Fall 12-18-2013

Data management in cloud environments: NoSQL and NewSQL data stores

Katarina Grolinger
kgroling@uwo.ca

Wilson A. Higashino
Western University, whigashi@uwo.ca

Abhinav Tiwari
Western University, atiwari8@uwo.ca

Miriam AM Capretz
Western University, mcapretz@uwo.ca

Citation of this paper:
Grolinger, Katarina; Higashino, Wilson A.; Tiwari, Abhinav; and Capretz, Miriam AM, "Data management in cloud environments: NoSQL and NewSQL data stores" (2013). *Electrical and Computer Engineering Publications*. 22.
https://ir.lib.uwo.ca/electricalpub/22

Journal of Cloud Computing: Advances, Systems and Applications

SpringerOpen

Data management in cloud environments: NoSQL and NewSQL data stores

Journal of Cloud Computing: Advances, Systems and Applications 2013, **2**:22 doi:10.1186/2192-113X-2-22

Katarina Grolinger (kgroling@uwo.ca)
Wilson A Higashino (whigashi@uwo.ca)
Abhinav Tiwari (atiwari8@uwo.ca)
Miriam AM Capretz (mcapretz@uwo.ca)

ISSN	2192-113X
Article type	Research
Submission date	1 October 2013
Acceptance date	13 December 2013
Publication date	18 December 2013
Article URL	http://www.journalofcloudcomputing.com/content/2/1/22

Data management in cloud environments: NoSQL and NewSQL data stores

Katarina Grolinger[1]
Email: kgroling@uwo.ca

Wilson A Higashino[1,2,*]
Email: whigashi@uwo.ca

Abhinav Tiwari[1]
Email: atiwari8@uwo.ca

Miriam AM Capretz[1]
Email: mcapretz@uwo.ca

[1] Department of Electrical and Computer Engineering, Faculty of Engineering, Western University, London, ON N6A 5B9, Canada

[2] Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brazil

[*] Corresponding author. Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brazil

Abstract

Advances in Web technology and the proliferation of mobile devices and sensors connected to the Internet have resulted in immense processing and storage requirements. Cloud computing has emerged as a paradigm that promises to meet these requirements. This work focuses on the storage aspect of cloud computing, specifically on data management in cloud environments. Traditional relational databases were designed in a different hardware and software era and are facing challenges in meeting the performance and scale requirements of Big Data. NoSQL and NewSQL data stores present themselves as alternatives that can handle huge volume of data. Because of the large number and diversity of existing NoSQL and NewSQL solutions, it is difficult to comprehend the domain and even more challenging to choose an appropriate solution for a specific task. Therefore, this paper reviews NoSQL and NewSQL solutions with the objective of: (1) providing a perspective in the field, (2) providing guidance to practitioners and researchers to choose the appropriate data store, and (3) identifying challenges and opportunities in the field. Specifically, the most prominent solutions are compared focusing on data models, querying, scaling, and security related capabilities. Features driving the ability to scale read requests and write requests, or scaling data storage are investigated, in particular partitioning, replication, consistency, and concurrency control. Furthermore, use cases and scenarios in which NoSQL and NewSQL data stores have been used are discussed and the suitability of various solutions for different sets of applications is examined. Consequently, this study has identified challenges in the field, including the immense diversity and inconsistency of terminologies, limited documentation, sparse comparison and benchmarking criteria, and nonexistence of standardized query languages.

Keywords

NoSQL, NewSQL, Big Data, Cloud computing, Distributed storage, Data management

Introduction

In recent years, advances in Web technology and the proliferation of sensors and mobile devices connected to the Internet have resulted in the generation of immense data sets that need to be processed and stored. Just on Facebook, 2.4 billion content items are shared among friends every day [1]. Today, businesses generate massive volume of data which has grown too big to be managed and analyzed by traditional data processing tools [2]. Indeed, traditional relational database management systems (RDBMS) were designed in an era when the available hardware, as well as the storage and processing requirements, were very different than they are today [3]. Therefore, these solutions have been encountering many challenges in meeting the performance and scaling requirements of this “Big Data” reality.

Big Data is a term used to refer to massive and complex datasets made up of a variety of data structures, including structured, semi-structured, and unstructured data. According to the Gartner group, Big Data can be defined by 3Vs: volume, velocity, and variety [4]. Today, businesses are aware that this huge volume of data can be used to generate new opportunities and process improvements through their processing and analysis [5,6].

At about the same time, cloud computing has also emerged as a computational paradigm for on-demand network access to a shared pool of computing resources (e.g., network, servers, storage, applications, and services) that can be rapidly provisioned with minimal management effort [7]. Cloud computing is associated with service provisioning, in which service providers offer computer-based services to consumers over the network. Often these services are based on a pay-per-use model where the consumer pays only for the resources used. Overall, a cloud computing model aims to provide benefits in terms of lesser up-front investment, lower operating costs, higher scalability, elasticity, easy access through the Web, and reduced business risks and maintenance expenses [8].

Due to such characteristics of cloud computing, many applications have been created in or migrated to cloud environments over the last few years [9]. In fact, it is interesting to notice the extent of synergy between the processing requirements of Big Data applications, and the availability and scalability of computational resources offered by cloud services. Nevertheless, the effective leveraging of cloud infrastructure requires careful design and implementation of applications and data management systems. Cloud environments impose new requirements to data management; specifically, a cloud data management system needs to have:

- Scalability and high performance, because today’s applications are experiencing continuous growth in terms of the data they need to store, the users they must serve, and the throughput they should provide;
- Elasticity, as cloud applications can be subjected to enormous fluctuations in their access patterns;
- Ability to run on commodity heterogeneous servers, as most cloud environments are based on them;

- Fault tolerance, given that commodity machines are much more prone to fail than high-end servers;
- Security and privacy features, because the data may now be stored on third-party premises on resources shared among different tenants;
- Availability, as critical applications have also been moving to the cloud and cannot afford extended periods of downtime.

Faced with the challenges that traditional RDBMSs encounter in handling Big Data and in satisfying the cloud requirements described above, a number of specialized solutions have emerged in the last few years in an attempt to address these concerns. The so-called NoSQL and NewSQL data stores present themselves as data processing alternatives that can handle this huge volume of data and provide the required scalability.

Despite the appropriateness of NoSQL and NewSQL data stores as cloud data management systems, the immense number of existing solutions (over 120 [10]) and the discrepancies among them make it difficult to formulate a perspective on the domain and even more challenging to select the appropriate solution for a problem at hand. This survey reviews NoSQL and NewSQL data stores with the intent of filling this gap. More specifically, this survey has the following objectives:

- To provide a perspective on the domain by summarizing, organizing, and categorizing NoSQL and NewSQL solutions.
- To compare the characteristics of the leading solutions in order to provide guidance to practitioners and researchers to choose the appropriate data store for specific applications.
- To identify research challenges and opportunities in the field of large-scale distributed data management.

NoSQL data models and categorization of NoSQL data stores have been addressed in other surveys [10-14]. In addition, aspects associated with NoSQL, such as MapReduce, the CAP theorem, and eventual consistency have also been discussed in the literature [15,16]. This paper presents a short overview of NoSQL concepts and data models; nevertheless, the main contributions of this paper include:

- A discussion of NewSQL data stores. The category of NewSQL solutions is recent; the first use of the term was in 2011 [17]. NewSQL solutions aim to bring the relational data model into the world of NoSQL. Therefore, a comparison among NewSQL and NoSQL solutions is essential to understand this new class of data stores.
- A detailed comparison among various NoSQL and NewSQL solutions over a large number of dimensions. By presenting this comparison in a table form, this paper helps practitioners to choose the appropriate data store for the task at hand. Previous surveys have included comparisons of NoSQL solutions [11]; nonetheless, the number of compared attributes was limited, and the analysis performed was not as comprehensive.
- A review of a number of security features is also included in the data store comparison. According to the surveyed literature [10-14], security has been overlooked, even though it is an important aspect of the adoption of NoSQL solutions in practice.
- A discussion of the suitability of various NoSQL and NewSQL solutions for different sets of applications. NoSQL and NewSQL solutions differ greatly in their characteristics; moreover, changes in this area are rapid, with frequent releases of new features and options. Therefore, this work discusses the suitability of NoSQL and NewSQL data stores for different use cases from the perspective of core design decisions.

The rest of this paper is organized as follows: the “Background and Related Work” section describes background concepts and studies related to this survey. The methodology used in this survey is presented in the “Methodology” section. The “Data Models” section presents the NoSQL and NewSQL data models and categorizes the surveyed data stores accordingly. Querying capabilities are discussed in the “Querying” section, while the “Scaling” section describes the solutions’ scaling properties and the “Security” section their security features. The suitability of NoSQL and NewSQL data stores for different use cases is discussed in the “Use Cases” section. The challenges and opportunities identified in this study are described in the “Opportunities” section, and the “Conclusions” section concludes the paper.

Background and related work

This section introduces relevant concepts and positions this paper with respect to other surveys in the NoSQL domain.

Cloud computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., network, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [7]. It denotes a model in which a computing infrastructure is viewed as a “cloud”, from which businesses and individuals can access applications on demand from anywhere in the world [18]. Essential characteristics of the cloud-computing model, according to the U.S. National Institute of Standards and Technology (NIST), include [7]:

- On-demand self-service, enabling a user to access cloud provider services without human interaction;
- Broad network access that enables heterogeneous thick and thin client applications to access the services;
- Pooling of service provider computing resources to serve multiple consumers;
- Automatic, rapid, and elastic provisioning of resources;
- Measured service in which resource usage is monitored and controlled.

Overall, a cloud computing model aims to provide benefits in terms of lesser up-front investment in infrastructure during deployment, lower operating costs, higher scalability, ease of access through the Web, and reduced business risks and maintenance expenses [8].

The CAP theorem

In order to store and process massive datasets, a common employed strategy is to partition the data and store the partitions across different server nodes. Additionally, these partitions can also be replicated in multiple servers so that the data is still available even in case of servers’ failures. Many modern data stores, such as Cassandra [19] and BigTable [20], use these and others strategies to implement high-available and scalable solutions that can be leveraged in cloud environments. Nevertheless, these solutions and others replicated networked data stores have an important restriction, which was formalized by the CAP theorem [21]: only two of three CAP properties (consistency, availability, and partition tolerance) can be satisfied by networked shared-data systems at the same time [21,22].

Consistency, as interpreted in CAP, is equivalent to having a single up-to-date instance of the data [22]. Therefore, *consistency* in CAP has a somewhat dissimilar meaning to and represents only a subset of *consistency* as defined in ACID (Atomicity, Consistency, Isolation and Durability) transactions of RDBMSs [22], which usually refers to the capability of maintaining the database in a consistent state at all times. The *Availability* property means that the data should be available to serve a request at the moment it is needed. Finally, the *Partition Tolerance* property refers to the capacity of the networked shared-data system to tolerate network partitions. The simplest interpretation of the CAP theorem is to consider a distributed data store partitioned into two sets of participant nodes; if the data store denies all write requests in both partitions, it will remain consistent, but it is not available. On the other hand, if one (or both) of the partitions accepts write requests, the data store is available, but potentially inconsistent.

Despite the relative simplicity of its result, the CAP theorem has had important implications and has originated a great variety of distributed data stores aiming to explore the trade-offs between the three properties. More specifically, the challenges of RDBMS in handling Big Data and the use of distributed systems techniques in the context of the CAP theorem led to the development of new classes of data stores called NoSQL and NewSQL.

NoSQL and NewSQL

The origin of the NoSQL term is attributed to Johan Oskarsson, who used it in 2009 to name a conference about “open-source, distributed, non-relational databases” [23]. Today, the term is used as an acronym for “Not only SQL”, which emphasizes that SQL-style querying is not the crucial objective of these data stores. Therefore, the term is used as an umbrella classification that includes a large number of immensely diverse data stores that are not based on the relational model, including some solutions designed for very specific applications such as graph storage. Even though there is no agreement on what exactly constitutes a NoSQL solution, the following set of characteristics is often attributed to them [11,15]:

- Simple and flexible non-relational data models. NoSQL data stores offer flexible schemas or are sometimes completely schema-free and are designed to handle a wide variety of data structures [11,12,24]. Current solution data models can be divided into four categories: key-value stores, document stores, column-family stores, and graph databases.
- Ability to scale horizontally over many commodity servers. Some data stores provide data scaling, while others are more concerned with read and/or write scaling.
- Provide high availability. Many NoSQL data stores are aimed towards highly distributed scenarios, and consider partition tolerance as unavoidable. Therefore, in order to provide high availability, these solutions choose to compromise consistency in favour of availability, resulting in AP (Available / Partition-tolerant) data stores, while most RDBMs are CA (Consistent / Available).
- Typically, they do not support ACID transactions as provided by RDBMS. NoSQL data stores are sometimes referred as BASE systems (Basically Available, Soft state, Eventually consistent) [25]. In this acronym, *Basically Available* means that the data store is available all the time whenever it is accessed, even if parts of it are unavailable; *Soft-state* highlights that it does not need to be consistent always and can tolerate inconsistency for a certain time period; and *Eventually consistent* emphasizes that after a certain time period, the data store comes to a consistent state. However, some NoSQL data stores, such as CouchDB [26] provide ACID compliance.

These characteristics make NoSQL data stores especially suitable for use as cloud data management systems. Indeed, many of the Database as a Service offerings available today, such as Amazon's SimpleDB [27] and DynamoDB [28], are considered to be NoSQL data stores. However, the lack of full ACID transaction support can be a major impediment to their adoption in many mission-critical systems. For instance, Corbert *et al.* [29] argue that it is better to deal with performance problems caused by the overuse of transactions rather than trying to work around the lack of transaction support. Furthermore, the use of low-level query languages, the lack of standardized interfaces, and the huge investments already made in SQL by enterprises are other barriers to the adoption of NoSQL data stores.

The category of NewSQL data stores, on the other hand, is being used to classify a set of solutions aimed at bringing to the relational model the benefits of horizontal scalability and fault tolerance provided by NoSQL solutions. The first use of the term is attributed to a report of the 451 group in 2011 [17]. The Google Spanner [29] solution is considered to be one of the most prominent representatives of this category, as is also VoltDB [30], which is based on the H-Store [31] research project. Clustrix [32] and NuoDB [33] are two commercial projects that are also classified as NewSQL. All these data stores support the relational model and use SQL as their query language, even though they are based on different assumptions and architectures than traditional RDBMSs. Generally speaking, NewSQL data stores meet many of the requirements for data management in cloud environments and also offer the benefits of the well-known SQL standard.

Related surveys

Several surveys have addressed the NoSQL domain [10-14]; nevertheless, this survey is different because it focuses on the comparison of available NoSQL and NewSQL solutions over a number of dimensions. Hecht and Jablonski [11] presented a use case-oriented survey, which, like this one, compares features of several NoSQL solutions, including the data models, querying capabilities, partitioning, replication, and consistency. However, for a large number of features, they use a "black and white" (+/−) approach to indicate that the solution either does or does not have the feature. This survey adopts a different approach by expressing degrees, aspects, and details of each solution's features. Moreover, this survey includes security features and NewSQL solutions, which are not addressed in their work.

Pokorny [13], Cattell [12], and Sakr *et al.* [14] have also reviewed NoSQL data stores. They portrayed a number of NoSQL data stores, describing their data models and their main underlying principles and features. However, in contrast to this work, they did not perform direct feature comparison among data stores. Sadalage and Fowler [15] described the principles on which NoSQL stores are based and why they may be superior to traditional databases. They introduced several solutions, but they did not compare features as is done in this work.

In addition, existing surveys have not described the rationale or method for choosing the specific data stores to include in their studies [11-14]. For example, Sakr *et al.* stated, "…we give a brief introduction about some of those projects" [14], or Hecht and Jablonski " The most prominent stores are …" [11]; however, the method for choosing the data stores included in their studies were not presented. In contrast, this work uses a systematic approach to choose which data stores to include in the study. Additionally, this survey includes different data stores than the existing surveys [11-14].

Methodology

Due to the large number of NoSQL and NewSQL solutions, it was not feasible to include all of them in this survey. While other NoSQL surveys did not specify the methodology for choosing the data stores to be included in their studies [11-14], this survey makes use of a systematic approach to select the solutions.

DB-Engine Ranking [34] ranks database systems according to their popularity by using parameters such as the number of mentions on Web sites, general interest according to Google Trends, frequency of technical discussions on the Web, number of job offers, and number of professional profiles in which the solutions are mentioned. As can be seen, the DB-Engine Ranking estimates overall popularity of a data store on the Web. Nevertheless, this work is also interested in popularity within the research community; therefore, it also considers how often each system has been mentioned in research publications. Even though various research repositories could have been used, this study focuses on the IEEE as it is one of the most prominent publishers of research papers in computer science and software engineering. Hence, the initial list of NoSQL solutions was obtained from DB-Engine Ranking [34] and includes all NoSQL solutions listed by DB-Engine Ranking. Next, the IEEE Xplore database was searched to determine how many times each data store was mentioned in the indexed publications. For each NoSQL category, the most often cited data stores were chosen to be included in this survey. The key-value category was further divided into in-memory and disk-persistent key-value stores, and the most prominent solutions within each subcategory were chosen.

The prevalent data stores found in IEEE publications are similar to the data stores ranked high by DB-Engine Ranking. In the document category, the same three data stores, MongoDB [35], CouchDB [26], and Couchbase [36] are the most popular according to DB-Engine Ranking and IEEE publications. Both popularity estimation approaches rank Cassandra [19] and HBase [37] as the most prominent in the column-family category. SimpleDB [27] and DynamoDB [28] are ranked high by both approaches. While DB-Engine Ranking considers them key-value stores, this work categorizes them as column-family stores because of their table-like data model. In the remaining two categories, key-value data stores and graph databases, a large number of solutions rank high in popularity according to both approaches, including Redis [38], Memcached [39], Riak [40], BerkeleyDB [41], and Neo4J [42].”

The selection of NewSQL data stores followed a similar approach. Nevertheless, because most of these solutions are very recent, only VoltDB and Spanner had a significant number of hits in the IEEE Xplore database. Therefore, in order to include a larger number of solutions in this survey, Clustrix and NuoDB were also selected because of their unique architectural and technical approaches.

The selected NoSQL and NewSQL solutions were compared with a focus on the data model, querying, scaling, and security-related capabilities. The categorization according to data model was used because the data model is the main factor driving other capabilities, including querying and scaling. In the querying context, support for MapReduce, SQL-like querying, REST (representational state transfer) and other APIs was considered. With regard to scaling, the study considered scaling read and write requests, or scaling data storage and analyzed four concepts closely related: partitioning, replication, consistency, and concurrency control. Finally, the following security related features were analyzed: authentication, authorization, encryption, and auditing.

Data models

The family of data stores belonging to the NoSQL category can be further sub-classified based on their data models. Many authors have proposed distinct interpretations for NoSQL categories, which has led to different sub-classifications [10,12]. In this paper, the classification provided by Hecht and Jablonski [11] has been used, which divides the various NoSQL data stores into four major categories: key-value stores, column-family stores, document stores, and graph databases. Figure 1 shows representations of these models. This study also reviews NewSQL as a hybrid between NoSQL stores and relational databases.

Figure 1 Different types of NoSQL data models.

Key-value stores

Key-value stores have a simple data model based on key-value pairs, which resembles an associative map or a dictionary [11]. The key uniquely identifies the value and is used to store and retrieve the value into and out of the data store. The value is opaque to the data store and can be used to store any arbitrary data, including an integer, a string, an array, or an object, providing a schema-free data model. Along with being schema-free, key-value stores are very efficient in storing distributed data, but are not suitable for scenarios requiring relations or structures. Any functionality requiring relations, structures, or both must be implemented in the client application interacting with the key-value store. Furthermore, because the values are opaque to them, these data stores cannot handle data-level querying and indexing and can perform queries only through keys. Key-value stores can be further classified as *in-memory key-value stores* which keep the data in memory, like Memcached [39] and Redis [38], and *persistent key-value stores* which maintain the data on disk, such as BerkeleyDB [41], Voldemort [43], and Riak [40].

Column-family stores

Most column-family stores are derived from Google Bigtable [20], in which the data are stored in a column-oriented way. In Bigtable, the dataset consists of several rows, each of which is addressed by a unique row key, also known as a *primary key*. Each row is composed of a set of column families, and different rows can have different column families. Similarly to key-value stores, the row key resembles the key, and the set of column families resembles the value represented by the row key. However, each column family further acts as a key for the one or more columns that it holds, where each column consists of a name-value pair. Hadoop HBase [37] directly implements the Google Bigtable concepts, whereas Amazon SimpleDB [27] and DynamoDB [28] have a different data model than Bigtable. SimpleDB and DymanoDB contain only a set of column name-value pairs in each row, without having column families. Cassandra [19], on the other hand, provides the additional functionality of super-columns, which are formed by grouping various columns together.

In column-family stores, a column family in different rows can contain different columns. Occasionally, SimpleDB and DynamoDB are classified as key-value stores [34]; however, this paper considers them as column-family stores due to their table-like data model in which each row can have different columns. Typically, the data belonging to a row is stored together on the same server node. However, Cassandra offers to store a single row across multiple server nodes by using composite partition keys. In column-family stores, the configuration of column families is typically performed during start-up. However, a prior definition of columns is not required, which offers huge flexibility in storing any data type.

In general, column-family stores provide more powerful indexing and querying than key-value stores because they are based on column families and columns in addition to row keys. Similarly to key-value stores, any logic requiring relations must be implemented in the client application.

Document stores

Document stores provide another derivative of the key-value store data model by using keys to locate documents inside the data store. Most document stores represent documents using JSON (JavaScript Object Notation) or some format derived from it. For example, CouchDB [26] and the Couchbase server [36] use the JSON format for data storage, whereas MongoDB [35] stores data in BSON (Binary JSON). Document stores are suitable for applications in which the input data can be represented in a document format. A document can contain complex data structures such as nested objects and does not require adherence to a fixed schema. MongoDB provides the additional functionality of grouping the documents together into collections. Therefore, inside each collection, a document should have a unique key.

Unlike an RDBMS, where every row in a table follows the same schema, each document inside these document stores can have a different structure. Document stores provide the capability of indexing documents based on the primary key as well as on the contents of the documents. This indexing and querying capability based on document contents differentiates this data model from the key-value stores model, in which the values are opaque to the data store. On the other hand, document stores can store only data that can be represented as a document. Like key-value stores, they are inefficient in multiple-key transactions involving cross-document operations.

Graph databases

Graph databases originated from graph theory and use graphs as their data model. A graph is a mathematical concept used to represent a set of objects, known as vertices or nodes, and the links (or edges) that interconnect these vertices. By using a completely different data model than key-value, column-family, and document stores, graph databases can efficiently store the relationships between different data nodes. In graph databases, the nodes and edges also have individual properties consisting of key-value pairs. Graph databases are specialized in handling highly interconnected data and therefore are very efficient in traversing relationships between different entities. They are suitable in scenarios such as social networking applications, pattern recognition, dependency analysis, recommendation systems and solving path finding problems raised in navigation systems [11,44].

Some graph databases such as Neo4J [42] are fully ACID-compliant. However, they are not as efficient as other NoSQL data stores in scenarios other than handling graphs and relationships. Moreover, existing graph databases are not efficient at horizontal scaling because when related nodes are stored on different servers, traversing multiple servers is not performance-efficient.

NewSQL

These solutions are by definition based on the relational model. VoltDB [30], Clustrix [32], and NuoDB [33] offer their clients a pure relational view of data. On the other hand, Google Spanner [29] is based on a semi-relational model in which tables are seen as mappings from the primary-key columns to the other columns. In its model, hierarchies of tables are created so that users can specify locality relationships between tables [29].

Even though clients interact with these data stores in terms of tables and relations, it is interesting to note that NewSQL solutions might use different data representations internally. For example, NuoDB can store its data into any compatible key-value store.

Querying

Similar to the selection of a data model, the querying capabilities of data stores play an important role when choosing among them for a particular scenario. Different data stores offer different APIs and interfaces to interact with them. This is directly dependent upon the data model that a particular data store possesses. For example, a key-value store cannot provide querying based on the contents of the values, because these values are opaque to the data store. On the other hand, a document store can do so because its data model provides the capability to index and query the document contents.

Another important query-related feature of NoSQL and NewSQL data stores is their level of support for MapReduce. MapReduce, which was first developed by Google, is a programming model and an associated implementation for processing large datasets [45]. It has now become a widely accepted approach for performing distributed data processing on a cluster of computers. Because one of the primary goals of NoSQL data stores is to scale over a large number of computers, MapReduce has been adopted by most of them. Similarly, SQL-like querying has been a preferred choice because of its widespread use over the past few decades, and it has now also been adopted in the NoSQL world. Therefore, some of the prominent NoSQL data stores like MongoDB [35] offer a SQL-like query language or similar variants such as CQL [46] offered by Cassandra and SparQL [47] by Neo4j and Allegro Graph [48].

As for the NewSQL category, the use of SQL as a query language is one of its defining characteristics, but the level of SQL support varies considerably. Clustrix [32] and NuoDB [33] are the most SQL-compliant of the solutions analyzed, having only minor incompatibilities with the standard. On the other hand, Corbett *et al.* state that the Google Spanner query language "looks like SQL with some extensions to support protocol-buffer-value fields" [29], but they do not provide details about the language. Finally, VoltDB [30] has a larger number of restrictions in place: it is not possible to use the *having* clause, tables cannot join with themselves, and all joined tables must be partitioned over the same value. It is also worth mentioning that the recommended way of interacting with VoltDB is through Stored Procedures. These procedures are written in Java, where programming logic and SQL statements are interspersed.

On the other hand, a command-line interface (CLI) is usually the simplest and most common interface that a data store can provide for interaction with itself and is therefore offered by almost all NoSQL and NewSQL products. In addition, most of these products offer API support for multiple languages. Moreover, a REST-based API has been very popular in the world of Web-based applications because of its simplicity [49]. Consequently, in the NoSQL world, a REST-based interface is provided by most solutions, either directly or indirectly through third-party APIs. Table 1 provides a detailed view of the different APIs support provided by the most prominent NoSQL and NewSQL solutions along with other querying capabilities offered.

Table 1 Querying capabilities

NoSQL Data Stores		Querying					License
		Map Reduce	REST	Query	Other API	Other features	
Key-value stores	**Redis** http://redis.io	No	Third-party APIs	Does not provide SQL-like querying	CLI and API in several languages	Server-side scripting support using Lua	Open source: BSD (Berkeley Software Distribution)
	Memcached http://memcached.org	No	Third-party APIs	Does not provide SQL-like querying	CLI and API in several languages. Binary and ASCII protocols for custom client development	No server-side scripting support	Open source: BSD 3-clause license
	BerkeleyDB http://www.oracle.com/us/products/database/berkeley-db/overview/index.html	No	Yes	SQLite	CLI and API in several languages	No secondary indices, no server-side-scripting support	Closed source: Oracle Sleepycat license
	Voldemort http://www.project-voldemort.com/voldemort	Yes	Under development	No	Clients for several languages		Open source: Apache 2.0 license
	Riak http://basho.com/riak	Yes	Yes	Riak search, secondary indices	CLI and API in several languages	Provides filtering through key filters. Configurable secondary indexing. Provides Solr search capabilities. Provides server-side scripting	Open source: Apache 2.0 license
Column family stores	**Cassandra** http://cassandra.apache.org	Yes	Third party APIs	Cassandra query language	CLI and API in several languages. Supports Thrift interface	Secondary indexing mechanisms include column families, super-columns, collections	Open source: Apache 2.0 license
	HBasehttp://hbase.apache.org	Yes	Yes	No, could be used with Hive	Java/Any Writer	Server-side scripting support. Several secondary indexing mechanisms	Open source: Apache 2.0 license
	DynamoDB (Amazon service) http://aws.amazon.com/dynamodb	Amazon Elastic MapReduce	Yes	Proprietary	API in several languages	Provides secondary indexing based on attributes other than primary keys	Closed source: Pricing as pay-per-use basis
	Amazon SimpleDB (Amazon service) http://aws.amazon.com/simpledb	No	Yes	Amazon proprietary	Amazon proprietary API	Automatic indexing for all columns	Closed source: Pricing as pay-per-use basis
Document stores	**MongoDB** http://www.mongodb.org	Yes	Yes	Proprietary	CLI and API in several languages	Server-side scripting and secondary indexing support. A powerful aggregation framework	Open source: Free GNU AGPL v3.0 license
	CouchDB http://couchdb.apache.org	Yes	Yes	SQL like UnQL, under development	API in several languages	Server-side scripting and secondary indexing support	Open source: Apache 2.0 license
	Couchbase Server http://www.couchbase.com	Yes	Yes	No	Memcached API + protocol (binary and ASCII) in several languages	Server-side scripting and secondary indexing support	Open source: Free Community Edition. Paid Enterprise Edition
Graph databases	**Neo4J** http://www.neo4j.org	No	Yes	Cypher, Gremlin and SparQL	CLI and API in several languages	Server-side scripting and secondary indexing support	Open source license: NTCL + (A)GPLv3
	HyperGraphDB www.hypergraphdb.org/	No	Yes	SQL like querying	Currently has Java API. Could be used with Scala	Provides a search engine and Seco scripting IDE	Open source license: GNU LGPLv3
	Allegro Graph http://www.franz.com/agraph/allegrograph	No	Yes	SparQL and Prolog	API in several languages	Support for Solr indexing and search	Closed source: free, developer and enterprise versions
NewSQL	**VoltDB** http://voltdb.com/	No	Yes	SQL	CLI and API in several languages. JDBC support	Stored procedures are written in Java. Tables cannot join with themselves, and all joined tables must be partitioned over the same value	Open source AGPL v3.0 license. Commercial enterprise edition
	Spanner	Yes	NA	SQL like language	NA	Tables are partitioned into hierarchies, which describe locality relationship between tables	Google internal use only
	Clustrix http://www.clustrix.com/	No	No	SQL	Wire protocol compatible with MySQL		Closed source. Available as a service in the AWS marketplace, as an appliance, and as standalone software
	NuoDB http://www.nuodb.com/	No	No	SQL	CLI and drivers for most common data access APIs (JDBC, ODBC, ADO.NET). Also provides a C++ API	No support for stored procedures	Closed source. Pro and Developers editions. Available as a service in the AWS marketplace

Scaling

One of the main characteristics of the NoSQL and NewSQL data stores is their ability to scale horizontally and effectively by adding more servers into the resource pool. Even though there have been attempts to scale relational databases horizontally, on the contrary, RDBs are designed to scale vertically by means of adding more power to a single existing server [3].

With regard to what is being scaled, three scaling dimensions are considered: scaling read requests, scaling write requests, or scaling data storage. The partitioning, replication, consistency, and concurrency control strategies used by the NoSQL and NewSQL data stores have significant impact on their scalability. For example, partitioning determines the distribution of data among multiple servers and is therefore a means of achieving all three scaling dimensions.

Another important factor in scaling read and write requests is replication: storing the same data on multiple servers so that read and write operations can be distributed over them. Replication also has an important role in providing fault tolerance because data availability can withstand the failure of one or more servers. Furthermore, the choice of replication model is also strongly related to the consistency level provided by the data store. For example, the master–slave asynchronous replication model cannot provide consistent read requests from slaves.

Finally, another influential factor in scaling read and write requests is concurrency control. Simple read/write lock techniques may not provide sufficient concurrency control for the read and write throughput required by NoSQL and NewSQL solutions. Therefore, most solutions use more advanced techniques, such as optimistic locking with multi-version concurrency control (MVCC).

In the following subsections, partitioning, replication, consistency, and concurrency control strategies of NoSQL and NewSQL data stores will be compared; an overview is presented in Table 2.

Table 2 Partitioning, replication, consistency, and concurrency control capabilities

NoSQL Data Stores		Partitioning	Replication	Consistency	Concurrency control
Key-value stores	**Redis**	Not available (planned for Redis Cluster release). It can be implemented by a client or a proxy.	Master–slave, asynchronous replication.	Eventual consistency. Strong consistency if slave replicas are solely for failover.	Application can implement optimistic (using the WATCH command) or pessimistic concurrency control.
	Memcached	Clients' responsibility. Most clients support consistent hashing.	No replication. Repcached can be added to memcached for replication.	Strong consistency (single instance).	Application can implement optimistic (using CAS with version stamps) or pessimistic concurrency control.
	BerkeleyDB	Key-range partitioning and custom partitioning functions. Not supported by the C# and Java APIs at this time.	Master–slave.	Configurable.	Readers-writer locks.
	Voldemort	Consistent hashing.	Masterless, asynchronous replication. Replicas are located on the first *R* nodes moving over the partitioning ring in a clockwise direction.	Configurable, based on quorum read and write requests.	MVCC with vector clock.
	Riak	Consistent hashing.	Masterless, asynchronous replication. The built-in functions determine how replicas distribute the data evenly.	Configurable, based on quorum read and write requests.	MVCC with vector clock.
Column family stores	**Cassandra**	Consistent hashing and range partitioning (known as order preserving partitioning in Cassandra terminology) is not recommended due to the possibility of hot spots and load balancing issues.	Masterless, asynchronous replication. Two strategies for placing replicas: replicas are placed on the next *R* nodes along the ring; or, replica 2 is placed on the first node along the ring that belongs to another data centre, with the remaining replicas on the nodes along the ring in the same rack as the first.	Configurable, based on quorum read and write requests.	Client-provided timestamps are used to determine the most recent update to a column. The latest timestamp always wins and eventually persists.
	HBase	Range partitioning.	Master–slave or multi-master, asynchronous replication. Does not support read load balancing (a row is served by exactly one server). Replicas are used only for failover.	Strong consistency.	MVCC.
	DynamoDB	Consistent hashing.	Three-way replication across multiple zones in a region. Synchronous replication.	Configurable.	Application can implement optimistic (using incrementing version numbers) or pessimistic concurrency control.
	Amazon SimpleDB	Partitioning is achieved in the DB design stage by manually adding additional domains (tables). Cannot query across domains.	Replicas within a chosen region.	Configurable.	Application can implement optimistic concurrency control by maintaining a version number (or a timestamp) attribute and by performing a conditional put/delete based on the attribute value.
Document Stores	**MongoDB**	Range partitioning based on a shard key (one or more fields that exist in every document in the collection). In addition, hashed shard keys can be used to partition data.	Master–slave, asynchronous replication.	Configurable. Two methods to achieve strong consistency: set connection to read only from primary; or, set *write concern* parameter to "Replica Acknowledged".	Readers–writer locks.
	CouchDB	Consistent hashing.	Multi-master, asynchronous replication. Designed for off-line operation. Multiple replicas can maintain their own copies of the same data and synchronize them at a later time.	Eventual consistency.	MVCC. In case of conflicts, the winning revision is chosen, but the losing revision is saved as a previous version.
	Couchbase Server	A hashing function determines to which bucket a document belongs. Next, a table is consulted to look up the server that hosts that bucket.	Multi-master.	Within a cluster: strong consistency. Across clusters: eventual consistency.	Application can implement optimistic (using CAS) or pessimistic concurrency control.
Graph databases	**Neo4J**	No partitioning (cache sharding only).	Master–slave, but can handle write requests on all server nodes. Write requests to slaves must synchronously propagate to master.	Eventual consistency.	Write locks are acquired on nodes and relationships until committed.

	Hyper GraphDB	Graph parts can reside in different P2P nodes. Builds on autonomous agent technologies.	Multi-master, asynchronous replication. Agent style communication based on Extensible Messaging and Presence Protocol (XMPP) .	Eventual consistency.	MVCC.
	Allegro Graph	No partitioning (federation concept which aims to integrate graph databases is abstract at the moment).	Master–slave.	Eventual consistency.	Unclear how locking is implemented “100% Read Concurrency, Near Full Write Concurrency”.
NewSQL	**VoltDB**	Consistent hashing. Users define whether stored procedures should run on a single server or on all servers.	Updates executed on all replicas at the same time.	Strong consistency.	Single threaded model (no concurrency control).
	Spanner	Data partitioned into tablets. Complex policies determine in which tablet the data should reside.	Global ordering in all replicas (Paxos state machine algorithm).	Strong consistency.	Pessimistic locking in read-write transactions. Read-only transactions are lock-free (versioned reads).
	Clustrix	Consistent hashing. Also partitions the table indices using the same approach.	Updates executed on all replicas at the same time.	Strong consistency.	MVCC.
	NuoDB	No partition. The underlying key-value store can partition the data, but it is not visible by the user.	Multi-master (distributed object replication). Asynchronous.	Eventual consistency.	MVCC.

Partitioning

Most NoSQL and NewSQL data stores implement some sort of horizontal partitioning or sharding, which involves storing sets or rows/records into different segments (or shards) which may be located on different servers. In contrast, vertical partitioning involves storing sets of columns into different segments and distributing them accordingly. The data model is a significant factor in defining strategies for data store partitioning. For example, vertical partitioning segments contain predefined groups of columns; therefore, data stores from the column-family category can provide vertical partitioning in addition to horizontal partitioning.

The two most common horizontal-partitioning strategies are range partitioning and consistent hashing. *Range partitioning* assigns data to partitions residing in different servers based on ranges of a partition key. A server is responsible for the storage and read/write handling of a specific range of keys. The advantage of this approach is the effective processing of range queries, because adjacent keys often reside in the same node. However, this approach can result in hot spots and load-balancing issues. For example, if the data are processed in the order of their key values, the processing load will always be concentrated on a single server or a few servers. Another disadvantage is that the mapping of ranges to partitions and nodes must be maintained, usually by a routing server, so that the client can be directed to the correct server. BerkeleyDB, Cassandra, HBase, and MongoDB implement range partitioning as depicted in Table 2.

In *consistent hashing*, the dataset is represented as a circle or ring. The ring is divided into a number of ranges equal to the number of available nodes, and each node is mapped to a point on the ring. Figure 2 illustrates consistent hashing on an example with four nodes N1 to N4. To determine the node where an object should be placed, the system hashes the object’s key and finds its location on the ring. In the example from Figure 2, object *a* is located between nodes N4 and N1. Next, the ring is walked clockwise until the first node is encountered, and the object gets assigned to that node. Accordingly, object *a* from Figure 2 gets assigned to node N1. Consequently, each node is responsible for the ring region between itself and its predecessor; for example, node N1 is responsible for data range 1, node N2 for data range 2, and so on. With consistent hashing, the location of an object can be calculated very fast, and there is no need for a mapping service as in range partitioning. This approach is also efficient in dynamic resizing: if nodes are added to or removed from the ring, only neighbouring regions are reassigned to different nodes, and the majority of records remain unaffected [16]. However, consistent hashing negatively impacts range queries because neighbouring keys are distributed across a number of different nodes. Voldemort, Riak, Cassandra, DynamoDB, CouchDB, VoltDB, and Clustrix implement consistent hashing.

Figure 2 Consistent hashing.

The in-memory stores analyzed, Redis and Memcache, do not implement any partitioning strategy and leave it to the client to devise one. Amazon SimpleDB, the NoSQL solution which is provided as a service, offers its clients simple, manual mechanisms for partitioning data, as described in Table 2. However, the service provider might implement additional partitioning to achieve the throughput capacity specified in the service level agreement.

Partitioning graph databases is significantly more challenging than partitioning other NoSQL stores [50]. The key-value, column-family, and document data stores partition data according to a key, which is known and relatively stable. In addition, data are accessed using a lookup mechanism. In contrast, graphs are highly mutable structures, which do not have stable keys. Graph data are not accessed by performing lookups, but by exploiting relations among entities. Consequently, graph partitioning attempts to achieve a trade-off between two conflicting requirements: related graph nodes must be located on the same server to achieve good traversal performance, but, at the same time, too many graph nodes should not be on the same server because this may result in heavy and concentrated load. A number of graph-partitioning algorithms have been proposed [50], but their adoption in practice has been limited. One of the reasons is the rapid pace of changes in graphs, which may trigger intensive rebalancing operations. For this reason, the graph databases investigated, Neo4J, HypergraphDB, and AllegroGraph, do not offer partitioning in the traditional sense. However, Neo4J offers cache sharding, while HypergraphDB relies on autonomous agents to provide communication among graphs residing in different peer nodes, as summarized in Table 2.

The NewSQL data stores investigated also use diverse partitioning strategies. VoltDB uses a traditional approach in which each table is partitioned using a single key and rows are distributed among servers using a consistent hashing algorithm. Stored procedures can be executed on a single partition or on all of them; however, the drawback is that the user is responsible for selecting between these options. The Clustrix data store also partitions the data using a consistent hashing algorithm over a user-defined primary key. In addition, Clustrix also partitions the table indices using the indexed columns as the keys. Theoretically, this strategy enables parallel searches over these indices, leading to faster query resolution.

Google's Spanner uses a different partitioning model. A Spanner deployment contains a set of servers known as *spanservers*, which are the nodes responsible for serving data to clients. A *spanserver* manages hundreds to thousands of *tablets*, each of which contains a set of directories. A directory is basically a set of rows that shares a common key prefix, as specified by the user-defined table hierarchy mentioned in Section "Data Models". A directory is also considered to be the basic unit of placement configuration, which is used to define constraints for data partitioning and replication among the available tablets. Some of the criteria that can be defined are the datacentres where replicas should reside, the number of replicas, the distance of the data to their clients, and the distance among replicas. The data store automatically moves the directories among the *spanservers* to respect these criteria and to improve general data access performance.

NuoDB is another NewSQL solution that uses a completely different approach for data partitioning. A NuoDB deployment is made up of a number of Storage Managers (SM) and Transaction Managers (TM). The SMs are the nodes responsible for maintaining the data, while the TMs are the nodes that process the queries. Each SM has a complete copy of the entire data, which basically means that no partitioning takes place within the SM. Nevertheless, the underlying key-value store used by the SMs can partition the data by itself, although this is neither controllable nor viewable by the user.

Replication

In addition to increasing read/write scalability, replication also improves system reliability, fault tolerance, and durability. Two main approaches to replication can be distinguished: master–slave and multi-master replication.

In *master–slave replication*, shown in Figure 3.a, a single node is designated as a master and is the only node that processes write requests. Changes are propagated from the master to the slave nodes. Examples of data stores with master–slave replication are Redis, BerkeleyDB, and HBase. In *multi-master replication*, illustrated in Figure 3.b, multiple nodes can process write requests, which are then propagated to the remaining nodes. Whereas in master–slave replication the propagation direction is always from master to slaves, in multi-master replication, propagation happens in different directions. CouchDB and Couchbase Server are examples of *multi-master data stores*. Three other data stores, Voldemort, Riak, and Cassandra, support *masterless* replication, which is similar to multi-master replication as multiple nodes accept write requests, but as highlighted by the term *masterless*, all nodes play the same role in the replication system. Note that all three of the data stores with masterless replication use consistent hashing as a partitioning strategy. The strategy for placing replicas is closely related to node position on the partitioning ring, as shown in Table 2.

Figure 3 Replication models.

NewSQL replication schemes can be considered as multi-master or masterless schemes because any node can receive update statements. In VoltDB and Clustrix, a transaction/session manager receives the updates, which are forwarded to all replicas and executed in parallel. On the other hand, Google Spanner uses the Paxos state-machine algorithm [29] to guarantee that a sequence of commands will be executed in the same order in all the replica nodes. Note that Paxos is a distributed algorithm without central arbitration, which differs significantly from the other solutions. Finally, in NuoDB, the table rows are represented as in-memory distributed objects which communicate asynchronously to replicate their state changes.

The choice of replication model impacts the ability of the data store to scale read and write requests. Master–slave replication is generally useful for scaling read requests because it allows the many slaves to accept read requests – examples are BerkeleyDB and MongoDB. However, some data stores such as HBase do not permit read requests on the slave nodes. In this case, replication is used solely for failover and disaster recovery. In addition, *master–slave data stores* do not scale write requests because the master is the only node that processes write requests. An interesting exception is the Neo4J database, which is able to handle write requests on the slave nodes also. In this case, write requests are synchronously propagated from slaves to master and therefore are slower than write requests to master. Finally, *multi-master* and *masterless* replication systems are usually capable of scaling read and write requests because all nodes can handle both requests.

Another replication characteristic with a great impact on data stores throughput is how write operations are propagated among nodes. Synchronization of replicas can be synchronous or asynchronous. In *synchronous* or *eager replication*, changes are propagated to replicas before the success of the write operation is acknowledged to the client. This means that synchronous replication introduces latencies because the write operation is completed only after change propagation. This approach is rarely used in NoSQL because it can result in large delays in the case of temporary loss or degradation of the connection. In *asynchronous* or *lazy replication*, the success of a write operation is acknowledged before the change has been propagated to replica nodes. This enables replication over large distances, but it may result in nodes containing inconsistent copies of data. However, performance can be greatly improved over synchronous replication. As illustrated in Table 2, the majority of the data stores studied use asynchronous replication. Typically, NoSQL solutions use this approach to achieve the desired performance, yet CouchDB uses it to achieve off-line operation. In CouchDB, multiple replicas can have their own copies of the same data, modify them, and then synchronize these changes at a later time.

Consistency

Consistency, as one of the ACID properties, ensures that a transaction brings the database from one valid state to another. However, this section is concerned with consistency as used in the CAP theorem, which relates to how data are seen among the server nodes after update operations. Basically, two consistency models can be distinguished: strong and eventual consistency. *Strong* or *immediate consistency* ensures that when write requests are confirmed, the same (updated) data are visible to all subsequent read requests. Synchronous replication usually ensures strong consistency, but its use can be unacceptable in NoSQL data stores because of the latency it introduces. Among the observed NoSQL data stores with replication, HBase is the only one exclusively supporting strong consistency. In the *eventual consistency* model, changes eventually propagate through the system given sufficient time. Therefore, some server nodes may contain inconsistent (outdated) data for a period of time. Asynchronous replication, if there are no other consistency-ensuring mechanisms, will lead to eventual consistency because there is a lag between write confirmation and propagation. Because NoSQL data stores typically replicate asynchronously, and eventual consistency is often associated with them, it was expected that the reviewed NoSQL solutions provide eventual consistency. Nevertheless, as illustrated in Table 2, the majority of these data stores allow configuration of the consistency model using alternate consistency-ensuring mechanisms; however, choosing strong consistency may have a performance impact.

The data stores with consistent hashing and masterless replication, specifically Voldemort, Riak, and Cassandra, use a quorum approach in their consistency models. In this approach, a read or write quorum is defined as the minimum number of replicas that must respond to a read or write request for it to be considered successful and confirmed to the requestor. Even though these data stores are designed for eventual consistency, they can achieve strong consistency by choosing *(read quorum + write quorum)* larger than the number of replicas.

MongoDB can achieve strong consistency using two different techniques. First, a connection can be set to read-only from the master, which removes the data-stores' ability to scale read requests. The second option is to set the *write concern* parameter to "*Replica Acknowledged*", which ensures that a write succeeds on all replicas before being confirmed. This makes the data store into a synchronous replication system and degrades its performance.

Finally, it is important to note that the NewSQL solutions analyzed, with the exception of NuoDB, are strongly consistent, fully transactional data stores.

Concurrency control

Concurrency control is of special interest in NoSQL and NewSQL data stores because they generally need to accommodate a large number of concurrent users and very high read and/or write rates. All the solutions studied facilitate concurrency by implementing partitioning and replication. However, this section focuses on concurrency control as a means of achieving simultaneous access to the same entity, row, or record on a single server node.

The main concurrency-control schemes can be categorized as pessimistic or optimistic. *Pessimistic concurrency control*, or *pessimistic locking*, assumes that two or more concurrent users will try to update the same record or object at the same time. To prevent this situation, a lock is placed onto the accessed entity so that exclusive access is guaranteed to a single operation; other clients trying to access the same data must wait until the first one finishes its work. The entity that is locked depends on the underlying data model. For example, key-value stores lock records consisting of key-value pairs, column-family stores lock rows, and document stores enforce locking at document level. In graph databases, specifically in Neo4J, locks are acquired on nodes and their relationships. BerkeleyDB and MongoDB implement readers-writer locks which allow either multiple readers to access data or a single writer to modify them. Pessimistic locking techniques can lead to performance degradation, especially in write-intensive scenarios.

Optimistic concurrency control or *optimistic locking* assumes that conflicts are possible, but rare. Therefore, instead of locking the record, the data store checks at the end of the operation to determine whether concurrent users have attempted to modify the same record. If a conflict is identified, different conflict-resolution strategies can be used, such as failing the operation immediately or retrying one of the operations. Several of the data stores investigated, including Voldemort, Riak, HBase, CouchDB, Clustrix, and NuoDB, implement optimistic concurrency control with *multi-version concurrency control* (MVCC). In MVCC, when the data store needs to update a record, it does not overwrite the old data, but instead adds a new version and marks the old version as obsolete. Multiple versions are stored, but only one is marked as current. With the MVCC approach, a read operation sees the data the way they were when it began reading, even if the data were modified or deleted by other operations in the meantime.

A number of NoSQL solutions allow applications to implement optimistic concurrency control by providing primitives such as *check and set* (CAS) in Memcached and Couchbase Server. The CAS method ensures that a write will be performed only if no other client has changed the record since it was last read. In Redis, the WATCH primitive performs a similar function. Optimistic concurrency-control implementations use various approaches to determine whether a record has been changed. For example, Memcached uses version stamps and AmazonDB incrementing version numbers. Often it is hard to tell which approach a data store uses internally to achieve check and set functionality based solely on the system documentation.

Cassandra has been recognized for its ability to handle large numbers of write requests [19], and therefore architecture characteristics contributing to Cassandra’s write scalability are highlighted. Although the storage structure in typical relational databases and a number of NoSQL data stores including MongoDB and CouchDB relies on a B-Tree, Cassandra takes advantage of a log-structured merge tree. When a write occurs, Cassandra stores the changes in two places: in the memory structure called memtable, and in the commit log on disk by appending to the existing data. When the memtable reaches a threshold, the memtable data are flashed to SSTables (sorted string tables) on disk, and data in the commit log corresponding to the flushed memtable are purged. When flashing the memtable, Cassandra writes entire sectors to disk using sequential I/O instead of modifying rows in place. This approach eliminates locking of data on disk for concurrency control because write operations only append data and do not modify existing data on disk. Consequently, Cassandra is especially suitable for applications with high write volume or those that require very fast writes.

Some of the NewSQL solutions analyzed also implement innovative approaches to concurrency control. For example, Google's Spanner uses a hybrid approach in which read-write transactions are implemented through read-write locks, but read-only transactions are lock-free. This is possible because Spanner stores multiple versions of data, and a read transaction is basically a read at a "safe" timestamp. On the contrary, VoltDB implements an interesting alternative to concurrency control. This data store assumes that the total available memory is large enough to store the entire data store. Moreover, it also assumes that all user transactions are short-lived and can be very efficiently executed over in-memory data. Based on these assumptions, all transactions are then executed sequentially in a single-threaded, lock-free environment.

Security

Security is an important aspect of data stores that is overlooked by many NoSQL implementations. In this section, the data stores surveyed are compared with regard to the following features:

- Authentication: mechanisms that enable verification of the identity of users who are accessing the data. This is usually achieved through a password associated with a user's login, but more sophisticated mechanisms are also possible, such as user certificates. For many enterprises, an important requirement for authentication is the capacity of integration with enterprise user-directory systems such as Lightweight Directory Access Control (LDAP) / Active Directory and Kerberos servers.
- Authorization: this refers to the capability to ensure access control to the data-store resources. Authorization is usually performed through association of each user with a set of permissions. For example, some data stores might require specific permissions for read and write requests on tables, creation of users, and execution of administrative functions. Authorization information might also be included in directory systems.
- Encryption: this refers to mechanisms that encrypt data so that they cannot be read by attackers and others unauthorized parties. A complete encryption solution should be present in at least three different levels:
  - Data at rest: data stored on disks can be read if an attacker has access to the servers' file systems. A data-at-rest encryption mechanism guarantees that the users' data are automatically encrypted when written to these files and unencrypted when retrieved.
  - Client-to-server communication: Most data stores allow remote connections of users and applications so that stored data can be obtained. This data flow must also be encrypted to guarantee private and secure communication.
  - Server-to-server connections: because many NoSQL and NewSQL data stores include some sort of replication and distributed processing functionalities, communications among the server nodes can also be eavesdropped to obtain unauthorized access to data. A server-to-server encryption mechanism guarantees that these flows cannot be read.
- Auditing: auditing functionalities are usually related to the creation of an audit trail that logs records of events that occurred in a data stores. This is especially important in forensic analysis of security events. Many security standards, such as PCI-DSS [51] and HIPAA [52], require the existence of audit trails.

Table 3 shows a summary of the security features found in the solutions surveyed. It is worth mentioning that very often the system documentation mentions nothing about some of the criteria analyzed, especially server-to-server communication and data-at-rest encryption. In these cases, the corresponding cells in the table contain “NA”

Table 3 Security features

NoSQL Data Stores		Encryption			Authentication	Authorization	Auditing
		Data at Rest	**Client/Server**	**Server/Server**			
Key-Value stores	**Redis**	No	No	No	Admin password sent in clear text for admin functions. Data access does not support authentication	No	No
	Memcached	NA, Memcache does store data on disk	No	No	Binary protocol supports Simple Authentication and Security Layer (SASL) authentication	No	No
	BerkeleyDB	Yes, the database needs to be created using encryption	NA, embedded data store	No	No	No	No
	Voldemort	Possibly if BerkeleyDB is used as the storage engine	No	No	No	No	No
	Riak	No	REST interface supports HTTPS. Binary protocol is not encrypted	Multiple data-centre replication can be done over HTTPS	No	No	No
Column Family Stores	**Cassandra**	Enterprise Edition only. Commit log is not encrypted	Yes, SSL based	Yes, configurable: all server-to-server communication, only between datacentres or between servers in the same rack	Yes, store credentials in a system table. Possible to provide pluggable implementations	Yes, similar to the SQL GRANT/REVOKE approach. Possible to provide pluggable implementations	Enterprise Edition only. Based on log4j framework. Logging categories include ADMIN, ALL, AUTH, DML, DDL, DCL, and QUERY. Possible to disable logging for specific keyspaces
	HBase	No, planned for future release	Yes	Communication of HBase nodes with the HDFS and Zookeeper clusters can be secured. Not clear whether the HBase nodes communicate via a secure channel	Yes, RPC API based on SASL, supporting Kerberos. REST API uses a HTTP gateway, which authenticates with the data store as one single user, and executes all operations on his/her behalf	Yes, permissions include read, write, create and admin. Granularity of table, column family, or column	No, planned for future release
	Amazon DynamoDB	No	Yes, HTTPS	NA	Integration with Identity and Access Management (IAM) services. The requests need to be signed using HMAC-SHA256	Allow the creation of policies that associate users and operations on domains. Possible to define policies for temporary access	Integrates with Amazon Cloud Watch service. Access information about latencies for operations, amount of data stored, and requests throughput
	Amazon SimpleDB	See DynamoDB					No
Document Stores	**MongoDB**	No, a third-party partner (Gazzang) provides an encryption plug-in	Yes, SSL-based	Yes	Yes, store credentials in a system collection. REST interface does not support authentication. Enterprise Edition supports Kerberos	Yes, permissions include read, read/write, dbAdmin, and userAdmin. Granularity of collections	No
	CouchDB	NA	Yes, SSL-based	Possible using HTTPS connections	Yes, HTTP authentication using cookies or BASIC method. Oauth supported	Three levels of users: server admin, database admin, and database member. Complex authorization can be done in validation functions	No
	Couchbase Server	No	No	No, planned for future release	Yes, SASL authentication – each bucket is differentiated by its name and password. REST API for administrative function uses HTTP BASIC authentication	No	No
Graph databases	**Neo4J**	No	Yes, SSL-based	No	No, developers can create a SecurityRule and register with the server	No	No

	Hyper GraphDB	No	NA, embedded data store	No	No	No	No
	Allegro Graph	No	Yes, HTTPS	NA	Yes	Yes, permissions include read, write, and delete. Predefined user attributes are used to define special administration capabilities	A structure audit log can be used to record specific changes. Not clear what types of changes are logged, nor how to customize this process
NewSQL	**VoltDB**	No	No	No	Yes, users are defined in a deployment file that needs to be copied to each node	Yes, roles are defined at the schema level, and each stored procedure defines which roles are allowed to execute it	Yes, logging categories include connections, SQL statements, snapshots, exports, authentication / authorization, and others
	Spanner	NA					
	Clustrix	NA	Yes	NA	Yes, SQL-like	Yes, SQL-like	NA
	NuoDB	Native store does not support it. Theoretically, it could use a pluggable store that supports it	Yes	Yes	Yes, SQL-like	Yes, SQL-like	Yes, logging categories include SQL statements, security events, general statistics, and others

Generally speaking, it is possible to affirm that the security features of NoSQL solutions are not as mature as those included in traditional RDBMSs. Many solutions, such as Redis, Memcached, Voldemort, and Riak, are designed to be used in secure networked environments only. Therefore, they assume that it is the network administrator's responsibility to ensure that only authorized applications have access to the data store, using mechanisms such as firewalls, operating system configurations, or the adoption of virtual private networks (VPN). In these cases, there is no fine-grained access control to the data store. Furthermore, audit features are not present in most cases, and when present, they are very simple and not customizable. For example, VoltDB can log all the queries executed on its data, but it cannot constrain this logging to only a subset of the tables.

Another interesting observation is that MongoDB and Cassandra offer additional security functionalities in their enterprise editions, acknowledging the fact that security is a particularly relevant concern for large companies. For instance, data-at-rest encryption and auditing functionalities are available only in Cassandra Enterprise Edition.

Among the NewSQL solutions, Clustrix and NuoDB use the authorization and authentication schemes of traditional RDBMS by supporting the GRANT / REVOKE statements. In its turn, VoltDB implements access control to execution of stored procedures, and no information regarding Google Spanner security could be found.

Cloud data management systems may also need to handle other security related concerns, such as legal issues associated to the data location, and the complete disposal of sensitive data [53], but they are out of scope of this survey.

Use cases

Due to the diversity of NoSQL and NewSQL solutions, making the choice of the most appropriate data store for a given use case scenario is a challenging task. This section discusses some general guidelines that can be used in this task and shows examples of applications that use different data stores. The following discussion is mostly focussed on selecting a specific data model over others, but when relevant, we also examine the appropriateness of specific data stores.

Key-value stores

Generally speaking, key-value data stores are appropriate for scenarios in which applications access a set of data as a whole using a unique value as the key. Sadalage and Fowler [15] use three examples for this category: storing Web session information, user profiles and configurations, and shopping cart data. In all three cases, the data are always accessed through user identification and are never queried based on the data content. The Web session and shopping cart examples are also representatives of another common key-value use case: the stored information is needed for a limited period of time only (the duration of the user session). Indeed, in many simple Web applications, these types of data are kept in the application server's memory because of their transient nature. Nevertheless, the use of a key-value store may be appropriate in scenarios where multiple application servers access the same session information. This is a commonly used strategy to make application servers stateless and to implement high availability and scalability requirements.

Similarly, key-value data stores are useful in content providing applications. The Riak documentation [54] uses as examples of this use case an advertisement platform that provides ads based on a campaign identifier and a content provider application that retrieves images and videos based on IDs.

Key-value data stores are also suitable for object caching, especially in-memory implementations. In this case, they are used to store the results of processing intensive requests such as database queries, page rendering, and API calls. For example, Memcached is used as a caching layer for large clusters of MySQL databases in Facebook [55]. The LinkedIn service also uses a key-value data store (Voldermort) as a cache on top of their primary storage and also to store the results of intensive algorithms [43]. The use of these data stores as a caching layer is very common and is often considered an integral part of cloud applications [56,57].

It is important to note that some key-value data stores provide enhanced functionalities that may increase their applicability. For example, Redis can interpret stored values as specific data types, such as lists, sets, and strings, and also provides many primitives to manipulate these types. On the other hand, Riak enables the integration of search engines to index the stored values and the attachment of tags on keys to facilitate complex searches. These extra functionalities are also relevant when choosing the most appropriate key-value store for a particular scenario.

Finally, it is essential to recognize that key-value data stores have limitations when dealing with:

- Highly interconnected data, because all relationships need to be explicitly handled in the client applications.
- Operations that manipulate multiple items, as data are often accessed using a single key and most data stores do not provide transactional capabilities.

Document stores

Document stores can be seen as key-value stores in which the value is not completely opaque and therefore can be examined [15]. As mentioned in the “Data Model” section, these data stores manage data that can be represented as documents, which are self-describing hierarchical data structures which may contain nested objects and list attributes and do not require adherence to a fixed schema.

The first use cases for document stores are for applications dealing with data that can be easily interpreted as documents, such as blogging platforms and content management systems (CMS). Both Sadalage and Fowler [15] and the MongoDB documentation [35] use these applications as canonical examples. A blog post or an item in a CMS, with all related content such as comments and tags, can be easily transformed into a document format even though different items may have different attributes. For example, images may have a resolution attribute, while videos have an associated length, but both share name and author attributes. Moreover, these pieces of information are mainly manipulated as aggregates and do not have many relationships with other data. Finally, the capability to query documents based on their content is also important to the implementation of search functionalities.

A second significant use case for document data stores is for storing items of similar nature that may have different structures. For example, document data stores can be used to log events or monitor information from enterprise systems. In this case, each event is represented as a document, but events from different sources log different information. This is a natural fit for the flexible document data model and enables easy extension to new log formats. This contrasts with the relational approach, in which a new table needs to be created for each new format or new columns needs to be added to existing tables. As an example, Liu *et al.* [58] used CouchDB for storing and analyzing log data from a Platform as a Service (PaaS). Similarly, document data stores have also been used to store sensor network data, as suggested by Ramaswamy *et al.* [59].

Document data stores have also been chosen in scenarios in which high development productivity and low maintenance cost are essential. The flexibility of the data model mentioned in the previous paragraphs, in tandem with easy mapping of documents to object oriented constructs [60], makes these data stores especially suited for fast application development. Moreover, many modern applications provide services using REST interfaces based on JSON representations that can be directly mapped to document data stores.

Finally, it is also worth mentioning that CouchDB has been used in scenarios, such as in Havlik *et al.* [61], which specifically explore its off-line replication capabilities. CouchDB allows the co-existence of multiple instances of a database that can be updated independently and be synchronized only when the instances can communicate with each other. This characteristic is explored in applications where servers and clients are not always on-line and also to provide low latency and local data access to remote clients.

Document data stores have similar limitations to key-value data stores, such as the lack of built-in support for relationships among documents and transactional operations involving multiple documents.

Column-family stores

Due to differences in the data models of the analyzed column-family stores, the use cases for this category will be discussed in two groups. The first group contains data stores which do not use the column-family concept, namely SimpleDB and DynamoDB, and the second group consists of HBase and Cassandra.

SimpleDB and DynamoDB are both based on a schema-free tabular model, in which each row can have different columns and a column can possibly contain more than one value. The expressiveness of this model is similar to the document-store model, but with the additional limitation that nested objects are not allowed. Therefore, SimpleDB and DynamoDB are appropriate for use cases comparable to those mentioned in the previous section - document stores. In addition, both data stores are managed services, which make them especially suitable for scenarios where the users want to avoid the cost and complexity of managing a data store.

Regarding the second group of column-family stores, both HBase and Cassandra have flexible data models, and it is difficult to choose only a few applications as representatives of their use cases. Sadalage and Fowler [15] cite event logging, CMS, and blogging platforms as column-family use cases, which are once again similar to document store examples. On the other hand, we opt to show applications and benchmarks which are diverse, but which help to show the strengths and limitations of these data stores.

As mentioned in the "Concurrency Control" section, Cassandra is a data store optimized for handling a large number of write requests, and different benchmarks have confirmed this capability. In Cooper *et al.* [62], Cassandra achieved the highest update throughput on an update heavy workload in comparison to HBase, MySQL, and Yahoo's PNUTS [63]. Similarly, Rabl *et al.* [64] showed that Cassandra can achieve good throughput on 50% / 50% read-write workloads and 99% write workloads, and most importantly, can scale linearly as a function of the number of nodes in the cluster. On this benchmark, HBase had similar scalability results, but at the cost of a much smaller throughput rate. In addition, both Cooper *et al.* [62] and Rabl *et al.* [64] stated that generally HBase can handle write requests with latency orders of magnitude faster than Cassandra, even though the opposite happens when comparing read latency. Nevertheless, a different performance comparison performed by Altoros Systems [65] showed that Cassandra and HBase had similar latency and throughput in both reads and writes and that HBase had slightly better results in most cases.

The flexibility, scalability, and high performance of these data stores, in conjunction with MapReduce support, make them a good fit for analytics scenarios. For example, Chang *et al.* [20] demonstrated the use of BigTable in two applications that are representative of this use case: Web analytics and personalized search. In the first application, webmasters instrument their pages to keep track of how visitors use them. All user actions are logged to the database, and a MapReduce task is run to aggregate and transform these data into statistics useful for the Web page administrator. In the personalized search application, all user searches and actions in diverse Google services are stored, and a MapReduce task generates profiles that are used to personalize the user interaction experience.

It is also worth mentioning that Cassandra was originally designed to fulfill the storage requirements of the inbox search application [19], which Facebook's users can use to search for conversations with specific friends or using specific terms. This application also has a write-intensive workload, but at the same time requires low-latency results when these indices are queried. More recently, Facebook has revealed that they are using HBase in applications that require high write throughput and efficient random reads [55], but they do not discuss the limitations of Cassandra in addressing these requirements. They justify the choice of HBase based on their confidence in addressing missing features using their own engineering team and in the resiliency of the system against disk failures.

Finally, the limitations of column-family data stores are similar to those of other NoSQL categories, such as the lack of built-in support for relationships and transactional operations that involve more than one row. In addition, HBase and Cassandra are not very appropriate for scenarios where queries are highly dynamic because changes in queries may impact the column-family design.

Graph databases

Graph databases are a suitable choice for the following types of applications: location-based services, recommendation engines, and complex network-based applications including social, information, technological, and biological networks [15,66]. For instance, user location history data which are used to generate patterns that associate people with their frequently visited places could be efficiently stored and queried using Neo4J in location-based socio-spatial network applications [67]. Similarly, recommendation-based systems in which users are provided directed content based on their preferences could be efficiently built using graph databases. As an example, news broadcasters could create an aggregated global profile of a user, link it with their preferences for events and news, and effectively feed personalized RSS feeds to users using a graph database like Allegrograph [68].

Moreover, graph databases are being increasingly used since the rise of large social computing platforms like YouTube, Flicker, LiveJournal, and Orkut [69]. These solutions offer graph data storage and a graph processing system which provides indexing on nodes and edges, making them very efficient in storing closely related data and performing highly complex queries similar to those involving multiple joins in relational databases [69]. Another interesting application of graph databases was proposed by Sor and Srirama [70] for memory leak detection in distributed applications. To detect memory leaks, a leak cause analysis was required, which involved finding the shortest path from leaking objects to garbage collection roots with the intention of detecting the object responsible for holding the references which are no longer used. However, their use case required implementing custom graph database solutions over existing ones due to the high reliance on shortest-path search over other kind of traversals.

NewSQL

Generally speaking, the use of NewSQL data stores is appropriate in scenarios in which traditional DBMS have been used, but which have additional scalability and performance requirements.

First, NewSQL data stores are appropriate for applications which require the use of transactions that manipulate more than one object, or have strong consistency requirements, or even both. The classical examples are applications in the financial market, where operations such as money transfers need to update two accounts automatically and all applications need to have the same view of the database. Most of the analyzed NoSQL data stores do not support multi-object transactions, and many of them are eventually consistent solutions, which make them inappropriate for these use cases.

Second, the relational model is appropriate in scenarios where the data structure is known upfront and unlikely to change. The overhead of creating a schema beforehand is compensated by the flexibility of querying the data using SQL [60], a very powerful mechanism that can be used to implement almost any kind of data manipulation.

Finally, when selecting the most appropriate solution for an application, it is essential to consider the investment already made in tools and personnel training. In this regard, NewSQL data stores are especially attractive because they are compatible with most DBMS tools and use SQL as their main interaction language.

Opportunities

Although NoSQL and NewSQL data stores deliver powerful capabilities, the large number and immense diversity of available solutions make choosing the appropriate solution for the problem at hand especially difficult. Moreover, such diversity presents challenges in obtaining a perspective on the field and establishing directions for future research. Analysis and comparison of a number of NoSQL and NewSQL solutions in this study has revealed the following opportunities for future research in the field:

A common terminology needs to be established, at least for data stores having the same data model. Different terminology makes comparison of solutions challenging. An example of a terminology discrepancy is Riak's quorum read and write requests, which are referred to as routing parameters in Voldemort. Establishing a common terminology will not only help in comparing different data stores, but will also help in understanding the concepts of a new data store when a user is switching between different NoSQL products.

It is important to create a clear distinction between the term *consistency* as used in the ACID acronym and *consistency* as used in "eventual consistency". The overloading of this term has led to the general belief that an eventual-consistency data store cannot be ACID, which Bailis *et al.* [71] have already shown is not true.

Possibilities for establishing a standard SQL-like querying mechanism need to be explored, at least for data stores having the same data model. Today, with NoSQL data stores, performing even a simple query requires significant programming expertise and often solution-specific code. Therefore, switching to another data store may require changing the majority of the application code. Solutions such as Hive [72] have provided a great help in this direction, but their use is still limited to only a few data stores such as HBase and Cassandra. Additionally, some NoSQL data stores such as Cassandra, MongoDB, and Neo4J natively provide SQL-like querying. Standardizing querying mechanisms based on the capabilities of their data models would increase adoption of NoSQL in practice and would ease migration among different solutions.

- Standardized performance benchmarking is required. The popularity of NoSQL stores for cloud data management has been growing, especially in the Big Data domain. However, little has been done to compare the performance of different solutions under different processing loads. Although there have been some attempts to establish benchmarking standards, for example the Yahoo Cloud Serving Benchmark (YCSB) [62], the adoption of these standards in practice has been limited. Establishing a benchmarking standard would help in comparing different data stores with a view to selecting one for a particular application.
- Another consideration arises from modern-day business needs. Businesses now rely heavily on business intelligence (BI) tools. Although an analysis platform called Pig [73] provides some basic analytical functionalities for NoSQL data stores, it is not yet as powerful as the BI tools available for RDBMSs. Therefore, BI tools need to provide support for NoSQL data stores to obtain the most benefit from them.
- Sophisticated security and privacy provisions are needed. The review of the security properties offered by NoSQL solutions has revealed that in comparison to relational databases, the security capabilities of NoSQL solutions are limited. It is expected that future development in this area will increase adoption of NoSQL in practice.
- Use of more than one NoSQL data store in a single application needs to be explored. This consideration arises from the fact that NoSQL is not just one product, but encompasses several different data stores, each offering features specific to a particular type of use case or data need. Therefore, to cover a wider range of application scenarios, a solution might need to incorporate more than one NoSQL data store to address the need for different kinds of data. Sadalage and Fowler [15] use the term *polyglot persistence* to refer to the use of different data stores for different purposes within the same application. As an

example of this type of work, Atzeni *et al.* [74] recently proposed a common interface for accessing key-value, document, and column-family data stores.

This list includes the prominent opportunities and illustrates the great potential for future research in this domain. It can be expected that further research, together with the use of NoSQL and NewSQL in practice, will lead to emergence of preferred solutions for specific requirements. It is also important to note the significance of documentation and a user community: better documentation, a more active user community, or both may be the deciding factors because they can effectively support application development and ease data store administration.

Conclusions

In recent years, cloud computing has emerged as a computational paradigm that can be used to meet the continuously growing storage and processing requirements of today's applications. This study has focused on the storage aspect of cloud computing systems, in particular, NoSQL and NewSQL data stores. These solutions have presented themselves as alternatives to traditional relational databases, capable of handling huge volumes of data by exploiting the cloud environment.

Specifically, this paper has reviewed NoSQL and NewSQL data stores with the objectives of providing a perspective on the field, providing guidance to practitioners and researchers to choose appropriate storage solutions, and identifying challenges and opportunities in the field. A comparison among the most prominent solutions was performed on a number of dimensions, including data models, querying capabilities, scaling, and security attributes. Use cases and scenarios in which NoSQL and NewSQL data stores have been used were discussed and the suitability of various solutions for different sets of applications was examined. The discussion of the use cases, together with the comparison of data stores, will assist practitioners in choosing the best storage solution for their needs. In addition, this work has identified challenges in the domain, including terminology diversity and inconsistency, limited documentation, sparse comparison and benchmarking criteria, occasional immaturity of solutions and lack of support, and non-existence of a standard query language.

Abbreviations

ACID, Atomicity Consistency Isolation Durability; API, Application Programming Interface; BASE, Basically Available Soft-state Eventually consistent; BI, Business Intelligence; BSON, Binary JSON; CAP, Consistency Availability Partition tolerance; CAS, Check And Set; CLI, Command Line Interface; CMS, Content Management System; CQL, Cassandra Query Language; JSON, JavaScript Object Notation; LDAP, Lightweight Directory Access Control; MVCC, Multi-Version Concurrency Control; NIST, National Institute of Standards and Technology; PaaS, Platform as a Service; RDBMS, Relational Database Management System; REST, Representational State Transfer; SM, Storage Manager; SQL, Structured Query Language; TM, Transaction Manager; VPN, Virtual Private Network; YCSB, Yahoo Cloud Serving Benchmark

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

KG contributed towards defining the survey methodology and establishing the relation of this survey with other cloud data management surveys. Also, KG studied the scaling aspect of NoSQL stores and participated in identifying the challenges and opportunities of the cloud data management. WAH contributed towards the selection and study of the NewSQL data stores. WAH studied the security aspects of all the data stores included in this study and also developed the Use Cases section. Finally, WAH worked on the contextualization of the study in the cloud computing field, and participated into the discussions about the challenges and future opportunities of the cloud data management solutions. AT contributed towards the work of choosing the NoSQL data stores included in the study and carried out the studies of cloud computing. AT studied various NoSQL data stores, their data models, and their querying capabilities in detail and also contributed towards exploring the challenges and future opportunities regarding NoSQL data stores. MAMC provided direction and advice, participated in the critical and technical revision of the manuscript. All authors read and approved the final manuscript.

References

1. Facebook Newsroom A New data center for Iowa. http://newsroom.fb.com/News/606/A-New-Data-Center-for-Iowa. Accessed 29 Sep 2013

2. Ohlhorst FJ (2013) Big Data Analytics: Turning Big Data into Big Money. John Wiley & Sons, Inc, Hoboken, New Jersey, USA

3. Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P (2007) The end of an architectural era: (it's time for a complete rewrite). Proc 33rd Int Conf Large Data Bases:1150–1160

4. Beyer MA, Laney D (2012) The Importance of "Big Data": A Definition. http://www.gartner.com/id=2057415. Accessed 29 Sep 2013

5. Agrawal D, Das S, El Abbadi A (2011) Big data and cloud computing: Current State and Future Opportunities. Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT'11. ACM Press, New York, NY, USA, pp 530–533

6. Bughin J, Chui M, Manyika J (2010) Clouds, big data, and smart assets: Ten tech-enabled business trends to watch. McKinsey Quarterly 2010:1–14

7. Mell P, Grance T (2011) The NIST definition of cloud computing. NIST special publication 800–145. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed on 29 Sep 2013

8. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state-of-the-art and research challenges. J Intern Serv Appl 1:7–18. 10.1007/s13174-010-0007-6

9. Venters W, Whitley EA (2012) A critical review of cloud computing: researching desires and realities. J Info Technol 27:179–197. 10.1057/jit.2012.17

10. Tudorica BG, Bucur C (2011) A comparison between several NoSQL databases with comments and notes. 2011 10th International Conference RoEduNet. IEEE:1–5

11. Hecht R, Jablonski S (2011) NoSQL evaluation: A use case oriented survey. Proc 2011 Int Conf Cloud Serv Computing:336–341

12. Cattell R (2011) Scalable SQL and NoSQL Data Stores. ACM SIGMOD Record 39(4):12–27

13. Pokorny J (2011) NoSQL Databases: a step to database scalability in Web environment. Int J Web Info Syst 9(1):69–82

14. Sakr S, Liu A, Batista DM, Alomari M (2011) A survey of large scale data management approaches in cloud environments. IEEE Commun Surv Tutorials 13(3):311–336

15. Sadalage PJ, Fowler M (2013) NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley, Upper Saddle River, NJ

16. Abiteboul S, Manolescu I, Rigaux P, Rousset M-C, Senellart P (2012) Web Data Management. Cambridge University Press, New York

17. Aslett M (2011) How will the database incumbents respond to NoSQL and NewSQL? https://451research.com/report-short?entityId=66963. Accessed 29 Sep 2013

18. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gen Computer Syst 25(6):599–616. http://dx.doi.org/10.1016/j.future.2008.12.001

19. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst Rev 44(2):35–40. 10.1145/1773912.1773922

20. Chang F, Dean J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, Gruber R (2006) Bigtable: A distributed structured data storage system. 7th OSDI 26:305–314

21. Gilbert S, Lynch N (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2):51–59. 10.1145/564585.564601

22. Brewer E (2012) CAP twelve years later: How the "rules" have changed. Computer 45:23–29. 10.1109/MC.2012.37

23. NOSQL meetup. Eventbrite, San Francisco. http://nosql.eventbrite.com/. Accessed 29 Sep 2013

24. Konstantinou I, Angelou E, Boumpouka C, Tsoumakos D, Koziris N (2011) On the elasticity of NoSQL databases over cloud management platforms. Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM ’11. ACM Press, New York, NY, USA, pp 2385–2388

25. Pritchett D (2008) BASE: An ACID Alternative. Queue 6:48–55. 10.1145/1394127.1394128

26. Apache CouchDB.. http://couchdb.apache.org/. Accessed 29 Sep 2013

27. Murty J (2008) Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB. O’Reilly Media, Inc

28. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: Amazon’s highly available Key-value store. ACM SIGOPS Operating Syst Rev 41:205. 10.1145/1323293.1294281

29. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D (2012) Spanner: Google’s globally-distributed database. Osdi 2012:1–14

30. VoltDB Inc (2013) VoltDB Technical Overview. 1–4. http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf . Accessed 29 Sep 2013

31. Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EPC, Madden S, Stonebraker M, Zhang Y, Hugg J, Abadi DJ (2008) H-store: a high-performance, distributed main memory transaction processing system. Proc VLDB Endowment 1(2):1496–1499

32. Clustrix Inc (2012) A New Approach: Clustrix Sierra Database Engine. 1–10. http://www.clustrix.com/Portals/146389/docs/clustrix_a_new_approach.pdf. Accessed 29 Sep 2013

33. NuoDB Greenbook Publication (2013) NuoDB Emergent Architecture. 1–20. http://go.nuodb.com/rs/nuodb/images/Greenbook_Final.pdf. Accessed 29 Sep 2013

34. DB-Engines Ranking. http://db-engines.com/en/ranking. Accessed 29 Sep 2013

35. MongoDB. http://www.mongodb.org/. Accessed 29 Sep 2013

36. Couchbase Server The NoSQL document database. http://www.couchbase.com/couchbase-server/overview. Accessed 29 Sep 2013

37. Apache HBase. http://hbase.apache.org/. Accessed 29 Sep 2013

38. Redis. http://redis.io/. Accessed 29 Sep 2013

39. Memcached. http://memcached.org/. Accessed 29 Sep 2013

40. Klophaus R (2010) Riak Core: building distributed applications without shared state. Proceedings of CUFP’10 - ACM SIGPLAN Commercial Users of Functional Programming. ACM Press, New York, NY, USA, p 1

41. Oracle Berkeley DB 12c. http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html. Accessed 29 Sep 2013

42. Neo4j - What is a Graph Database? http://www.neo4j.org/. Accessed 29 Sep 2013

43. Auradkar A, Botev C, Das S, De Maagd D, Feinberg A, Ganti P, Gao L, Ghosh B, Gopalakrishna K, Harris B, Koshy J, Krawez K, Kreps J, Lu S, Nagaraj S, Narkhede N, Pachev S, Perisic I, Qiao L, Quiggle T, Rao J, Schulman B, Sebastian A, Seeliger O, Silberstein A, Shkolnik B, Soman C, Sumbaly R, Surlaker K, Topiwala S, Tran C, Varadarajan B, Westerman J, White Z, Zhang D, Zhang J (2012) Data Infrastructure at LinkedIn. Proceedings of 2012 IEEE 28th International Conference on Data Engineering. IEEE:1370–1381

44. Buerli M (2012) The current state of graph databases. http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/08%20Graph_Databases_Survey.pdf. Accessed 29 Sep 2013

45. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Comm ACM 51(1):107–113. 10.1145/1327452.1327492

46. Cassandra Query Language (CQL) v3.1.1. http://cassandra.apache.org/doc/cql3/CQL.html. Accessed 29 Sep 2013

47. Harris S, Seaborne A (2013) SPARQL 1.1 Query Language. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/. Accessed 29 Sep 2013

48. AllegroGraph 4.11.. http://www.franz.com/agraph/allegrograph/. Accessed 29 Sep 2013

49. Battle R, Benson E (2008) Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). Web Semantics: Sci Serv Agents World Wide Web 6:61–69. 10.1016/j.websem.2007.11.002

50. Soni Madhulatha T (2012) Graph partitioning advance clustering technique. Int J Computer Sci Eng Surv 3(1):91–104. 10.5121/ijcses.2012.3109

51. PCI Security Standards Council (2010) Payment card industry (PCI) data security standard - requirements and security assessment procedures - version 2.0. https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf. Accessed 29 Sep 2013

52. Health insurance portability and accountability Act of 1996 (HIPAA). http://www.cms.gov/Regulations-and-Guidance/HIPAA-Administrative-Simplification/HIPAAGenInfo/downloads/hipaalaw.pdf. Accessed 29 Sep 2013

53. Gonzalez N, Miers C, Redígolo F, Simplício M, Carvalho T, Näslund M, Pourzandi M (2012) A quantitative analysis of current security concerns and solutions for cloud computing. J Cloud Computing: Adv Syst Appl 1:11. 10.1186/2192-113X-1-11

54. Basho Technologies (2012) From relational to riak. http://basho.com/assets/RelationaltoRiak.pdf. Accessed 11 Dec 2013

55. Borthakur D, Rash S, Schmidt R, Aiyer A, Gray J, Sen SJ, Muthukkaruppan K, Spiegelberg N, Kuang H, Ranganathan K, Molkov D, Menon A (2011) Apache hadoop goes realtime at Facebook. Proc 2011 Int Conf Manage Data - SIGMOD '11 1071. 10.1145/1989323.1989438

56. Petcu D, Macariu G, Panica S, Crăciun C (2012) Portable cloud applications—from theory to practice. Future Gen Computer Syst 29(6):1417–1430. 10.1016/j.future.2012.01.009

57. Vaquero LM, Rodero-Merino L, Buyya R (2011) Dynamically scaling applications in the cloud. ACM SIGCOMM Computer Comm Rev 41(1):45–52. 10.1145/1925861.1925869

58. Liu Z, Wang Y, Lin R (2012) A novel development and analysis solution to PaaS log by using CouchDB. 2012 3rd IEEE Int Conf Network Infrastr Digital Content:251–255

59. Ramaswamy L, Lawson V, Gogineni SV (2013) Towards a quality-centric Big data architecture for federated sensor services. 2013 IEEE Int Congr Big Data:86–93. 10.1109/BigData.Congress.2013.21

60. Redmond E, Wilson JR (2013) Seven databases in seven weeks: a guide to modern databases and the NoSQL movement. O'Reilly Media. 978-1-934356-92-0

61. Havlik D, Egly M, Huber H, Kutschera P, Falgenhauer M, Cizek M, et al. (2013) Robust and Trusted Crowd-Sourcing and Crowd-Tasking in the Future Internet. In: IFIP Advances in Information and Communication Technology, 413th edition, pp 164–176

62. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM Symposium on Cloud Computing. 154:143–154

63. Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H-A, Puz N, Weaver D, Yerneni R (2008) PNUTS: Yahoo!'s hosted data serving platform. Proc VLDB Endowment 1(2):1277–1288

64. Rabl T, Gómez-Villamor S, Sadoghi M, Muntés-Mulero V, Jacobsen HA, Mankovskii S (2012) Solving big data challenges for enterprise application performance management. Proc VLDB Endowment 5(12):1724–1735

65. Bushik S (2012) A Vendor-independent Comparison of NoSQL Databases□: Cassandra, HBase, MongoDB, Riak.. http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html. Accessed 11 Dec 2013

66. Angles R, Gutierrez C (2008) Survey of graph database models. ACM Computing Surv 40:1–39. 10.1145/1322432.1322433

67. Doytsher Y, Galon B, Kanza Y (2012) Querying socio-spatial networks on the world-wide web. Proceedings of 21st International Conference Companion on World Wide Web - WWW’12 Companion. ACM Press, New York, NY, USA, pp 329–332

68. Mannens E, Coppens S, Pessemier T, Dacquin H, Deursen D, Sutter R, Walle R (2011) Automatic news recommendations via aggregated profiling. Multimed Tools Appl 63:407–425. 10.1007/s11042-011-0844-8

69. Ho L-Y, Wu J-J, Liu P (2012) Distributed graph database for large-scale social computing. 2012 IEEE Fifth Int Conf Cloud Computing:455–462

70. Sor V, Srirama SN (2012) Evaluation of embeddable graph manipulation libraries in memory constrained environments. Proceedings of the 2012 ACM Research in Applied Computation Symposium - RACS’12. ACM Press, New York, NY, USA, pp 269–275

71. Bailis P, Fekete A, Ghodsi A, Hellerstein JM, Stoica I (2013) HAT, not CAP: highly available transactions. arXiv preprint arXiv:1302.0309

72. Thusoo A, Sarma J, Sen JN, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a Map-reduce framework. Proc VLDB Endowment 2(2):1626–1629

73. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD’08. ACM Press, New York, NY, USA, pp 1099–1110

74. Atzeni P, Bugiotti F, Rossi L (2013) Uniform access to NoSQL systems. Information systems (in press). 10.1016/j.is.2013.05.002

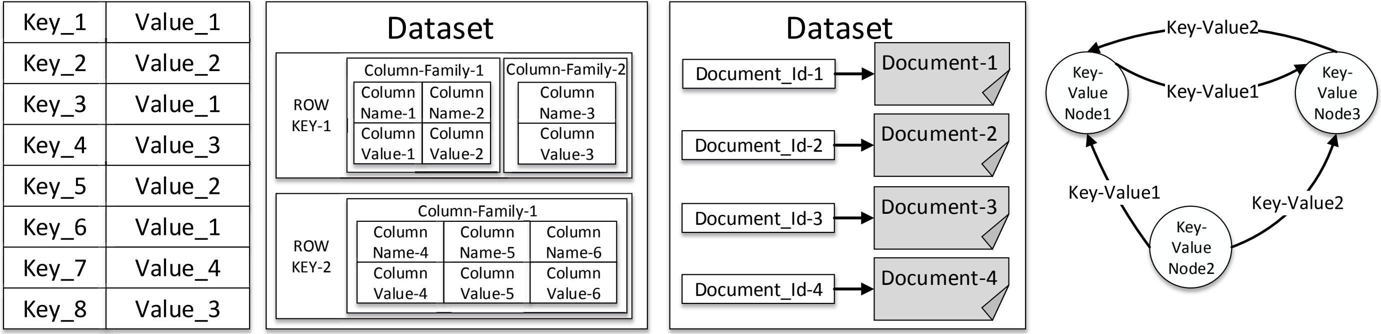

Figure 1

Key-Value Store

Column-Family Store

Document Store

Graph Databases

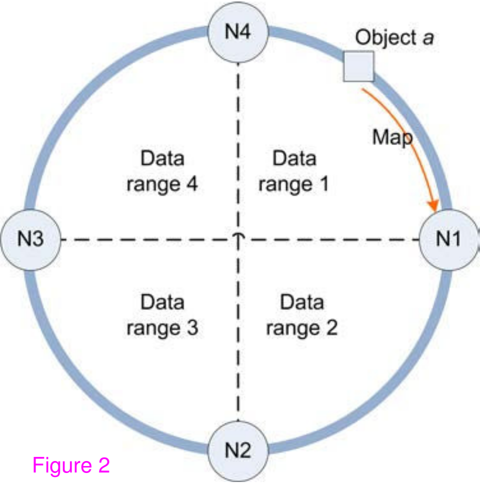
N4
Object *a*
Map
Data range 4
Data range 1
N3
N1
Data range 3
Data range 2
N2

Figure 2

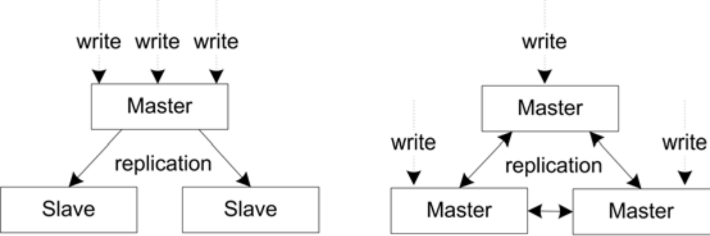

Figure 3

a) Master-slave replication

b) Multi-master replication