Information Systems 63 (2017) 1–23

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Persisting big-data: The NoSQL landscape

Alejandro Corbellini*, Cristian Mateos, Alejandro Zunino, Daniela Godoy, Silvia Schiaffino

ISISTAN (CONICET-UNCPBA) Research Institute[1], UNICEN University, Campus Universitario, Tandil B7001BBO, Argentina

ARTICLE INFO

Article history:
Received 11 March 2014
Accepted 21 July 2016
Recommended by: G. Vossen
Available online 30 July 2016

Keywords:
NoSQL databases
Relational databases
Distributed systems
Database persistence
Database distribution
Big data

ABSTRACT

The growing popularity of massively accessed Web applications that store and analyze large amounts of data, being Facebook, Twitter and Google Search some prominent examples of such applications, have posed new requirements that greatly challenge traditional RDBMS. In response to this reality, a new way of creating and manipulating data stores, known as NoSQL databases, has arisen. This paper reviews implementations of NoSQL databases in order to provide an understanding of current tools and their uses. First, NoSQL databases are compared with traditional RDBMS and important concepts are explained. Only databases allowing to persist data and distribute them along different computing nodes are within the scope of this review. Moreover, NoSQL databases are divided into different types: Key-Value, Wide-Column, Document-oriented and Graph-oriented. In each case, a comparison of available databases is carried out based on their most important features.

1. Introduction

Relational databases or RDBMSs (Relational Database Management Systems) have been used since the 1970s and, as such, they can certainly be considered a mature technology to store data and their relationships. However, storage problems in Web-oriented systems pushed the limits of relational databases, forcing researchers and companies to investigate non-traditional forms of storing user data [105]. Today's user data can scale to terabytes per day and they should be available to millions of users worldwide under low latency requirements.

The analysis and, in particular, the storage of that amount of information is challenging. In the context of a single-node system, increasing the storage capacity of any computational node means adding more RAM or more disk space under the constraints of the underlying hardware. Once a node reaches its storage limit, there is no alternative but to distribute the data among different nodes. Traditionally, RDBMSs systems were not designed to be easily distributed, and thus the complexity of adding new nodes to balance data is high [67]. In addition, database performance often decreases significantly since joins and transactions are costly in distributed environments [19,86]. All in all, this does not mean RDBMSs have became obsolete, but rather they have been designed with other requirements in mind and work well when extreme scalability is not required.

Precisely, NoSQL databases have arisen as storage alternatives, not based on relational models, to address the mentioned problems. The term “NoSQL” was coined by Carlo Strozzi in 1998 to refer to the open-source database called NoSQL not having an SQL interface [108]. In 2009, the term resurfaced thanks to Eric Evans in the context of

* Corresponding author. Fax: +54 249 4385681.
E-mail addresses: alejandro.corbellini@isistan.unicen.edu.ar (A. Corbellini), cristian.mateos@isistan.unicen.edu.ar (C. Mateos), alejandro.zunino@isistan.unicen.edu.ar (A. Zunino), daniela.godoy@isistan.unicen.edu.ar (D. Godoy), silvia.schiaffino@isistan.unicen.edu.ar (S. Schiaffino).
[1] Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

http://dx.doi.org/10.1016/j.is.2016.07.009
0306-4379/

an event about distributed databases.[2] Since then, some researchers [58,57] have pointed out that new information management paradigms such as the Internet of Things would need radical changes in the way data is stored. In this context, traditional databases cannot cope with the generation of massive amounts of information by different devices, including GPS information, RFIDs, IP addresses, Unique Identifiers, data and metadata about the devices, sensor data and historical data.

In general, NoSQL databases are unstructured, i.e., they do not have a fixed schema and their usage interface is simple, allowing developers to start using them quickly. In addition, these databases generally avoid joins at the data storage level, as such operations are often expensive, leaving this task to each application. The developer must decide whether to perform joins at the application level or, alternatively, denormalize data. In the first case, the decision may involve gathering data from several physical nodes based on some criteria and then join the collected data. This approach requires more development effort but, in recent years, several frameworks such as MapReduce [31] or Pregel [74] have considerably eased this task by providing a programming model for distributed and parallel processing. In MapReduce, for example, the model prescribes two functions: a *map* function that process key-value pairs in the original dataset, producing new pairs, and a *reduce* function that merges the different results associated to each pair produced by the *map* function.

Instead, if denormalization is chosen, multiple data attributes can be replicated in different storage structures. For example, suppose a system to store user photos. To optimize those queries for photos belonging to users of a certain nationality, the Nationality field may be replicated in the User and Photo data structures. Naturally, this approach rises special considerations regarding updates of the Nationality field, since inconsistencies between the User and Photo data structures might occur.

Many NoSQL databases are designed to be distributed, which in turn allows increasing their capacity by means of just adding nodes to the infrastructure, a property also known as *horizontal scaling*. In NoSQL databases (as in most distributed database systems), a mechanism often used to achieve horizontal scaling is *sharding*, which involves splitting the data records into several independent partitions or *shards* using a given criterion, e.g. the record ID number. In other cases, the mechanism employed is *replication*, i.e. mirroring data records across several servers, which while not scaling well in terms of data storage capacity, allows increasing throughput and achieving high availability. Both *sharding* and *replication* are orthogonal concepts that can be combined in several ways to provide *horizontal scaling*.

In most implementations, the hardware requirements of individual nodes should not exceed those of a traditional personal computer, in order to reduce the costs of building such systems and also to ease the replacement of faulty nodes.

[2] NoSQL Meetup 2009, http://nosql.eventbrite.com/.

NoSQL databases can be divided into several categories according to the classification proposed in [113,19,67,49], each prescribing a certain *data layout* for the stored data:

- *Key-Value*: These databases allow storing arbitrary data under a key. They work similarly to a conventional hash table, but by distributing keys (and values) among a set of physical nodes.
- *Wide Column or Column Families*: Instead of saving data by row (as in relational databases), this type of databases store data by column. Thus, some rows may not contain part of the columns, offering flexibility in data definition and allowing to apply data compression algorithms per column. Furthermore, columns that are not often queried together can be distributed across different nodes.
- *Document-oriented*: A document is a series of fields with attributes, for example: name="John", lastname="Smith" is a document with 2 fields. Most databases of this type store documents in semi-structured formats such as XML [16] (eXtensible Markup Language), JSON [28] (JavaScript Object Notation) or BSON [77] (Binary JSON). They work similarly to Key-Value databases, but in this case, the key is always a document's ID and the value is a document with a pre-defined, known type (e.g., JSON or XML) that allows queries on the document's fields.

Moreover, some authors also conceive Graph-oriented databases as a fourth category of NoSQL databases [49,113]:

- *Graph-oriented*: These databases aim to store data in a graph-like structure. Data is represented by arcs and vertices, each with its particular attributes. Most Graph-oriented databases enable efficient graph traversal, even when the vertices are on separate physical nodes. Moreover, this type of database has received a lot of attention lately because of its applicability to social data. This attention has brought accompanied new implementations to accommodate with the current market. However, some authors exclude Graph-oriented databases from NoSQL because they do not fully align with the relaxed model constraints normally found in NoSQL implementations [19,67]. In this work, we decided to include Graph-oriented databases because they are essentially non-relational databases and have many applications nowadays [2].

In the following sections we introduce and discuss the most prominent databases in each of the categories mentioned before. Table 1 lists the databases analyzed, grouped by category. Although there are many products with highly variable feature sets, most of the databases are immature compared to RDBMSs, so that a very thorough analysis should be done before choosing a NoSQL solution. Some factors that may guide the adoption of NoSQL as data storage systems are:

- *Data analysis*: In some situations it is necessary to extract knowledge from data stored in a database. Among the

Table 1
Databases analyzed grouped by category.

Category	NoSQL databases analyzed
Key-Value (Section 4)	Hazelcast [48] Redis [97] Membase/Couchbase [26] Riak [11] Voldemort [70] Infinispan [75]
Wide-Column (Section 5)	HBase [40] Hypertable [54] Cassandra [64]
Document-Oriented (Section 6)	CouchDB [68] MongoDB [21] Terrastore [15] RavenDB [50]
Graph-Oriented (Section 7)	Neo4J [80] InfiniteGraph [83] InfoGrid [81] HypergraphDB [55] AllegroGraph [1] BigData [109]

approaches for running jobs over big data stands out MapReduce [31]. In many of these processing frameworks, the developer must code the query in a given imperative programming language. Although this is much more complex than just executing a SELECT… GROUP BY query against the database, it is more suitable for large data volumes. Languages such as Pig[3] and Hive [112] simplify the development of applications with MapReduce, significantly reducing its learning curve.

- *Scalability*: NoSQL databases are designed to store large amounts of data or to support demanding processing by adding new nodes to the system. Additionally, they are usually designed under a "let it crash" philosophy, where nodes are allowed to crash and their replicas are always ready to receive requests. This type of design provides a sense of robustness in terms of system failover capabilities. In some cases, when the crash seems to be permanent, the data is automatically redistributed among the available nodes in the computer cluster.
- *Flexible schema*: The NoSQL databases presented in this paper do not have a fixed schema. Key-Value databases do not make assumptions about the values of keys (except for Redis and Hazelcast which allow users to store lists and sets). Document-oriented and Wide Column databases tolerate discrepancies between fields/rows of elements. In Graph-oriented databases, vertices and arcs can have any structure. Oppositely, relational databases are composed of tables with a fixed scheme and all tuples have the same number of fields.
- *Fast deployment*: In general terms, NoSQL systems can be easily deployed in a cluster. In addition, replication and sharding configuration are usually automatic, speeding up their adoption.
- *Location awareness*: In general, as NoSQL databases are designed to be distributed, the location of the data in the cluster is leveraged to improve network usage, usually by caching remote data, and making queries to those nodes located closer in the network topology (e.g., nodes located in the same local-area network). This mechanism is often referred to as *data affinity*.

Moreover, the choice of a NoSQL database must be based on the type of data to be stored as well as the form of access (read and write). An extreme example would be a Web site that gets millions of hits per second whose data may drastically change to support new functionality. Facebook, Google Search and Amazon are some notable examples. In these systems, data may grow without estimable bounds and, therefore, the system infrastructure must allow increasing storage space without losing performance. In these situations, the use of relational databases is scarce and several non-relational technologies can accommodate these requirements.

It is worth noting that there is a minority of hybrid databases that store more than one data layout. Examples of multi-layout databases are OpenLink Virtuoso [35], OrientDB [82] and AlchemyDB [94]. However, we will not discuss these efforts in a separate section as the mechanisms for supporting the different layouts do not conceptually differ from those offered by single-layout databases.

As an alternative to NoSQL and traditional RDMBSs, new database systems have recently emerged under the name of "NewSQL databases" [104]. NewSQL databases are relational databases supporting sharding, automatic replication and distributed transaction processing, i.e., providing ACID guarantees even across shards. Examples of this type of database are NuoDB,[4] VoltDB[5] and Clustrix[6] As an example, Google's Spanner [25] is a globally distributed database system created at Google that supports distributed transactions, designed as a replacement to Megastore [9], a BigTable-based storage. Nevertheless, like NoSQL databases, NewSQL databases must undergo a strict analysis before being adopted by any organization.

It is worth mentioning that although this work aims at assisting in the selection of a NoSQL database for a given situation, the analysis cannot be reduced only to benchmarking each database in a use case context. This is due to the diversity of techniques, implementations and querying methods that NoSQL databases exhibit, which makes it hard to establish a common ground of comparison. Nonetheless, benchmarking frameworks for NoSQL databases are mentioned in Section 2 since they are useful for understanding the performance of NoSQL databases under different workloads. As such, these frameworks might complement in practice the selection criteria presented in this paper.

[3] Apache Pig Web Site, https://pig.apache.org/.

[4] NuoDB Web Page, http://www.nuodb.com/.
[5] VoltDB Web Page, http://voltdb.com/.
[6] Clustrix Web Page, http://www.clustrix.com/.

The rest of the paper is organized as follows. Section 2 explores related reviews and benchmarking tools used to compare NoSQL databases. Section 3 describes some preliminary concepts that need to be explained before starting with the description and analysis of the different implementations of databases available on the market. Section 4 introduces Key-Value databases. Section 5 presents Wide Column or Column Families databases. Document-oriented and Graph-oriented databases are described in Sections 6 and 7, respectively. Section 8 offers a discussion about the application of NoSQL databases and when they should be considered in the selection of a data storage support. Finally, Section 9 presents conclusions, perspectives on the findings and tools described, as well as future trends in the area.

2. Related works

There are some works listing and comparing NoSQL databases, exposing their virtues and weaknesses. Catell [19] analyzed and compared several databases with respect to their concurrency control methods, where they store data (i.e., in main memory or disk), the replication mechanism used (synchronous or asynchronous), and their transaction support. This comparison includes both commercial and non-commercial databases but does not include Graph-oriented databases, which are generally considered part of NoSQL. As indicated previously, graphs are the essential data layout of Web applications such as social networks [63]. Similarly, Padhy et al. [88] present a comparison of six relevant NoSQL databases, including databases of different type or schema model. They also exclude from their analysis Graph-oriented databases and other relevant implementations of NoSQL databases. Hecht and Jablonski [49] present another comparison between NoSQL databases, including graph databases. The authors focus on data partitioning and replication aspects over 14 NoSQL databases, whereas in this paper we perform a more in-depth analysis over 19 databases.

Other studies [110,107,115] list the most influential NoSQL databases along with their characteristics and basic concepts. However, they do not include a full comparison between databases, since they only expose some of their advantages and disadvantages.

There are other studies that analyze NoSQL databases using a given dataset or application. For example, Sakr et al. [95] carried out a thorough analysis of data stores suited for Cloud Computing environments [18], which includes NoSQL databases. The authors present a set of goals that a data-intensive application should accomplish in the Cloud. They also describe essential structures and algorithms of well-known databases such as BigTable and Dynamo. In addition, they compare several APIs related to massive data query and manipulation.

Another comparison of NoSQL databases in this line is presented by Orend [86]. The ultimate goal of the study was to select a NoSQL solution for a Web Collaboration and Knowledge Management Software. MongoDB, a Document-oriented database, was selected from the available databases because of its support for queries on multiple fields. Then, the study makes a performance comparison of MongoDB against MySQL and HyperSQL.

Todurica and Bucur [113] provide an extensive list of available NoSQL databases, and benchmark two of them – Cassandra and HBase – against MySQL and Sherpa, a variation of MySQL. Their results indicate that, at high load, Cassandra and HBase keep their response latency relatively constant, whereas MySQL and Sherpa increase their response latency. On the other hand, Lith and Mattson [71] present a study based on an application of their own where a MySQL-based approach gives better performance than using a NoSQL solution. In the study, five NoSQL databases were considered. The authors claim that the difference in performance is due to the application data structure and the way it is accessed.

Although this work does not focus on database benchmarking, it is worth mentioning some of the existing benchmarks and benchmarking tools that may complement the analysis carried out in this review. Benchmarks are very important to determine strengths and weaknesses of each database under different stress scenarios and deployment environments. YCSB [23] (Yahoo Cloud Serving Benchmark) is one of the most relevant frameworks for benchmarking both NoSQL and RDBMS databases. It provides an extensible framework for querying databases and a workload generator to benchmark various access patterns. Recent extensions of YCSB can be found in the literature, including the use of distributed clients [89] and support for transactional operations [33]. LinkBench [6] takes a similar approach to YCSB, but targets graph-structured data, in particular, the Facebook social graph. Other efforts explore different types of databases and workloads. For example, HiBench [53] is designed for Hadoop and uses a set of microbenchmarks as well as real-world application workloads. In an attempt to create real-world scenarios, some benchmarks such as BigBench [41] and BigDataBench [117] focus their efforts on querying different data types, such as structured, semi-structured and unstructured data, under diverse types of workload.

3. Background concepts

This section covers background concepts required to understand the decisions taken in the design of NoSQL databases.

3.1. The CAP theorem

A fundamental trade-off that affects any distributed system forces database designers to choose only two out of these three properties: data consistency, system availability or tolerance to network partitions. This intrinsic limitation of distributed systems is known as the CAP theorem, or Brewer's theorem [44], which states that it is impossible to simultaneously guarantee the following properties in a distributed system:

- *Consistency*: If this attribute is met, it is guaranteed that once data are written they are available and up to date for every user using the system.

Table 2
Grouping of NoSQL systems grouped by data layout and CAP properties.

Data layout	AP	CP	AC
Key-Value (Section 4)	Riak, Infinispan, Redis, Voldemort, Hazelcast	Infinispan, Membase/CouchBase, BerkeleyDB, GT.M	Infinispan
Wide Column (Section 5)	Cassandra	HBase, Hypertable	–
Document-oriented (Section 6)	MongoDB, RavenDB , CouchDB, Terrastore	MongoDB	–
Graph-oriented (Section 7)	Neo4J, HypergraphDB, BigData, AllegroGraph, InfoGrid, InfiniteGraph	InfiniteGraph	–

- *Availability*: This property refers to offering the service uninterruptedly and without degradation within a certain percentage of time.
- *Partition Tolerance*: If the system meets this property, then an operation can be completed even when some part of the network fails.

In 2000, Eric Brewer conjectured that at any given moment in time only two out of the three mentioned characteristics can be guaranteed. A few years later, Gilbert and Lynch [44] formalized and proved this conjecture, concluding that only distributed systems accomplishing the following combinations can be created: AP (*Availability-Partition Tolerance*), CP (*Consistency-Partition Tolerance*) or AC (*Availability-Consistency*). Table 2 summarizes NoSQL databases reviewed in this paper organized according to the supported data layout. Within each group, databases are further grouped according to the properties of the CAP theorem they exhibit. As illustrated, most of the surveyed databases fall in the "AP" or the "CP" group. This is because resigning P (Partition Tolerance) in a distributed system means assuming that the underlying network will never drop packages or disconnect, which is not feasible. There are few exceptions to this rule given by NoSQL databases – e.g., Infinispan [75] – that are able to relax "P" while providing "A" and "C". Because some databases, such as Infinispan and MongoDB, can be configured to provide full consistency guarantees (sacrificing some availability) or eventual consistency (providing high availability), they appear in both AP and CP columns.

3.2. ACID and BASE properties

In 1970, Jim Gray proposed the concept of transaction [46], a work unit in a database system that contains a set of operations. For a transaction to behave in a safe manner it should exhibit four main properties: atomicity, consistency, isolation and durability. These properties, known as ACID, increase the complexity of database systems design and even more on distributed databases, which spread out data in multiple partitions throughout a computer network. This feature, however, simplifies the work of the developer by guaranteeing that every operation will leave the database in a consistent state. In this context, operations are susceptible to failures and delays of the network itself. Extra precautions should be taken to guarantee the success of a transaction.

Distributed RDBMSs allow, for some time now, to perform transactions using specific protocols to maintain the consistency of data across the partitions. An example of this type of RDBMs is Megastore [9] a distributed database that supports ACID transactions in specific tables and limited transaction support across different data tables. Megastore is supported by BigTable [20] (Section 5.1), but unlike BigTable, it provides a schema language that supports hierarchical relationships between tables, providing a semi-relational model. Although Megastore did not provide good performance [25] (in comparison to using directly BigTable), several applications needed the simplicity of the schema and the guarantees of replica synchronization. As in many databases, replicas in Megastore are synchronized using a variation of the Paxos algorithm [66]. The idea of hierarchical semi-relational schemas was then used by Spanner, a global-scale database that supports transactions, based on a timestamp API implemented using GPS and atomic clocks. These high-precision timestamps allowed Spanner to reduce the latency of transaction processing. Spanner thus served as the new supporting storage of the revenue-critical AdSense service backend, called F1 [100], which was previously supported by a sharded MySQL-based solution. F1 is a globally consistent and replicated database developed for supporting Google advertising services. F1 is a rather extreme example showing how hard building consistent distributed databases is, while, at the same time, meeting performance requirements.

In these systems, one of the most commonly used protocols for this purpose is 2PC (*Two-phase commit*), which has been instrumental in the execution of transactions in distributed environments. The application of this protocol has spread even to the field of Web Services [27], allowing transactions in REST (REpresentational State Transfer) architectures otherwise not possible [29]. The 2PC protocol consists of two main parts:

1. A stage in which a coordinator component asks to the databases implicated by the transaction to do a *pre-commit* operation. If all of the databases can fulfill the operation, stage 2 takes place. Conversely, if any of the databases rejects the transaction (or fails to respond), then all databases roll back their changes.
2. The coordinator asks the databases to perform a *commit* operation. If any of the databases rejects the *commit*, then a *rollback* of the databases is carried out.

According to the CAP theorem the use of a protocol, such as 2PC (i.e., in a CP system) impacts negatively on system

availability. This means that if a database fails (e.g., due to a hardware malfunction), all transactions performed during the outage will fail. In order to measure the extent of this impact, an operation availability can be calculated as the product of the individual availability of the components involved in such operation. For example, if each database partition has a 99.9% of availability, i.e., 43 min out of service are allowed per month, a *commit* using 2PC over 2 partitions reduces the availability to 99.8%, which is translated to 86 min per month out of service [91].

Additionally, 2PC is a blocking protocol, which means that the databases involved in a transaction cannot be used in parallel while a *commit* is in progress. This increases system latency as the number of transactions occurring simultaneously grows. Because of this, many NoSQL databases approaches decided to relax the consistency restrictions. These approaches are known as BASE (*Basically Available, Soft State, Eventually Consistent*) [91]. The idea behind the systems implementing this concept is to allow partial failures instead of a full system failure, which leads to a perception of a greater system availability.

The design of BASE systems, and in particular BASE NoSQL databases, allows certain operations to be performed leaving the replicas (i.e., copies of the data) in an inconsistent state. As its name indicates, BASE systems prioritize availability by introducing replicated soft state, i.e., each partition may fail and be reconstructed from replicas. Besides, these systems also establish a mechanism to synchronize replicas. Precisely, this mechanism is known as *Eventual Consistency*, a technique that solves inconsistencies based on some criteria that ensures to return to a consistent state. Although Eventual Consistency provides no guarantees that clients will read the same value from all replicas, the bounds for stalled reads are acceptable for many applications considering the latency improvements. Moreover, the expected bounds for stalled reads have been analyzed by Bailis et al. [8]. For example, the Cassandra NoSQL database [64] implements the following high-level update policies:

- *Read-repair*: Inconsistencies are corrected during data reading. This means that writing might leave some inconsistencies behind, which will only be solved after a reading operation. In this process, a coordinator component reads from a set of replicas and, if it finds inconsistent values, then it is responsible for updating those replicas having stale data, slowing the operation. It is worth noticing that conflicts are only resolved for the data involved in the reading operation.
- *Write-repair*: When writing to a set of replicas, the coordinator may find that some replicas are unavailable. Using a write-repair policy, the updates are scheduled to run when the replicas become available.
- *Asynchronous-repair*: The correction is neither part of the reading nor of the writing. Synchronization can be triggered by the elapsed time since the last synchronization, the amount of writes or other event that may indicate that the database is outdated.

In addition to consistency of reads and writes, in distributed storage systems the concept of durability arises, which is the ability of a given system of persisting data even in the presence of failures. This causes data to be written in a number of non-volatile memory devices before informing the success of an operation to a client. In eventually consistent systems there are mechanisms to calibrate the system durability and consistency [116]. Next, we will clarify these concepts through an example. Let N be the number of nodes a key is replicated on, W the number of nodes needed to consider a writing as successful and R the number of nodes where a reading is performed on. Table 3 shows different configurations of W and R as well as the result of applying such configurations. Each value refers to the number of replicas needed to confirm an operation success.

Strong Consistency is reached by fulfilling $W+R>N$, i.e., the set of writings and readings overlaps such that one of the readings always obtain the latest version of a piece of data. Usually, RDBMs have $W=N$, i.e., all replicas are persisted and $R=1$ since any reading will return up-to-date data. Weak Consistency takes place when $W+R\leq N$, in which readings can obtain outdated data. *Eventual Consistency* is a special case of weak consistency in which there are guarantees that if a piece of data is written on the system, it will eventually reach all replicas. This will depend on the network latency, the amount of replicas and the system load, among other factors.

If writing operations need to be faster, they can be performed over a single or a set of nodes with the disadvantage of less durability. If $W=0$, the client perceives a faster writing, but the lowest possible durability since there is no confirmation that the writing was successful. In the case of $W=1$, it is enough that a single node persists the writing for returning to the client, thereby enhancing durability compared to $W=0$. In the same way it is possible to optimize data reading. Setting $R=0$ is not an option, since the same reading confirms the operation. For reading to reach the optimum, $R=1$ can be used. In some

Table 3
Configurations of eventual consistency.

Value	Writing (W)	Reading (R)
0	No confirmation is awaited from any node (it can fail)	N/A
1	A single node confirmation is enough (optimized for writings)	Reading is performed from a single replica (optimized for readings)
M, with M < N (Quorum)	Confirmations of several replicas are awaited	Reading is performed from a given set of replicas (conflicts on the client side might need to be solved)
N (all nodes)	Confirmations of all replicas are awaited (reduces availability, but increases durability)	Reading is performed from all replicas increasing the reading latency

situations (e.g., if using Read-repair), it might be necessary to read from all the nodes, this is $R=N$, and then merge the different versions of the data, slowing down such operation. An intermediate scheme for writing or reading is quorum, in which the operation (reading or writing) is done over a subset of nodes. Frequently, the value used for quorum is $N/2+1$, such that 2 subsequent writings or readings share at least one node.

3.3. Comparison dimensions

In this paper, available NoSQL databases are analyzed and compared across a number of dimensions. The dimensions included in the analysis are expected to help in the decision of which NoSQL database is the most appropriate for a given set of requirements of the user's choice. The dimensions considered are the following:

- *Persistence*: It refers to the method of storing data in non-volatile or persistent devices. Several alternatives are possible, for example, indexes, files, databases and distributed file systems. Another alternative used in some cases is to keep data in RAM and periodically make snapshots of them in persistent media.
- *Replication*: It refers to the replication technique provided by the database to ensure high availability. Data is replicated in different computational nodes as backups in case the original node fails when performing writing operations on the NoSQL database. Replication can also mean an increase in performance when reading or writing from the replicas is permitted and, thus, relieving the load on the original node. Depending on the consistency level desired, the clients may be allowed to read stalled versions of the data from the replicas. A classical example of replication is the Master–Slave replication mechanism, in which a "master" node receives all the write operations from the client and replicates them on the "slave" node. When a client is allowed to write on any replica the mechanism is called Master–Master. Using such scheme may lead to inconsistencies among the different replicas.
- *Sharding*: Data partitioning or *sharding* [73] is a technique for dividing a dataset into different subsets. Each subset is usually assigned to a computing node, so as to distribute the load due to executing operations. There are different ways of sharding. An example would be hashing the data to be stored and divide the hashing space into multiple ranges, each assigned to a *shard*. Another example is to use a particular field from the data schema for driving partitioning. A database that supports sharding allows to decouple the developer from the network topology details, automatically managing node addition or remotion. Thus, sharding creates the illusion of a single super node, while in fact there is a large set of nodes.
  In this paper, the Sharding dimension indicates whether the database supports sharding and how it can be configured. If sharding is not integrated into the database functionality, then the client has to deal with the partitioning of data among nodes.
- *Consistency*: A large number of NoSQL databases are designed to allow concurrent readings and/or writings and, therefore, control mechanisms are used to maintain data integrity without losing performance. This dimension indicates the type of consistency provided by a database (ACID transactions or eventual consistency) and the methods used to access data concurrently.
- *API*: It refers to the type of programming interface used to access the database. In general, NoSQL databases in the Web era allow access through the HTTP protocol. However, accessing a database using an HTTP client is cumbersome for the developer, and hence it is common to find native clients in certain programming languages. This dimension lists the programming languages that have native clients for the current database. In addition, the message format that is used to add or modify items in the database is indicated.
- *Query Method*: It describes the methods to access the database and lists the different ways of accessing these methods through the database API. This dimension indicates the strategies or query languages supported by each database.
- *Implementation Language*: It describes the programming language the database is implemented with. In some cases, it may indicate a preference of the database developer for a particular required technology.

Only databases that persist data on disk, providing a considerable degree of durability, are analyzed in this paper. That is, when a user performs a writing operation on the database, data is eventually stored in a non-volatile device such as a hard disk. Moreover, we considered only databases that aim at increasing performance or storage capacity by adding new nodes to the network. There are several databases that store their data structures in main memory [87,106] and use a persistent, disk-based log as a backup (i.e., in case of power outages, the database must be rebuilt from this backup). This new trend of in-memory databases is often targeted to applications with low-latency requirements, such as real-time applications. However, despite the sustained drop in prices of RAM memory, the gap in costs with hard-disk drives is still noticeable, especially if building a support for large-scale data.

4. Key-Value databases

Databases belonging to this group are, essentially, distributed hash tables that provide at least two operations: get(key) and put(key, value). A Key-Value database maps data items to a key space that is used both for allocating key/value pairs to computers and to efficiently locate a value, given its key. These databases are designed to scale to terabytes or even petabytes as well as millions of simultaneous operations by horizontally adding computers.

A simple example of a key-value store, shown in Fig. 1, is a distributed web content service where each key represents the URL of the element and the value may be anything from PDFs and JPEGs to JSON or XML documents. This way, the application designers may leverage the

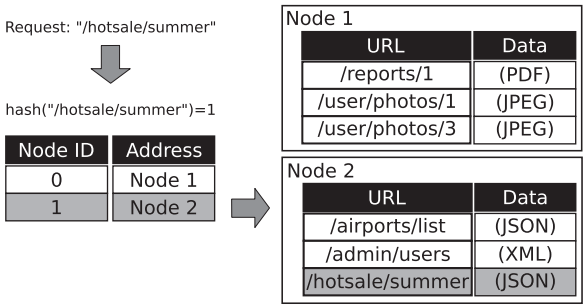


Fig. 1. A simple key-value store example for serving static web content.

nodes available in a cluster of machines to manage a large number of requests and big amounts of web content.

The following sections enumerate some techniques used in Key-Value databases. In Section 4.1 we describe Consistent Hashing, a mechanism frequently used to distribute keys among nodes. In Section 4.2 we describe Virtual Buckets, an alternative mechanism used to distribute keys among nodes. In Section 4.3, we describe a set of techniques used by Dynamo [32], a Key-Value store created by Amazon that influenced several other databases. In Section 4.4 we describe some of the most notorious Key-Value databases available.

4.1. Consistent hashing

Key-Value stores commonly allocate key/value pairs by applying some hashing function to the key and using the result to obtain a specific node of the physical network where the value will be finally stored. The distribution of keys using a standard hashing allocation mechanism (i.e., based on the amount of computers in the network) is sensible to node failures, i.e., if one of the nodes is not available, all the key/value pairs need to be reallocated because of the change in the size of the network. Several databases [32,103,48,11,75] take advantage of the concept of *Consistent Hashing* [60,61] to deal with this situation.

Consistent Hashing models a key space of size K as a circular array or *key ring* (i.e., the successor of key $K-1$ is the 0 key). Thereby, the original key is hashed to fit the $[0, K)$ range and each of the N nodes in the system is assigned a range of contiguous keys. In its simplest form, Consistent Hashing assigns ranges of K/N keys to nodes, although ranges can vary according to each node's characteristics (larger ranges may be assigned to nodes with larger capacity). If one of the nodes in the ring fails, only the keys belonging to such node should be rebalanced, putting them in the next node. Using this scheme, key replication can be performed in the $N-1$ nodes succeeding each node in the *keyring*.

The problem of using physical nodes for partitioning the key space is that successive changes in the network, such as the failure or addition of new nodes may unbalance the distribution of keys among nodes (i.e., how many keys a node is responsible for). This problem is further accentuated when the network has heterogeneous hardware, for example if a node is serving as failover of a node with greater capacity.

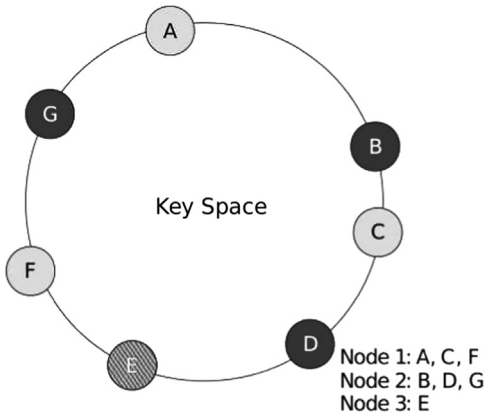


Fig. 2. Example of Consistent Hashing with virtual nodes.

An improvement over this partitioning scheme is to create *virtual nodes*, as proposed in Dynamo's [32] implementation. In this technique, instead of assigning a single contiguous range of keys to each node, a node may be responsible for one or more key ranges that act as virtual nodes in the keyring. In consequence, when one node crashes, its assigned key ranges are distributed among different physical nodes. Similarly, when a new physical node is added, it creates new virtual nodes in the keyring and may receive data from different physical nodes.

Fig. 2 shows an example of using virtual nodes distribution. In the figure it can be seen that the virtual nodes A, C and F are stored in the physical node 1, the virtual nodes B, D and G in the physical node 2 and the virtual node E on the physical node 3. In this example, the virtual node A is responsible for storing the keys in the range (G, A]. Also, the virtual node A is responsible for replicating data about the keys on the precedent $N-1$ ranges. If N equals to 2, A would be responsible for replicating the keys in the range (F, G]. The rest of the virtual nodes behave similarly.

If the physical node 2 fails, then the range of keys (F, G] becomes part of the virtual node A, the range (A, B] moves to the virtual node C and the range (C, D] to the virtual node E. Then, the keyset is distributed among the physical nodes 1 and 3.

4.2. Virtual buckets

Like Consistent Hashing, Virtual Buckets or vBuckets is a technique introduced by Membase/CouchBase [26] to overcome the problem of redistributing keys when a node fails. Similar to Consistent Hashing, vBuckets provide a level of indirection between keys and server addresses. The aim of vBuckets is dividing the key space in a fixed amount of vBuckets (e.g., 4096) and map every vBucket to a server.

From a server perspective, a vBucket can be in one of three states:

- *Available*: The current server contains the data about the vBucket.

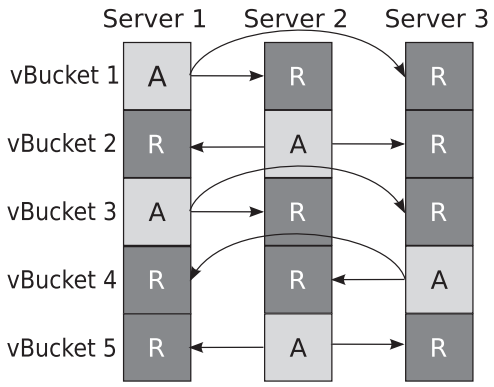


Fig. 3. Virtual Buckets example using 3 servers, a vBucket size of 5 and a 1:*N* replication scheme.

- *Replica*: The current server can only receive replica requests for the current vBucket.
- *Dead*: The server cannot receive requests for the vBucket.

From a client perspective, only one server can serve requests for a given vBucket.

As a result, Available vBuckets can be replicated to other servers by using the corresponding vBuckets marked as *Dead*. In this way, replication can be set in a 1:*N* or a chained configuration. In the first case, every vBucket replicates to *N* servers. On the other hand, in a chained configuration, a vBucket replicates to another server, and in turn, the replicated vBucket has a replica on a third server. Fig. 3 shows a possible vBuckets scenario where the amount of vBuckets is 5 and the replication scheme is 1:*N*. The vBuckets marked as "A" are available vBuckets, whereas vBuckets marked as "R" are replicas.

4.3. Dynamo-based mechanisms

One of the most influential databases in the development of highly scalable Key-Value databases is Amazon's Dynamo [32]. Dynamo was used for the shopping cart and session management service of Amazon, each supporting tens of millions of requests. This database proved to be a highly available system in addition to meet strong latency requirements [32]. Then, Dynamo offers data storage and access mechanisms that represent the inspiration of many existing Key-Value databases.

The most important techniques involved in the implementation of Dynamo are introduced below:

Vector clocks: Dynamo provides eventual consistency, which allows to achieve high availability. Inconsistencies are solved during reading (read-repair), which implies that a writing can return to the client before the actual writing has spread to all replicas. The different values are unified using a versioning scheme known as vector clocks [10,65]. Originally proposed by Leslie Lamport [65] in 1978, a vector clock is a structure containing a list of pairs (physical node, counter). One vector clock is associated with every version of every key/value pair. This versioning scheme allows nodes to discard old versions of the same object (if all counters are less or equal than the current counters), or to reconcile conflicted versions. The latter case arises when the same version of an object is modified in different servers, leading to inconsistencies in the state of that object. For example, if a user modifies the same version of a shopping cart from different clients (e.g. a notebook and a phone), these two modifications may be handled by different servers. Due to eventual consistency, the servers may not be aware of the modifications carried out on other servers, creating two valid versions of the same cart. The reconciliation mechanism can be automatic, e.g. the different versions of a shopping cart can be merged adding all items into a unique cart, which may involve that items deleted by the user appear again in the cart.

Sloppy quorum and hinted handoff: Sloppy quorum persists (or reads) an element on the first *N* available nodes of a preference list when the node's replicas are not available. Then, some copies of the data to be written can be found in nodes that are not replicas of such data. When this situation occurs, the node receives, along with the data, a hint about the copy owner. This mechanism, known as Hinted Handoff, along with Sloppy Quorum, allows the client to return as soon as possible without waiting to persist data in all replicas.

Merkle trees: For persistent failures in which hinted copies cannot return to the original replica, elements of the remaining replicas must be re-synchronized by detecting outdated keys. Comparing the keys one by one according to their hash value can take too long and consume a lot of bandwidth. Dynamo uses a tree structure known as Merkle Tree [78] where each node represents a hash value, calculated starting from their children, which in turn are also hashes. The leaves of the tree are the hash values calculated using the stored keys. This allows a fast comparison of sets of keys, but updating a key range when a node fails can be expensive.

Dynamo implements all the concepts described above to create a highly scalable and available system. However, it is a proprietary system used within Amazon and only accessible through the services provided by the company. There are, nevertheless, several open-source implementations of Dynamo and other Key-Value databases that can be installed in a network. The next section summarizes some of the relevant databases in this line.

4.4. Discussion of Key-Value databases

Table 4 summarizes several relevant Key-Value databases that support persistent storage and distribution. Other databases such as Kyoto Cabinet [36], LevelDB [45], Memcached [38], BerkeleyDB [85] and Scalaris [98] were excluded from this review since they do not meet these requirements.

The listed databases are Riak [11], Infinispan [75], Hazelcast [48], Redis [97], Membase/CouchBase [26] and Voldemort [70]. At the time of writing this review, all of these databases were under active development and heavily employed by some user community. This means that each database is supported by a group of developers releasing periodical updates, fixing bugs and incorporating new functionality. Since adopting a currently active database for a new project assures the existence of

Table 4
Comparison of reviewed Key-Value databases.

Name	Persistence	Replication	Sharding	Consistency	Implementation language	API	Query method
Riak	Bitcask (log-structured store), LevelDB, In-Memory and Multi-backend (different stores for different keys)	Ring (next $N-1$)	Consistent Hashing	Eventual Consistency	Erlang	PBC (Protocol Buffer Client), HTTP, Java, Erlang, C++, PHP, Ruby, Python	Get, MapReduce, Link Walking
Infinispan	Simple File Storage, BerkeleyDB, JDBM, JDBC	Ring (next $N-1$)	Consistent Hashing	Strong Consistency or Eventual Consistency	Java	HTTP, Java	Get, MapReduce, others
Hazelcast	User-defined MapStore, which can be persistent	Ring (next $N-1$)	Consistent Hashing	Strong Consistency	Java	HTTP, Java, C# and any Memcache client	Get, MapReduce
Redis	Snapshots at specified intervals by default or an Append-only file. Both can be combined	Master–Slave (Slave chains can be formed)	No (in charge of the application)	Eventual Consistency	C	Java, C, C#, Ruby, Perl, Scala	Get (also depends on the value structure)
Membase/ CouchBase	SQLite or CouchDB	vBuckets 1:N Replication	vBuckets	Strong Consistency	C/C++, Erlang	Java, C, C#	Get
Voldemort	BerkeleyDB, In-Memory, MySQL	Ring (next $N-1$)	Consistent Hashing	Eventual Consistency	Java	Java, Python	Get

documentation and assistance from those responsible for the database software, persistent and distributed databases whose development has been abandoned were not included in the comparison. The Key-Value databases analyzed are described below:

Riak and voldemort: Among the listed databases, the ones which are more related to Dynamo are Riak and Voldemort since they are direct implementations of the associated Amazon specification [32]. Hence, they use consistent hashing for partitioning and replicating, and provide eventual consistency based on read-repair.

Redis: Redis is distinguished for providing more complex structures such as lists, hash tables, sets and ordered sets for representing values. This functionality makes Redis very similar to Document-oriented databases, which are described in Section 6. However, in Redis, different keys may have different type of values, i.e. one key can refer to a list and the other key can refer to a set. In Document-oriented databases, all values are documents of the same schema. One limitation of Redis is that sharding has to be performed by the client application, which implies that the client must know the network topology to distribute keys among nodes.

Infinispan: Infinispan, formerly known as JBoss Cache, was born as a support tool to scale up Web applications and achieve fault tolerance in the well-known JBoss Application Server. In particular, JBoss Cache was used to replicate and synchronize session state among servers in a cluster of JBoss servers. The main difference with the other databases is that Infinispan has traditionally been biased towards consistency and availability, sacrificing partition tolerance.

Hazelcast: Hazelcast differs from the rest of the reviewed Key-Value databases in that it easily and seamlessly integrates with existing non-distributed Java programs. To achieve this, Hazelcast provides distributed implementations of the typical built-in Java data structures such as List, Vector, Set, and Hashtable. Then, using Hazelcast in a program is just a matter of replacing Java import statements in the client code, and then tuning several parameters such as nodes belonging to the cluster, replication mode, distributed data structures used, etc. Hazelcast does not provide persistence support by default, but allows developers to define their own storage support, which can be persistent instead.

Membase: Membase is a Key-Value database that has been recently merged with the Document-oriented database CouchBase. It uses the vBuckets mechanism to distribute and replicate data across servers. Writing consistency is immediate because writing and reading is always performed on the key's master node (which has the only Available vBucket). Thus, a client always gets the latest value written. One of the most attractive features of CouchBase is its configuration simplicity. Once installed on the nodes, the network can be configured through a friendly Web interface. On the downside, Membase requires manual rebalancing of keys when a server is down and needs to be removed from the cluster. Key rebalancing is a costly operation that depends on the amount of keys handled by the removed server.

Some Key-Value databases maintain a subset of key/value pairs stored in RAM, dramatically improving performance. From the databases analyzed in this review, Hazelcast, Membase, Redis and Riak use this strategy. However, this decision comes with a cost: as the number of keys increases, the use of RAM increases. Riak and Membase always keep keys in memory, whereas values are removed from memory if space is needed for new keys. Hazelcast and Redis keep all key/value pairs in memory and eventually persist them on disk. In all cases, new requests to add keys are rejected when RAM runs out of space. For this reason, it is necessary to take into account whether the number of keys to be stored exceeds the amount of RAM in the network and, if this is the case, chose another alternative or augment the network size.

A further feature to consider in the selection of a database is the expected data durability. The level of durability must be decided according to the importance of the data stored in the network, which sometimes can be configured. Redis is a case of configurable durability. By default, Redis offers the possibility of making data snapshots in memory at time intervals. If a failure occurs during such interval, the current data of the node are lost. For this reason, the database offers to do more frequent writings to disk in a file that only supports appends. Conceptually, this is similar to a log in a log-structured file system. This type of files is often used when high writing throughput needs to be achieved.

The query method varies from database to database, but the Get operation (i.e., get a value by key) is always present. Some alternative query methods are worth mentioning. For example, Riak provides a graph-like querying method called Link Walking. This method consists in creating relationships between keys and tagging each relationship. For example, if there is a relationship tagged “friend” between a key named “Mark” and all its friends’ keys, Riak can be queried using the “friend” tag to obtain all Mark's friends. Other databases provide alternative query methods like Cursors (a well-known structure in SQL), XQuery (an XML Query language) and even MapReduce (Section 5.1.3). Some databases also allow the user to perform “bulk gets”, i.e., getting the values of several keys in a single operation, resulting in considerable performance improvements and network communication savings.

5. Wide column databases

Wide Column or Column Families databases store data by columns and do not impose a rigid scheme to user data. This means that some rows may or may not have columns of a certain type. Moreover, since data stored by column have the same data type, compression algorithms can be used to decrease the required space. It is also possible to do functional partitioning per column so that columns that are frequently accessed are placed in the same physical location.

Most databases in this category are inspired by BigTable [20], an ordered, multidimensional, sparse, distributed and persistent database, that was created by Google to store data in the order of petabytes. Therefore, a brief description of the most important features of BigTable and how it achieves its objectives is given in the following section.

Unlike Key-Value databases, all Wide-Column databases listed in this section are based on BigTable's data scheme or mechanisms. This lack of diversity can be explained by the fact that this type of databases has a very specific objective: to store terabytes of tuples with arbitrary columns in a distributed environment.

5.1. BigTable

BigTable [20] was developed in order to accommodate the information from several Google services: Google Earth, Google Maps and Blogger, among others. These applications use BigTable for different purposes, from high throughput batch job processing, to data provisioning to the end user considering data latency constraints. BigTable does not provide a relational model, which allows the client application to have complete control over data format and arrangement.

In BigTable, all data are arrays of bytes indexed by columns and rows, whose names can be arbitrary strings. In addition, a timestamp dimension is added to each table to store different versions of the data, for example the text of a Web page. Moreover, columns are grouped into sets called column families. For example, the column family *course* can have *Biology* and *Math* columns, which are represented as *course:Biology* and *course:Math*. Column families usually have the same type of data, with the goal of being compressed. Moreover, disk and memory access is optimized and controlled according to column families.

5.1.1. SSTable, tablets and tables

BigTable works on the GFS (Google File System) [42,43] distributed file system and has three types for storage structures: SSTables, Tablets and Tables, which are shown in Fig. 4. SSTable is the most basic structure, which provides an ordered key/value map of byte strings. This basic block consists of a sequence of blocks (usually of 64 KB) and a search index to determine which block is a certain datum, avoiding the unnecessary load of other blocks in memory and decreasing disk operations. Each SSTable is immutable, so no concurrency control is needed for reading. A garbage collector is required to free deleted SSTables.

A set of SSTables is called a Tablet. A Tablet is a structure that groups a range of keys and represents a distribution unit for load balancing. Each table is composed of multiple tablets and, as the table grows, it is divided into more Tablets. The sub-Tables often have a fixed size of 100–200 MB.

The location of each Tablet is stored in the network nodes using a tree structure of three levels, like a B+ tree. First, the file that contains the physical location of the root can be found. The root of the tree is a special Tablet named Root Tablet. The leaves of the tree are called Metadata Tablets and are responsible for storing the location of the user Tablets.

Chubby [17], a distributed lock service, is used to find and access each Tablet. Chubby keeps the location of the

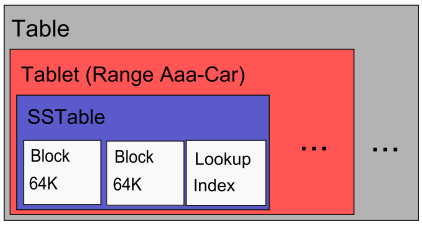


Fig. 4. SSTable, Tablet and Table structures.

Root Tablet, information about the database scheme (the column families and tables) and access checklists. In addition, it synchronizes and detects Tablets nodes (a.k.a. servers) that store Tablets.

A Master Server is in charge of assigning Tablets to Tablet servers. The Master Server monitors the addition and expiration of Tablet servers, balances the load of such servers and performs garbage collection of files stored in the GFS. It also monitors changes in the scheme, i.e., addition of new column families and Tables.

5.1.2. Tablets servers

Fig. 5 depicts how a Tablet is stored on a Tablet server. The updates performed on Tablets belonging to a Tablet Server are stored in a Commit log that saves records to redo committed operations in case that the Tablet Server dies. The most recent records are kept in memory in a buffer known as Memtable.

To obtain a Tablet the server reads from a table called METADATA that contains the list of SSTables that form a Tablet and a set of pointers to the Commit log called redo points. The server then applies the updates carried out starting from the redo points to rebuild the memtable. Finally, to read from a Tablet, the Tablet Server forms a merged view from the set of SSTables and the memtable.

For writing operations, after verifying that the operation is well formed and that the user is authorized (through Chubby), a valid mutation (write, update or delete) is registered in the Commit log. Groups of commits are used to improve the throughput of small mutations. After making the commit, its content is inserted into the memtable. If a certain limit of the memtable is exceeded, a new memtable is created, the previous memtable is transformed into a SSTable.

For reading operations, well-formedness and authorization are also checked, after which the user is presented with a joint view of the SSTables sequence and the memtable. Thus, the latest updates are shown to the user without keeping the last SSTables on disk.

5.1.3. MapReduce

MapReduce is a framework based on the division of labor for parallelizing data-intensive computations on large datasets [31,69]. Basically, this framework consists of two programming abstractions that must be instantiated by programmers: map and reduce. The map function is responsible for processing key-value pairs and generating as output a set of key-value pairs with intermediate results. In turn, the reduce function is responsible for generating a list of results from such intermediate results. As MapReduce consists of programming abstractions, a support materializing them at the middleware level including storage support is needed McCreadie et al. [76].

BigTable (and many other databases) supports passing data to map jobs and storing data from reduce jobs by defining input and output wrappers for MapReduce. This way, MapReduce jobs allow to query and transform data stored in BigTable in parallel, provided that queries can be expressed in this paradigm.

MapReduce queries can be written in an imperative language like Java or C, although even relatively simple queries can often span multiple lines of code. For example, in Java, implementing a distributed summatory using a list of values requires creating a “map” method to divide the list to create jobs, and implementing a “reduce” method that sums the results of the tasks. The task of dividing a list of elements into jobs and summing the results can be easily generalized and offered as a generic operation. For this reason, Google developed Sawzall [90], an interpreted procedural language to act as an interface to MapReduce. Sawzall focuses on providing “syntactic sugar” that allows programmers to implement the *map* function of MapReduce, i.e., query operations, at a higher level of abstraction. *Reduce* functions, i.e., aggregation of intermediate results, are much less varied and thus are usually provided as generic operations. Sum and average are examples of generic *reduce* functions frequently used in MapReduce applications. A simple program in Sawzall to process a file that stores floating point numbers is shown below:

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float=input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

First, the code is interpreted and distributed among the different nodes where the files to be processed are located. The first three lines define how to aggregate results. The table *count* stores the amount of records found, *total* stores the sum of floating point numbers, and *sum_of_squares* stores the sum of squares. The predefined variable *input* holds the input record that, in this case, it is interpreted like a floating point number and stored in the *x* variable. Hence, the last three lines aggregate the intermediate values in the result tables. Usually, aggregations are performed in other “aggregator” nodes that receive intermediate data and combine them. Finally, when all records have been read the values are displayed or stored in a file.

5.2. List of Wide-Column databases

BigTable is a system used internally by Google, i.e., the community has no access to its source code or executables. However, several open source alternatives offering similar services were developed based on the academic publications made by the company.

Table 5 summarizes these alternatives. The table compares the HBase [40], Hypertable [54] and Cassandra [64] databases. Following, we further describe them:

HBase and hypertable: HBase belongs to the Apache Software Foundation and it is based directly on BigTable. The storage support is HDFS (Hadoop Distributed File

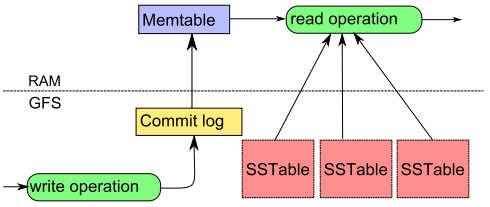


Fig. 5. Operating diagram of a Tablet server.

System) [101], which in turn is based on GFS. Hypertable is also modeled based on BigTable, but implemented in C++. Like HBase, Hypertable relies on HDFS for storage and replication. Both databases allow consulting the database using Hadoop MapReduce [101], an open source implementation of Google MapReduce. Furthermore, querying is done through languages similar to Sawzall, such as Pig [39] and Hive [112]. Querying can also be combined with a workflow scheduler system like Apache Oozie [56], which allows the creation of DAGs (Directed Acyclic Graphs) of jobs that obtain data from HDFS.

Cassandra: Cassandra, which recently moved to the Apache Software Foundation, uses columns and columns families to model data, but uses Dynamo mechanisms to manage storage and replication, namely Consistent Hashing, Read-Repair, Vector Clocks, Gossip Protocol, among others. Cassandra does not have a single point of failure because of its peer-to-peer architecture, which can be considered an advantage. Hypertable or HBase, both based on HDFS, have a single point of failure, the so-called NameNode. This component is a Master Server that manages the file system namespace and controls its access from the clients. The drawback is that this component is unique for the entire file system. NameNode replication can be done by any software that can copy all disk writings to a mirror node. An example of this type of software is DRBD (Distributed Replicated Block Device), a distributed mirroring system analogous to a RAID-1 array [34]. Another feature of Cassandra is the possibility to choose between two partitioning schemes: Order Preserving Partitioning and Random Partitioning. Order Preserving Partitioning distributes keys in a ring preserving their order. This allows to efficiently perform range queries, i.e., obtain consecutive keys. However, this partitioning scheme tends to unbalance load across nodes, e.g., frequent write operations on a key range may fall on the same node. This problem translates into an administrative overhead to try to distribute key ranges according to their access patterns. Regarding querying, Cassandra offers a SQL-Like query language called CQL (Cassandra Query Language) and Hadoop MapReduce for distributed job processing. The Hadoop support also extends to Pig and Hive query languages, and Apache Oozie.

Some of the surveyed databases in this paper provide native clients in many languages with the help of an interface definition language, which specifies the service interface and data types, and a code-generation software, which generates clients and servers in a specified language. Thrift [102] is an example of such a tool. It allows to specify service interfaces and data types in the Thrift IDL (Interface Definition Language) and, using a set of code-generation tools, create native

Table 5
Characteristics of Wide Column databases.

Name	Persistence	Replication	Sharding	Consistency	Implementation language	API	Query method
HBase	HDFS (Hadoop File System)	HDFS replication	By key ranges	Strong Consistency	Java	Java, HTTP + JSON, Avro, Thrift [102]	Hadoop MapReduce, Pig, Hive
Hypertable	HDFS by default (other supports are available)	HDFS replication	By key ranges	Strong Consistency	C++	Thrift	HQL (Hypertable Query Language), Hadoop MapReduce, Hive, Pig
Cassandra	Proprietary format	Ring (next $N-1$)	Consistent Hashing	Eventual Consistency	Java	Thrift	CQL (Cassandra Query Language), Hadoop MapReduce, Pig, Hive

RPC clients and servers in languages such as Java, Ruby or Python, among others. Avro [3] is a similar tool to Thrift. In this case, Avro provides its own IDL, but supports JSON to specify interfaces and data types.

It is notable the reduced number of Wide Column databases available with respect to other types of NoSQL databases. In principle, this can be attributed to two reasons. First, the complexity in the development of such databases is considerably high. Considering Bigtable, for example, a storage medium such as GFS, a distributed lock server as Chubby and a Table server similar to the Master Server are required. Second, the application domain of Wide Column databases is limited to particular problems: data to be stored need to be structured and potentially reaching the order of petabytes, but search can only be done through the primary key, i.e., the ID of the row. Queries on certain columns are not possible as this would imply to have an index on the entire dataset or traverse it.

6. Document-oriented databases

Document-oriented or Document-based databases can be seen as Key-Value databases where the value to store has a known structure determined at database design time. As a consequence, the architecture of these systems is based on several concepts, some of them defined earlier in this work, to achieve scalability, consistency and availability. In contrast to Wide-Column and Key-Value databases, there is no reference Document-oriented database design (like BigTable or Dynamo), which reflects in the diversity of techniques and technologies applied by database vendors.

In this context, documents are understood as semi-structured data types, i.e., they are not fully structured as tables in a relational database, but new fields can be added to the structure of each document according to certain rules. These rules are specified in the standard format or encoding of the stored documents. Some popular document formats are XML, JSON and BSON. Like Wide Column databases, Document-oriented ones are schemaless, i.e., they have no predefined schema data to conform with. Then, the number and type of fields in the documents in the same database may be different.

Since the database knows the type of data stored, operations that are not available in traditional Key-Value databases become possible in document-oriented databases. Among these operations we can mention add and delete value fields, modify certain fields and query the database by fields. If a Key-Value database is used to store documents, a “document” represents values, and the addition, deletion and modification of fields imply replacing the entire document for a new document. Instead, a Document-oriented database can directly access the fields to carry out the operations.

In a Key-Value database, queries are performed by providing one or more keys as an input. In Document-oriented databases queries can be done on any field using patterns. Then, ranges, logical operators, wildcards, and more, can be used in queries. The drawback is that for each type of query a new index needs to be created, because document databases index elements by the document identifier. In general, the document identifier is a number that uniquely identifies a document in the database.

A Document-oriented database is useful when the number of fields cannot be fully determined at application design time. For example, in an image management system, a document could be written in JSON format as follows:

```
{
  route: /usr/images/img.png,
  owner:
  {
    name: Alejandro
    surname: Corbellini
  },
  tags: ["sea", "beach"]
}
```

New features can be added as the system evolves. For example, when updating the previous document with the ability to access the image owner's Web site, a checksum of the image and user ratings, the above document would become:

```
{
  route: /usr/images/img.png,
  owner:
  {
    name: Alejandro
    surname: Corbellini
    web: http://www.alejandrocorbellini.com.ar
  },
  tags: ["sea", "beach"],
  md5: 123456789abcdef123456789abcdef12,
  ratings:
  {
    {
      user: John Doe
      comment: "Very good!"
      stars: 4
    },
    {
      user: Jane Doe
      comment: "Bad Illumination"
      stars: 1
    }
  }
}
```

New tables must be created to achieve the same goal in a relational database, which can lead to perform joins between tables containing numerous rows or modify the existing table schema, upgrading all rows in the database.

Table 6 summarizes the most representative NoSQL Document-oriented databases: CouchDB [68], MongoDB [21], Terrastore [15] and RavenDB [50]. Following, we describe them:

CouchDB: CouchDB is a database maintained by the Apache Foundation and supported mostly by two companies: Cloudant and CouchBase. This database uses Multiple Version Concurrency Control (MVCC) [12] to provide concurrent access to stored documents. This mechanism allows multiple versions of a document to co-exist in the database, similar to the *branch* concept in a Version Control System (VCS). Each user editing a document receives a snapshot of the current document and after working on it, a version is saved with the most recent timestamp. Earlier versions are not deleted so that readers can continue

Table 6
Comparison of different document-oriented databases.

Name	Persistence	Replication	Sharding	Consistency	Implementation language	API	Query method
CouchDB	CouchDB Storage Engine (B-Tree)	Master–Master	No, but there are extensions to CouchDB that allow sharding	Eventual Consistency	Erlang	HTTP + JSON, and clients for different languages (including Java)	Views using JavaScript + MapReduce
MongoDB	BSON Objects or GridFS for big files	Replica Sets (sets of Master–Slaves) or simply Master–Slave	By field, which can be any field in a document collection	Strict Consistency by default, but can be relaxed to Eventual Consistency	C++	Mongo Wire Protocol + BSON, HTTP + JSON, and clients for most languages	Queries per field, Cursors and MapReduce
Terrastore	Terrastore storing support	Master–Slave with *N* replicas in hot-standby	Consistent Hashing	Eventual Consistency	Java	HTTP + JSON, and clients for some languages (Java, Clojure [51], Scala [84])	Conditional queries, queries by range Predicates, MapReduce
RavenDB	Microsoft's ESE (Extensible Storage Engine)	Master–Slave on *N*-replicas	Allows the user to define a sharding function based on the documents' fields	Eventual Consistency	C#	HTTP + JSON, .Net	LINQ

accessing them. When a reader wants to access the document, the database resolves which is the newest version using the timestamps. This flexibility comes at an extra cost of storage space and has the disadvantage that conflicts between versions of documents might arise. The last issue is usually solved by alerting the client that is trying to write a conflicting version, just like a VCS would. Unlike a VCS, the database must ensure obsolete documents are periodically cleaned, i.e., those that are not in use and correspond to older versions. CouchDB provides ACID transactions only per document, i.e., each operation on a document is atomic, consistent, complete and durable. This is achieved by serializing operations made by clients and never overwriting documents on disk. Replication follows the Master–Master model, i.e., the replicas also serve requests from the clients, both for writing and reading. The goal is that server nodes can be distributed among different networks and clients can write or read from the closest servers. Updates between replicas are bidirectional and, if there are network failures, synchronization waits for the connectivity to be reestablished. This replication approach may result in clients reading old documents from replicas that have not received updates yet. For querying documents, CouchDB uses the concept of views, which is borrowed from RDBMSs. These structures are defined in JavaScript and enable to display structured contents starting from documents. The code of a view is equivalent to the map function of MapReduce, but it is not done in a distributed manner. To overcome this limitation, there are extensions to use CouchDB in a cluster, facilitating the addition and remotion of nodes. As a result, the end user has the illusion that there is only one node. Freely available extensions for distributing CouchDB are BigCouch [22], Lounge [37] and Pillow [52]. Additionally, some of these extensions provide an automatic sharding technique such as Consistent Hashing (Section 4.1).

MongoDB: MongoDB is a free Document-oriented database that runs on a wide range of platforms. It is developed and maintained by the 10gen company.[7] This database implements different techniques for storing documents. Firstly, documents are encoded in BSON, a binary version JSON. BSON provides faster reading and less space usage than JSON. To achieve the former goal, BSON uses prefixes that indicate the size of each element and its position. BSON documents have the disadvantage of having a space limit up to 16 MB. BSON files larger than 16 MB are stored in GridFS [13], a distributed file system that allows large files to be split into smaller parts to access them separately, to deal with big files in parallel. Regarding data sharding, MongoDB integrates the functionality for distributing data and queries through different nodes. For this purpose, the database uses a router for queries, called Mongo, which evenly distributes queries to nodes to balance the cluster load. For simpler queries, MongoDB provides an API *find*() that uses BSON documents to highlight where the fields and query values match. These documents are traversed using a cursor allowing to visit each document matching the query as in a RDBMS. For

[7] 10gen Web Page, http://www.10gen.com/.

more advanced queries, that require to group data belonging to multiple documents, MongoDB can run MapReduce jobs. Replication in MongoDB can be achieved through the classic Master–Slave scheme but it also introduces an alternative known as Replica Sets. Like in a Master–Slave schema, Replica Sets allow replicated nodes to be grouped while being one of them the primary node and the remaining are secondary ones, but also offer fail-over and automatic fail recovery. Optionally, reading can be done from the secondary nodes balancing the load over the Replica set, but reducing the consistency level with respect to the primary node. The goal of the Replica Sets is improving the plain Master–Slave scheme, easing cluster maintenance. For supporting concurrency, MongoDB provides atomic operations per document which means that, when a document is updated using atomic operations, the database ensures that the operation will succeed or fail without leaving inconsistencies. Unlike CouchDB, MongoDB does not provides MVCC, whereby readings can be blocked until the atomic operations are completed.

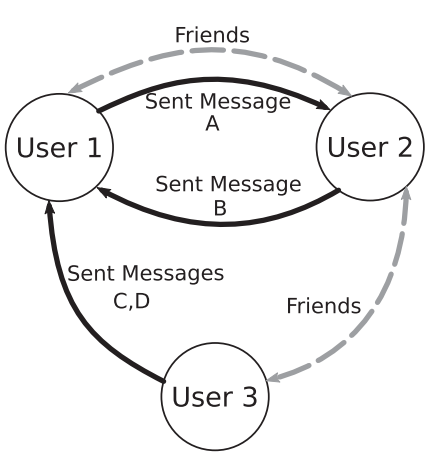
Terrastore: Another alternative for storing documents in a distributed manner is Terrastore. This database is based on Terracotta [111], a distributed application framework based on Java. Documents must follow the JSON notation and can be directly accessed both via HTTP or specific clients in Java, Clojure and Scala, among others. Like MongoDB, Terrastore has integrated sharding support to transparently add and remove nodes. Replication is done using Master–Slave, where slaves are kept in hot-standby, i.e., they can replace the master at any time if it fails. Terracotta operations are consistent at document level and concurrency is handled with the read-committed strategy. Write-locks are used throughout a write transaction, but readings are only blocked for each query or SELECT within the transaction allowing to alternate writing and reading operations. As a result of this strategy, it may be possible that during a transaction a reading returns a value, the document is modified and then the same reading within the transaction returns a different value.

RavenDB: Finally, RavenDB is an alternative developed on the .Net platform. Although implemented in C#, it provides an HTTP interface to access the database from other languages. ACID transactions are supported, but RavenDB is based on an optimistic transaction scheme to avoid the use of locks. To access stored documents RavenDB allows defining indexes through LINQ queries, a query language developed by Microsoft with a syntax similar to SQL. LINQ queries can define free fields that can be passed as parameters by the user, filtering the indexed documents. A feature that differentiates RavenDB from other Document-oriented databases is the mechanism to configure the sharding of documents. Albeit sharding support is integrated, it is not automatic. The database client must define the shards and strategies to distribute the documents across the different network nodes. One of the major drawbacks of RavenDB is the requirement of a paid license for use in commercial products or services.

7. Graph-oriented databases

Nowadays, graphs are used for representing many large real-world entities [63] such as maps and social networks. For example, OpenStreetMap, an open geographic data repository maintained by a huge user community, reached more than 1800 million nodes in 2013. On the other hand, Twitter has experienced a tremendous growth [5], and nowadays has more than 200 million active users tweeting 400 million tweets per day and supporting billions of follower/followed relationships. The interest for storing large graphs and, more importantly, querying them efficiently has resulted in a wide spectrum of NoSQL databases known as Graph-oriented or Graph-based databases. These databases address the problem of storing data and their relationships as a graph, allowing to query and navigate it efficiently. Similar to Document-oriented databases, there is no Graph-oriented database that can be used as a "reference design".

Browsing graphs stored in a large RDBMS is expensive because each movement through the edges implies a join operation. Usually, Graph-oriented databases represent data and their relationships in a natural way, i.e., using a



Vertex Data

Vertex	Name	Email
1	John	john@...
2	Bob	bob@...
3	Alice	alice@...

Friends

Origin	Dest.
1	2
2	1,3
3	2

Messages Sent

Origin	Dest.
1	2
2	1
3	1

Messages Rcvd

Origin	Dest.
1	2
1	3
2	1

Message Data

Vertices	Data
1-2	A
2-1	B
3-1	C,D

Fig. 6. Example of vertex representation in a Graph-oriented database.

structure that contains the vertex information and a list of pointers to other vertices. Fig. 6 shows an example where users in a system may have friendship relationships (i.e., undirected edges) and send messages to each other (i.e., directed edges). Undirected edges are commutative and can be stored once, whereas directed edges are often stored in two different structures in order to provide faster access for queries in different directions. Additionally, vertices and edges have specific properties associated to them and thereby, these properties must be stored in separate structures. In this example, a simple sharding policy may involve splitting each table using the vertex ID in order to keep all data related to a single vertex in a single machine.

In addition to the interest for storing graphs, recent studies have addressed the problem of efficiently processing large graphs stored in a distributed environment. In this context, distributed graph processing presents a challenge in terms of job parallelization. For example, in [24], we propose a framework for storing Twitter adjacency lists and query them to efficiently provide recommendations of users to follow. In our experiments, storing 1.4 billion of relationships among 40 million users proved to be challenging in terms of storage and distributed graph algorithm design.

Generally, graph algorithms are dictated by the arc–vertex pairs of the graph being processed, i.e., the execution of the algorithm depends on the structure of the graph. Additionally, a partitioning strategy is difficult to express in source code because the structure of the graph (unlike lists or hashes) is not known *a priori*. This also directly affects the computing locality, especially in distributed environments. Furthermore, although parallelism can be achieved, the algorithms do not perform too much processing on the vertices, but focus on traversing the graph through their arcs. This generates a significant communication overhead when the vertices are located in different computing nodes [72]. The former is one of the greatest pitfalls in the application of MapReduce-based frameworks in graph processing [59]. The problems of MapReduce on graphs have led to the creation of new processing models and frameworks such as Pregel [74], Apache Giraph (based on Pregel) [4], Trinity [99], HipG [63] or Mizan (also based on Pregel) [62]. However, graph processing frameworks store all data in memory, i.e., they obtain data from files on disk and load the structure of the graph on each node in RAM to perform processing afterwards. Once processing is completed, updates made on the graph residing in main memory are not persisted. In this survey, special attention is paid to those databases that allow persisting data and, therefore, frameworks for intensive graph computing are not included in the comparison.

Then, Table 7 presents a comparison between the following Graph-oriented databases: Neo4J [80], InfiniteGraph [83], InfoGrid [81], HyperGraph [55], AllegroGraph [1] and BigData [109]. As mentioned before, open source or free distributed Graph-oriented databases are presented, limited to those providing some type of persistence and bringing a certain durability. Furthermore, we provide a description of the Graph-oriented databases analyzed:

Neo4J: Neo4J is an open source project that uses Lucene indexes to store data so that access to vertices and

Table 7
Comparison of graph databases.

Name	Persistence	Replication	Sharding	Consistency	Implementation language	API	Query method
Neo4J	Indexes on disk (Apache Lucene by default)	Master–Slave	Manual	Eventual Consistency	Java	Java, HTTP + JSON, bindings in Ruby, Clojure, Python, among others	SPARQL (RDF and OWL), Java API, Gremlin
InfiniteGraph	Objectivity/DB	Synchronous replication of Objectivity/DB	Rule based sharding	Strong Consistency or Eventual Consistency	C++	C++, Java, Python and C#	API for graph traversing and Predicate Queries
InfoGrid	MySQL, HadoopFS, among others	Peer-to-Peer (XPRISO protocol)	Manual	Eventual Consistency	Java	Java, HTTP + JSON	Viewlets, Templates HTML, Java
HypergraphDB	2 tiers: Primitive (Raw Data) and Model (relations + caching + indexes)	Peer-to-Peer (Agent-based)	Manual	Eventual consistency	Java	Java	Java API, Prolog, OWL, RDF via Sesame
BigData	Indexes (B+ trees)	Master–Slave	"Dynamic" sharding (by key-range shards, also called index partitions)	Eventual Consistency	Java	Java and service discovery through JNI	SPARQL, RDFS++
AllegroGraph	Indexes	Master–Slave (Warm Standby)	Manual	Eventual Consistency	Lisp	HTTP + JSON and clients in several languages (included Java)	SPARQL, Prolog, RDFS++, and graph traversal through API

relationships is more efficient. Neo4J also uses an MVCC mechanism with a read-committed strategy to increase reading and writing concurrency. These decisions enable the storage of millions of vertices in a single computational node. In Neo4J, sharding of the vertices across nodes is done manually (by the developer) using domain-specific knowledge and access patterns. Additionally, a periodical defragmentation (vertex relocation) using rules has been proposed but not implemented yet.[8] A Neo4J graph is accessed via a Java API or graph query languages such as SparQL [92] or Gremlin [93]. SparQL is a language used for querying data stored in RDF (Resource Description Framework) format, a metadata data model originally created by the World Wide Web Consortium (W3C) to describe resources (i.e., adding semantic information) on the Web. In general, RDF data are stored as 3-tuples indicating a subject, a predicate and an object. Intrinsically, it represents a labeled directed graph: the source vertex, the relationship type and the destination vertex. Gremlin is a language for doing graph traversal over graphs stored in various formats. In addition to a free version, Neo4J has enterprise versions that add monitoring, online backup and high availability clustering. The Neo4J module that enables to define a Master–Slaves node structure is only available in the paid version.

InfiniteGraph: InfiniteGraph has its own storage media called Objectivity/DB. Unlike Neo4J, it features automatic sharding using "managed placement" to distribute a graph over a cluster. Managed placement allows the user to define rules for custom sharding and, for example, keep related vertices close to each other. Unfortunately, the free license allows storing just 1 million edges and vertices.

InfoGrid: InfoGrid is an open-source storage based on structures known as NetMeshBase, which contains the vertices of the graph, called MeshObjects, and its relationships. It can be persisted using a RDBMS like MySQL or PostgresSQL, or using a distributed file system like HDFS or Amazon S3. If a distributed file system like HDFS is used, the benefits of availability and performance of the system can be attained. Furthermore, NetMeshBase structures can communicate with each other so that graphs can be distributed among different clusters.

HyperGraphDB: HyperGraphDB [55] introduces a different approach for representing stored data through the use of hypergraphs. A hypergraph defines an *n*-ary relation between different vertices of a graph. This reduces the number of connections needed to connect the vertices and provides a more natural way of relating the nodes in a graph. An example of a hypergraph is the relationship *borderlines*, where nodes such as *Poland*, *Germany* and *Czech Republic* can be added. The database only requires a hypergraph containing these vertices to store the relationship, whereas a graph representation uses three arcs among the nodes. For storage, HyperGraphDB relies on BerkeleyDB [85], providing two layers of abstraction over it: a primitive layer, which includes a graph of relations between vertices, and a model layer, which includes the relations among the primitive layers adding also indexes and caches. These abstractions allow defining different graph interpretations, including RDF, OWL (an extension to RDF that allows to create ontologies upon RDF data), a Java API and a Prolog API, among others.

BigData: Finally, BigData is an RDF database scalable to large numbers of vertices and edges. It relies on a log-structured storage and the addition of B+ indexes, which are partitioned as the amount of data increases. For single node configurations, BigData can hold up to 50 billion vertices or arcs without sacrificing performance. If vertices need to be distributed, the database provides a simple dynamic sharding mechanism that consists in partitioning RDF indexes and distributing them across different nodes. Moreover, it provides the possibility of replicating nodes with the Master–Slave mechanism. BigData offers an API for SparQL and RDFS++ queries. The latter is an extension of RDFS (Resource Description Framework Schema), a set of classes or descriptions for defining ontologies in RDF databases and to make inferences about the data stored.

AllegroGraph: AllegroGraph is a RDF Store that supports ACID transactions marketed by a company named Franz Inc., which offers a free version limited to 3 million RDF triplets in the database. Like HyperGraphDB, this database has a very wide range of query methods including: SPARQL, RDFS++, OWL, Prolog and native APIs for Java, Python, C# among others.

In addition to the listed databases, there are other similar *application-specific* databases that deserve mention. One is FlockDB [114], a Graph-oriented database developed by Twitter. This is a database with a very simple design since it just stores the followers of the users, i.e., their adjacency list. Among its most important features are its horizontal scaling capabilities and the ability to perform automatic sharding. Although it was not officially announced by Twitter, the FlockDB project was abandoned and it may have been replaced by other solution, such as Manhattan,[9] a distributed database also built by Twitter, or Cassandra.

A second database worth mentioning is Graphd [79], the storage support of FreeBase [14], a collaborative database that stores information about movies, arts and sports, among others. Graphd storage medium is an append-only file in which tuples are written. Each tuple can define either a node or a relationship between nodes. Tuples are never overwritten, when a new tuple is created the modified tuple is marked as deleted. Inverted indexes are used to accelerate access, going directly to the positions of tuples in the file. MQL (Metaweb Query Language), a language of Freebase analogous to SparQL, is used for querying the database. Graphd is a proprietary system, therefore, it was not included in the comparison. However, access to the database query API is available on the Web.[10]

Finally, in [96] the authors propose an extension to SPARQL, named G-SPARQL, and, more important to this work, an hybrid storage approach in which the graph structure and its attributes are stored in a relational

[8] On Sharding Graph Databases, http://jim.webber.name/2011/02/on-sharding-graph-databases/.

[9] Manhattan announced in Twitter Blog,https://www.blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.

[10] http://www.freebase.com/queryeditor.

database. Queries on the graph attributes are executed on the relational database whereas topological queries are executed on an in-memory graph representation. This strategy avoids the performance penalty of making recursive join operations on the database tables while still benefiting from the querying and storage efficiency of a relational database. However, the authors do not mention a distributed variant of the mentioned approach.

8. Discussion

As can be observed in this review, the spectrum of NoSQL databases is very broad and each of them is used for different applications today. However, NoSQL solutions cannot be seen as the law of the instrument or Maslow's Hammer[11] and should not be used for every application. In fact, many alternatives could be in general considered when selecting a storage solution. At the other extreme of the above example are the majority of low and medium scale applications, such as desktop applications and low traffic Web sites where stored data is in the order of megabytes, up to gigabytes, and do not have higher performance requirements. The use of relational databases can easily overcome the storage requirements in those situations. Nevertheless, the simplicity of the API of a Document-oriented database or a Key-Value database may also be a good fit.

Furthermore, in some situations, it may be necessary to define a hybrid data layer dividing the application data in multiple databases with different data layouts. For example, application data requiring a relational schema and high consistency, such as data from user accounts, can be stored in a RDBMS. On the other hand, if there is data of instant messaging between users requiring no transaction consistency but a fast access is necessary, a NoSQL database can provide an adequate solution.

Therefore, the goal of this review was to better understand the characteristics of the different types of NoSQL databases available. Particularly, in this paper we have reviewed NoSQL databases that support sharding and persist data storage. The aim of these restrictions was to compare databases that can be used as horizontally scalable data stores. This excludes many other storage solutions including: (1) databases that cannot be distributed, (2) in-memory stores, which are usually used as caches, and (3) distributed processing frameworks that generate a temporal (in-memory) representation of data extracted from a secondary database. On one hand, the distribution or sharding of data between different computing nodes allows the user to increase the storage capacity just by adding new nodes. Moreover, many NoSQL databases use this distribution to parallelize data processing among nodes having relevant data for the execution. On the other hand, data persistence is essential when the nodes of the cluster may suffer electric power outages.

[11] Abraham Maslow: "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

Broadly speaking, for NoSQL systems like BigTable, the main design problems to solve are consistency management and fast access to large amounts of data. In this case, the database must scale to petabytes of data running on standard hardware. Contrarily, relational databases scale with certain difficulty because their latency is dramatically affected with each new node addition.

Some studies have shown that relaxing consistency can benefit system performance and availability. An example of this situation is a case study in the context of Dynamo [32], which models the shopping cart of an ecommerce site. Dynamo premise is that an operation "Add to cart" can never be rejected as this operation is critical to the business success. It might happen that a user is adding products to the shopping cart and the server saving the information suffers a failure and becomes no longer available. Then, the user can keep adding products on another server, but the shopping cart version of the original server was distributed to other replicas, generating different versions. In scenarios where faults are normal, multiple cart versions can coexist in the system. Versions are differentiated in Dynamo using Vector Clocks (Section 4.3). However, branching of versions can lead to several shopping carts, possibly valid, but with different products. To solve this conflict, Dynamo tries to unify the cart versions by merging them into a new cart that contains all user products, even if the new version contains previously deleted products.

Compared to these supports, relational databases, alternatively, provide much simpler mechanisms to manage data updates maintaining consistency between tables. Instead, NoSQL databases with eventual consistency delegate the problem of solving inconsistencies to developers, which causes such functionality to be error-prone.

Document-oriented databases are useful for semi-structured data without a fixed schema, but complying to certain formatting rules, such as XML, JSON, and BSON documents, among others. The goal of these databases is to store large amounts of text and provide support for querying on their fields, which in turn are indexed in different ways (keywords, exact word, numeric, etc.). A typical example application of such databases is text mining, where recognized elements of a textual document can vary in type and quantity. For example, a document may be composed of the syntactic structure of the text and also entities such as cities, names and people. xTAS [30] is a project that examines multilingual text and stores the internal results in MongoDB.

Moreover, there are applications that have large amounts of highly correlated data, with diverse relationships. Usually, this type of data is extracted from social networks or the Semantic Web. Graph-oriented databases are more suitable for such data as they allow to efficiently explore vertex-to-vertex relationships. Furthermore, there are also graph processing frameworks for executing distributed algorithms, although they are not usually designed to store data. Examples of such frameworks are Pregel [74], Trinity [99] and HipG [63]. These frameworks, which are to NoSQL-based applications what the model layer represents to conventional Web applications, provide a middleware layer on top of the data store layer where the logic related to data traversal and parallelism resides.

In general, it is not necessary to know SQL for using NoSQL databases. However, since each NoSQL database has its own API, consistency, replication and sharding mechanisms, every change of NoSQL technology involves learning a new storage paradigm. In some cases, such as Key-Value databases, this learning is fast as the API can be very easy to learn (get and set), but in other cases, such as Wide-Column databases, it involves learning concepts such as Column Families and MapReduce. Some NoSQL databases provide SQL-Like languages for querying data so this transition is less drastic. There are also efforts in creating a unified API to query different NoSQL databases with different types of schema. For example, SOS (Save Our Systems) [7] provides a unified API consisting in 3 basic operations: GET, PUT and DELETE. The implementation of those operations depends on the database selected (SOS was tested against MongoDB, Redis and HBase). By unifying the API for different databases, the application code can be reused for a different type of database. However, hiding the specifics of the underlying database also hides its features and, therefore, possible optimization opportunities.

9. Conclusions

NoSQL databases are now part of the software designer's toolbox and are relentlessly occupying a market niche once completely owned by RDBMSs. Currently, the “NoSQL movement” is going through a hype where the technology is receiving substantial attention and the expectations on it may be exaggerated. This is, in part, due to the proliferation of NoSQL databases created by many companies and open-source communities, each of which promoting its own implementation. This also makes using NoSQL technology in a production environment a tough decision. However, there is plenty of community support and, in many cases, official support can be acquired.

As mentioned, NoSQL databases have been adopted by many organizations, including organizations that provide storage solutions themselves. There are some databases that use NoSQL databases as the underlying storage support. For example, Titan[12] is a graph-oriented database that allows the user to choose from three underlying storage supports: Cassandra, BerkeleyDB and HBASE. This in principle suggests that in the future, different data layouts at both the representation and the storage level might coexist, thus increasing the available options for developers.

There is also a growing number of Cloud storage systems that use NoSQL databases as storage support. As an example, Riak CS[13] is a file storage system on the Cloud that uses Riak as a persistent storage and provides an API (similar to Amazon S3[14]) to store files containing terabytes of data. Another kind of Cloud storage systems that is being increasingly adopted is database-as-a-service (DBaaS) – a.k.a. data as a service – systems [47]. A DBaaS is a database, installed and maintained by a vendor, in which developers can store data usually in exchange for a fee. A number of DBaaS providers are based on NoSQL databases. For example, OpenRedis[15] provides a hosted Redis database that can be purchased and accessed remotely. Another example is MongoLab,[16] a DBaaS based on MongoDB. As stated in [47], DBaaS presents security challenges if appropriate strategies are not implemented. Data encryption and third party data confidentiality are examples of security concerns inherent to DBaaS.

In many PaaS (Platform as a Service) environments the databases offered range from RDBMSs to NoSQL databases. For example, PaaS vendors such as Heroku[17] run applications written in a variety of languages and frameworks. As a storage backend, a Heroku user can choose from a set of databases including Postgres, ClearDB (a MySQL-based distributed database), Neo4J, Redis and MongoDB. Another example is OpenShift,[18] a PaaS vendor that provides three database supports, namely MySQL, Postgres and MongoDB. Therefore, data storage layers where RDBMSs and NoSQL solutions coexist also seems to be in the agenda of backend providers. This evidences the fact that RDBMSs and NoSQL databases are indeed complementary technologies rather than competitors.

Acknowledgments

This work has been partially funded by ANPCyT (Argentina) under Project PICT-2011-0366 and by CONICET (Argentina) under Project PIP no. 112-201201-00185.

References

[1] Jans Aasman, Allegro Graph: RDF Triple Database. Technical Report 1, Franz Incorporated, 2006.
[2] Renzo Angles, A comparison of current graph database models, in: IEEE 28th International Conference on Data Engineering Workshops (ICDEW 2012), IEEE, Washington, DC, USA, 2012, pp. 171–177.
[3] Apache Foundation, Apache Avro, http://avro.apache.org/, 2013 (accessed 05.07.13).
[4] Apache Foundation, Apache Giraph, http://giraph.apache.org/, 2013 (accessed 04.11.13).
[5] Marcelo Armentano, Daniela Godoy, Analía Amandi, Towards a followee recommender system for information seeking users in Twitter, in: Proceedings of the International Workshop on Semantic Adaptive Social Web (SASWeb'11) at the 19th International Conference on User Modeling, Adaptation, and Personalization (UMAP 2011), 2011.
[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, Mark Callaghan, LinkBench: a database benchmark based on the Facebook social graph, in: SIGMOD'13 Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 1185–1196. http://dx.doi.org/10.1145/2463676.2465296.
[7] Paolo Atzeni, Francesca Bugiotti, Luca Rossi, Uniform access to NoSQL systems, Inf. Syst. 43 (2014) 117–133, http://dx.doi.org/10.1016/j.is.2013.05.002.
[8] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, Ion Stoica, Quantifying eventual consistency with PBS, VLDB J. 23 (2) (2014) 279–302, http://dx.doi.org/10.1007/s00778-013-0330-1.

[12] Titan Web Page, http://thinkaurelius.github.io/titan/.
[13] Riak CS Web Page, http://basho.com/riak-cloud-storage.
[14] Amazon S3 Web Page, http://aws.amazon.com/es/s3.
[15] OpenRedis, http://openredis.com/.
[16] MongoLab Web Page, http://mongolab.com/welcome/.
[17] http://www.heroku.com/.
[18] OpenShift Web Page, http://www.openshift.com/.

[9] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, Vadim Yushprakh, Megastore: Providing scalable, highly available storage for interactive services, in: Proceedings of the Conference on Innovative Data Systems Research, 2011, pp. 223–234, ISBN 9781450309776, http://dx.doi.org/10.1037/h0054295.
[10] Roberto Baldoni, Matthias Klusch, Fundamentals of distributed computing: A practical tour of vector clock systems, IEEE Distrib. Syst. Online 3 (2002).
[11] Basho Technologies, Inc. Riak, http://docs.basho.com/riak/latest/, 2013 (accessed 05.07.13).
[12] Philip A. Bernstein, Nathan Goodman, Concurrency control in distributed database systems, ACM Comput. Surv. 13 (2) (1981) 185–221.
[13] Dheeraj Bhardwaj, Manish K. Sinha, GridFS: highly scalable i/o solution for clusters and computational grids, Int. J. Comput. Sci. Eng. 2 (5) (2006) 287–291.
[14] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, Jamie Taylor, Freebase: A collaboratively created graph database for structuring human knowledge, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2008, pp. 1247–1250.
[15] Sergio Bossa. Terrastore—Scalable, Elastic, Consistent Document Store, http://code.google.com/p/terrastore/, 2013 (accessed 05.07.13).
[16] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, François Yergeau, Extensible markup language (XML), World WideWeb J. 2 (4) (1997) 27–66.
[17] Mike Burrows, The Chubby lock service for loosely-coupled distributed systems, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, WA, USA, 2006, pp. 335–350.
[18] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, Ivona Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.
[19] Rick Cattell, Scalable SQL and NoSQL data stores, ACM SIGMOD Rec. 39 (4) (2011) 12–27.
[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, BigTable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) 1–26.
[21] Kristina Chodorow, Michael Dirolf, MongoDB: The Definitive Guide, O'Reilly Media, Inc., Sebastopol, CA, USA, 2010.
[22] Cloudant. BigCouch, http://bigcouch.cloudant.com/, 2013 (accessed 05.07.13).
[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, New York, NY, USA, 2010, pp. 143–154.
[24] Alejandro Corbellini, Cristian Mateos, Daniela Godoy, Alejandro Zunino, Silvia Schiaffino, Supporting the efficient exploration of large-scale social networks for recommendation, in: Proceedings of the II Brazilian Workshop on Social Network Analysis and Mining (BraSNAM 2013), Maceió, AL, Brazil, 2013.
[25] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al., Spanner: Google's globally-distributed database, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI 2012), Hollywood, CA, USA, 2012, pp. 251–264.
[26] Couchbase, Membase, http://www.couchbase.org/membase, 2013 (accessed 05.07.13).
[27] Marco Crasso, Alejandro Zunino, Marcelo Campo, A survey of approaches to Web service discovery in service-oriented architectures, J. Database Manag. 22 (1) (2011) 103–134.
[28] Douglas Crockford, JSON: the fat-free alternative to XML, in: Proceedings of XML 2006, vol. 2006, 2006.
[29] Luiz Alexandre Hiane da Silva Maciel, Celso Massaki Hirata, A timestamp-based two phase commit protocol for Web services using REST architectural style, J. Web Eng. 9 (3) (2010) 266–282.
[30] Ork de Rooij, Andrei Vishneuski, Maarten de Rijke, xTAS: Text analysis in a timely manner, in: Proceedings of the 12th Dutch-Belgian Information Retrieval Workshop (DIR 2012), Gent, Belgium, 2012.
[31] Jeffrey Dean, Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels, Dynamo: Amazon's highly available key-value store, ACM SIGOPS Oper. Syst. Rev. 41 (6) (2007) 205–220.
[33] Akon Dey, Alan Fekete, Raghunath Nambiar, Uwe Röhm, YCSB+T Benchmarking web-scale transactional databases, in: 2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW), IEEE, Los Alamitos, CA, USA, 2014, pp. 223–230.
[34] Lars Ellenberg, DRBD 8.0. x and Beyond: Shared-Disk Semantics on a Shared-Nothing Cluster, LinuxConf Europe, 2007.
[35] Orri Erling, Ivan Mikhailov, Virtuoso: RDF support in a native RDBMS, Semantic Web Information Management, Springer, Berlin, Germany, 2010, 501–519.
[36] FAL Labs. Kyoto Cabinet: A Straightforward Implementation of DBM, http://fallabs.com/kyotocabinet/, 2013 (accessed 05.07.13).
[37] Kevin Ferguson, Vijay Raghunathan, Randall Leeds, Shaun Lindsay, Lounge, http://tilgovi.github.io/couchdb-lounge/, 2013 (accessed 05.07.13).
[38] Brad Fitzpatrick. Memcached—A Distributed Memory Object Caching System, http://memcached.org/, 2013 (accessed 05.07.13).
[39] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, Utkarsh Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, Proc. VLDB Endow. 2 (2) (2009) 1414–1425.
[40] L. George, HBase: The Definitive Guide, O'Reilly Media, Inc., Sebastopol, CA, USA, 2011.
[41] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolette, Hans-Arno Jacobsen, Bigbench: Towards an industry standard benchmark for big data analytics, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 1197–1208. http://dx.doi.org/10.1145/2463676.2463712.
[42] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google file system, in: Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY, USA, 2003, pp. 29–43.
[43] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google file system, ACM SIGOPS Oper. Syst. Rev. 37 (5) (2003) 29–43.
[44] Seth Gilbert, Nancy Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News 33 (2) (2002) 51, http://dx.doi.org/10.1145/564585.564601.
[45] Google Inc. LevelDB—A Fast and Lightweight Key/Value Database Library by Google, http://code.google.com/p/leveldb/, 2013 (accessed 05.07.13).
[46] Jim Gray, Andreas Reuter, Transaction Processing: Concepts and Techniques, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[47] Kevin W. Hamlen, Bhavani Thuraisingham, Data security services, solutions and standards for outsourcing, Comput. Stand. Interfaces 35 (1) (2013) 1–5.
[48] Hazelcast, Inc. Hazelcast—In-Memory Data Grid, http://www.hazelcast.com/http://www.hazelcast.com/http://www.hazelcast.com/, 2013 (accessed 05.07.13).
[49] Robin Hecht, Stefan Jablonski, NoSQL evaluation: A use case oriented survey, in: IEEE International Conference on Cloud and Service Computing (CSC 2011), 2011, pp. 336–341.
[50] Hibernating Rhinos. RavenDB, http://www.ravendb.net/, 2013 (accessed 05.07.13).
[51] Rich Hickey, The Clojure programming language, in: Proceedings of the 2008 Symposium on Dynamic Languages (DLS'08), Paphos, Cyprus, 2008, pp. 1:1–1:1.
[52] B. Holt, Scaling CouchDB, O'Reilly Media, Inc., Sebastopol, CA, USA, 2011.
[53] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, Bo Huang, The HiBench benchmark suite: characterization of the MapReduce-based data analysis, New Frontiers in Information and Software as Services, Springer, Berlin, Germany, 2011, 209–228.
[54] Hypertable Inc. Hypertable, http://www.hypertable.org/, 2013 (accessed 04.11.13).
[55] Borislav Iordanov, HyperGraphDB: A generalized graph database, in: H. Shen, J. Pei, M. Özsu, L. Zou, J. Lu, T.-W. Ling, G. Yu, Y. Zhuang, J. Shao (Eds.), Web-Age Information Management, Lecture Notes in Computer Science, vol. 6185, Springer, Berlin, Germany, 2010, pp. 25–36.
[56] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, Alejandro Abdelnur, Oozie: Towards a scalable workflow management system for Hadoop. in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies,

SWEET '12, ACM, New York, NY, USA, 2012, pp. 4:1–4:10, http://dx.doi.org/10.1145/2443416.2443420.
[57] Anne James, Joshua Cooper, Challenges for database management in the Internet of things, IETE Tech. Rev. 26 (5) (2009) 320–329.
[58] Keith Jeffery, The Internet of things: the death of a traditional database? IETE Tech. Rev. 26 (5) (2009) 313–319.
[59] Tomasz Kajdanowicz, Przemyslaw Kazienko, Wojciech Indyk, Parallel Processing of Large Graphs, CoRR, abs/1306.0326, 2013.
[60] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, Daniel Lewin, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web, in: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97), El Paso, TX, USA, 1997, pp. 654–663.
[61] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi, Web caching with consistent hashing, Comput. Netw. 31 (11) (1999) 1203–1213, http://dx.doi.org/10.1016/S1389-1286(99)00055-9.
[62] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, Panos Kalnis, Mizan: A system for dynamic load balancing in large-scale graph processing, in: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13, ACM, New York, NY, USA, 2013, pp. 169–182, http://dx.doi.org/10.1145/2465351.2465369.
[63] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, Henri Bal, HipG: parallel processing of large-scale graphs, ACM SIGOPS Oper. Syst. Rev. 45 (2) (2011) 3–13.
[64] Avinash Lakshman, Prashant Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Oper. Syst. Rev. 44 (2) (2010) 35–40.
[65] Leslie Lamport, Time clocks and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, http://dx.doi.org/10.1145/359545.359563.
[66] Leslie Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169, http://dx.doi.org/10.1145/279227.279229.
[67] Neal Leavitt, Will NoSQL databases live up to their promise? Computer 43 (2) (2010) 12–14.
[68] Joe Lennon, Introduction to CouchDB, Beginning CouchDB, Apress, Berkely, CA, USA, 2009, 3–9.
[69] Ralf Lämmel, Google's MapReduce programming model—revisited, Sci. Comput. Program. 70 (1) (2008) 1–30.
[70] LinkedIn Corporation, Voldemort, http://www.project-voldemort.com/voldemort/, 2013 (accessed 05.07.13).
[71] Adam Lith, Jakob Mattsson, Investigating storage solutions for large data (Ph.D. thesis, Master's thesis), Department of Computer Science and Engineering—Chalmers University of Technology, 2010.
[72] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, Jonathan Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (1) (2007) 5–20.
[73] Mani Malarvannan, Srini Ramaswamy, Rapid scalability of complex and dynamic web-based systems: Challenges and recent approaches to mitigation, in: Proceedings of the 5th International Conference on System of Systems Engineering (SoSE 2010), 2010, pp. 1–6.
[74] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, Pregel: A system for large-scale graph processing. in: Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10), Indianapolis, IN, USA, 2010, pp. 135–146.
[75] Francesco Marchioni, Infinispan Data Grid Platform, Packt Pub Limited, Birmingham, UK, 2012.
[76] Richard McCreadie, Craig Macdonald, Iadh Ounis, MapReduce indexing strategies: Studying scalability and efficiency, Inf. Process. Manag. 48 (5) (2012) 873–888.
[77] P. Membrey, E. Plugge, T. Hawkins, The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing, Apress, Berkely, CA, USA, 2010.
[78] Ralph Merkle, A digital signature based on a conventional encryption function, in: Carl Pomerance (Ed.), Advances in Cryptology (CRYPTO'87), Lecture Notes in Computer Science, vol. 293, Springer, Berlin, Germany, 2006, pp. 369–378.
[79] Scott M. Meyer, Jutta Degener, John Giannandrea, Barak Michener, Optimizing schema-last tuple-store queries in Graphd, in: Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10), Indianapolis, IN, USA, 2010, pp. 1047–1056.
[80] Inc. Neo Technology, Neo4J, http://www.neo4j.org/, 2013 (accessed 05.08.13).
[81] NetMesh Inc., InfoGrid Web Graph Database, URL http://infogrid.org/trac/, 2013 (accessed 05.07.13).
[82] NuvolaBase, OrientDB Graph-Document NoSQL DBMS, http://www.orientdb.org/, 2013 (accessed 05.07.13).
[83] Objectivity Inc., InfiniteGraph, http://www.objectivity.com/infinitegraph, 2013 (accessed 05.08.13).
[84] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, The Scala Language Specification, http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2004.
[85] Michael A Olson, Keith Bostic, Margo Seltzer, Berkeley DB, in: Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, 1999, pp. 183–192.
[86] Kai Orend, Analysis and classification of NoSQL databases and evaluation of their ability to replace an object-relational persistence layer (Master's thesis), Technische Universität München, 2010.
[87] John Ousterhout, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, Stephen Yang, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, The RAMCloud storage system, ACM Trans. Comput. Syst. 33 (3) (2015) 1–55, http://dx.doi.org/10.1145/2806887.
[88] Rabi Prasad Padhy, Manas Ranjan Patra, Suresh Chandra Satapathy, RDBMS to NoSQL: Reviewing some next-generation non-relational database's, Int. J. Adv. Eng. Sci. Technol. 11 (1) (2011) 15–30.
[89] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, YCSB++: benchmarking and performance debugging advanced features in scalable table stores, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, New York, NY, USA, 2011, pp. 1–14. http://dx.doi.org/10.1145/2038916.2038925.
[90] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan, Interpreting the data: parallel analysis with Sawzall, Sci. Program. J. 13 (4) (2005) 277–298.
[91] San Pritchett, BASE: An Acid alternative, ACM Queue 6 (3) (2008) 48–55.
[92] Eric Prud'Hommeaux, Andy Seaborne, et al., SPARQL query language for RDF, W3C Recomm. 15 (2008).
[93] Marko A. Rodriguez, Gremlin, http://github.com/tinkerpop/gremlin/wiki, 2013 (accessed 05.08.13).
[94] Sullivan Russell, Alchemy Database—A Hybrid Relational-Database/NOSQL-Datastore, URL https://code.google.com/p/alchemydatabase/, 2013 (accessed 05.07.13).
[95] Sherif Sakr, Anna Liu, Daniel.M. Batista, Mohammad Alomari, A survey of large scale data management approaches in cloud environments, IEEE Commun. Surv. Tutorials 13 (3) (2011) 311–336.
[96] Sherif Sakr, Sameh Elnikety, Yuxiong He, Hybrid query execution engine for large attributed graphs, Inf. Syst. 41 (May) (2014) 45–73, http://dx.doi.org/10.1016/j.is.2013.10.007. ISSN 03064379. URL http://linkinghub.elsevier.com/retrieve/pii/S0306437913001452.
[97] Salvatore Sanfilippo, Redis, http://redis.io, 2013 (accessed 05.07.13).
[98] Thorsten Schutt, Florian Schintke, Alexander Reinefeld, Scalaris: reliable transactional p2p key/value store, in: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, 2008, pp. 41–48.
[99] Bin Shao, Haixun Wang, Yatao Li, The Trinity Graph Engine, Technical Report MSR-TR-2012-30, Microsoft Research, March 2012.
[100] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, Phoenix Tong, F1—the fault-tolerant distributed RDBMS supporting Google's ad business, in: SIGMOD, 2012, Talk given at SIGMOD 2012.
[101] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, The Hadoop distributed file system, in: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2010), 2010, pp. 1–10.
[102] Mark Slee, Aditya Agarwal, Marc Kwiatkowski, Thrift: Scalable cross-language services implementation, Facebook White Pap. 5 (2007).
[103] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, ACM SIGCOMM Comput. Commun. Rev. 31 (4) (2001) 149–160.
[104] Michael Stonebraker, New opportunities for New SQL, Commun. ACM 55 (11) (2012) 10–11.
[105] Michael Stonebraker, Rick Cattell, 10 rules for scalable performance in 'simple operation' datastores, Commun. ACM 54 (6) (2011) 72–80.
[106] Michael Stonebraker, Ariel Weisberg, The VoltDB main memory DBMS, IEEE Data Eng. Bull. (2013) 21–27.
[107] Christof Strauch, Walter Kriha, NoSQL databases, Lecture Notes, Stuttgart Media University, Stuttgart, Germany, 2011.

[108] Carlo Strozzi, NoSQL Relational Database Management System: Home Page, URL http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home-Page, 1998 (accessed 05.07.13).
[109] SYSTAP, LLC. BigData, http://www.systap.com/bigdata.htm, 2013 (accessed 05.08.13).
[110] Clarence Tauro, S. Aravindh, A. Shreeharsha, Comparative study of the new generation, agile, scalable, high performance NOSQL databases, Int. J. Comput. Appl. 48 (20) (2012) 1–4.
[111] Inc Terracotta et al., The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability, Apress, Berkely, CA, USA, 2008.
[112] Ashish Thusoo, Joydeep Sarma Sen, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghotham Murthy, Hive: A warehousing solution over a Map-Reduce framework, Proc. VLDB Endow. 2(2) (2009) 1626–1629.
[113] Bogdan George Tudorica, Cristian Bucur, A comparison between several NoSQL databases with comments and notes, in: Proceedings of the 10th Roedunet International Conference (RoEduNet 2011), 2011, pp. 1–5.
[114] Twitter Inc, FlockDB, http://github.com/twitter/flockdb, 2013 (accessed 05.08.13).
[115] Ian Varley, No relation: the mixed blessings of non-relational databases (Master's thesis), The University of Texas, Austin, Texas, 2009.
[116] Werner Vogels, Eventually consistent, Commun. ACM 52 (1) (2009) 40–44.
[117] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, Bizhu Qiu, BigDataBench: A big data benchmark suite from Internet services, in: Proceedings—International Symposium on High-Performance Computer Architecture, 2014, pp. 488–499, http://dx.doi.org/10.1109/HPCA.2014.6835958.