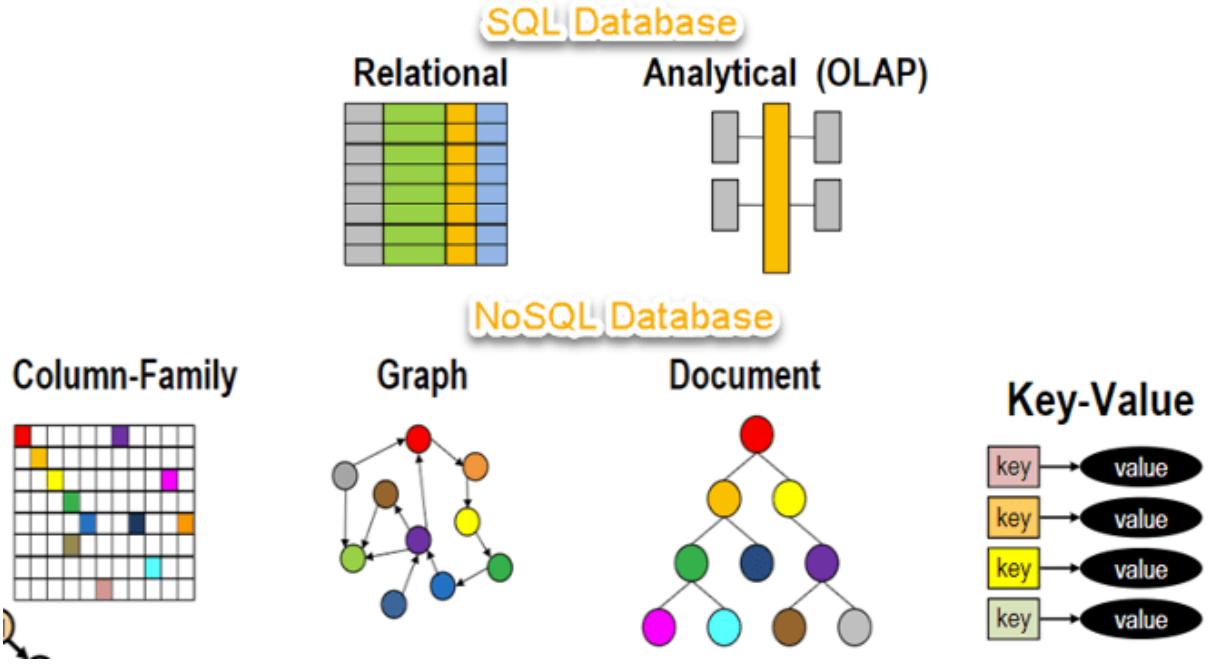
❖ Overview of NoSQL

What is NoSQL? : NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for “Not Only SQL” or “Not SQL.” Though a better term would be “NoREL”, NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let’s understand about NoSQL with a diagram in this NoSQL database tutorial:

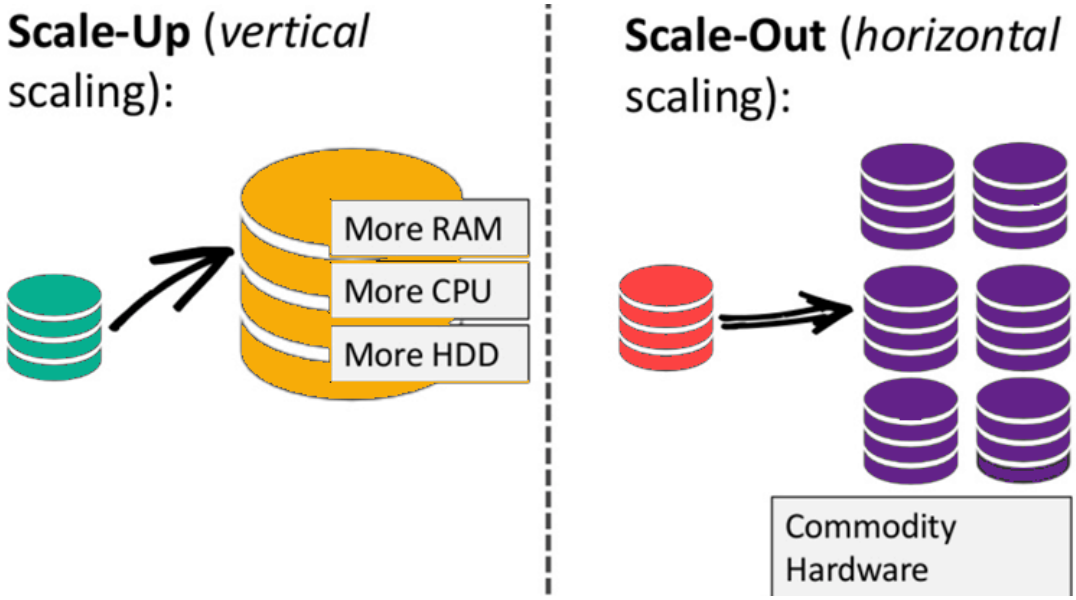


Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.

> The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as “scaling out.”



NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released
- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

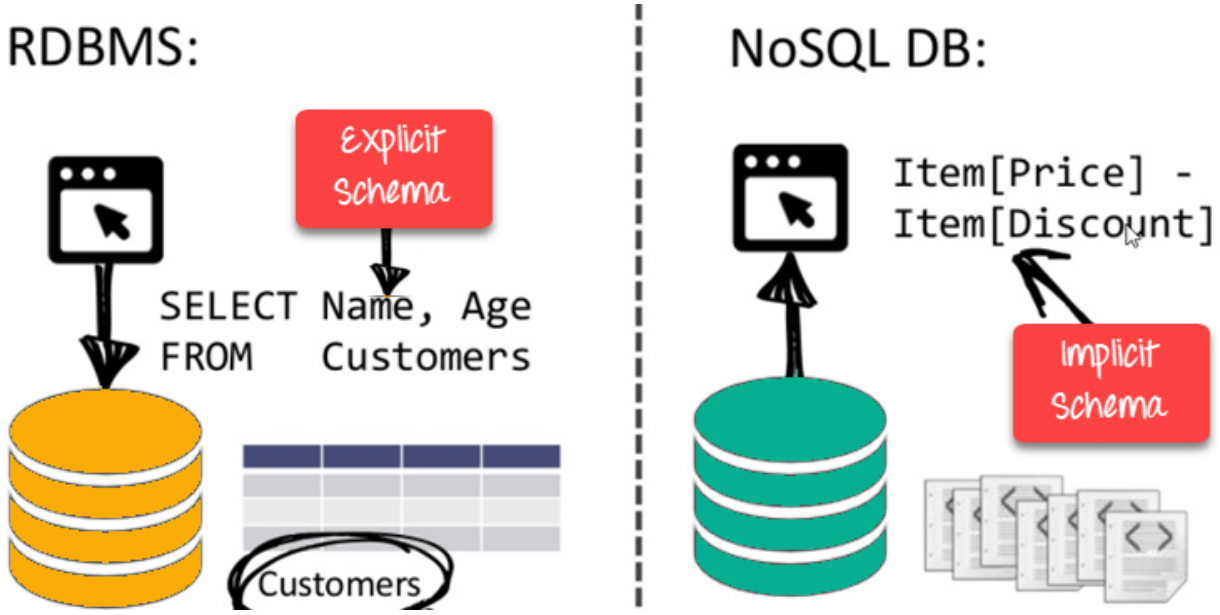
Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain



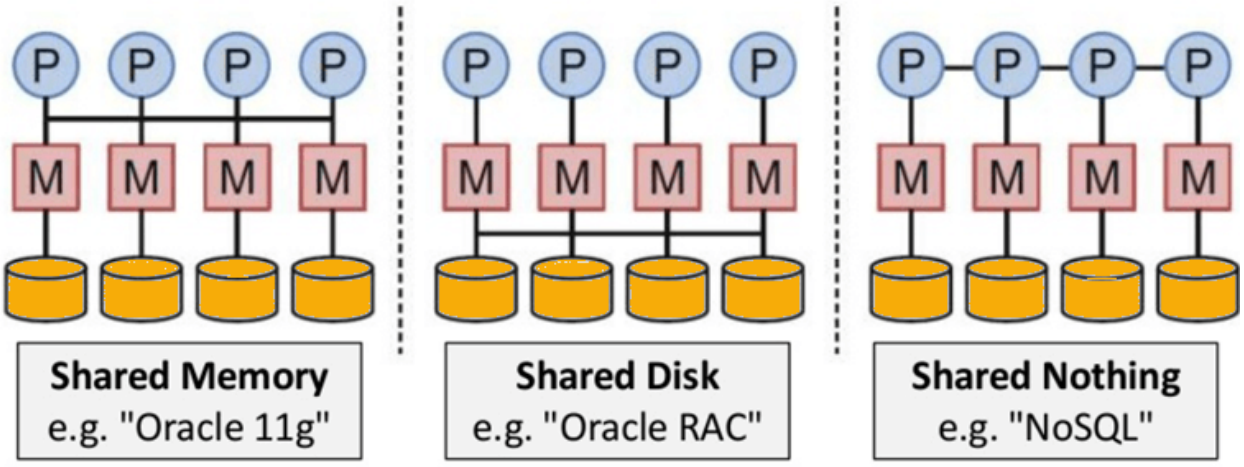
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

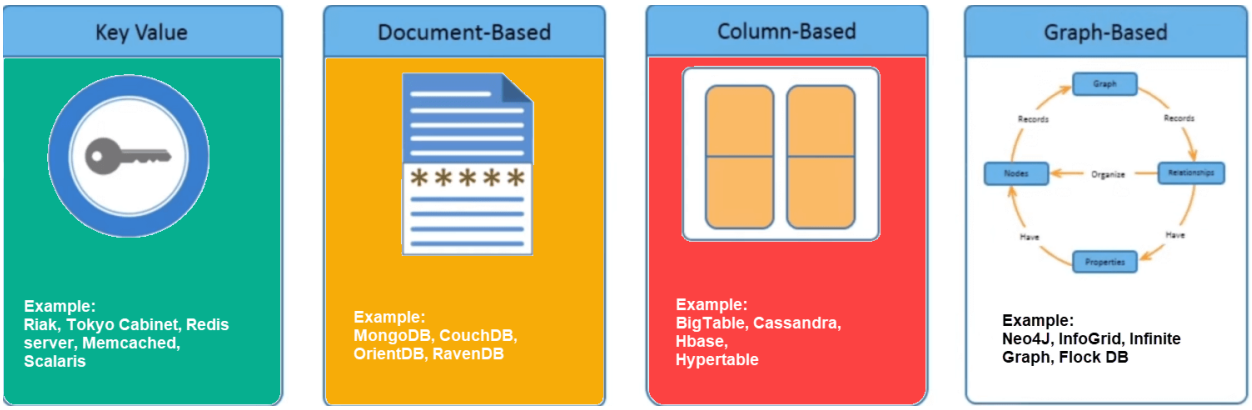


NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.
Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented



Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load. Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.
For example, a key-value pair may contain a key like “Website” associated with a value like “Guru99”.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.
Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

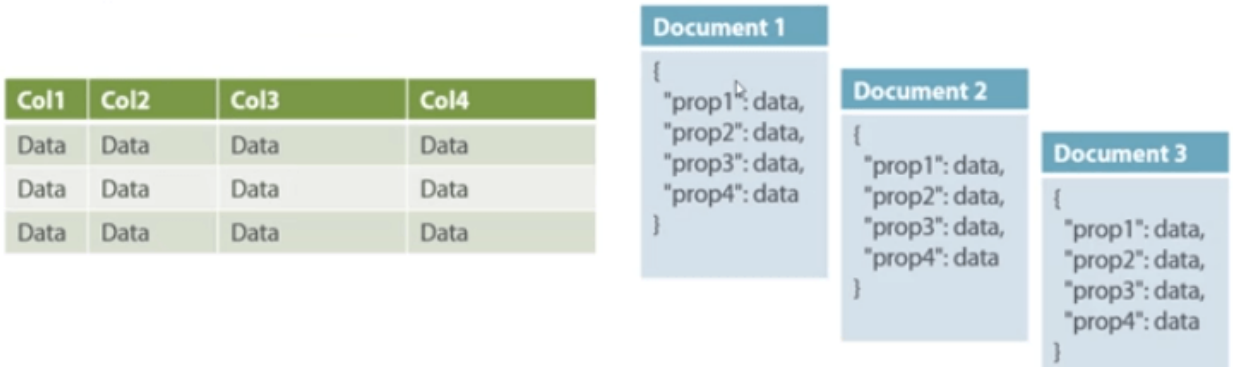
ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based NoSQL database

They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,
HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.



Relational Vs. Document

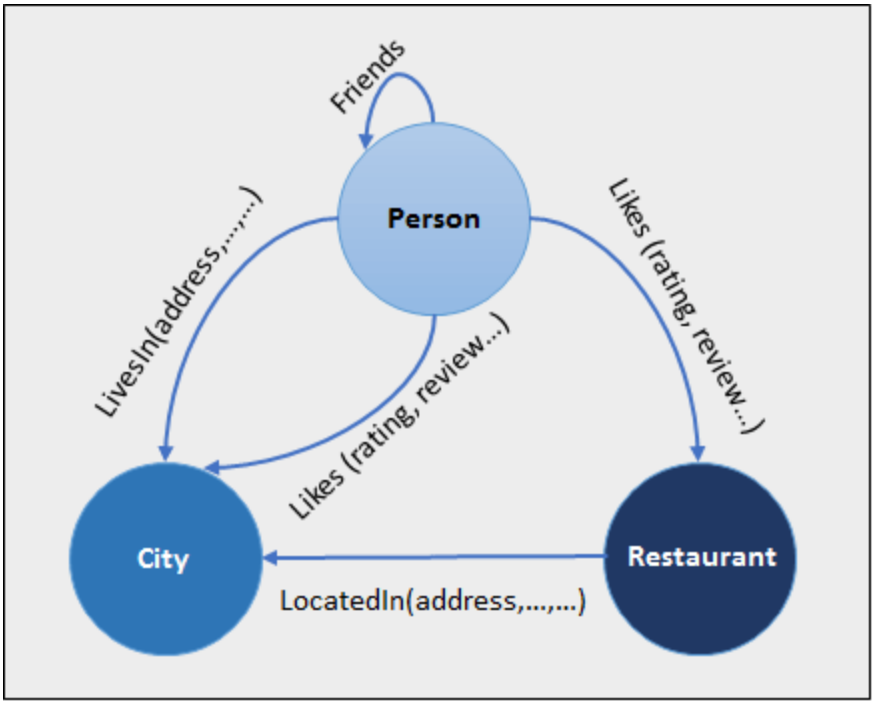
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.
Graph base database mostly used for social networks, logistics, spatial data.
Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Document-Oriented Databases:

What Is a Document Database, Document Database using MongoDB, MongoDB Data Types, JSON, JSON Syntax, Creating JSON Object, MongoDB Data Modelling, MongoDB CRUD Operations, MongoDB Collections: Creating CSV Files, Exploring dataset structures, Using MongoDB, Suitable Use Cases, and When Not to Use.

❖ What Is a Document Database?

- ➢ Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable.

❖ Document Database using MongoDB

- ➢ MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++.
- ➢ Collection: A collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

- Document: A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.
- Sample Document: The following example shows the document structure of a blog site, which is simply a comma-separated key-value pair.

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no SQL database',
   by: 'tutorials point',
   URL: 'http://www.tutorialspoint.com',
   tags: ['MongoDB, 'database', 'NoSQL'],
   likes: 100,
   comments: [
     {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2011,1,20,2,15),
        like 0
     },
     {
        user:'user2',
        message: 'My second comments',
        dateCreated: new Date(2011,1,25,7,45),
       like 5
     }
            ]
 }
```

- _id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, the next 3 bytes for the machine id, the next 2 bytes for the process id of the MongoDB server, and the remaining 3 bytes are simple incremental VALUE.

❖ MongoDB datatypes

- **String** − This is the most commonly used datatype to store the data. The string in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32-bit or 64-bit, depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or lists or multiple values into one key.
- **Timestamp** − timestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating an object of Date and passing a day, month, or year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.

- **Code** − This datatype is used to store JavaScript code in the document.
- **Regular expression** − This datatype is used to store regular expressions.

❖ JSON

- JSON stands for JavaScript Object Notation, JSON is a text format for storing and transporting data, JSON is "self-describing" and easy to understand, it is a lightweight data-interchange format, it is language independent.
- The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.
- Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.
- When storing data, the data has to be in a certain format, and regardless of where you choose to store it, the text is always one of the legal formats.
- JSON makes it possible to store JavaScript objects as text.

❖ JSON Syntax.

- JSON syntax is derived from JavaScript object notation syntax:
- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

❖ Creating a JSON Object

- To create an object we need to use opening and closing curly braces {} and then inside of that we'll put all of the key-value pairs that make up our object.
- Every single property inside the JSON is a key value pair. The key must be surrounded by double "" quotes followed by a colon: and then the value for that key.
- If we have multiple key-value pairs, we need commas, separating every single one of our key-value pairs, similar to how we would create an array in a normal programming language.
- MongoDB JSON document structure syntax.

```
{
        Field1: Value1,
        Field2: Value2,
        …..
        FieldN: ValueN
}
```

- In the above syntax, field1 to fieldN contains the field which was we have used in the JSON documents.
- Value1 to ValueN is the value of the JSON field.

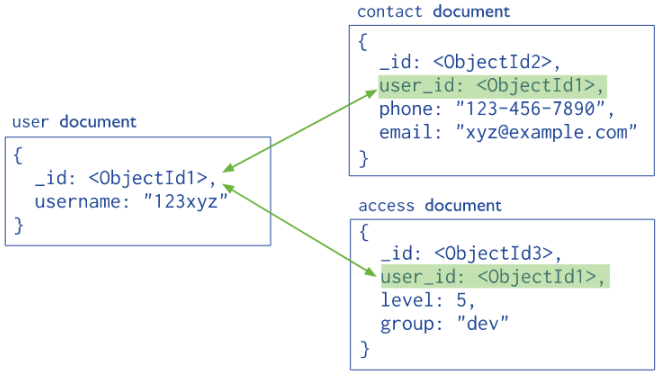
❖ MongoDB Data Modelling

- The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns.
- Flexible Schema
  - Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's collections, by default, do not require their documents to have the same schema. That is: the documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
  - To change the structure of the documents in a collection, such as adding new fields, removing existing fields, or changing the field values to a new type, update the documents to the new structure.

- Document Structure
  - The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. MongoDB allows related data to be embedded within a single document.
- Embedded Data
  - Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within

a document. These denormalized data models allow applications to retrieve and manipulate related data in a single database operation.

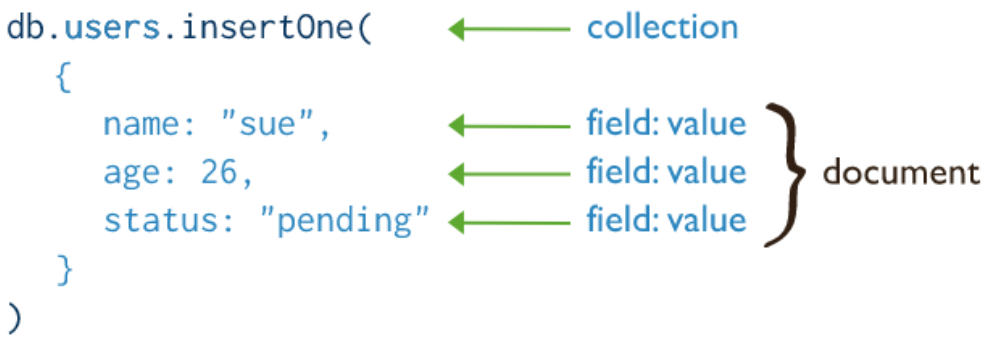
- ➢ References
  - References store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. Broadly, these are normalized data models.



- ➢ Single Document Atomicity
  - In MongoDB, a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents within a single document.
  - A denormalized data model with embedded data combines all related data in a single document instead of normalizing across multiple documents and collections. This data model facilitates atomic operations.
- ➢ Multi-Document Transactions
  - When a single write operation (e.g. db.collection.updateMany()) modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic.
  - When performing multi-document write operations, whether through a single write operation or multiple write operations, other operations may interleave.

❖ MongoDB CRUD Operations

- ➢ CRUD operations create, read, update, and delete documents.
- ➢ Create Operations
  - Create or insert operations and add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
  - MongoDB provides the following methods to insert documents into a collection:
    - db.collection.insertOne() New in version 3.2
    - db.collection.insertMany() New in version 3.2
  - In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.



➢ Read Operations

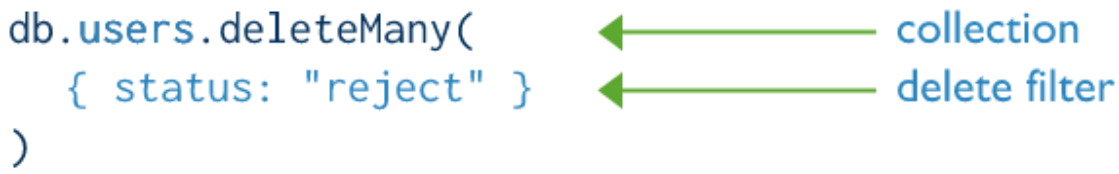
- Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:
  - db.collection.find()
- You can specify query filters or criteria that identify the documents to return.

➢ Update Operations

- Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:
  - db.collection.updateOne() New in version 3.2
  - db.collection.updateMany() New in version 3.2
  - db.collection.replaceOne() New in version 3.2
- In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
- You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

➢ Delete Operations

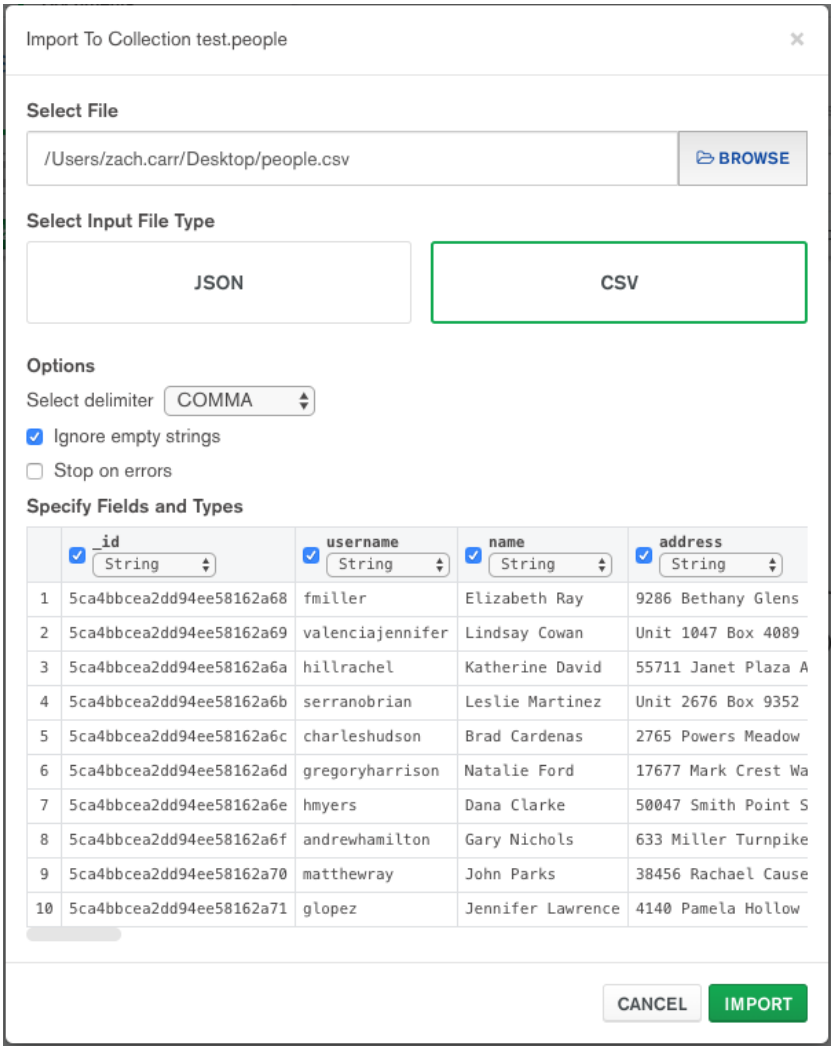
- Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:
  - db.collection.deleteOne() New in version 3.2
  - db.collection.deleteMany() New in version 3.2
- In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
- You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.



❖ Import and Export Data

- ➢ You can use MongoDB Compass to import and export data to and from collections. Compass supports import and export for both JSON and CSV files. To import or export data to or from a collection, navigate to the detailed collection view by either selecting the collection from the Databases tab or clicking the collection in the left-side navigation.
- ➢ Import Data into a Collection
  - ■ Format Your Data: Before you can import your data into MongoDB Compass you must first ensure that it is formatted correctly.
  - ■ When importing data from a CSV file, the first line of the file must be a comma-separated list of your document field names. Subsequent lines in the file must be comma-separated field values in the order corresponding with the field order in the first line.
  - ■ The following .csv file imports three documents:

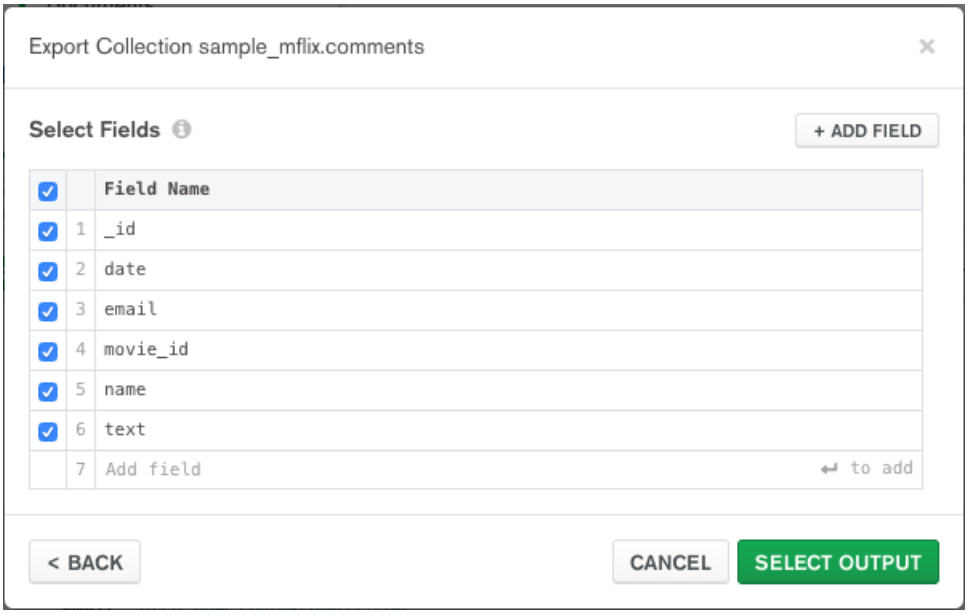
    ```
    name,age,fav_color,pet
    Jeff,25,green,Bongo
    Alice,20,purple,Hazel
    Tim,32,red,Lassie
    ```
  - ■ To import your formatted data into a collection:
    - Connect to the deployment containing the collection you wish to import data into.
    - Navigate to your target collection.
    - Click the Add Data dropdown and select Import File.
    - Select the location of the source data file under Select File.
    - Choose the appropriate file type
    - Under Select Input File Type, select either JSON or CSV.
    - If you are importing a CSV file, you may specify fields to import and the types of those fields under Specify Fields and Types. The default data type for all fields is a string.



- To exclude a field from a CSV file you are importing, uncheck the checkbox next to that field name. To select a type for a field, use the dropdown menu below that field name.
- Configure import options.
- Under Options, configure the import options for your use case.
  - If you are importing a CSV file, you may select how your data is delimited.
  - For both JSON and CSV file imports, you can toggle Ignore empty strings and Stop on errors:
  - If checked, Ignore empty strings drops fields with empty string values from your imported documents. The document is still imported with all other fields.
  - If checked, Stop on errors prevents any data from being imported in the event of an error. If unchecked, data is inserted until an error is encountered and successful inserts are not rolled back. The import operation will not continue after encountering an error in either case.
- Click Import.

➢ Export Data from a Collection

- MongoDB Compass can export data from a collection as either a JSON or CSV file. If you specify a filter or aggregation pipeline for your collection, Compass only exports documents that match the specified query or pipeline results.
  - To export an entire collection to a file:
  - Connect to the deployment containing the collection you wish to export data from.
  - Navigate to your desired collection.
  - Click Collection in the top-level menu and select Export Collection.
  - Select document fields to include in your exported file.



- Choose a file type and export location.
  - Under Select Export File Type, select either JSON or CSV. If you select JSON, your data is exported to the target file as a comma-separated array.
  - Then, under Output, choose where to export the file to.
  - Click Export.

❖ Suitable Usecases

- ➢ Event Logging
  - Applications have different event logging needs; within the enterprise, there are many different applications that want to log events. Document databases can store all these different types of events and can act as a central data store for event storage. This is especially true when the type of data being captured by the events keeps changing. Events can be sharded by the name of the application where the event originated or by the type of event such as order_processed or customer_logged.
- ➢ Content Management Systems, Blogging Platforms
  - Since document databases have no predefined schemas and usually understand JSON documents, they work well in content management systems or applications for publishing websites, and managing user comments, user registrations, profiles, and web-facing documents.
- ➢ Web Analytics or Real-Time Analytics
  - Document databases can store data for real-time analytics; since parts of the document can be updated, it's very easy to store page views or unique visitors, and new metrics can be easily added without schema changes.
- ➢ E-Commerce Applications
  - E-commerce applications often need to have a flexible schema for products and orders, as well as the ability to evolve their data models without expensive database refactoring or data migration

❖ When Not to Use

- ➢ Complex Transactions Spanning Different Operations
  - If you need to have atomic cross-document operations, then document databases may not be for you. However, there are some document databases that do support these kinds of operations, such as RavenDB.
- ➢ Queries against Varying Aggregate Structure
  - Flexible schema means that the database does not enforce any restrictions on the schema. Data is saved in the form of application entities. If you need to query these entities ad hoc, your queries will be changing (in RDBMS terms, this would mean that as you join criteria between tables, the tables to join keep changing). Since the data is saved as an aggregate, if the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity—basically, you need to normalize the data. In this scenario, document databases may not work.