NoSQL is an umbrella term to describe any alternative system to traditional SQL databases. NoSQL databases are all quite different from SQL databases. They use a data model that has a different structure than the traditional row-and-column table model used with relational database management systems (RDBMS).But NoSQL databases are all quite different from each other, as well. There are four main types of NoSQL databases. NoSQL ("non SQL" or "not only SQL") databases were developed in the late 2000s with a focus on scaling, fast queries, allowing for frequent application changes, and making programming simpler for developers. Relational databases accessed with SQL (Structured Query Language) were developed in the 1970s with a focus on reducing data duplication as storage was much more costly than developer time. SQL databases tend to have rigid, complex, tabular schemas and typically require expensive vertical scaling.

What NoSQL databases have in common

As explained in When to Use NoSQL Databases, NoSQL databases were developed during the internet era in response to the inability of SQL databases to address the needs of web-scale applications that handled huge amounts of data and traffic.

Companies are finding that they can apply NoSQL technology to a growing list of use cases while saving money in comparison to operating a relational database. NoSQL databases invariably incorporate a flexible schema model and are designed to scale horizontally across many servers, which makes them appealing for large data volumes or application loads that exceed the capacity of a single server.

The popularity of NoSQL has been driven by the following reasons:

- The pace of development with NoSQL databases can be much faster than with a SQL database
- The structure of many different forms of data is more easily handled and evolved with a NoSQL database
- The amount of data in many applications cannot be served affordably by a SQL database
- The scale of traffic and need for zero downtime cannot be handled by SQL
- New application paradigms can be more easily supported

NoSQL databases deliver these benefits in different ways.

Understanding differences in the four types of NoSQL databases

Here are the four main types of NoSQL databases:

- Document databases
- Key-value stores
- Column-oriented databases
- Graph databases

Document databases

A document database stores data in JSON, BSON, or XML documents (not Word documents or Google Docs, of course). In a document database, documents can be nested. Particular elements can be indexed for faster querying.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications, which means less translation is required to use the data in an application. SQL data must often be assembled and disassembled when moving back and forth between applications and storage.

Document databases are popular with developers because they have the flexibility to rework their document structures as needed to suit their application, shaping their data structures as their application requirements change over time. This flexibility speeds development because, in effect, data becomes like code and is under the control of developers. In SQL databases, intervention by database administrators may be required to change the structure of a database.

The most widely adopted document databases are usually implemented with a scale-out architecture, providing a clear path to scalability of both data volumes and traffic.

Use cases include ecommerce platforms, trading platforms, and mobile app development across industries.

Comparing MongoDB vs. PostgreSQL offers a detailed analysis of MongoDB, the leading NoSQL database, and PostgreSQL, one of the most popular SQL databases.

Key-value stores

The simplest type of NoSQL database is a key-value store. Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska").

Use cases include shopping carts, user preferences, and user profiles.

Column-oriented databases

While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). Use cases include analytics.

Unfortunately, there is no free lunch, which means that while columnar databases are great for analytics, the way in which they write data makes it very difficult for them to be strongly consistent as writes of all the columns require multiple write events on disk. Relational databases don't suffer from this problem as row data is written contiguously to disk.

Graph databases

A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.

A graph database is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.

Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases.

Use cases include fraud detection, social networks, and knowledge graphs.

As you can see, despite a common umbrella, NoSQL databases are diverse in their data structures and their applications.

Differences between SQL and NoSQL

The table below summarizes the main differences between SQL and NoSQL databases.

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune

	SQL Databases	NoSQL Databases
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

What are the benefits of NoSQL databases?

NoSQL databases offer many benefits over relational databases. NoSQL databases have flexible data models, scale horizontally, have incredibly fast queries, and are easy for developers to work with.

- Flexible data models

  NoSQL databases typically have very flexible schemas. A flexible schema allows you to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

- Horizontal scaling

  Most SQL databases require you to scale-up vertically (migrate to a larger, more expensive server) when you exceed the capacity requirements of your current server. Conversely, most NoSQL databases allow you to scale-out horizontally, meaning you can add cheaper commodity servers whenever you need to.

- 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.

- Easy for developers

  Some NoSQL databases like MongoDB map their data structures to those of popular programming languages. This mapping allows developers to store their data in the same way that they use it in their application code. While it may seem like a trivial advantage, this mapping can allow developers to write less code, leading to faster development time and fewer bugs.

What are the drawbacks of NoSQL databases?

One of the most frequently cited drawbacks of NoSQL databases is that they don’t support ACID (atomicity, consistency, isolation, durability) transactions across multiple documents. With appropriate schema design, single-record atomicity is acceptable for lots of applications. However, there are still many applications that require ACID across multiple records.

To address these use cases, MongoDB added support for multi-document ACID transactions in the 4.0 release, and extended them in 4.2 to span sharded clusters.

Since data models in NoSQL databases are typically optimized for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.

Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analyzing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.

MongoDB Commands:

```
C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"

C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe



use classdb

db

show dbs

db.createCollection("books")

show collections

db.createCollection("players", {capped : true,  size : 6142800, max : 10000 } )

db.movies.insert({"name" : "chamber of secrets"})

show collections

db.books.insert({

... _id : ObjectId("507f191e810c19729de860ea"),

... title: "MongoDB Overview",

... description: "MongoDB is no sql database",

... by: "packt publishing",

... url: "http://www.packtpubl.com",

... tags: ['mongodb', 'database', 'NoSQL'],

... downloads: 100

... })


db.books.insert([

{
```

```
title: "Mastering Apache Storm",
by: "oreily",
url: "http://www.oreily.com",
tags: ['storm', 'streaming', 'engine'],
downloads: 100,
comments:[{
user:"user1",
message: "My first comment",
dateCreated: new Date(2013,11,10,2,35),
like: 0
}]
},
{
title: "Spark Overview",
description: "Spark Introduction Book",
by: "packt publishing",
url: "http://www.packtpubl.com",
tags: ['spark', 'dstreams', 'RDD'],
downloads: 100000
}
])

db.books.find()
db.books.find().pretty()
db.books.findOne()

db.books.find({$and:[{"by":"packt publishing"},{"title": "Spark Overview"}]}).pretty()

db.books.find({$or:[{"by":"packt publishing"},{"title": "Spark Overview"}]}).pretty()

db.books.insert([
{"title":"D3JS Overview"},
{"title":"Python Overview"},
{"title":"Lua Overview"}
])

db.books.remove({'title':'MongoDB Overview'})
db.books.update({'title':'Spark Overview'}, {$set:{'title':'New Spark Tutorial'}},{multi:true})
```

Data Modeling Introduction

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. 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.

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.

This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the document has substantial variation from other documents in the collection.

In practice, however, the documents in a collection share a similar structure, and you can enforce document validation rules for a collection during update and insert operations. See Schema Validation for details.

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.

For many use cases in MongoDB, the denormalized data model is optimal.

See Embedded Data Models for the strengths and weaknesses of embedding documents.

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.

click to enlarge

See Normalized Data Models for the strengths and weaknesses of using references.

Atomicity of Write Operations

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.

For details regarding transactions in MongoDB, see the Transactions page.

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.

For situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports multi-document transactions:

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
		_id: ,
		Emp_ID: "10025AE336"
		Personal_details:{
				First_Name: "Radhika",
				Last_Name: "Sharma",
				Date_Of_Birth: "1995-09-26"
		},
		Contact: {
				e-mail: "radhika_sharma.123@gmail.com",
				phone: "9848022338"
		},
		Address: {
				city: "Hyderabad",
				Area: "Madapur",
				State: "Telangana"
		}
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
		_id: <ObjectId101>,
		Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
		_id: <ObjectId102>,
		empDocID: " ObjectId101",
		First_Name: "Radhika",
		Last_Name: "Sharma",
		Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
		_id: <ObjectId103>,
		empDocID: " ObjectId101",
		e-mail: "radhika_sharma.123@gmail.com",
            phone: "9848022338"
}
```

Address:

```
{
            _id: <ObjectId104>,
            empDocID: " ObjectId101",
            city: "Hyderabad",
            Area: "Madapur",
            State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

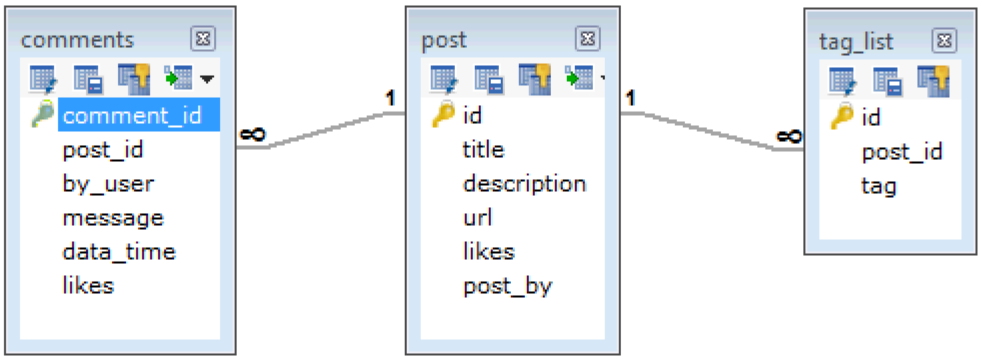
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure –

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

Data Types

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. 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 list or multiple values into one key.
- **Timestamp** – ctimestamp. 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 object of Date and passing day, month, 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 into the document.
- **Regular expression** – This datatype is used to store regular expression.