ISSN 0361-7688, Programming and Computer Software, 2014, Vol. 40, No. 6, pp. 323–332. © Pleiades Publishing, Ltd., 2014.
Original Russian Text © S.D. Kuznetsov, A.V. Poskonin, 2014, published in Programmirovanie, 2014, Vol. 40, No. 6.

NoSQL Data Management Systems

S. D. Kuznetsov[a] and A. V. Poskonin[b]

[a] *Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia*
[b] *Moscow State University, Moscow, 119992 Russia*
e-mail: kuzloc@ispras.ru; aposk@yandex.ru

Received May 28, 2014

Abstract—In the last decade, a new class of data management systems collectively called NoSQL systems emerged and are now intensively developed. The main feature of these systems is that they abandon the relational data model and the SQL, do not fully support ACID transactions, and use distributed architecture (even though there are non-distributed NoSQL systems as well). As a result, such systems outperform the conventional SQL-oriented DBMSs in some applications; in addition, such systems are highly scalable under increasing workloads and huge amounts of data, which is important, in particular, for Web applications. Unfortunately, the absence of transactional semantics imposes certain constraints on the class of applications where NoSQL systems can be effectively used and the choice of a particular system significantly depends on the application. In this paper, a review of the main classes of NoSQL data management systems is given and examples of systems and applications where they can be used are discussed.

DOI: 10.1134/S0361768814060152

1. INTRODUCTION

The term NoSQL was first used in 1998 by Carlo Strozzi as the name of his small relational DBMS that did not use SQL for data manipulation [1]. Starting in 2009, the term NoSQL is used for the growing number of distributed data management systems that abandoned the support of ACID transactions (Atomicity, Consistency, Isolation, Durability), which is a key principle of relational databases [2]. Strozzi believes that modern NoSQL system should be better called not relational (NoREL) [1]. Nowadays, the term NoSQL is typically interpreted as Not Only SQL [3].

The main precondition of the emergence of NoSQL systems was the growing need in the horizontal scalability of applications, that is, the need to improve the performance by adding new computational nodes to the existing ones. Thus, the majority of NoSQL systems were initially designed and created for the operation in distributed environments—a cluster or cloud—where the application of conventional SQL-oriented systems is difficult (see, e.g., [4, 5]). The main reason for the rejection of transactional semantics was the complexity of the efficient implementation in a distributed environment—in general a two-phase transaction commitment protocol is needed, which requires the transfer of a large number of messages over the network (e.g., see [6]). Although NoSQL systems typically do not fully support ACID transactions, they sometimes support, for example, atomic read-write operations, optimistic locks, and some other mechanisms that simplify the development of applications that require concurrent data access.

In this paper, we consider the systems that are classified as NoSQL data management systems. Unfortunately, it is impossible to review all available NoSQL systems (at the time this paper was written, the list [3] contained about 150 systems), and many of them are fairly complicated and have a lot of capabilities; for that reason, we only describe the key features of some specific systems, such as data model, scalability, etc. In addition, NoSQL is a new approach—many projects are at the stage of intensive development and they undergo considerable changes from version to version. The up-to-date and complete information about the systems can be found in the literature cited in this paper.

2. FEATURES OF NoSQL SYSTEMS

We have already mentioned that the majority of NoSQL systems are distributed. The distributed architecture not only ensures the horizontal scalability (ideally, the linear increase in performance as new nodes are added) but also improves the reliability of the system by maintaining several copies of data. In distributed data management systems, two basic techniques are used (almost always they are used in combination).

1. *Sharding* is an approach under which each node of the system contains a piece of data called shard and performs operations on that piece. This method is the main tool for maintaining horizontal scalability. How-

ever, operations on several objects can involve several nodes, which requires intensive data transmission over the network. In addition, as the number of nodes that store data increases, the probability of faults increases as well, which requires redundancy, that is, data replication. Sharding also gives rise to such tasks as distribution of data over the nodes, workload balancing, optimization of network communications, and others.

2. *Replication* is an approach under which the same data are stored on several nodes of the network. This method improves the system reliability and helps counteract both failures of individual nodes and entire clusters. In addition, replication can scale read operations (and more rarely write operations). A significant problem in replication is to maintain the consistency of data copies (replicas)—if the replicas are updated synchronously, then the response time of the system increases; if they are updated asynchronously, there is a time interval where the replicas are in inconsistent state.

There are two basic replication schemes (both of them support synchronous and asynchronous variations).

1. *Master–slave.* In this case, the data modification operations are processed only on the master node, and the updates are synchronously or asynchronously propagated to slave nodes. The data may be read both from the master node (which always contains the latest data version) and slave nodes that may contain outdated data if the replication is performed asynchronously.

2. *Master–master or multi-master.* In this case, all the nodes can process update operations and propagate these updates to other nodes. In this method, it is difficult to implement synchronous replication, and delays due to network communications can be significant. If the updates are performed asynchronously, there is another difficulty—conflicting data versions may occur, which require techniques for detecting and resolving conflicts (automatically or on the application level).

2.1. Consistency Models

The use of replication in a distributed system gives rise to the problem of maintaining the identical state of the copies of data on different nodes (which are visible for different clients). To describe the guarantees of data consistency, the term *consistency model* is used. Note that the term *consistency* here differs from the consistency property as it is used in the definition of ACID transactions. The consistency model describes the physical consistency of the state of the data on different nodes, while ACID assumes the logical consistency in the sense of data integrity defined in the database. Below we discuss examples of models that are often used in NoSQL systems (e.g., see [7]).

Eventual consistency guarantees that if there are no new updates, all the data replicas become consistent after certain time.

The *monotonic reads model* is a strengthening of the eventual consistency model—it guarantees that, if a value was read by a client, then the further reads will never return older values.

The *read your writes model* also strengthens the eventual consistency model—it guarantees that the client always reads the data that it wrote earlier. This model can also be combined with the monotonic reads model.

The *immediate consistency model* guarantees that as soon as the data modification operation has finished, all the clients will immediately see the updated data. For example, this model is supported by systems with synchronous replication.

Most often, NoSQL systems support the eventual consistency model or its variations. In this case, there is a time interval during which the data copies are inconsistent (it is called the *inconsistency window*). The size of the inconsistency window (if there are no failures) depends on the update propagation rate, system load, and the number of nodes storing the replicas. The applications that use data management systems that allow inconsistency must know how to cope with somewhat "outdated" data (in [8], it is discussed how often this can happen in practice in various NoSQL cloud systems). In addition, if the replicated data are modified simultaneously by different clients, conflicting data versions can occur. To detect conflicts, such approaches as timestamps and vector clocks (e.g., see [9]) are used. There are various strategies for resolving such conflicts; however, the reasonable decision about the version to be used is made by the application itself (various conflict resolving strategies are discussed, e.g., in [10]).

In some systems, the consistency model may be varied by using different configurations, parameters, and (or) operations. Let us consider how this can be achieved in practice using quorum. Suppose that the data are replicated on N nodes. Denote by R the number of nodes accessed by a client to read data and by W the number of nodes from which the confirmation of the successful data write is expected. By varying these parameters, one can configure different behavior of the distributed system.

- If $R + W > N$, then the sets of replicas for reads and writes overlap; therefore, any read operation always returns the result of a successfully completed write operation (immediate consistency).
- If $R + W \le N$, then the return of the last updated data version cannot be guaranteed; in this case, we can ensure only the eventual consistency.
- If $R = N$ and $W = 1$, then the immediate consistency is supported and the write operations are executed quickly at the expense of slower and costlier read

operations (from N replicas). In the case $W = N$ and $R = 1$, we have the opposite situation.

Note that the closer R and W are to the total number of replicas N, the greater the probability that an operation fails due to the failure of some nodes.

Variations of eventual consistency (for example, the support of monotonic reads and read your writes models) largely depend on how clients interact with the system (whether each client is assigned to a specific server, or a router node is used, are data versions supported or not, and so on). Details of consistency models in NoSQL systems are discussed, for example, in [7, 11].

2.2. CAP Theorem

To justify tradeoffs chosen in NoSQL systems, an empirical proposition known as the CAP theorem or Brewer's theorem [12] is often invoked. This statement asserts that no distributed system can simultaneously ensure the following three properties.

1. *Consistency.* All the nodes contain the consistent data at any time (all the users see the same data at any time).

2. *Availability.* When certain nodes fail, the remaining nodes continue to operate.

3. *Partition tolerance.* If the system splits into disconnected groups of nodes, each group continues to operate.

Even though this proposition is not quite formal, it was elaborated and proved for certain particular cases [13]. However, the desire to ensure partition tolerance and high system availability are not the only reasons for weakening data consistency. As noted in [14], the formulation of the CAP theorem does not take into account the important property of the system response time. Generally, strengthening the consistency model results in increased response time (indeed, recall the synchronous replication model where the data must propagate through a great number of nodes or even all nodes in order for the data modification operation to complete). It is especially important to minimize the data transmission time over the network in the case when the clusters are geographically distributed).

2.3. Data Models and Classification

One more difference between NoSQL data management systems and relational DBMSs is the nonrelational data models and query methods. On the whole, the data models underlying NoSQL systems are much simpler than the classical relational model. Often, this considerably simplifies the work with NoSQL data. Typically (with some variations) NoSQL systems are split, depending on the data model, into the following classes.

1. *Key–Value stores.*

2. *Document stores.*

3. *Systems of the Google BigTable type (Extensible Record Stores or Wide Column Stores or Column Families).*

Sometimes, the term *NoSQL* is applied to all systems that are not relational (SQL oriented); however, more often this term applies to the systems of the three classes listed above. This subdivision is fairly general as the systems within each group can be significantly different in consistency models and data manipulation methods (for example, operation atomicity, the use of locks, and multi-version concurrency control (MVCC) [18]). However, it reflects the main aspects of the fields of application of the corresponding NoSQL systems. Moreover, many systems have features of more than one class, and they are sometimes difficult to classify using this principle. Below, we give a general description of each class and consider examples of some particular systems of each class.

3. KEY–VALUE STORES

NoSQL key–value systems store (structured or unstructured) data and provide access to them using only one unique key. The data are manipulated using simple insert, delete, and search by key operations. Secondary keys and indexes are not supported. A data structure enabling one to modify individual fields of an object may be supported, but no queries can be executed against those fields. This is the key difference between the key–value and document-oriented databases [15]. In this respect, key–value systems are similar to the popular distributed in-memory caching system called Memcached [19]; however, they provide persistent data storage and some other capabilities. There are NoSQL systems that have backward compatibility with Memcached, which makes it possible to use the same interface and the same client libraries; thus, the transition from Memcached is facilitated (e.g., MemcacheDB [20], CouchDB Server [21], and others). Below, we consider some key–value stores in more detail.

3.1. Project Voldemort

Project Voldemort [22] is an open source key–value store implemented in Java, which is intensively used at LinkedIn [23]. In addition to scalar values, Project Voldemort can also manipulate lists of values and records with named fields associated with a single key. Any entity for which serialization (rules of transforming the object into a sequence of bytes and the other way around) is defined can be used as the value and the key. The data are manipulated using three operations—put, get, and delete. Project Voldemort uses the MVCC mechanism for data modification.

Project Voldemort supports sharding and replication and several versions of the physical organization of the system. To distribute the keys over the logical ring of nodes, the consistent hashing method (see [24]) is used. Sharding is implemented transparently for applications; nodes may be added or deleted during system operation; the recovery after failures is also performed automatically. Project Voldemort uses asynchronous replication and maintains eventual consistency; conflicts are resolved during read operations (read-repair). The hinted handoff mechanism is used to save data even when the nodes storing them fail; this mechanism uses adjacent nodes. Project Voldemort can keep data in RAM, and it provides persistent data storage using one of the mechanisms, such as Berkley DB [25]. The architecture of Project Voldemort is described in more detail in [26].

3.2. DynamoDB

DynamoDB [27] is a cloud service introduced by Amazon in the beginning of 2012 [28]. This system is a sequel to such Amazon technologies a Dynamo [29] and SimpleDB. DynamoDB provides a fast and scalable data management system where sharding and replication are performed automatically. The high performance of the system is achieved due to the use of SSD disks and some other optimizations. The use of the cloud service relieves users of the necessity to install and maintain servers; they only pay for the consumed resources, such as traffic and the amount of stored data.

The DynamoDB data model is fairly flexible and rich, especially for the key–value systems. The data are stored in the so-called tables, which have a primary key (a simple or composite one) and a set of attributes (there is no predefined schema; scalar data types and sets are supported). The data are manipulated using the operations of search, insertion, and deletion by the primary key, conditional operations (e.g., modify if a condition is fulfilled), atomic modifications (e.g., increment an attribute value by one), and search by non-key attributes are performed by completely scanning the table. The last operation makes the system even closer to document stores; however, DynamoDB has no efficient mechanisms for querying data by non-key attributes. The desired consistency level can be specified for data reads (eventually consistent reads or strongly consistent reads). The strongly consistent reads always return the latest data version, but their use degrades the performance. Client libraries for interacting with DynamoDB are available for many programming languages, including Java, .NET, PHP, Perl, Python, Ruby, and others).

3.3. Redis

Redis [30] is an open source key–value data management system written in C. It also supports a fairly rich data model for such kind of systems. Values can contain not only strings, but also lists and other data structures. In addition to ordinary operations of reading, writing, and deleting data by their key, Redis supports atomic operations, such as increment of a number by one, adding an element to the list, etc. In addition, transactions consisting of groups of operations having the properties of isolation and atomicity are supported, as well as optimistic locks.

Redis works in memory, which ensures high performance. To provide persistent data storage, data snapshots at certain instants of time are made or the data modification operations are immediately written to the disk; operating without using disks is also possible.

Horizontal scalability can be achieved by distributing the data (the logic is implemented at the client side). Asynchronous master–slave replication is supported.

Client libraries for Redis are available for the majority of programming languages, and the system itself is used in such large-scale projects as Twitter, Instagram, Digg, Github, StackOverflow, Flickr, and others [31].

3.4. Riak

Riak [32] is a powerful open source key–value data management system written in Erlang. The architecture of Riak was considerably influenced by Amazon Dynamo [29].

The organization of the Riak cluster is similar to that used in Project Voldemort—consistent hashing and hinted handoff mechanisms are used. Such an architecture ensures reliability, decentralization, and easy addition of new physical nodes. Several virtual nodes (vnodes) may operate on the same physical node, which helps balance the workload. Replication is performed asynchronously, the number of replicas N can be flexibly configured. Riak makes it possible to specify the number of replicas for reads (R) and writes (W) and thus vary the consistency level. However, there are no atomic operations. Riak uses a version of MVCC for concurrent access. Update conflicts can be resolved either using the principle *the last update wins* or at the application level. In the latter case, the application is provided with the conflicting versions of objects. Triggers (called commit hooks) are supported, which enables one to run JavaScript or Erlang functions before and after object modification.

Keys in Riak are organized into buckets, which makes it possible to specify such parameters as the number of replicas, the list of triggers, and a method for conflict resolution at the bucket level. Thus, the object is uniquely identified by the pair (bucket, key).

Objects in Riak are represented using JSON [33]; they may have several data fields. In addition, objects can contain metadata; in particular, references to other objects are supported. Riak supports secondary indexes (the possibility to assign a pair (attribute, value) to the object for further queries and MapReduce [34] for more complicated queries. The functional capabilities of Riak make this system close to document-oriented databases; however, Riak does not support queries on data fields (the only possibility is to use MapReduce). For that reason, Riak is typically classified as a key–value system.

Riak supports pluggable storage backends, which makes it possible to use different implementations for different applications. The user can interact with Riak using the REST interface or a programming interface available for the majority of programming languages. Riak is used in many projects (see [35]).

3.5. Aerospike

Aerospike (earlier, it was called Citrusleaf) [36] is interesting in that it is a NoSQL key–value system, supports optimistic locks, atomic operations, synchronous replication, and immediate consistency. In the case of failures, the Aerospike cluster can operate in the partition tolerant mode (the system continues to operate in partitioned mode) or in the high consistency mode in which a part of the cluster may be disconnected to avoid data inconsistency [37] if a network split occurs.

Aerospike is optimized for working in RAM and on solid state devices (Flash, SSD); the keys are always kept in RAM. To optimize network communications, client libraries are used that make use of data about the cluster state. At the highest level, the Aerospke model contains namespaces that contain sets which, in turn, contain records with a unique key and typed attributes (bins). A detailed description of Aerospike architecture can be found in [38].

3.6. Summary

At this writing, the list of NoSQL key–value systems in [3] includes more than 30 systems, including Scalaris, Tokyo Cabinet, GenieDB, LevelDB, and others. All these systems have specific features and details, and they are appropriate for different applications. When a specific system is chosen, many factors, such as the desired consistency model, support of atomic operations, ease of scaling and administration, reliability, availability of client libraries for the desired programming language, and others must be taken into account. On the whole, the advantages of key–value systems are good horizontal scalability, simplicity, and high performance. However, their data models are often insufficient for developing serious applications that require, for example, search by a combination of attributes, support of embedded objects, indexes on object fields, etc. In this case, one should consider systems with richer data models, for example, document stores.

4. DOCUMENT STORES

Document-oriented DBMSs provide richer capabilities than key–value systems. The unit of storage in document stores is a *document*; this is an object that has an arbitrary set of attributes (fields), which can be represented, e.g., in JSON [33]. Document stores support search by document fields; support indexes; often allow embedded documents and arrays; however, they do not typically have a predefined data schema. In distinction from the key–value systems, document-oriented databases provide the capability of querying collections of documents using several constraints on the attributes; they can execute aggregate queries, sort the results, support indexes on document fields, etc. Document stores typically do not support ACID semantics, and they are considerably different from each other in the support of data consistency, availability of atomic operations, methods of controlling concurrent document access, and in many other respects.

4.1. MongoDB

MongoDB [39] is an open source document-oriented data management system written in C++, which is developed by 10gen company. MongoDB has rich capabilities, and it is presently among the most popular NoSQL systems [40].

MongoDB manipulates JSON documents (which are stored and transferred as BSON—a more compact binary representation of JSON), which are joined into collections, which in turn are joined into databases. Each document in a collection must have a unique identifier (which is generated automatically or assigned by the user) and cannot be modified after the document creation. In addition to the identifier, the document may contain an arbitrary set of fields, including arrays and embedded documents. There is no predefined schema—documents in the same collection may contain completely different sets of fields.

Documents are manipulated using search, insertion, deletion, and modification operations. To find documents in a collection, queries by example are used; sorting, projection, and iterating through the query results using a cursor are supported. In addition, MapReduce and the Aggregation Framework (which is a technique for forming a query from a sequence of steps, such as aggregation, projection, sorting, etc., which makes it possible to execute complex analytical queries) are supported. A document can be completely replaced (with preserving its identifier) or some of its field can be modified (including the addition of an element to an array, increasing a number, etc.). The oper-

ation of modifying a single document is always atomic (however, if several documents are modified, this is not the case). In addition, the atomic operation *findAndModify* is supported, which finds and modifies a document and returns its old or new version. MongoDB uses locks to synchronize parallel access within one node.

To speed up document search, indexes on one or several fields in a collection (they are implemented using B-trees) and two-dimensional spatial indexes are supported. One can hint the query optimizer which index should be used and analyze the query execution plan (explain).

Scalability in MongoDB is achieved by distributing documents of a collection between nodes using a shard key. Asynchronous master–slave replication is supported—writes are processed only by the master node, and reads can be made both from the master node and from one of the slave nodes. The clients can work in non-blocking mode (without waiting for the confirmation) and blocking mode (waiting for the confirmation from a specified number of nodes). Thus, MongoDB supports different consistency models, depending on whether reads from secondary nodes are allowed and from how many nodes the confirmation must be obtained. MongoDB can also be used as a file server due to the GridFS capabilities.

Client libraries for working with MongoDB are available for many programming languages, and the REST interface is supported. This system is used in many large companies and projects, which include SourceForge, Foursquare, The Guardian, Forbes, New York Times, and others [41].

4.2. CouchDB

CouchDB [42] is an open source project of the Apache Software Foundation implemented in Erlang. CouchDB is a distributed document-oriented data management system, which manipulates JSON documents.

There is no predefined data schema—documents may contain different sets of fields (scalar fields, arrays, embedded documents, and so on; they have a unique identifier and revision. Documents are organized into databases. Reports and queries on the databases are designed using MapReduce views, which are special functions in JavaScript that allow one to specify the form of the returned data and perform aggregation; these functions are placed into special documents called design documents. Queries against views make it possible to specify conditions on the returned data, sort, limit the number of the returned results, and so on. For views, B-tree indexes are supported; indexes are updated during data modifications. Modification operations possess the ACID properties on the document level, and readers are never locked due to the use of MVCC. CouchDB supports optimistic locks when manipulating documents. Upon each data modification operation, the data are immediately saved to a disk. The data are added at the end of the file, and the older versions of the data are saved; therefore, the database must be periodically compressed (compaction); however, the system remains available both for reads and writes in the process of compaction). JavaScript functions for data validation and verifying users' rights for data modification are supported. CouchDB can be scaled only by means of replication, which is performed asynchronously. Both master–slave and master–master replication schemes are supported. In addition, there is the mechanism of filtered replication when only certain documents are replicated. Each client sees a consistent state of the database; however, these states can be different for different clients (strengthened eventual consistency). When the same document is modified on different nodes, a conflict arises. When a conflict is detected, one version of the document becomes the winner, while the looser version is preserved and can be used for the conflict resolution. There is also the CouchDB Lounge project [43], which supports sharding for CouchDB.

Clients interact with CouchDB using the REST interface. Client libraries are available for many programming languages, including Java, .NET, Python, PHP, Ruby, and others. This system is used in many projects [44].

4.3. Couchbase Server

Couchbase Server [21] is a combination of Membase (which is a key–value system compatible with Memcached) and CouchDB discussed above.

Couchbase Server can be used both for maintaining key–value data compatible with the Memcached protocol and as a document store operating with JSON documents through the REST interface. Documents can contain an arbitrary set of fields, have a unique identifier, and are stored in data buckets. Queries are expressed and executed using MapReduce views in JavaScript as in CouchDB. Views are constructed incrementally and asynchronously; therefore, by default the consistency model is eventual consistency; however, the user may specify at the operation level that the data must be indexed immediately. The data storage subsystem works in the same fashion as in CouchDB—the data are written to the end of the file, and a periodic database compaction is required.

An important feature of Couchbase Server compared with CouchDB is the support of sharding that is automatic and transparent for applications. In addition, Couchbase supports two different kinds of replication—intracluster and intercluster (Cross Datacenter Replication, XDCR). The first kind of replication is performed within a cluster, where the nodes

contain both their own data and replicas of the other nodes; immediate consistency at the document level is maintained within the cluster—this is replication in the Membase style. The second kind of replication is designed for geographically distributed clusters connected through WAN; this replication is performed asynchronously thus ensuring the eventual consistency between clusters. In this case, conflicts are resolved as in CouchDB—the same winner is selected in all the clusters.

Couchbase Server is a new and intensively developed system with rich capabilities. The latest and detailed description of Couchbase Server can be found in [45].

4.4. Summary

Document stores have a flexible data model that is sometimes more convenient than a fixed schema and better fits object-oriented programming paradigm thus reducing the gap between programming languages and the database. Systems of this class are well scalable even though they do it in somewhat different ways. Methods for expressing queries are also different, but fairly complex queries including conditions on field values, aggregation, sorting, etc. are supported. Document-oriented databases support data consistency differently; typically, they make it possible to configure the consistency model depending on the application requirements. Additional mechanisms, such as optimistic locks, atomic operations, and others, can be implemented; this makes it possible to improve data consistency for the applications where this is required. Document stores typically support persistent data storage by writing to hard disks or SSD devices, while reliability is ensured by journaling and replication. Indexes and frequently used documents are typically stored in RAM to ensure faster access. The transactional semantics at the level of several documents is typically not supported; the only option is to implement it on the application level. However, document-oriented databases gradually become closer to conventional SQL-oriented DBMSs.

At this writing, the list of document stores in [3] includes about 20 systems, including Terrastore, RethinkDB, RavenDB, etc.

5. BIGTABLE-LIKE SYSTEMS

The development of Google BigTable [46] started in 2004 to support various Google services, such as Google Earth, Google Maps, Google Analytic and others. BigTable is based on the Google File System (GFS), which is used for storing data and journals, Chubby, which is used for coordination and storing certain metadata, and other Google projects; it is not distributed outside Google, but it can be used within Google App Engine. BigTable is designed so as to be easily scalable to hundreds and thousands nodes and maintain petabytes of data.

A BigTable table is a map of the row key, column key, and timestamp to a string value. Row keys and column keys are also arbitrary strings. Row keys are lexicographically ordered, and columns are joined into column families. Column families must be defined before use; then, columns can be dynamically added to each family. Column families typically store similar data, and they are not numerous (not more than a hundred). However, each family may contain an arbitrary number of columns. Each table cell can contain several data versions marked by timestamps and ordered by the timestamps. The current value has the latest timestamp. Automatic deletion of old version is supported. Cells may contain no data at all. Thus, each table row can contain an arbitrary number of attributes (columns) joined into predefined column families.

Table rows are divided by key ranges forming relatively small segments (tablets in terms of BigTable). Tablets are distribution units when the workload is balanced. Each BigTable cluster contains one master server and servers that store tablets. The master server is responsible for the distribution of tablets over the nodes, for workload balancing, operations on the schema, etc. Column families are units to which access rights and storage parameters are applied. Data are stored by columns, and column families that are typically accessed simultaneously can be joined into the so-called locality groups to optimize reads. At the locality group level, one may, for example, specify that the data of its column family must be always kept in RAM to be quickly read and configure data compression. BigTable supports asynchronous replication between clusters thus ensuring eventual data consistency.

BigTable has operations for creating and deleting tables and column families, modifying metadata (e.g., access rights), for writing and deleting values, reading certain rows, looking through subsets of data (e.g., columns from a certain column family). Atomic operations on table rows and execution of data processing scripts are also supported. Client libraries cache metadata about tablet locations, and they mostly immediately access the nodes that store the data. BigTable can also be used with MapReduce.

Details of BigTable implementation can be found in [46]. The success of BigTable gave rise to a new family of systems that use similar approaches to ensure scalability and high performance.

5.1. HBase

HBase [47] is an open source project written in Java and developed by Apache Software Foundation. HBase adheres to the principles of BigTable, and it

uses the Apache Hadoop framework [48] for distributed computations.

Instead of GFS, HBase uses HDFS (Hadoop Distributed File System) that is a part of Apache Hadoop designed to reliably store large files with blocks distributed over nodes. In addition, other file systems can be used. HBase also supports Hadoop MapReduce. The role of the Chubby service in HBase is played by Apache ZooKeeper.

The architecture and capabilities of HBase in many respects match those of BigTable [46]; however, there are certain differences. For example, HBase supports several master servers to improve system reliability. HBase does not support the concept of locality groups—the configuration is performed at the column family level. As BigTable, HBase does not fully support ACID semantics; however, certain properties that strengthen data consistency are provided (see [49]). HBase does not support secondary indexes—records can be queried only by the primary key or by scanning the table. Nevertheless, indexes can be built manually using additional tables.

One can work with HBase using Java API, REST interface, and using Avro and Thrift. HBase is used in big applications and projects with high workload, such as Facebook (for the Facebook Messages service) and Twitter (for supporting MapReduce, people search, and other tasks).

5.2. Cassandra

Cassandra [50] was developed and used in Facebook. It is based on the ideas of Google BigTable [46] and Amazon Dynamo [29]. Presently, Cassandra is an open source project (in Java) supported by Apache Software Foundation.

In respect to the data model organization, Cassandra is similar to BigTable and HBase, but the terminology and details are somewhat different. The database in Cassandra is called a keyspace; it contains column families, which are similar to tables and provide containers for rows, which are identified by unique row keys. Rows consist of columns or super columns. The column is the minimal unit of storage in Cassandra; it consists of the name, value, and timestamp (all these fields are provided by the client). Only the latest version of data (in contrast to BigTable and HBase) is stored. In turn, super columns contain columns thus adding another level of nestedness. In addition, special columns, such as counters or columns with the specified time to live (TTL) are supported. Different rows do not necessarily have the same set of columns or super columns. Column families are stored in individual files sorted by row keys, and they must contain columns that are assumed to be accessed simultaneously.

Cassandra supports an SQL-like language for working with data (it is called CQL (Cassandra Query Language). It also supports Hadoop MapReduce. To speed up query execution, secondary indexes are supported. Data modifications are atomic at the level of one table row. Persistence is ensured by logging; data compression is also supported. Cassandra makes it possible to flexibly vary the data consistency level at the level of operations. Conflicts are resolved based on timestamps (the latest version wins).

Cassandra was designed to ensure good scalability and reliability on a large number of inexpensive (and unreliable) computers. In distinction from BigTable and HBase, Cassandra clusters have no dedicated nodes; all the nodes are peer computers and perform identical functions. Consistent caching and hinted handoff are used for distributing data over the nodes; new nodes can be easily added to clusters, and failures and recovery are performed automatically. Rows can be distributed between nodes both randomly or preserving the order. Replication is supported within each cluster and between geographically distributed clusters.

Client libraries for working with Cassandra are available for the majority of programming languages (they are based on Thrift). This system is used in many highly loaded projects [51].

5.3. Summary

The systems considered in this section in many respects adhere to the architecture and principles used in BigTable. They are designed for supporting applications with high workloads and for working on large clusters consisting of inexpensive computers. These goals are achieved by using a somewhat more complicated data model than the document model. These systems require a thorough design of column families and row keys; the implementation of certain functions (e.g., secondary indexes in HBase) is shifted to the application developer.

6. OTHER SYSTEMS

We have already mentioned that NoSQL systems include all database management systems that are not relational (SQL-oriented). Many of them emerged long before NoSQL movement became popular; often, such systems support ACID transactions and they may be not distributed, which is unusual for new systems. Nevertheless, we list some classes of such NoSQL systems not discussed in this review but sometimes treated as NoSQL. These are object-oriented DBMSs, graph-based systems, XML-oriented DBMSs, multidimensional systems, etc.

7. CONCLUSIONS

In this paper, we gave a general review of the relatively new direction in the database management

called NoSQL. The motivation of such systems was intensive development of the Web, which gave rise to applications with huge workloads and enormous amounts of data. Thus, NoSQL systems manly aim at scalability (and fault tolerance) even at the expense of weakening data consistency guarantees and abandoning transactional semantics. The data models supported by NoSQL systems are typically simpler than the relational model, and they usually do not require a fixed database schema and integrity constraints. When NoSQL systems are used, the development of applications is often simpler due to the use of simpler and more flexible data models (e.g., document model) and lower impedance mismatch (i.e., the mismatch between the object-oriented model of the programming language and the data model used in the DBMS (e.g., see [52]). Unfortunately, NoSQL systems do not suite well for the applications requiring transactional semantics.

The great variety of NoSQL systems is explained by the general trend towards specialization in the DBMS field (e.g., see [53]). Each NoSQL system is adapted for a certain class of applications. Thus, the choice of a particular solution depends on the features of the task—the expected workload, read/write ratio, structure of records to be stored, the types of queries to be executed, the desired level of data consistency, reliability requirements, the availability of client libraries for the programming language of choice, etc. In this respect, the conventional SQL-oriented DBMSs claim certain universality although their scalability is limited. In addition, NoSQL systems are still young compared to SQL-oriented DBMSs, and there is no rich experience in that field. Nowadays, new distributed SQL-oriented systems emerge that have better scalability and support SQL and ACID transactions (e.g., VoltDB [54] and H-Store [55]).

REFERENCES

1. Strozzi, C., NoSQL: A relational database management system. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Accessed March 14, 2013.
2. Gray, J., The transaction concept: Virtues and limitations, *Seventh Int. Conf. on Very Large Databases*, 1981.
3. NOSQL databases. http://nosql-database.org/. Accessed March 14, 2013.
4. Wiggins, A., SQL databases don't scale. http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/. Accessed March 14, 2013.
5. Obasanjo, D., Building scalable databases: Denormalization, the NoSQL movement and Digg. http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx. Accessed March 14, 2013.
6. Philip, B.A., Hadzilacos, V., and Goodman, N., Distributed recovery, in *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987, pp. 240–264.
7. Vogels, W., Eventually consistent, *ACM Queue*, 2008, vol. 6, no. 6.
8. Wada, H., Fekete, A., Zhaoy, L., Lee, K., and Liu, A., Data consistency properties and the tradeoffs in commercial cloud storages: The consumers' perspective, in *Conf. on Innovative Data Systems Research,* 2011.
9. Baldoni, R. and Raynal, M., Fundamentals of distributed computing—a practical tour of vector clock systems. http://net.pku.edu.cn/~course/cs501/2008/reading/a_tour_vc.html. Accessed March 14, 2013.
10. Douglas, T.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., and Hauser, C.H., Managing update conflicts in Bayou, a weakly connected replicated storage system, in *SOSP'95 Proceedings of the fifteenth ACM symposium on Operating systems principles,* New York, 1995.
11. Merriman, D., On distributed consistency. http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1. Accessed March 14, 2013.
12. Brewer, E., Towards robust distributed systems, in *ACM Symposium on the Principles of Distributed Computing,* Portland, Oregon, 2000.
13. Gilbert, S. and Linch, N., Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, *ACM SIGACT News*, 2002, vol. 33, no. 2, pp. 51–59.
14. Abadi, D., Problems with CAP, and Yahoo's little known NoSQL system. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html. Accessed March 14, 2013.
15. Cattell, R. Scalable SQL and NoSQL data stores. http://www.cattell.net/datastores/Datastores.pdf. Accessed March 14, 2013.
16. Lith, A. and J. Mattsson, J., Investigating storage solutions for large data: A comparison of well performing and scalable data storage, Goteborg, Sweden: Chalmers University Of Technology, Department of Computer Science and Engineering, 2010.
17. Strauch, C., NoSQL databases. http://www.christof-strauch.de/nosqldbs.pdf. Accessed March 14, 2013.
18. Bernstein, P.A. and Goodman, N., Concurrency control in distributed database systems, *ACM Computing Surveys*, 1981, vol. 13, no. 2, pp. 185-221.
19. Memcached - a distributed memory object caching system. http://memcached.org/. Accessed March 14, 2013.
20. Memcachedb—a distributed key-value storage system designed for persistent. http://memcachedb.org/. Accessed March 14, 2013.
21. Couchbase server. http://www.couchbase.com/. Accessed March 14, 2013.
22. Project Voldemort. http://www.project-voldemort.com/. Accessed March 14, 2013.
23. Kreps, J., Project Voldemort: Scaling simple storage at LinkedIn. http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/. Accessed March 14, 2013.
24. Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins L., and

Yerushalmi, Y., Web caching with consistent hashing. http://www8.org/w8-papers/2a-webserver/caching/paper2.html. Accessed March 14, 2013.

25. Oracle Berkeley DB. http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html. Accessed March 14, 2013.
26. Project Voldemort design. http://www.project-voldemort.com/voldemort/design.html. Accessed March 14, 2013.
27. Amazon DynamoDB. http://aws.amazon.com/dynamodb/. Accessed March 14, 2013.
28. Vogels, W., Amazon DynamoDB – a fast and scalable NoSQL database service designed for internet scale applications. http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html. Accessed March 14, 2013.
29. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., Dynamo: Amazon's highly available key-value store, in *21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, 2007.
30. Redis. http://redis.io/. Accessed March 14, 2013.
31. Who's using Redis? http://redis.io/topics/whos-using-redis. Accessed March 14, 2013.
32. Riak. http://basho.com/riak/. Accessed March 14, 2013.
33. Introducing JSON. http://json.org/. Accessed March 14, 2013.
34. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters, in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, Calif., 2004.
35. Riak users. http://basho.com/riak-users/. Accessed March 14, 2013.
36. NoSQL database, in-memory or flash optimized and web scale—Aerospike. http://www.aerospike.com/. Accessed March 14, 2013.
37. Aerospike—ACID compliant database for mission-critical applications. http://www.aerospike.com/performance/acid-compliance/. Accessed March 14, 2013.
38. Srinivasan, V. and Bulkowski, B., Citrusleaf: A real-time NoSQL DB which preserves ACID, *Very Large Databases (VLDB)*, 2010.
39. MongoDB, 10gen. http://www.mongodb.org/. Accessed March 14, 2013.
40. DB-engines ranking. http://db-engines.com/en/ranking. Accessed March 14, 2013.
41. MongoDB production deployments. http://www.mongodb.org/about/production-deployments/. Accessed March 14, 2013.
42. Apache CouchDB. http://couchdb.apache.org/. Accessed March 14, 2013.
43. CouchDB lounge. http://tilgovi.github.com/couchdb-lounge/. Accessed March 14, 2013.
44. CouchDB in the wild. http://wiki.apache.org/couchdb/CouchDB_in_the_wild. Accessed March 14, 2013.
45. Couchbase, Learn about Couchbase server. http://www.couchbase.com/learn. Accessed March 14, 2013.
46. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., and Fikes, A., Bigtable: A distributed storage system for structured data, in *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, Seattle, WA, 2006.
47. Apache HBase. http://hbase.apache.org/. Accessed March 14, 2013.
48. Apache Hadoop. http://hadoop.apache.org/. Accessed March 14, 2013.
49. Apache HBase ACID properties. http://hbase.apache.org/acid-semantics.html. Accessed March 14, 2013.
50. Apache Cassandra. http://cassandra.apache.org/. Accessed March 14, 2013.
51. Cassandra users. http://www.datastax.com/cassandrausers. Accessed March 14, 2013.
52. Neward, T., The Vietnam of computer science. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx. Accessed March 14, 2013.
53. Stonebraker, M. and Çetintemel, U., One size fits all: An idea whose time has come and gone, in *ICDE'05: Proc. of the 21st International Conference on Data Engineering*, Washington, 2005.
54. VoltDB. http://voltdb.com. Accessed March 14, 2013.
55. H-Store. http://hstore.cs.brown.edu/. Accessed March 14, 2013.

Translated by A. Klimontovich