A Survey on NoSQL Databases

Santhosh Kumar Gajendran

1. Abstract

NoSQL databases have gained popularity in the recent years and have been successful in many production systems. The goal of this document is to understand the current needs that have led to the evolution of NoSQL data stores, why relational database systems were not able to meet these requirements and a brief discussion of some of the successful NoSQL data stores. We will study the common concepts underlying these data stores and how they compromise on ACID properties to achieve high scalability and availability. We also look at how the database community looks at this evolution: will it supersede the RDBMS (or) just a passing cloud?

2. Introduction

Data management systems began by automating traditional tasks like recording transactions in business, science, and commerce. These systems have evolved over the time from the manual methods through the several stages of automated data management. The idea of relational model emerged with E.F.Codd's 1970 paper [1] which made data modeling and application programming much easier than in the past. Beyond the intended benefits, the relational model was well-suited to client-server programming and have proved to be the predominant technology for storing structured data in web and business applications.

Applications also evolve with time and pose challenging demands for the data management. As stated by Jim Gray [6], the most challenging part is to understand the data and find patterns, trends, anomalies and extract the relevant information. With the advent of Web 2.0 applications, the data stores needed to scale to OLTP/OLAP-style application loads where millions of users read and update, in contrast to the traditional data stores. These data stores need to provide good horizontal scalability for the simple read/write operations distributed over many servers. The relational database systems have little capability to horizontally scale to these levels. So, this paved the way to seek alternative solutions for scenarios where relational database systems proved to be not the right choice.

3. Background

The term ”NoSQL” was first coined in 1998 by Carlo Strozzi [2] for his RDBMS, Strozzi NoSQL. However, Strozzi coined the term simply to distinguish his solution from other RDMBS solutions which utilize SQL (Strozzi's NoSQL still adheres to the relational model). He used the term NoSQL just for the reason that his database did not expose a SQL interface. Recently, the term NoSQL (meaning 'not only SQL') has come to describe a large class of databases which do not have properties of traditional relational databases and which are generally not queried with SQL (structured query language). The term revived in the recent times with big companies like Google/Amazon using their own data stores to store and process huge amounts of data as they appear in their applications and inspiring other vendors as well on these terms.

There are various reasons why people searched for alternate solutions from relational RDBMS. The rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases. Currently, the volume of data is increasing at an enormous rate and the cost associated with scaling of the relational RDBMs is also very expensive. In contrast, NoSQL data stores are designed to scale well horizontally and run on commodity hardware. Also, the 'one size fit's it all' notion does not work for the current application scenarios and it is a better to build systems based on the nature of the application and its work/data load. This can be seen in this document while discussing the various data stores.

So, it points to the fact that the needs for data storage has changed over the years. The RDBMs were designed in 1980s for large high-end machines and centralized deployments. But, today's companies use commodity hardware in a distributed environment. Also, today's data is not rigidly structured and does not require dynamic queries. Michael Stonebraker has emphasized these ideas in the 2007 paper: "The end of an architectural era" [3]:

- RDBMSs have been architected more than 25 years ago when the hardware characteristics, user requirements and database markets where different from those today. Even though there have been extensions, no system had a redesign since its inception.
- New markets and use cases have evolved since the 1970s when there was only business data processing. The user interfaces and usage model also changed over the past decades from terminals where "operators [were] inputting queries" to rich client and web applications today where interactive transactions and direct SQL interfaces are rare.

NoSQL has emerged as a solution with a "share nothing" horizontal scaling – replicating and partitioning data over many servers. This allows to support a large number of simple read/write operations in a unit time and meet the current application needs. NoSQL data stores come up with following key features [8]:

- Scale horizontally "simple operations"
    - key lookups, reads and writes of one record or a small number of records, simple selections
- Replicate/distribute data over many servers
- Simple call level interface (contrast with SQL)
- Weaker concurrency model than ACID
- Efficient use of distributed indexes and RAM
- Flexible schema (different records have different schema)

NoSQL data stores give up ACID constraints in order to achieve scalability and higher performance. In fact, updates are eventually propagated and there are limited guarantees on the consistency of reads. BASE acronym, in contrast to the ACID acronym, refers to this idea: an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known-state state eventually (eventual consistency). The BASE approach according to Brewer forfeits the ACID properties of consistency and isolation in favor of "availability, graceful degradation, and performance". For instance, in case of Twitter, it is fine that the tweets of a person are distributed to the followers eventually in order to provide high availability and scalability.

In a keynote titled "Towards Robust Distributed Systems" at ACM's PODC1 symposium in 2000 Eric Brewer [5] came up with the so called CAP-theorem. Proponents of NoSQL often cite Eric Brewer's CAP theorem, which states that a system can have only two out of three of the following properties: consistency, availability, and partition-tolerance. The NoSQL systems generally give up consistency. The CAP theorem can be summarized as follows:

- **Consistency**: how a system is in a consistent state after the execution of an operation. A Distributed system is typically considered to be consistent if after an update operation of some writer; all readers see his updates in some shared data source.

- **Availability** and especially high availability meaning that a system is designed and implemented in a way that allows it to continue operation (i. e. allowing read and write operations) if nodes in a cluster crash or some hardware or software parts are down due to upgrades.
- **Partition Tolerance** is understood as the ability of the system to continue operation in the presence of network partitions. These occur if two or more "islands" of network nodes arise which (temporarily or permanently) cannot connect to each other. Some people also understand partition tolerance as the ability of a system to cope with the dynamic addition and removal of nodes (e. g. for maintenance purposes; removed and again added nodes are considered an own network partition in this notion).

So, NoSQL has emerged as a solution for today's data store needs and has been a topic of discussion and research in the recent times.

4. Critical Reception

NoSQL has been received with mixed reactions. Some people consider it as a hype lacking to fulfill its promises. In fact, there is a notion that companies do not miss anything if they do not switch to NoSQL databases and if a relational DBMS does its job, there is no reason to replace it. Similar to such arguments, some critics look at NoSQL databases as nothing new as compared to other attempts like object databases which have been around for decades.

Michael Stonebraker reacts to this NoSQL buzz in his blog post "The "NoSQL" Discussion has Nothing to Do With SQL". Stonebraker sees two reasons for moving towards non-relational datastores—flexibility and performance. The net-net is that the single-node performance of a NoSQL, disk-based, non-ACID, multithreaded system is limited to be a modest factor faster than a well-designed stored-procedure SQL OLTP engine. In essence, ACID transactions are jettisoned for a modest performance boost, and this performance boost has nothing to do with SQL. Blinding performance depends on removing overheads in database systems (logging/locking/lathcing/buffer management). Such overhead has nothing to do with SQL, but instead revolves around traditional implementations of ACID transactions, multi-threading, and disk management. To go wildly faster, one must remove all the four sources of overhead. This is possible in either a SQL context or some other context.

So, incase of huge distributed systems, NoSQL data stores are expected to perform well to achieve scalability and availability in comparison to relational data stores.

5. Common concepts

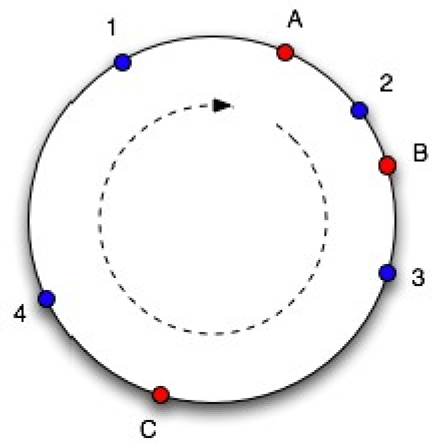
Following is a brief description of few common concepts in NoSQL databases. These concepts will be referred in this document.

Sharding

It is a partitioning mechanism in which records are stored on different servers according to some key. The data is partitioned in such a way that records, that are typically accesses/updated together, reside on the same node. The load is almost evenly distributed among the servers. Some systems also use vertical partitioning in which parts of a single record are stored on different servers.

Consistent hashing

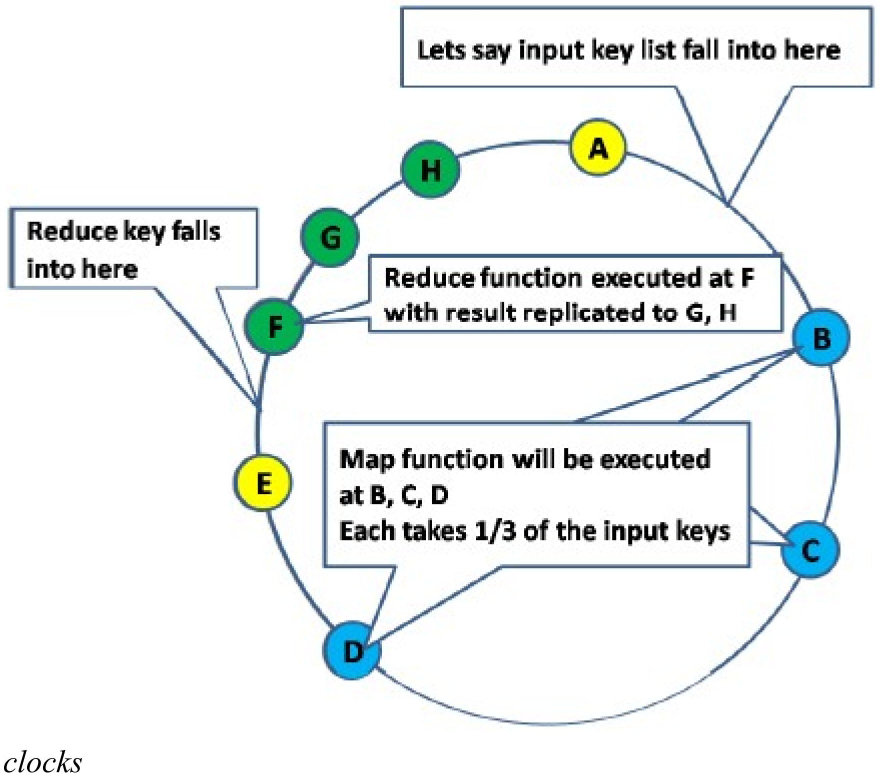
The idea behind consistent hashing is to use the same hash function for both the object hashing and the node hashing. Following figure illustrates this:



A,B,C are nodes and 1,2,3,4 are objects. Both of them are mapped to a hash range and imagined to be placed on a ring. Moving clockwise, the node following a object is mapped to that object. When a node leaves the system, cache objects will get mapped to their adjacent node (in clockwise direction) and when a node enters the system it will get hashed onto the ring and will overtake objects.

Map-reduce

MapReduce is a programming model for processing large data sets, and the name of an implementation of the model by Google. MapReduce is typically used to do distributed computing on clusters of computers. It uses a map function and a reduce function. The map function processes a key/value pair and generates a set of intermediate key/value pairs. The reduce function merges all intermediate values associated with the same intermediate key. When applied to databases, MapReduce means to process a set of keys by submitting the process logic (map- and reduce-function code) to the storage nodes which locally apply the map function to keys that should be processed and that they own. The intermediate results can be consistently hashed just as regular data and processed by the following nodes in clockwise direction, which apply the reduce function to the intermediate results and produce the final results. It should be noted that due to the consistent hashing of the intermediate results there is no coordinator needed to direct the processing nodes to find the intermediate results. This idea is illustrated in the following figure:


Lets say input key list fall into here
Reduce key falls into here
Reduce function executed at F with result replicated to G, H
Map function will be executed at B, C, D
Each takes 1/3 of the input keys
A
B
C
D
E
F
G
H

Vector clocks

If datasets are distributed among nodes, they can be read and altered on each node and no strict consistency is ensured by distributed transaction protocols, questions arise on how “concurrent” modifications and versions are processed and to which values a dataset will eventually converge to. There are several options to do this and vector clocks is one of them. A vector clock is defined as a tuple V [0], V [1], ..., V [n] of clock values from each node. In a distributed scenario node i maintains such a tuple of clock values, which represent the state of itself and the other (replica) nodes' state as it is aware about at a given time (Vi[0] for the clock value of the first node, Vi[1] for the clock value of the second node, . . . Vi[i] for itself, . . . Vi[n] for the clock value of the last node). Clock values may be real timestamps derived from a node's local clock, version/revision numbers or some other ordinal values. More details can be found at [42].

MVCC

Multiversion concurrency control (MVCC), is a concurrency control method commonly used by database management systems to provide concurrent access to the database. For instance, a database will implement updates not by deleting an old piece of data and overwriting it with a new one, but instead by marking the old data as obsolete and adding the newer version. Thus there are multiple versions stored, but only one is the latest. This allows the database to avoid overhead of filling in holes in memory or disk structures but requires (generally) the system to periodically sweep through and delete the old, obsolete data objects.

6. Classification

In recent years, a variety of NoSQL databases has been developed mainly by practitioners and web companies to fit their specific requirements regarding scalability performance, maintenance and feature-set. NoSQL is defined broadly as any database system that is not relational like Graph database systems, Object-oriented database systems etc. and there have been various approaches to classify and subsume NoSQL databases, each with different categories and subcategories.(unlike the NoSQL systems, these systems generally provide ACID transactions). This paper considers the NoSQL classification as stated by Rick Cattell in his paper on "Scalable SQL and NoSQL data stores" [8]:

- Key-value Stores: These systems store values and an index to find them, based on a programmer defined key.
- Document Stores: These systems store documents, as just defined. The documents are indexed and a simple query mechanism is provided.
- Extensible Record Stores: These systems store extensible records that can be partitioned vertically and horizontally across nodes. Some papers call these "wide column stores".

We will look at the general features of these data stores; requirements that motivated the various, recently developed NoSQL databases; their architecture and key features.

7. Key-value stores

Key-/value-stores have a simple data model in common: a map/dictionary, allowing clients to put and request values per key. Besides the data-model and the API, modern key-value stores favor high scalability over consistency and therefore most of them omit rich ad-hoc querying and analytics features. Even though key-value stores (like Berkeley DB) came into existence long time ago, they are heavily influenced by Amazon's Dynamo and many key-value stores derive significantly from Dynamo's principles. We will look at some of the key-value stores below.

Amazon's Dynamo

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Following are the key factors that Amazon has cited in their paper for the motivation behind Dyanmo:

- Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. These services requiree that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.
- Due to the use of commodity hardware, software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.
- Amazon's platform has a very diverse set of applications with different storage requirements.
- Most of the services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS.
- The available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

Dynamo, a highly available key-value store addresses the above requirements to provide an "always-on" experience. Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing, and consistency is facilitated by object versioning. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, Dynamo avoids routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

System Architecture

System Interface:
Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put().

Partitioning Algorithm:
Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing, the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Dynamo uses a variant of consistent hashing: instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node.

Replication:
To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured "per-instance". Each key, k, is assigned to a coordinator node which is in charge of the replication of the data items that fall within its range.

Data Versioning:
Dynamo is designed to be an eventually consistent system. This means that update operations return before all replica nodes have received and applied the update. Subsequent read operations therefore may return different versions from different replica nodes. The update propagation time between replicas is limited in Amazon's platform if no errors are present; under certain failure scenarios however "updates may not arrive at all replicas for an extend period of time".

Such inconsistencies need to be taken into consideration by applications. As an example, the shopping cart application never rejects add-to-cart-operations. Even when evident that the replica does not feature the latest version of a shopping cart (indicated by a vector clock delivered with update requests, see below), it applies the add-operation to its local shopping cart. As a consequence of an update operation, Dynamo always creates a new and immutable version of the updated data item. In Amazon’s production systems most of these versions subsume one another linearly and the system can determine the latest version by syntactic reconciliation. However, because of failures (like network partitions) and concurrent updates multiple, conflicting versions of the same data item may be present in the system at the same time. As the data store cannot reconcile these concurrent versions only the client application that contains knowledge about its data structures and semantics is able to resolve version conflicts and conciliate a valid version out of two or more conflicting versions (semantic reconciliation). So, client applications using Dynamo have to be aware of this and must “explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates)”. To determine conflicting versions, perform syntactic reconciliation and support client application to resolve conflicting versions Dynamo uses the concept of vector clocks.

Handling Failures:

For tolerating failures provoked by temporary unavailability storage hosts, Dynamo is not employing any strict quorum approach but a sloopy one. These approaches imply that in case of read and write operations the first N healthy nodes of a data item’s preference list are taken into account. These are not necessarily the first N nodes walking clockwise around the consistent hashing ring.

A second measure to handle temporary unavailable storage hosts are so called hinted handoffs. They come into play if a node is note accessible during a write operation of a data item it is responsible for. In this case, the write coordinator will replicate the update to a different node, usually carrying no responsibility for this data item (to ensure durability on N nodes). In this replication request, the identifier of the node the update request was originally destined to is contained as a hint. As this node recovers and is available again, it will receive the update; the node having received the update as a substitute can then delete it from its local database.

Project Voldemort

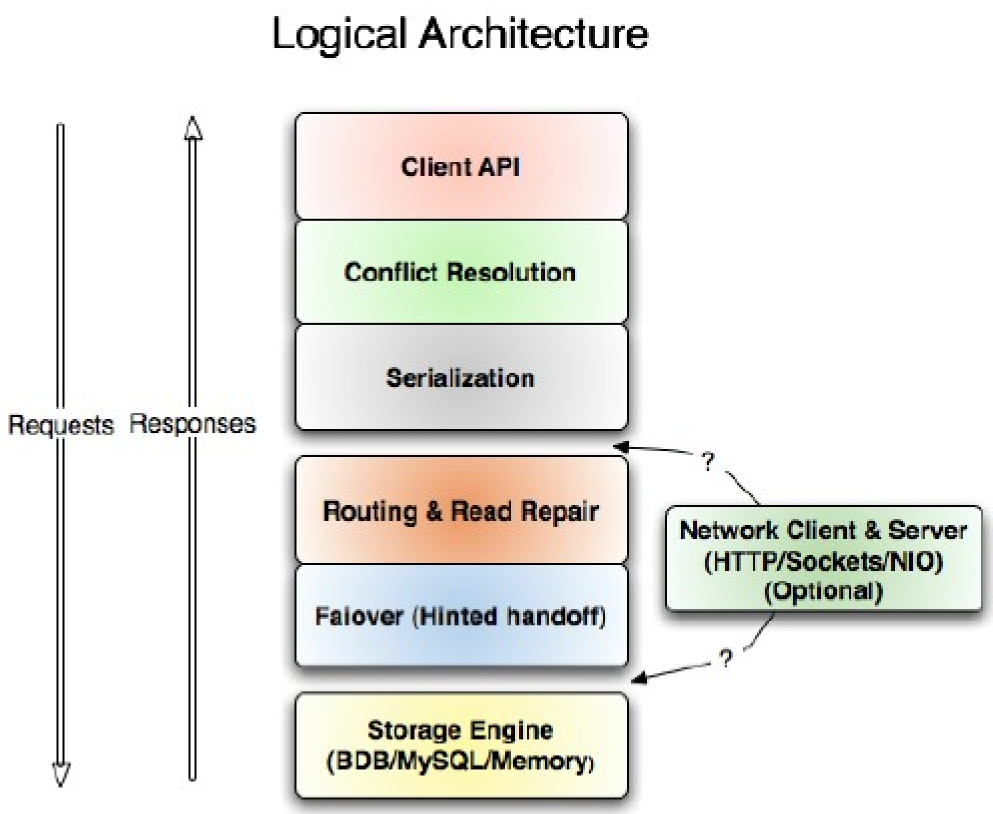
Project Voldemort is an open source implementation of the basic parts of Dynamo’s distributed key-value storage system. LinkedIn is using it in their production environment for "certain high-scalability storage problems where simple functional partitioning is not sufficient." Both, keys and values can be complex, compound objects as well consisting of lists and maps. When compared to relational databases—the simple data structure and API of a key-value store does not provide complex querying capabilities: joins have to be implemented in client applications while constraints on foreign-keys are impossible; besides, no triggers and views may be set up. However, they look at the following advantages of a simple key-value store:

- Only efficient queries are allowed.
- The performance of queries can be predicted quite well.
- Data can be easily distributed to a cluster or a collection of nodes.

- In service oriented architectures it is not uncommon to have no foreign key constraints and to do join in the application code as data is retrieved and stored in more than one service or data source.
- Gaining performance in a relational database often leads to de-normalized data structures or storing more complex objects as BLOBs or XML-documents.

- Application logic and storage can be separated nicely (in contrast to relational databases where application developers might get encouraged to mix business logic with storage operation or to implement business logic in the database as stored procedures to optimize performance).
- There is no such impedance mismatch between the object-oriented paradigm in applications and paradigm of the data store as it is present with relational databases.

System Architecture



Project Voldemort specifies a logical architecture consisting of a number of layers where each layer of the logical architecture has its own responsibility (e. g. TCP/IP network communication, serialization, version recovery, routing between nodes) and also implements an interface consisting of the operations get, put and delete.

The layered logical architecture provides certain flexibility for deployments of Project Voldemort as layers can be mixed and matched to meet the requirements of an application. For example, a compression layer may be introduced beneath to the serialization layer in order to compress all exchanged data. Likewise, intelligent routing (i. e. determining the node which manages the partition containing the requested data) can be provided transparently by the data store if the network layer is placed on top of the routing layer; if these layers are twisted, the application can do the routing itself reducing latency caused by network hops.

Project Voldemort allows namespaces for key-/value-pairs called "stores", in which keys are unique. While each key is associated with exactly one value, values are allowed to contain lists and maps as well as scalar values. Operations in Project Voldemort are atomic to exactly one key-/value-pair. Once a get operation is executed, the value is streamed from the server via a cursor. Documentation of Project Voldemort considers this approach to not work very well in combination with values consisting of large lists "which must be kept on the server and streamed lazily via a cursor"; in this case, breaking the query into sub-queries is seen as more efficient.

Like Amazon's Dynamo Project Voldemort is designed to be highly available for write operations, allows concurrent modifications of data and uses vector clocks to allow casual reasoning about different versions. If the data store itself cannot resolve version conflicts, client applications are requested for conflict resolution at read time. This read reconciliation approach is being favored over the strongly consistent but inefficient two-phase commit (2PC) approach. This is the case because it requires little coordination and provides high availability and efficiency as well as failure tolerance. On the downside, client applications have to implement conflict resolution logic that is not necessary in 2PC.

8. Document Stores

Document stores can be considered to be next step to the key-value stores because they store more complex data than the key-value stores. They store "documents" which allow values to be nested documents or lists as well as scalar values, and the attribute names are dynamically defined for each document at runtime.

Apache's CouchDB and MongoDB will be looked at in this survey. Before that, here is a quick view on SimpleDB. SimpleDB, from Amazon, is a highly available document store that provides eventual consistency. It creates and manages multiple geographically distributed replicas of the data automatically to enable high availability and data durability. SimpleDB provides simple APIs to perform operations on the documents (Select, Delete, GetAttributes, and PutAttributes). Amazon provides a simple web services interface to create and store multiple data sets, query the data easily, and return the results. SimpleDB supports more than one grouping in one database: documents are put into domains, which support multiple indexes. So domains and their metadata can be enumerated. Select operations are on one domain, and specify a conjunction of constraints on attributes, basically in the form:

```
select <attributes> from <domain> where <list of attribute value constraints>
```

Different domains may be stored on different Amazon nodes. Domain indexes are automatically updated when any document's attributes are modified. Unlike other document stores, it does not allow nested documents.

CouchDB

CouchDB, from Apache, is a "collection" of documents (similar to SimpleDB) whose data model is richer than SimpleDB. The main abstraction and data structure in CouchDB is a document. Documents consist of named fields that have a key/name and a value. The field values can be scalar (text, numeric, or boolean) or compound (a document or list). While relational databases are designed for structured and interdependent data; key-/value-stores operate on un-interpreted, isolated key-/value-pairs; document stores like CouchDB are designed for data (contained in documents) which do not correspond to a fixed schema but have some inner structure known to applications as well as the database itself. The advantages of this approach are that first there is no need for schema migrations which cause a lot of effort in the relational databases; secondly compared to key-/value-stores data can be evaluated more sophisticatedly. So, collections comprise the only schema in CouchDB, and secondary indexes must be explicitly created on fields in collections.

Queries are done through 'views' which are JavaScript functions that neither change nor save or cache the underlying documents but only present them to the requesting user or client application. Queries can be distributed in parallel over multiple nodes using a map-reduce mechanism. The map function gets a document as a parameter, does some calculation, may emit arbitrary data based on the view's criteria. The data structure emitted by the map function is a triplet consisting of the document id, a key and a value result. Documents get sorted by the key which does not have to be unique but may be present in more than one document. The value emitted by the map function is optional and may contain arbitrary data. The document id is set by CouchDB implicitly and represents the document that was given to the emitting map function as an argument. After the map function has been executed, the results get passed to an optional reduce function which can do some aggregation on the view. As all documents of the database are processed by a view's functions this can be time consuming and resource intensive for large databases. Therefore a view is not created and indexed when write operations occur but on demand (at the first request directed to it) and updated incrementally when it is requested again.

Replication

CouchDB provides asynchronous replication to achieve scalability and does not use sharding. The replication process operates incrementally where only modified data since the last replication gets transmitted to another. The whole documents are not transferred but only changed fields and attachment-blobs. CouchDB also supports partial replication where a JavaScript filter function can be defined which passes through the data for replication and rejects the rest of the database. CouchDB adopts a peer-approach for distribution where each server has the same set of responsibilities and there are no distinguished roles. Two database nodes can replicate databases (documents, document attachments, views) bilaterally if they reach each other via network. The replication process works incrementally and can detect conflicting versions in simple manner as each update of a document causes CouchDB to create a new revision of the updated document and a list of outdated revision numbers is stored. If there are version conflicts, then the participating node is aware of them and can escalate the conflicting versions to clients for conflict resolution. In case of no version conflicts, the node with the outdated version updates the document. In such a model, the reads can go to any server (not caring about the latest values) and updates must be propagated to all the servers.

ACID properties

For providing durability, all updates on documents and indexes are flushed to disk on commit, by writing to the end of a file. So, together with the MVCC mechanism it is claimed that CouchDB provides ACID semantics at the document level. The single update operations are either executed to completion or fail/rollback so that the database never contains partly saved or updated documents.

CouchDb does not guarantee consistency since each client sees a self-consistent view of the database. All replicas are always writable and they do not replicate with each other by themselves. This leads to a MVCC system in which version conflicts have to be resolved at read time by client applications: CouchDB will notify the application for any updates on the document since it was fetched by the application. The application can then try to combine the updates, or can just retry its update and overwrite.

MongoDB

MongoDB is also a document store that has many similarities to CouchDB. It is a schema-free document store that contains one or more collections consisting of documents. Like CouchDB, MongoDB also provided indexes on collections and supports map-reduce for complex aggregations across documents. But it differs from CouchDB in a number of ways as seen below.

MongoDB supports dynamic queries with automatic use of indices, like RDBMSs. MongoDB allows specifying indexes on document fields of a collection. The information gathered about these fields is stored in B-Trees and utilized by the query optimizing component to "to quickly sort through and order the documents in a collection" thereby enhancing read performance. As in relational databases, indexes accelerate select as well as update operations as documents can be found faster by the index than via a full collection scan; on the other side indexes add overhead to insert/dlete operations as the B-tree index has to be updated in addition to the collection itself. Therefore the MongoDB manual concludes that "indexes are best for collections where the number of reads is much greater than the number of writes. In CouchDB, data is indexed and searched by writing map-reduce views.

Automatic sharding

MongoDB does automatic sharding by distributing load/data across “thousands of nodes” with automatic failover and load balancing. It is inspired by Google’s BigTable. Sharding is done on a per-collection basis and not on the whole database. MongoDB automatically detects which collections grow much faster than the average so that they become eligible for sharding while the other collections may still reside on single nodes. MongoDB also detects imbalances in the load across the shards and can automatically rebalance data to reduce disproportionate load distribution. While CouchDB achieves scalability through asynchronous replication, MongoDB achieves it through sharding (however an extension of CouchDB called CouchDB Lounge supports sharding).

Replication

MongoDB uses asynchronous replication for redundancy and failover (not for a ‘dirty read’ as seen in CouchDB). In this model, only one database node (called primary node) is in charge of write operations at any instant. Read operations may go to this same server for strong consistency semantics or to any of its replica peers if eventual consistency is sufficient. It does not provide the global consistency of a traditional DBMS, but a local consistency on the up-to-date primary copy of a document. MongoDB may use 2 approaches for replication:

- master-slave replication: one server acts as a master for handling write requests and replication those operations to the other servers.
- replica sets: group of nodes work together to provide automated failover. It is an extension of the master-slave approach wherein automatic failover and recovery is added for all the member nodes.

Atomic updates

MongoDB provides atomic updates on fields by providing modifiers that update individual values and update a document only if field values match a given previous value. On the other hand, CouchDB provides MVCC on documents.

9. Extensible Record Stores

Extensible Record Stores is motivated by Google’s success with Big Table. Although most extensible record stores (like HBase, HyperTable,Cassandra) were patterned after BigTable, it appears that none of the extensible records stores come anywhere near to BigTable’s scalability at present.

The basic data model is rows and columns, and the basic scalability model is splitting both rows and columns over multiple nodes:

- Rows are split across nodes through sharding on the primary key. They typically split by range rather than a hash function. This means that queries on ranges of values do not have to go to every node.

- Columns of a table are distributed over multiple nodes by using "column groups". These may seem like a new complexity, but column groups are simply a way for the customer to indicate which columns are best stored together.

These two partitioning (horizontal and vertical) can be used simultaneously on the same table. For example, if a customer table is partitioned into three column groups (say, separating the customer name/address from financial and login information), then each of the three column groups is treated as a separate table for the purposes of sharding the rows by customer ID: the column groups for one customer may or may not be on the same server.

Big Table

Bigtable ,as described by Google, is "a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers". Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. In this course, Google has shared the following lessons learnt from experience with Big Table:

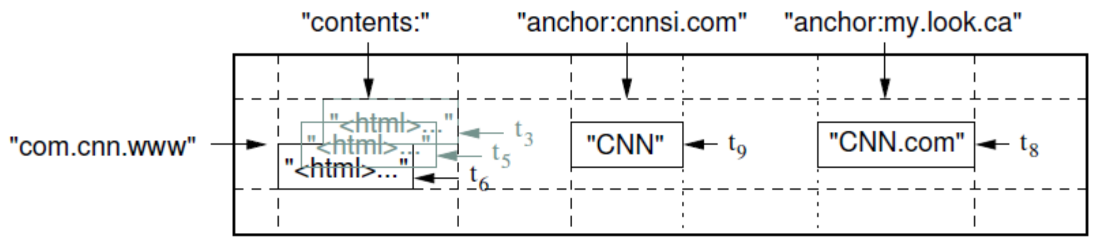
Failure Types in Distributed Systems: Large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. Hence, Google argue that such sources of failure also have to be addressed when designing and implementing distributed systems protocols.

Feature Implementation: A lesson learned at Google while developing Bigtable at Google is to implement new features into such a system only if the actual usage patterns for them are known. A counterexample are general purpose distributed transactions that were planned for Bigtable but never implemented as there was never an immediate need for them. It turned out that most applications using Bigtable only needed single-row transactions.

System-Level Monitoring: A practical suggestion is to monitor the system as well at its clients in order to detect and analyze problems.

Value Simple Designs: the most important lesson to be learned from Bigtable's development is that simplicity and clarity in design as well as code are of great value, especially for big and unexpectedly evolving systems like Bigtable.

Data Model: Big Table is "a sparse, distributed, persistent multidimensional sorted map". Values are stored as arrays of bytes which do not get interpreted by the data store. They are addressed by the triplet (row-key, column-key, timestamp).



*Rows:*The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row. Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a tablet, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines.

Column Families: Column keys are grouped into sets called column families, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column key is named using the syntax: family:qualifier. Access control and both disk and memory accounting are performed at the column-family level.

Timestamps: Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first. To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last n versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

The implementation consists of the following major components:

- Multiple tablet servers: each of which is responsible for a number of tablets. This implies the handling of read and write requests for tablets as well as the splitting of tablets “that have grown too large”. Tablet servers can be added and removed at runtime.

- A client library provided for applications to interact with Bigtable instances. The library responsible for looking up tablet servers that are in charge of data that shall be read or written, directing requests to them and providing their responses to client applications.

- One master server: Firstly, it manages the tablets and tablet servers: it assigns tablets to tablet servers, detects added and removed tablet servers, and distributes workload across them. Secondly, it is responsible to process changes of a Bigtable schema, like the creation of tables and column families. Lastly, it has to garbage-collect deleted or expired files stored in GFS for the particular Bigtable instance. Despite these responsibilities the load on master servers is expected to be low as client libraries lookup tablet location information themselves and therefore "most clients never communicate with the master". As the master server is a single point of failure for a Bigtable instances it is backed up by a second machine.

Big Table has been very successful at Google and is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth.

Hbase

HBase, from Apache, is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable. It is built using Java for providing a fault-tolerant way of storing large quantities of sparse data. It uses HDFS (Hadoop Distributed File System) which takes the same role as GFS in BigTable. It serves as source and destination for the jobs executed in the Map-Reduce framework in Hadoop. HBase is an open-source implementation to provide BigTable-like capabilities and borrows many of the BigTable features like compression, in-memory operation and bloom filters on a per-column basis. It is used in many data-driven websites and, in 2010, was elected for implementing Facebook's Messaging Platform.

Cassandra

Cassandra, originally developed by Facebook, adopts ideas and concepts of both, Amazon's Dynamo as well as Google's Bigtable. It has been adopted by other companies like Twitter, Dig and Rackspace. It can be described as a distributed storage system for managing structured data that is designed to scale to a very large size.

The 'Inbox Search' problem led to the initial design and development of Cassandra at Facebook. The users can exchange personal messages with their contacts which appear in the inbox of a recipient and the problem was to find an efficient way of storing, indexing and searching these messages. The major requirements of problems of such nature are listed below [18]:

- Processing of a large amount and high growth rate of data
- High and incremental scalability
- Cost-effectiveness
- Reliability at massive scale since outages in the service can have significant negative impact
- The ability to run on top of an infrastructure of hundreds of nodes (commodity servers) across different datacenters.
- A high write throughput while not sacrificing read efficiency
- No single point of failure
- Treatment of failures as a norm rather than an exception

Data Model

An instance of Cassandra typically consists of only one table which represents a "distributed multidimensional map indexed by a key". The values in the table are addressed by the triplet (row-key, column-key, timestamp) [columnkey as column-family:column (for simple columns contained in the column family) or column-family:supercolumn:column (for columns subsumed under a supercolumn)]. The dimensions in the triplet are as follows:

- row-key: rows are identified by a string-key of arbitrary length.
- Column-key: identifies a column in a row and is addressed with 'column families'/ 'column'/ 'super column'.
    - Column families: As in Bigtable, column-families have to be defined in advance, i. e. before a cluster of servers comprising a Cassandra instance is launched. A column family consists of columns and super columns which can be added dynamically (i. e. at runtime) to column-families and are not restricted in number.
    - Columns: have a name and store a number of values per row which are identified by a timestamp (like in Bigtable). Each row in a table can have a different number of columns, so a table cannot be thought of as a rectangle. Client applications may specify the ordering of columns within a column family and super column which can either be by name or by timestamp.
    - Super columns: have a name and an arbitrary number of columns associated with them. Again, the number of columns per super-column may differ per row.

Partitioning

A consistent hashing function, which preserves the order of row-keys, is used to partition and distribute data among the nodes. The order preservation property of the hash function is important to support range scans over the data of a table. Consistent hashing is also used by Dynamo, but Cassandra handles this differently: while Dynamo hashes physical nodes to the ring multiple times (as virtual nodes), Cassandra measures and analyzes the load information of servers and moves nodes on the consistent hash ring to get the data and processing load balanced. Lakshman and Malik claim that this method has been chosen as "it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing" [18].

Replication

Replication is managed by a coordinator node and the coordinator node for a key is the first node on the consistent hash ring that is visited when walking from the key's position on the ring in clockwise direction. Following replication strategies are available:

- Rack Unaware: replication strategy within a datacenter where (N − 1)^15 nodes succeeding the coordinator node on the consistent hash ring are chosen to replicate data to them (N is a replication factor).
- Rack Aware and Datacenter Aware: replication strategies in which a leader is elected for the cluster that is in charge of maintaining the invariant that no node is responsible for more than N-1 ranges in the ring. This is done with a system called Zookeeper which is part of Apache's Hadoop project for providing a "centralized service for maintaining configuration information, naming, providing distributed synchronization and providing group services". The metadata about the nodes' responsibilities for key ranges is cached locally at each node as well as in the Zookeeper system. Nodes that have crashed and start up again therefore can determine which key-ranges they are responsible for. This metadata is similar to the preference list maintained in Amazon's Dynamo.

Failure Detection

Nodes within a Cassandra cluster try to locally detect whether another node is up or down to avoid connection attempts to unreachable nodes. The mechanism employed for this purpose of failure detection is based on a modified version of the Accrual Failure Detector. The failure detector does not give a Boolean result for the failure detection but provide a suspicion level for the monitored nodes which indicates the probability about their availability. It is claimed that experience has shown that Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions [18].

Persistence

Unlike Bigtable and its derivatives, Cassandra persist its data to local files instead of a distributed file system. However, the data representation in memory/disk, the processing of read/write operations is borrowed from Bigtable.

- Write operations first go to a persistent commit log and then to an in-memory data structure. This in-memory data structure gets persisted to disk as an immutable file if it reaches a certain threshold of size. All writes to disk are sequential and an index is created "for efficient lookup based on row key" (like the block-indices of SSTables used in Bigtable).
- Read operations consider the in-memory data structure as well as the data files persisted on disk. "In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory". It "is first consulted to check if the key being looked up does indeed exist in the given file".

Cassandra also maintains indices for column families and columns to "jump to the right chunk on disk for column retrieval" and avoid the scanning of all columns on disk.

10. Conclusion

So far, this document discussed about the motivation, evolution and some implementations of the NoSQL databases. The NoSQL databases were broadly classified into 3 categories and we analyzed few data store implementations that fall into those categories. Each of them has been motivated by varying requirements which has led to their development mostly from the industry. Each data store and its implementation has strengths at addressing specific enterprise or cloud concerns such as being easy to operate, providing a flexible data model, high availability, high scalability and fault tolerance. Each NoSQL database should be used in a way that it meets its claims and the overall system requirements. It was seen as to how the different data stores were designed to achieve high availability and scalability at the expense of strong consistency. The different data stores use different techniques to achieve this goal and seem to suit well for their requirements. The following table gives a summary of some of the features across the data stores that have been discussed here.

Data store	Classification	License	Concurrency control	Data storage	Replication
Dynamo	Key-value store	Proprietary	MVCC	Plug-in	Asynchronous
Voldemort	Key-value store	Apache	MVCC	RAM	Asynchronous
CouchDB	Document store	Apache	MVCC	Disk	Asynchronous
MongoDB	Document store	GPL	Locks	Disk	Asynchronous
Big Table	Extensible store	Proprietary	Lock/stamps	GFS	Asynchronous/ Synchronous
HBase	Extensible store	Apache	Locks	HDFS	Asynchronous
Cassandra	Extensible store	Apache	MVCC	Disk	Asynchronous

11. Acknowledgement

I hereby thank Nikita Spirin for his guidance and comments during the course of this survey.

12. References

[1] Codd, Edgar F.: A Relational Model of Data for Large Shared Data Banks. In: Communications of the ACM 13 (1970), June, No. 6, p. 377–387

[2] Strozzi, Carlo: NoSQL – A relational database management system. 2007–2010. – http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page

[3] Stonebraker, Michael ; Madden, Samuel ; Abadi, Daniel J. ; Harizopoulos, Stavros ; Hachem, Nabil ; Helland, Pat: The end of an architectural era: (it's time for a complete rewrite). In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, 2007, p. 1150–1160

[4] M. Stonebraker, "The case for shared nothing," IEEE Database Eng. Bull., vol. 9, no. 1, pp. 4–9, 1986.

[5] Brewer, Eric A.: Towards Robust Distributed Systems. Portland, Oregon, July 2000. –

Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on 2000- 07-19. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[6] Data Management: Past, Present, and Future : Jim Gray - Microsoft Research - June 1996

[7] Stonebraker, Michael: The “NoSQL” Discussion has Nothing to Do With SQL.
November 2009. – Blog post of 2009-11-04. http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-withsql/fulltext

[8] Rick Cattell : Scalable SQL and NoSQL Data Stores. http://cattell.net/datastores/Datastores.pdf

[9] THEO HAERDER - Fachbereich Informatik, University of Kaiserslautern, West Germany : Principles of Transaction-Oriented Database Recovery

[10] Oracle Corporation: Oracle Berkeley DB Products. 2010. –
http://www.oracle.com/us/products/database/berkeley-db/index.html

[11] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine,M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing (El Paso,Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[12] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., andBalakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of
the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, California, United States). SIGCOMM '01.
ACM Press, New York, NY, 149-160.

[13] Gobioff, Sanjay Ghemawat H. ; Leung, Shun-Tak: The Google File System. In: SIGOPS Oper. Syst. Rev. 37 (2003), No. 5, p. 29–43. – http://labs.google.com/papers/gfs-sosp2003.pdf

[14] Kreps, Jay et al.: Project Voldemort – Design. 2010. – http://project-voldemort.com/design.php

[15] The Case for Shared Nothing - Michael Stonebraker University of California Berkeley, Ca.
http://pdf.aminer.org/000/255/770/the_case_for_shared_nothing.pdf

[16] Lakshman, Avinash: Cassandra - A structured storage system on a P2P Network. August 2008. – Blog post of 2008-08-25 -
http://www.facebook.com/note.php?note_id=24413138919

[17] http://cassandra.apache.org/

[18] Lakshman, Avinash ; Malik, Prashant: Cassandra – A Decentralized Structured Storage System. In: SIGOPS Operating Systems Review 44 (2010), April, p. 35–40. –
http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

[19] http://hbase.apache.org/

[20] Michael Stonebraker, Uğur Çetintemel : “One Size Fits All”: An Idea Whose Time Has Come and Gone -
http://www.cs.brown.edu/~ugur/fits_all.pdf

[21] A Comparison of Approaches to Large-Scale Data Analysis by Michael Stonebraker, Andrew Pavlo, Erik Paulson et al.:
http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf

[22] MapReduce: Simplified Data Processing on Large Clusters :
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/mapreduce-osdi04.pdf

[23] Directory of NoSQL databases with basic information on the individual datastores:
http://nosql-database.org/

[24] Why we're using HBase by Cosmin Lehene:
http://hstack.org/why-were-using-hbase-part-1/

[25] M. Stonebraker and R. Cattell, "Ten Rules for Scalable Performance in Simple Operation Datastores", Communications of the ACM, June 2011.

[26] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant web services", ACM SIGACT News 33, 2, pp 51-59, March 2002.

[27] N. Leavitt, "Will nosql databases live up to their promise?" Computer, vol. 43, no. 2, pp. 12–14, 2010.

[28] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on. IEEE, 2011, pp. 363–366.

[29] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in ACM SIGOPS Operating Systems Review, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[30] C. Strauch, U. Sites, and W. Kriha, "Nosql databases," Lecture Notes, Stuttgart Media University, 2011.

[31] W. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009.

[32] M. Stonebraker, "The case for shared nothing," IEEE Database Eng. Bull., vol. 9, no. 1, pp. 4–9, 1986.

[33] C. Feng, Y. Zou, and Z. Xu, "Ccindex for cassandra: A novel scheme for multi-dimensional range queries in cassandra," in Proceedings of the 2011 Seventh International Conference on Semantics, Knowledge and Grids, ser. SKG '11, 2011, pp. 130–136.

[34] Apache. (2012, April) Apache couchdb. [Online]. Available: http://couchdb.apache.org/

[35] M. Stonebraker, "Sql databases v. nosql databases," Communications of the ACM, vol. 53, no. 4, pp. 10–11, 2010.

[36] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[37] P. Xiang, R. Hou, and Z. Zhou, "Cache and consistency in nosql," in Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, vol. 6. IEEE, 2010, pp. 117–120.

[38] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.

[39] C. Zhang, H. De Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek, "Case study of scientific data processing on a cloud using hadoop," in High Performance Computing Systems and Applications. Springer, 2010, pp. 400–415.

[40] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, “Map-reduce-merge: simplified relational data processing on large clusters,” in Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007, pp. 1029–1040.

[41] http://en.wikipedia.org/wiki/Multiversion_concurrency_control

[42] http://en.wikipedia.org/wiki/Vector_clock

[43] http://www.christof-strauch.de/nosqldbs.pdf

[44] White, Tom: Consistent Hashing. November 2007. – Blog post of 2007-11-27. http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

[45] http://en.wikipedia.org/wiki/HBase

[46] Vogels, Werner: Eventually consistent. December 2008. – Blog post of 2008-12-23. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[47] Strozzi, Carlo: NoSQL – A relational database management system. 2007–2010. – http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page

[48] Stearn, Mathias ; Merriman, Dwight: mongoDB Manual – Inserting - Trees in MongoDB. March 2010. – Wiki article, version 21 of 2010-03-10. http://www.mongodb.org/display/DOCS/Trees+in+MongoDB

[49] Shalom, Nati: The Common Principles Behind The NOSQL Alternatives. December 2009. – Blog post of 2009-12-15. http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principlesbehind-the-nosql-alternatives.html

[50] http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principles-behind-the-nosql-alternatives.html

[51] http://hadoop.apache.org/

[52] http://aws.amazon.com/simpledb/