Chapter 1

NoSQL Databases

Johannes Zollmann

1.1 Introduction

Over the last years, distributed web applications have become more and more popular. Especially widely used services like Facebook, Google or Amazon have to store and process large amounts of data. Obviously such data can not be handled by single-node systems, thus distributed storage solutions are needed. Possibilities to *scale out* a relational database management system (RDBMS), i.e. increasing the number of nodes of the system, are often very limited, or not given at all [9].

A wide range of non-relational storage solutions have evolved, in order to overcome those scalability limits. Dynamo, developed by Amazon [14], Google's BigTable [10] or Hadoop, used by Facebook [6], are examples for distributed, non-relational databases. Such database systems are subsumed under the term "NoSQL".

Here the principles of NoSQL systems and their main differences to RDBMS's are discussed. For this at first a short introduction to the relational model is given in Section 1.2. Afterwards Section 1.3 introduces basic concepts of NoSQL systems, and gives an overview of the NoSQL landscape.

Section 1.4 finally analyses the NoSQL database MongoDB.

1.2 Basics

Here some basic characteristics of traditional, SQL-based systems are analysed, in order to understand the requirements different NoSQL approaches are trying to satisfy.

1.2.1 Relational databases

In [12] Edgar F. Codd, the inventor of the relational model, identifies three basic components defining a data model:

1. *Data structure* - the data types a model is build of
2. *Operators* - the available operations to retrieve or manipulate data from this structure
3. *Integrity rules* - the general rules defining consistent states of the data model

The *structure* of a relational data model is mainly given by relations, attributes, tuples and (primary) keys. Relations are typically visualized as tables, with attributes as columns and tuples as rows. The order of the attributes and tuples is not defined by the structure, thus can be arbitrary. An example for a relational model represented by tables is given in Figure 1.1.

Basic *operations* defined by the relational model are SELECT operations (including projections and joins) to retrieve data, as well as manipulative operations like INSERT, UPDATE and DELETE.

Two different sets of *integrity rules* can be distinguished for a relational model. Constraints like uniqueness of primary keys ensure the integrity within a single relation. Additionally there are referential integrity rules between different relations.

people

id	name	age
1	Alice	20
2	Bob	25

things

id	name
1	books
2	pets

likings

person	thing
1	1
1	2
2	1

aquaintances

person	knows
1	2

Figure 1.1: Example for a relational database used to store information about people, things they like and other people they know. Since the relations between people and things, as well between people and people are many-to-many relations, two join tables have to be used.

1.2.2 ACID properties

An important concept in relational database systems are *transactions*. In [18] Jim Gray defined three properties of a transaction: *atomicity*, *consistency* and *durability*. Later Härder and Reuter abbreviated those properties - together with a fourth one: *isolation* - by the acronym ACID [20]. Even though all four ACID properties are seen as key properties of transactions on relational databases, *consistency* is particularly interesting when investigating the scalability of a system.

1.2.3 Scalability

The scalability of a system is its capability to cope with a growing workload [5]. Basically a system can be scaled in two different directions: *vertically* and *horizontally*. Vertical scaling ("scale up") means increasing the capacity of the system's nodes. This can be achieved by using more powerful hardware. In contrary, a system is scaled horizontally ("scaled out") if more nodes are added [22].

Vertical scaling of systems is typically limited by the availability and affordability of better hardware, and tends to be inefficient. In contrary, scaling systems horizontally allows for a better performance, while using cheaper hardware [22]. Though, horizontally scaling often is a non-trivial task. This is mainly because guarantees of consis-

tency, as demanded by the ACID properties, are hard to be achieved on distributed systems, which will be discussed in more detail in Section 1.3.2.

1.3 NoSQL concepts

The term "NoSQL" was first used by Carlo Strozzi to name a database management system (DBMS) he developed. This system explicitly avoided SQL as querying language, while it was still based on a relational model [26]. Nowadays the term "NoSQL" is often interpreted as "**N**ot **o**nly **SQL**" and stands for (mostly distributed) *non-relational* storage systems.

Here at first an overview of the types of NoSQL systems is given. Afterwards the consistency guarantees characteristic to NoSQL databases are discussed. Finally the MapReduce model is introduced, which provides a framework for efficiently processing huge amounts of data and is an important component of different NoSQL implementations.

1.3.1 Types of NoSQL systems

In [16, pp.6] Edlich et al. identify four classes of NoSQL systems as "Core-NoSQL" systems: *Key-Value stores*, *Wide column stores*, *Graph databases* and *Document stores*. Other NoSQL-related storage solutions, e.g. Object- or XML-databases, are called "soft-NoSQL" systems. An extensive list of known NoSQL implementations, categorized according to this terminology can be found in [15]. Here only the "Core" classes will be explained further.

Key-Value stores

Key-Value based storage systems are basically associative arrays, consisting of keys and values. Each key has to be unique to provide non-ambiguous identification of values.

While keys are mostly simple objects, values can be lists, sets or also hashes again, allowing for more complex data structures [16, p.7]. Figure 1.2 shows an example.

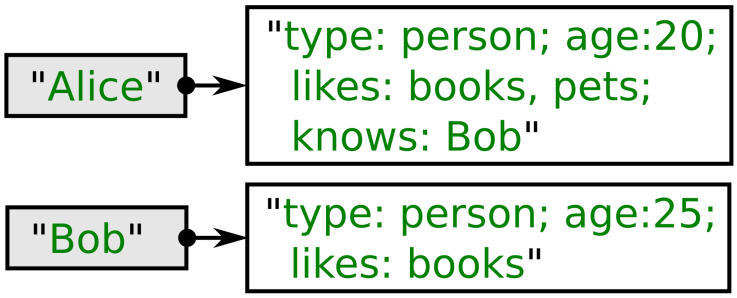

Figure 1.2: Example data represented in a Key-Value store. The stored value (here of type String) typically can not be interpreted by the storage system.

Typical operations offered by Key-Value stores are those known to programmers from Hash-Table implementations [9]:

- INSERT new Key-Value pairs
- LOOKUP value for a specified key
- DELETE key and the value associated with it

The simple model provided by a Key-Value store allows it to work very fast and efficiently. The price of this reduced complexity is a reduced flexibility of querying possibilities.

A famous representative of Key-Value stores is Amazon's Dynamo. Dynamo offers an efficient and highly scalable storage solution, at the cost of a very limited querying interface, as can be seen in [14].

Wide column stores

Storage systems of this class are also called *Extensible Record Stores* [9]. A wide column store can be seen as a Key-Value

store, with a two-dimensional key: A stored value is referenced by a *column key* and a *row key*. The key can be further extended by a *timestamp*, as is the case in Google's BigTable [10]. Depending on the implementation, there are more extensions to the key possible, mostly called "keyspaces" or "domains". Thus keys in wide column stores can have many dimensions, resulting in a *structure* similar to a multi-dimensional, associative array.

An example for storing data in a wide column system using a two-dimensional key is given in Figure 1.3.

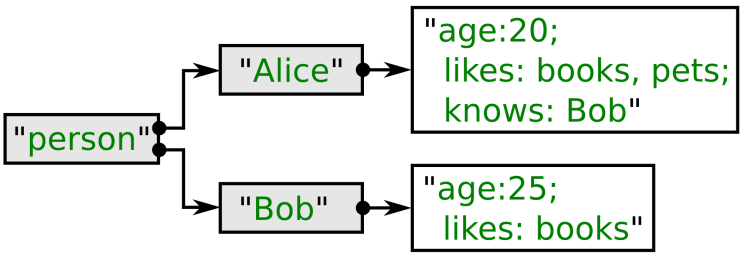

Figure 1.3: Example data represented in a wide column store. Here "person" is used as column key and each person's name as row key. Like in a Key-Value store, the stored value is not further interpreted by the system.

Graph databases

As the name indicates, in systems of this category data is represented by graphs.

Graph databases are best suited for representing data with a high, yet flexible number of interconnections, especially when information about those interconnections is at least as important as the represented data [2]. Such information can be, for example, social relations or geographic data. Figure 1.4 shows how data can be represented by a graph.

Graph databases allow for queries on the graph structure, e.g. on relations between nodes or shortest paths. Implementations of graph databases can support such queries efficiently by using well studied graph algorithms [19][11].

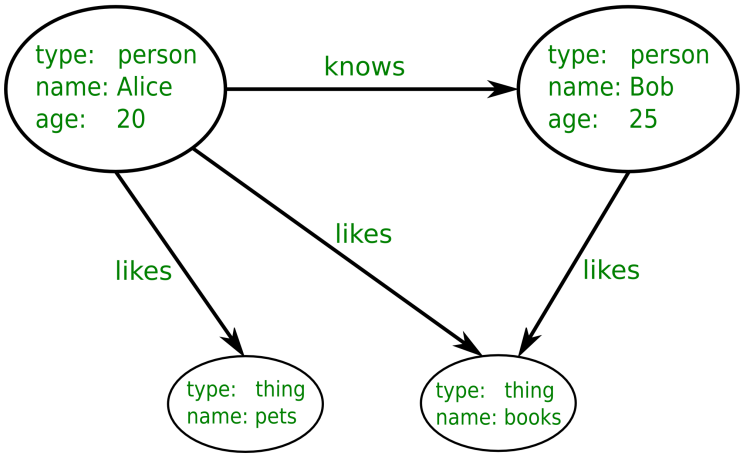

Figure 1.4: Example data represented as graph. Edges can be used to store relationship information, while other attributes of the objects have to be stored in the vertices.

Document stores

In a document store, data is stored in so-called documents. Here the term *documents* refers to arbitrary data in some structured data format. Examples for used formats are JSON [3], BSON (see Section 1.4.1) or XML [21]. While the type of data format is typically fixed by the document store, the structure is flexible. In a JSON-based document store, for example, documents with completely different sets of attributes can be stored together, as long as they are valid JSON documents. In Figure 1.5 data stored in a JSON-based document database is illustrated.

The chosen data format can be interpreted by the storage system. This allows the system to offer fine-grained *read* and *write* operations on properties of a stored document, in contrary to a Key-Value system, where the stored value typically is not further understood by the system.

Section 1.4 introduces *MongoDB*, a representative of the class of document stores. Another widely used storage system of this category is CouchDB, which extensively relies on the MapReduce framework for data querying [3].

people

```
{
  name:  "Alice",
  age:   20,
  likes: ["books", "pets"],
  knows: "Bob"
}
```

```
{
  name:  "Bob",
  age:   25,
  likes: ["books"]
}
```

Figure 1.5: Example data represented in a document store. Here JSON is used as data format. Since the format is understood by the system, direct queries on attributes (e.g. "name" or "age") are possible.

1.3.2 Eventual consistency

An important difference between relational databases and NoSQL systems are the provided consistency models. NoSQL systems have to soften the ACID guarantees given by relational transactions, in order to allow horizontal scalability. To understand the reason for this, at first three desirable properties of horizontally distributed systems are explained:

Consistency Consistency in a distributed system requires a total order on all operations throughout all nodes of the system [17]. This would, for example, be the case if after a successful *write* operation, all subsequent *read* operations return the written value, regardless on which node the operations are executed.

Availability A system satisfies *availability*, if all operations, executed on a node of the system, terminate in a response [17].

Partition tolerance A system is called partitioned, if there are at least two sets of nodes, such that all nodes of the

same set can communicate, while all messages sent between nodes of different sets are lost. An example for a partitioned system would be a system where one node gets unreachable due to a network error. A system is *partition tolerant* if it is available and consistent, even though arbitrary many internal messages get lost [17].

In 2000, Brewer stated the conjecture, that no web service can guarantee all those properties (*consistency*, *availability* and *partition tolerance*) at the same time [7]. Two years later this conjecture, referred to as CAP THEOREM or BREWER THEOREM, has been formalized and proven by Gilbert and Lynch [17].

In practice this means, distributed databases have to forfeit one of those properties. To avoid partitions, one would have to make sure that each single node of a system is always reachable by the other nodes, which can not be guaranteed in big distributed systems [27]. Thus for scalable systems the decision remains between *availability* and *consistency*.

In [25] Pritchett suggests BASE as an alternative to ACID. BASE stands for *basically available*, *soft state*, *eventually consistent* and focuses mainly on availability of a system, at the cost of loosening the consistency guarantees. The *eventually consistency* property of a BASE system accepts periods where clients might read inconsistent (i.e. out-dated) data. Though, it guarantees that those periods will eventually end. A system with BASE properties is no longer limited by the CAP THEOREM, thus offers a high horizontal scalability. An example of the length of inconsistent periods for BASE systems can be found in [4], where Amazon's S3 storage system is analysed. There the authors observe strongly fluctuating times of inconsistency, from few milliseconds up to several seconds.

1.3.3 MapReduce

MapReduce, originated by Google, is a framework, as well as a programming model, designed to process huge amounts of data using user-defined logic. The primary goal

of the original MapReduce framework was to provide an abstraction for processing data, without having to deal with the demands coming with scalability, like parallelization or load balancing [13]. Input to a MapReduce execution is a dictionary, i.e. a dataset consisting of key/value pairs. The output is again a dictionary. The user has to implement two functions, MAP and REDUCE, and provide them to the framework.

A MAP function receives a key and a value and returns a *list* of key/value pairs as *intermediate result*. The MapReduce framework invokes the given MAP function on all entries of the input dictionary. Since all those function calls are independent of each other, the framework can parallelize the processing as needed. For this the input data is split into several chunks, each handled by a separate MAP process.

Within the intermediate results, all key/value pairs are combined using the keys, such that each key maps to a list of values. Now for each key the REDUCE function is invoked. As input it receives a key, as well as the list of all values belonging to that key. The REDUCE function processes the values and combines them to a final result value, that gets associated with the key. Again all invocations of REDUCE can be executed in parallel.

The basic scheme of an execution of the MapReduce framework is sketched in Figure 1.6, while Figure 1.7 gives a simplified example.

1.4 MongoDB

MongoDB is an open source document database, initiated by the software company *10gen* [1], which also offers commercial support. As the name MongoDB - derived from the word "humongous" - indicates, its main goal is to handle huge amounts of data. While MongoDB is implemented in C++, it uses JavaScript as a querying language.

In the following sections, the data model provided by MongoDB is analysed, considering the three basic components

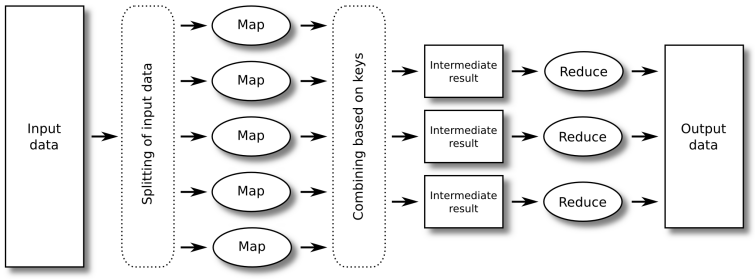

Figure 1.6: Model of the MapReduce framework. The split input data is handed to several processes running the MAP function. Return values of the map functions are combined to intermediate results. For each intermediate result, one REDUCE process is executed.

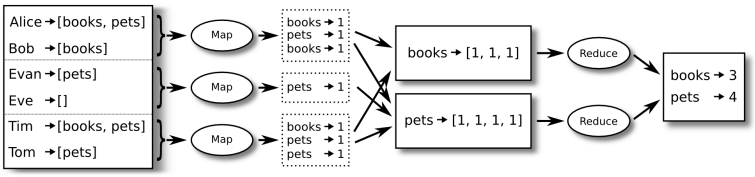

Figure 1.7: Example using MapReduce to analyse which things are liked by how many people. A list of people and there likings serves as input data. Here the input data is split into three segments, which all are processed in parallel.

of data models as defined before. Most of those sections are based on information from the current version of MongoDB's manual [23], which can be recommended for further reading.

1.4.1 Data structure

The basic building blocks used by MongoDB are collections. The relational equivalent to a collection is a relation (or table). In contrary to relations, a collection does not enforce a fixed schema, but can hold completely different documents. However, giving all documents of a collection a somewhat similar structure can allow for easier and more

efficient querying of data [16, p.133].

Documents are comparable to relational tuples and can be seen as associative arrays or hashes. MongoDB stores and transmits documents in BSON (Binary JSON) format. A BSON document is mostly a binary representation of a JSON document, extended by information for easier parsing of the data (e.g. length prefixes). Like JSON objects, BSON documents consist of attribute-value pairs, where each value can be a simple type (e.g. string or integer), again a complex BSON object, or a collection consisting of either simple types or objects. Additionally BSON extends the JSON specification by some simple types, e.g. types for dates and times. The complete BSON specification can be found in [8].

1.4.2 Operators

Reading data

One important operation to read data from a MongoDB collection is FIND. When used without additional parameters, it returns all documents from a collection. The FIND operation accepts a BSON formatted *criteria* object as parameter. MongoDB filters documents by the given criteria, returning only query results matching all specified attributes and values. Additionally, the criteria object can contain special conditional operators defined by MongoDB. Such operators range from simple comparisons (e.g. "$lt" for "lesser than") and regular expressions up to complex user-defined JavaScript functions, which are evaluated by the database. Listing 1.1 shows several examples for querying data using FIND.

Another way of querying data is using the MAPREDUCE operator of MongoDB. This operator expects a MAP and a REDUCE function, both given as JavaScript functions of a specified format. As explained in Section 1.3.3, first the MAP function is executed on all documents. Afterwards the intermediate results are processed by the REDUCE function. The return values of the REDUCE step represent the query

```
1  db.people.find({name: 'Alice'})
2  // {_id: 1234, name: 'Alice', age: 20, likes: ['
       books', 'pets'], knows: ['Bob']}
3
4  db.people.find({age: 25})
5  // {_id: 5678, name: 'Bob', age: 25, likes: ['books
       ']}
6
7  db.people.find({age: {$lt: 30}})
8  // [{_id: 1234, name: 'Alice', ... }, {_id: 5678,
       name: 'Bob', ... }]
9
10 db.people.find({knows: {$exists: true}})
11 // {_id: 1234, name: 'Alice', ... }
12
13 db.people.find({likes: {$size: 1}})
14 // {_id: 5678, name: 'Bob', ... }
```

Listing 1.1: Example queries using FIND. The queried "people" collection is assumed to have two entries: "Alice" and "Bob".

result. An example can be seen in Listing 1.2.

Manipulating data

The INSERT operation can be used to add documents to a collection. The document to be inserted is handed to the operation in BSON format. Before storing the given document, MongoDB adds an additional "*_id*" attribute, with a value unique throughout the collection. This can be seen as primary key, uniquely identifying the document.

To change existing objects of a collection, the UPDATE operation can be used. It expects again a criteria object, as well an object representing the changes to be applied. If the latter object represents a normal document, the first document matching the given criteria gets completely replaced. To only change individual attributes of matched documents, MongoDB provides additional modifier syntax. Some examples are given in Listing 1.3.

Deletion of documents is possible using the REMOVE operation. The documents to be deleted can again be addressed

```
1  map = function() {
2    for (i in this.likes) {
3      emit(this.likes[i], {count: 1});
4    }
5  };
6
7  reduce = function(key, values) {
8    var total = 0;
9    for (i in values) {
10     total += values[i].count
11   }
12   return {count: total}
13 };
14
15 db.people.mapReduce(map, reduce)
16 // [{ books: {count: 3} },
17 //  { pets: {count: 4} }]
```

Listing 1.2: MongoDB implementation of the MapReduce example from Figure 1.7 (output simplified).

by a criteria object.

1.4.3 Integrity Rules

In contrary to relational systems, offering a wide range of integrity rules, MongoDB only offers possibilities to enforce the uniqueness of documents. As already mentioned, each document has a unique "*_id*" attribute. Additional attributes, as well as combinations of attributes, can be defined as unique indexes too (see Section 1.4.4). Unique indexes ensure, that values for the specified attributes are unique throughout a collection.

Since MongoDB has no build-in support for joining different documents, it also offers no rules for referential integrity at all.

1.4.4 Indexing

Database indexes are important data structures, when it comes to optimizing *read* queries. In relational databases,

```
1  db.people.insert({name: 'Alice', age: 20})
2  db.people.update({name: 'Alice'}, {name: 'Eve'})
3  db.people.find({name: 'Eve'})
4  // {_id: 123, name: 'Eve'}
5
6  db.people.insert({name: 'Alice', age: 20})
7  db.people.update({name: 'Alice'}, {$set: {name: 'Eve
       '}})
8  db.people.find({name: 'Eve'})
9  // {_id: 456, name: 'Eve', age: 20}
10
11 db.people.insert({name: 'Alice', age: 20})
12 db.people.update({name: 'Alice'}, {$inc: {age: 5}})
13 db.people.find({name: 'Alice'})
14 // {_id: 789, name: 'Alice', age: 25}
```

Listing 1.3: Examples for manipulating documents in MongoDB using INSERT and UPDATE.

the *primary key* column is typically indexed by default. Same is true for the "*_id*" attribute of a MongoDB document. In order to allow efficient data access, MongoDB offers the possibility to create arbitrary many different indexes for a collection. Basically any attribute of a document can be indexed, as well as any combination of attributes. This extends to attributes within sub-documents (i.e. attributes of attributes).

As discussed before, the structure of a document is not fixed by the collection. Since indexes are created per collection, there may be documents within a collection, which do not have a specific attribute, even though an index for the attribute exists. MongoDB treats such documents as having the attribute with a value of *null*. For collections containing strongly varying documents, this causes a significant overhead. Therefore MongoDB offers another indexing option, called *sparse indexing*. If an index is sparse, it ignores all documents of a collection, that do not have the indexed attribute. This allows for better performance and reduces the storage overhead, since no "empty" indexes have to be created and maintained.

As known from relational systems, even though indexes make *read* operations more efficient, this comes at the cost of more expensive *write* operations. Whenever a document is created, updated or deleted, the indexes have to be up-

dated. Hence, especially for write-intensive applications, creating too many indexes might significantly reduce the systems performance.

1.4.5 Scalability

Since MongoDB does not handle any references between collections, any *read* or *write* request from a client always involves exactly one collection. So scaling applications with many smaller collections vertically can be achieved by just putting different collections on different machines.

However, vertical scaling gets more complicated when handling huge amounts of documents stored in a single collection. MongoDB offers a mechanism called "sharding" to distribute collections over multiple nodes, transparently for the application. For this the number of shards to be used, as well as a so-called *shard key* have to be specified.

A shard key is an attribute that all documents of a collection, which should be sharded, must have. The shard key is used to split the collection into multiple *chunks*. A chunk is a subset of the collection, holding all documents, which have values for the shard key within a certain range [24].

So if attribute a is defined as shard key and a collection has the n chunks $(c_i)_{i=1..n}$, then each chunk c_i contains all documents with values v for attribute a, such that $min_i \leq v < max_i$. The values for min_i and max_i , i.e. the shard key ranges for each chunk, are determined by the system, in such a way that all documents can be distributed equally across the available shards. Whenever a chunk gets bigger than a predefined threshold, the system splits it again and the chunks get redistributed, to ensure an equal load on all shards again.

The efficiency of *read* and *write* access to a sharded collection depends significantly on the choice of the shard key. Using an attribute as shard key, that has the same value for most documents, obviously prevents the chunk holding those documents from being split. Thus the collection can not be distributed equally across all shards [24]. An

attribute "home country", for example, would make a bad shard key, if all people in the collection most likely come from the same country. In that case MongoDB could not create an evenly distributed sharding.

1.4.6 Consistency

As long as for each shard there is only one server handling all *read* and *write* requests, MongoDB offers strong consistency on single entities. That means, all *reads* and *writes* on single documents are atomic and FIND operations involving only single documents are guaranteed to reflect the latest state of the database.

Additionally, FIND operations in MongoDB can specify an option to *allow reads from secondaries*. In that case the query may be answered by a replicated (i.e. slave) node, if replication is used. In this way the availability of the database can be increased. Though, this comes at the cost of consistency, as demanded by the CAP THEOREM. Since the replicated node may still have outdated information about the requested entity, here only *eventual consistency* can be guaranteed.

1.5 Summary

The relational data model, with its complex set of operators and integrity rules, offers a very flexible storage solution, that can be adopted to many problems. Additionally, relational transactions, having ACID properties, give strong guarantees on consistency. NoSQL storage solutions, in contrary, typically aim at providing simple, yet very efficient, solutions for specific problems. One goal of many NoSQL systems is to provide high scalability. For that the strong consistency guarantees, characteristic for relational systems, have to be relaxed, in favour of the system's availability. This results in the notion of eventual consistency.

From the wide landscape of NoSQL systems, MongoDB, a

document store, has been introduced in more detail. While relational systems work with fixed data models, a document store allows to store arbitrary objects together, as long as they use the same data format. Though, document stores usually do not support complex relations between stored objects. One way to overcome this would be to accept denormalized data, i.e. to store related information redundantly. While other NoSQL systems, like CouchDB, have limited possibilities for querying data and enforce the usage of MapReduce for data processing, MongoDB offers a flexible API with a JSON-like criteria syntax. Though, this still does not reach the high flexibility relational systems are providing via the SQL language.

It is obvious, that there is not *one* NoSQL system, that can be used as generic storage solution, or to replace a relational database system. Rather a suitable system to solve the respective task has to be picked. If fast data access is needed and querying via a single key is sufficient, Key-Value stores are an excellent choice. More complex key structures are possible using Wide column stores. A graph database suits best for storing highly interrelated, yet simple objects, where queries refer to the relationship structure. In contrary, querying relations in document-oriented systems can be more complicated. Though, document stores allow storing objects with complex structure in a scalable way and offer efficient mechanisms to query on that structure.

Bibliography

[1] 10gen. The mongodb company. http://www.10gen.com/, July 2012.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[3] Apache. Couchdb - a database for the web. http://couchdb.apache.org/, July 2012.

[4] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[5] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, WOSP '00, pages 195–203, New York, NY, USA, 2000. ACM.

[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[7] E. A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 2000.

[8] BSON. Bson - binary json. http://bsonspec.org, July 2012.

[9] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[11] J. Cheng, Y. Ke, and W. Ng. Efficient query processing on graph databases. *ACM Trans. Database Syst.*, 34(1):2:1–2:48, Apr. 2009.

[12] E. F. Codd. Data models in database management. *SIGMOD Rec.*, 11(2):112–114, June 1980.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[15] S. Edlich. Nosql databases. `http://nosql-database.org/`, July 2012.

[16] S. Edlich, A. Friedland, J. Hampe, and B. Brauer. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 10 2010.

[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[18] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[19] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94,

pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[20] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[21] H. Lu, J. X. Yu, G. Wang, S. Zheng, H. Jiang, G. Yu, and A. Zhou. What makes the differences: benchmarking xml database implementations. *ACM Trans. Internet Technol.*, 5(1):154–194, Feb. 2005.

[22] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

[23] MongoDB. The mongodb manual. `http://docs.mongodb.org/manual/`, July 2012.

[24] MongoDB. Sharding - mongodb. `http://www.mongodb.org/display/DOCS/Sharding`, July 2012.

[25] D. Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[26] C. Strozzi. Nosql relational database management system. `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/HomePage`, July 2012.

[27] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, Oct. 2008.

Typeset August 20, 2012