COMP 1842

Introduction to NoSQL

Matt Prichard

Introduction

SQL vs NoSQL

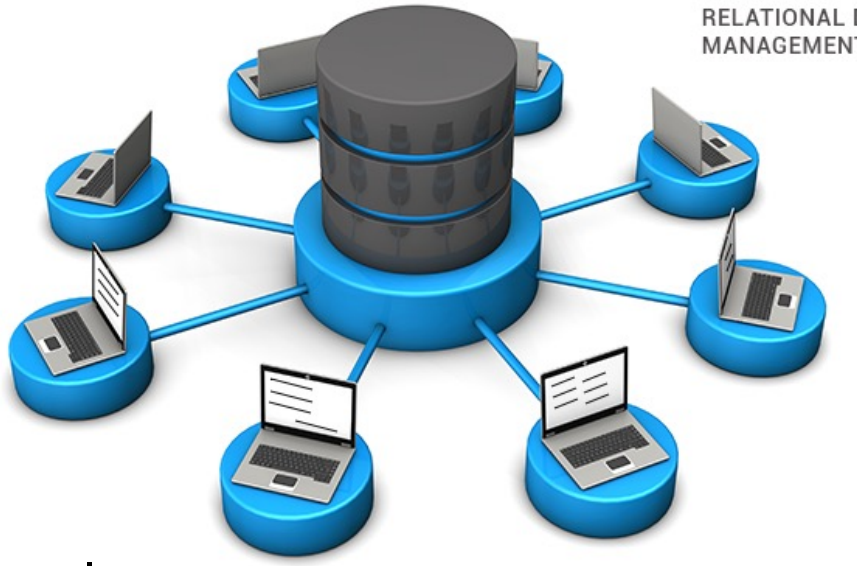
Scaling

MongoDB

SQL recap

- It is a query language that allows retrieving specific data from databases.
- A relational database is a type of database (usually organized into tables) that enables the recognition and access of data in relation to another piece of data within the same database.
- In other words, it stores related data across multiple tables, which are organized into columns and rows, and allow the user to query data (or information) from various tables simultaneously.

RDBMS



- A relational database is a database that follows the relational model of data. To maintain a relational database, a Relational Database Management System (RDBMS) is used. SQL is a language that allows for communication with data in an RDBMS.
- An important clarification is that SQL is not a database system itself. When comparing SQL vs NoSQL, the main differences being assessed are relational databases vs non-relational databases

SQL

- Another aspect to consider is that SQL is not the only programming language able to query relational databases, but it is definitely the most popular one.
- Therefore, the terms "SQL databases" and "relational databases" are often interchangeably used. MySQL, PostgreSQL, Microsoft SQL Server and Oracle Database are among the most well-known RDBMS using SQL.

What is NoSQL ?

- NoSQL refers to non-relational databases and to distributed databases.

  NoSQL can also stand for "Not Only SQL" to highlight that some NoSQL systems may also support SQL query language.

- In fact, before moving on it is important to keep in mind that NoSQL does not necessarily mean that a database does not support SQL.

- Instead, it means that the database is not an RDBMS or in other words a non-relational database.

Brief history

- NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased, removing the need to create a complex, difficult-to-manage data models in order to avoid data duplication.
- Cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. They wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up.

Scaling

What is scaling

- If an application can handle a maximum of 'x' requests simultaneously. As soon as this number exceeds x (say, x+1), critical hardware resources are exhausted, and the application cannot process further requests. Scaling refers to the adjustments made to system hardware resources to handle increased traffic and requests. This could be in the form of:
- Adjustments to network bandwidth
- Upgrades to CPU capacity and physical memory
- Basic hard drive alterations

Vertical scaling

- It is defined as the process of increasing the capacity of a single machine by adding more resources such as memory, storage, etc. to increase the throughput of the system. No new resource is added, rather the capability of the existing resources is made more efficient. This is called Vertical scaling. Vertical Scaling is also called the Scale-up approach.
- Example: MySQL

Advantages of Vertical Scaling

- It is easy to implement
- Reduced software costs as no new resources are added
- Fewer efforts required to maintain this single system

Disadvantages of Vertical Scaling

- Single-point failure
- Since when the system (server) fails, the downtime is high because we only have a single server
- High risk of hardware failures

Horizontal scaling

- The process of adding more instances of the same type to the existing pool of resources and not increasing the capacity of existing resources like in vertical scaling. Horizontal Scaling is also called the Scale-out approach.
- In this process, the number of servers is increased and not the individual capacity of the server. A Load Balancer routes the user requests to different servers according to the availability of the server. Thereby, increasing the overall performance of the system. In this way, the entire process is **distributed** among all servers rather than just depending on a single server.

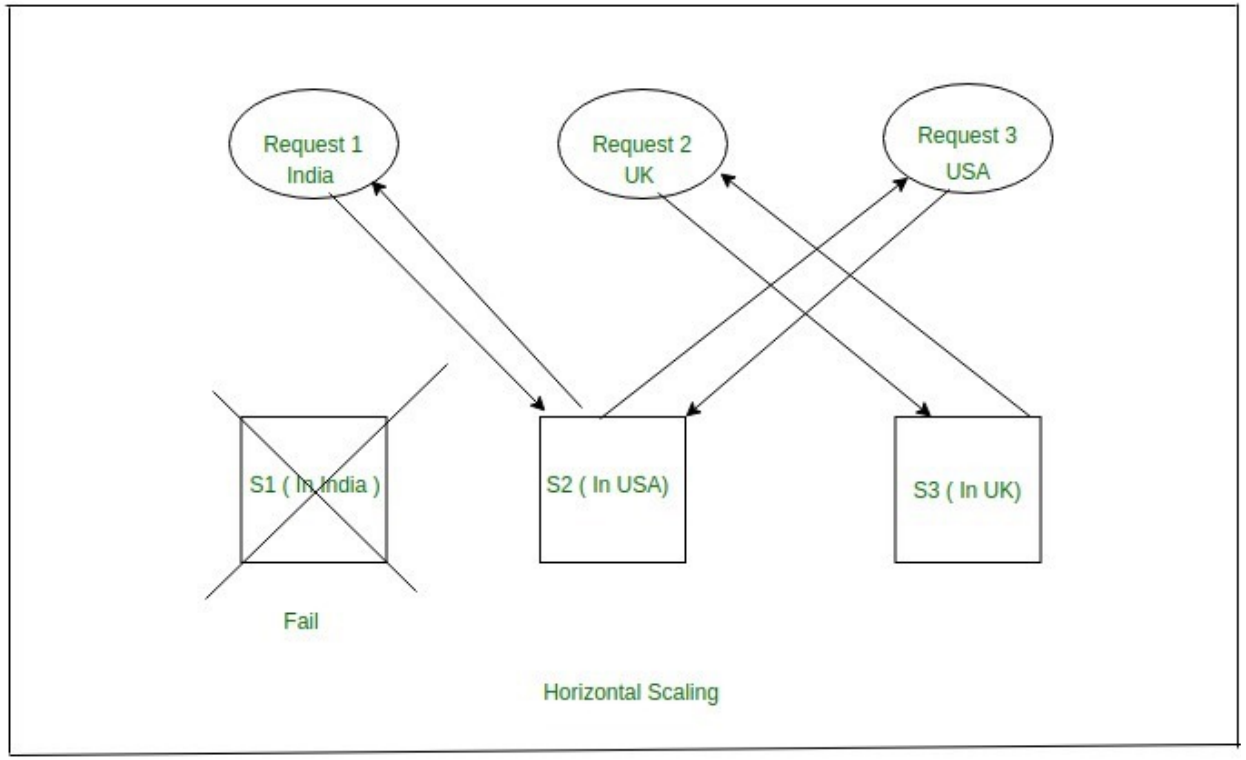
  Example: NoSQL, Cassandra, and MongoDB

Advantages of Horizontal Scaling

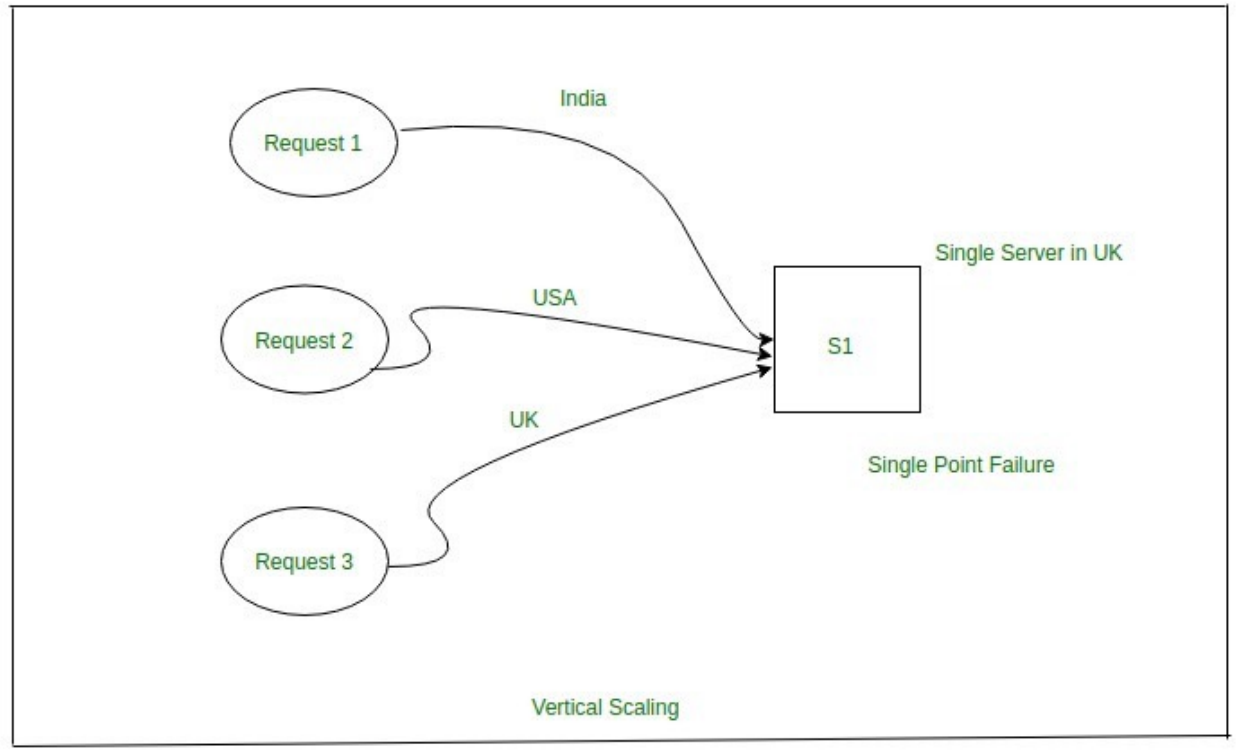
- There is no single point of failure in this kind of scale because there are multiple servers. So if any one fails then there will be other servers for backup.
- Low Latency: Latency refers to how late or delayed our request is being processed.
- Built-in backup

Disadvantages of Horizontal Scaling

- Not easy to implement as there are a number of components needed
- Cost is high
- Networking components like, routers and load balancer are required



Horizontal Scaling



Vertical Scaling

Back to NoSQL

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

- Flexible schemas – next slide
- Horizontal scaling
- Fast queries due to the data model – next slide but one
- Ease of use for developers

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 add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

Fast Queries

Queries in NoSQL databases can be faster than SQL databases. Why?

- Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries.
- The rule of thumb when you use MongoDB is **Data that is accessed together should be stored together.** Queries typically do not require joins, so the queries are very fast.

Types of NoSQL databases

Over time, four major types of NoSQL databases emerged:

1. Document databases - store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
2. Key-value databases - are a simpler type of database where each item contains keys and values.
3. Wide-column stores - store data in tables, rows, and dynamic columns.
4. Graph databases - store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

RDBMS vs NoSQL: Data Modelling Example

- Let's consider an example of storing information about a user and their hobbies. We need to store a user's first name, last name, phone number, city, and hobbies.
- In a relational database, we'd likely create two tables: one for Users and one for Hobbies.

Users

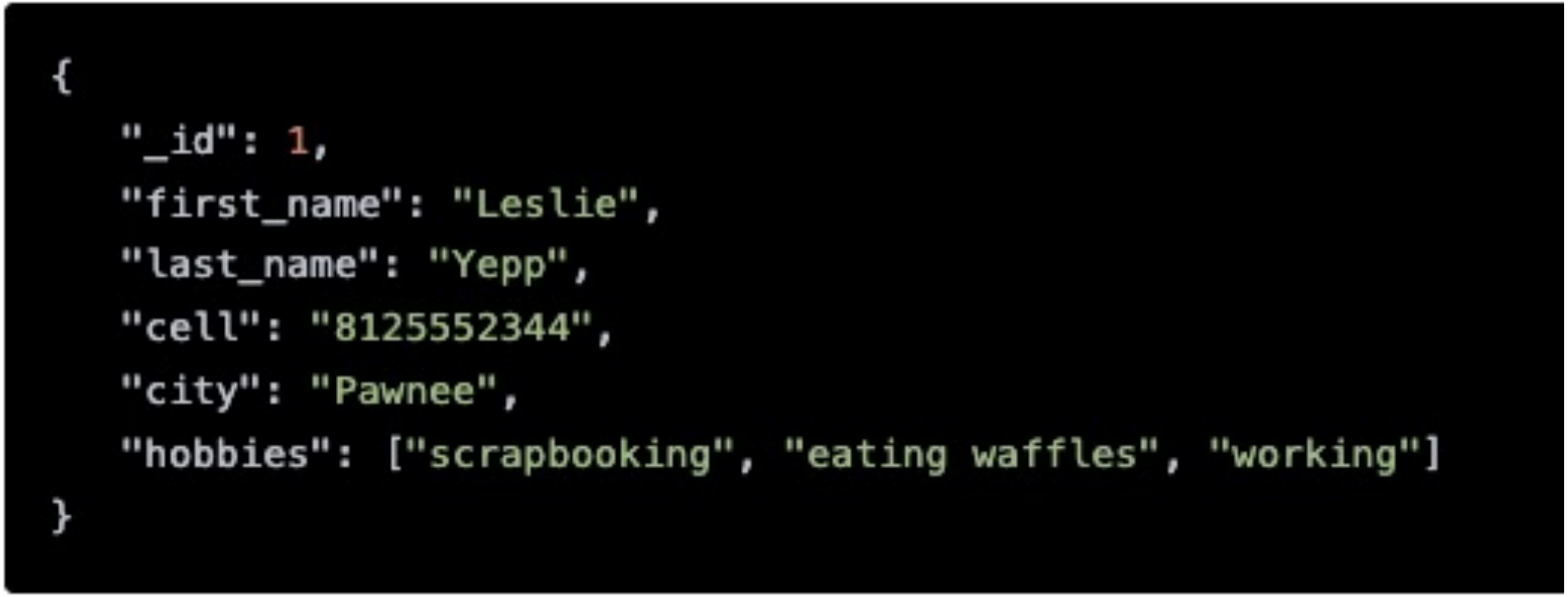
ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies

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

In order to retrieve all of the information about a user and their hobbies, information from the Users table and Hobbies table will need to be **joined** together.

How to store the same information about a user and their hobbies in a document database like MongoDB. – look familiar ?



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

In order to retrieve all of the information about a user and their hobbies, a single document can be retrieved from the database. No joins are required, resulting in faster queries.

When should NoSQL be used?

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

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

https://www.mongodb.com/

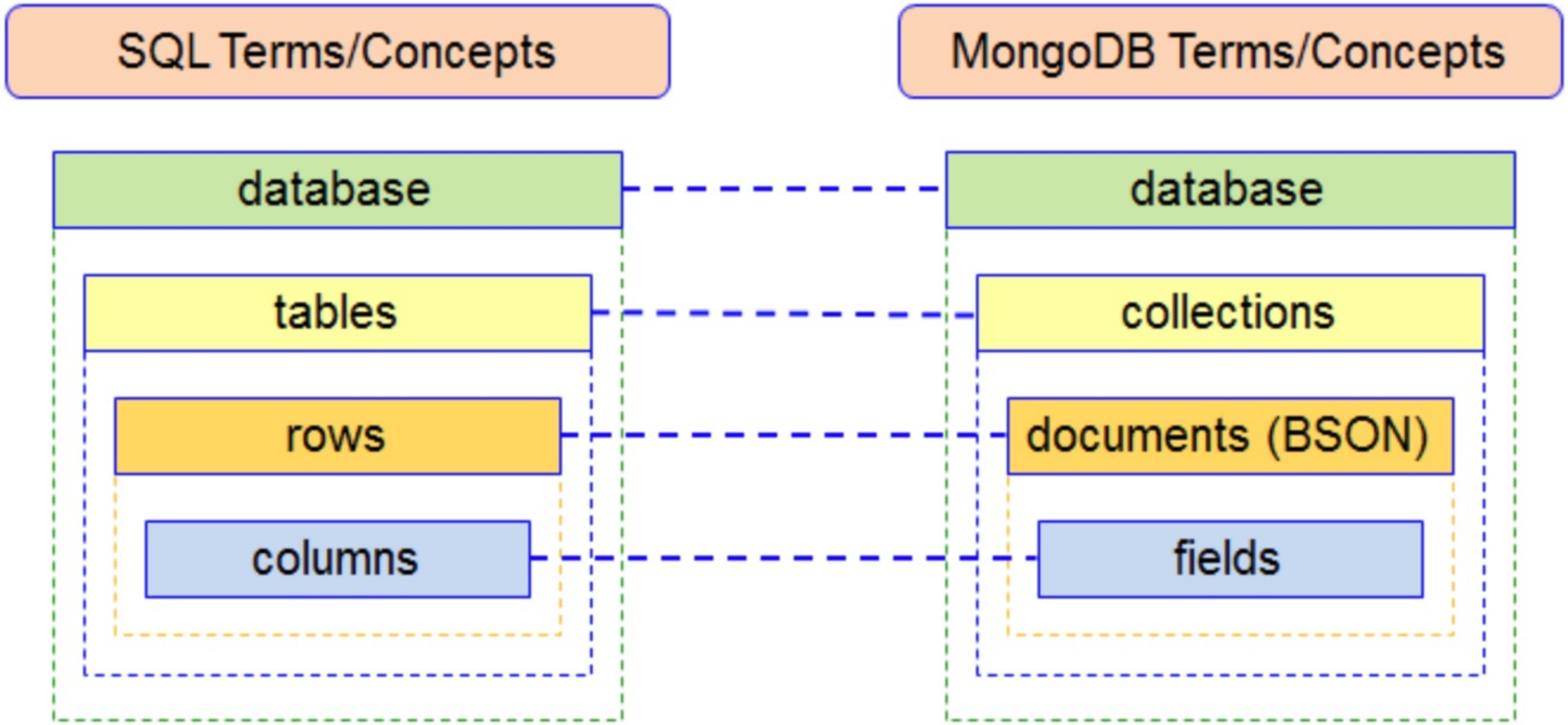
https://docs.mongodb.com/guides/

MongoDB is a source-available, cross-platform, **document-oriented database** program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas.

MongoDB is developed by MongoDB Inc.

Basic terminology

Many of the components that go into the MongoDB data structure have their counterparts in relational database management systems based on the Structured Query Language (SQL).



MongoDB object	Description	SQL object
database	A container for holding document collections. A MongoDB instance can support multiple databases. Each database is associated with its own set of files. MongoDB creates these files when you add the first collection. The database does not physically exist until you add that collection.	database
collection	A grouping of documents in a common namespace. Unlike an SQL table, documents within the same collection can have different structures, resulting in more flexibility than you get with an SQL database.	table
document	Basic unit of data in a MongoDB database. MongoDB leverages the Extended JSON format to store and work with documents. In the background, MongoDB uses BSON (*binary* plus *JSON*) to represent the documents. BSON is a binary-encoded format that extends the JSON model to provide additional data types and other capabilities.	row (or record)

MongoDB object	Description	SQL object
field	A name-value pair in which the value conforms to a BSON data type. MongoDB automatically assigns data types to the field values when you add a document to a collection. You can override this behavior by using a constructor in the document definition to specify a data type.	column
data type	A data attribute that enforces the type of data that can be used in a field. Each field is assigned a data type.	data type
primary key	A required value that uniquely identifies a document within a collection. The primary key is always implemented as the _id field.	

Mapping terms and concepts

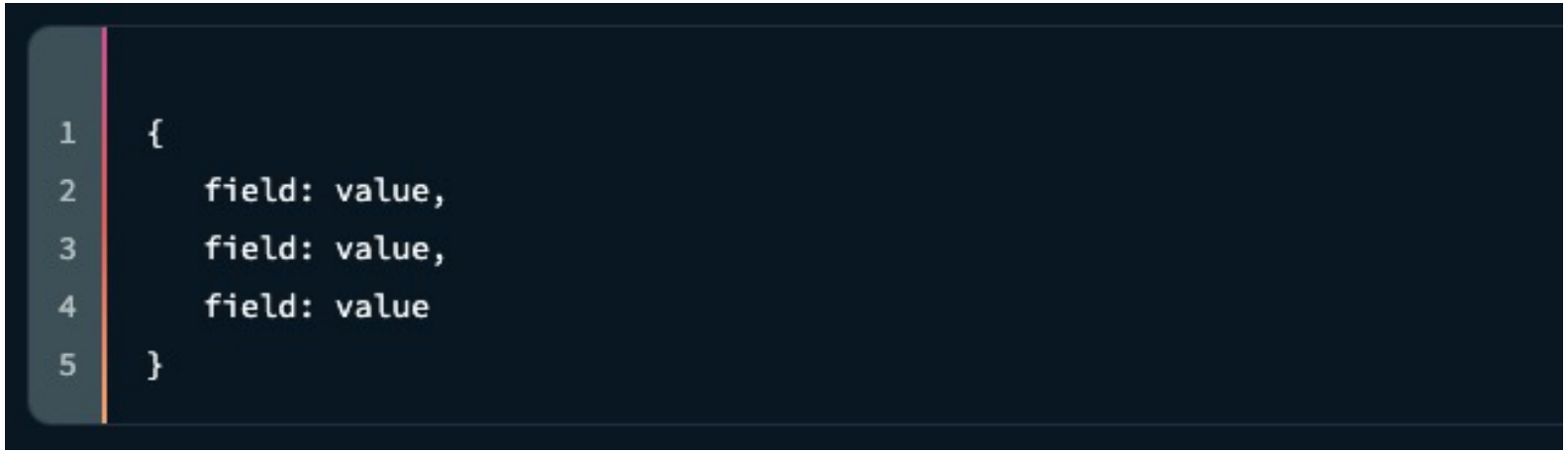
The document model – Instead of tables, MongoDB stores data in documents, BSON documents which are a binary representations of JSON documents.

Documents typically store information about one object as well as any information related to that object. Related documents are grouped together in collections. Related collections are grouped together and stored in a database.

https://www.mongodb.com/developer/article/map-terms-concepts-sql-mongodb/

Every document begins and ends with curly braces.

Inside the curly braces, you'll find an unordered set of field/value pairs that are separated by commas.



Field / Values

- The fields are strings that describe the pieces of data being stored.
- The values can be any of the BSON data types. BSON has a variety of data types including Double, String, Object, Array, Binary Data, ObjectId, Boolean, Date, Null, Regular Expression, JavaScript, JavaScript (with scope), 32-bit Integer, Timestamp, 64-bit Integer, Decimal128, Min Key, and Max Key.
- Every document is required to have a field named _id. The value of _id must be unique for each document in a collection and is immutable,

Example Documents

Let's say we need to store information about a user named Leslie. We'll store her contact information including her first name, last name, cell phone number, and city. We'll also store some extra information about her including her location, hobbies, and job history.

Storing Contact Information

Let's begin with Leslie's contact information. When using SQL, we'll create a table named Users. We can create columns for each piece of contact information and we'll include an ID column.

Users

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

MongoDB version

We create a new document for Leslie where we'll add field/value pairs for each piece of contact information we need to store. We'll use _id to uniquely identify each document. We'll store this document in a **collection** named Users.

Storing Latitude and Longitude

Now that we've stored Leslie's contact information, let's store the coordinates of her current location.

When using SQL, we'll need to split the latitude and longitude between two columns.

Users

ID	first_name	last_name	cell	city	latitude	longitude
1	Leslie	Yepp	8125552344	Pawnee	39.170344	-86.536632

MongoDB version

MongoDB has an array data type, so we can store the latitude and longitude together in a single field.

```
{
    "_id": 1,
    "first_name": "Leslie",
    "last_name": "Yepp",
    "cell": "8125552344",
    "city": "Pawnee",
    "location": [ -86.536632, 39.170344 ]
}
```

Storing lists of information

Now let's store her hobbies. Since a single user could have many hobbies (meaning we need to represent a one-to-many relationship), we'd create a separate table just for hobbies. Each row in the table will contain information about one hobby for one user. When we need to retrieve Leslie's hobbies, we'll join the Users table and our new Hobbies table.

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

MongoDB version

Since MongoDB supports arrays, we can simply add a new field named "hobbies" to our existing document. The array can contain as many or as few hobbies as we need (assuming we don't exceed the 16 megabyte document size limit). When we need to retrieve Leslie's hobbies, we don't need to do an expensive join to bring the data together; we can simply retrieve her document in the Users collection.

```
{
    "_id": 1,
    "first_name": "Leslie",
    "last_name": "Yepp",
    "cell": "8125552344",
    "city": "Pawnee",
    "location": [ -86.536632, 39.170344 ],
    "hobbies": ["scrapbooking", "eating waffles", "working"]
}
```

Storing groups of related information

Let's say we also need to store Leslie's job history.

Just as we did with hobbies, we create a separate table just for job history information. Each row in the table will contain information about one job for one user.

JobHistory

ID	user_id	job_title	year_started
20	1	"Deputy Director"	2004
21	1	"City Councillor"	2012
22	1	"Director, National Parks Service, Midwest Branch"	2014

```
{
   "_id": 1,
   "first_name": "Leslie",
   "last_name": "Yepp",
   "cell": "8125552344",
   "city": "Pawnee",
   "location": [ -86.536632, 39.170344 ],
   "hobbies": ["scrapbooking", "eating waffles", "working"],
   "jobHistory": [
      {
         "title": "Deputy Director",
         "yearStarted": 2004
      },
      {
         "title": "City Councillor",
         "yearStarted": 2012
      },
      {
         "title": "Director, National Parks Service, Midwest Branch",
         "yearStarted": 2014
      }
   ]
}
```

MongoDB version

So far we've used arrays in MongoDB to store geolocation data and a list of Strings.

Arrays can contain values of any type, including objects.

Let's create a document for each job Leslie has held and store those documents in an array.

Skipping data and adding new info

We find we need to store information about a new user: Lauren. We need to store a lot of the same information about Lauren as we did with Leslie : her first name, last name, city, and hobbies. However, Lauren doesn't have a phone, location data, or job history. We also discover that we need to store a new piece of information: her school.

In SQL

Users

ID	first_name	last_name	cell	city	latitude	longitude
1	Leslie	Yepp	8125552344	Pawnee	39.170344	-86.536632
2	Ron	Swandaughter	8125559347	Pawnee	NULL	NULL
3	Lauren	Burhug	NULL	Pawnee	NULL	NULL

Hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working
13	2	woodworking
14	2	fishing
15	3	soccer

Skipping data and adding new info cont

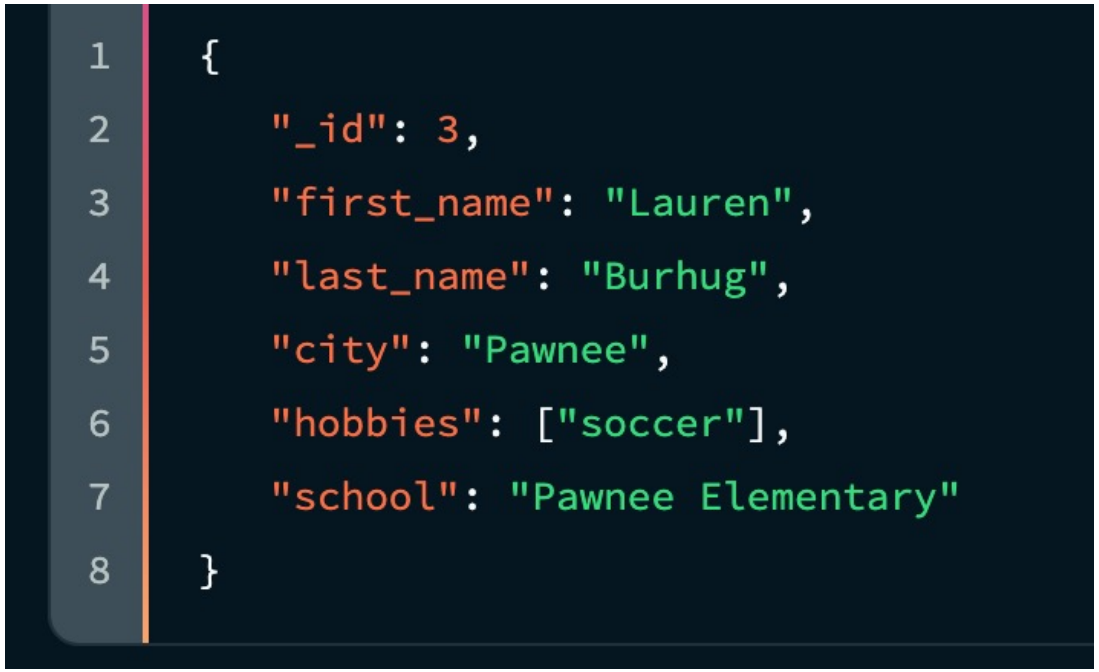
We have two options for storing information about Lauren's school. We can choose to add a column to the existing Users table, or we can create a new table.

Let's say we choose to add a column named "school" to the Users table. Depending on our access rights to the database, we may need to talk to the DBA and convince them to add the field.

Most likely, the database will need to be taken down, the "school" column will need to be added, NULL values will be stored in every row in the Users table where a user does not have a school, and the database will need to be brought back up.

MongoDB version

As you can see below, we've added a new field named "school" to Lauren's document. We do not need to make any modifications to Leslie's document when we add the new "school" field to Lauren's document. MongoDB has a flexible schema, so every document in a collection does not need to have the same fields.

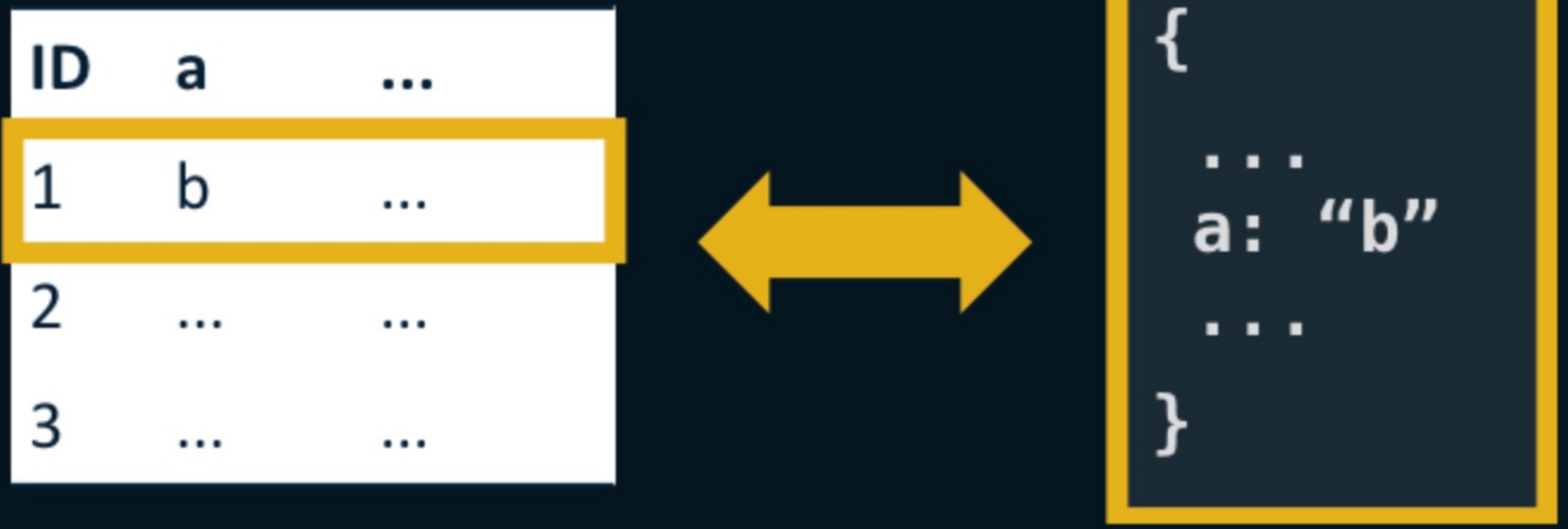


This can be an unsettling idea coming from a strict SQL background where the schema is locked down.

To re-enforce

Row ⇒ Document

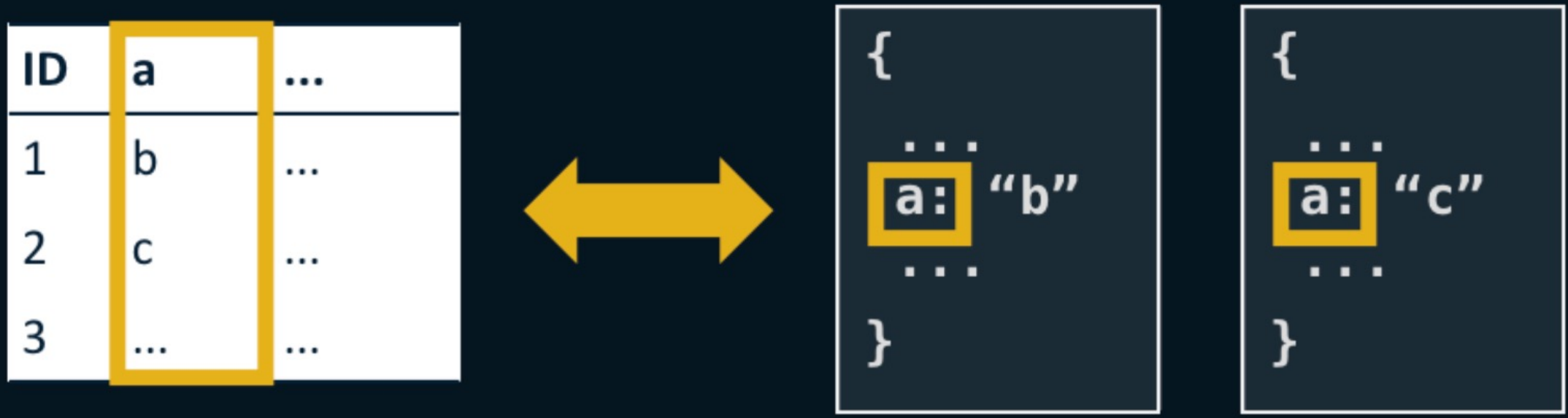
A row maps roughly to a document.



To re-enforce

Column ⇒ Field

A column maps roughly to a field. For example, when we modeled Leslie's data, we had a `first_name` column in the `Users` table and a `first_name` field in a User document.



To re-enforce

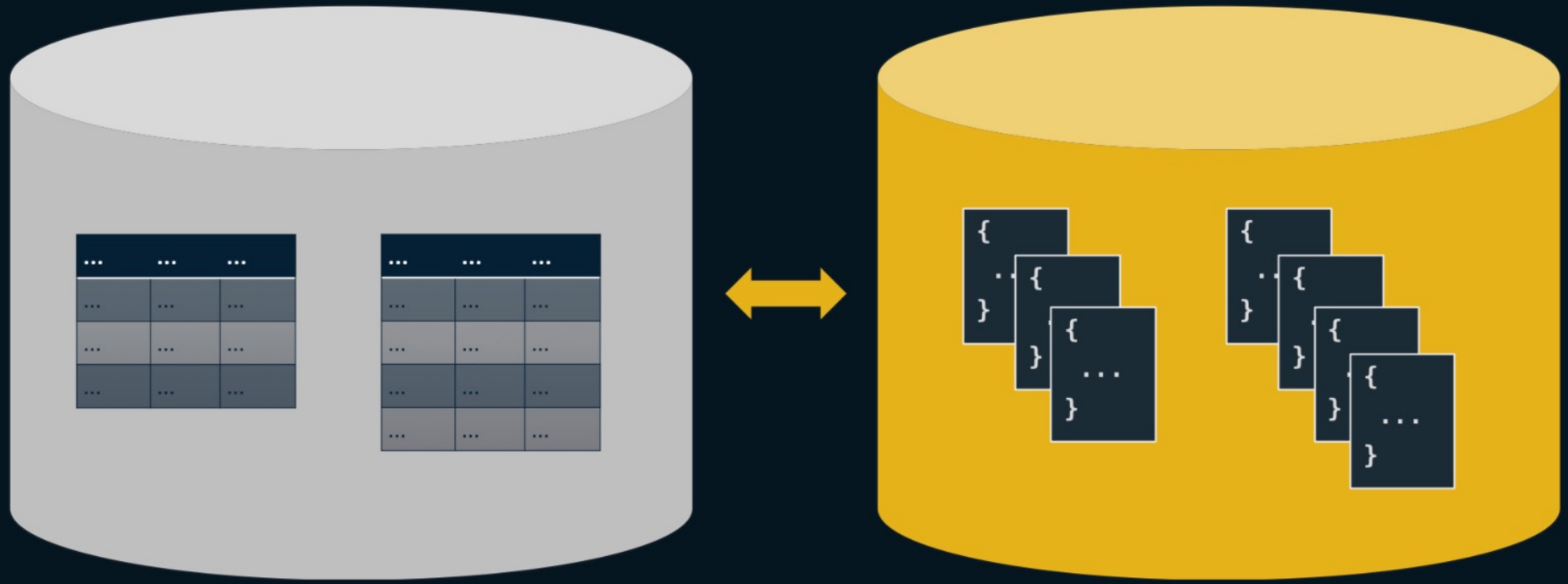
Table ⇒ Collection

A table maps roughly to a collection. Recall that a collection is a group of documents. Continuing with our example above, our `Users` table maps to our `Users` collection.

To re-enforce

Database ⇒ Database

The term `database` is used fairly similarly in both SQL and MongoDB. Groups of tables are stored in SQL databases just as groups of collections are stored in MongoDB databases.



To re-enforce

Index ⇒ Index

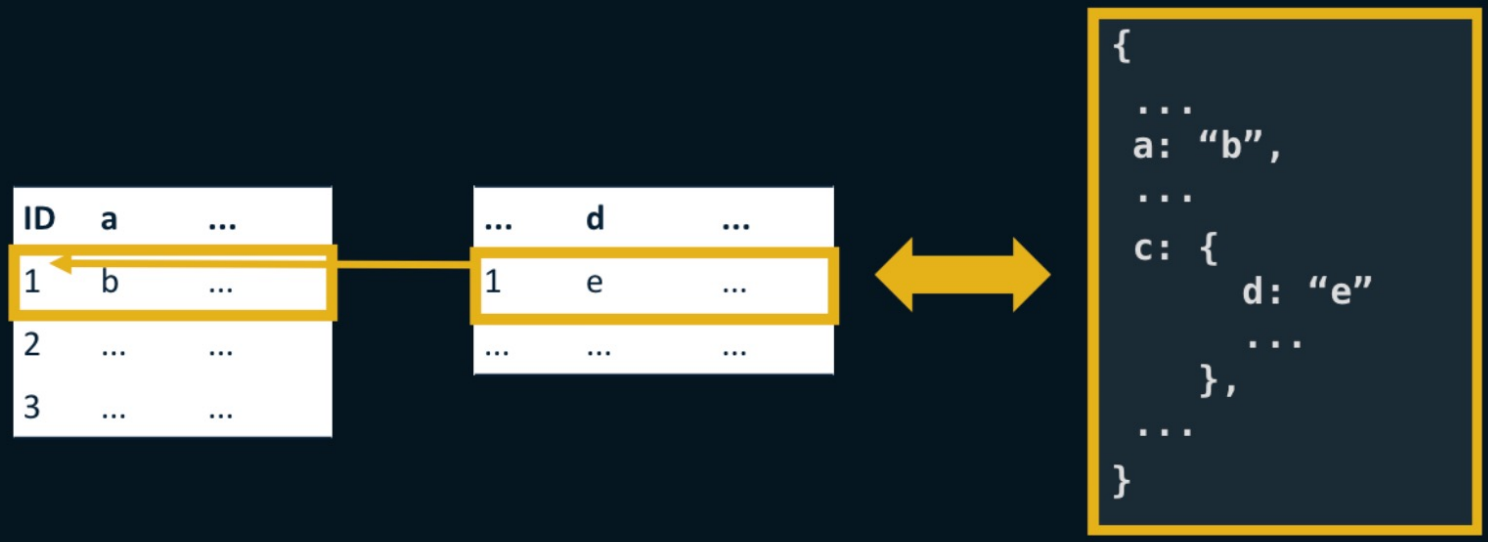
Indexes provide fairly similar functionality in both SQL and MongoDB. Indexes are data structures that optimize queries. You can think of them like an index that you'd find in the back of a book; indexes tell the database where to look for specific pieces of information. Without an index, all information in a table or collection must be searched.

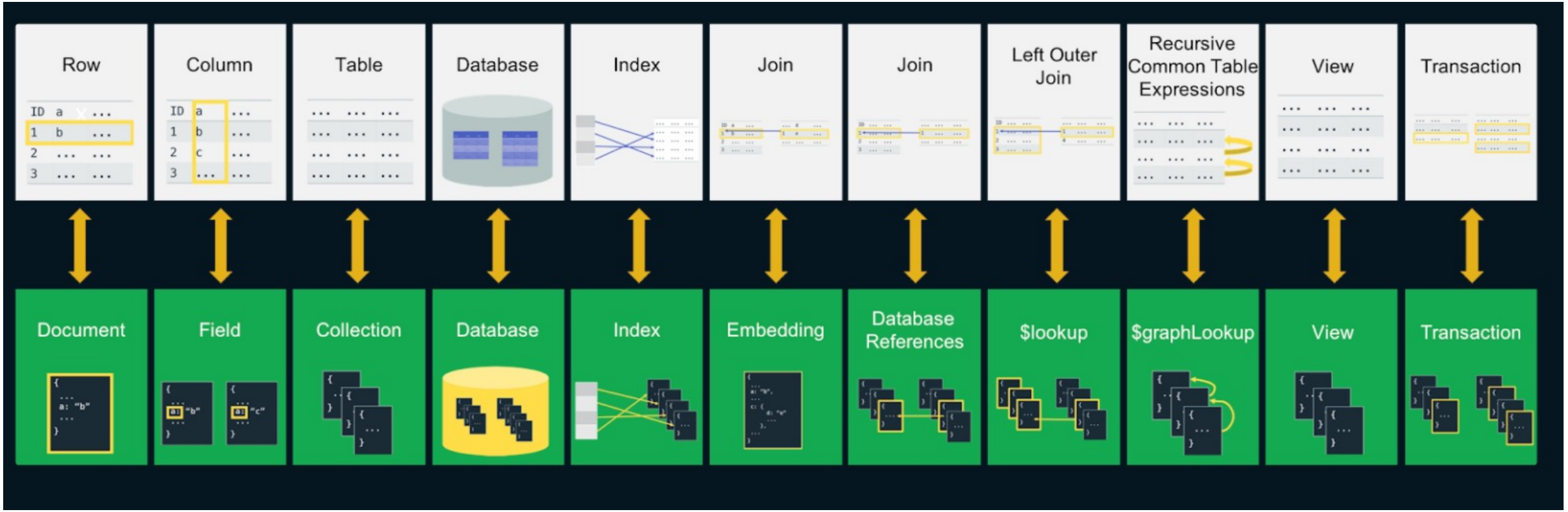
Join ⇒ Embedding

When you use SQL databases, joins are fairly common. You normalize your data to prevent data duplication, and the result is that you commonly need to join information from multiple tables in order to perform a single operation in your application

In MongoDB, we encourage you to model your data differently. Our rule of thumb is *Data that is accessed together should be stored together*. If you'll be frequently creating, reading, updating, or deleting a chunk of data together, you should probably be storing it together in a document rather than breaking it apart across several documents.

You can use embedding to model data that you may have broken out into separate tables when using SQL. When we modeled Leslie's data for MongoDB earlier, we saw that we embedded her job history in her User document instead of creating a separate `JobHistory` document.





Full article can be found below – I suggest you read it !!

https://www.mongodb.com/developer/article/map-terms-concepts-sql-mongodb/

Sources

https://www.imaginarycloud.com/blog/sql-vs-nosql/

https://www.mongodb.com/nosql-explained/nosql-vs-sql

https://www.mongodb.com/nosql-explained

https://www.geeksforgeeks.org/overview-of-scaling-vertical-and-horizontal-scaling/

https://middleware.io/blog/vertical-vs-horizontal-scaling/