There is no doubt that the way web applications deal with data has changed significantly over the past decade. More data is being collected and more users are accessing this data concurrently than ever before. This means that scalability and performance are more of a challenge than ever for relational databases that are schema-based and therefore can be harder to scale.

**The Evolution of NoSQL**

The SQL scalability issue was recognized by Web 2.0 companies with huge, growing data and infrastructure needs, such as Google, Amazon, and Facebook. They came up with their own solutions to the problem – technologies like [BigTable](https://cloud.google.com/bigtable/" \t "_blank), [DynamoDB](https://aws.amazon.com/dynamodb/" \t "_blank), and [Cassandra](http://cassandra.apache.org/).

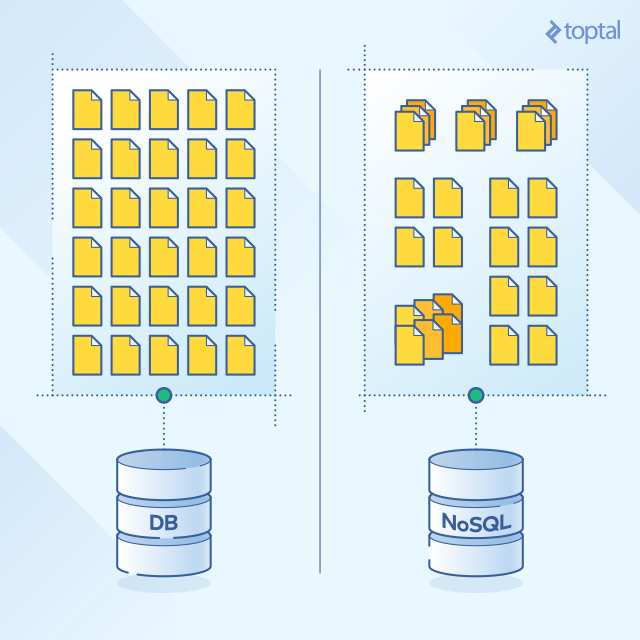
This growing interest resulted in a number of NoSQL Database Management Systems (DBMS’s), with a focus on performance, reliability, and consistency. A number of existing indexing structures were reused and improved upon with the purpose of enhancing search and read performance.

First, there were proprietary (closed source) types of NoSQL databases developed by big companies to meet their specific needs, such as Google’s BigTable, which is believed to be the first NoSQL system, and Amazon’s DynamoDB.

The success of these proprietary systems initiated development of a number of similar open-source and proprietary database systems, the most popular ones being Hypertable, Cassandra, MongoDB, DynamoDB, HBase, and Redis.

**What Makes NoSQL Different?**

One key difference between NoSQL databases and traditional relational databases is the fact that NoSQL is a form of *unstructured storage*.



This means that NoSQL databases do *not* have a fixed table structure like the ones found in relational databases.

**Advantages and Disadvantages of NoSQL Databases**

**Advantages**

NoSQL databases have many advantages compared to traditional, relational databases.

One major, underlying difference is that NoSQL databases have a simple and flexible structure. They are schema-free.

Unlike relational databases, NoSQL databases are based on key-value pairs.

Some store types of NoSQL databases include column store, document store, key value store, graph store, object store, XML store, and other data store modes.

Usually, each value in the database has a key. Some NoSQL database stores also allow developers to store serialized objects into the database, not just simple string values.

Open-source NoSQL databases don’t require expensive licensing fees and can run on inexpensive hardware, rendering their deployment cost-effective.

Also, when working with NoSQL databases, whether they are open-source or proprietary, expansion is easier and cheaper than when working with relational databases. This is because it’s done by horizontally scaling and distributing the load on all nodes, rather than the type of vertical scaling that is usually done with relational database systems, which is replacing the main host with a more powerful one.

**Disadvantages**

Of course, NoSQL databases are not perfect, and they are not always the right choice.

For one thing, most NoSQL databases do not support *reliability features* that are natively supported by relational database systems. These reliability features can be summed up as atomicity, consistency, isolation, and durability. This also means that NoSQL databases, which don’t support those features, trade consistency for performance and scalability.

In order to support reliability and consistency features, [developers](https://www.toptal.com/database) must implement their own proprietary code, which adds more complexity to the system.

This might limit the number of applications that can rely on NoSQL databases for secure and reliable transactions, like banking systems.

Other forms of complexity found in most NoSQL databases include incompatibility with SQL queries. This means that a manual or proprietary querying language is needed, adding even more time and complexity.

**NoSQL vs. Relational Databases**

This table provides a brief feature comparison between NoSQL and relational databases:

|  |  |  |
| --- | --- | --- |
| **Feature** | **NoSQL Databases** | **Relational Databases** |
| **Performance** | High | Low |
| **Reliability** | Poor | Good |
| **Availability** | Good | Good |
| **Consistency** | Poor | Good |
| **Data Storage** | Optimized for huge data | Medium sized to large |
| **Scalability** | High | High (but more expensive) |

It should be noted that the table shows a comparison on the *database level*, not the various *database management systems* that implement both models. These systems provide *their own proprietary techniques* to overcome some of the problems and shortcomings in both systems, and in some cases, significantly improve performance and reliability.

**NoSQL Data Store Types**

**Key Value Store**

In the Key Value store type, a hash table is used in which a unique key points to an item.

Keys can be organized into logical groups of keys, only requiring keys to be unique within their own group. This allows for identical keys in different logical groups. The following table shows an example of a key-value store, in which the key is the name of the city, and the value is the address for Ulster University in that city.

|  |  |
| --- | --- |
| **Key** | **Value** |
| **"Belfast"** | {“University of Ulster, Belfast campus, York Street, Belfast, BT15 1ED”} |
| **“Coleraine"** | {“University of Ulster, Coleraine campus, Cromore Road, Co. Londonderry, BT52 1SA”} |

Some implementations of the key value store provide caching mechanisms, which greatly enhance their performance.

All that is needed to deal with the items stored in the database is the key. Data is stored in a form of a string, JSON, or BLOB (Binary Large OBject).

One of the biggest flaws in this form of database is the lack of consistency at the database level. This can be added by the developers with their own code, but as mentioned before, this adds more effort, complexity, and time.

The most famous NoSQL database that is built on a key value store is Amazon’s DynamoDB.

**Document Store**

Document stores are similar to key value stores in that they are schema-less and based on a key-value model. Both, therefore, share many of the same advantages and disadvantages. Both lack consistency on the database level, which makes way for applications to provide more reliability and consistency features.

There are however, key differences between the two.

In Document Stores, the values (documents) provide encoding for the data stored. Those encodings can be XML, JSON, or [BSON (Binary encoded JSON)](http://bsonspec.org/).

Also, querying based on data can be done.

The most popular database application that relies on a Document Store is MongoDB.

**Column Store**

In a Column Store database, data is stored in columns, as opposed to being stored in rows as is done in most relational database management systems.

A Column Store is comprised of one or more Column Families that logically group certain columns in the database. A key is used to identify and point to a number of columns in the database, with a keyspace attribute that defines the scope of this key. Each column contains tuples of names and values, ordered and comma separated.

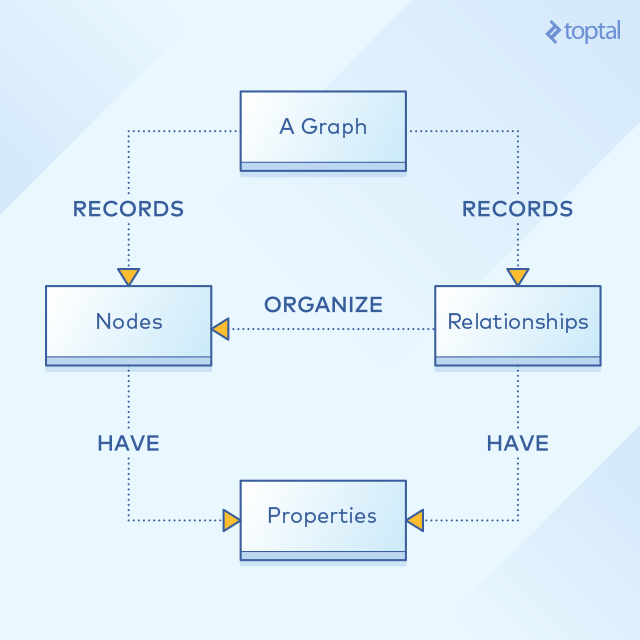
Column Stores have fast read/write access to the data stored. In a column store, rows that correspond to a single column are stored as a single disk entry. This makes for faster access during read/write operations.

The most popular databases that use the column store include Google’s BigTable, HBase, and Cassandra.

**Graph Base**

In a Graph Base NoSQL Database, a directed graph structure is used to represent the data. The graph is comprised of edges and nodes.

Formally, a graph is a representation of a set of objects, where some pairs of the objects are connected by links. The interconnected objects are represented by mathematical abstractions, called vertices, and the links that connect some pairs of vertices are called edges. A set of vertices and the edges that connect them is said to be a graph.



This illustrates the structure of a graph base database that uses edges and nodes to represent and store data. These nodes are organized by some relationships with one another, which is represented by edges between the nodes. Both the nodes and the relationships have some defined properties.

Graph databases are most typically used in social networking applications. Graph databases allow developers to focus more on relations between objects rather than on the objects themselves. In this context, they indeed allow for a scalable and easy-to-use environment.

Currently, InfoGrid and InfiniteGraph are the most popular graph databases.

Top of Form

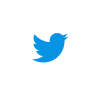
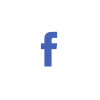
**Like what you're reading?**

*Get the latest updates first.*





No spam. Just great engineering posts.

* 
* 
* 

Bottom of Form

**NoSQL Database Management Systems**

For a brief comparison of the databases, the following table provides a brief comparison between different NoSQL database management systems.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Storage Type** | **Query Method** | **Interface** | **Programming Language** | **Open Source** | **Replication** |
| **Cassandra** | Column Store | Thrift API | Thrift | Java | Yes | Async |
| **MongoDB** | Document Store | Mongo Query | TCP/IP | C++ | Yes | Async |
| **HyperTable** | Column Store | HQL | Thrift | Java | Yes | Async |
| **CouchDB** | Document Store | MapReduce | REST | Erlang | Yes | Async |
| **BigTable** | Column Store | MapReduce | TCP/IP | C++ | No | Async |
| **HBase** | Column Store | MapReduce | REST | Java | Yes | Async |

MongoDB has a flexible schema storage, which means stored objects are not necessarily required to have the same structure or fields. MongoDB also has some optimization features, which distributes the data collections across, resulting in overall performance improvement and a more balanced system.

Other NoSQL database systems, such as Apache CouchDB, are also document store type database, and share a lot of features with MongoDB, with the exception that the database can be accessed using RESTful APIs.

REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within the World Wide Web. It relies on a stateless, client-server, cacheable communications protocol (e.g., the HTTP protocol).

RESTful applications use HTTP requests to post, read data, and delete data.

As for column base databases, Hypertable is a NoSQL database written in C++ and is based on Google’s BigTable.

Hypertable supports distributing data stores across nodes to maximize scalability, just like MongoDB and CouchDB.

One of the most widely used NoSQL databases is Cassandra, developed by Facebook.

Cassandra is a column store database that includes a lot of features aimed at reliability and fault tolerance.

Rather than providing an in-depth look at each NoSQL DBMS, Cassandra and MongoDB, two of the most widely used NoSQL database management systems, will be explored in the next subsections.

**Cassandra**

Cassandra is a database management system developed by Facebook.

The goal behind Cassandra was to create a DBMS that has no single point of failure and provides maximum availability.

Cassandra is mostly a column store database. Some studies referred to Cassandra as a hybrid system, inspired by Google’s BigTable, which is a column store database, and Amazon’s DynamoDB, which is a key-value database.

This is achieved by providing a key-value system, but the keys in Cassandra point to a set of column families, with reliance on Google’s BigTable distributed file system and Dynamo’s availability features (distributed hash table).

Cassandra is designed to store huge amounts of data distributed across different nodes. Cassandra is a DBMS designed to handle massive amounts of data, spread out across many servers, while providing a highly available service with no single point of failure, which is essential for a big service like Facebook.

The main features of Cassandra include:

* **No single point of failure.** For this to be achieved, Cassandra must run on a cluster of nodes, rather than a single machine. That doesn’t mean that the data on each cluster is the same, but the management software is. When a failure in one of the nodes happens, the data on that node will be inaccessible. However, other nodes (and data) will still be accessible.
* **Distributed Hashing** is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. This provides the ability to distribute the load to servers or nodes according to their capacity, and in turn, minimize downtime.
* **Relatively easy to use Client Interface**. Cassandra uses Apache Thrift for its client interface. Apache Thrift provides a cross-language RPC client, but most developers prefer open-source alternatives built on top of Apple Thrift, such as Hector.
* **Other availability features.** One of Cassandra’s features is data replication. Basically, it mirrors data to other nodes in the cluster. Replication can be random, or specific to maximize data protection by placing in a node in a different data center, for example. Another feature found in Cassandra is the partitioning policy. The partitioning policy decides where on which node to place the key. This can also be random or in order. When using both types of partitioning policies, Cassandra can strike a balance between load balancing and query performance optimization.
* **Consistency.** Features like replication make consistency challenging. This is due to the fact that all nodes must be up-to-date at any point in time with the latest values, or at the time a read operation is triggered. Eventually, though, Cassandra tries to maintain a balance between replication actions and read/write actions by providing this customizability to the developer.
* **Read/Write Actions.** The client sends a request to a single Cassandra node. The node, according to the replication policy, stores the data to the cluster. Each node first performs the data change in the commit log, and then updates the table structure with the change, both done synchronously. The read operation is also very similar, a read request is sent to a single node, and that single node is the one that determines which node holds the data, according to the partitioning/placement policy.

**MongoDB**

MongoDB is a schema-free, document-oriented database written in C++. The database is document store based, which means it stores values (referred to as documents) in the form of encoded data.

The choice of encoded format in MongoDB is JSON. This is powerful, because even if the data is nested inside JSON documents, it will still be *queryable* and *indexable*.

The subsections that follow describe some of the key features available in MongoDB.

**Shards**

Sharding is the partitioning and distributing of data across multiple machines (nodes). A shard is a collection of MongoDB nodes, in contrast to Cassandra where nodes are symmetrically distributed. Using shards also means the ability to horizontally scale across multiple nodes. In the case that there is an application using a single database server, it can be converted to sharded cluster with very few changes to the original application code because the way sharding is done by MongoDB. oftware is almost completely decoupled from the public APIs exposed to the client side.

**Mongo Query Language**

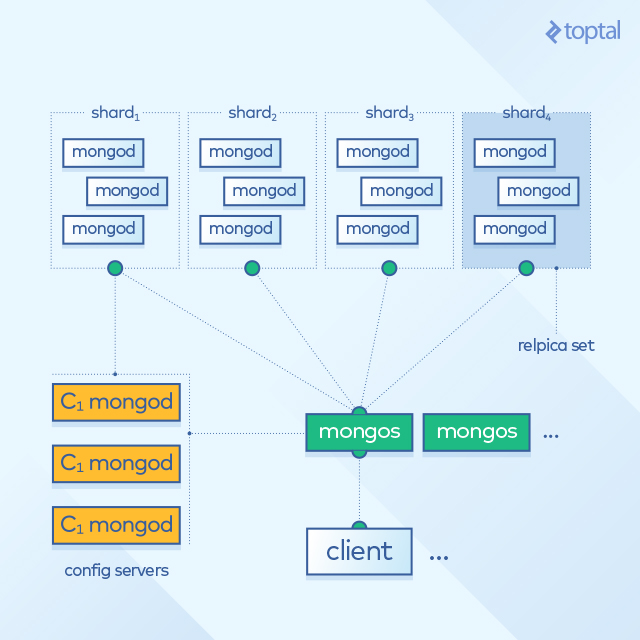
As discussed earlier, MongoDB uses a RESTful API. To retrieve certain documents from a db collection, a query document is created containing the fields that the desired documents should match.

**Actions**

In MongoDB, there is a group of servers called routers. Each one acts as a server for one or more clients. Similarly, The cluster contains a group of servers called configuration servers. Each one holds a copy of the metadata indicating which shard contains what data. Read or write actions are sent from the clients to one of the router servers in the cluster, and are automatically routed by that server to the appropriate shards that contain the data with the help of the configuration servers.

Similar to Cassandra, a shard in MongoDB has a data replication scheme, which creates a replica set of each shard that holds exactly the same data. There are two types of replica schemes in MongoDB: Master-Slave replication and Replica-Set replication. Replica-Set provides more automation and better handling for failures, while Master-Slave requires the administrator intervention sometimes. Regardless of the replication scheme, at any point in time in a replica set, only one shard acts as the primary shard, all other replica shards are secondary shards. All write and read operations go to the primary shard, and are then distributed evenly (if needed) to the other secondary shards in the set.

In the graphic below, we see the MongoDB architecture explained above, showing the router servers in green, the configuration servers in yellow, and the shards that contain the blue MongoDB nodes.



It should be noted that sharding (or sharing the data between shards) in MongoDB is completely automatic, which reduces the failure rate and makes MongoDB a highly scalable database management system.

**Indexing Structures for NoSQL Databases**

Indexing is the process of associating a key with the location of a corresponding data record in a DBMS. There are many indexing data structures used in NoSQL databases. The following sections will briefly discuss some of the more common methods; namely, B-Tree indexing, T-Tree indexing, and O2-Tree indexing.

**B-Tree Indexing**

B-Tree is one of the most common index structures in DBMS’s.

In B-trees, internal nodes can have a variable number of child nodes within some predefined range.

One major difference from other tree structures, such as AVL, is that B-Tree allows nodes to have a variable number of child nodes, meaning less tree balancing but more wasted space.

The B+-Tree is one of the most popular variants of B-Trees. The B+-Tree is an improvement over B-Tree that requires all keys to reside in the leaves.

**T-Tree Indexing**

The data structure of T-Trees was designed by combining features from AVL-Trees and B-Trees.

AVL-Trees are a type of self-balancing binary search trees, while B-Trees are unbalanced, and each node can have a different number of children.

In a T-Tree, the structure is very similar to the AVL-Tree and the B-Tree.

Each node stores more than one {key-value, pointer} tuple. Also, binary search is utilized in combination with the multiple-tuple nodes to produce better storage and performance.

A T-Tree has three types of nodes: A T-Node that has a right and left child, a leaf node with no children, and a half-leaf node with only one child.

It is believed that T-Trees have better overall performance than AVL-Trees.

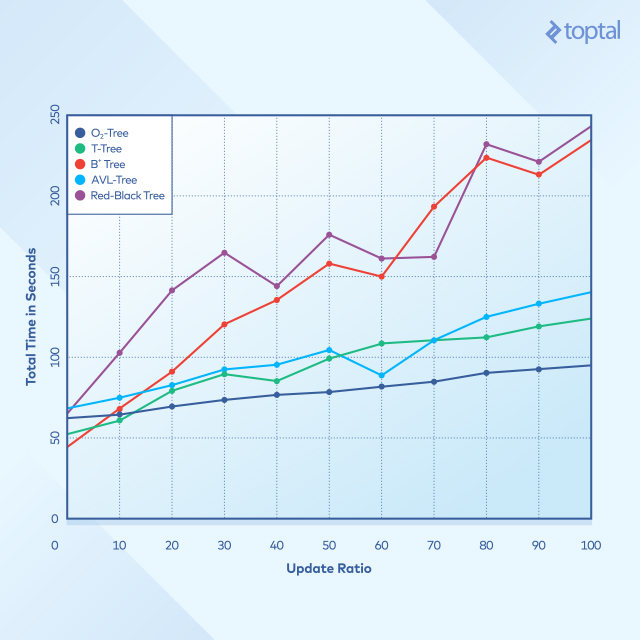
**O2-Tree Indexing**

The O2-Tree is basically an improvement over Red-Black trees, a form of a Binary-Search tree, in which the leaf nodes contain the {key value, pointer} tuples.

O2-Tree was proposed to enhance the performance of current indexing methods. An O2-Tree of order m (m ≥ 2), where m is the minimum degree of the tree, satisfies the following properties:

* Every node is either red or black. The root is black.
* Every leaf node is colored black and consists of a block or page that holds “key value, record-pointer” pairs.
* If a node is red, then both its children are black.
* For each internal node, all simple paths from the node to descendant leaf-nodes contain the same number of black nodes. Each internal node holds a single key value.
* Leaf-nodes are blocks that have between ⌈m/2⌉ and m “key-value, record-pointer” pairs.
* If a tree has a single node, then it must be a leaf, which is the root of the tree, and it can have between 1 to m key data items.
* Leaf nodes are double-linked in forward and backward directions.

Here, we see a straightforward performance comparison between O2-Tree, T-Tree, B+-Tree, AVL-Tree, and Red-Black Tree:



The order of the T-Tree, B+-Tree, and the O2-Tree used was m = 512.

Time is recorded for operations of search, insert, and delete with update ratios varying between 0%-100% for an index of 50M records, with the operations resulting in adding another 50M records to the index.

It is clear that with an update ratio of 0-10%, B-Tree and T-Tree perform better than O2-Tree. However, with the update ratio increasing, O2-Tree index performs significantly better than most other data structures, with the B-Tree and Red-Black Tree structures suffering the most.

**The Case for NoSQL?**

A quick introduction to NoSQL databases, highlighting the key areas where traditional relational databases fall short, leads to the first takeaway:

*“While relational databases offer consistency, they are not optimized for high performance in applications where massive data is stored and processed frequently.”*

NoSQL databases gained a lot of popularity due to high performance, high scalability and ease of access; however, they still *lack features that provide consistency and reliability.*

Fortunately, a number of NoSQL DBMSs address these challenges by offering new features to enhance scalability and reliability.

Not all NoSQL database systems perform better than relational databases.

MongoDB and Cassandra have similar, and in most cases better, performance than relational databases in write and delete operations.

There is no direct correlation between the store type and the performance of a NoSQL DBMS. NoSQL implementations undergo changes, so performance may vary.

Therefore, performance measurements across database types in different studies should **always** be updated with the latest versions of database software in order for those numbers to be accurate.

While I can’t offer a definitive verdict on performance, here are a few points to keep in mind:

* Traditional B-Tree and T-Tree indexing is commonly used in traditional databases.
* One study offered improvements and enhancements by combining the characteristics of multiple indexing structures to come up with the O2-Tree.
* The O2-Tree outperformed other structures in most tests, especially with huge datasets and high update ratios.
* The B-Tree structure delivered the worst performance of all indexing structures covered in this article.

Further work can and should be done to enhance the consistency of NoSQL DBMSs. The integration of both systems, NoSQL and relational databases, is an area to further explore.

Finally, it’s important to note that NoSQL is a good addition to existing database standards, but with a few important caveats. NoSQL trades reliability and consistency features for sheer performance and scalability. This renders it a specialized solution, as the number of applications that can rely on NoSQL databases remains limited.

The upside? Specialization might not offer much in the way of flexibility, but when you want to get a specialized job done as quickly and efficiently as possible, you don’t need a Swiss Army Knife. You need NoSQL.