Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 97 (2016) 94 – 103

Procedia Computer Science

CLOUD FORWARD: From Distributed to Complete Computing, CF2016, 18-20 October 2016, Madrid, Spain

A classification of NoSQL data stores based on key design characteristics

Antonios Makris[a], Konstantinos Tserpes[a,b,]*, Vassiliki Andronikou[b], Dimosthenis Anagnostopoulos[a]

[a]*Harokopio University of Athens, 9 Omirou, Tavros 17778, Greece*
[b]*National Technical University of Athens, 9 Iroon Polytechniou, Zografou 15773, Greece*

Abstract

Traditional Relational Database Management Systems are continuously being replaced by NoSQL data stores as a result of the growing demand for big data applications. The emergence of a large number of implementations of such like systems is a contributing indicator. This paper deals with the analysis of some key design characteristics of NoSQL systems and uses these for their characterization based on their capabilities. Furthermore, it highlights the relationship between NoSQL systems and cloud infrastructures and explains the impact that the existence of one has to the other.

Keywords: NoSql; RDBMS; Cloud Computing; Data stores; Design characteristics

1. Introduction

Relational database management systems (RDBMS) are widely used in many applications in order to store and retrieve data. However, these systems cannot cope with the unprecedented scale factors that modern web-based applications have introduced. These applications need to support large numbers of concurrent users (tens of

* Corresponding author. Tel.: +30 2109549413; fax: +30 2109549401.
E-mail address: tserpes@hua.gr

doi:10.1016/j.procs.2016.08.284

thousands, even millions), deliver highly responsive experiences to a globally distributed base of users, be always available, handle semi- and unstructured data and rapidly adapt to changing requirements with frequent updates and new features. Thus traditional relational database management systems (RDBMSs) may not be the right fit anymore for the emerging generation of interactive applications.

To mitigate the issue, a growing number of solutions have turned to non-relational databases, commonly referred to as NoSQL databases, in order to enable the setup of massively parallel and geographically distributed database systems to support primarily large-scale web application demands. NoSQL is a class of database management systems (DBMS) that do not follow all of the rules of a relational DBMS and cannot use traditional SQL queries over the data. The term "NoSQL" means "Not Only SQL" as this type of database is not generally a replacement but, rather, a complementary addition to RDBMSs and SQL. NoSQL-based systems (or NoSQL data stores) are typically used in very large databases, which are particularly prone to performance problems caused by the limitations of SQL and the relational model of databases. The main feature of NoSQL data stores is the "shared nothing" horizontal scaling, which enables them to support a large number of simple read/write operations per second.

The focus of this paper is to highlight the differences between the relational databases and NoSQL data stores while at the same time to present a categorization of the widely used NoSQL data stores according to key architectural characteristics they support. Also the relationship between cloud infrastructures and NoSQL data stores is presented as well as the reason why these data stores constitute a major component in cloud computing. Based on the literature review and to the best of our knowledge the most prominent differentiators for NoSQL data stores are their data model and a number of capabilities that are supported based on key decisions made at the design phase. Such capabilities relate to the way huge datasets are partitioning among the distributed resources, the way they replicate data for fault tolerance and performance, the particular fault tolerance strategy employed and finally, the mechanism that is used to maintain consistency across the data items.

The rest of this paper is organized as follows. Section 2 introduces the basics of NoSQL databases. Section 3 presents a categorization of various off-the-shelf NoSQL data stores into four major categories. Section 4 represents the key characteristics of the systems we examined while Section 5 presents the final conclusions.

2. From RDBMS to NoSQL

There's no simple answer as to why market and research started looking for alternatives to RDBMS. With the increased volume of data and the expensive cost associated with scaling the relational RDBMSs there was a genuine need for something different. NoSQL data stores are designed to scale horizontally and run on commodity hardware. From an implementation point of view, they leverage on cloud infrastructures that are adaptive to the scaling, availability and performance requirements. This goes hand in hand with the modern applications' needs which call for a more adaptive approach in all aspects (e.g. data structure or query processing), something that contradicts the 'one size fits all' notion of the RDBMS. It is more efficient and cheap to build systems based on the nature of the application and its work/data load.

It therefore seems that the ACID properties (Atomicity, Consistency, Isolation and Durability) implemented by RDBMSs might set unnecessary constraints to the particular applications and use cases which in turn prohibits them to tackle the challenges at hand. Indeed, guaranteeing ACID properties in a distributed transaction across a distributed database where no single node is responsible for all data affecting the transaction, presents complications. Most of the NoSQL data stores generally do not provide strict ACID properties: updates are eventually propagated, but there are limited guarantees on the consistency of reads. The idea is that by giving up strict ACID constraints high performance and scalability can be achieved.

Instead of ACID, NoSQL databases provide BASE semantics which are compliant with the eventual consistency model, that is, a) Basically available: Systems seems to work always, b) Soft state: May not be consistent always, and c) Eventual consistency: Will become consistent at some later time.

Also, in distributed databases there is a trade-off between consistency and availability which was formalized by the CAP theorem: it is impossible for a distributed system to simultaneously provide all three of the guarantees in terms of Consistency, Availability and Partition tolerance. Rather, it is realistic for a distributed system to aim for two out of the three guarantees. The simplest interpretation of the CAP theorem is to consider a distributed data store partitioned into two sets of participant nodes. If the data store denies all write requests in both partitions, it will remain consistent, but it is not available. On the other hand, if one (or both) of the partitions accepts write requests, the data store is available, but potentially inconsistent[1].

A comparison between NoSQL and RDBMS is presented in Table 1:

Table 1. NoSQL vs RDBMS.

Feature	**RDBMS**	**NoSQL**
Data Validity	Lower guarantees	Higher guarantees
Query Language	Structured Query Language (SQL)	No declarative query language
Data type	Supports relational data and its relationships are stored in separate tables	Supports unstructured and unpredictable data
Data Storage	Stored in a relational model, with rows and columns. Rows contain all of the information about one specific entry/entity, and columns are all the separate data points	The term "NoSQL" encompasses a host of databases, each with different data storage models. The main ones are: document-based-store, graph-based, key-value-store, column-based-store
Schemas and Flexibility	Each record conforms to fixed schema	Schemas are dynamic. Each 'row' doesn't have to contain data for each 'column'
Scalability	Vertically	Horizontally, meaning across servers
ACID Compliancy	The vast majority of relational databases are ACID compliant.	Sacrifice ACID compliancy for performance and scalability Based on BASE principle

3. NoSQL Data and Query Models

As already mentioned, there are different types of NoSQL databases. A high-level taxonomy of the NoSQL data stores based on the data model can classify them into four major categories: key-value stores, column-family stores, document stores, and graph databases. This categorization is rather common and we follow the definitions provided in[2]. Concerning querying in NoSQL databases, there is no universal query language which works for all the types of data model. Each database has its own query language which is data-model specific. So far there have been efforts for the creation of a standard query language UnQL[3] (Unstructured Query Language) for NoSQL databases. The syntax for the querying is SQL-like and is used across various document-oriented databases. Also, most of the NoSQL databases allow RESTful interfaces to the data and query APIs.

3.1. Key-value stores

The use of this model implies that the stored values correspond to specific keys. They follow a similar rationale as with maps or dictionaries where data is addressed by a unique key. The key can be synthetic or auto-generated while the value can be any object type, including String, JSON, BLOB etc. The key value model uses a hash table which is comprised of a unique key and a pointer to a particular item of data. The values are stored as uninterpreted byte arrays, therefore the keys are the only way to retrieve stored data. The grouping of key value pairs into collections is the only option to add some kind of structure to the data model. Also, key value stores supports simple operations, which are applied on key attributes only. Most key value stores hold the whole data in memory (in memory key value stores-IMKVS), which make them good candidates for caching and high throughput system requirements. The key value stores considered in this paper are Redis[3], MemCached[4], and Dynamo[5].

The query operations for stored objects are associated with a key. The type of operations that are offered through APIs include functions like get, put and delete. These interfaces enable simple queries to be performed (as complex queries are not supported) rendering the query language unnecessary. For instance, Dynamo exposes two operations: a) get (key) which returns a list of objects and a context and b) put (key, context, object) which performs a write operation.

3.2. Column family stores

The column family stores data model is described by Google in BigTable[6] as "sparse, distributed, persistent multidimensional sorted map". In this map, an arbitrary number of key value pairs can be stored within rows. To support versioning and achieve performance, multiple versions of the values are sorted in chronological order. The columns can be grouped to column families to support organization and partitioning. The column family stores data model 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. They can be visualized the same way as RDMS do, due to their table format. The main difference lies in their handling of null values, i.e. what constitutes the substantial difference is that RDBMS would store a null value in each column but column family stores only store a key value pair in a row only if a dataset needs it. Column family stores considered in this paper are PNUTS[7] and Cassandra[8]. Cassandra's data model differs from the typical column family stores, in that it maintains one more dimension called super column. A super column family can be visualized as a column family within a column family[2].

Column family stores support range queries and the use of regular expression on the indexed values or on the row keys for querying. This has an impact on the extent to which the databases are affected, i.e. operations are only affecting the keys that are related to the query rather –in an extreme case for the sake of the example- the entire row from disk. Apart from the typical accessors and mutators, the column-oriented databases frequently support operations like OR, IN, AND. A relevant example is the Cassandra API which provides three main operations: get (table, key, columnName), insert (table, key, rowMutation), delete (table, key, columnName).

3.3. Document stores

In Document stores the data are again a collection of key value pairs, but now they are compressed as a document, i.e. entities that provide some structure and encoding of the managed data. All keys in the document have to be unique, and every document contains a special key "ID", which is also unique within a collection of documents and therefore identifies a document explicitly. Document stores do not have any schema restrictions and support multi attribute lookups on records which may have different kinds of key value pairs. Document stores considered in this paper are MongoDB[9] and CouchDB[10].

Document stores embeds attribute metadata associated with stored content, which essentially - in contrast to key value stores, provides a way to query the data based on the contents. Range queries, indexing and nested document querying are supported. Also, allows the use of operations like OR, AND and BETWEEN and queries can be implemented as Map Reduce jobs. MongoDB for example supports operators like: create, insert, read, update, remove. It also supports manual indexing, indexing on embedded documents and index location-based data.

3.4. Graph stores

In graph stores a flexible graphical representation is used which is perfect to address scalability concerns. Graph structures are used with edges, nodes and properties which provides index-free adjacency. 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. Data can be easily transformed from one model to the other using a Graph Base NoSQL database. Graph databases are specialized on efficient management of heavily linked data and are optimized for highly connected data[11]. 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[2]. Graph stores considering this paper is Neo4j[11] and HyperGraphDB[12].

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 such as graph traversal and graph pattern matching techniques. In graph traversal, the query processing starts from one node and then the other nodes are traversed as per the description of the query. The graph pattern matching technique tries to locate in the original graph, the defined pattern which is to be searched for. For example in Neo4j each vertex and edge in the graph store a “mini-index” of the objects connected to it. This means that the size of the graph has no performance impact upon a traversal and the cost of a local step (hop) remains the same. There is also global adjacency index but it is only used when trying to find the starting point of a traversal. Indexes are required in order to quickly retrieve vertices based on their values. They provide a starting point at which to begin a traversal.

4. Other Key Characteristics

A more in-depth analysis of the existing data stores would reveal some key technical characteristics that can distinguish one solution from another. As such, one of the main characteristics of the NoSQL data stores is their ability to scale horizontally and effectively by adding more servers into the resource pool. The data placement, partition, replication, consistency and fault tolerance strategies used by the NoSQL data stores have significant impact on their scalability. In this section we describe and analyze in depth the above strategies that consists of the key characteristics of NoSQL data stores.

4.1. Data Placement

Data placement relates to the problem of optimal distribution of the data objects among the existing servers. According to[13], data placement policies can be separated into two categories: a) geographical placement policies – data moved between geographically sparse data-centers and b) data partitioning policies – data moved between nodes.

Below are represented the main techniques that are used in distributed databases for data placement:

- Consistent hashing: this method uses a hash function to map an object key into a server. Servers are organized in a logical ring, and assigned to them a unique identifier. Then, with the use of a hash function, key values mapped into identifiers in the server identifier space[14]
- Multi-Attribute Sharding: data distribution by primary key regardless to other attributes[15]
- Bloom filters: these constitutes of efficient probabilistic data structures and used to check if an object is mapped to a given node
- Probabilistic Associative Array: this method is a space efficient mapping data structure and can be used to store arbitrary associations among keys and nodes. Multiple Bloom filters are used for tracking the inserted data items and also Decision Tree classifiers[16]
- Dynamic data placement: changing the placement of some or all data items in response of workload changing

To tackle the data placement problem for a utility function, the NoSQL systems are often leveraging on elastic infrastructures, thus the they add or remove VMs and storage space according to the needs. Recent literature examines the problem from the performance optimization perspective. To this end, the authors are considering the pros and cons of an adaptive cluster resizing process according to dynamic workload changes types and rates in three NoSQL systems (HBASE, Cassandra and Riak). For the experiments, they employed an open-source project called Tiramola[17] which is used as elasticity-testing framework consisting of a) a Command Issuing Module which is responsible for adding or removing VMs; b) a Monitoring Module which collects performance metrics and statistics; c) a Rebalancing Module which is responsible for the rebalance operations of client requests after the adding or removing a virtual node; d) a Cloud management module which receives commands for cluster resizing; and, e) a Cluster coordinator, which is a module responsible for the remote execution of shell scripts of the commands like add or remove node from the NoSQL cluster.

According to the experiments in the three NoSQL systems HBASE was the fastest and can further scale with node additions. Cassandra scales without transitional phase through its decentralized nature and Riak can scale and supports automatically rebalancing but only in low client request rates[18].

4.2. Partitioning

In order to store and process massive datasets, a commonly employed strategy is to partition the data and store the partitions across different server nodes. The NoSQL systems that were examined in this work use partition strategies to implement highly-available and scalable solutions that can be leveraged in cloud environments.

In the case of Redis, partitioning means that every instance will only contain a subset of keys. There are many ways to perform partitioning but the main ones are range and hash partitioning and consistent hashing. In range partitioning, ranges of objects are mapped into specific Redis instances. The other alternative is hash partitioning, where the scheme works with any key, without requiring the key to be in the form object_name:<id>[3].

In MemCached (MC) the partitioning is performed through consistent hashing and clients select a unique server per key, requiring only the knowledge of the total number of servers and their IP addresses[4].

In Dynamo the partition relies on consistent hashing. The output range of consistent hashing is treated as a "ring". Each node is assigned a random value in the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. To avoid non-uniform data and non-heterogeneity, each node is mapped to multiple points in the ring rather than a single point (virtual nodes). Each virtual node seems like a single node in the network, but each node will be in charge of more than one virtual node. The number of virtual nodes corresponding to each physical node is determined based on the capacity and heterogeneity in the infrastructure[5].

In Cassandra the partition of the data across the cluster is achieved through consistent hashing with the use of an order-preserving hash function to do so. The consistent algorithm that is used is the same as in the case of Dynamo without virtual nodes[8].

In PNUTS the data are organized into tables of attributed records. Data tables are horizontally partitioned into groups of records called tablets. Tablets are scattered across many servers. Each server might have hundreds or thousands of tablets, but each tablet is stored on a single server within a region[7].

In MongoDB the partitioning is accomplished through sharding. Two types of sharding are provided: auto and manual. Manual sharding is when the user can set up two MongoDB master servers and store half data on one and the rest of the data on the other. In auto sharding, a dedicated system component takes care of all the data splitting and recombination. It makes sure the data goes to the right server and that queries are run and combined in the most efficient manner possible[9].

CouchDB partitioning is based on consistent hashing (sharding) just like MongoDB. A set of CouchDB servers is logically assigned to a position on a ring, and documents are stored on the server that follows their hash value on the ring. The topology of the ring is replicated on each server, so that Client requests can be forwarded in one message to the relevant server[19].

Neo4j supports only cache-based partition. With this type of partition Neo4j can make direct reads to particular instances where the system will already have those datasets in memory. This approach is significantly beneficial when your total active dataset is much larger than can fit in memory in any particular instance[20].

In HyperGraphDB, the partitioning is implemented at the model layer by using an agent-based, P2P framework. Algorithms are developed using communication protocols built using the Agent Communication Language (ACL) FIPA standard[21]. Each agent maintains a set of conversations implementing dialog workflows with a set of peers. All activities are asynchronous and incoming messages are dispatched by a scheduler and processed in a thread pool [12].

4.3. Replication

An important operation that relates to dependability on NoSQL systems is replication. Replication is the process by which stored the same data on multiple servers so that read and write operations can be distributed over them. Replication also has an important role in providing fault tolerance because data availability can withstand the failure of one or more servers. Furthermore, the choice of replication model is also strongly related to the consistency level provided by the data store.

Redis supports master-slave replication that allows slave Redis servers to be exact copies of master servers. A Redis slave sends a SYNC command and the master then, starts background saving and buffer all new commands received that will modify the dataset. When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory. Subsequently, the master will send to the slave all buffered commands[3].

Initially MemCached did not support replication. Over the years, many alternatives have been supported for scaling. For example, Repcached[22] can be added to MC for replication. Another solution is R-Memcached[19], a consistent cache replication scheme. Finally another solution, describes how to scale MC by storage replication, pools replication and region replication[23].

Data replication is used for providing availability and durability in Dynamo. Each data item is replicated N-times where N can be configured “per-instance” of Dynamo. The node, which is in charge of the data item with key value k, is also responsible for storing the updated replicas in N-1 nodes. For each key k, there will be a preference list which contains a list of nodes where the data item with key k has to be stored[5].

In Cassandra each data item is replicated at N hosts, where N is the replication factor configured “per-instance”. Each key k, is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring[8].

For replication and logging, a publish/subscribe system, the Yahoo! message broker (YMB) is used. Data updates are considered “committed” when they have been published to YMB. At some point after being committed, the update will be asynchronously propagated to different regions and applied to their replicas. YMB is used for replication and logging for two reasons. First, YMB takes multiple steps to ensure messages are not lost before they are applied to the database. YMB guarantees that published messages will be delivered to all topic subscribers even in the presence of single broker machine failures. It does this by logging the message to multiple disks on different servers. Secondly, YMB is designed for wide-area replication: YMB clusters reside in different, geographically separated datacenters, and messages published to one YMB cluster will be relayed to other YMB clusters for delivery to local subscribers. This mechanism isolates individual PNUTS clusters from dealing with update propagation between regions[7].

MongoDB uses master-slave replication. In practice, this means that only one database is active for writing at any given time. By passing all writes to the first database (the master database) and to a replica (the slave database) of the master database, there is no worry if the master database fails because the slave database can carry on in its place. Of course, it is possible that some of the data written by the master database will not yet have made it to the slave database at the point a failure occurs. Also MongoDB supports replica pairs[9].

In CouchDB, different database nodes can -by design- operate completely independent and process read and write requests. For example, two database nodes can replicate databases bilaterally. The replication process works incrementally and can detect conflicting versions in a 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. Incrementally means that only data changed since the last replication gets transmitted to another node and that not even whole documents are transferred but only changed fields and attachment-blobs[10].

The replication in Neo4j follows a master - slave architecture. Neo4j commits all writes from a single master instance. So before a write is persisted to all other instances in the cluster, the slave which received the write must synchronize with the master[11].

Finally, HyperGraphDB[24] implements a P2P framework for distributed processing for replication. It supports a multi-master, asynchronous replication and the communication conducted by using an agent-based P2P framework[12].

4.4. Fault tolerance

NoSQL database environments are no exception when it comes to hardware failures. However due to their distributed architecture there is no single point of failure and there is built-in redundancy of both function and data. If one or more database server or node goes down, the other nodes in the system are able to continue their operations without data loss, thereby showing true fault tolerance. In this way, NoSQL database environments are able to provide continuous availability whether in single locations, across data centers and in the cloud. There are many methods to achieve fault tolerance such as replication, the institution of communication protocols between nodes, monitoring and so on. Table 2 presents a taxonomy of the NoSQL data stores in question based on the underlying fault tolerance mechanisms they are employing.

Table 2. Fault tolerance .

NoSQL database systems	Fault tolerance mechanism
Redis	ping - pong protocol: • All nodes continuously ping other nodes (each packet contains a gossip section) • A node marks another node as possibly failing when there is a timeout longer than N seconds
MemCached	Drops old data as new data come in, if memory limits are reached. Fault-tolerance is not part of the MemCached system.
Dynamo	Fault tolerance is achieved in Dynamo through replication. Each data is replicated at a number of hosts. Also instead of mapping a node to a single point, each node gets assigned to multiple points.
Cassandra	Fault tolerance is achieved in Cassandra through replication. Each data is replicated in a number of hosts. Each node that crashes and comes back, knows the ranges that is responsible for, from the locally cached metadata (ranges of nodes) that exist into Zookeeper. Zookeeper is a system that uses Cassandra to elect a leader among nodes. Cassandra is configured such that each row is replicated across multiple data centers.
PNUTS	YMB acts as a fault tolerant publish-subscribe system by mapping tablets to its replicas
MongoDB	MongoDB uses replica pairs. In the case of two servers, these automatically decide which server is the master and which is the slave. If a server fails, the two servers will automatically sort out how to proceed when the failed server comes back online. Also in MongoDB supported automatic failover and automatic recovery.
CouchDB	Fault tolerance through replication. Database nodes can replicate databases bilaterally. 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. Also it allows partial replicas.
Neo4j	Neo4j HA is fault tolerant and can continue to operate from any number of machines down to a single machine. If the master fails a new master will be elected automatically.
HyperGraphDB	Fault tolerance in achieved through asynchronous replication through a P2P communication protocol.

4.5. Consistency

Consistency is a system property that ensures that a transaction brings the database from one valid state to another. The main consistency models are: strong and eventual consistency. Strong or immediate consistency ensures that when write requests are confirmed, the same (updated) data are visible to all subsequent read requests. Synchronous replication on one hand ensures strong consistency, but on the other hand it introduces latency. Asynchronous replication, will lead to eventual consistency because there is a lag between write confirmation and propagation. In the eventual consistency model, changes eventually propagate through the system given sufficient time. Therefore, some server nodes may contain inconsistent (outdated) data for a period of time. In general, there is a trade-off between consistency and high availability of data in NoSQL data stores.

Redis is not able to guarantee strong consistency because it uses asynchronous replication. If master crashes before send the writes from the client to its slaves, these writes will be lost forever[3].

In MemCached all operations are atomic. Thus, it achieves strong consistency[4].

In Dynamo 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[5].

In Cassandra cluster a read/write request for a key gets routed to any node. For writes, the system routes the requests to the replicas and waits for a quorum of replicas to acknowledge the completion of the writes. For reads, based on the consistency guarantees required by the client, the system either routes the requests to the closest replica or routes the requests to all replicas and waits for a quorum of responses[8].

PNUTS provides per-record timeline consistency: all replicas of a given record apply all updates to the record in the same order. One of the replicas is designated as the master, independently for each record, and all updates to that record are forwarded to the master[7].

MongoDB supports immediate consistency which means that limit the application of updates to a single master node for a given slice of data. All the updates made in-place. This means that MongoDB can update the data wherever it happens to be[9].

CouchDB achieves eventual consistency between multiple databases by using incremental replication. Incremental replication is a process where document changes are periodically copied between servers[10].

Neo4j supports eventual consistency. All updates propagate from the master to other slaves eventually so a write from one slave may not be immediately visible on all other slaves[11]. Finally, HyperGraphDB supports eventual consistency. Each agent broadcasts interest in certain atoms. Each peer then listens to atom events from the database and upon addition, update or removal of an atom, it notifies interested peers by sending an inform communication act. Finally, to ensure consistency, local transactions are linearly ordered by a version number and logged so that they can eventually reach all interested peers[24].

4.6. Summary

Table 3 presents the examined NoSQL systems classified based on the criteria of partitioning, replication, fault tolerance mechanism and consistency.

Table 3. Summary of key characteristics.

NoSQL Database Systems	Data Model	Partitioning	Replication	Consistency
Redis	Key-value store	Range partitioning, hash partitioning, consistent hashing	Master–slave, asynchronous replication	Eventual consistency
MemCached	Key-value store	Clients' responsibility. Consistent hashing prevailing	Repcached, R-MemCached, storage, pools, region replication	Strong consistency (single instance)
Dynamo	Key-value store	Consistent hashing but each node gets assigned to multiple points to the ring	Synchronous/ Asynchronous	Eventual consistency
Cassandra	Column family store	Consistent hashing and range partitioning	Masterless, asynchronous replication	Configurable, based on quorum read and write requests
PNUTS	Column family store	Data tables are horizontally partitioned into groups of records called tablets. Tablets are scattered across many servers	Asynchronous via YMB	Eventual consistency via per-record timeline consistency
MongoDB	Document store	Range partitioning based on a shard key	Master–slave, asynchronous replication	Immediate consistency
CouchDB	Document store	Consistent hashing	Multi-master, asynchronous replication	Eventual consistency
Neo4j	Graph store	Cache-based	Master–slave	Eventual consistency
HyperGraphDB	Graph store	Graph parts can reside in different P2P nodes	Multi-master, asynchronous replication	Eventual consistency

5. Conclusion

In this paper we reviewed the concepts of the relational and NoSQL databases, and the differences between them in aspects like schema, transaction methodology, complexity, fault tolerance, consistency and dealing with storing big data. NoSQL comprise an alternative to traditional relational databases, capable of handling huge volumes of data leveraging on the capabilities of cloud environments. We classified the NoSQL databases into four categories and presented examples for each of the categories. Each of these systems and its implementation has certain key characteristics and we analyze in further the partitioning, replication, fault tolerance and consistency mechanisms. Each NoSQL database should be used in a way that it meets its claims and the overall system requirements. Also it is presented how the different data stores were designed to achieve high availability and scalability at the expense of strong consistency. There is a trade-off between consistency and high availability of data and all of the NoSQL databases address two of three between availability, consistency, partition tolerance (CAP theorem). 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. The different data stores use different techniques to achieve company or users requirements. Finally, in the future due to the fact that most applications and software tend to depend on web and the size of data required to be stored is continuously increasing rapidly, NoSQL databases will continue to dominate and be adapted to the needs.

References

1. Grolinger K, Higashino WA, Tiwari A, Capretz MA. Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: Advances, Systems and Applications. 2013;2(1):1.
2. Hecht R, Jablonski S, editors. Nosql evaluation. International conference on cloud and service computing; 2011: IEEE.
3. UnQL http://unql.sqlite.org/index.html/timeline.
4. Carra D, Michiardi P, editors. Memory partitioning in memcached: An experimental performance analysis. 2014 IEEE International Conference on Communications (ICC); 2014: IEEE.
5. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, et al. Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Systems Review. 2007;41(6):205-20.
6. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS). 2008;26(2):4.
7. Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H-A, et al. PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment. 2008;1(2):1277-88.
8. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review. 2010;44(2):35-40.
9. Hows D, Membrey P, Plugge E, Hawkins T. Introduction to MongoDB. The Definitive Guide to MongoDB: Springer; 2015. p. 1-16.
10. Strauch C, Sites U-LS, Kriha W. NoSQL databases. Lecture Notes, Stuttgart Media University. 2011.
11. Miller JJ, editor Graph database applications and concepts with Neo4j. Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA; 2013.
12. Iordanov B, editor HyperGraphDB: a generalized graph database. International Conference on Web-Age Information Management; 2010: Springer.
13. Swift BP. Data placement in a scalable transactional data store. Master's thesis, Vrije Universiteit, Amsterdam, Netherland. 2012.
14. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D, editors. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proceedings of the twenty-ninth annual ACM symposium on Theory of computing; 1997: ACM.
15. DISTRIBUTED ALGORITHMS IN NOSQL DATABASES https://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/.
16. Paiva J, Rodrigues L. On data placement in distributed systems. ACM SIGOPS Operating Systems Review. 2015;49(1):126-30.
17. Tiramola https://code.google.com/archive/p/tiramola/.
18. Konstantinou I, Angelou E, Boumpouka C, Tsoumakos D, Koziris N, editors. On the elasticity of NoSQL databases over cloud management platforms. Proceedings of the 20th ACM international conference on Information and knowledge management; 2011: ACM.
19. Liu C, Ouyang K, Chu X, Liu H, Leung Y-W, editors. R-Memcached: A Reliable In-Memory Cache System for Big Key-Value Stores. International Conference on Big Data Computing and Communications; 2015: Springer.
20. The Neo4j Operations Manual v3.0. http://neo4j.com/docs/operations-manual/current/#ha-architecture.
21. Agent Communication Language Specifications http://www.fipa.org/repository/aclspecs.html.
22. repcached - add data replication feature to memcached 1.2.x. http://repcached.lab.klab.org/.
23. Nishtala R, Fugal H, Grimm S, Kwiatkowski M, Lee H, Li HC, et al., editors. Scaling memcache at facebook. Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13); 2013.
24. HyperGrahDB http://www.hypergraphdb.org/?project=hypergraphdb&page=About.