Highly Scalable, Ultra-Fast and Lots of Choices

A Pattern Approach to NoSQL

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

November 07, 2012

Introduction

The main motivation to evaluate a non-relational database system often comes from non-functional requirements on performance and scalability that your favorite relational database system cannot support, at least not at reasonable cost.

NoSQL products promise to scale structured data storage beyond the limits of relational database systems. While this has already proven true for quite a number of applications, the ability to serve a huge number of concurrent users and great volumes of data comes at a price.

First of all, the term "NoSQL" encompasses a variety of approaches to store and retrieve data. Coming from the world of relational database systems, you will quickly realize that standards such as SQL do not exist (at least not yet). In fact, many NoSQL products have little in common other than that they are different from relational databases.

Fortunately, there are some commonalities among groups of NoSQL databases. The goal of this paper is to clearly distinguish the different types of NoSQL data stores. In reality, there are products that combine multiple approaches. For an introduction into the world of non-relational database systems, this paper emphasizes the individual advantages and drawbacks of every approach.

This paper does not go into details to explain advanced features. Neither does it contain an overview of the currently available NoSQL products since these products will change quickly. This paper rather aims at general insights into NoSQL data stores, which will hopefully remain valid for a longer period.

The data store types are described as patterns. The pattern approach has been taken to put emphasis on the question why you would choose a specific data store type. In other words, you will get to know what problem you should have so that a specific data store type is an appropriate solution.

If you want to quickly skim over the paper, just read the highlighted paragraphs. The paragraphs in *italic* explain the context in which the patterns can be applied. The paragraphs in **bold** face contain the main problem and solution statements. The paper ends with a couple of things to be aware of when introducing a NoSQL database for the first time and some final remarks.

The most basic concept of non-relational data storage is the KEY/VALUE STORE. Such a database stores little-structured data that is accessible by their keys only. Several variants exist that form groups of their own: BLOB STORES are specialized to hold large binary data. COLUMN FAMILY STORES persist values of the same type closely together for better analyzing capabilities. DOCUMENT STORES provide means to easily create rich domain models. GRAPH DATABASES rely on a different concept, focusing on the relations between data entries.

Key/Value Store

Your domain model or an important part of it is rather simple, i.e. there are few interdependencies and constraints. But the application's data changes often or your application generates a huge amount of it.

There are many cases when you need to store data that is not critical to the application, but you've got plenty of it. Think about statistical data, status information of an online game or similar scenarios where small chunks of data are written often and read more or less often. Using a relational database in such cases does not scale well because of transactional overhead or database locks.

Speed is your main concern when reading and writing data. Your application must react quickly even at peak time. The ability to scale well is more important than data integrity and the ability to comfortably query data.

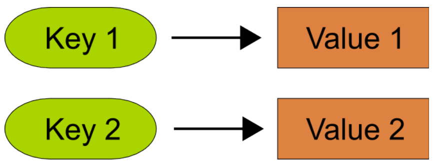
Typical features of relational database systems (such as transactional support and referential integrity) are not important to you. At least, they are less important than throughput. But your data still needs to be stored safely and high availability is a key issue.

Your domain model is not highly complex and your application's data has little structure, but you need at least support for some data types.

Your application's data might change rapidly, triggered by many simultaneous users, but read operations also need to execute quickly.

Therefore:

Choose a Key/Value Store, which stores little-structured data associated with unique keys.



The general idea of a Key/Value Store is to dispose of the concept of highly structured tables, columns and values in favor of a very simple programing interface: You associate simple data values such as numbers or texts with unique keys, and the database very efficiently stores and retrieves your data. Some Key/Value Stores provide operations to manipulate structures such as lists and sets whereas other Key/Value Stores only support basic or no explicit data types.

This simplicity provides the basis for supreme scalability. Most Key/Value Stores keep all data in memory all the time to achieve high performance. Their strategies to write data to disk and to distribute data within a cluster differ significantly. Keeping all data in memory limits the amount of data that every node can hold. Unless you are able to set up large clusters, this may also limit the maximum size of the stored data.

Key/Value Stores typically transfer the responsibility for the creation of unique keys to the application. It can be difficult to find a good strategy to choose appropriate keys (i.e. both meaningful and unique).

If you need to search for values, Key/Value Stores typically provide little comfort. The programming interface to a Key/Value Store is much more limited than what you might be used to from object-relational mapping tools, for example.

There are several variants of Key/Value Stores that focus on specific application development use cases and provide more sophisticated functionality. If you need to store large binary data, consider using a Blob Store. If you need to analyze large amounts of data at the same time as it is gathered, a Column Family Store might be a better option. If your application consists of a rich domain model, a Document Store might be appropriate.

Examples of Key/Value Stores are Memcached ([6]), Couchbase ([7]), Redis ([8]) and Riak ([9]).

Blob Store

Your application relies on binary data that needs to be distributed to the application's users.

Think about web sites with plenty of large images. If the content of your web site changes regularly, the images often do so, too. Statically bundling the images with your application is not an option because of their size and because every image change would cause a re-deployment.

You need to dynamically store and publish huge amounts of binary data such as images.

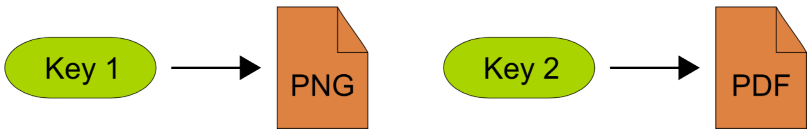
Relational databases provide the data type BLOB to store binary data in tables. But if you keep many and potentially large BLOB entries in a relational databases, you quickly drive the database to its limits. In addition, relational databases typically cannot serve their content directly to end users (and you wouldn't want them to do so). So you need to stream the BLOB contents from the database through an application server, which puts load on two critical system components.

You could store binary data in a file system and set up a web server to make these files accessible. But large numbers of files and big files push many file systems to their limits. Moreover, in case of many simultaneous requests, the underlying storage system may soon become the bottleneck. Caching might help, but web servers are not optimized for caching large numbers of large files. In addition, neither file systems nor web servers provide means to manage files, e.g. they don't create unique keys.

To overcome the caching problem, you could employ a content delivery network, which acts as a global proxy. Requests for your files are handled by the content delivery network and not by your own system. But that would add another layer of complexity, which makes the system difficult to test. And you still need to manage the files in your file system first. Additionally, you must expect significant delays in distributing your files to all nodes of the content delivery network.

Therefore:

Choose a Blob Store, which is specialized to manage a large number of potentially large files.



A Blob Store keeps the complexity of actually storing the data away from you. At the same time, a Blob Store is optimized to provide the content to your client applications, typically via http URLs.

A Blob Store associates every file with a unique key, by which the file can be accessed. Keys can be defined by your application or they can be automatically generated. Blob Stores provide little search capabilities. You need to keep track of your binary data in your main application.

Setting up and running a Blob Store is not trivial. Therefore, Blob Stores are often used as a service. Some Blob Store products (or rather vendors) physically distribute your files around the globe. You still have a single point of access to manage your data from within your application. But accessing the files becomes a (geographically) local operation from many points in the world. A Blob Store (again: as a service from a vendor) may therefore avoid the need to use a dedicated content delivery network.

On the downside, Blob Stores are like an attic into which you store everything that does not fit elsewhere. After a while, you might lose track of what is inside and what is not. Blob Stores are typically bad at categorizing your data – you've only got an identifier. And because Blob Stores are specialized at binary data only, they do not provide a means to store all your data. Rather, they add complexity to your data storage architecture.

The main difference to Key/Value Stores is that Blob Stores are made to distribute binary data to users whereas Key/Value Stores are made to handle frequent but small application data changes.

Examples of Blob Stores are Amazon S3 ([10]) and the Blob service of Windows Azure ([11]).

Column Family Store

Your application collects masses of little-structured data that is read more often than written.

Your application may evaluate metrics about user operations or generate reports on business data. This kind of functionality typically includes analytical operations such as calculating aggregates and minimum or maximum values.

Your application needs quick response times when analyzing masses of data. After some data has been collected, the data must show quickly in the analysis.

The traditional solution to analyze data is to set up a data warehouse. The data is stored in a normalized schema in one relational database. For the analysis, it is transformed into a denormalized star schema and transferred into a separate database. Queries on a data warehouse are fast and don't harm the main database's performance. But extracting and transferring data from one database to another is complicated, inefficient and error-prone. The transfer itself is time-consuming so that it may take a long time until new or updated data shows up.

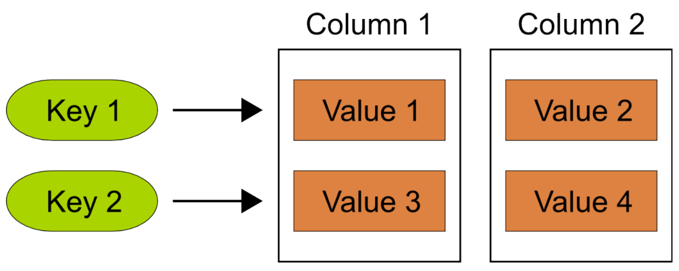
Without a data warehouse, you would need to analyze the data in the main database. But calculating aggregates in a relational database leads to full table scans. Such operations are not only slow themselves but also harm the performance of parallel requests, in particular write requests.

Apart from that, you could pre-calculate the requested data and store the results separately. But refreshing the calculations is costly, in particular if masses of data are involved, and it might take a while until new data shows up.

In addition, you might need to change the set of collected data regularly. But changing the schema of a relational database can lock a relational database for hours if a table is already very large.

Therefore:

Choose a Column Family Store, which can store masses of structured data by partitioning the data at a column-level. All read operations and calculations on individual columns are very fast. Adding and removing columns are cheap operations.



A Column Family Store keeps data in columns rather than rows. The values of a column are kept in a contiguous space on the physical storage. The database can quickly retrieve all values of a column and perform operations on them. The less the values in a column vary, the better the content of a column can be compressed. The compression alone can lead to an improvement in an order of magnitude to row-based data stores.

New columns can easily be added and existing columns can be easily deleted even if the database already holds large amounts of data. On the downside, the values of a data set are spread among several columns. Inserting, updating or reading the complete data set can be more time-consuming than in a relational database as the data columns may even be distributed even among multiple machines.

A Column Family Store can be seen as a Key/Value Store where data is stored in a format ready to be consumed and analyzed very efficiently.

Examples of Column Family Stores are HBase ([12]) and Cassandra ([13]).

Document Store

Your application has a rich domain model and does not collect huge masses of data.

When data is stored in a relational database, a normalized entity-relationship schema is the typical way to go. A normalized data structure has many merits, e.g. it guarantees the consistency of your business data. The more complex the domain model of an applications becomes, the more tables are needed to store the data.

Given a rich and complex domain model, you need to efficiently search for and modify your domain entities. But mapping a complex domain model to a relational structure often leads to slow performance.

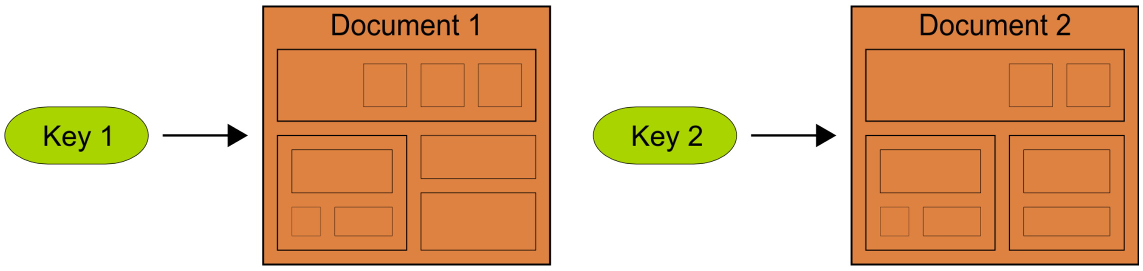
When a complex domain model is mapped to a normalized data structure in a relational database, you typically get a large number of tables and foreign key constraints in between. Querying such a data structure causes many join operations. But with concurrent read and write access, too many database joins quickly demolish the overall retrieval performance. Still, you want to keep your rich domain model instead of relying on a simple domain model as provided by a KEY/VALUE STORE.

The ability to query data by searching for occurrences of business values is important to your application but following relations among entities that are stored in separate tables of a relational database is an expensive operation.

An expressive, preferably object-oriented domain model improves the understandability of the business domain but mapping such a model on a fine-granular data structure may cause performance degradation. The results are often a large number of round-trips to the data store and therefore inefficient load operations.

Therefore:

Choose a DOCUMENT STORE, which manages hierarchically structured data records. Modifying and retrieving such records are a cheap operations.



A DOCUMENT STORE keeps all information related to a single entity in one document. The complete data of a document can be stored and retrieved atomically. Whereas in a relational database, you need to create and fill multiple tables to model one-to-many relationships, you can easily store such data in single documents.

Typically, data records in DOCUMENT STORES are saved in a JSON-like data structure. JSON, which is short for JavaScript Object Notation ([19]), can hold any kind of hierarchically structured data. Developers who have developed Rich Internet Applications are typically quite familiar with this data format.

Mapping an object-graph to a JSON notation and back is easy and fast, both done manually and by libraries. On the downside, a JSON-based structure cannot model many-to-many relationships. In addition, some details such as non-standardized data types (e.g. the format of dates) may cause problems in creating a rich data model based on JSON.

DOCUMENT STORES work best if the entities of your domain model are mostly AGGREGATES, as propagated by DOMAIN-DRIVEN DESIGN ([20]). An aggregate entity is a complex data structure to which there is only a single point of reference.

The ability to express queries for entities based on their internal data structure is an essential feature of all Document Stores. Still, not all such products provide a dynamic query language to execute search requests. Some products mandate that you develop queries outside your application code in a different programming language (mostly JavaScript) as Map-Reduce ([21]) functions.

As another downside, few products provide the capability to span queries over several data records, i.e. joining them. Also, you typically cannot easily prefetch data that is stored in multiple data records in a single call.

Document Stores can be seen as Key/Value Stores with extra capabilities to structure and query the data. In reality, there is a smooth transition between both. Some products that started as basic Key/Value Stores slowly grow to become more like Document Stores.

Examples of Document Stores are MongoDB ([14]) and CouchDB ([15]).

Graph Database

Your application needs to dynamically manage and evaluate relationships between your entities.

For example, your application keeps track of who of your users knows whom and you would like to show common friends of any two users. Or, a mobile application needs to show how a specific location can be reached via public transport from the current location, etc.

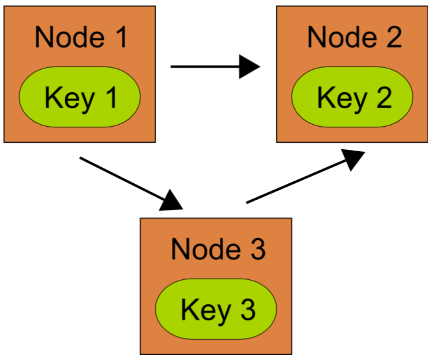
Your application focuses on relationships between entities more than on the actual entities and their properties. The application needs to traverse the relationships of a large number of entities efficiently.

Storing relationships between entities is exactly what relational databases are made for. But querying recursive or transitive relationships in large entity sets is difficult due to limitations in SQL and the way relational databases actually store entity data.

Efficient traversal algorithms are the key to a high query performance, but implementing and optimizing such algorithms for large sets of data is a difficult task.

Therefore:

Choose a Graph Database, which make it possible to explicitly model the relationships between entities as a graph and provides means to efficiently traverse the graph.



Graph Databases offer search capabilities among entity relations better than any other database type. Traversal along the edges of large graphs can be done very fast. Graph Databases typically provide a set of standard graph search algorithms to aid the development of efficient queries.

The performance of searching within large graphs is very high if the database can execute a search request in a single call. If your application needs to traverse the graph explicitly, every traversal step may lead to an inter-process round-trip with the potential to harm performance significantly.

If you need to search nodes by their properties, Graph Databases typically provide some kind of support, for example in form of indexes. Be aware, however, that Graph Databases are not made to search for sets of nodes by their properties. Their support for such kinds of searches often is very limited. For such use cases, you may need to employ additional technologies such as full text search engines on top of the Graph Database.

Every domain model that is based on a relational or a key/value structure can be transformed into a graph. In theory, it is therefore possible to migrate any domain model to a graph database and to benefit from the search capabilities of graph databases. Even if a relational database or Key/Value Store remains master of the data, you could periodically load a snapshot of the data into a Graph Database to perform graph-related search requests there (although the creation and transfer of such a snapshot may be a costly operation by itself).

It may be difficult to partition a graph of data among several servers to support clustering in case a single server does not support the amount of data or number of requests.

Examples of Graph Databases are Neo4j ([16]), Sones ([17]) and OrientDB ([18]).

A couple of things to be aware of

Although NoSQL databases are in fashion nowadays, the decision how to store an application's data is difficult. And there are more obstacles to be aware of.

A NoSQL database enforces you to learn a technology that often differs significantly from a relational database. You need to quickly build up knowledge or spend money on support or for training sessions. As most of the NoSQL product are quite new and few experienced developers exist, expect difficulties in finding appropriate assistance.

Even if the developers in your team are eager to learn new technologies, errors and wrong decisions that hurt the project's progress will happen. Do not assume that such a technological change will happen smoothly. Developers need to learn new APIs, administrators need to learn new tools and analysts need to understand the implications on domain modeling.

While you gain many powerful new features, you probably also lose features that you relied on so far. The implications of the loss of these features (such as referential integrity or transactional support) might not be fully understood at the beginning.

You may also lose the power of integration libraries that, for example, simplify the creation of an object-oriented domain layer on top of a non-object-oriented data store. The available libraries to integrate a specific product into your development environment (i.e. programming language, IDE, etc.) may lack all but the most essential support.

If you are part of a company where the development of an application is strictly separated from the operation of the application, you need to approach the operations team early. These people are responsible to keep the application running and they also need to learn and employ new tools to setup, monitor and backup the data store. They may even veto the introduction of a technology that is yet unknown to them.

You might be used to set up and manage a production environment where server applications are run on powerful, non-clustered machines. Do not underestimate the complexity of handling and managing server clusters – both in terms of software processes and hardware. For example, if you create a cluster based on commodity hardware, you have to accept and cope with failures of all core components.

Check out early how support is given for your chosen product. Many NoSQL products are supported by their developers on mailing lists or in forums. If severe problems happen (both at development time and in production), you may find out that no one feels responsible to help you solve your problem. Commercial support may not be available as you may be used to.

Final remarks

In general, you have little choice but to employ a NoSQL data store if the expected size of persistent data or the number of simultaneous requests exceed the capabilities of your relational database system. However, even if a relational database might still work for your project, NoSQL data stores provide value. So you could still decide on taking a NoSQL product; you just need to be aware of the consequences.

NoSQL databases provide a wide range of solutions when you need highly scalable data storage ([1], [2]). But there is no all-in-one solution available that promises to replace the currently dominating relational databases systems. Rather, most NoSQL databases are best suited to address specific problems and use cases ([3]).

While the data storage type of a NoSQL product is an important decision criterion, there are more criteria to consider. The CAP theorem ([22]) allows different tradeoffs to make, i.e. whether a product restricts the consistency, availability or partition tolerance of the system. Furthermore, different products support different clustering and replication strategies that may affect an application's general architecture. All of these criteria (and several more) are out of scope of this paper and deserve papers of their own.

The term *polyglot persistence* ([4], [5]) expresses the fact that a single data storage may not suit all needs of an application. Instead, an application might encompass several different approaches to data persistence. To give an example: while you would probably keep financial records in a relational database, catalog data that describes a variety of products may be better kept in a DOCUMENT STORE. Session information of the users of a web site may be best stored in a KEY/VALUE STORE. Data that tracks the users' behavior for latter analysis is a candidate to be kept in a COLUMN FAMILY STORE. Add a BLOB STORE for binary data such as images and a GRAPH DATABASE to model and analyze the relations between your users and you've got a very rich data persistence landscape.

Acknowledgments

Cyrille Martraire gave a lot of constructive and valuable feedback as shepherd for this paper previous to the conference EuroPLoP 2012. During the conference, the paper was workshopped by Veli-Pekka Eloranta, Ioanna Lytra, Christopher Preschern and Vallidevi Krishnamurthy who provided even more suggestions for improvement.

References

[1] A catalog of NoSQL databases
http://nosql-database.org/

[2] Alex Popescu: myNoSQL – A Blog on a variety of NoSQL topics,
http://nosql.mypopescu.com/

[3] Kristof Kovacs: Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs Membase vs Neo4j,
http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

[4] Scott Leberknight: Polyglot Persistence,
http://www.nearinfinity.com/blogs/scott_leberknight/polyglot_persistence.html

[5] Martin Fowler: Polyglot Persistence,
http://martinfowler.com/bliki/PolyglotPersistence.html

[6] Memcached: http://memcached.org/

[7] Couchbase: http://www.couchbase.com/membase

[8] Redis: http://redis.io/

[9] Riak: http://basho.com/products/riak-overview/

[10] S3: https://aws.amazon.com/s3/

[11] Azure: http://www.windowsazure.com/en-us/

[12] HBase: https://hbase.apache.org/

[13] Cassandra: https://cassandra.apache.org/

[14] MongoDB: http://www.mongodb.org/

[15] CouchDB: https://couchdb.apache.org/

[16] Neo4j: http://neo4j.org/

[17] Sones: http://www.sones.de/home

[18] OrientDB: http://www.orientdb.org/orient-db.htm

[19] JSON: http://json.org

[20] Eric Evans: Domain-Driven Design, Addison Wesley, 2004

[21] Map-Reduce algorithm: https://en.wikipedia.org/wiki/MapReduce

[22] CAP theorem: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem