



LESSON

MongoDB 101: Non-Relational for Beginners

Google slide deck available [here](#)

This work is licensed under the [Creative Commons
Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)
(CC BY-NC-SA 3.0)

Overview



Learning Objectives

At the end of this course, learners will be able to:

- Describe the differences between relational and non-relational databases.
- Identify the strengths of each of the four types of non-relational databases.
- Describe what a document is in terms of the document model.
- Identify the correct syntax and structure used in the document model.
- Define a collection in the document model.
- Define the concepts of linking and embedding data, and when one should be used over the other during schema design.
- Identify the types of data relationships between data (one to one, one to many, many to many).

Suggested Uses

- Lecture for one hour class
- Handouts / asynchronous learning
- Supplemental reading material - read on your own / not part of formal teaching
- Complement to University courses [Introduction to MongoDB](#).

At a Glance



Length:
60 minutes



Level:
Foundational



Prerequisites:
None

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)
(CC BY-NC-SA 3.0)

Share your feedback: We hope these curriculum materials will be a valuable resource for you and your learners. Let us know how the materials work for you, what we can improve on, and how MongoDB for Academia can support you via our brief [feedback form](#).

MongoDB for Academia: MongoDB for Academia offers resources for educators and students to support teaching and learning MongoDB. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).

Last Update: March 2025

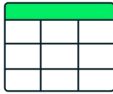


Overview of Non-Relational Databases



Where it Began: Relational

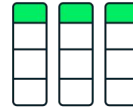
Key features of relational databases



Related data is stored in rows and columns in one table.



SQL (Structured Query Language)



A table uses columns to define the information being stored and rows for the actual data.

To understand non-relational databases, or “NoSQL” databases, we first need to look at SQL or relational databases.

Key features of relational databases:

- Modeled similarly to an excel spreadsheet with related data being stored in rows and columns in one table.
- SQL (Structured Query Language) is the most common way of interacting with relational database systems. Developers can write SQL queries to perform CRUD (Create, Read, Update, Delete) operations.
- A table uses columns to define the information being stored and rows for the actual data. Each table will have a column that must have unique values—known as the primary key. This column can then be used in other tables, if relationships are to be defined between them. When one table’s primary key is used in another table, this column in the second table is known as the foreign key.



Filling in the Gap



When traditional relational databases were introduced, they were able to handle the growth of the data size by running on bigger machines.

With the emergence of the web came a huge data explosion that made it difficult to scale with hardware. You could not scale the database by running it on a bigger server, so companies were left to horizontally scale by distributing data across multiple servers or by running on more powerful servers. However, these scaling options were often complex and costly to maintain.

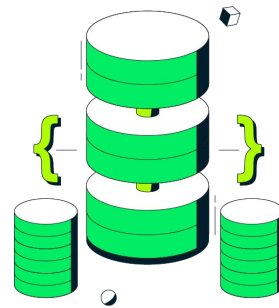
1. **Relational Databases:** To mitigate the cost of the first navigational databases and allow for searching, E.F Codd released his paper on a new way of storing data, relational. SQL was added to the field.
2. **World Wide Web:** The invention of the web fueled the demand for client-server database systems and high efficiency. Companies forced to scale use more servers at a high cost.
3. **NoSQL:** NoSQL (non-relational) databases were created to allow for faster processing of larger, more varied datasets. Emphasis on flexibility.



To fix the problem, various technology and software companies introduced new databases referred to as **NoSQL** or **non-relational**.

- Polymorphic data structures
- Flexible schemas
- Easy to scale large workloads

What is a non-relational database?



Non-relational databases differ from relational databases in that they do not store data in a tabular form.

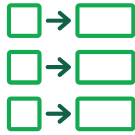
Instead, non-relational databases might be based on data structures like documents, graphs, or dictionaries. NoSQL databases also come in a variety of types based on their data model.

They provide flexible schemas and scale easily with large amounts of data and high user loads. They were designed when it was expected that data would be partitioned across multiple machines to scale, in contrast to relational databases which assumed the data would stay on a single machine.



Non-Relational Database Types

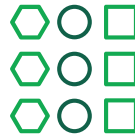
Non-Relational Database Types



Key/Value



Graph



Column



Document

There are four main types of non-relational databases: key/value, graph, column, and document, and we'll investigate each in this lesson.

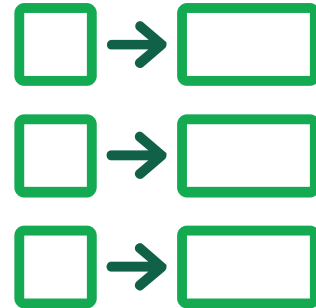


Structure

- A unique key is paired with a collection of values, where the values can be anything from a string to a large binary object

Strength

- Simple data model



Key/Value Database

Key-value databases use a very simple schema: a unique key is paired with a collection of values, where the values can be anything from a string to a large binary object.

One way that databases using this structure gain in performance is that there are no complex queries. The system knows on which server the data is located and sends the request to just that server.

Example: Redis is one of the most popular examples of a key/value store database. .

Key/Value: Example



Key	Value
Name	Sherlock Holmes
Age	40
Address	221B Baker Street

As the simplest of the non-relational databases, key/value is exactly as it sounds, data is organized based on an identifying key and its corresponding value. This simplicity makes it beneficial for large datasets, but not when complex relationships are at play.

Structure

- Captures connected data
- Each element is stored as a node
- Connections between nodes are called links or relationships

Strength

- Traverses the connections between data rapidly



Graph Database

Graph databases are another type within the non-relational family of databases.

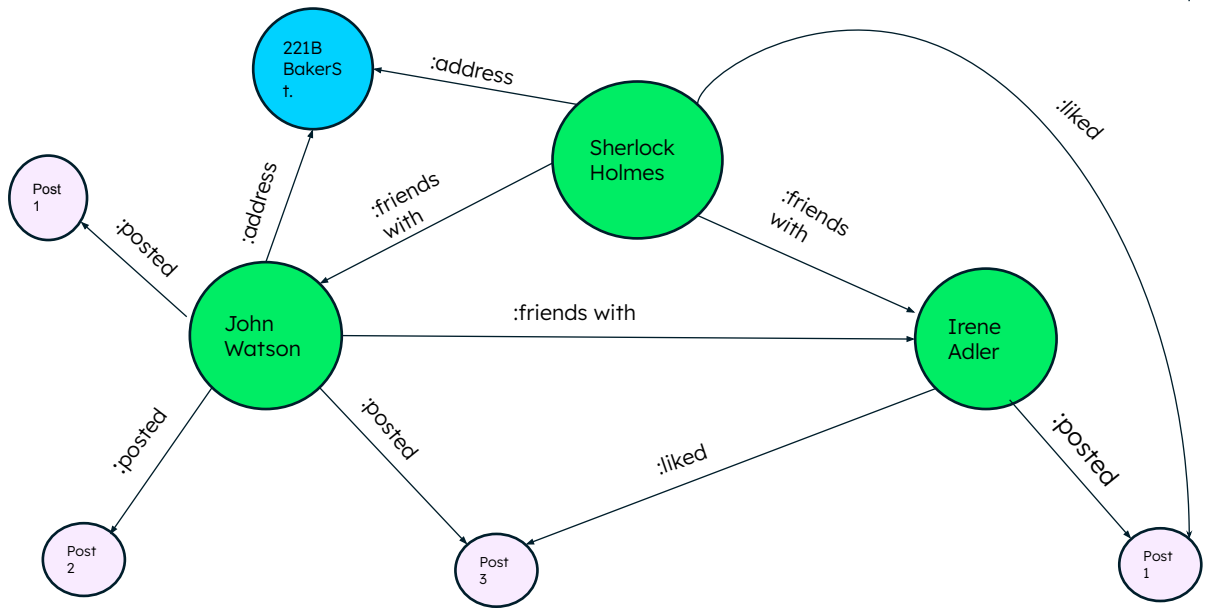
They have been designed to deal with problems around relationships with and focuses on connected data. This type of model does capture the richness of the relationships, however many problems are not naturally modelled as connected data or relationship problems. This makes this database well matched to these problems but as these are niche, it does not have a wider or broader applicability to other problems.

This databases stores information as a collection of nodes and edges, where the edges represent the relationships between the nodes.

Storing the relationships between data means that related data can often be retrieved in a single operation. The concept of relationships between data and this interconnectedness is the key principle behind this type of non-relational database. This approach counters the approach required in SQL/Relational Databases where many joins on several attributes in a number of tables would often be required to retrieve these kinds of relations in the data.

The most popular example of a graph database is Neo4J.

Graph: Example



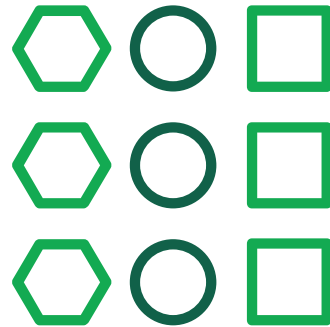
This is a rudimentary example of how data could be stored in a graph database. As you will see, this type of store is useful for social media applications where multiple objects will have multiple relationships or links.

Structure

- Data is stored using key rows that can be associated with one or more dynamic columns

Strengths

- Highly performant queries
- Designed for analytics



Column Oriented
or Wide Column

A column oriented or wide column non-relational database is primarily designed for analytics. Today, Cassandra is a commonly used column oriented database. The advantage of column versus record/row oriented databases is that column oriented databases return data in columns making the query much more performant as it will not return many irrelevant fields that are not required for the query being serviced. The primary key in a column oriented database is the data / value which is then mapped to row keys. This is the inverse, or opposite, of how the primary key works in a relational database.

The structure of the column data is flexible and can vary from row to row. Associated with storing large amounts of data: billions of rows with millions of columns

This does not necessarily require a separate database and can be implemented as indexes on existing data structures to add this type of functionality to a database.



Column Oriented Example

Name	ID
Sherlock	001
John	002
Irene	003

Age	ID
40	001
45	002
43	003

Height	ID
6'2	001
5'9	002
5'7	003

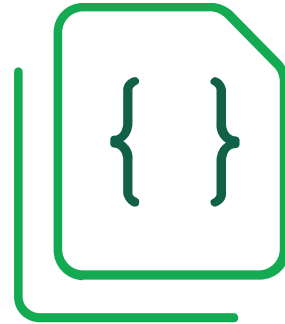
Here we see an example of what data might look like in a column-oriented database. Though at first glance it might seem similar to the known relational tabular format, it is very different in that the data is sorted via column IDs. Therefore the relationships between data are identified via the column key.

Structure

- Polymorphic data models
- Each document contains markup that identifies fields and values

Strengths

- Obvious relationships using embedded arrays and documents
- No complex mapping



Document Database

Document databases, such as MongoDB, store data in a single document which can have different shapes within the single collection or table that stores the documents.

It provides a clear means of capturing relationship using sub-documents and embedded arrays within a single document.

The document is a close analogy to the object in object oriented programming and provides a clear natural representation of a 'thing' and it's data.

This clear representation often means that there is no requirement for object mapping between the database and the application/programming language. The document is often the exact match for the object that the programmer wishes to use. The flexibility of the document to hold many shapes or multiple parallel schemas at any point in time gives great flexibility for modeling with documents when compared to relational database tables.

Document Model Example



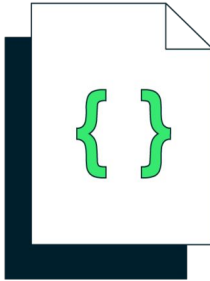
```
{
  "_id":
  ObjectId("5ef2d4b45b7f11b6d7a"
),
  "user_id": "Sherlock
Holmes",
  "age": 40,
  "address":
    {
      "Country": "England"
      "City": "London",
      "Street": "221B Baker
St."
    },
  "Hobbies":[ violin,
crime-solving ]
}
```

```
{
  "_id":
  ObjectId("6ef8d4b32c9f12b6d4a")
,
  "user_id": "John Watson",
  "age": 45,
  "address":
    {
      "Country": "England"
      "City": "London",
      "Street": "221B Baker
St."
    },
  "Medical license": "Active"
}
```

The key thing to understand about the document model is that data that is accessed together is stored together. It is also important to note that just because one document has one field does not mean another related document has to have the same field when stored together. We will discuss this more in the lesson when we talk about collections.



The Document Model



For **general purpose** use, the document model prevails as the preferred model by developers and database administrators.

Due to the fact that the document model implements data structures using programming languages, it is the preferred model by developers as it most closely matches the way they think and work already.

We will take a closer look at the document model and understand how it functions.

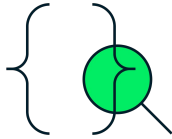


The Document Model and MongoDB

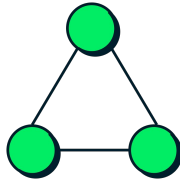
For more information watch: MongoDB in Five Minutes
<https://www.youtube.com/watch?v=EE8ZTQxa0AM>



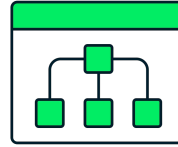
Key Features



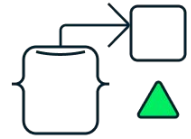
**API query or
query language**



**Distributed and
resilient**



**Flexible
schema**



**Object
mapping**

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.”

Distributed and resilient: Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) which distributes the data across multiple machines rather than making one machine bigger as the data increases. This system also allows for data to achieve high availability and resiliency as the data lives in replica sets which creates redundancy, so if one machine fails the secondary machine will take over and keep the data alive. This system is also referred to as sharding.

Object mapping: Documents easily map to objects, the most frequently used data structure in the most popular programming languages. This allows developers to rapidly develop their applications as it is an intuitive process.

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 both mandatory or defined.



```
{
  "_id": ObjectId(
    "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Sean",
  "age": 29,
  "Status": "A"
}
```

The Document Model: Structure and Syntax

To the left is an example of a document representing a user details including user_id, age, and a status category.

In order to better understand a document, let's take an example in MongoDB. This document represents a user, their id in the system, their age, and their status. You can also note the “_id” field which holds the ObjectId for the document. The _id is used as a primary key; its value must be unique in the collection, it is immutable, and may be of any type other than an array. In this example, it uses a number but typically these will be automatically generated ObjectIDs. An ObjectId is a small, likely unique, fast to generate, and ordered 12 byte value. An ObjectId's 12 bytes consist of a 4-byte timestamp value, representing the ObjectId creation time, measured in seconds since the Unix epoch, a 5-byte random value, and a 3-byte incrementing counter, initialized to a random value.



```
{
  "_id": ObjectId(
    "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Sean",
  "age": 29,
  "Status": "A"
}
```

The Document Model: Structure and Syntax

A document in MongoDB uses the JavaScript Object Notation (JSON) format.

This format uses **curly brackets** to mark the start and the end of the document.

In order to better understand a document, let's take an example in MongoDB. This document represents a user, their id in the system, their age, and their status. You can also note the “_id” field which holds the ObjectId for the document. The _id is used as a primary key; its value must be unique in the collection, it is immutable, and may be of any type other than an array. In this example, it uses a number but typically these will be automatically generated ObjectIds. An ObjectId is a small, likely unique, fast to generate, and ordered 12 byte value. An ObjectId 12 bytes consist of a 4-byte timestamp value, representing the ObjectId creation time, measured in seconds since the Unix epoch, a 5-byte random value, and a 3-byte incrementing counter, initialized to a random value.



```
{
  "_id": ObjectId(
    "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Sean",
  "age": 29,
  "Status": "A"
}
```

The Document Model: Structure and Syntax

MongoDB refers to keys
as **fields**.

The **field-values** within a
pair in a document are
separated by **colons** (:).

In order to better understand a document, let's take an example in MongoDB. This document represents a user, their id in the system, their age, and their status. You can also note the “_id” field which holds the ObjectId for the document. The _id is used as a primary key; its value must be unique in the collection, it is immutable, and may be of any type other than an array. In this example, it uses a number but typically these will be automatically generated ObjectIDs. An ObjectId is a small, likely unique, fast to generate, and ordered 12 byte value. An ObjectId 12 bytes consist of a 4-byte timestamp value, representing the ObjectId creation time, measured in seconds since the Unix epoch, a 5-byte random value, and a 3-byte incrementing counter, initialized to a random value.



The Document Model: Structure and Syntax

Each **field** must be enclosed within **quotation marks**. String values are often quoted as good practice.

```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a"),  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

Each field in a MongoDB document must be enclosed within quotation marks. String values are often quoted as good practice.



```
{  
  "_id": ObjectId(  
    "5f4f7fef2d4b45b7f11b6d7a"),  
  "user_id": "Sean",  
  "age": 29,  
  "Status": "A"  
}
```

The Document Model: Structure and Syntax

Each **field-value pair** is separated within the document by **commas**.

Each of the field-value pairs in the document are separated by a comma from the next record. The final field-value pair doesn't require a comma as the final curly brace indicates the end of the document.

The background of the slide is a dark green color. On the right side, there is a lighter green abstract shape that resembles a stylized leaf or a drop. In the top right corner of this lighter green shape, there is a small, dark green leaf icon.

Collections in the Document Model

Document



A way to organize and store data as a set of field-value pairs in MongoDB.



Collection



An organized store of documents in MongoDB, usually with common fields between documents

Another way data is stored in MongoDB's document model is through collections. A collection is a group of documents. Collections typically store documents that have similar contents. In MongoDB, these usually have common fields between documents but this is not a requirement unless you are using schema validation to enforce specific common fields.

A document is a way to organize and store data as a set of field-value pairs in MongoDB.

A collection is an organized store of documents in MongoDB, these usually have common fields between documents but this is not a requirement unless you are using schema validation to enforce specific common fields.



Example

Two documents in the same collection but with different fields

```
{
  "_id": ObjectId(
    "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Sean",
  "age": 29,
  "Status": "A"
}
```

```
{
  "_id": ObjectId(
    "5f4f7fef2d4b45b7f11b6d7a"),
  "user_id": "Daniel",
  "age": 25,
  "Status": "A",
  "Country": "USA"
}
```

MongoDB collections do not by default enforce a single schema on a collection so whilst documents can have common fields, they are not required to have the same fields. There is no issue having two documents where the first document has a “country” field and the second document does not have the “country” field.

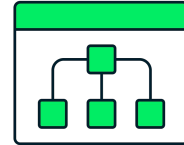


Collections and Schema Validation

The document model used by MongoDB can enforce a schema if required, the recommended approach is to do so using JSON Schema.

JSON Schema

- Allows a prescribed document structure to be configured on a per collection basis.
- Can tune schema validation according to use case.
- Can be used by any query to inspect document structure and content.



MongoDB can enforce a schema if required, the recommended approach is to do so using JSON Schema. This allows a prescribed document structure to be configured on a per collection basis.

Document validation allows restrictions to be made when new content is added, it allows for the presence, the type, and the values to be validated as part of this process as well.

Schema validation in MongoDB has tunable controls. Administrators have the flexibility to tune schema validation according to use case – for example, if a document fails to comply with the defined structure, it can either be rejected, or still written to the collection while logging a warning message. Structure can be imposed on just a subset of fields – for example, requiring a valid customer name and address, while others fields can be freeform, such as the social media handle and cell phone number. And, validation can be turned off entirely, allowing complete schema flexibility

The schema definition can be used by any query to inspect document structure and content. For example, DBAs can identify all documents that do not conform to a prescribed schema.

This avoids having to implement this validation logic in your application or in

middleware.

Data modeling is critical to setting up any database to meet the needs of an application, but in a document based non-relational database, such as MongoDB, there is great flexibility on how to model the data. How do you know which way to store your data?

We will cover some best practices when it comes to modeling data in MongoDB.



Data Modeling and MongoDB



Schema Design



The design comes from the needs of the application first. Therefore, the schema should evolve as the application changes.

At the core of all database models are their schemas. Schema design refers to the organization of data into separate entities, determining how to create relationships between the organized entities, and how to apply constraints on the data. Designers create database schemas to give other database users, such as programmers and analysts, a logical understanding of the data.

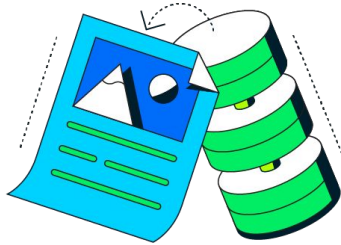
Schema design is defined at the application level and it is likely it will change over the application's lifetime.

The design comes from the needs of the application first. Therefore, the schema should evolve as the application changes. One of the advantages of using a document model database is that the schema is able to be flexible, meaning if a developer or database designer needs to make changes to it they can do so with minimal to no downtime.



Data Modeling and the Document Model

The core of data modeling in the document model is to understand what data is needed by your queries. Once that information is known, can you begin designing the schema.

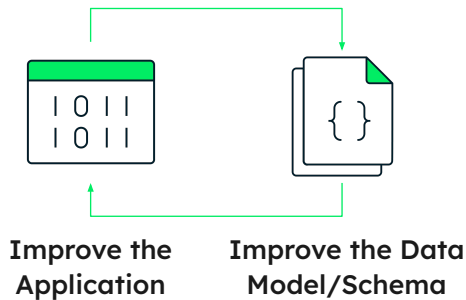


When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

The data model in MongoDB is:

- Document-oriented: MongoDB stores data as documents that tend to have all data for a given record in a single document.
- Flexible: You can store and combine any type of data while benefiting from sophisticated data access and rich indexing features.
- Dynamic: Having a predefined schema or structure for your data is not necessary with MongoDB. You can create documents without first defining the data structure (e.g., fields, types of their values) and you can easily change the structure of documents by adding new fields or deleting existing ones.

Data Modeling with MongoDB



Several design possibilities

Design for the usage pattern

Evolving the schema is easy

No migrations or downtime required for a new version of the schema

MongoDB supports the iterative and rapid development of an application.

This opens out many design possibilities or options. MongoDB is designed for the application's usage pattern.

Evolving a schema in MongoDB is easy, you just add or remove the field(s) and continue. This evolution doesn't require a migration or a downtime where the database is taken offline while the new schema is created or updated.



Schema Design: Considerations

Your queries and the specific data your application requires.

How your application reads the data (**read** patterns).

How your application writes the data (**write** patterns).

What are the relationships between your data (**linked** or **embedded**).

There are a number of considerations you should focus on designing your schema in the document model.

Schema Design - Link or Embed?



Embedded vs Linked relationship in the Post-Comment example

Embedded



Linked



The question of whether to link or to embed your data is a critical factor to modeling in a document-oriented database.

Embedding:

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.

Linking:

Linking stores 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.

In general, the rule of thumb is to favour embedding over linking. This typically allows for the required data to be retrieved with a single query. It is also a better design when data that will be deleted exists together.

This all depends on the situation and if there is a large amount of unused data in the document or say only the last 20 comments are required then it might be good to

consider an alternative design.



Schema Design - Link or Embed? Cont.

Do I want most of the data's information embedded?

Do I need to search within the embedded data?

How frequently will the embedded data change?

Is the embedded data shared or private?

The questions here should all be answered when you are creating your schema design as they can help ensure you fully consider all aspects of the design.

Firstly, Do I want most of the data's information embedded? In this case, if all the information is present in a single document then retrieval will be faster than if it is in two documents or across two or more collections. The frequency of the specific data and the size of the data must also be considered as very large documents will not be retrieved as fast as tiny documents even if those documents are in two separate collections.

A second important question to ask is "Do I need to search within the embedded data?" You might also want to follow up to determine what you are searching for and if it is only a fraction of the embedded data, should only some of it be embedded and the rest linked.

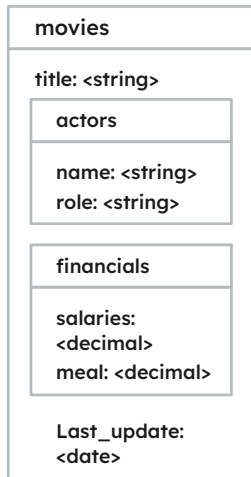
Finally, you should determine whether the embedded data can be shared or whether it is private. Indeed, this should be double checked with all stakeholders to ensure there are no doubts on this question which is increasingly important in the age of both data and digital privacy.

An important set of questions that should be answered when designing your schema is whether you should link to other documents with the data or whether the data should be embedded in a single document. This should also be considered in terms of the different users of the database as this might not be the same answer depending on the specific user.

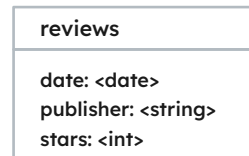
Example: Movies and Reviews



Embedded



Linked



Let's look at an application where we store details on movies and reviews related to those movies.

In this example, we have a movie document where the actors and the financials are embedded in a single document whilst reviews are linked. You can use both embedding and linking in your schema. Here is an example where both are used. In terms of movies, the number of actors with their details and the financial details of the movie will be limited and typically bound to relative small size. In contrast, reviews are definitely unbounded data where we don't know how many may be written for the movie so it makes sense to only link to this data. An aggregated stars field for all reviews might be embedded in the movie document but not the 'raw' reviews for again similar reasons.



Relationships



One to one (1-1)

One to many (1-N)

Many to many (N-N)

Relationships and Data Modeling

It is a common misconception that just because a database is labeled “non-relational” then the data within it has no defined relationships. In fact, the relationships between data is a key component to designing the schema of a non-relational database and deciding whether to link or embed the data.



Relationships

One-to-One (1-1)

A one-to-one relationship is represented and stored in a single document, this would typically be data like a person's name and the customer id.

customers
<code>name</code> <code>customer_id</code>

In designing this relationship, if you only consider one side of the relationship you may classify it as a “one” or as a “many” if you don’t consider both sides. The best advice is to ensure you ask the question of associativity from both directions and that you review your model a few times, especially for less apparent relationships.

All these fields have a one-to-one relationship with each other. More clearly, a user in our system has one and only one name, and is associated with one and only one customer id.

Embedding is the preferred way to model a 1:1 relationship as it's more efficient to retrieve the document.



One to One (1-1)

Scenario:

You have to map patron and address relationships. In this example, you'll need to view one data entity in context of the other.

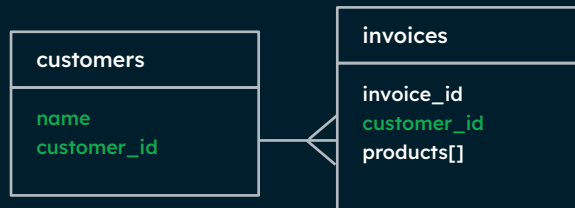
In this one-to-one relationship between [patron] and [address] data, the [address] belongs to the [patron]. If the [address] data is frequently retrieved with the [name] data, then with referencing (or linking), your application would need to issue multiple queries. The better data model would be to embed the [address] data in the [patron] data.



Relationships

One-to-Many (1-N)

A one-to-many relationship can be considered when an object of a given type is associated with N objects of a second type.



A one-to-many relationship can be considered as that when an object of a given type is associated with N objects of a second type.

In this type of relationship, the customer can have many invoices. It can be modelled by either linking the data (as shown) or by embedding the data (where the invoices are within the customer documents).

There is an additional technical called bucketing which is a combination of both linking and embedding. Bucketing works best when you can split your documents into batches, it can speed up document retrieval. Time based data (IOT) or where there are a number of entries (e.g. like comments pagination).



One to Many (1-N)

Scenario (Link):

You have to map publisher and book relationships. Suppose you had the same publisher data for the same book. Embedding the [publisher] document inside the [book] document would lead to repetition of publisher information.

Instead of embedding in this scenario, we can use references (or links) to keep the publisher information in a separate collection entirely from the book collection. This avoids the issue of repetition.



One to Many (1-N)

Scenario (Embed):

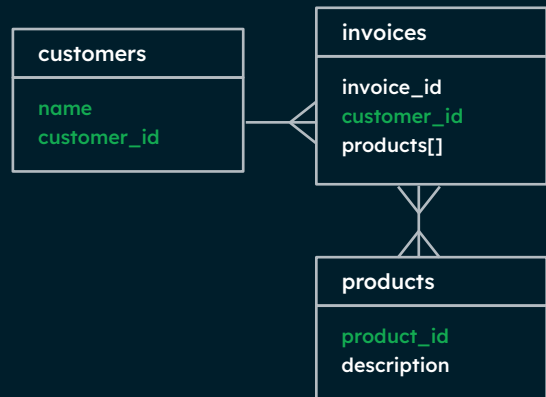
You have to map a patron with multiple address relationships. In this one-to-many relationship between [patron] and [address] data, the [patron] has multiple [address] entities.

If the [address] data is frequently retrieved with the [name] data, then with referencing, your application would need to issue multiple queries. A more optimal schema would be to embed the [address] data entities in the [patron] data. If the [address] data is frequently retrieved with the [na

Relationships

Many-to-Many (N-N)

A Many-to-Many relationship between two entities where they both might have many relationships between each other.



Documents on the first side can be associated with many documents on the second side.

In terms of documents on the second side, it equally means that these documents can be associated with many documents on the first side.

To reiterate an important modelling point about designing relationships, if you only consider one side of the relationship you may classify it as a “one” or as a “many” if you don’t consider both sides. The best advice is to ensure you ask the question of associativity from both directions and that you review your model a few times, especially for less apparent relationships.

There are several strategies but the 1-way embedding strategy optimises the read performance of a N:N relationship by embedding the references in one side of the relationship. The key step in this strategy is establishing the relationship balance and choosing the side which has one or two orders magnitude of difference in the number of entities. If the relationships are close to an even ratio then 2-way embedding is probably a better strategy.



Many to Many (N-N)

Scenario:

Consider a scenario where a book was written by multiple authors and similarly, one of the authors has written multiple books. How would we go about mapping these relationships?

When it comes to a true “many to many” relationship, such as this one, there are several ways this can be mapped. One way that would trick the system (so to speak) is to make the relationship into two “one to many” relationships. In this scenario, that would mean having a document for the book and its authors and a separate document for the author and its books, with each document having the other one embedded. However, in MongoDB you don’t have to do this option. The best option would be to embed an array of subdocuments on both many sides. That way the book document has all the authors and their subdocuments as well as the author document having all the books in a subdocument their other authors’ information.

Embed

For integrity with read operations

For integrity with write operations

On one-to-one and one-to-many

For data that is deleted together
by default

Link

When the "many" side is a huge
number

For integrity on write operations on
many-to-many

When a piece is frequently used,
but not the other and memory is
an issue

Going back to our discussion on linking and embedding, the relationship between the documents is a factor in your modeling decisions.

In order to help in the choice of relationship when you are modeling there are several helpful rules of thumb:

For embedding data, this should be favoured for read operations to support integrity as well as for write operations on one-to-one and one-to-many relationships. Embedding is also recommended when data is going to be deleted together. In the majority of modeling designs, embedding data should be the default approach or choice taken.

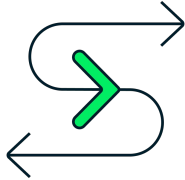
In terms of linking data, it is recommended that this is done where there is a very large number of objects on the "many" side in either a one-to-many or a many-to-many relationship.

It is also recommended for write operations on many-to-many relationships. Finally, linking is recommended where only a subset of the data is frequently used whilst the rest of the data is not and where memory may be an issue. We'll flag the subset pattern which relates to this concept and which you can look up on the MongoDB documentation.

Finally, the most important factor to keep to the forefront when you are modelling is the frequency of all the queries. This will help you make a better and more informed decision as to how you should model any specific data or relationship.

Wrap-Up

When to Use Non-Relational?



Working with data
that changes
frequently



Cloud computing



Promoting developer
productivity

A very common question when you are considering non-relational databases is when is it appropriate to use them?

Non-relational, particularly the document model, is well suited to polymorphic data that can change frequently. The document model allows for different shapes of data within the same collection, this means that documents with different fields can be present. It is sometimes described as holding multiple schema for the collection.

This is a key feature for enabling developer productivity as it provides rapid iteration of schema versions for data to co-exist within the same database and collection. This means that developers can rapidly change what fields are in a document and not worry about the impact or side effects that occur in the database. Essentially, the database is not a hurdle or an additional burden of work that must be additionally updated for example when a new field is added.

Another aspect of non-relational databases is that they often offer exact (or close to exact) mappings to what objects the developer desire to use in their application code. This means that the data can be transferred as-is directly to the application without requiring any additional mapping. This also enables greater developer productivity as there is less code to translate between the database and the application.

Non-relational systems are typically cloud-native and designed as distributed systems. A known pain point for relational databases stems from the initial focus on scaling vertically, where additional resources were added to the machine or a large machine as used to support scaling the database.

Non-relational systems took this pain point and deliberately focused on scaling horizontally, where additional machines were added to the existing machine(s) when scaling the database. This scaling approach has simplified any changes required for these databases to support multiple public cloud providers. It has also focused non-relational systems, firstly on being virtualisation friendly, and more recently on being container friendly, as two key provisioning technologies that assist in deploying the database.

Continue Learning!



[MongoDB University](#) has free self-paced courses and labs ranging from beginner to advanced levels.

GitHub Student Developer Pack



Sign up for the [MongoDB Student Pack](#) to receive \$50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out [MongoDB's University](#) page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).