NoSQL databases

Introduction

Relational databases were designed with optimal efficiency in mind and over the last 40 years the technology has simply got better and better. However, the huge growth of unstructured and semi-structured data is testing the limits of the model and a number of alternative architectures are now available. Before abandoning the relational model for something new and shiny, however, it is important to know a little about the alternatives and their advantages and disadvantages.

The general term for the range of new architectures is NoSQL. This is not a suggestion that SQL is no longer relevant and the generally accepted interpretation of the name is that it means Not Only SQL. The implication is that the selection of database platform depends on the context of use and that the relational model is still the best choice in certain cases.

Types of NoSQL database

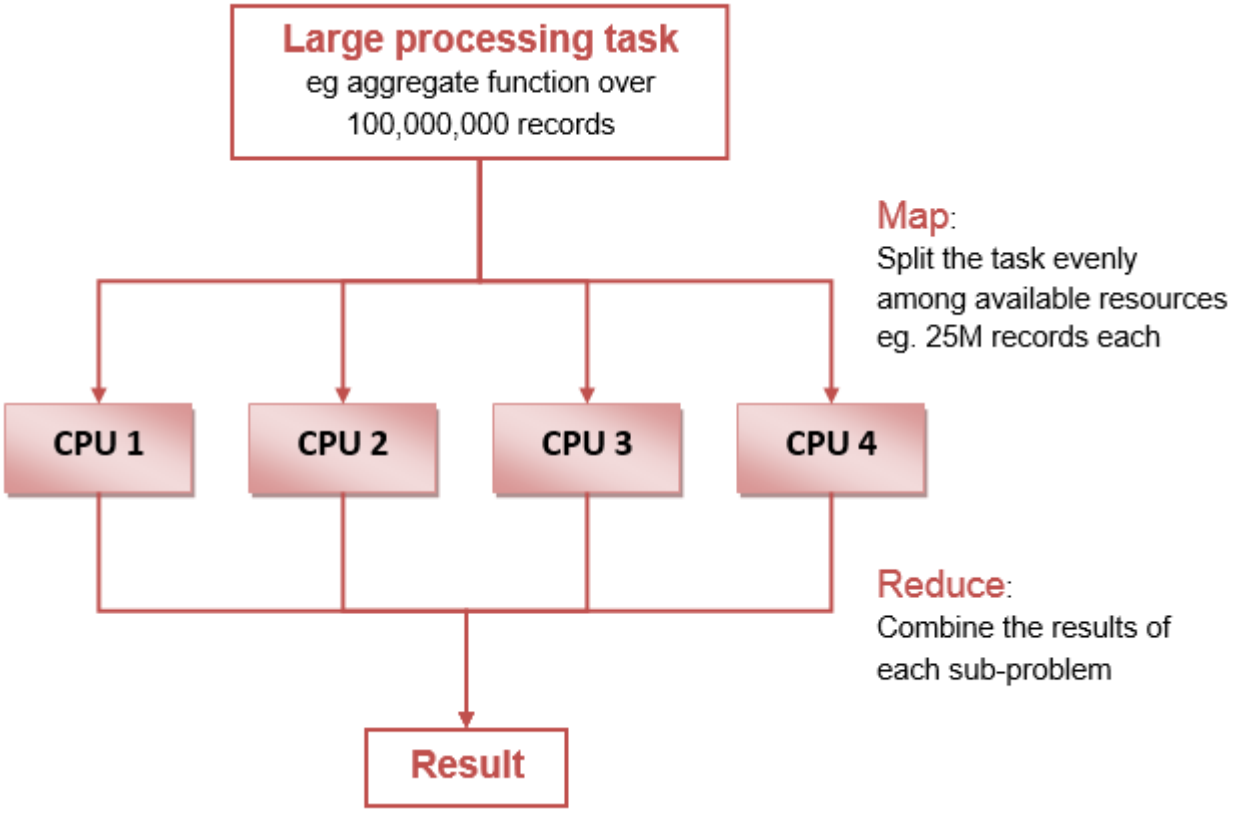
The top link on the right will take you to a page which lists around 150 different databases which fall under the NoSQL heading. There is no one single architectural model, but there are four main categories summarised in the table below. Further description is provided by Moniruzzaman & Hussain (2013).

Type	Description
Key-Value stores	The main goal is to scale to very large data sets while providing acceptable performance. The model is a global collection of key-value pairs based on deCandia's 2007 paper for Amazon. A key-value store reduces the complexity of the data structure to the minimum. Queries can be performed against keys but not against any part of the associated values. Only exact matches are supported to maintain high performance.
Column stores	As with key-value stores, the goals are scalability and speed but the data is assumed to be more structured. This model is based on the work by Chang et al. (2008) on Google's BigTable storage system. The model is similar to the relational model in that it can be visualised as a grid, but a major difference is that column stores are designed to be distributed to allow parallelisation of queries. Operations rely heavily on the MapReduce framework described below.
Document stores	Designed for convenient storage of unstructured and semi-structured data using common object notations such as JavaScript Object Notation (JSON), document stores integrate very well with Web applications. Document stores provide a combination of indexed keys for fast retrieval and semi-structured values which can also be addressed by queries. They therefore represent a compromise between the structured data in a relational database and the stripped-down and limited structure of a key-value store.
Graph stores	Based on mathematical graph theory, graph stores represent data as a set of key-value pairs (nodes) which are connected explicitly by relationships (edges). The main goal of the model is not performance but the convenient visual representation of relationships between objects.

Scalability

Part of the reason for the growing interest in new database models comes from the need to handle very large datasets. Relational databases still provide superior performance in transactional systems where the quantities of data are relatively low. However, they are essentially designed to run on a single machine, and the only way they can be scaled to cope with higher processing demands is to use a higher-specification machine - ie. one with faster processors, more processor cores, more memory, higher-throughput drives, etc. This is known as vertical scaling. NoSQL database, in contrast, are designed for horizontal scaling in which the processing load can be spread across more machines. This model has several advantages for cost and reliability. The computers on which the database runs can be low-cost commodity machines rather than the specialist, high-specification machines required for an equivalent relational database. Also, if one of those machines fails for some reason, processing can continue on the rest. A further advantage of the distributed model is that it works well with virtual cloud-based resources. In an environment such as Amazon's commodity cloud infrastructure, new virtual machines can be added to a cluster as the processing load increases.

Some NoSQL databases directly implement a well-known model for parallel processing called *MapReduce*. While not the only approach, MapReduce provides a good example of how a complex processing operation can be efficiently split over a cluster of processors:



NoSQL databases typically include another approach to horizontal scaling called *sharding*. In contrast to MapReduce, sharding involves splitting the data into subsets according to some data value. A set of student data, for example, might be split into 100 shards using the matriculation number by dividing the range of values into 100 roughly equal sets. A query based on matriculation number would then only be run on the relevant shard, thus reducing the total amount of processing required. Sharding reduces the *search space* in a similar way to the indexes found in traditional relational databases.

BASE

Relational databases are designed around the principles of atomicity, consistency, isolation and durability of transactions. The distributed nature of NoSQL databases means that these principles cannot be guaranteed. In particular, consistency is sacrificed in favour of performance. This follows from Brewer's CAP theorem (Gilbert & Lynch, 2002), which says that only two out of the three principles of consistency, availability and partition tolerance are possible at any one time. The thinking is that if a database is spread across multiple partitions, then different partitions can receive conflicting commands. The principle of availability says that each should respond immediately, but that would lead to inconsistent responses; the principle of consistency would require the conflicts to resolved before replying which would take time thus reducing availability.

NoSQL databases are designed around an alternative set of principles known as BASE: Basically Available Soft state Eventually consistent. While ensuring that the system is basically available to any requests, a consistent state cannot be guaranteed and the state of the system is treated as soft - ie temporary inconsistencies are tolerated. The system eventually becomes consistent at some point, but this is typically the responsibility of the application developer rather than the database management system as it is in relational databases.

Pritchett (2008) provides a detailed illustration of the BASE approach using a worked example and the standard concepts from relational transactions.

Comparison with SQL

Terminology

Conceptually, many of the terms already familiar from relational databases can be mapped onto equivalents in the NoSQL world. The following table illustrates this using the terminology employed by MongoDB, a document store.

SQL terminology	MongoDB terminology
database	database
table	collection
row	document
column	field
index	index
table joins	embedded documents and linking
primary key	primary key

Performance

Most NoSQL databases claim to have better performance than traditional relational systems. It is important to approach this sort of claim critically, however. NoSQL databases are designed to solve certain types of data management problem, and it is on those specific problems where performance tends to be the best. Simple inserts or updates based on the primary key, for example, tend to be very fast, but the traditional SQL approach performs better when more complex processing is required, or when accessing data based on non-key attributes.

The graphs below illustrate some of the performance differences observed between Microsoft SQL Server and MongoDB in a redent study (Parker et al, 2013). In their tests, the authors performed

equivalent operations on the two databases on datasets of different sizes as shown in the following table..

Test case	Departments	Users	Projects
1	4	128	16
2	4	256	64
3	16	1024	512
4	128	4096	8192

Figs. 1 and 2 show the superior performance of MongDB for update operations that rely on key field, and the poorer performance when updates reply on a non-key field to locate documents/records. This difference in performance can be explained with reference to the design of the MongoDB data objects: key fields are indexed for fast searching, but because the document is unstructured, other attributes may or may not be present. This entails a lot of processing overhead for checking each document for the required value. In relational databases, however, the regular structure of each record support ad-hoc queries much better.

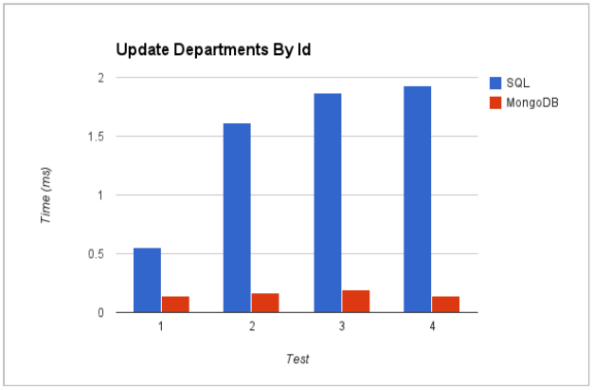


Figure 1: Update using a key field

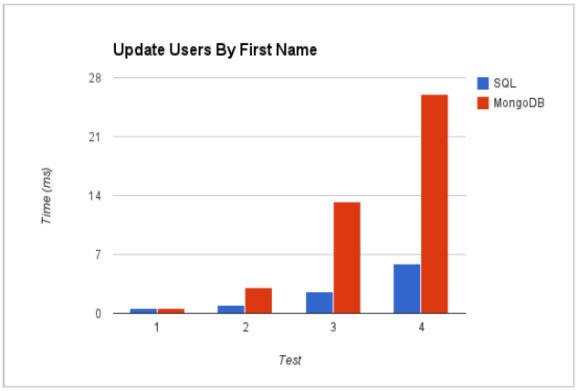


Figure 2: Update using a non-key field

Figs. 3 and 4 show the fast response of MongoDB to simple queries, but the superior performance of SQL on complex queries including aggregate functions. One reason for the good performance of MongoDB in the simple case is that it holds the whole dataset in memory whereas SQL will almost always have to retrieve data from disk. These experiments were conducted on small datasets which could fit easily into memory; to get a complete picture of the relative performance of the two platforms, similar tests would need to be done on datasets too large to fit into memory. In that situation MongoDB would also need to fetch data from disk and the performance might be more similar (Honours project, anyone?).

An obvious explanation for the good performance of SQL on aggregate functions is that relational database are designed for this type of operation while NoSQL databases are not. MongoDB includes the MapReduce operation, but it is slow in comparison to the optimised architecture of a relational database, and typically aggregate calculations would need to be done in application code.

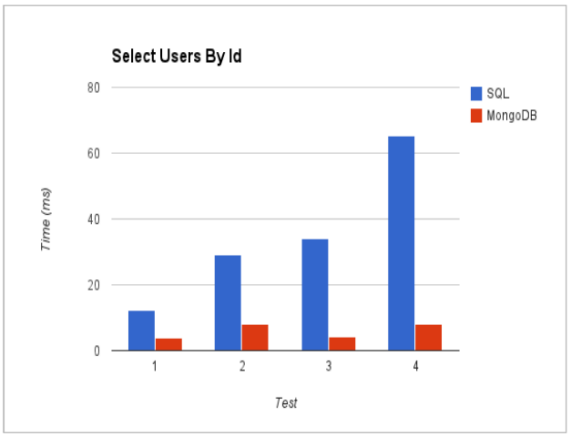


Figure 3: Simple select using a key field

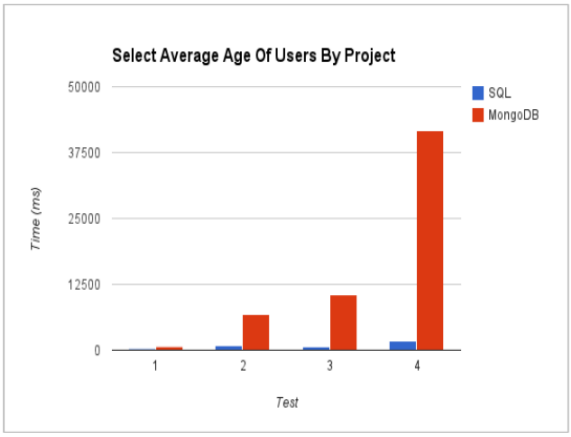


Figure 4: Complex select including an aggregate function

Another point to remember when looking at results like this is that it is dangerous to draw general conculsions from one set of tests. The test conditions - including choice of database platforms, nature of the datasets, choice of operations, etc. - all have an effect on the results. It is not possible to say from these results that NoSQL database perform better than relational databases in generqal; it is not even possible to say that MongoDB performs better than SQL Server. A different set of tests would almost certainly show other disparities between the two. All we can say is that under certain circumstances, one platform will outperform the other and that care needs to be taken in the early stages of a development project to select the most appropriate database for the application.

MongoDB

MongoDB: an example document store

The name comes from a contraction of humongous, an informal word meaning *very large*. The reason for this is that MongoDB is primarily designed for handling very large sets of semi-structured data, typically referred to as Big Data. In addition, MongoDB is mainly intended for use in building Web applications and seeks to take advantage of existing database knowledge. It therefore uses a syntax familiar to javascript developers and concepts from the field of relational databases.

Like other NoSQL models, MongoDB relies heavily on a single key field to retrieve data efficiently. The MongoDB `_id` field thus acts very much like a primary key in a relational database but it is associated with a set of data values which are much less regular in structure than the tables in a relational database.

In a MongoDB database, data is stored as a collection of *objects* which are described using Javascript Object Notation (JSON). The code snippet below shows an example of a JSON object which represents a student.

```
{ "_id" : 1,
   "matric" : 40000001,
   "name" : { "first" : "John", "last" : "Johnson" },
   "programme" : { "qualification" : "BEng", "title" : "Computing" },
   "modules" : [ {"code": "SET07102", "title": "Software Development 1", "Result": 75 },
                 {"code": "CSN07101", "title": "Computer Systems 1", "Result": 66 },
                 {"code": "SET08108", "title": "Software Development 2", "Result": "NULL"},
                 {"code": "INF08104", "title": "Database Systems", "Result": "NULL"}
               ]
}
```

The example illustrates several of the important features of JSON:

- An object is enclosed in braces: { ... }
- An object is composed of elements which are key-value pairs separated by a colon - eg. line 2
- Elements are separated by commas
- Objects can be nested - eg. line 3
- A value can be an array enclosed by square brackets: [...] - eg. lines 5 - 9

The example also illustrates the use of the MongoDB _id field (line 1) and the use of null values (lines 7 and 8).

In MongoDB, a collection of similar objects is the equivalent of a relational table. Unlike a table, however, objects in a collection do not necessarily have exactly the same structure, and there is no need to define the collection in advance of storing an object. This means there is essentially no independently defined schema.

Although MongoDB actually uses an extension of the JSON notation (Binary JSON, or BSON), the syntax makes it extremely easy to perform database operations in a javascript program. Operations are specified for a collection of objects in the current database using standard javascript commands. The code snippet below illustrates the process of inserting a new object into the **STUDENTS** collection, querying it, updating it and finally deleting it.

```
1  db.students.insert( {
2     matric: 40000001,
3     name: {first: "John", last: "Johnson"},
4     programme: {qualification: "BEng", title: "Computing"} );
5
6  db.students.find( {matric: 40000001} );
7
8  db.students.update( {matric: 40000001},
9     {$set: programme.qualification: "BSc"} );
10
11 db.students.remove( {matric: 40000001} );
```

The correspondences between the MongoDB syntax and SQL are easy to see. For example, the query (line 6), update (lines 8-9) and remove statements (line 11) all contain the string `{matric: 40000001}` which performs the same function as the criteria in an SQL WHERE clause. Notice the use of a special operator `$set` in the update command to indicate which fields should be changed.

What is not so obvious is that for every new object, MongoDB will add an `_id` field and populate it with a value which is unique across all objects in the collection. In addition, if the collection does not already exist then just inserting a new object will create it. Further objects with different structure (ie different key - value pairs) can be added to the same collection. In this way, MongoDB provides what it calls a flexible schema. This makes creating data very straightforward, but provides very loose control over the structure of the data.

Complex operations

Although missing in the early releases, MongoDB now provides support for several complex operations that were taken for granted in relational databases. For example, aggregation operations (group functions) are supported in the current version. Again, looking at an example, it is possible to see the correspondence with the SQL approach:

```
1  db.students.aggregate( {
2     $match: { $and: [ {programme.qualification: "BEng"},
3                       {programme.title: "Computing" } ] },
4     $group: { _id: "$matric",
5               average: { $avg: modules.result } }
6  } )
```

In the example, line 2 corresponds to the WHERE clause in an SQL statement, and line 4 covers the job of the GROUP BY clause and the aggregate function. In effect, this statement creates a new set of temporary documents with the fields `_id` and `average` corresponding to the students on the BEng Computing programme.

Indexing

Another concept imported into MpngoDB from traditional database models is that of the index which works in more or less the same way. An index is a copy of some subset of the data arranged into a structure that supports efficient searching with links back to the original documents. An index is automatically created for the `_id` of a collection just like the primary key of a relational table is automatically indexed. Database adminstrators can also define additional indexes on single or multiple fields in a collection. Indexes may be *sparse* in that they only include values for documents which contain the field in question. The implementation of indexes in MongoDB therefore maintains the support for flexible schemas.

One of the advantages of MongoDB indexes is that they are meant to reside in RAM which makes queries run extremely quickly. However, this is also a limitation since in a large database it may not be possible to hold all indexes in RAM. If the index size exceeds the available RAM, some idexes will be destroyed and will have to be rebuilt when needed. This leads to a significant performance reduction. Careful management of the number of indexes and the total index size is therefore a major concern for MongoDB adminsitrators.

A cautionary tale

If you just read the marketing material from a NoSQL database provider like MongoDB, or you only focus on the theoretical advantages of the model, it will seem as if relational databases are dead and the sooner everyone migrates to a NoSQL model the better. However, it is important not to jump to conclusions. Often what seems like a good idea from a theoretical point of view does not prove to be so good in practice. The blog post by Sara Mei from 2013 (see the links panel) provides an excellent insight into the problems of using MongoDB in what seemed like the ideal development context and highlights some of the big advantages of the relational model. The main messages that you should come away with are that no technical decision is black and white, careful thought needs to go into the choice between technological options, and there is no substitute for experience.

References

Chang, F. et al. (2008) Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems, 26(2). Available online at http://dx.doi.org/10.1145/1365815.1365816

DeCandia, G. et al (2007) Dynamo: Amazon's highly available key-value store. *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles*, pp. 205-220. Available online at http://dx.doi.org/10.1145/1323293.1294281

Gilbert, S. and Lynch, N. (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2). Available online at http://dx.doi.org/10.1145/564585.564601

Mei, S. (2013) Why You Should Never Use MongoDB. Available online at http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/

Moniruzzaman, A.B.M and Hossain, S.A. (2013) NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application* 6(4). Available online at http://arxiv.org/abs/1307.0191v1

Parker, Z., Poe, S., Vrbsky, S.V. (2013) Comparing NoSQL MongoDB to an SQL DB. Proceedings of the 51st ACM Southeast Conference. Available online at http://dx.doi.org/10.1145/2498328.2500047

Pritchett, D. (2008) BASE: An Acid Alternative. Queue - Object-Relational Mapping, 6(3). Available online at http://dx.doi.org/10.1145/1394127.1394128