# NoSQL Evaluation

## A Use Case Oriented Survey

Robin Hecht
Chair of Applied Computer Science IV
University of Bayreuth
Bayreuth, Germany
robin.hecht@uni-bayreuth.de

Stefan Jablonski
Chair of Applied Computer Science IV
University of Bayreuth
Bayreuth, Germany
stefan.jablonski@uni-bayreuth.de

*Abstract* – **Motivated by requirements of Web 2.0 applications, a plethora of non-relational databases raised in recent years. Since it is very difficult to choose a suitable database for a specific use case, this paper evaluates the underlying techniques of NoSQL databases considering their applicability for certain requirements. These systems are compared by their data models, query possibilities, concurrency controls, partitioning and replication opportunities.**

***NoSQL, Key Value Stores, Document Stores, Column Family Stores, Graph Databases, Evaluation, Use Cases***

### I. MOTIVATION

The current NoSQL trend is motivated by applications stemming mostly of the Web 2.0 domain. Some of these applications have storage requirements which exceed capacities and possibilities of relational databases.

In the past SQL databases were used for nearly every storage problem, even if a data model did not match the relational model well. The object-relational impedance mismatch is one example, the transformation of graphs into tables another one for using a data model in a wrong way. This leads to an increasing complexity by using expensive mapping frameworks and complex algorithms. Even if a data model can easily be covered by the relational one, the huge feature set offered by SQL databases is an unneeded overhead for simple tasks like logging. The strict relational schema can be a burden for web applications like blogs, which consist of many different kinds of attributes. Text, comments, pictures, videos, source code and other information have to be stored within multiple tables. Since such web applications are very agile, underlying databases have to be flexible as well in order to support easy schema evaluation. Adding or removing a feature to a blog is not possible without system unavailability if a relational database is being used.

The increasing amount of data in the web is a problem which has to be considered by successful web pages like the ones of Facebook, Amazon and Google. Besides dealing with tera- and petabytes of data, massive read and write requests have to be responded without any noticeable latency. In order to deal with these requirements, these companies maintain clusters with thousands of commodity hardware machines. Due to their normalized data model and their full ACID support, relational databases are not suitable in this domain, because joins and locks influence performance in distributed systems negatively. In addition to high performance, high availability is a fundamental requirement of many companies. Amazon guarantees for its services availability of at least 99.9% during a year [1]. Therefore, databases must be easily replicable and have to provide an integrated failover mechanism to deal with node or datacenter failures. They also must be able to balance read requests on multiple slaves to cope with access peaks which can exceed the capacity of a single server. Since replication techniques offered by relational databases are limited and these databases are typically based on consistency instead of availability, these requirements can only be achieved with additional effort and high expertise [1].

Due to these requirements, many companies and organizations developed own storage systems, which are now classified as NoSQL databases. Since every store is specialized on the specific needs of their principals, there is no panacea on the market covering all of the above mentioned requirements. Therefore, it is very difficult to select one database out of the plethora of systems, which is the most suitable for a certain use case.

Even if NoSQL databases have already been introduced and compared in the past [2] [3] [4] [5], no use case oriented survey is available. Since single features of certain databases are changing on a weekly basis, an evaluation of the different features of certain stores is outdated at the moment it is published. Therefore, it is necessary to consider the impact of the underlying techniques on specific use cases in order to provide a durable overview. This paper highlights the most important criteria for a database selection, introduces the underlying techniques and compares a wider range of open source NoSQL databases including graph databases, too.

In order to evaluate the underlying techniques of these systems, the most important features for solving the above mentioned requirements have to be classified. Since the relational data model is considered as not suitable for certain use cases, chapter two will examine structure and flexibility of different data models offered by NoSQL systems. Afterwards, query possibilities of these stores and their impact on system complexity will be analyzed in chapter three. In order to deal with many parallel read and write requests, some stores loose concurrency restrictions. Different strategies of these systems for handling concurrent requests are inspected in chapter four. Since huge amounts of data and high performance serving exceed the capacities of single machines, partitioning methods of NoSQL databases are analyzed in chapter five. Replication

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

techniques and their effects on availability and consistency are examined in chapter six.

## II. DATA MODEL

Mostly NoSQL databases differ from relational databases in their data model. These systems are classified in this evaluation into 4 groups.

### A. Key Value Stores

Key value stores are similar to maps or dictionaries where data is addressed by a unique key. Since values are uninterpreted byte arrays, which are completely opaque to the system, keys are the only way to retrieve stored data. Values are isolated and independent from each other wherefore relationships must be handled in application logic. Due to this very simple data structure, key value stores are completely schema free. New values of any kind can be added at runtime without conflicting any other stored data and without influencing system availability. The grouping of key value pairs into collection is the only offered possibility to add some kind of structure to the data model.

Key value stores are useful for simple operations, which are based on key attributes only. In order to speed up a user specific rendered webpage, parts of this page can be calculated before and served quickly and easily out of the store by user IDs when needed. Since most key value stores hold their dataset in memory, they are oftentimes used for caching of more time intensive SQL queries. Key value stores considered in this paper are Project Voldemort [6], Redis [7] and Membase [8].

### B. Document Stores

Document Stores encapsulate key value pairs in JSON or JSON like documents. Within documents, keys have to be unique. Every document contains a special key “ID”, which is also unique within a collection of documents and therefore identifies a document explicitly. In contrast to key value stores, values are not opaque to the system and can be queried as well. Therefore, complex data structures like nested objects can be handled more conveniently. Storing data in interpretable JSON documents have the additional advantage of supporting data types, which makes document stores very developer-friendly. Similar to key value stores, document stores do not have any schema restrictions. Storing new documents containing any kind of attributes can as easily be done as adding new attributes to existing documents at runtime.

Document stores offer multi attribute lookups on records which may have complete different kinds of key value pairs. Therefore, these systems are very convenient in data integration and schema migration tasks. Most popular use cases are real time analytics, logging and the storage layer of small and flexible websites like blogs. The most prominent document stores are CouchDB [9], MongoDB [10] and Riak [11]. Riak offers links, which can be used to model relationships between documents.

### C. Column Family Stores

Column Family Stores are also known as column oriented stores, extensible record stores and wide columnar stores. All stores are inspired by Googles Bigtable [12], which is a “distributed storage system for managing structured data that is designed to scale to a very large size” [12]. Bigtable is used in many Google projects varying in requirements of high throughput and latency-sensitive data serving. The data model is described as ”sparse, distributed, persistent multidimensional sorted map” [12]. In this map, an arbitrary number of key value pairs can be stored within rows. Since values cannot be interpreted by the system, relationships between datasets and any other data types than strings are not supported natively. Similar to key value stores, these additional features have to be implemented in the application logic. Multiple versions of a value are stored in a chronological order to support versioning on the one hand and achieving better performance and consistency on the other one (chapter four). Columns can be grouped to column families, which is especially important for data organization and partitioning (chapter five). Columns and rows can be added very flexibly at runtime but column families have to be predefined oftentimes, which leads to less flexibility than key value stores and document stores offer. HBase [13] and Hypertable [14] are open source implementations of Bigtable, whereas Cassandra [15] differs from the data model described above, since another dimension called super columns is added. These super columns can contain multiple columns and can be stored within column families. Therefore Cassandra is more suitable to handle complex and expressive data structures.

Due to their tablet format, column family stores have a similar graphical representation compared to relational databases. The main difference lies in their handling of null values. Considering a use case with many different kinds of attributes, relational databases would store a null value in each column a dataset has no value for. In contrast, column family stores only store a key value pair in one row, if a dataset needs it. This is what Google calls “sparse” and which makes column family stores very efficient in domains with huge amounts of data with varying numbers of attributes. The convenient storage of heterogeneous datasets is also handled by document stores very well. Supporting nested data structures, links and data types JSON documents can be more expressive than the data model of column families. Additionally, document stores are very easy to maintain and are therefore suitable for flexible web applications. The data model of column family stores is more suitable for applications dealing with huge amounts of data stored on very large clusters, because the data model can be partitioned very efficiently.

Key value stores, document stores and column family stores have in common, that they do store denormalized data in order to gain advantages in distribution. Besides a few exceptions like nested documents and the link feature of Riak, relationships must be handled completely in the application logic. The famous friend-of-a-friend problem and similar recommendation queries would lead to many queries, performance penalties and complex code in the application layer if one of these databases was used for such use cases.

### D. Graph Databases

In contrast to relational databases and the already introduced key oriented NoSQL databases, graph databases are specialized on efficient management of heavily linked data.

Therefore, applications based on data with many relationships are more suited for graph databases, since cost intensive operations like recursive joins can be replaced by efficient traversals.

Neo4j [16] and GraphDB [17] are based on directed and multi relational property graphs. Nodes and edges consist of objects with embedded key value pairs. The range of keys and values can be defined in a schema, whereby the expression of more complex constraints can be described easily. Therefore it is possible to define that a specific edge is only applicable between a certain types of nodes.

Property graphs are distinct from resource description framework stores like Sesame [18] and Bigdata [19] which are specialized on querying and analyzing subject-predicate-object statements. Since the whole set of triples can be represented as directed multi relational graph, RDF frameworks are considered as a special form of graph databases in this paper, too. In contrast to property graphs, these RDF graphs do not offer the possibility of adding additional key value pairs to edges and nodes. On the other handy, by use of RDF schema and the web ontology language it is possible to define a more complex and more expressive schema, than property graph databases do.

Twitter stores many relationships between people in order to provide their tweet following service. These one-way relationships are handled within their own graph database FlockDB [20] which is optimized for very large adjacency lists, fast reads and writes.

Use cases for graph databases are location based services, knowledge representation and path finding problems raised in navigation systems, recommendation systems and all other use cases which involve complex relationships. Property graph databases are more suitable for large relationships over many nodes, whereas RDF is used for certain details in a graph. FlockDB is suitable for handling simple 1-hop-neighbor relationships with huge scaling requirements.

## III. QUERY POSSIBILITIES

Since data models are tightly coupled with query possibilities, the analysis of queries that should be supported by the database is a very important process in order to find a suitable data model. NoSQL databases do not only differ in their provided data model, they also differ in the richness of their offered query functionalities. The current state of the NoSQL landscape can be compared to the time before Codds SQL introduction. Similar to that time a lot of technical heterogeneous databases raised in the last years which differ in their offered data model, query languages and APIs. Therefore, some research is done in order to achieve a wider adoption of NoSQL systems by developing standardized query languages [21] [22] for some of these stores. Up to know developers have still to cope with the specific characteristics of each NoSQL store.

Due to their simple data model, APIs of key value stores provide key based put, get and delete operations only. Any query language would be an unnecessary overhead for these stores. If additional query functionalities are required, they have to be implemented on the application layer which can quickly lead to much more system complexity and performance penalties. Therefore, key value stores should not be used, if more complex queries or queries on values are required. Very useful in the domain of web applications are REST interfaces. Heterogeneous clients can directly interact with the store in a uniform way, while requests can be load balanced and results can be cached by proxies. Membase is the only key value store, which offers a REST API natively.

In contrast to key value stores, document stores offer much richer APIs. Range queries on values, secondary indexes, querying nested documents and operations like “and”, “or”, “between” are features, which can be used conveniently. Riak and MongoDB queries can be extended with regular expression. While MongoDB supports additional operations like count and distinct, Riak offers functionalities to traverse links between documents easily. REST interfaces are supported by document stores, as well. Due to their powerful interfaces, a query language increasing user-friendliness by offering an additional abstraction layer would be useful for document stores. Since none of these stores offer any query language, the UnQL project [22] is working on a common query language offering a SQL-like syntax for querying JSON document stores.

Column family stores only provide range queries and some operations like “in”, “and/or” and regular expression, if they are applied on row keys or indexed values. Even if every column family store offers a SQL like query language in order to provide a more convenient user interaction, only row keys and indexed values can be considered in where-clauses, as well. Since these languages are specialized on the specific features of their stores, there is no common query language for column family stores available yet.

As clusters of document and column family stores are able to store huge amounts of structured data, queries can get very inefficient if a single machine has to process the required data. Therefore, all document and column family stores provide MapReduce frameworks which enable parallelized calculations over very large datasets on multiple machines. However, MapReduce jobs are written on a very low abstraction level and lead to custom programs which are hard to maintain and reuse. Data analytics platforms like Pig [23] and Hive [24] try to bridge the gap between declarative query languages and procedural programmed MapReduce jobs by offering high level query languages which can be compiled into sequences of MapReduce jobs.

Graph databases can be queried in two different ways. Graph patter matching strategies try to find parts of the original graph, which match a defined graph pattern. Graph traversal on the other hand starts from a chosen node and traverses the graph according to a description. The traversal strategies differ in detecting one matching node as fast as possible (breadth-first) and in finding the shortest path (depth-first). Most graph databases offer REST APIs, program language specific interfaces and store specific query languages in order to access data using one of the two described strategies. In contrast to other NoSQL databases, there exist

some query languages, which are supported by more than one graph database. SPARQL [25] is a popular, declarative query language with a very simple syntax providing graph pattern matching. It is supported by most RDF stores (also Sesame and BigData) and by Neo4j. Gremlin [26] is an imperative programming language used to perform graph traversals based on XPATH. Besides support for Neo4j and GraphDB, also Sesame can be queried with Gremlin after a graph transformation. FlockDB do not offer any query language, since only simple 1-hop-neighbor relationships are supported.

NoSQL databases differ strongly in their offered query functionalities. Besides considering the supported data model and how it influences queries on specific attributes, it is necessary to have a closer look on the offered interfaces in order to find a suitable database for a specific use case. If a simple, language unspecific API is required, REST interfaces can be a suitable solution especially for web applications, whereas performance critical queries should be exchanged over language specific APIs which are available for nearly every common programming language like Java. Query languages offering a higher abstraction level in order to reduce complexity. Therefore, their use is very helpful when more complicated queries should be handled. If calculation intensive queries over large datasets are required, MapReduce frameworks should be used.

TABLE I. QUERY POSSIBILITIES

| Database | | Query Possibilities | | | |
|---|---|---|---|---|---|
| | | *REST API* | *JAVA API* | *Query Language* | *Map Reduce Support* |
| Key Value Stores | *Voldemort* | - | + | - | - |
| | *Redis* | - | + | - | - |
| | *Membase* | + | + | - | - |
| Document Stores | *Riak* | + | + | - | + |
| | *MongoDB* | + | + | - | + |
| | *CouchDB* | + | + | - | + |
| Column Family Stores | *Cassandra* | - | + | + | + |
| | *HBase* | + | + | + | + |
| | *Hypertable* | - | + | + | + |
| Graph Databases | *Sesame* | + | + | + | - |
| | *BigData* | + | + | + | - |
| | *Neo4J* | + | + | + | - |
| | *GraphDB* | + | + | + | - |
| | *FlockDB* | - | + | - | - |

## IV. CONCURRENCY CONTROL

If many users have access to a data source in parallel, strategies for avoiding inconsistency based on conflicting operations are necessary. Traditional databases use pessimistic consistency strategies with exclusive access on a dataset. These strategies are suitable, if costs for locking are low and datasets are not blocked for a long time. Since locks are very expensive in database clusters which are distributed over large distances and many web applications have to support very high read request rates, pessimistic consistency strategies can cause massive performance loss.

Multiversion concurrency control (MVCC) relaxes strict consistency in favor of performance. Concurrent access is not managed with locks but by organization of many unmodifiable chronological ordered versions. Since datasets are not reserved for exclusive access, read requests can be handled by offering the latest version of a value, while a concurrent process accomplishes write operations on the same dataset in parallel. In order to cope with two or more conflicting write operations, every process stores, additional to the new value, a link to the version the process read before. Therefore, algorithms on database or client side have the possibility to resolve conflicting values by different strategies. HBase, Hyptertable, Bigdata and GraphDB use the storage of different versions not only for conflict resolving but also for offering versioning. Besides consistency cutbacks MVCC also causes higher storage space requirements, because multiple versions of one value are stored in parallel. Additional to a garbage collector, that deletes no longer used versions, conflict resolving algorithms are needed to deal with inconsistencies. Therefore, MVCC causes higher system complexity.

TABLE II. CONCURRENCY CONTROL

| Database | | Concurrency Control | | |
|---|---|---|---|---|
| | | *Locks* | *Optimistic Locking* | *MVCC* |
| Key Value Stores | *Voldemort* | - | + | - |
| | *Redis* | - | + | - |
| | *Membase* | - | + | - |
| Document Stores | *Riak* | - | - | + |
| | *MongoDB* | - | - | - |
| | *CouchDB* | - | - | + |
| Column Family Stores | *Cassandra* | - | - | - |
| | *HBase* | + | - | - |
| | *Hypertable* | - | - | + |
| Graph Databases | *Sesame* | + | - | - |
| | *BigData* | - | - | - |
| | *Neo4J* | + | - | - |
| | *GraphDB* | - | - | + |
| | *FlockDB* | - | - | - |

In order to support transactions without reserving multiple datasets for exclusive access, optimistic locking is provided by many stores. Before changed data is committed, each transaction checks, whether another transactions made any conflicting modifications to the same datasets. In case of conflicts, the transaction will be rolled back. This concept functions well, when updates are executed rarely and chances for conflicting transactions are low. In this case, checking and rolling back is cheaper than locking datasets for exclusive access.

Locking and transactions are supported by Sesame and Neo4j. HBase is the only key based NoSQL system, which supports a limited variant of classic locks and transactions, since these techniques can only be applied on single rows. Applications which do not have to deal with many conflicting operations can choose optimistic locking in order to provide consistent multi row update and read-update-delete operations. Optimistic locking is supported by Voldemort, Redis and Membase. Applications which need to respond too many parallel read and write requests and which only have to provide a certain level of consistency should use Riak, CouchDB, Hypertable or GraphDB. If last-update-wins-strategies are sufficient, MongoDB, Cassandra and FlockDB can be used in order to handle huge request rates by avoiding MVCC overhead at the same time.

## V. PARTITIONING

At the time huge amounts of data and very high read and write request rates exceed the capacity of one server, databases have to be partitioned across database clusters. Due to their normalized data model and their ACID guarantees, relational databases do not scale horizontally. Doubling the amount of relational database server, does not double the performance of the cluster. Due to that lack, big web 2.0 companies like Google, Facebook and Amazon developed their own so called web-scale databases which are designed to scale horizontally and therefore satisfy the very high requirements on performance and capacity of these companies.

NoSQL databases differ in their way they distribute data on multiple machines. Since data models of key value stores, document stores and column family stores are key oriented, the two common partition strategies are based on keys, too. The first strategy distributes datasets by the range of their keys. A routing server splits the whole keyset into blocks and allocates these blocks to different nodes. Afterwards, one node is responsible for storage and request handling of his specific key ranges. In order to find a certain key, clients have to contact the routing server for getting the partition table. This strategy has its advantages in handling range queries very efficiently, because neighbor keys are stored with high percentile on the same server. Since the routing server is responsible for load balancing, key range allocation and partition block advices, the availability of the whole cluster depends on the failure proneness of that single server. Therefore, this server is oftentimes replicated to multiple machines. Higher availability and much simpler cluster architecture can be achieved with the second distribution strategy called consistent hashing [27]. In this shared nothing architecture, there exists no single point of failure. In contrast to range based partitioning, keys are distributed by using hash functions. Since every server is responsible for a certain hash region, addresses of certain keys within the cluster can be calculated very fast. Good hash functions distribute keys intuitively even wherefore an additional load balancer is not required. Consistent hashing also scores by dynamic cluster resizing. In contrast to other approaches, addition or removal of nodes only affects a small subset of all machines in the cluster. This simple architecture leads to performance penalties on range queries caused by high network load since neighbored keys are distributed randomly across the cluster.

All aforementioned key value stores and the document stores Riak and CouchDB are based on consistent hashing, whereas MongoDB documents are partitioned by the range of their ID. In contrast to key value and document stores, column family stores can be partitioned vertically, too. Columns of the same column family are stored on the same server in order to increase attribute range query performance. Cassandra datasets are partitioned horizontally by consistent hashing, whereas the BigTable clones HBase and Hypertable use range based partitioning. Since column family datamodels can be partitioned more efficiently, these databases are more suitable for huge datasets than document stores.

In contrast to key value based NoSQL stores, where datasets can easily be partitioned, splitting a graph is not straightforward at all. Graph information is not gained by simple key lookups but by analyzing relationships between entities. On the one hand, nodes should be distributed on many servers evenly, on the other hand, heavily linked nodes should not be distributed over large distances, since traversals would cause huge performance penalty due to heavy network load. Therefore, one has to trade between these two limitations. Graph algorithms can help identifying hotspots of strongly connected nodes in the graph schema. These hotspots can be stored on one machine afterwards. Since graphs can rapidly mutate, graph partitioning is not possible without domain specific knowledge and many complex algorithms. Due to these problems, Sesame, Neo4j and GraphDB do not offer any partitioning opportunities. In contrast, FlockDB is designed for horizontal scalability. Since FlockDB does not support multi-hop graph traversal, a higher network load is no problem.

Since distributed systems increase system complexity massively, partitioning should be avoided if it is not absolutely necessary. Systems which do mostly struggle with high-read-request-rates can scale this workload more easily through replication.

TABLE III. PARTITIONING

| Database | | Partitioning | |
|---|---|---|---|
| | | *Range based* | *Consistent Hashing* |
| Key Value Stores | *Voldemort* | - | + |
| | *Redis* | - | + |
| | *Membase* | - | + |
| Document Stores | *Riak* | - | + |
| | *MongoDB* | + | - |
| | *CouchDB* | - | + |
| Column Family Stores | *Cassandra* | - | + |
| | *HBase* | + | - |
| | *Hypertable* | + | - |
| Graph Databases | *Sesame* | - | - |
| | *BigData* | - | + |
| | *Neo4J* | - | - |
| | *GraphDB* | - | - |
| | *FlockDB* | - | + |

## VI. REPLICATION AND CONSISTENCY

In addition to better read performance through load balancing, replication brings also better availability and durability, because failing nodes can be replaced by other servers. Since distributed databases should be able to cope with temporary node and network failures, Brewer noticed [28], that only full availability or full consistency can be guaranteed at one time in distributed systems. If all replicas of a master server were updated synchronously, the system would not be available until all slaves had committed a write operation. If messages got lost due to network problems, the system would not be available for a longer period of time. For platforms which rely on high availability, this solution is not suitable because even a few milliseconds of latency can have big influences on user behavior. According to Amazon, their sales sank at 1% for a delay of 100 milliseconds during a test [29]. Google noticed 25% less traffic as 30 instead of 10 search

results were presented on the start page causing 900 milliseconds instead of 400 milliseconds page generation time [30]. In order to increase availability, one has to deal with consistency cut backs. If all replicas were updated asynchronously, reads on replicas might be inconsistent for a short period of time. These systems are classified by Brewer as BASE systems. They are basically available, have a soft state and they are eventually consistent. Tweets or status updates in social networks can be inconsistent, but the system must be high available and high performant at all time. In contrast, consistency is the most important feature in finance applications. Therefore, BASE systems are not suitable for bank applications. NoSQL systems are not only full ACID or full BASE systems. Many databases offer a lot of configuration opportunities to customize consistency and availability cut backs to ones needs. These systems offer optimistic replication, since they are able to configure the level of consistency by determining the number of nodes which have to commit a write process or which have to respond with the same value on a read request.

Systems which are eventually consistent in order to increase availability are Redis, CouchDB and Neo4j. Considering that CouchDB and Neo4j also offer master-master replication, these systems are suitable for offline support needed e.g. in mobile applications. Voldemort, Riak, MongoDB, Cassandra and FlockDB offer optimistic replication, wherefore they can be used in any context. Since Membase, HBase, Hypertable, Sesame and GraphDB do not use replication for load balancing, these stores offer full consistency. BigData is the only store, which supports full consistency and replication natively.

TABLE IV. REPLICATION

| Database | | Replication | | |
|---|---|---|---|---|
| | | *Read from Replica* | *Write to Replica* | *Consistency* |
| Key Value Stores | *Voldemort* | + | - | +/- |
| | *Redis* | + | - | - |
| | *Membase* | - | - | + |
| Document Stores | *Riak* | + | - | +/- |
| | *MongoDB* | + | - | +/- |
| | *CouchDB* | + | + | - |
| Column Family Stores | *Cassandra* | + | - | +/- |
| | *HBase* | - | - | + |
| | *Hypertable* | - | - | + |
| Graph Databases | *Sesame* | - | - | + |
| | *BigData* | + | - | + |
| | *Neo4J* | + | + | - |
| | *GraphDB* | - | - | + |
| | *FlockDB* | + | + | +/- |

## VII. CONCLUSION

"Use the right tool for the job" is the propagated ideology of the NoSQL community, because every NoSQL database is specialized on certain use cases. Since there is no evaluation available which answers the question "which tool is the right tool for the job?", advantages and disadvantages of these stores were compared in this paper.

First of all, developers have to evaluate their data in order to identify a suitable data model to avoid unnecessary complexity due to transformation or mapping tasks. Queries which should be supported by the database have to be considered at the same time, because these requirements massively influence the design of the data model. Since no common query language is available, every store differs in its supported query feature set. Afterwards, developers have to trade between high performance through partitioning and load balanced replica servers, high availability supported by asynchronous replication and strict consistency. If partitioning is required, the selection of different partition strategies depends on the supported queries and cluster complexity. Beside these different requirements, also durability mechanism, community support and useful features like versioning influence the database selection. In general, key value stores should be used for very fast and simple operations, document stores offer a flexible data model with great query possibilities, column family stores are suitable for very large datasets which have to be scaled at large size, and graph databases should be used in domains, where entities are as important as the relationships between them.

## REFERENCES


[1] G. DeCandia, *et al.*, "Dynamo: amazon's highly available key-value store," in *SOSP '07 Proceedings of twenty-first ACM SIGOPS*, New York, USA, 2007, pp. 205-220.

[2] K. Orend, "Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer," Master Thesis, Technical University of Munich, Munich, 2010.

[3] R. Cattell, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record,* vol. 39, December 2010.

[4] C. Strauch, "NoSQL Databases" unpublished.

[5] M. Adam, "The NoSQL Ecosystem," in *The Architecture of Open Source Applications*, A. Brown and G. Wilson, Eds., lulu.com, 2011, pp. 185-204.

[6] "Project Voldemort." Internet: http://project-voldemort.com, [30.09.2011]

[7] "Redis." Internet: http://redis.io, [30.09.2011]

[8] "Membase." Internet: http://couchbase.org/membase, [30.09.2011]

[9] "CouchDB." Internet: http://couchdb.apache.org, [30.09.2011]

[10] "MongoDB." Internet: http://mongodb.org, [30.09.2011]

[11] "Riak." Internet: http://basho.com/Riak.html, [30.09.2011]

[12] F. Chang, *et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems,* vol. 26, pp. 1-26, 2008.

[13] "HBase." Internet: http://hbase.apache.org, [30.09.2011]

[14] "Hypertable." Internet: http://hypertable.org, [30.09.2011]

[15] "Cassandra." Internet: http://cassandra.apache.org, [30.09.2011]

[16] "Neo4j." Internet: http://neo4j.org, [30.09.2011]

[17] "GraphDB." Internet: http://www.sones.com, [30.09.2011]

[18] "Sesame." Internet: http://www.openrdf.org, [30.09.2011]

[19] "Bigdata." Internet: http://www.systap.com/bigdata.htm, [30.09.2011]

[20] "FlockDB." Internet: http://github.com/twitter/flockdb, [30.09.2011]

[21] E. Meijer and G. Bierman, "A Co-Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* vol. 54, pp. 49-58, March 2011 2011.

[22] "UnQL." Internet: http://unqlspec.org, [30.09.2011]

[23] " *Pig*." Internet: http://pig.apache.org, [30.09.2011]

[24] " Hive." Internet: http://hive.apache.org, [30.09.2011]

[25] "*SPARQL*." Internet: http://www.w3.org/TR/rdf-sparql-query, [30.09.2011]

[26] " Gremlin." Internet: http://github.com/tinkerpop/gremlin, [30.09.2011]

[27] D. Karger, *et al.*, "Web caching with consistent hashing," in *Proceedings of the eighth international conference on World Wide Web*, Elsevier North-Holland, Inc. New York, 1999, pp. 11-16.

[28] E. A. Brewer, "Towards Robust Distributed Systems," in *Proceedings of the nineteenth annual ACM Symposium on Principles of Distributed Computing (PODC)*, Portland, Oregon, 2000.

[29] G. Linden. "Make Data Useful." Internet: http://www.scribd.com/doc/-4970486/Make-Data-Useful-by-Greg-Linden-Amazoncom, [30.09.2011]

[30] M. Mayer, "In Search of... A better, faster, stronger Web," presented at the Velocity 09, San Jose, CA, 2009.