A survey on NoSQL stores

Article *in* ACM Computing Surveys · April 2018

DOI: 10.1145/3158661

CITATIONS
56

READS
5,354

3 authors:

Ali Davoudian
Carleton University
4 PUBLICATIONS **56** CITATIONS

SEE PROFILE

Liu Chen
Carleton University
6 PUBLICATIONS **63** CITATIONS

SEE PROFILE

Mengchi Liu
Carleton University
123 PUBLICATIONS **1,124** CITATIONS

SEE PROFILE

A Survey on NoSQL Stores

ALI DAVOUDIAN and LIU CHEN, Carleton University
MENGCHI LIU, Carleton University and Wuhan University

40

Recent demands for storing and querying big data have revealed various shortcomings of traditional relational database systems. This, in turn, has led to the emergence of a new kind of complementary nonrelational data store, named as NoSQL. This survey mainly aims at elucidating the design decisions of NoSQL stores with regard to the four nonorthogonal design principles of distributed database systems: data model, consistency model, data partitioning, and the CAP theorem. For each principle, its available strategies and corresponding features, strengths, and drawbacks are explained. Furthermore, various implementations of each strategy are exemplified and crystallized through a collection of representative academic and industrial NoSQL technologies. Finally, we disclose some existing challenges in developing effective NoSQL stores, which need attention of the research community, application designers, and architects.

CCS Concepts: • **Information systems → Data management systems**; **Database design and models**; **Parallel and distributed DBMSs**;

Additional Key Words and Phrases: NoSQL, data model, consistency model, partitioning, CAP theorem, replication, elasticity, ACID

ACM Reference format:
Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2, Article 40 (April 2018), 43 pages.
https://doi.org/10.1145/3158661

1 INTRODUCTION

Database technologies have so far experienced several major generations including hierarchical and network, relational, object-oriented, NoSQL, and NewSQL in chronological order. Figure 1 illustrates the continuous development of major database technologies and some representative database systems.

- The first generation occurred in the mid-1960s and early 1970s. It aims at providing a convenient and efficient way to store and access persistent data through developing general-purpose database management systems. These early systems are based on the hierarchical

This work was partly supported by the Natural Sciences and Engineering Research Council of Canada Discovery grant and the Natural Science Foundation of China under grant no. 61672389.
Authors' addresses: A. Davoudian and L. Chen, Advanced Database Laboratory, School of Computer Science, 1125 Colonel By Dr, Carleton University, Ottawa, ON, Canada, K1S 5B6; emails: {alidavoudian, liuchen}@cmail.carleton.ca; M. Liu (corresponding author), Advanced Database Laboratory, School of Computer Science, 1125 Colonel By Dr, Carleton University, Ottawa, ON, Canada, K1S 5B6; School of Computer, 299 Bayi Road, Wuhan University, Wuhan, Hubei, China, 430072; email: mengchi@scs.carleton.ca.

https://doi.org/10.1145/3158661

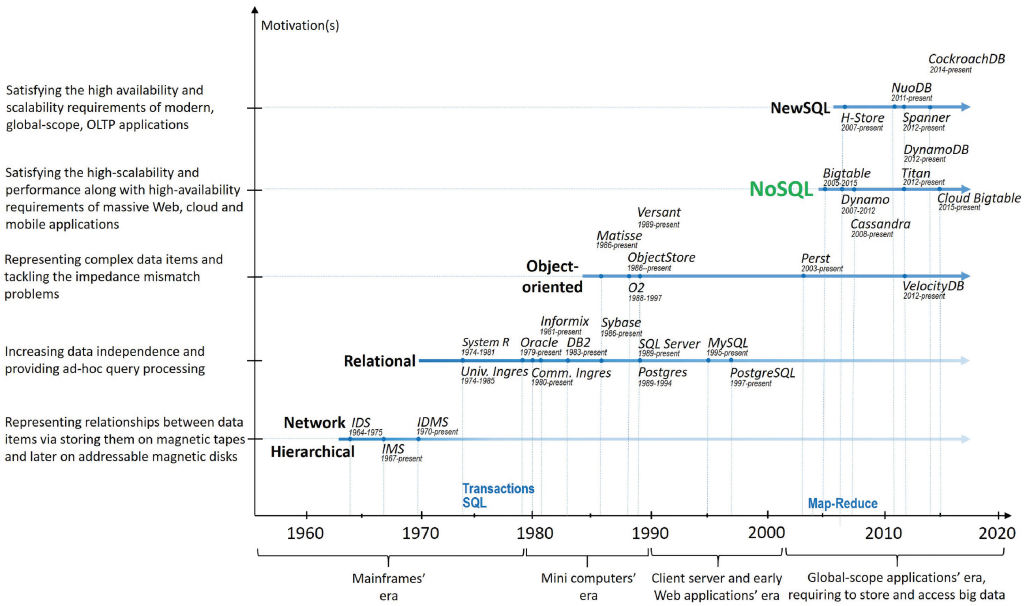

Fig. 1. The continuous development of major database technologies and some corresponding database systems.

(Tsichritzis and Lochovsky 1976) and network (DBTG Codasyl 1971; Taylor and Frank 1976) data models that are exemplified by IDS (network), IMS (hierarchical) (Meltz et al. 2004), and IDMS (network). These systems store data as records linked together so that the database has either a hierarchical or network structure. They bring high performance and throughput by providing low-level procedural operations for navigating through the maze of linked records until the target is found. As a result, data access is tied closely to how it is processed. Owing to no or limited data independence, application programs are complex to write and modify, even for simple queries.

- The second generation occurred in the early 1970s, based on the relational data model proposed by Codd (1970). This model represents structured data as tuples that are grouped into relations with a strong mathematical foundation to naturally describe data and therefore provides a substantial degree of data independence by decoupling the logical and physical representations of data. It allows people to solve various problems by means of logical and mathematical tools. Subsequently, set-oriented declarative query languages can be defined whereby application developers get rid of the burden of programming navigational access to data. Since then, Structured Query Language (SQL) (Chamberlin and Boyce 1974) has become the standard, paradigmatic language for defining, manipulating, and querying data. Due to their ease of use, these systems have replaced hierarchical and network ones and have become dominant, now widely used for various business data processing applications.
- In the early 1980s, programmers of complex database applications in domains such as Computer Aided Design/Manufacturing (CAD/CAM), Geographical Information Systems (GIS), and many other applications with complex data structures and specific operations were facing two issues. First, the relational data model has very limited modeling capabilities. Second, with regard to the increasing use of object-oriented programming languages for

application programs, software developers encountered impedance mismatch[1] (Maier 1989; Zaniolo et al. 1985), which requires mapping class objects to one or more flat relational tuples during data manipulation and many costly join operations during query processing. These issues led to the emergence of object-oriented database systems (Beeri 1990; Kim 1990) that store data as true objects identified by OIDs and classified into classes, which can be organized hierarchically so that encapsulated class structures and operations can be inherited by subclasses. In this way, the application program and the database are integrated seamlessly. However, these systems failed to become dominant due to the enormous investments put into the relational ones.

On the other hand, relational advocates tried to extend relational database systems by incorporating key object-oriented features, which resulted in object-relational database systems (Stonebraker and Moore 1995). Major relational database systems, such as Oracle and DB2, have adopted such extensions by still storing data as flat relations instead of true objects, but the mapping and joins are done automatically.

- Since the early 2000s, the advances in web technology, social networking, mobile devices, and Internet of Things have resulted in the sudden explosion of structured, semistructured, and unstructured data generated by global-scope applications. Such applications have a variety of requirements from database systems, including *horizontal scalability*[2] to linearly adapt to the massive amounts of data and the increasing rate of query processing by making use of additional resources, *high availability* and *fault tolerance* to respond to client requests even in the case of hardware/software failure or upgrade events, *transaction reliability* to support strongly consistent data, and *database schema maintainability* to reduce the cost of schema evolution.

  Achieving the above requirements through the traditional relational database systems (RDBMSs) is very difficult or even unattainable. First, the predefined relational schema makes the evolution of relational databases costly due to complex data transformation/migration; thus, this technology cannot satisfy agile and highly iterative application development approaches required by the existing processing scenarios of big data (McAfee et al. 2012). Second, scaling up these systems necessitates their movement to standalone servers with more enhanced hardware; this is a costly process and causes significant system unavailability during each movement. Third, by scaling out (i.e., employing more commodity servers) these systems even in a single datacenter, the complexity and overhead of joining distributed normalized data are increased; in addition, these systems encounter availability and performance issues owing to the use of distributed ACID transactions (Gray et al. 1996; Helland 2007).

  According to recent publications, the above mutually exclusive requirements can be tackled through making trade-offs or sacrificing the ones that are not necessary from the applications' point of view (Abadi 2012; Brewer 2000; Stonebraker 2010b). This has resulted in the fourth generation of database technologies and a new emerging trend of nonrelational data stores[3], called NoSQL[4] stores (Cattell 2011), which aim at satisfying the high availability and scalability requirements of global-scope applications. NoSQL stores have a number of design characteristics in common:

[1]This denotes the problems resulting from the difference between the programming language model and database model. This necessitates the expensive back and forth binding of data structures between applications and storage layer.
[2]It is viable through *data partitioning*.
[3]According to Cattell (2011), contemporary database systems are referred to as "data stores" in which more flexible data models are used and DBMS functionalities may not be fully provided.
[4]It is an acronym for "Not Only SQL," since some of these systems support SQL-like queries.

  - —In contrast to the above database technologies that support schema-full data models, NoSQL stores adopt more flexible data models that can be schemaless[5], and data may need to be interpreted at the application level.
  - —Weak consistency transaction models are allowed by relaxing the strict ACID properties. This allows NoSQL stores to scale out while achieving the high availability and low latency required by web-based applications.
  - —By placing relevant data items together in the same storage node and using lots of data duplication, NoSQL stores facilitate joining data as there is no need to move related data over the network, which allows them to achieve better query performance. However, this may lead to costly updates on duplicated data.
  - —There is a clever use of distributed indices, hashing and caching for data access and storage.
  - —Most of them are cluster and datacenter friendly. In other words, data can easily be replicated and horizontally partitioned across local and remote servers.
  - —They provide a web-friendly access through a simple client interface or protocol to query data.

  These features make NoSQL stores not as a revolutionary replacement for the current relational database systems, but as a remedy for their shortcomings in handling big data.
- The fifth generation happened in the late 2000s by a new emerging category of relational data stores that aim to tackle the high scalability and reliability requirements of modern OLTP applications (that handle high volumes of data beyond a single datacenter) by providing a new architecture whereby scalability and performance are improved. They are called NewSQL data stores (Stonebraker 2012), as they provide some functionalities of relational technology, such as multikey ACID transactions and SQL query language; however, the join operation is superfluous.

So far, more than 200 different NoSQL stores have been reported and the list is still growing[6]. This is a big challenge for application designers and architects who want to migrate successfully from current enterprise data management systems to large-scale ones. They need to have an in-depth understanding of existing NoSQL stores and their various features in order to select an appropriate one for satisfying their application requirements. On the other hand, by taking into account the ongoing and intense debate between relational technology and NoSQL advocates (Stonebraker 2011), NoSQL researchers, designers, and developers educate themselves about the challenges of existing NoSQL stores through determining how much they meet the expectations of contemporary, global-scope applications, and improve them to make effective data stores.

In this survey, we reflect on the work conducted with respect to the following four design aspects of NoSQL stores: *data model*, *consistency model*, *data partitioning*, and the *CAP theorem*. By taking into account their various design decisions, we have selected a collection of representative academic and industrial NoSQLs in order to exemplify and crystallize the implementations of different strategies, their strengths, limitations, and some suggested improvements. However, there are other core concepts, such as security and privacy models, that are not covered by the survey owing to space limitation. Also, graph stores (Angles and Gutierrez 2008; McColl et al. 2014) have encountered such progressive research, which necessitates a separate survey on their underlying features[7]. In addition, NewSQL is beyond the scope of this article, as we focus on NoSQL stores.

[5]A *schemaless* data model allows data to have arbitrary structures as they are not explicitly defined by a data definition language (schema-on-write). Instead, they are implicitly encoded by the application logic (schema-on-read).
[6]https://nosql-database.org/.
[7]This survey covers the graph data model and their partitioning strategies.

The rest of this survey is organized into six sections and three Appendices. Section 2 provides a classification of NoSQL stores by taking into account their underlying data models. Sections 3 and 4 categorize and explain different protocols of *data consistency* and *partitioning*, respectively. Section 5 illustrates the CAP theorem and categorizes the representative NoSQL stores based on it. We present our conclusions in Section 6. Owing to space limitation, the exemplifications of the discussed core strategies are provided in Appendix A, a review of different data replication protocols are provided in Appendix B, and well-known partitioning schemes are presented in Appendix C.

2 NOSQL DATA MODELS

A data model specifies how real-world entities and their relationships are represented and operated (Silberschatz et al. 1996). Accordingly, NoSQL stores are mainly categorized into key-value, wide-column, document, and graph stores. Each of these models is characterized by a number of features that make the corresponding stores suited to specific application scenarios.

2.1 Key-Value Stores

These are the simplest and most popular NoSQL stores, in which data are managed and represented as (*key*, *value*) pairs stored in efficient, highly scalable, *key*-based lookup structures such as Distributed Hash Tables (DHTs) (see Section 4.1.3) and Log-Structured Merge-trees (LSM-trees)[8] (O'Neil et al. 1996). A *key* is either simple (e.g., a URI, hash, or filename) or structured (e.g., composite keys in Oracle NoSQL[9]). It may also be system generated or application defined. A *value* represents data with an arbitrary type, structure, and size (e.g., a string, document, or image) that is uniquely identified by an indexed key. The *value* is encoded as a byte array (e.g., BLOB) in which its serialization/deserialization is left to the client application. Owing to the schemaless structure of stored values, indexing and querying based on *values* are not supported by the system. Any scenario that requires querying on the internal structure of data values needs to be implemented by the client application. Therefore, key-value stores are suitable solely for applications that only use a single *key* to access data, such as an online shopping cart, user profile/configuration, and web session information. This simple data model results in the easy partitioning and efficient querying of data, which, in turn, leads to the high scalability of key-value stores.

In practice, since many applications require a *value*-based lookup of data, (advanced) key-value stores provide additional functionalities, such as indexing and querying the content of *values* of specific data types. For example, Redis[10] and Aerospike[11] support the list data type. They allow performing atomic operations on list values, such as pushing into a list without replacing the entire value. IBM Spinnaker (Rao et al. 2011), HyperDex (Escriva et al. 2012), and Yahoo Pnuts (Cooper et al. 2008) support tabular data types with either a fixed or flexible schema, in which each table row is uniquely identified by a *key*. In addition, Riak KV[12] supports document types such as JSON.

[8] B^+-tree (Comer 1979) is also one of the popular key-based lookup structures that is commonly used as the physical layout of relational database systems and some NoSQL stores and libraries such as MongoDB, Tokyo Cabinet/Tyrant, and Berkeley DB. High fanout (i.e., the number of (*key*, *value*) pairs in a tree node) of the tree reduces the number of I/O operations for a query. Although B^+-trees are efficient for read operations, write-intensive workloads may result in a costly tree maintenance. In addition, random write operations cause a lot of costly disk seeks and lead to a highly degraded performance. Despite providing a better write optimization by some variations of B^+-tree (Ahn et al. 2016), an increasing number of NoSQL stores have used LSM-trees (that are optimized from the beginning) whereby random writes are transformed into sequential ones.

[9] https://oracle.com/database/nosql/index.html.

[10] https://redis.io.

[11] https://aerospike.com/.

[12] https://basho.com/products/riak-kv/.

On the other hand, Oracle NoSQL[9] supports multi-*key* operations. These extra functionalities have blurred the border between key-value stores and other kind of NoSQL stores. For example, Riak KV[12] can be considered as a document store (see Appendix A).

Based on data persistence, key-value stores can be classified into the following three kinds:

- *In-memory* key-value stores, such as Memcached[13], provide an extremely fast access to information by keeping it in memory. They are suitable for managing transient information that is needed for a limited time period. For example, transient user session information is accessed by application servers. These stores are commonly used as a caching layer in cloud applications in which the processing results of intensive requests—such as API calls, database queries, and page rendering—are stored (Petcu et al. 2013; Vaquero et al. 2011). For example, a cluster of MySQL servers[14] in Facebook uses Memcached[13] as a caching layer (Borthakur et al. 2011). In addition, Project Voldemort[15] is used as a caching layer on top of LinkedIn's primary storage (Auradkar et al. 2012).
- *Persistent* key-value stores, such as Riak KV[12] and Oracle NoSQL[9], provide a highly available access to nontransient information by storing it in HDD/SSD.
- *Hybrid* key-value stores, such as Redis[10] and Aerospike[11], first keep data in memory and then persist them when some conditions are satisfied.

A typical key-value store provides a simple set of *key*-based query operations, such as `get(key)`, `put(key,value)` and `delete(key)`. `Get(key)` retrieves a *value* (or a list of *values* with different versions) associated with the *key*. `Put(key,value)` adds the key-value pair to the store only if the *key* is not present in it. Otherwise, the stored *value* is updated with a new version. Note that updating even a part of a stored *value* requires replacing the whole *value*. `Delete(key)` removes the *key* and its associated *value(s)*. The details of the above operations depend on factors such as consistency model and indexing. These single-*key* operations do not allow manipulating multiple *values* with one operation. These operations can be easily performed through REST (Massé 2012), or Lucene[16] interfaces. Figure 2(a) shows a sample basic key-value store used by a health information management system, assuming that the medical records of patients are mainly queried based on their Social Security Numbers (SSNs), and patient's data are rarely modified.

Representative systems of key-value stores include Redis[10], Riak KV[12], Oracle NoSQL[9], Hyperdex (Escriva et al. 2012), Yahoo Pnuts (Cooper et al. 2008), Oracle Berkeley DB[17], and Project Voldemort[15], in the order of highest to lowest rank according to DB-Engines ranking[18]. It ranks database systems and stores based on parameters such as general interest according to Google Trends, the number of job offers, and the number of mentions on websites. In addition, six academic key-value stores—including Amazon Dynamo (DeCandia et al. 2007), Scalaris (Schütt et al. 2008b), Spinnaker (Rao et al. 2011), Scatter (Glendenning et al. 2011), Walter (Sovran et al. 2011), and COPS (Lloyd et al. 2011)—are investigated.

2.2 Wide-Column Stores

These stores (also known as column-family stores) are inspired by Google Bigtable (Chang et al. 2008), in which data are represented in a tabular format of *rows* and (a fixed number of) *column-*

[13] https://memcached.org/.
[14] https://mysql.com/.
[15] https://project-voldemort.com/voldemort/.
[16] https://lucene.apache.org.
[17] https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.
[18] https://db-engines.com/en/.

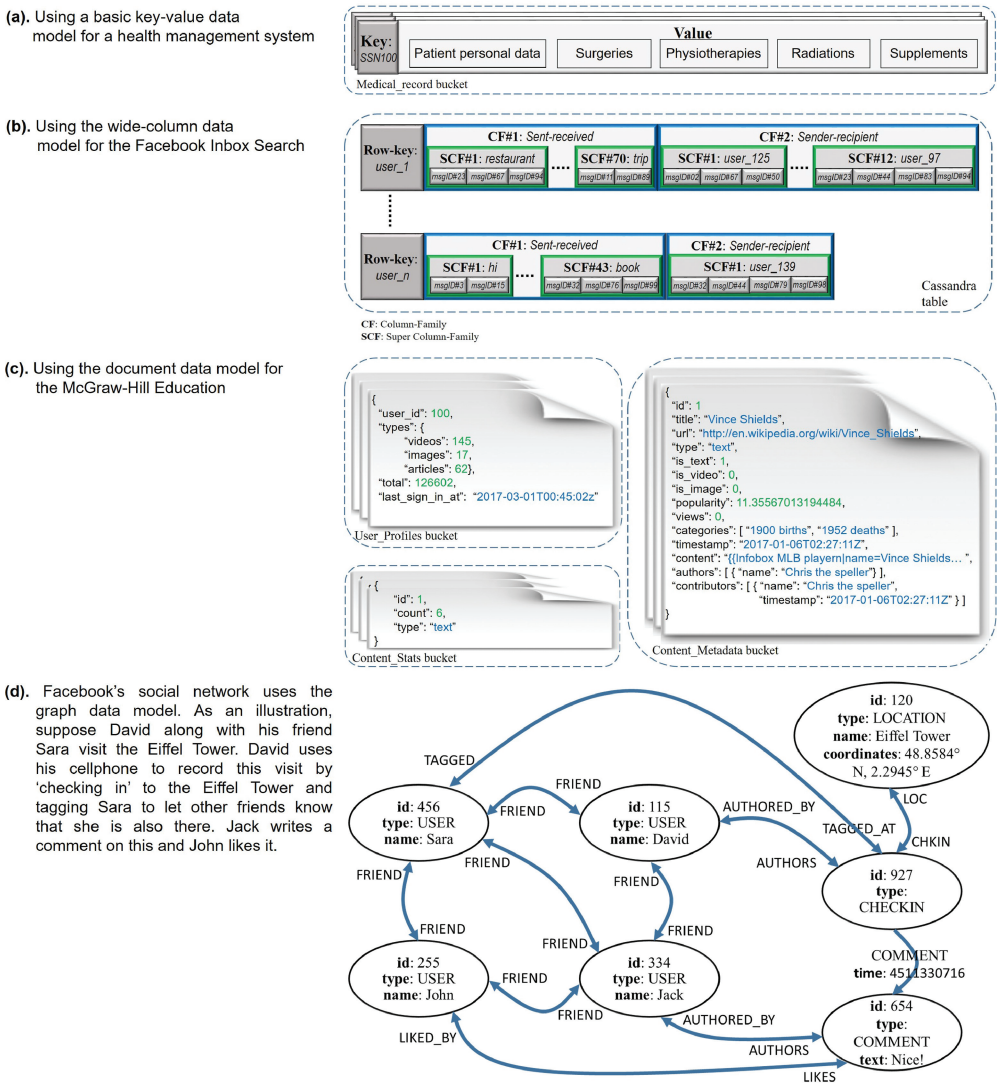

Fig. 2. Four major categories of NoSQL stores according to the supported data model.

families. A *column-family* is constructed by an arbitrary number of *columns* that are logically related to each other and usually accessed together. This justifies why data in a wide-column store are physically stored per *column-family* instead of *row*. The schema of a *column-family* is flexible as its *columns* can be added or removed at runtime. In addition, a *column* (or cell) has a name and a value with a simple or complex structure. By taking into account *column* and *column-family* structures, each *row* represents a highly structured data item that is uniquely identified by a string row-key. In other words, wide-column stores are extended key-value stores in which the value is represented as a sequence of nested (*key*, *value*) pairs.

Wide-column stores usually allow storing of a configurable number of versions of each cell value, indexed by timestamps; a value is retrieved through a triple <*row-key, column-key, timestamp*>. A timestamp is either automatically assigned by the store or explicitly specified by

the client application. Some wide-column stores provide additional *aggregates* (or embedded data structures). For instance, Apache Cassandra (Lakshman and Malik 2010) allows a *column-family* to be nested by other column-families called *super column-families*. Wide-column stores offer more extended client interfaces than key-value stores, as their indexing and querying facilities are based on various *aggregates*, such as *rows*, *column-families*, and *columns*.

With respect to the above concepts, wide-column stores support diverse modeling structures such as *rows*, *column-families*, and *nested column-families*. These structures can be used to create a hierarchy of *aggregates* based on query workload whereby query performance is increased by accessing collocated data. Storey and Song (2017) explain and exemplify how to design such hierarchies of *aggregates* based on query requirements. Figure 2(b) shows an example of how Facebook used the wide-column data model for its Inbox Search service. This service enables the user to search through one's sent/received messages based on either a keyword (called term query) or the name of a sender/receiver (called interaction query). These query requirements are facilitated by creating different *aggregates*. More precisely, for both term and interaction queries, the user-ID is the row-key. Two column-families, *Sent-received* and *Sender-recipient*, represent two different *aggregates* (with regard to the same user) that satisfy the requirements of term and interaction searches, respectively. For the *Sent-received* column-family, the keywords that make up the user's messages become super column-families. For each super column-family, the individual message-IDs (or links to messages) become the columns, which, in turn, minimizes redundancy. Similarly, for the *Sender-recipient* column-family, the user-IDs belonging to all senders/recipients of the user's messages become super column-families. Also, for each super column-family. the individual message-IDs become the columns.

Data in wide-column stores can be efficiently partitioned horizontally (by *rows*) and vertically (by *column-families*), which make them suitable for storing huge datasets. Note that many wide-column-stores, such as Apache Cassandra (Lakshman and Malik 2010), Apache Hbase[19] and Google Bigtable (Chang et al. 2008), use an LSM-tree data structure (O'Neil et al. 1996) to implement a highly efficient storage backbone per *column-family*. Figure 3 illustrates the simplest form of LSM tree architecture, including an in-memory tree (or buffer) and a number of on-disk trees (or stored files) that are immutable copies of the buffer. In this architecture, incoming written (*key*, *value*) pairs are efficiently stored and sorted (based on their keys) in the buffer (called *MemTable* in Apache Cassandra/Google Bigtable and referred to as *MemStore* in Apache Hbase[19]). When the buffer passes a threshold, such as data size, data are flushed to a stored file (called *SSTable* in Apache Cassandra/Google Bigtable and referred to as *StoreTable* in Apache Hbase) in append-only batches, which is the main reason for fast-write operations. (*key*, *value*) pairs in a stored file are sorted and indexed based on their key values. Over time, many stored files may be created (e.g., stored files 1 to *n* in Figure 3). Thus, in order to save the disk space and improve the performance of read operations, they are periodically compacted to a single file. This I/O-intensive compaction[20] removes deleted table rows and consolidates those rows that have the same row-key and scattered across multiple stored files. LSM-trees also support efficient random read operations. More precisely, read operations are first applied on the buffer before applying on files on disk. In order to find the latest version of a value, stored files are looked up in the order of latest to oldest. In addition, buffered data are first written in a commit log in order to prevent their loss in the case of node failure. Each stored file is also equipped with a Bloom filter (Bloom 1970) that prevents looking up the file for a key that does not exist in it (e.g., in Figure 3, a read operation finds a required value in the *n*th stored file). Note that owing to the immutability of stored files, updating a value just needs

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

[20] Some variations of LSM-tree enhanced the compaction process (Wu et al. 2015).

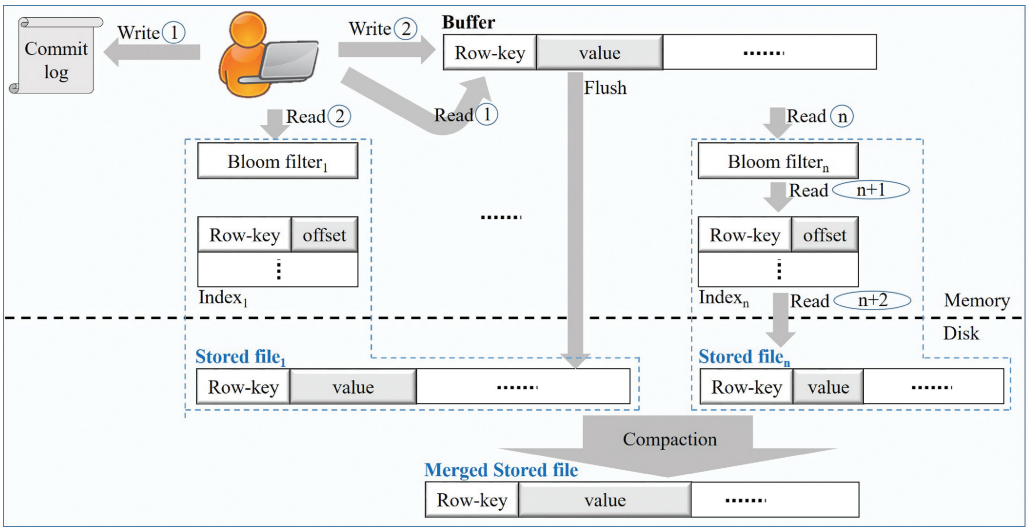

Fig. 3. The use of a simplified LSM-tree architecture per column-family in wide-column stores.

appending its new value. Therefore, reading a value necessitates accessing the most recent value. In addition, reading a whole table row may incur reading several stored files.

The high scalability and flexibility along with the support of MapReduce tasks (Dean and Ghemawat 2008) (for the parallel processing of large aggregated datasets) make wide-column stores suitable for analytical applications. Web analytics application is an example, in which pages are instrumented (by webmasters) for keeping track of their visitors' actions. More precisely, after storing all user actions to the database, they are aggregated and transformed by a MapReduce task. The resulting statistical data are suitable for the webpage administrator. However, any change in the application-level features will significantly impact the design. It limits the ability for ad-hoc querying. Also, the predefined set of *column-families* makes it difficult to use wide-column stores for applications with evolving schemas (Corbett et al. 2013; Stonebraker 2011).

Representative systems in this category include Apache Cassandra (Lakshman and Malik 2010), Apache Hbase[19], Hypertable[21], and Google Cloud Bigtable[22] in the order of highest to lowest rank.

2.3 Document Stores

These are extended key-value stores in which the value is represented as a *document* encoded in standard semistructured formats such as XML, JSON, or BSON (Binary JSON). A document has a flexible schema through adding or removing its attributes at runtime, when an attribute has a name along with one or more values. Unlike the opaque content of values in key-value stores, document stores know the format of *documents* and support indices and search functionalities based on their attribute names and values.

Document stores fit applications whose data can be simply represented in a document format such as Content Management Systems (CMS) and blogging platforms. For example, a blog post including various (nested) attributes—such as tags, comments, images, and videos—can be easily represented in a document format. These stores are also suitable for the high development

[21] http://hypertable.org/.
[22] https://cloud.google.com/bigtable/.

productivity and low maintenance cost of modern Web 2.0 applications, for two main reasons. First, these applications have a constant evolution of data schema and benefit from the flexible data model of document stores. For example, consider a monitoring application that stores and analyzes log data from various sources, with each source generating different data. Evolution to new log formats can be easily enabled by the flexible schema of documents. However, such an extension would be costly in relational databases, as it needs the creation of new tables for new formats or the addition of new columns to the existing tables. Second, Web 2.0 applications support data models such as JSON with a tight integration with popular programming languages such as Python, JavaScript, and Ruby. This has radically decreased the impedance mismatch (Maier 1989; Zaniolo et al. 1985) between these programming languages and document stores, as object-oriented constructs can be easily mapped to documents.

Some document stores, such as MongoDB[23] and Couchbase Server[24], provide an additional *aggregate* called *collection* or *bucket* that contains a set of documents representing the same category of information. These collections look like *tables* in relational databases, with each row being a document with a unique key, but not necessarily the same schema as others. By using *buckets*, features such as resources, replication, persistence, and security can be managed for each group of documents instead of individual ones. Figure 2(c) shows an example of how McGraw-Hill Education (MHE) used the document data model for building a self-adapting, interactive learning portal that delivers personalized search results[25]. It integrates Couchbase Server[24] and Elastic-Search (ES)[26] (to handle the full-text search and content discovery). The vision of MHE divides a textbook into media objects including articles, images, and videos. Data are stored in JSON documents and categorized in several buckets, including (1) the *Content_Metadata* bucket, which stores media objects' metadata along with the content of text articles; (2) the *User_Profiles* bucket, which stores user view details per media object, used for customizing ES search results based on user preferences; and (3) the *Content_Stats* bucket, which stores the view details of each media object, used for boosting ES search results based on document popularity.

Document stores allow querying data inside a document without having to retrieve the whole document (via its key) and then inspect it. The following query (based on Figure 2(c)) shows how to use the SQL-like Non-first normal form Query Language (N1QL)[27] in CouchBase to search for the documents of bucket *Content_MetaData*. This query filters the documents by attribute *title*, which has the value "Vince Shields," and then returns the value of attributes *url* and *categories*.

```
SELECT c.url, c.categories FROM Content_MetaData c WHERE title = 'Vince Shields'
```

Database designers may use either embedding or referencing documents in order to model the relationship between documents. Through embedding one-to-one or one-to-many relationships in a single document, the client application does not need to join data across documents; a single operation can retrieve all data. More precisely, a one-to-one relationship can be modeled as a flat list of attributes. On the other hand, a one-to-many relationship can be modeled either by embedding the "one" side in "many" side or embedding "many" side in "one" side. However, embedding results in the denormalization and duplication of data; this, in turn, may lead to data redundancy and consistency problems, especially in write-intensive scenarios. Thus, many-to-many relationships are usually modeled via links and require joins. In other words, multiple documents are linked via their document identifiers just like using foreign keys to relate rows of tables in relational databases.

[23] https://mongodb.org/.
[24] https://couchbase.com/nosql-databases/couchbase-server.
[25] https://youtube.com/watch?v=0mQt5gEOIhI.
[26] https://github.com/elastic/elasticsearch.
[27] https://www.couchbase.com/products/n1ql.

Some document stores, such as ArangoDB[28] and OrientDB[29], use referencing documents to implement a hybrid graph/document data model. These multimodel NoSQL stores aim at tackling applications with polyglot persistence (Sadalage and Fowler 2012; Schaarschmidt et al. 2015). These applications have a complex implementation, as they need to query and store their data using several data models simultaneously. For example, a bookstore application that requires querying books based on both their fields and the deep levels of their similarities needs to be modeled using both document and graph stores. Multimodel NoSQL stores mitigate this complexity by integrating multiple NoSQL data models into a single unique system in which a single query language and API is used.

Note that native XML stores (e.g., Marklogic Server[30] and Virtuoso[31]) are the predecessors of modern JSON document stores. They implement a variety of XML tools and standards (e.g., XQuery[32]) for XML view, storage, keyword search (Le and Ling 2016), and query processing and optimization. However, some applications have preferred JSON as an alternative to XML owing to its relative compactness, simplicity, and tight interaction with popular programming languages. In this respect, XML and JSON stores support different applications and use cases. XML stores are usually used for organizing and maintaining a collection of XML files in content management applications such as health care, science, and digital libraries, whereas JSON stores are used by more interactive and dynamic web applications.

Representative systems in this category include MongoDB[23], Amazon DynamoDB[33], Couchbase[24], Apache CouchDB[34] and ArangoDB[28] from the highest to the lowest rank.

2.4 Graph Stores

The focus of the above data models is to store information about *entities* as either binary values, rows in multidimensional tables, or documents. However, the increasing number of graph-oriented datasets—such as Semantic Web (Berners-Lee et al. 2001), Web Mining (Schenker et al. 2005), and the interaction of proteins in biological systems (Eckman and Brown 2006; Knisley and Knisley 2007)—has generated the need for efficient *entity relationship* traversals. This has motivated the emergence of graph stores for storing these datasets in an efficient manner and providing effective operations for their querying and analyzing. These stores are based on the strong graph theoretical foundation, in which a graph consists of *vertices* representing entities and *edges* representing relationships between them.

Table 1 shows a list of graph structures that are not usually mutually exclusive. For example, *property* graphs are an amalgam of *directed*, *labeled*, *attributed*, and *multigraphs*, which are widely adopted in practice. The popularity of property graphs is due to their flexibility to express other structures. For instance, by not using attributes in property graphs, a Resource Description Framework (RDF)[35] or semantic graphs are generated. Similar to edges in a graph database, an RDF graph is a set of RDF statements (or triples), each of which represents a simple relationship between two entities.

[28] https://arangodb.com.
[29] http://orientdb.com/.
[30] https://marklogic.com—Since 2014, Marklogic has started extra support for JSON documents and is considered to be a multimodel store.
[31] https://virtuoso.openlinksw.com.
[32] https://w3.org/TR/xquery-30/.
[33] https://aws.amazon.com/dynamodb/.
[34] https://couchdb.apache.org/.
[35] https://w3.org/RDF/.

Table 1. Various Graph Structures

Graph structure	Description
Undirected/directed graphs	All relationships in an undirected graph are symmetric. On the other hand, in a directed graph, each edge *e* from vertex *src* to vertex *des* is a directed tuple, such that (1) edge *e* is an *in-edge* of *des* and an *out-edge* of *src*, (2) vertex *des* is the *src*'s *out-neighbor* and vertex *src* is the *in-neighbor* of *des*, and (3) the number of *incoming/outgoing* edges of a vertex determines its *(in/out)* degree.
Labeled graphs	Vertices and edges are tagged with scalar values (labels or types) that may represent either their roles in different application domains or some attached metadata.
Attributed graphs	A variable list of attributes as (*key*, *value*) pairs are attached to vertices and edges, representing their properties. It is suitable for social networking sites that involve social interaction of individuals.
Multigraphs	Multiple edges (even with the same labels) between the same two vertices as well as self-loops are allowed.
Hypergraphs	These graphs can represent N-ary relationships through hyperedges that can connect any number of vertices (Bretto 2013). An undirected hyperedge can be represented by a subset of nodes (vertices) (Berge 1973); and a directed hyperedge can be represented by a tuple (ordered set) of nodes (Boley 1992) or head-and-tail sets (Gallo et al. 1993; Nguyen and Pallottino 1989). Hypergraphs are used to represent complex relationships in areas such as Artificial Intelligence. HyperGraphDB (Iordanov 2010) is based on hypergraphs and supports N-ary relationships by representing a link as a tuple (ordered set) of nodes. Figure 4 shows a simple hypergraph.
Nested graphs	Each vertex, in turn, can be a graph. At this time, no store supports nested graphs.

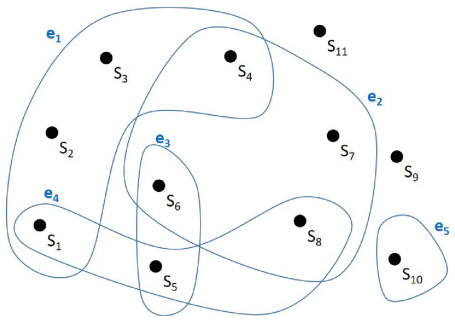

Fig. 4. Assume that I is an institute with 5 classes $(C_1, \ldots, C_5)$ and 11 students $(S_1, \ldots, S_{11})$. Also, assume that each class is attended by at least one student. The set of vertices is the set of students; the set of hyperedges is $(e_i), i \in \{1, 2, \ldots 5\}$, where e_i is the subset of students who attend C_i. There are two isolated vertices (s_9 and s_{11}), as they do not belong to any hyperedge. Incident hyperedges, such as e_1 and e_2, have some common vertices. Adjacent vertices, such as s_1 and s_2, belong to the same hyperedge.

In terms of storage techniques used by graph stores, they are categorized into *nonnative* and *native* graph stores. The logical model of a *nonnative* graph store is implemented on top of a nongraph data store, such as a document store or a relational database system, which, in turn, needs to use indices to navigate graph traversals. However, index lookups may lead to performance degradation, specially for deep traversals[36]. For example, OrientDB[29] and ArangoDB[28] use a document-storage

[36] A graph can be easily represented on top of a relational data model. However, the modelling of very large graphs may encounter performance issues, as traversing graph data requires complex join operations and the overhead of index lookups. In addition, graph traversals cannot be easily performed in SQL, specially for traversals whose depth is unbounded or unknown. Sun et al. (2015) tackle this issue by translating Gremlin graph traversal queries into SQL statements instead of directly using SQL for graph processing.

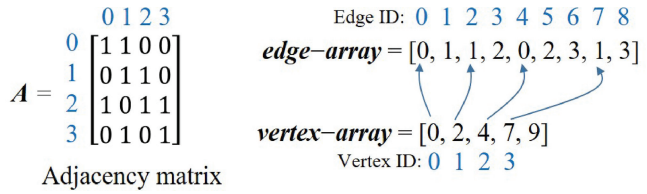

Fig. 5. Compressed Sparse Row (CSR) representation. Every graph can be represented as an adjacency matrix, which can be encoded using the CSR representation.

approach. In addition, Titan[37] allows two storage options (i.e., wide-column and key-value). Non-native stores may not use a graph-specific partitioning strategy since data partitioning is inherited from their underlying stores. For example, OrientDB[29] uses a typewise data partitioning in which vertices are considered as typed documents. By contrast, in a *native* graph store, the storage is adapted based on the features of graph data models. For example, Neo4j (Webber 2012) provides efficient real-time graph processing via a direct index-free access of adjacent nodes from a given node.

Three well-known techniques for graph-optimized storage are *Compressed Sparse Row (CSR)*, *adjacency list*, and *edge list*:

- *Compressed Sparse Row (CSR).* CSR is one of the most widely used graph representations. Assuming a graph with n vertices and m directed edges, *CSR* is composed of two integer arrays: (1) *Edge-array (E)*, with size m, that is formed from the concatenation of the adjacency lists of all vertices. It maps an edge ID (between 0 to $m-1$) into its corresponding destination vertex ID. Accordingly, all edges of a graph are stored continuously; this, in turn, improves locality as outgoing edges of a vertex are stored together and facilitates their rapid identification. (2) *Vertex-array (V)*, with size $n+1$, is an index to the array E, as it maps a vertex ID (between 0 to $n-1$) into the ID of its first outgoing edge. The number of vertices of the ith adjacency list, where $i \in \{0 \ldots n-1\}$, is $V[i+1] - V[i]$. The last element in the array V is set to m. Note that, for undirected graphs, each edge is stored twice, one for each direction. As a drawback, the cost for edge insertion or removal is $O(m)$. By default, *CSR* allows storing outgoing edges; however, a *compressed sparse column (CSC)* that is an identical representation can be used to store incoming edges of each vertex. Many graph engines, such as GraphChi (Kyrola and Guestrin 2014) and Grace (Prabhakaran et al. 2012), use CSR or one of its variants. Figure 5 illustrates a sample *CSR* data representation.
- *Adjacency list.* In this representation, each graph is represented by a set of lists. More specifically, each vertex of an undirected graph is associated with a list of its neighboring vertices and each vertex of a directed graph is associated with two lists of its *(in/out)-neighbors*. This representation reduces the storage overhead of storing sparse graphs, and large adjacency lists are usually compressed. In addition, via creating efficient index structures over edges and vertices, random accesses to data elements can be extremely fast (Wang et al. 2004; Zeng et al. 2007). However, adjacency lists incur data redundancy, as each edge is stored twice (one instance for each vertex). Therefore, for a graph of n edges, the space complexity of this representation is $2n$ nodes and $2n$ pointers. This redundancy necessitates graph stores to automatically enforce the consistency of data when the graph data is updated. In addition, there may be a costly access to the list of neighbors when they are not sequentially

[37] https://thinkaurelius.github.io/titan/.

stored on disk. By storing adjacency lists in separate (growable) arrays, this graph representation suits those systems that prioritize updates and/or optimize for online transaction processing (OLTP) workloads. Some graph engines, such as TurboGraph (Han et al. 2013), improve the locality of access by assigning the adjacency lists to pages with some extra empty space. However, there is still a need to add new pages when all the extra space is assigned. In some graph engines, such as Microsoft Trinity (Shao et al. 2013), adjacency lists are stored in a key-value store. As another example, Twitter FlockDB[38] stores its adjacency list in MySQL[14].

- *Edge list.* In the above *adjacency lists*, for any vertex *src*, an outgoing edge to a vertex *des* is represented just by *des*. In addition, all the information associated with the edge is stored along with *des*. However, there are some situations (such as the representation of hypergraphs in which an edge is associated with a set of vertices) that necessitate representing a graph as a list of edges instead of adjacency lists. This data structure can be simply represented by a *relational table* with two columns, in which each row indicates an edge and the IDs of corresponding incident vertices are stored in its columns. Through creating B-tree (Comer 1979) indices over these columns, all *incoming/outgoing* edges of a vertex can be quickly accessed. This has some advantages, as the modification of an edge requires a single access since it is stored only once. However, there are also some drawbacks, as the addition or removal of edges incurs the costly update of indices, and the space required for indices is usually more than the space needed by the table of edges. Some graph stores, such as Neo4j (Webber 2012), use this representation of graph data.

Some graph stores use more complicated representations of graph data, such as *compressed bitmap indices* in DEX[39], in which compressed, partitioned bitmaps are used to enable the efficient combination of multiple adjacency list through set operations. Figure 2(d) illustrates an example of how Facebook uses the graph data model for its social network, in which users, physical locations, relationships (such as users' friendships) and actions (such as liking, writing a comment, and checking-in to a location) are encoded with typed nodes and typed directed associations of a graph. Each node is uniquely identified by a node identifier and each association by a triple of source and destination node identifiers plus the association type. In addition, each association type is accompanied by a set of properties (including the *time* property) as key-value pairs. Note that all associations in Figure 2(d) are bidirectional (either symmetric such as *FRIEND* or asymmetric such as *AUTHORS/AUTHORED_BY*) except for the association whose type is *COMMENT*, because there is no need to traverse from the *COMMENT* to the *CHECKIN* object. Facebook implemented a distributed graph data store named TAO (Bronson et al. 2013) in which for each node a list of associations (of a given type) are stored in descending order by their *time* property. TAO uses sharded MySQL[14] databases for the persistent storage of the social graph. However, it acts as a geographically distributed graph-aware caching layer and mediates all graph accesses. TAO also keeps each bidirectional association in synch with its inverse. TAO API provides efficient access to its social graph. It supports operations for the definition and manipulation of nodes and associations. It also allows simple queries, such as reading data of a given node, and reading the most recent outgoing edge (or all edges). For example, in Figure 2(d), the *CHECKIN* object can be displayed by accessing all tagged users and the most recent comments.

[38] https://github.com/twitter/flockdb.
[39] https://sparsity-technologies.com/.

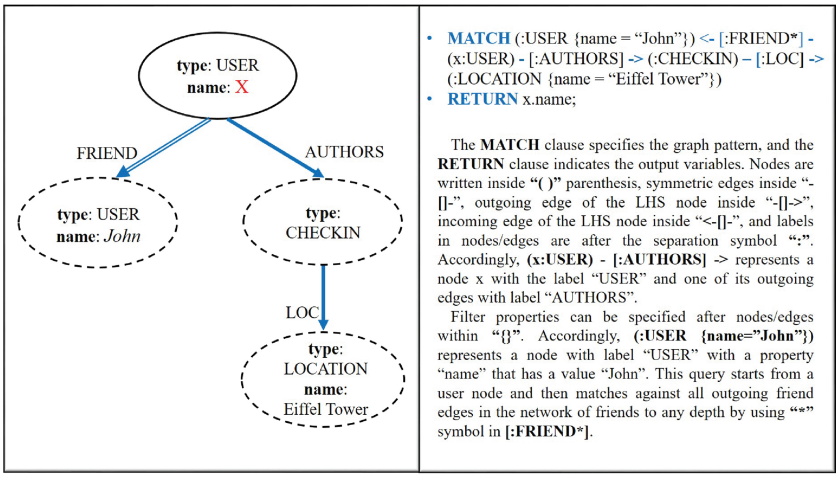

Fig. 6. A graph pattern finding the person names who are in John's network of friends and visited the Eiffel Tower.

Graph data models allow deep and complex queries over highly connected entities to be easily expressed and quickly processed. Common graph query workloads can be divided into two categories:

— *Online graph navigations*, in which small fractions of vertices and edges of the entire graph are explored and a fast response time is required. Online graph queries can be classified into two major types (Angles 2012; Angles et al. 2016): *path queries* and *pattern matching queries*.

- A *basic path query* is used to determine the existence of a path connecting two nodes of a property graph regardless of edge labels. However, in practice, path queries may specify conditions on the retrieved paths, such as specifying a regular expression over the set of (edge/vertex) labels of retrieved paths when concatenated. For example, in Figure 2(d), assume that we like to see the name of all users in *John*'s network of friends (and friends of a friend, and so on). This can be expressed with the following regular path query:

$$John \xrightarrow{\text{FRIEND}^+} x$$

  Here, the symbol '+' denotes "one-or-more," in which the regular expression $FRIEND^+$ is used to specify all paths from a sequence of one-or-more forward-directed edges with the label *FRIEND*. The endpoint x is matched to any node in the graph connected to *John* by such a path. Due to the possibility of having paths involving cycles, there may be an infinite number of paths. In order to prevent this situation, some semantics—such as looking for just the existence of a path, shortest path, no-repeated node, or no-repeated edge—are used.
- A *pattern matching query* is used to find all subgraphs of a data graph that are isomorphic (i.e., equivalent in both structure and labels) to a given pattern graph. For example, in order to find the names of all users in *John*'s network of friends who have visited the *Eiffel Tower* in Figure 2(d), a tree-shaped pattern in Figure 6(a) can be used. This pattern is closely related to graph patterns identified by Angles et al. (2016), in which a single

edge in the pattern matches a single edge in the graph with the same label; a double edge matches a path in the graph whose edges have the same label as in the pattern (e.g., the *FRIEND* edge in the pattern matches any of the following paths: *Jack* to *John*, *Sara* to *John*, and *David* to *John*); the solid nodes in the pattern are output nodes; and the dashed nodes are ordinary nodes. The pattern shows a variable *X*. By evaluating the pattern on the graph in Figure 2(d), matches of the pattern to the graph are found; for each such match, the values matched by the variable *X* are returned. This evaluation returns *David*. Note that graph-pattern matching is an NP-Complete problem (Gallagher 2006); however, tree-shaped patterns can be evaluated in polynomial time (Czerwinski et al. 2017). A detailed classification of graph queries can be found in Angles et al. (2016).

SPARQL[40], Neo4j Cypher[41] and Gremlin[42] are the three most prominent graph query languages, having graph-pattern matching capability. SPARQL is designed to query RDF graphs. It has been officially standardized by W3C since 2008. Both SPARQL and Cypher support declarative querying. On the other hand, Neo4j Cypher and Gremlin are designed to query property graphs. Gremlin is a major part of the Apache TinkerPop graph computing framework. It supports both imperative and declarative querying. Gremlin is designed more for graph traversal rather than graph-pattern matching. Figure 6(b) shows the corresponding pattern query in Cypher.

— *Offline analytical graph computations* in which significant fractions of vertices and edges of the entire graph are accessed (e.g., the investigation of graph topology and finding connected components). However, their corresponding algorithms often require iterative computations that are not supported by the basic MapReduce framework (Dean and Ghemawat 2008), as this is used for one-pass batch data processing. Instead, efficient processing of graphs is performed through graph processing tools such as Google Pregel (Malewicz et al. 2010), Apache Giraph[43], GraphLab (Low et al. 2010), and Pegasus (Kang et al. 2009). These tools prioritize *high-throughput*, offline graph computations over *low-latency*, online graph navigations. Graph analytical tools provide limited support for online database features, such as fast inserts/updates, or are immutable. However, they are designed for the fast traversal of a huge number of edges and vertices of graphs. By contrast, graph stores allow high-throughput updates on small portions of graphs by a lot of concurrent users. In general, graph analytical tools and graph stores provide different internal data structures for storing graphs. However, a few graph engines (either graph stores or analytical tools)—such as GraphChi-DB (Kyrola and Guestrin 2014), MAGS (Das et al. 2016), Microsoft Trinity (Shao et al. 2013), and Kineograph (Cheng et al. 2012)—support both analytical computation and database functionality.

Representative systems in graph stores include Neo4j (Webber 2012) and Titan[37] from the highest to the lowest rank. In addition, two academic graph stores, including Microsoft Trinity (Shao et al. 2013) and TAO (Bronson et al. 2013), will be investigated. Table 2 summarizes the characteristics of the discussed NoSQL data models.

3 CONSISTENCY MODELS

A consistency model determines the effect of concurrent operations on shared data as viewed by different clients of the system. This may dictate an ordering over the operations whether they

[40] https://w3.org/TR/sparql11-query/.
[41] https://neo4j.com/docs/developer-manual/current/cypher/.
[42] https://github.com/tinkerpop/gremlin/wiki.
[43] http://giraph.apache.org/.

Table 2. Comparison of NoSQL Data Models

	Fit scenario(s)	Strength(s)	Limitation(s)
Key-value	Objects are only accessed via a single *Key*, object caching, and where objects are not related.	Scalable, a very fast random access via *Key*, and ease of data partitioning	The responsibility of applications for the modeling of *values*, the indexing and querying of objects just by their *Keys*, and the user needs to have the key of an object in order to query it.
Wide-column	Batch-oriented parallel processing of large aggregated datasets	With regard to query workload, a hierarchy of aggregates, such as column-families, are designed that, in turn, increase the performance of queries. A suitable model for storing huge amounts of data, as it can be efficiently partitioned horizontally (by rows) and vertically (by column-families).	Limited ad-hoc querying as any change in the application-specific access patterns will impact the design to a large degree; the predefined set of column-families makes it difficult to use wide-column stores for applications with evolving schemas.
Document	Data can be easily interpreted as documents and are constantly evolving.	A rich data model to store data with arbitrary complexity, such as nested structures, arrays, and scalar values; each component of a document can be accessed via secondary indices.	There is no standard API or query languages.
Graph	There is the need to traverse several levels of relationships among intensely related data.	Fast and simple querying of linked datasets, and easy mapping of entity-relationship diagrams	There is no standard API or query languages. Partitioning of large graphs reduces the performance owing to high amount of internode communications.

are individual reads and writes in a nontransactional system or transactions enclosing multiple reads and writes in a transactional system. In a distributed storage system, there is a wide range of consistency models with different guarantees for maintaining semantical relationships between data items as well as their replicas (Viotti and Vukolić 2016). In this section, we explain various consistency models enforced by NoSQL stores. In addition, existing attempts to support multikey ACID transactions in some NoSQL stores are discussed.

3.1 Linearizability[44]

Linearizability dictates a *total*, *real-time* ordering of all operations in a nontransactional system in which a read operation observes the result of the most recent updates (Herlihy and Wing 1990; Lipton and Sandberg 1988). Likewise, strict serializability (Papadimitriou 1979) (or external consistency (Gifford 1981))—which is the ultimate amount of isolation in transactional systems—ensures the same ordering as linearizability, but on transactions. Both linearizability and strict serializability are the strongest form of consistency in nontransactional and transactional systems, respectively. With regard to replicated and/or partitioned data, these models provide an abstraction of a single, consistent image of the system. In other words, the behavior of operations makes the illusion of a single copy of data. This abstraction simplifies distributed systems' programming and prevents anomalous behaviors of applications, such as dirty reads, lost updates, and nonrepeatable reads (Gray and Reuter 1992). However, it necessitates using hard-to-scale and expensive strategies either for the commit of updates among replicas (via consensus protocols, such as *2-Phase Commit (2PC)* (Gray and Reuter 1992; Skeen 1981) and Paxos commit protocol (Gray and Lamport 2006)) or pessimistic concurrency control (such as *Strict 2-Phase Locking (S2PL)* (Eswaran et al. 1976)). These

[44]This is informally referred to as strong consistency.

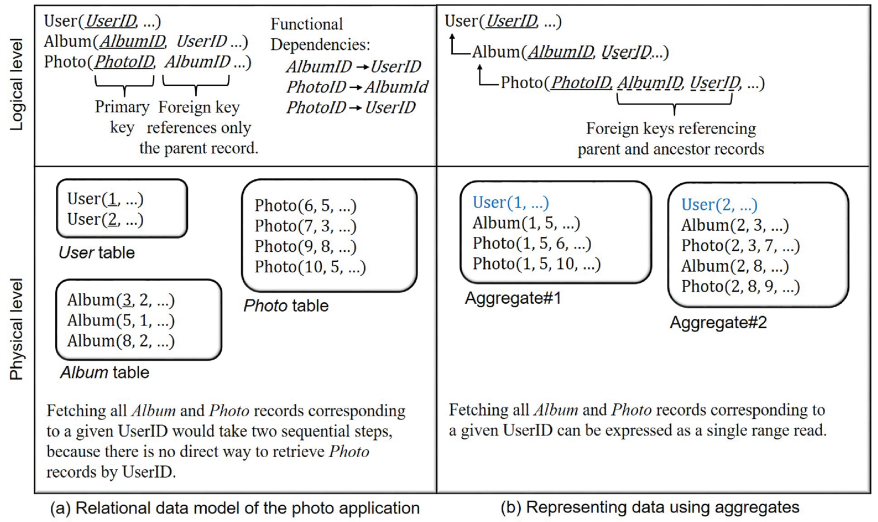

Fig. 7. Normalized data scattered in *users*', *albums*', and *photos*' tables are denormalized and represented via aggregates.

strategies severely impact availability and performance, especially in wide-area networks (Abadi 2012; Brewer 2012).

Likewise, sequential consistency enforces a ***total*** ordering of all operations in a nontransactional system, along with providing a single, consistent image of the system (Attiya and Welch 1994; Lamport 1979). However, it is weaker than linearizability, as a read operation may observe a stale value. Serializability (Papadimitriou 1979)—which is a weaker form of transaction isolation compared to strict serializability—ensures the same ordering as sequential consistency. It is typically implemented using optimistic reads and writes based on *Multi-version Concurrency Control (MVCC)* (Reed 1978), which increases the concurrency of transactions. However, the monotonic ordering of operations necessitates using a global sequencer mechanism, which can be a performance bottleneck in wide-area environments.

As providing a single, consistent image of the whole data in a distributed storage system is highly expensive, some NoSQL stores enforce a limited transactional support by introducing *atomic aggregates*. As discussed in Section 2, *aggregates* allow storing denormalized data together rather than having scattered normalized data. A single business entity can be represented by an *aggregate* such as a key-value pair in key-value stores, a column-family, super column-family, the whole row of a table in column-family stores, or a document in document stores. With regard to *atomic aggregates*, some NoSQL stores—such as Scatter (Glendenning et al. 2011), IBM Spinnaker (Rao et al. 2011), and HyperDex (Escriva et al. 2012)—enforce the strict serializability of transactions over *individual* data items (see Appendix A). In addition, Google Bigtable (Chang et al. 2008), preserves the serializability of transactions over *individual* rows of tables (see Appendix A).

For example, Figures 7(a) and 7(b) depict how users', albums', and photos' data accessed by a photo application are logically and physically represented in both *traditional relational* and *clustered hierarchical* schemas, respectively. Both schemas have the same collection of three schematized tables at the logical level: *User* with primary key (*UserID*), *Album* with primary key (*AlbumID*), and *Photo* with primary key (*PhotoID*). The traditional relational schemas are in 3rd normal form. However, the *clustered hierarchical schema* incurs data redundancy in order to take into account the locality relationships among ancestor and child tables, in which a child table references its

parent and ancestor table(s) (via foreign keys). Accordingly, table *Album* has a foreign key (*UserID*) referencing its parent table *User*. In addition, table *Photo* has two foreign keys (*UserID*) and (*AlbumID*) referencing its ancestor table *User* and its parent table *Album*, respectively. At the physical level, rows from a child table and its parent and ancestor tables are joined using a simple ordered merge. Based on the *traditional relational* schema, fetching all albums and photos corresponding to a given *UserID* would take two sequential steps, as there is no way to retrieve photo records by *UserID*. On the other hand, in the *clustered hierarchical* schema, each user—along with its related albums and photos—are clustered in the same *aggregate*. Each *aggregate* is stored in a single node and is read in a single request. Although this denormalized redundant data may incur some update anomalies, they have tackled the latency effects of having remote data.

Some modern data stores, such as Google Spanner[45] (Corbett et al. 2013), provide ACID transactions over *multiple aggregates* (called *tablets*). Spanner allows strictly serializable transactions over multiple tablets. It isolates update transactions via *S2PL*. However, concurrency is enhanced through a lock-free execution of read-only transactions in which data versioning is used to prevent them facing inconsistent snapshots. More specifically, Spanner uses an accurate physical clock (TrueTime) whereby update transactions are globally ordered based on the order of their commits. Thus, a read-only transaction can respect strict serializability by asking for the latest snapshot whose upper bound is TrueTime's current clock. Spanner replicates the commit logs of a cross-datacenter update transaction through running *2PC* on top of the Multi-Paxos protocol (Chandra et al. 2007). Likewise, Scalaris (Schütt et al. 2008b) enforces the serializability of transactions over multiple data items (see Appendix A).

Since distributed systems cannot avoid network partitions in practice, they focus only on the trade-off between (strong) *consistency* with either *availability* (stated by CAP (Brewer 2000; Gilbert and Lynch 2002)) or *low latency* (stated by PACELC (Abadi 2012)). This results in adapting weaker models of consistency by NoSQL stores. These models require less coordination between replicas; this, in turn, decreases the latency of operations. In the following, some of the well-known weak consistency models are described.

3.2 Eventual Consistency

Eventual consistency does not dictate any ordering of operations but rather ensures the gradual and eventual convergence of replicas to identical values after receiving the same set of asynchronously propagated updates (Fekete et al. 1996; Vogels 2009). According to Vogels (2009), when there is no failure, factors such as the number of replicas in the replication scheme, system load and communication delays determine the length of inconsistency window. This notion of consistency suffices for those applications for whom the high availability of data requests is so critical that even a tiny impact on it causes user dissatisfaction and loss of revenue (e.g., advertisement records or social media). On the other hand, eventual consistency allows anomalous behaviors of applications; this, in turn, increases the complexity of application design, testing, and debugging. It also suffers from conflicting write operations on the same data item. This necessitates using *conflict detection and resolution* mechanisms in order to prevent the permanent divergence of replicas by applying an identical sequence of updates on them. Three common strategies are as follows:

- *Client-driven* semantic reconciliation, whereby an application-specified reconciliation is performed on divergent versions. For example, a shopping cart service merges the different versions of a shopping cart. However, the generalization of such reconciliation functions is hard as various applications require different customized reconciliation. In addition,

[45] It is one of the NewSQL stores.

the complexity of client applications and their performance overhead are increased. Some NoSQL stores, such as Voldemort[15], use this strategy.

- *Timestamp-based* syntactic reconciliation, such as *last-write-wins* (also called Thomas's write rule (Thomas 1979)), whereby a timestamp is assigned to each modified replica. Accordingly, the conflict of peer replicas is resolved by selecting the one with the best timestamp (i.e., the most recent and accurate). A potential drawback is that this strategy may cause some lost updates, especially when modifying large objects. Lost updates may endanger data correctness and make *last-write-wins* inappropriate for most OLTP applications. For an example of lost updates, suppose that an object has two attributes *a* and *b* and has two replicas *R1* and *R2*. If a client modifies *a* in *R1* and another client modifies *b* in *R2*, then by detecting the conflict, one of the replicas overwrites the other; hence, one of the modifications is lost. This issue can be solved by reducing the unit of conflict detection and resolution (Lakshman and Malik 2010). Continuing with the above example, by assigning timestamps to the individual modified attributes (instead of the whole modified objects), the conflict is resolved while keeping the modifications on both *a* and *b*. Some NoSQL stores—such as Apache Cassandra (Lakshman and Malik 2010), Amazon Dynamo (DeCandia et al. 2007), COPS (Lloyd et al. 2011), and CouchDB[34]—use this resolution.
- *Vector clocks* (or version vectors) syntactic reconciliation (Lamport 1978; Mattern 1989; Parker et al. 1983), whereby a partial ordering across the write operations of each data item is established by capturing their potential happens-before relationships. Therefore, the system is relieved from the burden of clock synchronization across all servers. More specifically, for a data item D with r replicas stored on different physical nodes, a vector V_i^D of logical clocks represents the last version of D that is known by a node N_i, where $i \in \{1...r\}$. $V_i^D = [<N_1, C_1^i>, \ldots, <N_r, C_r^i>]$ is a list of pairs where C_k^i is the clock value of node N_k when D is last updated on it. A node's clock value can be derived from its local clock or some ordinal values. As an implementation of the latter, C_k^i starts at zero, and is incremented by subsequent modifications of D at N_k. Data item D may have different vector clocks on different nodes. Through comparing vector clocks V_i^D and V_j^D ($i <> j$), it is determined whether there is a conflict or not. More precisely, if $V_i^D = V_j^D$, then no action is taken; if $V_i^D < V_j^D$ (or $\forall\ k \in \{1...r\}, C_k^i < C_k^j$), it means that N_i does have the stale version; thus, V_i^D is replaced with V_j^D, along with sending the value of D to node N_i (and vice versa). Otherwise, there is a conflict that may be handled at the application level. As a challenge, by growing the number of replicas, along with their addition and removal, the maintenance cost of vectors is increased. Some NoSQL stores, such as Amazon Dynamo (DeCandia et al. 2007) and Riak KV[12], use this strategy.
- *Commutative Replicated Data Types (CRDT)* syntactic reconciliation, in which the convergence of replicas of certain data structures is guaranteed irrespective of the execution order of operations (Letia et al. 2010). Therefore, concurrent updates commute and there is no need to perform conflict detection and resolution. For example, Sovran et al. (2011) design a commutative data structure called *counting set (cset)*, which is like a multiset, and a count is kept for each element. Unlike multisets, the count could be negative. Accordingly, a *cset* supports two commutative operations *add(x)* and rem(x) to add and remove an element *x*, respectively. The former increments the counter of x and the latter decrements it. They uses *csets* to store message timelines, photo albums, friend lists, and message walls. Some NoSQL stores, such as Walter (Sovran et al. 2011) and Riak KV[12], use this strategy.

Note that none of the above concurrency control strategies can satisfy OLTP applications, as they cannot handle the well-known critical section problem. For example, two concurrent transactions

can modify the same data item at the same time. The convergence of replicas can also be ensured using *repairing* strategies. Three common strategies are as follows:

- —*Read-repair* (DeCandia et al. 2007), in which the coordinator of a *read* operation (on a data item) initially sends it to all corresponding replicas. It waits until receiving responses from a configurable number of replicas. It then determines the latest version of returned values and returns it to the user. In addition, obsolete replicas are asynchronously updated. Some NoSQL stores—such as Amazon Dynamo (DeCandia et al. 2007), Apache Cassandra (Lakshman and Malik 2010), Voldemort [15], and Riak KV[12]—use this strategy.
- —*Hinted-handoff* (DeCandia et al. 2007), which ensures that all *write* requests are eventually applied to their intended target replicas. Assume that the coordinator of a *write* operation notices the temporary failure of a replica. It then stores all the updates (related to the replica) on a local hinted-handoff table. By recovering the replica, all updates are pushed back to it. However, in the case of a memory crash, the table may be lost. Some NoSQL stores, such as Amazon Dynamo (DeCandia et al. 2007) and Apache Cassandra (Lakshman and Malik 2010), use this strategy.
- —*Anti-entropy* (Demers et al. 1987) using Merkle-trees (Merkle 1989), in which each member of a replica group periodically creates a Merkle-tree and sends it to other members. The tree received by a replica node can be efficiently compared with its own tree to determine out-of-synch data portions along with sending them to obsolete replicas. However, this strategy requires network bandwidth owing to transferring of trees over the network. In addition, leaving or joining nodes may invalidate some Merkle-trees (DeCandia et al. 2007). Gonçalves et al. (2015) introduce a lightweight anti-entropy mechanism without the use of Merkle-trees. Some NoSQL stores, such as Amazon Dynamo (DeCandia et al. 2007) and Apache Cassandra (Lakshman and Malik 2010), use this strategy.

A majority of NoSQL stores—such as Apache Cassandra (Lakshman and Malik 2010), Amazon Dynamo (DeCandia et al. 2007), Riak KV[12], Voldemort (Feinberg 2011), CouchDB[34], Couchbase Server[24], MongoDB[23], Neo4j (Webber 2012), and Amazon DynamoDB[33]—preserve this consistency model.

3.3 Regular Causal Consistency

The infeasibility of global clocks in distributed systems, as emphasized by Lamport (1978), motivated systems to consider partially ordered operations. Accordingly, causal consistency enforces a *partial* order (called *happens-before* relation) among causally dependent operations. More precisely, an operation b is causally dependent on an operation a if one of the following three conditions holds: (1) a and b are issued by the same client and b happens after a (a.k.a. session order), (2) b is a *read* operation that returns the value *written* by a (a.k.a. read-from order), and (3) they are transitively dependent. Intuitively, causal consistency guarantees that when committing a *write* operation at a replica, all *write* operations causally preceding it have already been committed at the replica (Ahamad et al. 1995; Birman 1985; Lamport 1978; Raynal and Schiper 1995).

For example, consider a scalable instant messaging service that allows creating different message groups. Users can use the service by joining one or more groups and sending messages to the members of each group, as well as reading messages posted by others. Due to the availability and scalability requirements of the service, key data (such as the timeline of each group) are replicated on several application servers around the world. Suppose that users u_1, u_2, and u_3 are the members of a message group G whose timeline has three replicas and these users have been redirected to different application servers. Now, consider the following sequence of actions: (1) u_1 sends message m_1: "I lost my wallet at the university☹"; (2) both users u_2 and u_3 read m_1; (3) after a while, u_1 sends

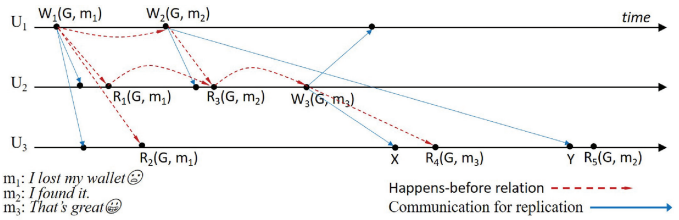

Fig. 8. An invalid scheduling of operations with regard to the causal consistency since by the (local) commit of W_3 at a G's replica (occurred at point X), W_2—which is causally preceding W_3—has not been (locally) committed at the replica yet. According to causality, this schedule can be legalized by committing W_3 after point Y.

m_2: "I found it"; (4) u_2 reads m_2; and (5) u_2 sends m_3: "That's great☺". As Figure 8 shows, if causality is not respected, u_3 could read m_3 before reading m_2. Therefore, she may think that u_2 is pleased about the loss of u_1's wallet! However, with regard to causality, it is impossible for u_3 to read m_3 before reading m_2 because the write of m_3 on a replica of G's timeline is causally dependent to the write of m_2 on the same replica.

Since this consistency does not guarantee any ordering between causally independent concurrent operations, the efficiency of implementation is increased, as there is no need to determine a serialization point between unrelated concurrent *write* operations so that they can be replicated in any order. For example, in the above messaging service, if users u_1 and u_2 send two messages concurrently, they can appear in any order on the replicas of G's timeline. However, there is a conflict when the same data item is updated by concurrent operations. This is undesirable, as the conflict may result in the continuous divergence of replicas (Ahamad et al. 1995). In addition, *read* operations may not always access the latest versions of read data items. Conflicting writes are usually handled using the above conflict-handling strategies.

Causal consistency is typically enforced by the following three steps: (1) committing the updates on corresponding local replicas, (2) asynchronous propagation of updates to remote replicas, and (3) performing *causal dependency checking* to determine when each update can be applied on a corresponding replica. It needs carrying some *causal dependency metadata* that is propagated through distinct messages or along with the updates (Belaramani et al. 2006; Lloyd et al. 2011, 2014; Petersen et al. 1997). However, there is a challenge of communication overhead due to transferring dependency metadata, which, in turn, affects the throughput. There are various representations of *dependency metadata*, such as *vector clocks* (Almeida et al. 2013; Birman et al. 1991; Du et al. 2014a; Ladin et al. 1992; Terry et al. 1994; Zawirski et al. 2013), *real-time clocks* (Akkoorath et al. 2016; Du et al. 2013, 2014b), or a hybrid of physical/logical clocks (Didona et al. 2017). Note that *real-time clocks* have the problem of clock skew among different servers.

Causal consistency makes a trade-off between performance and ease of programming as it allows data to be replicated asynchronously and avoids some of the anomalous behaviors of eventual consistency. Mahajan et al. (2011) prove that causal consistency is one of the strongest notions of consistency that are compatible with *low latency*, *high availability*, and *partition tolerance*. It also enforces all the following session guarantees in which a session denotes the sequence of operations issued by a given client on the store (Brzezinski et al. 2004).

— *Monotonic-Reads (MR)*, in which successive *read* operations (of a session) issued on a data item observe a nondecreasing order of the item's versions. For example, consider a social network application in which the posts of each user are written to both the user's own wall

and friends' walls. According to *MR*, when a user observes a post in the user's wall, then that user's succeeding read operations will certainly include the same post (unless the post was removed).

— *Monotonic-Writes (MWs)*, in which successive *write* operations (of a session) issued on a data item take effect in the same order. According to *MWs*, when a social network user performs two consecutive writes, any successive read operation that contains the second post also contains the first one.

— *Read-My-Writes (RMWs)*, in which *read* operations (of a session) issued on a data item always see the effect of the most recent *write* operation (of the session) issued on the same item. According to *RMW*, when a social network user reads one's wall, the user observes one's latest posts (unless having already removed them).

— *Writes-Follow-Reads (WFR)*, in which *write* operations (of a session) issued on a data item take effect on versions of the item that are equal to or newer than versions seen by preceding *read* operations (of the session) issued on the same item. According to *WFRs*, when a social network user replies to a post, any successive read operation that contains the reply also contains the original post.

Causal consistency is preserved in a lot of academic geo-replicated data stores (Akkoorath et al. 2016; Almeida et al. 2013; Bailis et al. 2013b; Didona et al. 2017; Du et al. 2014b; Lloyd et al. 2011, 2013; Zawirski et al. 2015), which use various strategies for representing causal relationship metadata and dependency tracking. These strategies make the implementation of causal consistency more expensive than eventual consistency, such that no existing commercial data store preserves it.

3.4 Per-Object Timeline Consistency

Per-object sequential, or prefix, consistency ensures a *total* ordering of all operations on each data item along with respecting their order as issued by each client (Cooper et al. 2008; Lamport 1979; Terry 2013; Terry et al. 1994). It was initially used for the design of Yahoo! Pnuts (Cooper et al. 2008), the storage system used for Yahoo! web applications. It avoids using transaction serializability (on multiple data items) because of the tendency of web applications to issue operations on single data items. Using this consistency, concurrent updates on the same data item are serialized at the corresponding primary replica. It can perform this ordering by assigning monotonically increasing numbers to the updates. Then, the updates (along with their associated sequence numbers) are asynchronously replicated on all corresponding replicas. These sequence numbers are required to ensure the order-preserving delivery of updates. In this model, all write operations and strongly consistent reads on a data item are forwarded to the corresponding master replica. However, timeline consistent read operations on a data item are answered locally. This may result in observing stale data. However, a client's *read* operation never returns a new version before an old one. Some NoSQL stores, such as Yahoo Pnuts (Cooper et al. 2008), preserve this consistency.

3.5 Parallel Snapshot Isolation (PSI)

This notion of consistency relaxes Snapshot Isolation (SI) (Berenson et al. 1995) over fully geo-replicated data stores. More precisely, PSI does not impose a *global* commit ordering of transactions (Sovran et al. 2011). Instead, it locally enforces SI on transactions executed in each datacenter where a local sequence number is used. It also preserves the *causal ordering* of transactions executed across datacenters. This relaxation allows the asynchronous propagation of updates across datacenters and improves the scalability and performance of the system. By starting a transaction T_i at a datacenter D, it takes a consistent snapshot that does not necessarily reflect the outcome

of all transactions committed before starting T_i. This strategy guarantees to reflect the outcome of those transactions that are *causally* preceded T_i as well as the ones that have locally committed at D, before T_i. The *causality* means that if T_i reads from T_j committed at D, then T_i's commit should be ordered after T_j's commit at any other datacenter. After the local commit of T_i at D, its updates are asynchronously propagated to and committed at remote datacenters. Some NoSQL stores, such as Walter (Sovran et al. 2011), enforce PSI over transactions.

Table 3 summarizes the characteristics of the discussed consistency models[46]. Note that modern applications interact with a mix of weakly and strongly consistent data. For instance, in the discussed messaging service example, the timeline of each message group is causally consistent data and the membership information is strongly consistent data as servers must agree about the receivers of each sent message. One strategy is to store data in a single data store that provides tunable consistency guarantees—such as Oracle NoSQL[9], MongoDB[23], Yahoo Pnuts (Cooper et al. 2008), Apache Cassandra (Lakshman and Malik 2010), and Amazon Dynamo (DeCandia et al. 2007)—whereby the application programmer can select an appropriate level of consistency for each operation. However, accessing a piece of data that needs the interaction between different consistency levels makes programming difficult and error prone. Another strategy is to store data on a mix of data stores with heterogeneous consistency guarantees. However, the application programmer needs not only to reason about native consistency guarantees preserved by each individual store but also about the behavior when accessing a piece of data across these stores. In order to tackle this situation, Milano and Myers (2016) introduce a mixed-consistency programming model whereby programmers can choose the level of consistency on a per-object basis. Recently, there have been some studies on relieving the application programmer of the manual and error-prone selection of the correct level of consistency, whereby the decision process is automatically performed by the application (Li et al. 2014; Liu et al. 2014; Terry et al. 2013).

As already mentioned, implementing distributed ACID transactions using heavyweight coordination protocols, such as 2PC (Gray and Reuter 1992; Skeen 1981) and Paxos commit (Gray and Lamport 2006), makes it difficult to build scalable and available distributed data stores. Therefore, many NoSQL stores avoid supporting distributed transactions that access data spanning multiple nodes. However, many web applications (such as online gaming and collaborative editing) require an atomicity guarantee at multikey granularity. Recently, there have been many attempts at supporting ACID transactions in NoSQL stores without sacrificing scalability.

Levandoski et al. (2011) provide an architecture called Deuteronomy for scaling databases and supporting ACID transactions (using a lock-based approach). Contrary to the classical data stores whose data storage and transaction execution logic are tightly coupled, Deuteronomy decomposes data store engine functions into separate data storage component *DC* and transaction management component *TC*. The former is responsible for the physical data organization (such as data storage and indexing) and caching, along with supporting an interface with atomic operations to access data. The latter is responsible for transaction concurrency control and recovery (e.g., logging/locking). Applications submit their requests to *TC* that, in turn, guarantees to pass the nonconflicting concurrent requests to the appropriate *DC*. Upon completion, *TC* commits the transaction and tells *DC* to persist data items. In this architecture, if *TC* fails, a new one is instantiated and performs recovery using the log. This architecture is scalable since, by growing data volume, more *DCs* can be added while the functionality of *TC* remains isolated from scaling decisions at the *DC* layer. On the other hand, it is flexible, as multiple *DCs* and *TCs* can be deployed in different ways from mobile devices to clusters or cloud environments. However, in order to prevent *TC*

[46]Note that the consistency model preserved by a distributed storage system has a tight relation with the choice of replication strategy.

Table 3. Comparison of Well-Known Consistency Models Preserved in NoSQL Stores

Characteristics		Suitable applications	Suitable NoSQL store data models
Strong consistency	– Dictating a total, real-time ordering of all operations/transactions – Providing a single, consistent image of the whole data – Severe impact on availability and performance (especially in wide-area networks) – Using a synchronous primary/ update-anywhere replication strategy	Scenarios requiring ACID reliability, such as financial services	It can be practically preserved over one or more *aggregates* (along with their corresponding replicas). An *aggregate* can be a key-value pair in *key-value stores*, a column-family, super column-family, the whole row of a table in *wide-column stores*, or a document in *document stores*.
Causal consistency	– Enforcing a partial ordering (called happens-before relation) among causally dependent operations – The efficiency of implementation is more than strong consistency and less than the eventual one. – The possibility of conflicting writes – Read operations may not always access the latest versions of read data items. – Using an asynchronous primary/ update-anywhere replication strategy	Online human-facing services, such as commenting services in social networks	It can be preserved in all data models while enforcing in a lot of academic NoSQL stores. However, owing to its communication overhead (of dependency tracking), no industrial NoSQL store enforces this model.
Per-object Timeline consistency	– Ensuring a total ordering of all operations on each data item along with respecting their order as issued by each client – Read operations may access stale data items. – A client's read operation never returns the new version of a data item before an old one. – Using an asynchronous primary replication strategy	Online human-facing services, such as the update of photos/albums	It can be practically preserved over an *aggregate* in a key-value, wide-column or document stores. It is enforced in some industrial NoSQL stores, such as Yahoo Pnuts (Cooper et al. 2008).
Parallel snapshot isolation	– Enforcing snapshot isolation on transactions executed in each datacenter – Preserving the causal ordering of transactions executed across datacenters – Using an asynchronous primary/ update-anywhere replication strategy	Similar to causal consistency	Preserved in some academic NoSQL stores, such as Walter (Sovran et al. 2011)
Eventual consistency	– It does not dictate any ordering of operations. – It ensures the gradual and eventual convergence of replicas to identical values after receiving the same set of asynchronously propagated updates. – It allows anomalous behaviors of applications. – It suffers from conflicting write operations on the same data item. – It uses an asynchronous primary/ update-anywhere replication strategy.	It suffices for many services in web applications for whom the high availability of data requests is so critical that even a tiny impact on it causes user dissatisfaction and loss of revenue.	It can be preserved in all data models. It is enforced in a lot of NoSQL stores.

from being a performance bottleneck for high loads, they propose using multiple *TCs* instantiated on separate machines, assuming that each transaction is serviced by a specific *TC*. To do so, users need a kind of partitioning technique to divide the database across multiple *TCs*. In addition, they need to synchronize the transactions on different *TCs* using a distributed transaction protocol such as 2PC.

Recently, Levandoski et al. (2015) improved the *TC* component by using modern MVCC techniques to reduce transaction conflicts along with batching techniques to reduce the cost of *TC/DC* traffic. They are also working on fault-tolerant scale-out transactional stores using multiple *TCs* and *DCs*, which can fail over with minimum downtime.

Warp (Escriva et al. 2014) is a transactional library that provides support for distributed ACID transactions using a commit protocol named linear transactions. This protocol uses a dependency tracking technique in which, instead of computing a total order on all transactions, which is costly, it orders those transactions that have some data items in common. Some NoSQL stores, such as Scalaris (Schütt et al. 2008b), use the Paxos commit protocol to support multikey ACID transactions whose performance is less than Warp transactions. The reason is that Paxos requires a server to perform a broadcast followed by waiting for a quorum of servers, which divides overall throughput by the number of servers involved. Warp is used as an add-on for the HyperDex NoSQL store (Escriva et al. 2012).

4 DATA PARTITIONING

Data partitioning includes solutions whereby data in a database are split into some disjoint partitions and spread over different storage nodes. This can be achieved either *horizontally* (also called sharding) or *vertically*. Horizontal partitioning, which is typically used by NoSQL stores, divides data at the row level (e.g., rows in a wide-column table or documents in a collection) into disjoint partitions (or shards). Vertical partitioning divides data based on predefined groups of columns that are accessed together (e.g., column-families in wide-column stores) into disjoint partitions.

Data partitioning has some advantages. First, it improves the scalability of a system as it tackles the situations when large volumes of data and high request rates overwhelm the storage and processing capacity of any single server. Second, it improves the performance of a system by increasing the degree of parallel processing on multiple partitions. Finally, it works well for geographically dispersed data, whose reads and writes are primarily within one geographic location[47].

There are several studies (Chen et al. 2013; Huang et al. 2015; Schall and Härder 2015; Turk et al. 2014) on enhancing the partitioning strategies used by NoSQL stores. However, there are still some challenges that affect the efficiency of a partitioner with regard to the throughput and latency of user requests. First, the number of multi-partition requests (i.e., the processing of a query requires contacting several partitions) should be decreased. This reduces the amount of data transferred over a network during query processing. Second, the distribution of processing and storage load should be uniform with regard to the capacity of nodes (e.g., processing power, disk speed, and network capabilities). Third, the amount of data transferred during repartitioning events (e.g., a node addition or removal) should be reduced. Finally, the storage, retrieval, and manipulation of mapping between partitions and storage nodes should be efficient (Stonebraker 1986; Stonebraker et al. 2013).

In NoSQL stores, data commonly accessed together are aggregated in the value of a key-value pair in the columns of a column family and in each document. This is how data are clumped up as data items by different NoSQL stores. Sharding can be either *key oriented* as data lookup is just based on some (shard) keys or *traversal oriented* as data lookup is based on analyzing the relationships of data items. Data models have a significant effect on selecting the strategy of partitioning. Graph stores use *traversal-oriented* sharding strategies, whereas other data models use *key-oriented* strategies. Key-oriented hashing partitioning strategies are applicable for applications, such as shopping cart and product catalog, where data are represented by key-value pairs or documents and each data item can be accessed independently. Traversal-oriented strategies, which partition

[47] In order to avoid any single point of failure, partitioning strategies are supplemented with replication mechanisms.

less connective nodes and group highly related nodes, are suitable for applications such as recommendation, in which objects are linked as graphs and links between friends, product purchases, and ratings are traversed rapidly. Both vertical and horizontal partitioning can be provided by the wide-column stores; for other kinds of NoSQL stores, only horizontal partitioning (sharding) is being studied for now. In this section, some of well-known sharding strategies are explained.

4.1 Key-Oriented Static Sharding

This kind of sharding aims at balancing the utilization of static space, while *dynamic* data and query workloads are entirely disregarded. Some of the corresponding strategies are as follows.

4.1.1 Range-Based. Data items are clustered according to the contiguous intervals of (shard) keys, such as the range of a unique identifier. An advantage is that range queries on short intervals are handled efficiently as they involve communicating with only a single node or a few nodes. In other words, the number of multi-partition requests is decreased. This kind of partitioning is very useful for applications such as data warehousing (Lee et al. 2000), Web servers (Luo and Naughton 2001), and online games (Knutsson et al. 2004), which involve distributed order-preserving data structures. However, there are some drawbacks. First, popular keys (e.g., 'E') cause query workload to be skewed so that a single or a few nodes receive a larger fraction of requests than others. Second, nonuniform distribution of inserting data items' keys causes data workload to be skewed so that storage nodes become unbalanced. Finally, it needs to maintain a central *lookup table* or *directory* to store the mapping of partitions to storage nodes. The *directory service* may decrease performance and scalability, as a round-trip delay is added to the critical data access path. This strategy is used by some NoSQL stores, such as Google Bigtable (Chang et al. 2008), Apache Hbase[19], MongoDB[23], Yahoo Pnuts (Cooper et al. 2008), and IBM Spinnaker (Rao et al. 2011).

4.1.2 Simple Hashing. Data items' keys are randomly hashed to their hosting nodes via simple hashing schemes, such as modulo hashing, where:

$$HostingNodeNumber = Key \ modulo \ NumberOfNodes$$

Relying on hash functions results in efficient data lookups, as they are performed locally (DeCandia et al. 2007). However, in simple hashing, adding or removing a node necessitates redistributing a lot of data items as the hash of keys severely reshuffles. This incurs a high cost for regularly expanding datasets. Furthermore, there is poor data locality and the number of multi-partition requests is increased as relevant data items are randomly spread over different nodes.

4.1.3 Consistent Hashing. Consistent hashing considers the scope of a hash function as a "ring" in which both nodes' IDs (e.g., *IP* addresses) and data items' keys are randomly hashed to its positions (Karger et al. 1997; Lewin 1998). The hosting node of a data item is the first node encountered when walking clockwise from the data item's position on the ring. Therefore, a node with position p is responsible for an individual set of data items whose keys are hashed to an arc (or partition) of the ring between $p.predecessor$[48] and p. When a new node joins and is hashed at position q, the old arc corresponding to its immediate *successor* is split into two new adjacent arcs between $q.predecessor$ and q, as well as q and $q.successor$. On the other hand, by leaving or failing a node, the old arcs corresponding to the node and its *successor* are merged with each other.

As a result, in contrast to simple hashing, the addition or removal of a node in consistent hashing incurs only a redistributing $O(1/N)$ fraction of data items that are hosted on the node's *successor*, where N is the number of existing nodes. This means that consistent hashing scales much better

[48] Assume that, for a given node hashed at position p, the positions of its immediate *predecessor* and *successor* are specified by $p.predecessor$ and $p.successor$, respectively.

than simple hashing. However, there are some drawbacks. First, it still suffers from poor data locality. Second, for a uniform distribution of data items over the key space, there is an $O(\log N)$ imbalance factor between N nodes in terms of stored data items and query workload (Stoica et al. 2001). Third, the performance heterogeneity of nodes is not taken into account, as it may result in data and query workload imbalance. Finally, by a node joining or leaving, data redistribution may overwhelm the load capacity of its *successor*.

Most of these issues can be addressed through using *virtual nodes* or shards (Dabek et al. 2001; Stoica et al. 2001) by which multiple noncontiguous positions in the hash ring can be assigned to a physical node. The number of positions or virtual nodes corresponds to the capacity of the node. More specifically, the stronger physical server will split into more virtual nodes. However, some systems (e.g., Apache Cassandra (Lakshman and Malik 2010)) put a limit on the number of virtual nodes, as they have several side-effects (Hogqvist et al. 2008), such as increased agitation owing to the failure of a physical node and increasing state maintenance. Huang et al. (2015) propose a method to compute the optimal positions of joining nodes (or virtual nodes) in the hash ring, with no impact on load balance. Chen et al. (2013) present a hybrid partitioning approach by combining *consistent hashing* and *range-based* strategies with regard to heterogeneity of nodes. In this method, a cluster with a set of N nodes is divided into k subsets or virtual nodes, in which *consistent hashing* and *range-based* strategies are used for intercluster and intracluster data partitioning, respectively. Some NoSQL stores—such as Apache Cassandra (Lakshman and Malik 2010), Amazon Dynamo (DeCandia et al. 2007), Voldemort (Feinberg 2011), Infinispan (Marchioni 2012), Scatter (Glendenning et al. 2011), COPS (Lloyd et al. 2011), Riak KV[12], and Microsoft Trinity (Shao et al. 2013)—use this strategy.

Variants of *consistent hashing* are commonly used in DHTs (Ratnasamy et al. 2001; Rowstron and Druschel 2001; Stoica et al. 2001), in which data are stored and looked up in a totally decentralized manner. DHT nodes maintain a structured P2P overlay network in which they are organized in predefined topologies in order to support a fast, scalable routing. A *key lookup* issued in one of the nodes is reliably routed through the overlay network to a node storing a data item related to the *key*. A variety of *topologies* and *routing protocols* are used by different DHTs, such as Chord (Stoica et al. 2001), CAN (Ratnasamy et al. 2001), Pastry (Rowstron and Druschel 2001), Tapestry (Zhao et al. 2001), Plaxton (Plaxton et al. 1999), Apache Cassandra (Lakshman and Malik 2010) and Amazon Dynamo (DeCandia et al. 2007). Most DHTs take more than one hop to locate data, along with maintaining a small routing state per node. More precisely, for a network with N nodes, each node possesses a limited view of the network and usually maintains the information of $O(\log N)$ neighbor nodes (consisting of its 1-hop distant nodes and closest *successors*). In addition, the *lookup* operation mostly takes $O(\log N)$ hops. Table 4 shows a specification of Chord DHT. For more information on existing DHT-based routing protocols, see Abid et al. (2015) and Lua et al. (2005).

Some DHTs, such as Apache Cassandra (Lakshman and Malik 2010) and Amazon Dynamo (DeCandia et al. 2007), are one-hop (a.k.a. zero-hop) DHTs: through providing enough state at each node, its requests can be directly routed to destination nodes. Note that maintaining the full routing state per node simplifies routing. However, with regard to the high churn (arrival and planned/unplanned departure) of nodes, maintaining the consistency of all nodes' routing states may impact the scalability of the system. As *consistent hashing* spreads adjacent keys over participant nodes, range queries are not efficiently supported. One of the approaches to tackle this drawback is the use of *key-order preserving* mapping functions, in which adjacent ranges of keys are mapped to contiguous ranges of nodes (Schütt et al. 2006, 2007, 2008a). This strategy is used in some NoSQL stores, such as Scalaris (Schütt et al. 2008b) and Apache Cassandra (Lakshman and Malik 2010).

Table 4. The Specification of Chord DHT

Chord
Overlay network: Key/node *m-bit* identifiers are represented as a ring of hashed positions in the range $[0, 2^m)$, in which each node with position n knows its immediate *successor* (or $n.successor$) and *predecessor* (or $n.predecessor$).
Assigning keys to nodes: Consistent hashing.
Per-node routing state: A node with position n maintains a lookup table T_n consisting of m entries (fingers), where i^{th} entry $T_n^i = successor((n + 2^{i-1}) \bmod 2^m)$ for $1 \leq i \leq m$.
Routing protocol: Assume that $Lookup(k)$ is initiated at a node with position n. If $n < k \leq n.successor$, the *successor* of n is returned; otherwise, lookup recursively starts at a *finger* of T_n, which immediately precedes k.
Routing state maintenance: (1) By *joining* a node with position n, an arbitrary node is asked about $x = successor(n)$. Then, n's *successor* and *predecessor* are set to x and x's *predecessor*, respectively. (2) By *planned leaving* a node, its *successor* and *predecessor* nodes are notified about the departure and adjust their pointers and successive lists.

4.1.4 Hyperspace Hashing. The above partitioning strategies partition data based on the single dimension of data items' keys. These strategies result in inefficient queries on properties other than primary keys, as all partitions must be searched. This problem can be tackled by partitioning on multiple dimensions. As such, hyperspace hashing is an extension to *consistent hashing*, in which several attributes (instead of just a key attribute) are taken into account for the mapping of data items to nodes (Almeida et al. 2013). More specifically, a data item with a set of attributes is mapped into a multidimensional space (called *hyperspace*) in which each dimension is defined by an attribute of the data item. Accordingly, the data item's position (in the *hyperspace*) is determined through the hashing of each attribute value along its corresponding dimension.

A *coordinator* separates *hyperspace* into disjoint *zones*; each *zone* is assigned to a virtual node. Therefore, a set of (*zone*, *node*) pairs, called a *hyperspace map*, is maintained for each *hyperspace* and is distributed to clients and servers that use it for the insertion, removal, and search of data items. This map may be changed by joining or leaving nodes. Through this geometric reasoning, adding more search terms to a query potentially restricts the search space to a smaller set of nodes, increasing query efficiency.

However, *hyperspace*'s volume grows exponentially with the number of attributes (or dimensions); thus, its coverage may not be feasible even for large datacenters. An efficient search on a *hyperspace* with n dimensions requires at least 2^n *zones* (and corresponding nodes). This nullifies the advantage of hyperspace hashing since a *partial* search operation incurs contacting a lot of nodes containing irrelevant data for the search. This exponential growth is avoided through partitioning of *hyperspace* into smaller independent *subspaces* with lower dimensionality. Then, each search is issued on a *subspace* that necessitates contacting the smallest number of nodes. The system administrator can configure the hyperspace hashing through defining different sets of *subspaces* along with selecting their used attributes. As this configuration may affect system performance (by changing both the efficiency of search and cost of maintaining consistent *subspaces*), Diegues et al. (2014) present an approach for automatic selection of the optimal configuration for a given workload. Another issue is that a *key* lookup in hyperspace hashing is inefficient, as it is a partial search and likely incurs contacting multiple servers. This is solved through the above *subspace* partitioning by constructing an individual *key subspace* containing only the key of the data item, as it can be assigned to one node.

Like hyperspace hashing, a Content-Addressable Network (CAN) (Ratnasamy et al. 2001) maps nodes onto N-dimensional coordinate space and the entire hash table is divided into chunks (called zones) stored on CAN nodes. A key is mapped to a point in the coordinate space; then, the corresponding entry is stored on the node to whom the chunk containing the point belongs. The purpose of a CAN is to provide efficient overlay routing. Each node keeps the coordinates of its immediate neighbors; thus, it can directly route through its neighbors to destination coordinates. To take advantage of single-key DHTs for multi-attribute queries, techniques such as Space Filling Curve (SFC) (Sagan 1994) and pyramid technique (Berchtold et al. 1998) can be used to reduce the dimensions of the multi-attribute point in a high-dimensional space to a single value in the one-dimensional index (e.g., Schmidt and Parashar (2008) and Sen et al. (2015)).

4.2 Key-Oriented Workload-Aware Sharding

In the above sharding strategies, *dynamic* query and data workloads are not taken into account. For example, the static hash function used in *consistent hashing* cannot tackle hotspots (i.e., nodes that are heavily loaded) caused by the nonuniform query distributions of real-world workloads. A *workload-aware partitioner* identifies hotspots and then dynamically adjusts the load distribution (Kwon et al. 2010; Stonebraker et al. 2013). This dynamic adjustment must guarantee that there are no *ping-pong phenomena* (Fleisch and Popek 1989) or continuous relocation of a data item across nodes. In the following, some of the strategies and modules for elastic partitioning are explained.

4.2.1 Migration of Virtual Nodes. This strategy assumes that each physical node can accommodate several *virtual nodes* according to both its capacity and query distribution (Chen and Tsai 2008; Godfrey et al. 2004; Rao et al. 2003; Stoica et al. 2001). It alleviates detected hotspots through reassigning some of their *virtual nodes* to light nodes. A set of centralized directory nodes is maintained by the system. They periodically receive the load information of different nodes and, based on that, assign virtual nodes to physical servers. Rao et al. (2003) propose three partitioning schemes whereby virtual nodes are exchanged through periodic contacts between physical nodes. With the first strategy, called *one-to-one*, a contact between a hotspot and a light node initiates a virtual node transfer. With the second strategy, called *one-to-many*, a light node randomly selects a directory node and reports its load to it. Likewise, a hotspot randomly selects a directory node and finds the most appropriate light node where a virtual node is transferred. With the third strategy, called *many-to-many*, every node (hotspot or light loaded) reports its load to a subset of directories. Then, each directory determines and initiates the most efficient virtual node migration based on its local information. This strategy does a greedy allocation, and load is not distributed evenly.

4.2.2 Item Balancing. Karger and Ruhl (2006) propose a strategy in which each node N_i periodically contacts other nodes such as N_j at random. Assume that L_i and L_j denote the load of nodes N_i and N_j, respectively; and $L_j \leq \alpha L_i$, where $0 < \alpha < \frac{1}{4}$. There are two cases: (1) N_j is the predecessor of N_i, then the address of N_j is increased such that $\frac{L_i - L_j}{2}$ data items are transferred from N_i to N_j; (2) N_j is not the predecessor of N_i; thus, if $L_{j+1} \leq L_i$, then N_j's data items are taken up by its successor (N_{j+1}). Also, N_j migrates as the predecessor of N_i by accepting half of N_i's data items; otherwise, set $i = j + 1$, and goto case 1.

DBalancer (Konstantinou et al. 2013) is a generic configurable load-balancing module that can be installed on top of a typical NoSQL store. It is fed by an expandable suite of various *range-based partitioners* that support data item balancing methods proposed by Karger and Ruhl (2006) and Konstantinou et al. (2011). It includes *Neighbor Item eXchange (NIX)*, *Node MIGration (MIG)*, a hybrid method (*NIXMIG*), and *Item Balancing (IB)*. Assume that storage nodes (with unique identifiers) are organized as a ring in which neighboring nodes host adjacent key ranges with the

increased order of their values in the clockwise direction and each node keeps pointers to its successor and predecessor nodes in clockwise and counterclockwise directions, respectively. Then, through *MIG*, a less loaded node N_i with load L_i helps a remote overloaded node N_j with load L_j by migrating to a position between N_{j-1} and N_j to capture half of the N_j's data items. This migration incurs the transfer of L_i to N_{i+1} and then transferring a portion of L_j to N_i. *MIG* causes a fast convergence to data load balance. However, it is not cost-effective, as it may incur transferring a lot of data items; as with *NIX*, an overloaded node transfers its data items with largest keys to its successor or its data items with smallest keys to its predecessor. Still, it acts poorly for overloaded neighborhoods where several adjacent nodes suffer from similar load stress. *NIXMIG* is a fast and cost-effective hybrid of the above methods that alleviates the load in overloaded neighborhood situations by adding a *MIG* phase (Konstantinou et al. 2011).

4.2.3 Self-Tuning Data Placement – Autoplacer. This strategy aims at maximizing the locality of dynamically changing data access patterns in a NoSQL key-value store by placing data closer to clients (Paiva and Rodrigues 2015; Paiva et al. 2015). Each node tracks and identifies its hotspot data items (that have the largest number of remote read/write requests) using a stream analysis algorithm. Then, a scalable Integer Linear Programming (ILP) solver redistributes the replicas of detected hotspots into appropriate nodes with regard to their capacity. Fast data lookup is preserved using a hybrid approach of a highly efficient *directory service* replicated on each node and *consistent hashing*. The former is used to encode the location of repartitioned hotspot items and the latter is used to define the placement of remaining noncritical ones. This strategy was integrated into the Infinispan (Marchioni 2012) key-value store; the obtained throughput is reportedly six times better than the original one (Paiva et al. 2015).

4.3 Traversal-Oriented Partitioning

By taking into account the high connectivity of data items in a graph store, the partitioning of data may create some *interpartition dependencies* that, in turn, increase network latency during cross-partition graph traversals. As a result, graph traversal execution time is increased. A sophisticated graph partitioner aims at achieving the following major goals:

- —Minimizing the number of cross partition links (i.e., edge cuts) as well as the required state synchronization of data replicas.
- —Maintaining a balanced distribution of data and query workload across partitions with regard to the capacity constraints of servers.
- —Preserving the quality of partitions assuming the modifications of graph topology in dynamic graphs and changes of elastic system capacity and query access patterns. Vertex and edge weights determine the amount of computational requirements (e.g., frequency of queries) for each vertex and the traversal rate of edges, respectively. These patterns cause vertices and edges to have different weights. For example, low-degree vertices in social networks have less query frequencies than high-degree ones. Workload-aware partitioners improve the quality of partitions by preventing heavyweight edge cuts and by considering an equal aggregate weight of vertices.
- —Reducing the memory and time requirements by making no assumption about the entire view of the graph. In other words, it is desirable to have a local partitioning in each server as there is no need to synchronize a shared global state between servers.

Satisfying the above conflicting goals is a challenge to graph stores and processing tools, as graph partitioning is an NP-hard problem (Garey et al. 1974). This resulted in lots of heuristic approaches (Kim and Candan 2012; Rahimian et al. 2015; Schloegel et al. 2000; Stanton and Kliot

2012; Tsourakakis et al. 2014). Standalone graph engines—such as Neo4j (Webber 2012), Graphchi-DB (Kyrola and Guestrin 2014), HypergraphDB (Iordanov 2010), DEX[39], MAGS (Das et al. 2016), and Graphchi (Kyrola et al. 2012)—avoid the partitioning of graph data. Owing to lack of space, common graph partitioning schemes are presented in Appendix C.

5 *CAP* PRINCIPLE

Assume a distributed storage system that stores a data item D replicated on three nodes N_1, N_2, and N_3, and a communication fault that splits the network (of the nodes) into two subnetworks: $\{N_1, N_2\}$ and $\{N_3\}$. If a modification request on D is submitted to N_3, then there are two possible scenarios. First, the request is successfully completed, knowing that by healing the partition the modified value of D is propagated to its replicas in N_1 and N_2. This scenario chooses the *availability* of the request. However, it may result in inconsistent values of D. Second, the request is aborted, knowing that contacting with N_1 and N_2 is not possible until healing the partition. This scenario chooses the strong consistency of data item D. However, it results in the unavailability of the request (Hale 2010). This simple example raises the question of whether both the *strong consistency* of data and *availability* of requests can be achieved simultaneously.

This trade-off was first observed by Rothnie and Goodman (1977). However, the increasing commercial popularity of the Web along with the growing demand for the geographic replication of data and high availability of operations motivated Fox and Brewer (Brewer 2000; Fox and Brewer 1999) to reclaim this trade-off as the *CAP principle*. This principle indicates that at most two of the three desirable properties (*Consistency*, *Availability*, and *Partition tolerance*) can be achieved simultaneously by a distributed data store. Later, Gilbert and Lynch (2002, 2012) formalized and proved *CAP*, which became known as the *CAP theorem*. In this context, the *CAP* properties are defined as follows:

- *Consistency* is viewed as a qualitative property denoting *linearizability*. Based on this definition, *CAP consistency* is not a dynamically observed metric determined by the operational status of the system. Instead, it is statically determined with regard to the system's employed algorithms whether they guarantee linearizability or not.
- *Availability* is viewed as a qualitative property denoting that every request sent by a client eventually (within a finite time) receives a successful (nonerror) response. Based on this definition, the "availability" or "unavailability" of a system is statically determined with regard to its used algorithms. However, this definition has some ambiguities. First, some systems that are highly available owing to their high uptime may not be considered as *CAP available*. For example, a distributed data store that uses a *quorum-based* (see Appendix B) or *Paxos-driven* (see Appendix B) synchronous replication is not *CAP available* because, during a network partition, read/write operations on the minority side of the partition may not be able to successfully complete. Second, there is no bound on the response time. For instance, an operation that is successfully completed after one week is considered *CAP-available*. However, such an operation is unavailable according to the instinctive notion of availability. In other words, "latency" (response time), which is practically an important feature, is not taken into account in this definition.
- *Partition tolerance* is viewed as a qualitative property (of a system's employed algorithms) denoting that even in the presence of a network partition, the system continues providing its *CAP-availability* or *CAP-consistency* guarantee. This definition is also fuzzy and unclear, as network partitions are not the only failures in a distributed system. In other words, there are also other failures, such as lost messages and node failures.

Accordingly, by forfeiting any one of the discussed *CAP* properties, distributed data stores are categorized as follows:

- *Consistency + Availability – CA* systems. The algorithms used by *CA* systems do not have any assumption about network partitions. Consequently, achieving this combination is practically impossible in distributed systems, as the occurrence of network partitions is inevitable (Hale 2010). Therefore, the fundamental *CAP* trade-off is between *consistency* and *availability*. This trade-off became a justification for supporting weak consistencies and justifying the design decisions of distributed data stores, especially NoSQL stores in which *consistency* is sacrificed more than *availability* (Vogels 2009; Wada et al. 2011).
- *Consistency + Partition tolerance – CP* systems. This combination is achieved by distributed data stores that preserve *CAP-consistency*. However, in the case of a network partition, a read/write request may not be responded to owing to avoiding the risk of inconsistency. Therefore, *CP* makes sense for systems designed to operate in a reliable network, such as a single datacenter, owing to the infrequency of network partitions (Gilbert and Lynch 2012). This combination is achieved in some systems, such as Scalaris (see Table 5), Google's Chubby lock service (Burrows 2006), and Spanner (Corbett et al. 2013), where the strong notion of consistency along with *Paxos-driven* synchronous replication are provided.
- *Availability + Partition tolerance – AP* systems. This combination is achieved by distributed data stores that enforce a weak notion of consistency. However, the execution of conflicting writes is allowed, which may result in the divergence of replicas and necessitates implementing a conflict-resolution mechanism. These systems are typically used by applications whose users are geographically scattered in a wide-area network and require a high level of availability along with a fast response time (instead of strong consistency), such as web caching. Many NoSQL stores—such as Amazon Dynamo (DeCandia et al. 2007), CouchDB[34], and Apache Cassandra (Lakshman and Malik 2010)—achieve this combination.

Brewer (2012) criticizes the misleading interpreted limitations of the *CAP* principle, in which a system is designed by taking into account a black-white trade-off between *consistency* and *availability*. He stated that a system should be designed in such a way as to provide *continuous CAP* properties. The weights of these properties can be tuned against each other in order to optimize them for different application requirements. It requires using a strategy for the detection of network partitions and management of a service's invariants and operations. For example, in an airline reservation application (Yu and Vahdat 2002), as long as there are many available seats in the airplane, the application can rely on inconsistent data. However, as the plane gradually gets filled, the level of data consistency should be increased in order to prevent flight overbooking. An application may impose various levels of CAP requirements in many dimensions, as outlined in Gilbert and Lynch (2012):

- Different types of maintained data. For example, *product information* managed by an online shopping cart application may be inconsistent. However, *checkout, billing, and shipping data* are strongly consistent.
- Different types of operations. For example, Yahoo Pnuts (Cooper et al. 2008) provides *strongly consistent read/write* operations that access data in a primary replica as well as *timeline consistent read* operations that access data in local replicas (see Appendix A).
- Correlation between data whereby data that are more likely to be accessed together are placed in a common partition. It increases the availability of operations among correlated data, as they are less vulnerable to unavailability in network partitions. For example, social network users belonging to the same group of friends may be placed in the same partition.

Table 5. Categorization of the Representative NoSQL Stores Based on the CAP Theorem

CAP principle	NoSQL store	The sacrifice of CAP consistency	The sacrifice of CAP availability
CP	**Scalaris**	No sacrifice	It is sacrificed during the DHT-based Paxos commit, when the majority of *participants* or *transaction managers* are not available.
AP	**Amazon Dynamo**	It supports a tunable eventual consistency.	No sacrifice
	Apache Cassandra	It supports a tunable eventual consistency.	No sacrifice
	COPS	It supports only causal$^+$ consistency across clusters.	No sacrifice
	CouchDB	There is no support for multi-document transactions	No sacrifice
P	**Yahoo PNUTS**	Lack of bundled write operations that span multiple data items.	Unavailability of the primary copy during the primary election.
	Google Bigtable	Lack of cross-row, multikey transactions.	Bigtable relies on the replication strategy of GFS, which is restricted within a datacenter. In addition, since Chubby is a *CP* system, in the case of network partitioning, clients in the minority side of the partition may not have access to data.
	IBM Spinnaker	Lack of cross-row, multikey transactions.	Due to the use of Paxos-based replication, in the case of network partitioning, writes and *strongly consistent* reads are unavailable for clients in the minority side of the partition.
	HyperDex	Lack of cross-row, multikey transactions.	When the number of replica node failures (in the block of replicas corresponding to each *subspace*) is more than a threshold.
	Scatter	Lack of cross-row, multikey transactions.	Owing to the use of Paxos-based replication. However, as long as a majority of nodes of a *group* remains alive, the used 2PC protocol is not blocking.
	Walter	Causal consistency is enforced on transactions across datacenters.	Failure of a datacenter D causes unavailability of writes on those objects whose *preferred datacenter* is D.
	MongoDB	Lack of cross-document, multikey transactions.	In the case of network partitions, nodes belonging to the minority side of a partitioned *replica set* are not available for *write* operations.
	Couchbase	There is no support for multi-document transactions	Similar to MongoDB.
	Microsoft Trinity	Lack of multikey transactions.	It is vulnerable to loss of data in the case of the failure of the datacenter where the Trinity cluster is deployed.
	Neo4j	Eventual consistency is enforced on operations across datacenters.	In the case of network partitions, nodes belonging to the minority side of the network are not available for *write* operations.

Accordingly, a mix of both weaker *CAP consistency* and weaker *CAP availability* is provided by many NoSQL stores in varying ways. These systems can be classified as *P* category. Table 5 categorizes the representative NoSQL stores based on *CAP*.

Despite the influential role of *CAP* formulation in justifying the design decisions of distributed data stores, there are some reasons that motivated researchers to propose alternative frameworks or formulations.

—The lack of correct and formal definitions of *CAP* features led to many ambiguities and different interpretations around it. According to Abadi (2012), *CAP* focuses just on the trade-off between *consistency* and *availability*. However, when there is no network partition (as

Table 6. Open Research Challenges of NoSQL Stores

Research aspect	Open research challenges
Elasticity control and measurement	– Selecting a proper set of elasticity metrics along with their suitable threshold values is a difficult task which may lead toinaccurate elasticity measurements. – What benchmarks and workloads should be used to assess the elasticity? For example, over provisioning scenarios and those that cover a fluctuating number of users should be taken into account. – Elasticity measurement approaches do not provide common metrics to compare their results with other works. – The question of which NoSQL database has the best elasticity is still unanswered.
Transaction processing	– Providing scalable, fault-tolerant transaction management models for distributed ACID transactions (especially multi key ones), which have high throughput and low response time. – Designing dynamically configurable, mixed-mode consistency models when different applications using the same database require different levels of consistency.
Data partitioning	– Designing efficient data partitioning methods to distribute data based on appropriate criteria, such as transaction types, data-access patterns, replication utilization, and the balancing of workload. – In most graph stores, data partitioning either is not included or simply implemented through random partitioning. – Few previous studies aim at making partitioning as transparent to the users as possible by minimizing its effect on normal transaction processing. – Most of NoSQL stores treat storage nodes and data items homogeneously, and data items are randomly and evenly distributed over storage nodes based on their size. They neglect the heterogeneity of storage node capacities and data access patterns. According to data access patterns, data items frequently accessed together should be aggregated in one node and hot data partitions need to be dispersed across all nodes. – The state-of-the-art dynamic graph partitioners suffer from a poor scalability for large-scale graphs. In addition, they are agnostic to query workloads, as they simply assume a uniform weight of edges and vertices throughout query processing.
Query processing	– NoSQL stores rely on heterogeneous query languages and there is still no user-friendly unified query language for them. In addition, there is little support for ad-hoc query and analysis (e.g., Apache Hive[49] and Apache Pig[50]), where sophisticated programming skills are required by users and developers to write these queries. Many NoSQL stores usually use MapReduce framework (Dean and Ghemawat 2008) to run querying applications. However, the significant overhead of launching processes in the MapReduce engine and the difficulty of expressing MapReduce functions as easily as SQL queries make this framework inappropriate for online query workloads (Gudivada et al. 2015).
Security	– They have security issues, as these systems provide a weak authentication with no support for soft-grained authorization, automatic auditing, and encryption techniques. By taking into account the geographical distribution of data, along with their unstructured nature, enforcing security in NoSQL stores is difficult.

its probability is really low (Stonebraker 2010a)), *availability* can be perfectly provided. In this situation, *CAP* ignores an arguably more influential trade-off between *consistency* and *latency* (which specifies a response time less than the topmost delay across replicas (Lloyd et al. 2011)). Abadi (2012) proposes to unify this trade-off with the fundamental *CAP* trade-off in a formulation called *PACELC*, in which *PAC* represents the inherent trade-off of *CAP* and *ELC* reasons about a trade-off between *consistency* and *latency*. Kleppmann (2015)

[49] https://hive.apache.org.
[50] https://pig.apache.org.

proposes a *delay-sensitivity framework* in which *availability* is modeled based on the latency of operations. For instance, it is measured as the proportion of operations that satisfy the distinctions of latency defined by a Service Level Agreement (SLA). A probabilistic variation of CAP theorem is formulated and proved by Rahman et al. (2017).

—The existing resurgence of transactional data stores, such as NewSQL stores (Stonebraker 2012). Bailis et al. (2013a) explore the connection between *consistency*, *availability*, and *transaction semantics* by analyzing the weak levels of transactional isolation in order to provide Highly Available Transactions (HAT). Ahsan and Gupta (2016) formulate the tradeoffs between the *Abort rate*, *Contention*, and *Throughput* of transactions in a new framework called *CAT*.

6 CONCLUSION AND OPEN RESEARCH CHALLENGES

We are currently working with a complementary technology in database management called NoSQL. These systems are used not as a revolutionary replacement for the relational database systems but as a remedy for certain types of distributed applications involved with a massive amount of data that need to be highly scalable and available. This survey presents a comprehensive study of various design decisions of NoSQL stores with regard to their data model, consistency model, data partitioning, and the CAP theorem. In addition, the strengths and drawbacks of each choice are explained in depth, along with exemplifying them in a set of cutting-edge NoSQL stores in Appendix A. Furthermore, we have discussed some existing challenges that have been emerged through relying on NoSQL stores and have been confronted by their developers and designers, which need to be addressed through alternative design decisions. Table 6 summarizes some of these open research challenges. The article should be beneficial to the research and industrial communities to achieve effective NoSQL stores, as well as NoSQL store users to select a suitable store to fulfill their specific application requirements. Tackling this situation necessitates an in-depth understanding of existing NoSQL technologies.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their critical reading of the article and their valuable feedback. Their many valuable comments and suggestions have substantially helped to improve the quality and accuracy of this survey.

REFERENCES

Daniel J. Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 2, 37–42.

Shahbaz Akhtar Abid, Mazliza Othman, and Nadir Shah. 2015. A survey on DHT-based routing for large-scale mobile ad hoc networks. *ACM Computing Surveys* 47, 2 (2015), 1–46.

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distributed Computing* 9, 1, 37–49.

Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers* 65, 3, 902–915.

Shegufta Bakht Ahsan and Indranil Gupta. 2016. The CAT theorem and performance of transactional distributed systems. ACM Press, 1–6.

Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In *36th International Conference on Distributed Computing Systems (ICDCS'16)*. IEEE, 405–414.

Sérgio Almeida, João Leitão, and Luis Rodrigues. 2013. ChainReaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 85–98.

Renzo Angles. 2012. A comparison of current graph database models. In *28th International Conference on Data Engineering Workshops (ICDEW'12)*. IEEE, 171–177.

Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. 2016. Foundations of modern graph query languages. *arXiv preprint arXiv:1610.06264*.

Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys* 40, 1, 1.

Hagit Attiya and Jennifer L. Welch. 1994. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems* 12, 2, 91–122.

Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, and others. 2012. Data infrastructure at LinkedIn. In *28th International Conference on Data Engineering (ICDE'12)*. IEEE, 1370–1381.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013a. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment* 7, 3, 181–192.

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013b. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 761–772.

Catriel Beeri. 1990. A formal approach to object-oriented databases. *Data & Knowledge Engineering* 5, 4 (1990), 353–382.

Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI replication. In *NSDI*, Vol. 6.

Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. 1998. The pyramid-technique: Towards breaking the curse of dimensionality. In *ACM SIGMOD Record*, Vol. 27. 142–153.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, Vol. 24. ACM, 1–10.

Claude Berge. 1973. Graphs and hypergraphs. North-Holland Publishing Company, Amsterdam.

Tim Berners-Lee, James Hendler, Ora Lassila, and others. 2001. The semantic web. *Scientific American* 284, 5, 28–37.

Kenneth Birman, Andre Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 9, 3 (1991), 272–314.

Kenneth P. Birman. 1985. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM SOSP*. ACM.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7, 422–426.

Harold Boley. 1992. Declarative operations on nets. *Computers & Mathematics with Applications* 23, 6–9, 601–637.

Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, and others. 2011. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 1071–1080.

Alain Bretto. 2013. *Hypergraph Theory*. Springer International Publishing.

Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2, 23–29.

Eric A. Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7.

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, and others. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*. 49–60.

Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. 2004. From session causality to causal consistency. In *PDP*. 152–158.

Mike Burrows. 2006. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 335–350.

Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record* 39, 4, 12–27.

Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 249–264.

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 398–407.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2, 4.

Chyouhwa Chen and Kun-Cheng Tsai. 2008. The server reassignment problem for load balancing in structured P2P systems. *IEEE Transactions on Parallel and Distributed Systems* 19, 2, 234–246.

Zhikun Chen, Shuqiang Yang, Shuang Tan, Ge Zhang, and Huiyu Yang. 2013. Hybrid range consistent hash partitioning strategy—A new data partition strategy for NoSQL database. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'13)*. IEEE, 1161–1169.

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 85–98.

DBTG Codasyl. 1971. CODASYL data base task group report. In *Conference on Data Systems Languages, ACM, New York*.

Edgar F. Codd. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13, 6, 377–387.
Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys* 11, 121–137.
Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2, 1277–1288.
James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31, 3, 8.
Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. 2017. Optimizing tree patterns for querying graph- and tree-structured data. *ACM SIGMOD Record* 46, 1, 15–22.
Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. 2001. Wide-area cooperative storage with CFS. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 202–215.
Mahashweta Das, Alkis Simitsis, and Kevin Wilkinson. 2016. A hybrid solution for mixed workloads on dynamic graphs. In *Proceedings of the 4th International Workshop on Graph Data Management Experiences and Systems*. ACM.
Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1, 107–113.
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 205–220.
Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1–12.
Diego Didona, Kristina Spirovska, and Willy Zwaenepoel. 2017. Okapi: Causally consistent geo-replication made faster, cheaper and more available. *arXiv preprint arXiv:1702.04263*.
Nuno Diegues, Muhammet Orazov, João Paiva, Luís Rodrigues, and Paolo Romano. 2014. Optimizing hyperspace hashing via analytical modelling and adaptation. *ACM SIGAPP Applied Computing Review* 14, 2, 23–35.
Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, 11.
Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014a. Closing the performance gap between causal consistency and eventual consistency. *Proceedings of the 1st Workshop on Principles and Practice of Eventual Consistency (PaPEC'14)*.
Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014b. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.
Barbara A. Eckman and Paul G. Brown. 2006. Graph data management for molecular and cell biology. *IBM Journal of Research and Development* 50, 6, 545–560.
Robert Escriva, Bernard Wong, and Emin Gun Sirer. 2012. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review* 42, 4, 25–36.
Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2014. *Warp: Lightweight multi-key transactions for key-value stores*. Technical Report, Cornell University.
Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Communications of the ACM* 19, 11, 624–633.
Alex Feinberg. 2011. Project Voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering (ICDE'11)*. IEEE.
Alan Fekete, David Guptab, Victor Luchangcob, and Nancy Lynchb. 1996. Eventually-serializable data services. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*. ACM, 300–309.
Brett Fleisch and Gerald Popek. 1989. Mirage: A coherent distributed shared memory design. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, 211–223.
Armando Fox and Eric A. Brewer. 1999. Harvest, yield, and scalable tolerant systems. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*. IEEE, 174–178.
Brian Gallagher. 2006. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS* 6, 45–53.
Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics* 42, 2–3, 177–201.
Michael R. Garey, David S. Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*. ACM, 47–63.
David Kenneth Gifford. 1981. *Information Storage in a Decentralized Computer System*. Ph.D. Dissertation. Stanford University, Stanford, CA.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2, 51–59.

Seth Gilbert and Nancy Ann Lynch. 2012. Perspectives on the CAP theorem. *IEEE Computer Society* 45, 2 (2012), 30–36.

Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 15–28.

Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. 2004. Load balancing in dynamic structured P2P systems. In *23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, Vol. 4. IEEE, 2253–2262.

Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2015. Concise server-wide causality management for eventually consistent data stores. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 66–79.

Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *ACM SIGMOD Record*, Vol. 25. ACM, 173–182.

Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems* 31, 1, 133–160.

Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Elsevier.

Venkat N. Gudivada, Ricardo Baeza-Yates, and Vijay V. Raghavan. 2015. Big data: Promises and problems. *Computer* 48, 20–23.

Coda Hale. 2010. You can't sacrifice partition tolerance. Retrieved February 20, 2018 from https://codahale.com/you-cant-sacrifice-partition-tolerance/.

Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 77–85.

Pat Helland. 2007. Life beyond distributed transactions: An apostate's opinion. In *CIDR*. 132–141.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3, 463–492.

Mikael Hogqvist, Seif Haridi, Nico Kruber, Alexander Reinefeld, and Thorsten Schutt. 2008. Using global information for load balancing in DHTs. In *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW'08)*. IEEE, 236–241.

Xiangdong Huang, Jianmin Wang, Yu Zhong, Shaoxu Song, and Philip S. Yu. 2015. Optimizing data partition for scaling out NoSQL cluster. *Concurrency and Computation: Practice and Experience* 27, 18, 5793–5809.

Borislav Iordanov. 2010. HyperGraphDB: A generalized graph database. In *International Conference on Web-Age Information Management*. Springer, 25–36.

U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. Pegasus: A peta-scale graph mining system implementation and observations. In *9th IEEE International Conference on Data Mining (ICDM'09)*. IEEE, 229–238.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. ACM, 654–663.

David R. Karger and Matthias Ruhl. 2006. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory of Computing Systems* 39, 6, 787–804.

Mijung Kim and K. Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering* 72, 285–303.

Won Kim. 1990. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering* 2, 3, 327–341.

Martin Kleppmann. 2015. A critique of the CAP theorem. *arXiv preprint arXiv:1509.05393* (2015).

Debra Knisley and Jeff Knisley. 2007. Graph theoretic models in chemistry and molecular biology. *Handbook of Applied Algorithms*. Wiley-IEEE Press, 85–113.

Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. 2004. Peer-to-peer support for massively multiplayer games. In *23rd Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, Vol. 1. IEEE.

Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. 2011. Fast and cost-effective online load-balancing in distributed range-queriable systems. *IEEE Transactions on Parallel and Distributed Systems* 22, 8, 1350–1364.

Ioannis Konstantinou, Dimitrios Tsoumakos, Ioannis Mytilinis, and Nectarios Koziris. 2013. DBalancer: Distributed load balancing for NoSQL data-stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1037–1040.

YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2010. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 75–86.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 31–46.

Aapo Kyrola and Carlos Guestrin. 2014. GraphChi-DB: Simple design for a scalable graph database system–on just a PC. *arXiv preprint arXiv:1403.0701*.

Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10, 4, 360–391.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2, 35–40.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7, 558–565.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* 100, 9, 690–691.

Thuy Ngoc Le and Tok Wang Ling. 2016. Survey on keyword search over XML documents. *ACM SIGMOD Record* 45, 3, 17–28.

Sin Yeung Lee, Tok Wang Ling, and Hua-Gang Li. 2000. Hierarchical compact cube for range-max queries. In *VLDB*. 232–241.

Mihai Letia, Nuno Preguiça, and Marc Shapiro. 2010. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS Operating Systems Review* 44, 2, 29–34.

Justin Levandoski, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Transaction processing techniques for modern hardware and the cloud. In *IEEE Data Engineering Bulletin*. 50.

Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. 2011. Deuteronomy: Transaction support for cloud data. In *CIDR*, Vol. 11. 123–133.

Daniel Mark Lewin. 1998. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT.

Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the choice of consistency levels in replicated systems. In *USENIX Annual Technical Conference*. 281–292.

Richard J. Lipton and Jonathan S. Sandberg. 1988. *PRAM: A Scalable Shared Memory*. Technical Report TR-180-88. Princeton University, Department of Computer Science, Princeton, NJ.

Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. 2014. Warranties for faster strong consistency. In *NSDI*, Vol. 14. 503–517.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles*. ACM, 401–416.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, Vol. 13. 313–328.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2014. Don't settle for eventual consistency. *Communications of the ACM* 57, 5, 61–68.

Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. 2010. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence* 5, 8 (2010), 716–727.

Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. 2005. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials* 7, 2, 72–93.

Qiong Luo and Jeffrey F. Naughton. 2001. Form-based proxy caching for database-backed web sites. In *VLDB*. 191–200.

Prince Mahajan, Alvisi Lorenzo, and Dahlin Mike. 2011. *Consistency, Availability, and Convergence*. Technical Report TR-11-22. University of Texas at Austin.

D. Maier. 1989. Why database languages are a bad idea. In *Proceedings of the International Workshop on Database Programming Languages*. 277–287.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 135–146.

Francesco Marchioni. 2012. *Infinispan Data Grid Platform*. Packt Publishing Ltd, Birmingham, UK.

Mark Massé. 2012. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Sebastopol, CA.

Friedemann Mattern. 1989. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms* 1, 23, 215–226.

Andrew McAfee, Erik Brynjolfsson, Thomas H. Davenport, D. J. Patil, and Dominic Barton. 2012. Big data. *The Management Revolution. Harvard Business Review* 90, 61–67.

Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. 2014. A performance evaluation of open source graph databases. ACM Press, New York, NY, 11–18.

Dean Meltz, Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls. 2004. *An Introduction to IMS: Your Complete Guide to IBM's Information Management System*. IBM Press, Indianapolis, IN.

Ralph C. Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology*. Springer, 218–238.

Matthew P. Milano and Andrew C. Myers. 2016. *Mixing Consistency in Geodistributed Transactions: Technical Report*. Cornell University.

Sang Nguyen and Stefano Pallottino. 1989. Hyperpaths and shortest hyperpaths. *Combinatorial Optimization* 1403, 258–271.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4, 351–385.

João Paiva and Luís Rodrigues. 2015. On data placement in distributed systems. *ACM SIGOPS Operating Systems Review* 49, 1, 126–130.

João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. 2015. Auto placer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems* 9, 4, 19.

Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM* 26, 4, 631–653.

D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 3, 240–247.

Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. 2013. Portable cloud applications–from theory to practice. *Future Generation Computer Systems* 29, 6, 1417–1430.

Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, Vol. 31. ACM, 288–301.

C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. 1999. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems* 32, 3, 241–280.

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. 2012. Managing large graphs on multi-cores with graph awareness. In *USENIX (ATC'12)*. 41–52.

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. 2015. A distributed algorithm for large-scale graph partitioning. *ACM Transactions on Autonomous and Adaptive Systems* 10, 2, 12.

Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. 2017. Characterizing and adapting the consistency-latency trade-off in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems* 11, 4, 20.

Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. 2003. Load balancing in structured P2P systems. In *Peer-to-Peer Systems II*. Springer, 68–79.

Jun Rao, Eugene J. Shekita, and Sandeep Tata. 2011. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment* 4, 4, 243–254.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network, Vol. 31. ACM SIGCOMM.

Michel Raynal and André Schiper. 1995. From causal consistency to sequential consistency in shared memory systems. In *Foundations of Software Technology and Theoretical Computer Science*. Springer, 180–194.

David Patrick Reed. 1978. Naming and synchronization in a decentralized computer system. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

James B. Rothnie and Nathan Goodman. 1977. A survey of research and development in distributed database management. In *Proceedings of the 3rd International Conference on Very Large Databases*. VLDB Endowment, 48–62.

Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*. Springer, 329–350.

Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

Hans Sagan. 1994. *Space-Filling Curves*. Springer-Verlag, Berlin.

Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. 2015. Towards automated polyglot persistence. In *BTW*. 73–82.

Daniel Schall and Theo Härder. 2015. Dynamic physiological partitioning on a shared-nothing database cluster. In *31st International Conference on Engineering (ICDE'15)*. IEEE, 1095–1106.

Adam Schenker, Abraham Kandel, Horst Bunke, and Mark Last. 2005. *Graph-Theoretic Techniques for Web Content Mining*. Vol. 62. World Scientific, Singapore.

Kirk Schloegel, George Karypis, and Vipin Kumar. 2000. Parallel multilevel algorithms for multi-constraint graph partitioning. In *European Conference on Parallel Processing*. Springer, 296–310.

Cristina Schmidt and Manish Parashar. 2008. Squid: Enabling search in DHT-based systems. *J.ournal of Parallel and Distrib.uted Computing* 68, 7, 962–975.

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. 2006. Structured overlay without consistent hashing: Empirical results. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID' 06)*, Vol. 2. IEEE, 8–8.

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. 2007. A structured overlay for multi-dimensional range queries. In *European Conference on Parallel Processing*. Springer, 503–513.

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. 2008a. Range queries on structured overlay networks. *Computer Communications* 31, 280–291.

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. 2008b. Scalaris: Reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ACM, 41–48.

Ayon Sen, ASM Sohidull Islam, and Md Yusuf Sarwar Uddin. 2015. MARQUES: Distributed multi-attribute range query solution using space filling curve on DTHs. In *International Conference on Networking Systems and Security (NSysS'15)*. 1–9.

Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 505–516.

Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 1996. Data models. *ACM Computing Surveys* 28, 1, 105–108.

Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 133–142.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 385–400.

Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1222–1230.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4, 149–160.

Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Engineering Bulletin* 9, 1, 4–9.

Michael Stonebraker. 2010a. Errors in database systems, eventual consistency, and the cap theorem. *Communications of the ACM, BLOG@ACM.* https://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext.

Michael Stonebraker. 2010b. SQL databases v. NoSQL databases. *Communications of the ACM* 53, 4, 10–11.

Michael Stonebraker. 2011. Why enterprises are uninterested in NoSQL. *Communications of the ACM, BLOG@ACM.* https://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext.

Michael Stonebraker. 2012. Newsql: An alternative to NoSQL and old SQL for new OLTP apps. *Communications of the ACM, BLOG@ACM.* https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext.

Michael Stonebraker, Jennie Duggan, Leilani Battle, and Olga Papaemmanouil. 2013. SciDB DBMS research at MIT. *IEEE Database Engineering Bulletin* 36, 4, 21–30.

Michael Stonebraker and Dorothy Moore. 1995. Object Relational DBMSs: The Next Great Wave. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Veda C. Storey and Il-Yeol Song. 2017. Big data technologies and management: What conceptual modeling can do. *Data & Knowledge Engineering* 108 (2017), 50–67.

Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1887–1901.

Robert W. Taylor and Randall L. Frank. 1976. CODASYL data-base management systems. *ACM Computing Surveys* 8, 1, 67–103.

Doug Terry. 2013. Replicated data consistency explained through baseball. *Communications of the ACM* 56, 12, 82–89.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 140–149.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 309–324.

Robert H. Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4, 2, 180–209.

D. C. Tsichritzis and Frederick H. Lochovsky. 1976. Hierarchical data-base management: A survey. *ACM Computing Surveys* 8, 1, 105–123.

Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. ACM, 333–342.

Ata Turk, R. Oguz Selvitopi, Hakan Ferhatosmanoglu, and Cevdet Aykanat. 2014. Temporal workload-aware replicated partitioning for social networks. *IEEE Transactions on Knowledge and Data Engineering* 26, 11, 2832–2845.

Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. 2011. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review* 41, 1, 45–52.

Paolo Viotti and Marko Vukolić. 2016. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys* 49, 1, 19.

Werner Vogels. 2009. Eventually consistent. *Communications of the ACM* 52, 1, 40–44.

Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data consistency properties and the trade-offs in commercial cloud storage: The consumers' perspective. In *CIDR*, Vol. 11. 134–143.

Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu, and Baile Shi. 2004. Scalable mining of large disk-based graph databases. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 316–325.

Jim Webber. 2012. A programmatic introduction to Neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 217–218.

Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*.

Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems* 20, 3, 239–282.

C. Zaniolo, H. A. t Kaci, D. Beech, S. Cammarata, L. Kerschberg, and D. Maier. 1985. *Object-Oriented Database and Knowledge Systems*. Technical Report DB-038-85, Microelectronics and Computer Consortium (MCC), Austin, Tex.

Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. 2013. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. *arXiv preprint arXiv:1310.3107*.

Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 75–87.

Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2007. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Transactions on Database Systems* 32, 2 (2007), 1–40.

Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. 2001. *Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing*. Technical Report UCB-CSD-01-1141. University of California Berkeley, Berkley, CA.

Received October 2016; revised October 2017; accepted November 2017