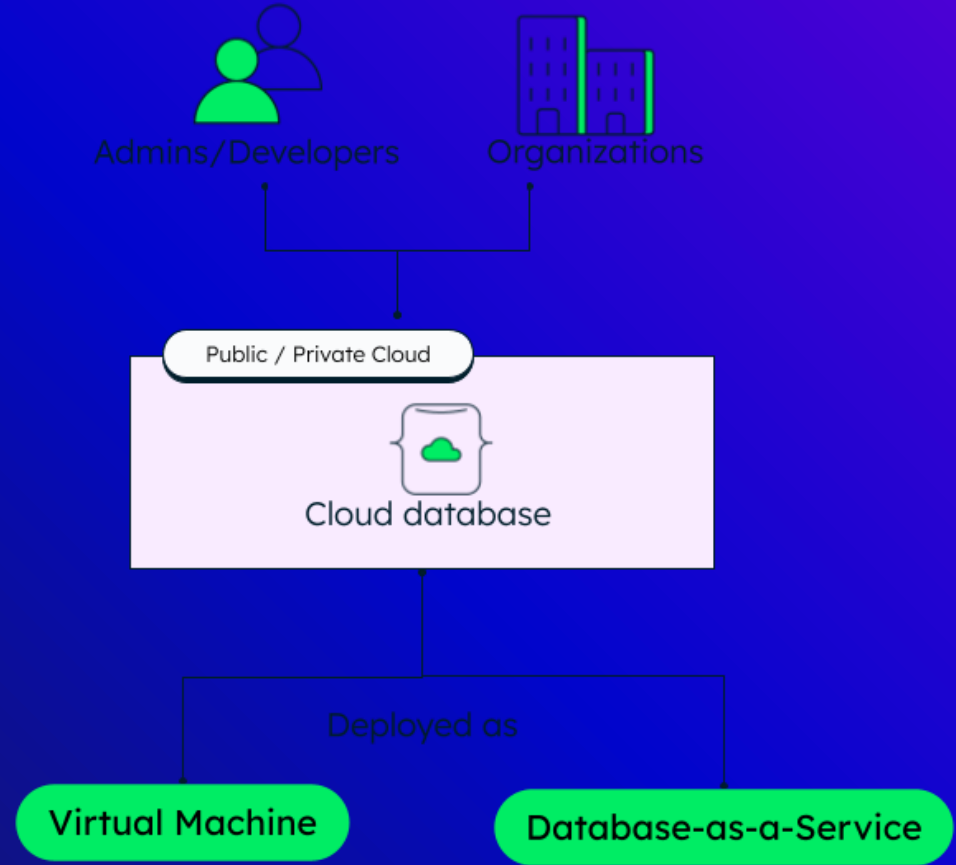
What is a Cloud Database?

- ❖ With a cloud database, there is no need for dedicated hardware to host a database. Rather than the organization itself installing, configuring, and maintaining a database instance or instances, the cloud provider can provision, manage, and scale the underlying database cluster.



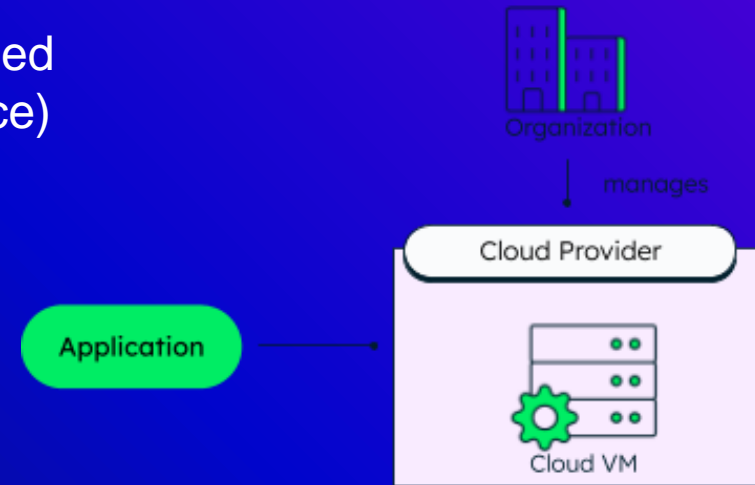
What is a Cloud Database?

- ❖ You can deploy any type of database in the cloud. This includes traditional SQL databases and more modern NoSQL types of databases. MongoDB Atlas is a general-purpose document database that can be deployed on any major cloud provider, like **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud**.



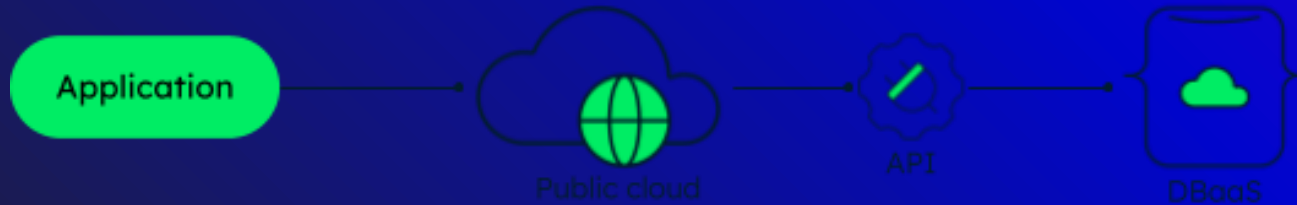
Cloud database deployment models

- ❖ Cloud databases can be deployed on a cloud provider's infrastructure (self-managed) or accessed as a service (fully-managed Database-as-a-Service) on a subscription basis.
- ❖ In a self-managed database, the organization's system administrators or software developers are responsible for the database management. They purchase virtual machine instances and run their database on the virtual machines. The cloud provider takes care of the infrastructure provisioning.



Cloud database deployment models

- ❖ In a fully-managed model, the database runs on the infrastructure of the cloud provider, and the backup, scaling, provisioning, security, and health monitoring are all taken care of by the cloud database provider. The database is accessed as a service by the organization. Fully-managed database services handle the complexities of maintaining a consistently available, high performance cluster in a way that allows you, the developer, to access it as a simple, globally available resource.



Why use a cloud database?

➤ Ease of access and agility:

- ❖ Whether your team is already developing software on cloud infrastructure, or you're in the process of migrating legacy applications to the cloud, it makes sense that cloud-native database offerings are growing in popularity.
- ❖ Modern Database-as-a-Service platforms enable easy (but controlled) access from cloud systems through consistent APIs and drivers, simplifying access to critical resources. Microservice architectures in particular benefit from centralized and easily accessible database resources, as many applications need to access and share data.

Why use a cloud database?

➤ Scalability and performance:

- ❖ The real test of a data management system is how it adapts and performs under high load.
- ❖ This is why cloud database services are typically designed to automatically scale up to accommodate data growth, and scale out to handle load with consistent performance characteristics. A good cloud database will automatically alert you of performance issues, so that you can optimize your indexes and access patterns in order to hit your performance targets.
- ❖ Not only is it usually cheaper to use a fully managed cloud database than to maintain your own, it requires less manual work, so you and your team can focus on delivering value.

Why use a cloud database?

➤ Reliability and disaster recovery:

- ❖ There's little worse than having a mission-critical database go offline—except to lose all of the data inside it.
- ❖ Cloud databases are usually replicated and backed up by default, so that no single point of failure can bring your application offline, and even catastrophic incidents are mitigated by regular, automated backup and disaster recovery.

Outlines

- Introduction.
- *Types*
- Cloud Document Databases
- MongoDB Use Cases

What is MongoDB – Working and Features

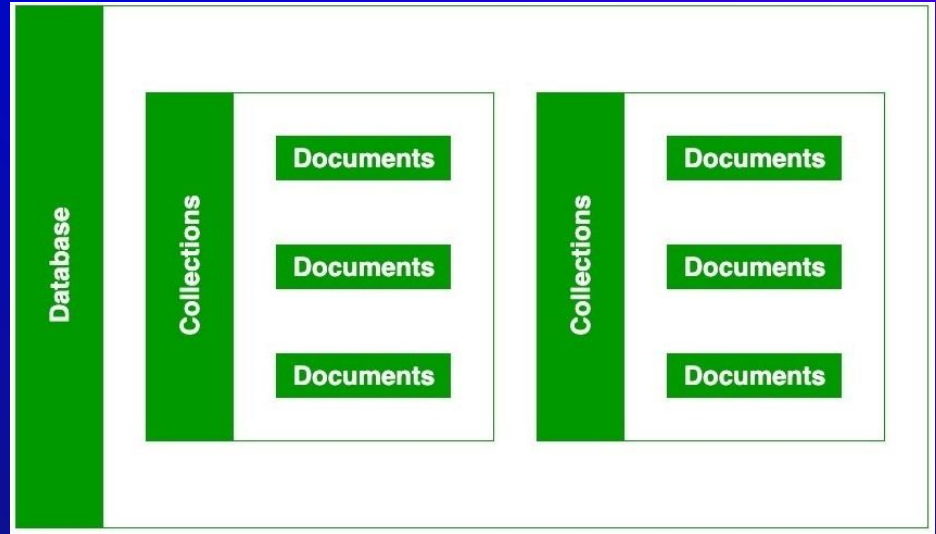
- MongoDB is an open-source document-oriented database that is designed to store a large scale of data and also allows you to work with that data very efficiently. It is categorized under the NoSQL (Not only SQL) database because the storage and retrieval of data in the MongoDB are not in the form of tables.
- The MongoDB database is developed and managed by MongoDB.Inc under SSPL(Server Side Public License) and initially released in February 2009. It also provides official driver support for all the popular languages like C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid. So, that you can create an application using any of these languages. Nowadays there are so many companies that used MongoDB like Facebook, Nokia, eBay, Adobe, Google, etc. to store their large amount of data.

How it works ?

- Now, we will see how actually thing happens behind the scene. As we know that MongoDB is a database server and the data is stored in these databases. Or in other words, the MongoDB environment gives you a server that you can start and then create multiple databases on it using MongoDB.
- Because of its NoSQL database, the data is stored in the collections and documents. Hence the database, collection, and documents are related to each other as shown below:

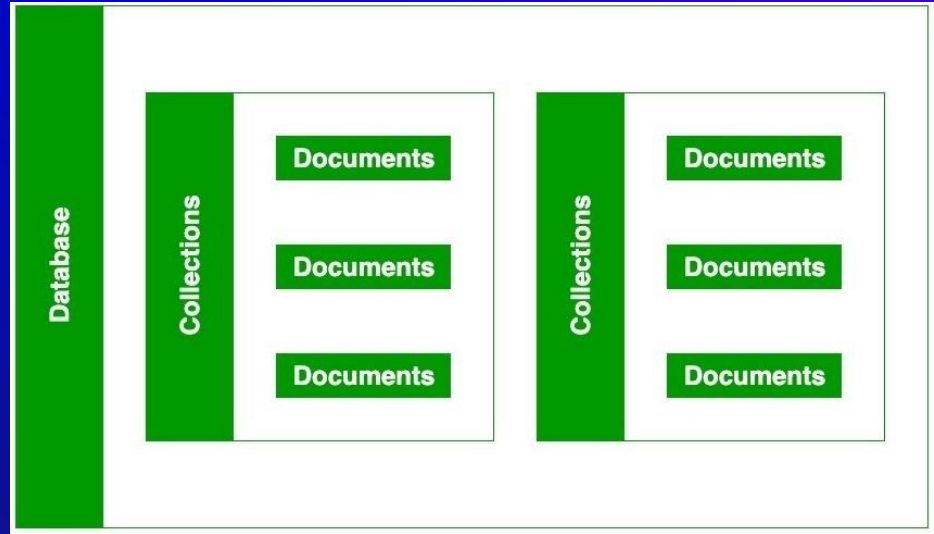
How it works ?

- The MongoDB database contains collections just like the MySQL database contains tables. You are allowed to create multiple databases and multiple collections.
- Now inside of the collection we have documents. These documents contain the data we want to store in the MongoDB database and a single collection can contain multiple documents and you are schema-less means it is not necessary that one document is similar to another.



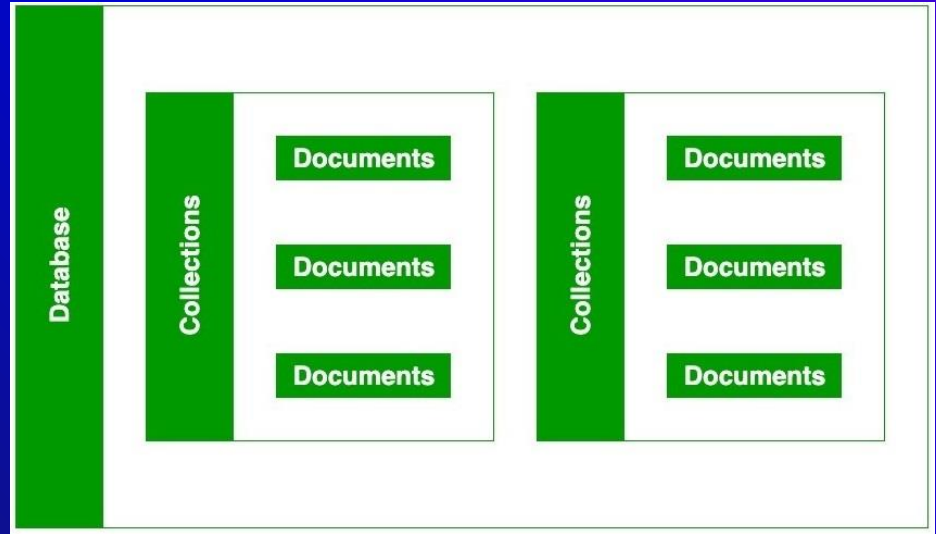
How it works ?

- The documents are created using the fields. Fields are key-value pairs in the documents, it is just like columns in the relation database. The value of the fields can be of any BSON data types like double, string, boolean, etc.
- The data stored in the MongoDB is in the format of BSON documents. Here, BSON stands for Binary representation of JSON documents. Or in other words, in the backend, the MongoDB server converts the JSON data into a binary form that is known as BSON and this BSON is stored and queried more efficiently.



How it works ?

- In MongoDB documents, you are allowed to store nested data. This nesting of data allows you to create complex relations between data and store them in the same document which makes the working and fetching of data extremely efficient as compared to SQL. In SQL, you need to write complex joins to get the data from table 1 and table 2. The maximum size of the BSON document is 16MB.



NOTE: In MongoDB server, you are allowed to run multiple databases.

What are the challenges of managing and storing unstructured data

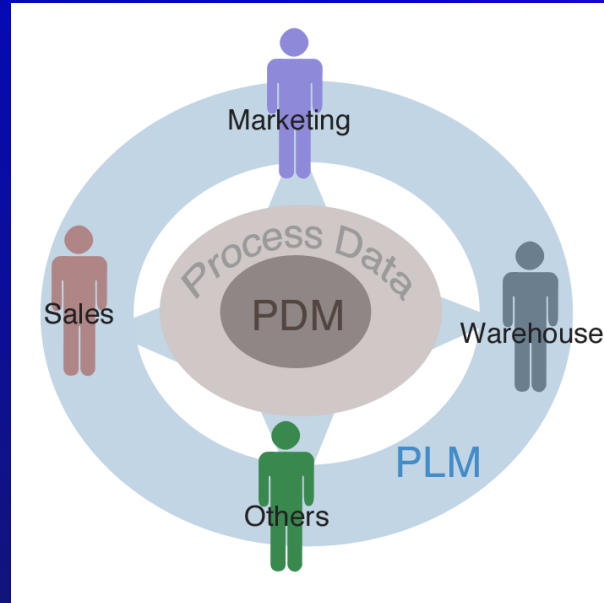
- Unstructured data cannot be forced to conform to the columns and rows format of a traditional relational database. Some relational databases provide support for the BLOB (Binary Large Object) type allowing storage of unstructured data but offer little additional functionality; you can store and retrieve blobs, but you still cannot query it well. You must also define the structure of data before it can be written to the database.
- An unstructured database like MongoDB takes a different approach to data storage. Text files and other unstructured assets are stored as JSON formatted documents.
- Because of the sheer physical volumes of data involved, NoSQL databases can scale infinitely. By building on top of a data lake or similar, capacity can be added quickly using inexpensive commodity hardware. This will be essential when dealing with real-time data like social media updates or IoT sensor feedback.
- The MongoDB NoSQL engine can also be connected to your AWS, Azure, or Google Cloud platforms for maximum scalability.

Outlines

- Introduction.
- *Types*
- Cloud Document Databases
- MongoDB Use Cases

MongoDB Use Case 1: Product Data Management

- MongoDB is perfect for Product Data Management. It enables product data and related information to be managed and processed in a single, central system. This allows for Detailed Cost Analysis, Increased Productivity, and Improved Collaboration.



MongoDB Use Case 1: Product Data Management

➤ Step 1: Connect to MongoDB



```
import pymongo
# Replace "mongodb://localhost:27017/" with your MongoDB connection string
client = pymongo.MongoClient("mongodb://localhost:27017/")
```

MongoDB Use Case 1: Product Data Management

- Step 2: Create or access the database



```
db = client["product_data_management"]
```


MongoDB Use Case 1: Product Data Management

- Step 3: Design the data model
- Define the structure of your documents. In this use case, we'll design a simple product data model containing information like product name, price, description, and stock quantity. Each product will have a unique identifier (e.g., a SKU or a generated ObjectId) as the primary key.

```
product_model = {  
    "_id": None, # Primary key, you can use a unique identifier or let MongoDB generate it (e.g., ObjectId)  
    "name": "",  
    "price": 0.0,  
    "description": "",  
    "stock_quantity": 0,  
    # Add more fields as needed (e.g., categories, manufacturer, etc.)  
}
```

MongoDB Use Case 1: Product Data Management

- Step 4: Create a collection A collection is a group of MongoDB documents. Create a collection for products:



```
products_collection =  
db["products"]
```


MongoDB Use Case 1: Product Data Management

- Step 5: Insert data into the collection
- Now, you can insert product data into the collection using the `insert_one` or `insert_many` method:

```
# Single product insertion
product_data = {
    "name": "Product 1",
    "price": 19.99,
    "description": "This is a sample product.",
    "stock_quantity": 100,
}
result =
product.insert_one(product_data)
```

MongoDB Use Case 1: Product Data Management

- Step 8: Query data from the collection
- Retrieve product data using the “find_one” or “find” method:

```
# Get a single product by name
product = products_collection.find_one({"name": "Product 1"})
if product:
    print("Product found:")
    print(product)
else:
    print("Product not found.")

# Get all products
all_products = products_collection.find()
for product in all_products:
    print(product)
```

MongoDB Use Case 1: Product Data Management

- Step 9: Update data in the collection
- Update existing product data using the `update_one` or `update_many` method:



```
# Update product price for a specific product
products_collection.update_one(
    {"name": "Product 1"},
    {"$set": {"price": 24.99}}
)
```

MongoDB Use Case 1: Product Data Management


- Step 10: Delete data from the collection
- Remove products from the collection using the `delete_one` or `delete_many` method:



```
# Delete a specific product
products_collection.delete_one({"name": "Product
1"})
```

MongoDB Use Case 1: Product Data Management

- Step 11: Fetch data from MongoDB and create a Pandas DataFrame
- Remove products from the collection using the `delete_one` or `delete_many` method:



```
# Step 13: Fetch data from MongoDB and create a Pandas DataFrame
all_products = products_collection.find()
product_data_list = list(all_products)
df = pd.DataFrame(product_data_list)
```

MongoDB Use Case 1: Product Data Management

- Step 12: Close the MongoDB connection
- Don't forget to close the connection when you're done:



```
client.close()
```

MongoDB Use Case 2: Operational Intelligence

- Another real-world MongoDB use case is Operational Intelligence, as it aids in Real-time Decision-making. It allows companies to seamlessly gather various data feeds representing their ongoing business operations and information of related external factors. They can then analyze these feeds as the data arrives for developing profitable and functional business strategies.



MongoDB Use Case 2: Operational Intelligence

➤ Step 1: Connect to MongoDB



```
import pymongo
# Replace "mongodb://localhost:27017/" with your MongoDB connection string
client = pymongo.MongoClient("mongodb://localhost:27017/")
```


MongoDB Use Case 2: Operational Intelligence

- Step 2: Create or access the database



```
db =  
client["operational_intelligence"]
```


MongoDB Use Case 2: Operational Intelligence

- Step 3: Design the data model
- Define the structure of your documents. In this use case of Operational Intelligence, the data model will depend on the specific metrics and events you want to monitor and analyze. As an example, we'll consider a simple data model for tracking server performance metrics. Each document will represent a server's performance snapshot and will contain metrics like CPU usage, memory usage, disk usage, timestamp, etc.

```
performance_model = {  
    "server_id": None, # Identifier for the server  
    "cpu_usage": 0.0, # CPU usage percentage  
    "memory_usage": 0.0, # Memory usage percentage  
    "disk_usage": 0.0, # Disk usage percentage  
    "timestamp": None, # Timestamp of the performance snapshot  
    # Additional metrics and fields can be added as needed  
}
```

MongoDB Use Case 2: Operational Intelligence

- Step 4: Create a collection A collection is a group of MongoDB documents:



```
performance_collection = db["server_performance"]
```

MongoDB Use Case 2: Operational Intelligence

- Step 5: Insert data into the collection
- Now, you can insert data into the collection using the `insert_one` or `insert_many` method:

```
# Example performance data for a server
performance_data = {
    "server_id": "server-01",
    "cpu_usage": 40.5,
    "memory_usage": 75.2,
    "disk_usage": 90.0,
    "timestamp": "2023-07-29T12:00:00",
}
result = performance_collection.insert_one(performance_data)
print(f"Inserted performance snapshot ID: {result.inserted_id}")
```

MongoDB Use Case 2: Operational Intelligence

- Step 8: Query data from the collection
- Retrieve data using the “find_one” or “find” method:

```
# Get the latest performance snapshot for a specific server
latest_performance = performance_collection.find_one(
    {"server_id": "server-01"},
    sort=[("timestamp", pymongo.DESCENDING)]
)
print("Latest performance snapshot:")
print(latest_performance)

# Get all performance snapshots for a specific server within a time range
from datetime import datetime

start_time = datetime(2023, 7, 29, 0, 0, 0)
end_time = datetime(2023, 7, 29, 23, 59, 59)

performance_snapshots = performance_collection.find(
    {"server_id": "server-01", "timestamp": {"$gte": start_time, "$lte":
end_time}}
)
for snapshot in performance_snapshots:
    print(snapshot)
```


MongoDB Use Case 2: Operational Intelligence

- Step 9: Update data in the collection
- Update existing data using the `update_one` or `update_many` method:

```
# Update CPU usage for a specific server's performance snapshot
performance_collection.update_one(
    {"server_id": "server-01", "timestamp": "2023-07-29T12:00:00"},
    {"$set": {"cpu_usage": 50.2}}
)
```

MongoDB Use Case 2: Operational Intelligence


- Step 10: Delete data from the collection
- Remove from the collection using the `delete_one` or `delete_many` method:



```
# Delete a specific performance snapshot
performance_collection.delete_one({"server_id": "server-01", "timestamp": "2023-07-29T12:00:00"})
```

MongoDB Use Case 2: Operational Intelligence

- Step 11: Fetch data from MongoDB and create a Pandas DataFrame
- Remove products from the collection using the `delete_one` or `delete_many` method:



```
performance_snapshots = performance_collection.find()  
performance_data_list = list(performance_snapshots)  
df = pd.DataFrame(performance_data_list)
```


MongoDB Use Case 2: Operational Intelligence

- Step 12: Close the MongoDB connection
- Don't forget to close the connection when you're done:



```
client.close()
```

MongoDB Use Case 3: Product Catalog

- Product catalogs have been in existence for years in the ever-evolving digital space. However, with the rapid evolution in technology, product catalogs sometimes feel like a new digital experience. This is because the richness and volume of data feed product catalogs' interactions today are remarkable.
- MongoDB is useful in such applications, as it provides an excellent tool for storing different types of objects. In addition, its Dynamic Schema Capability ensures that product documents only contain attributes relevant to that product.


MongoDB Use Case 3: Product Catalog

➤ Step 1: Connect to MongoDB



```
import pymongo
# Replace "mongodb://localhost:27017/" with your MongoDB connection string
client = pymongo.MongoClient("mongodb://localhost:27017/")
```

➤ Step 2: Create or access the database



```
db = client["product_catalog"]
```


MongoDB Use Case 3: Product Catalog

- Step 3: Design the data model
- Define the structure of your documents. In this use case, the data model will represent the products in the catalog. Each product will have a unique identifier (e.g., a SKU or a generated ObjectId) as the primary key, along with other attributes such as product name, description, price, and category.

```
product_model = {  
    "_id": None,          # Primary key, you can use a unique identifier or let MongoDB generate it (e.g., ObjectId)  
    "name": "",          # Name of the product  
    "description": "",   # Description of the product  
    "price": 0.0,        # Price of the product  
    "category": "",      # Category of the product  
    # Additional attributes as needed (e.g., brand, stock_quantity, etc.)  
}
```

MongoDB Use Case 3: Product Catalog

- Step 4: Create a collection A collection is a group of MongoDB documents. Create a collection for products:



```
products_collection =  
db["products"]
```

MongoDB Use Case 3: Product Catalog

- Step 5: Insert data into the collection
- Now, you can insert product data into the collection using the `insert_one` or `insert_many` method:

```
# Example product data for insertion
product_data = {
    "name": "Product 1",
    "description": "This is a sample product.",
    "price": 19.99,
    "category": "Electronics",
}

result =
print(f"Inserted product ID: {product_data['_id']}")
```

MongoDB Use Case 3: Product Catalog

- Step 8: Query data from the collection
- Retrieve product data using the “find_one” or “find” method:

```
# Get a single product by name
product = products_collection.find_one({"name": "Product 1"})
if product:
    print("Product found:")
    print(product)
else:
    print("Product not found.")

# Get all products in a specific category
category_products = products_collection.find({"category":
"Electronics"})
for product in category_products:
    print(product)
```


MongoDB Use Case 3: Product Catalog

- Step 9: Update data in the collection
- Update existing product data using the `update_one` or `update_many` method:

```
# Update the price of a specific product
products_collection.update_one(
    {"name": "Product 1"},
    {"$set": {"price": 24.99}}
)
```

MongoDB Use Case 3: Product Catalog


- Step 10: Delete data from the collection
- Remove products from the collection using the `delete_one` or `delete_many` method:



```
# Delete a specific product
products_collection.delete_one({"name": "Product
1"})
```

MongoDB Use Case 3: Product Catalog

- Step 11: Fetch data from MongoDB and create a Pandas DataFrame
- Remove products from the collection using the `delete_one` or `delete_many` method:



```
# Fetch data from MongoDB and create a Pandas DataFrame
products_cursor = products_collection.find()
product_data_list = list(products_cursor)
df = pd.DataFrame(product_data_list)
```

MongoDB Use Case 3: Product Catalog

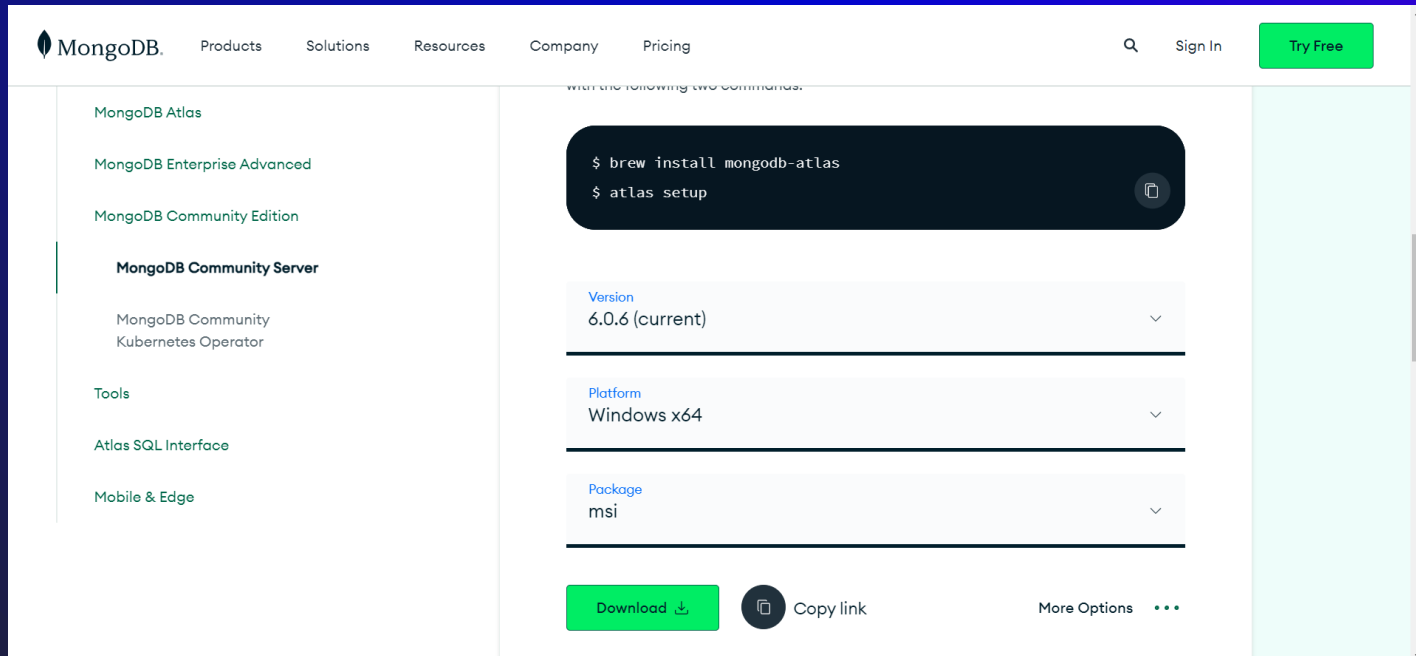
- Step 12: Close the MongoDB connection
- Don't forget to close the connection when you're done:



```
client.close()
```

Tools we need.

🟡 MongoDB: <https://www.mongodb.com/try/download/community>



The screenshot shows the MongoDB website's download page for the Community Edition Server. The left sidebar contains a navigation menu with links to MongoDB Atlas, MongoDB Enterprise Advanced, MongoDB Community Edition, MongoDB Community Server (which is highlighted), MongoDB Community Kubernetes Operator, Tools, Atlas SQL Interface, and Mobile & Edge. The main content area displays the text "with the following two commands:" above a dark terminal window containing the commands: `$ brew install mongodb-atlas` and `$ atlas setup`. Below this, there are three dropdown menus for selecting the version (6.0.6 (current)), platform (Windows x64), and package (msi). At the bottom of the main content area, there are buttons for "Download" (with a download icon), "Copy link" (with a copy icon), and "More Options" (with a three-dot menu icon).

MongoDB

Products Solutions Resources Company Pricing

Search Sign In Try Free

MongoDB Atlas

MongoDB Enterprise Advanced

MongoDB Community Edition

MongoDB Community Server

MongoDB Community Kubernetes Operator

Tools

Atlas SQL Interface

Mobile & Edge

with the following two commands:

```
$ brew install mongodb-atlas
$ atlas setup
```

Version
6.0.6 (current)

Platform
Windows x64

Package
msi

Download

Copy link

More Options

Tools we need.

🟡 MongoDB Shell: <https://www.mongodb.com/try/download/shell>

MongoDB Atlas

MongoDB Enterprise Advanced

MongoDB Community Edition

Tools

MongoDB Shell

MongoDB Compass (GUI)

Atlas CLI

Atlas Kubernetes Operator

MongoDB CLI for Cloud Manager and Ops Manager

MongoDB Cluster-to-Cluster Sync

Note: MongoDB Shell is an open source (Apache 2.0), standalone product developed separately from the MongoDB Server.

Learn more

Version

1.10.1

▼

Platform

Windows 64-bit (8.1+)

▼

Package

zip

▼

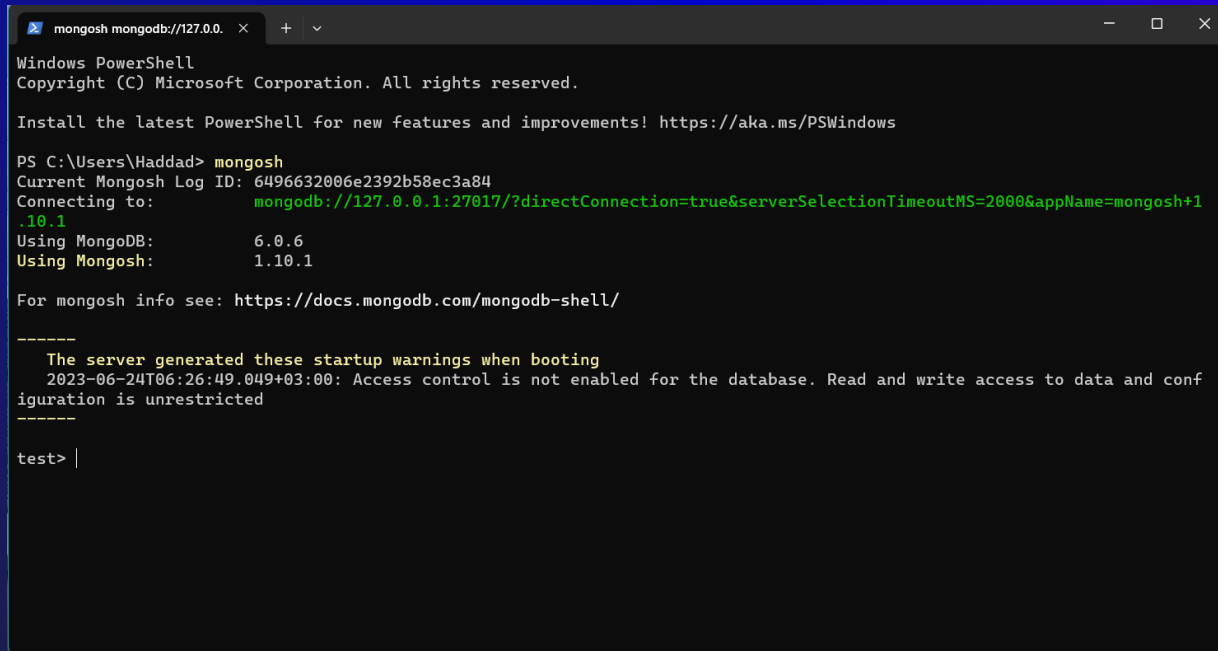
Download

Copy link

More Options

MongoDB

- ◊ To make sure that you installed it well open a terminal and write “mongosh” it will appear the version of the program and the shell



```
mongosh mongodb://127.0.0.1:27017/7
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Haddad> mongosh
Current Mongosh Log ID: 6496632006e2392b58ec3a84
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.10.1
Using MongoDB:      6.0.6
Using Mongosh:      1.10.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
  The server generated these startup warnings when booting
  2023-06-24T06:26:49.049+03:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

test> |
```

MongoDB

Connect to Database using Mongo compass

Connect Edit View Help

Compass



New connection +

 Saved connections

 Recents

New Connection

Connect to a MongoDB deployment



FAVORITE

URI ⓘ

Edit Connection String ☒

mongodb://localhost:27017/

➤ Advanced Connection Options

Save

Save & Connect

Connect

New to Compass and don't have a cluster?

If you don't already have a cluster, you can create one for free using [MongoDB Atlas](#)

[CREATE FREE CLUSTER](#)

How do I find my connection string in Atlas?

If you have an Atlas cluster, go to the Cluster view. Click the 'Connect' button for the cluster to which you wish to connect.

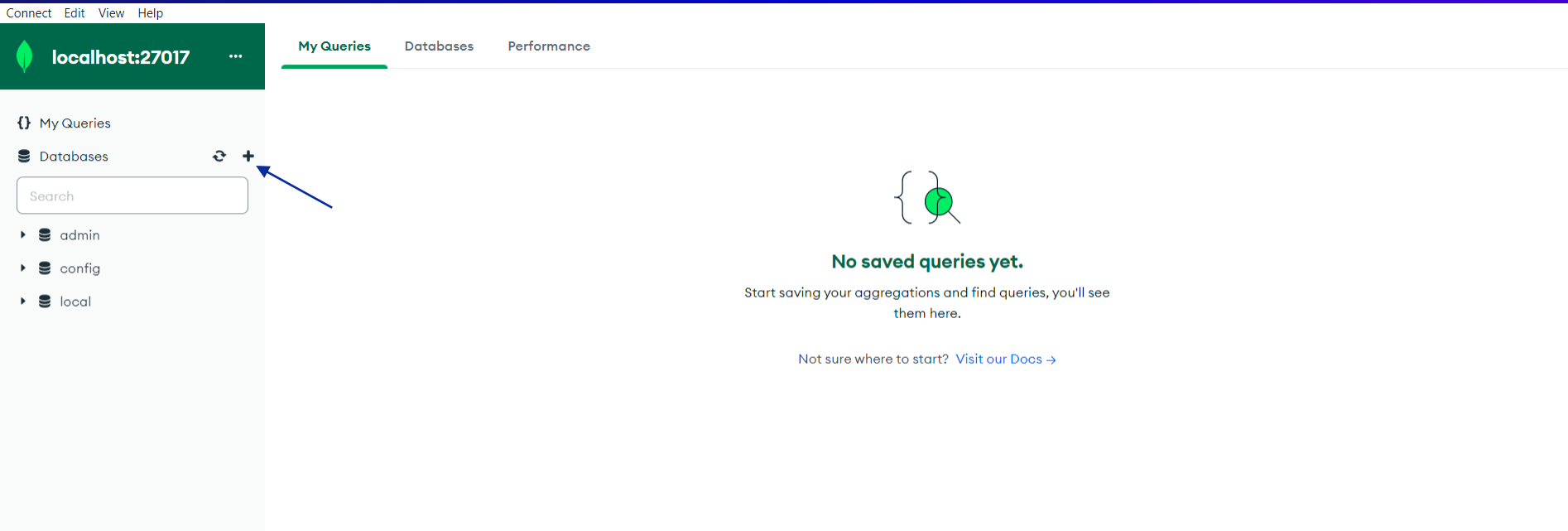
[See example](#)

How do I format my connection string?

[See example](#)

MongoDB

Create a new Database



The screenshot shows the MongoDB Compass web interface. At the top, there's a navigation bar with 'Connect', 'Edit', 'View', and 'Help'. Below this, a green header bar displays 'localhost:27017' and a menu icon. The main content area has three tabs: 'My Queries' (active), 'Databases', and 'Performance'. On the left sidebar, under 'My Queries', there's a 'Databases' section with a search bar and a list of databases: 'admin', 'config', and 'local'. A blue arrow points to a '+' icon next to the 'Databases' section header. The main area shows a message: 'No saved queries yet. Start saving your aggregations and find queries, you'll see them here.' with a link to 'Visit our Docs'.

Connect Edit View Help

localhost:27017 ...

My Queries Databases Performance

{ } My Queries

Databases

Search

- admin
- config
- local

No saved queries yet.

Start saving your aggregations and find queries, you'll see them here.

Not sure where to start? [Visit our Docs](#) →

MongoDB

🛡️ Create a new Database

Create Database ✕

Database Name

Bookstore

Collection Name

Books

☐ **Time-Series**
Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

> Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Cancel Create Database

MongoDB

🏠 Create a new Database

The screenshot shows the MongoDB Compass web interface. At the top, the connection is to **localhost:27017**. The left sidebar shows the database structure: **Bookstore** > **Books**. The main panel displays the **Documents** tab for the **Bookstore.Books** collection. A blue arrow points from the text "Documents" to the "Documents" tab. The interface shows 0 documents and 1 index. A search bar is present with the placeholder "Type a query: { field: 'value' }". Below the search bar are buttons for "ADD DATA" and "EXPORT DATA". At the bottom, a message states "This collection has no data" with a subtext "It only takes a few seconds to import data from a JSON or CSV file." and an "Import Data" button.

MongoDB

🛡️ Add a document

The screenshot shows the MongoDB Compass web interface. At the top, there's a navigation bar with 'Connect', 'Edit', 'View', 'Collection', and 'Help'. Below this, a green header bar displays 'localhost:27017' and a dropdown menu for 'Documents' showing 'Bookstore.Books'. The left sidebar contains 'My Queries', 'Databases', and a tree view of collections under 'Bookstore', including 'Books', 'admin', 'config', and 'local'. The main area is titled 'Bookstore.Books' and shows '0 DOCUMENTS' and '1 INDEXES'. Below the title are tabs for 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. The 'Documents' tab is active, showing a search bar with a filter icon and a query input field containing '{ field: 'value' }'. Below the search bar are 'ADD DATA' and 'EXPORT DATA' buttons. At the bottom, a message states 'This collection has no data' with a subtext 'It only takes a few seconds to import data from a JSON or CSV file.' and an 'Import Data' button.

MongoDB

🟡 Add a document

The screenshot displays the MongoDB Compass web interface. At the top, a green header bar shows 'localhost:27017'. Below this, a sidebar on the left lists databases: 'Bookstore', 'admin', 'config', and 'local'. The main content area is titled 'Bookstore.Books' and shows '0 DOCUMENTS' and '1 INDEXES'. A tabbed interface below the title includes 'Documents', 'Aggregations', 'Schema', 'Explain Plan', 'Indexes', and 'Validation'. The 'Documents' tab is active, showing a search bar with the placeholder 'Type a query: { field: 'value' }' and buttons for 'Filter', 'Reset', 'Find', and 'Options'. Below the search bar are buttons for 'ADD DATA' and 'EXPORT DATA'. At the bottom of the main area, a message states 'This collection has no data' with a subtext 'It only takes a few seconds to import data from a JSON or CSV file.' and an 'Import Data' button. The bottom status bar shows '>_MONGOSH'.

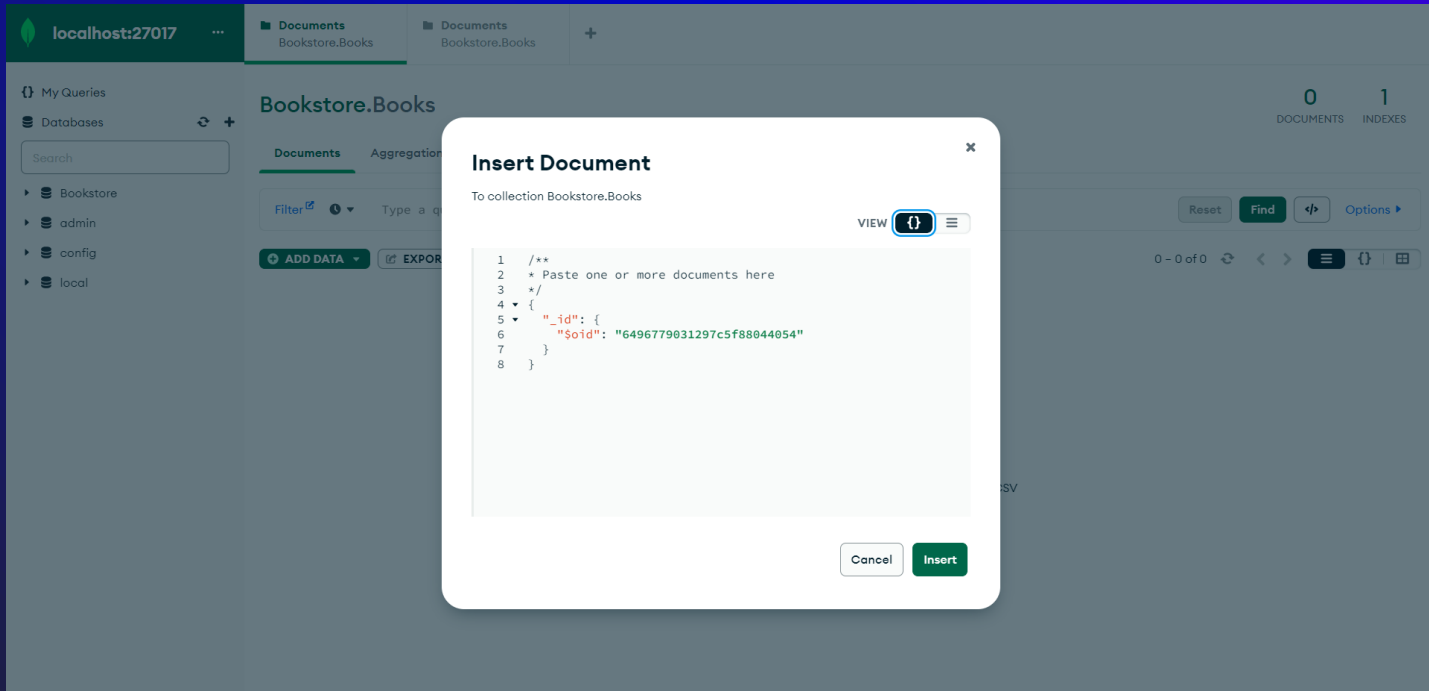
MongoDB

Insert Data into document

The screenshot shows the MongoDB Compass interface for a local instance at localhost:27017. The left sidebar displays a database named 'Bookstore' with collections 'admin', 'config', and 'local'. The main panel is focused on the 'Bookstore.Books' collection, showing 0 documents and 1 index. The 'Documents' tab is active, displaying a filter bar with a query field and buttons for 'Reset', 'Find', and 'Options'. Below the filter bar, the 'ADD DATA' dropdown menu is open, showing options to 'Import JSON or CSV file' and 'Insert document'. A blue arrow points to the 'Insert document' option. The main content area displays a message: 'This collection has no data. It only takes a few seconds to import data from a JSON or CSV file.' with an 'Import Data' button.

MongoDB

⬡ Here you can type the data from JSON type.



MongoDB

🟡 After adding the data click on “Insert”

The screenshot shows the MongoDB Compass interface. In the background, the 'Bookstore.Books' collection is selected, showing 0 documents and 1 index. The 'Insert Document' dialog box is open, displaying the following JSON data:

```
1  [{
2    "title": "1984",
3    "author": "George Orwell",
4    "publication_year": 1949,
5    "isbn": "9780451524935",
6    "genres": ["Dystopian", "Political Fiction"],
7    "publisher": "Signet Classics",
8    "language": "English",
9    "pages": 328,
10   "price": 8.99,
11   "availability": true
12 }, {
13   "title": "To Kill a Mockingbird",
14   "author": "Harper Lee",
15   "publication_year": 1960,
16   "isbn": "9780061120884",
17   "genres": ["Classic", "Coming-of-Age"],
18   "publisher": "Harper Perennial Modern Classics",
19   "language": "English",
20   "pages": 336,
21   "price": 12.99,
22   "availability": true
23 }, {
24   "title": "The Hobbit",
25   "author": "J.R.R. Tolkien",
```

The dialog box has 'Cancel' and 'Insert' buttons at the bottom right.

MongoDB

🟡 Or we can do this using mongo Shell

```
test> use bookstore
switched to db bookstore
bookstore> db
bookstore
bookstore> db.books
bookstore.books
bookstore> db.books.insertOne({title: "The Color of Magic", author: "Terry Pratchett", pages: 300, rating: 7, genres: ["fantasy", "magic"]})
{
  acknowledged: true,
  insertedId: ObjectId("622b2bf63ef6e94f63150666")
}
bookstore> db.authors.insertOne({ name: "Brandon Sanderson", age: 60 })
{
  acknowledged: true,
  insertedId: ObjectId("622b2c473ef6e94f63150667")
}
bookstore>
```

MongoDB

🟡 Finding Documents

The screenshot shows the MongoDB Compass web interface. The top bar indicates the connection to `localhost:27017`. The left sidebar shows the database structure with `Bookstore` expanded and `Books` selected. The main panel displays the `Bookstore.Books` collection with 3 documents and 1 index. A filter is applied: `{language: "English"}`. Below the filter, there are buttons for `ADD DATA` and `EXPORT DATA`. The results show two documents:

```
{
  "_id": ObjectId('6496788b31297c5f88044056'),
  "title": "To Kill a Mockingbird",
  "author": "Harper Lee",
  "publication_year": 1960,
  "isbn": "9780061120084",
  "genres": Array,
  "publisher": "Harper Perennial Modern Classics",
  "language": "English",
  "pages": 336,
  "price": 12.99,
  "availability": true
}
```

```
{
  "_id": ObjectId('6496788b31297c5f88044057'),
  "title": "The Hobbit",
  "author": "J.R.R. Tolkien",
  "publication_year": 1937,
  "isbn": "9780261102217",
  "genres": Array,
  "publisher": "HarperCollins",
  "language": "English",
  "pages": 310,
  "price": 10.99,
  "availability": true
}
```

MongoDB

🟡 Finding Documents

```
bookstore> db.books.find({author: "Terry Pratchett"})
[
  {
    _id: ObjectId("622b2bf63ef6e94f63150666"),
    title: 'The Color of Magic',
    author: 'Terry Pratchett',
    pages: 300,
    rating: 7,
    genres: [ 'fantasy', 'magic' ]
  },
  {
    _id: ObjectId("622b2fc1d28eddc86c808d2f"),
    title: 'The Light Fantastic',
    author: 'Terry Pratchett',
    pages: 250,
    rating: 6,
    genres: [ 'fantasy' ]
  }
]
bookstore> |
```

MongoDB

🟡 Limit

```
bookstore> db.books.find().limit(3)
[
  {
    _id: ObjectId("622a3bc0f493f1330ba0cd37"),
    title: 'Name of the Wind',
    author: 'Patrick Rothfuss',
    pages: 600,
    genres: [ 'fantasy', 'magical' ],
    rating: 9
  },
  {
    _id: ObjectId("622a3d1df493f1330ba0cd3b"),
    title: 'The Final Empire',
    author: 'Brandon Sanderson',
    pages: 450,
    genres: [ 'fantasy', 'dystopian' ],
    rating: 8
  },
  {
    _id: ObjectId("622a3d1df493f1330ba0cd3c"),
    title: 'The Way of Kings',
    author: 'Brandon Sanderson',
    pages: 350,
    genres: [ 'fantasy', 'dystopian' ],
    rating: 9
  }
]
```

MongoDB

- Sorting - 1 for ascending order and -1 for descending order

```
bookstore> db.books.find().sort({ title: 1 })
[
  {
    _id: ObjectId("622b2fc1d28eddc86c808d30"),
    title: 'Dune',
    author: 'Frank Herbert',
    pages: 500,
    rating: 10,
    genres: [ 'sci-fi', 'dystopian' ]
  },
  {
    _id: ObjectId("622a3bc0f493f1330ba0cd37"),
    title: 'Name of the Wind',
    author: 'Patrick Rothfuss',
    pages: 600,
    genres: [ 'fantasy', 'magical' ],
    rating: 9
  },
  {
    _id: ObjectId("622a3d1df493f1330ba0cd3d"),
    title: 'The Call of the Weird',
    author: 'Louis Theroux',
    pages: 350,
    genres: [ 'non-fiction', 'strange', 'comedy' ],

```

MongoDB

Operator - greater than 7

```
bookstore> db.books.find({ rating: {$gt: 7}})
[
  {
    _id: ObjectId("622b70c1d28eddc86c808d31"),
    title: 'The Way of Kings',
    author: 'Brandon Sanderson',
    rating: 9,
    pages: 400,
    genres: [ 'fantasy' ],
    reviews: [
      { name: 'Yoshi', body: 'Great book!!' },
      { name: 'mario', body: 'so so' }
    ]
  },
  {
    _id: ObjectId("622b746cd28eddc86c808d3f"),
    title: 'The Name of the Wind',
    author: 'Patrick Rothfuss',
    pages: 500,
```

MongoDB

◻ Logical operators

```
bookstore> db.books.find({$or: [{rating: 7}, {rating: 9}]})
[
  {
    _id: ObjectId("622b70c1d28eddc86c808d31"),
    title: 'The Way of Kings',
    author: 'Brandon Sanderson',
    rating: 9,
    pages: 400,
    genres: [ 'fantasy' ],
    reviews: [
      { name: 'Yoshi', body: 'Great book!!' },
      { name: 'mario', body: 'so so' }
    ]
  },
  {
    _id: ObjectId("622b746cd28eddc86c808d3e"),
    title: 'The Light Fantastic',
    author: 'Terry Pratchett',
    pages: 250,
    rating: 7,
    genres: [ 'fantasy', 'magic' ],
    reviews: [
```

MongoDB

Complex query

```
bookstore> db.books.find({$or: [{pages: {$lt: 300}}, {pages: {$gt: 400}}]})
[
  {
    _id: ObjectId("622b746cd28eddc86c808d3e"),
    title: 'The Light Fantastic',
    author: 'Terry Pratchett',
    pages: 250,
    rating: 7,
    genres: [ 'fantasy', 'magic' ],
    reviews: [
      { name: 'luigi', body: 'it was pretty good' },
      { name: 'bowser', body: 'loved it!!' }
    ]
  },
  {
    _id: ObjectId("622b746cd28eddc86c808d3f"),
    title: 'The Name of the Wind',
    author: 'Patrick Rothfuss',
    pages: 500,
    rating: 10,
    genres: [ 'fantasy' ],
    reviews: [ { name: 'peach', body: 'one of my favs' } ]
  }
]
```


MongoDB

🟡 \$in and \$nin

```
bookstore> db.books.find({ rating: {$in: [7,8,9]}})
[
  {
    _id: ObjectId("622b70c1d28eddc86c808d31"),
    title: 'The Way of Kings',
    author: 'Brandon Sanderson',
    rating: 9,
```



Thank
you!

