Overview of NoSQL Databases and MongoDB

What Is NoSQL?

NoSQL stands for “Not Only SQL”. NoSQL database environment is, simply put, a non relational, schema-less and largely distributed database. There are more than one storage mechanism that could be used based on the needs.

Developed in late 2000s to deal with limitations of SQL databases, especially scalability, multi-structured data, geo-distribution and agile development sprints.

NoSQL refers to a broad class of non-relational databases that differ from classical RDBMS in some significant aspects, most notably because they do not use SQL as their primary query language, instead providing access by means of Application Programming Interfaces (APIs).

NoSQL vs SQL

NoSQL is free from joins and relationships while RDBMS use expensive join and relationships

NoSQL has a much lower maintenance cost compared to RDBMS

NoSQL increases the need for developers and database designers while RDBMS does not need much.

Structure and data types are fixed in advance in RDBMS. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.

Why NoSQL?

Agility

Relational databases are a major roadblock to agility, since they do not support agile development very well due to their fixed data model. Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

Relational databases require that schemas be defined before you can add data. This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. If the database is large, this is a very slow process that involves significant downtime.

A core principle of agile development is adapting to evolving application requirements: when the requirements change, the data model also changes. This is a problem for relational databases because the data model is fixed and defined by a static schema. So in order to change the data model, developers have to modify the schema, or worse, request a “schema change” from the database administrators.

Handling Unstructured/Semi-structured Data

RDBMS is sufficient to store and manipulate all the structured data efficiently this makes unavoidable the need to handle unstructured data which is non-relational and schema-less in nature. For RDBMS it becomes a real challenge to provide the cost effective and fast Create, Read, Update and Delete (CRUD) operation as it has to deal with the overhead of joins and maintaining relationships amongst various data. Therefore a new mechanism is required to deal with such data in an easy and efficient way.

Joins are expensive

First, they have a quadratic complexity (O(n^2)) in both time and space. This makes them by far the most expensive operation in a relational algebra implementation, particularly as the number of records increases. Because the space complexity is also quadratic, large joins often run out of memory and need to be stored on disk which makes processing even slower. Joins on small sets of data are usually quite fast.

Second, the reason so many big data systems "don't do joins" is that the algorithms they typically use scale out very poorly. You can easily implement joins on any big data platform but the classic single-server algorithms used in databases don't work well in distributed environments. In summary, joins are "bad" because (1) they are inherently computationally expensive, (2) the unsophisticated implementations in typical big data environments make them even *more* expensive.

Scaling

Scaling vertically meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required, and core relational features such as JOINs, referential integrity and transactions are typically lost.

Horizontal scaling means to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.

Auto-sharding

Because of the way they are structured, relational databases usually scale vertically a single server has to host the entire database to ensure acceptable performance for cross- table joins and transactions. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure.

The solution to support rapidly growing applications is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

Sharding' a database across many server instances can be achieved with SQL databases, other complex arrangements are made for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool.

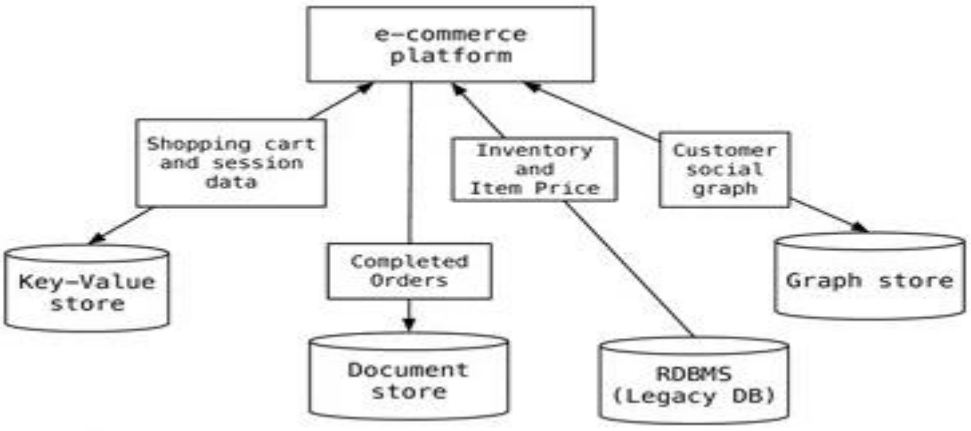
Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary infrastructure administration tasks.

PolyGlot Persistence

Polyglot Persistence means that when storing data, it is best to use multiple data storage technologies, chosen based upon the way data is being used by individual applications or components of a single application. Different kinds of data are best dealt with different data stores. In short, it means picking the right tool for the right use case. It’s the same idea behind Polyglot Programming, which is the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for tackling different problems.

We are entering an era of polyglot persistence, a technique that uses different data storage technologies to handle varying data storage needs.

Polyglot persistence can apply across an enterprise or within a single application.

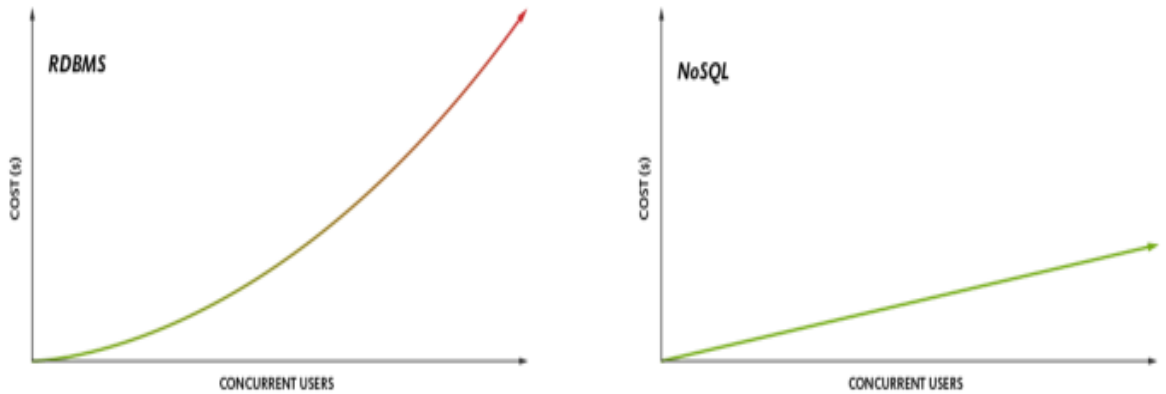


Data Storage Needs

The rise of the web as a platform also created a vital factor change in data storage as the need to support large volumes of data by running on clusters.

Relational databases were not designed to run efficiently on clusters. The data storage needs of an ERP application are lot more different than the data storage needs of a Facebook or an Etsy.

Ensuring global availability is difficult for relational databases where separate add-ons are required – which increases complexity. The business world is undergoing massive change as industry after industry shifts to the Digital Economy.



The Shift to the Digital Economy is Driving NoSQL Adoption

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed.

it is not uncommon for mobile games to reach tens of millions of users in a matter of months e.g. Pokemon GO, Clash of Clans etc. With a distributed, scale-out database, mobile applications can start with a small deployment and expand as the user base grows, rather than deploying an expensive, large relational database server from the beginning.

Replication

Most NoSQL databases also support automatic database replication to maintain availability in the event of outages or planned maintenance events. More sophisticated NoSQL databases are fully self-healing, offering automated failover and recovery, as well as the ability to distribute the database across multiple geographic regions to withstand regional failures and enable data localization.

Unlike relational databases, NoSQL databases generally have no requirement for separate applications or expensive add-ons to implement replication. While relational databases like Oracle require separate software for replication, for example, Oracle Active Data Guard, NoSQL databases do not.

Internet of Things.

Today, some 20 billion devices are connected to the Internet – everything from smartphones and tablets to home appliances and systems installed in cars, hospitals and warehouses. The volume, velocity, and variety of machine-generated data are increasing with the proliferation of digital telemetry, which is semi-structured and continuous. Relational databases struggle with the three well-known challenges from big data IoT applications: Scalability, Throughput, and data variety.

Personalization

A personalized experience requires data, and lots of it – demographic, contextual, behavioral and more. The more data available, the more personalized the experience. Real-Time Big Data. The ability to extract information from operational data in real-time is critical for an agile enterprise.

Integrated Caching

A number of products provide a caching tier for SQL database systems. any NoSQL database technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer.

Analytics and Business Intelligence

Extracting meaningful business intelligence from very high volumes of data is a very difficult task to achieve with traditional relational database systems. Modern NoSQL database systems not only provide storage and management of business application data but also deliver integrated data analytics that deliver instant understanding of complex data sets and facilitate flexible decision-making.

TYPES of NO SQL

Document Stored

Instead of using tables and rows as in relational databases, Document Stored NoSQL databases are built on an architecture of collections and documents. Document databases are essentially the next level of key-value, allowing nested values associated with each key. Document databases support querying more efficiently. An example of a document stored database is MongoDB, CouchDB, RethinkDB. Documents are stored in XML / JSON / BSON.

Documents are the main concept in document databases.. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values.

Documents can contain many different key-value pairs, or key-array pairs, or even nested documents. Document databases are generally useful for content management systems, blogging platforms, web analytics, real-time analytics, ecommerce-applications. We would avoid using document databases for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures.

Key-Value

The main idea here is using a hash table where there is a unique key and a pointer to a particular item of data. Redis, Riak, Local Storage (browser) and Amazon DynamoDB are examples of key-value NoSQL databases . Key-value databases are generally useful for storing session information, user profiles, preferences, shopping cart data. We would avoid using Key-value databases when we need to query by data.

Graph stores

Graph stores are used to store information about networks of data, such as social connections. Graph stores include Neo4J and Giraph.

Column Oriented

These were created to store and process very large amounts of data distributed over many machines. There are still keys but they point to multiple columns. The columns are arranged by column family. Examples of column-oriented databases are Cassandra and HBase. Wide-column stores such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows. Column families are groups of related data that is often accessed together.

CAP THEOREM

As RDBMS follows the ACID property, NoSQL databases are “BASE” Systems. The BASE acronym was defined by Eric Brewer, who is also known for formulating the CAP theorem whose properties are used by BASE System.

The CAP theorem states that a distributed computer system cannot guarantee all of the following three properties at the same time. Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs.

Consistency – once data is written, all future read requests will contain that data

Availability – the database is always available and responsive

Partition tolerance – if one part of the database is unavailable, other parts are unaffected.

Usage of NoSQL Databases at scale

Database stack of Facebook

RocksDB

Embeddable persistent key-value store for fast storage, developed and maintained by Facebook Database Engineering Team.

Cassandra

Column-based database

Memcached

An in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

Database stack of Pokemon GO

Niantic Labs stores and indexes Pokémon Go's data from the game using Google Cloud Datastore's NoSQL database.

Database stack of Clash of Clans

Amazon Web Services(AWS) DynomoDB.

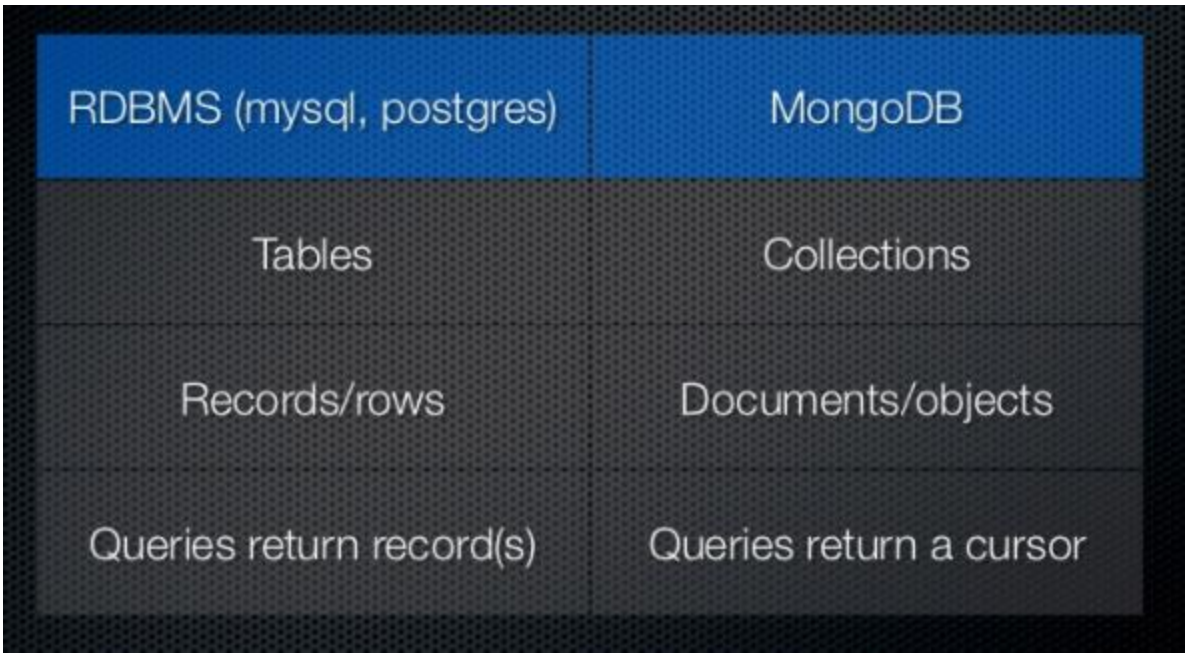
Besides Amazon,Google is also providing backend services for games and apps aswell.Google App Engine is one of the sevices used by Rovio Games ,Ubisoft etc.

MongoDB

MongoDB is a document store with BSON (extension of JSON) type system. Similar documents are organized in databases with collections. To connect to a MongoDB server, use Mongo client (not HTTP).

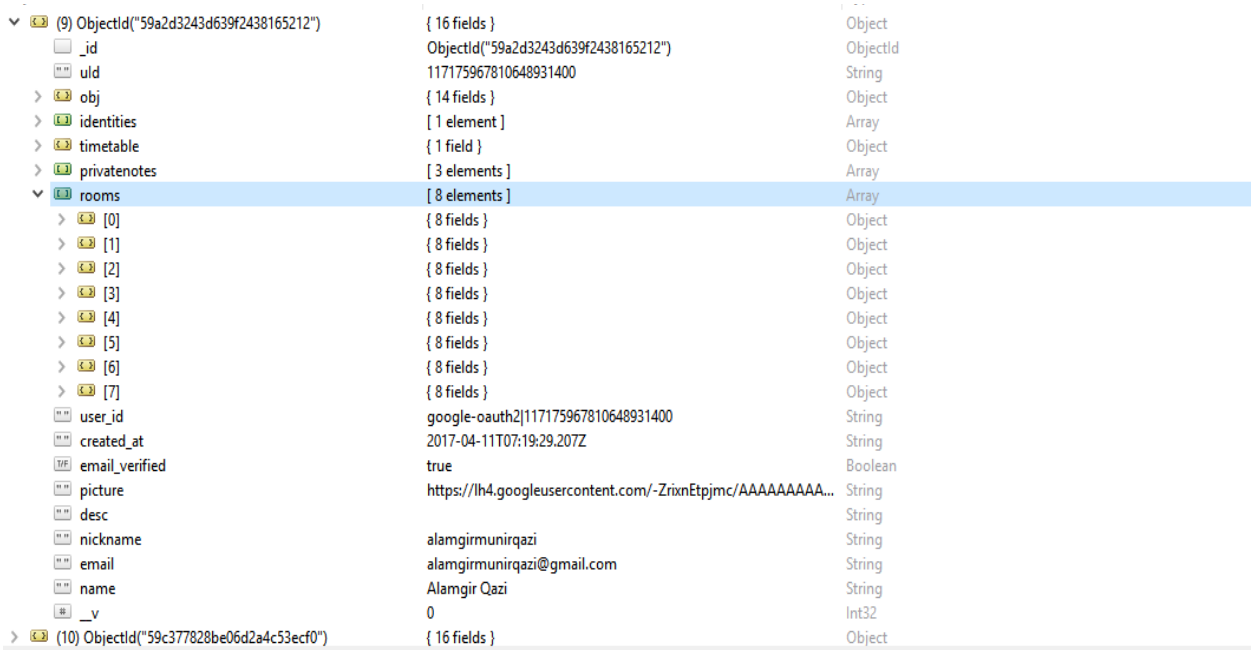
- Each Document usually has a specific key (_id)
- Querying not over HTTP but instead native drivers for each language
- CP in CAP Theorem
- No need of serialization in JavaScript as MongoDB understands JSON directly.
- Individual records can have different fields
- Schema-free structure
- Individual records can have different fields
- Can perform nested queries (more flexible compared to key-value pairs)
- Values may contain documents, arrays and arrays of documents
- Joins or cross collection queries are not available.

MongoDB can store arbitrary deep nested data structures, but cannot search them efficiently. If your data forms a tree, forest or graph, you effectively need to store each node and its edges in a separate document



A cursor allows you to iterate through the result set.

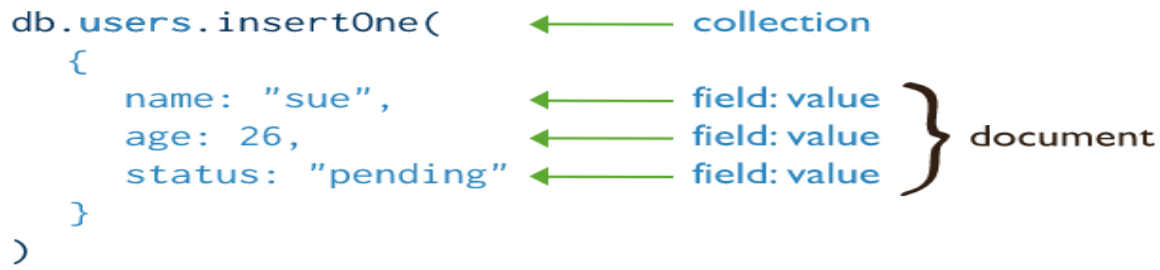
The find() function of MongDB returns a cursor object.



MongoDB CRUD Operations

You can use Mongo shell and write commands on it directly for CRUD operations or you can use an Object Document Mapper (ODM) like Mongoose.

Create



Read

In Mongoose

```
User.find({
    user_id: "123"
  })
   .then(docs => {

     socket.emit("sync success", data);

   })
   .catch(err => {
     console.log("err", err.stack);
   });
```

Delete

```
db.users.deleteMany(
   { status: "reject" }
)
```

Update

```
db.users.updateMany(                    <-- collection
   { age: { $lt: 18 } },                <-- update filter
   { $set: { status: "reject" } }       <-- update action
)
```

MongoDB relationships

- Embedded Documents
- Reference

Embedded Documents

An embedded document is when one document (often a structured text file, or a binary, or anything else) is embedded within another.

Model One-to-One Relationships with Embedded Documents

From

```
{
  _id: "123",
  name: "Leo Messi"
}
{
  user_id: "123",
  street: "123 Fake Street",
  city: "Faketon"
}
```

To

```
{
  _id: "123",
  name: "Leo Messi",
  address: {
            street: "123 Fake Street",
            city: "Faketon"
           }
}
```

Model One-to-Many Relationships with Embedded Documents

```
{
  _id: "123",
  name: "Leo Messi",
  addresses: [
            {
              street: "123 Fake Street",
              city: "Faketon"
            },
            {
              street: "1 Some Other Street",
              city: "Boston"
            }
          ]
}
```

MongoDB Aggregation

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

Aggregation Pipeline

MongoDB's aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document.

Map Reduce

Map function is applied on each document to filter our irrelevant documents, and to emit data for all documents of interest. The emitted data is passed in groups to reduce for aggregation. In general, map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document, and *reduce* phase that combines the output of the map operation. Optionally, map-reduce can have a *finalize* stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional finalize operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline.

Map can be run fully distributed thus allowing analysis of large data sets.