9 An Overview of the NoSQL World

Liang Zhao, Sherif Sakr, and Anna Liu

CONTENTS

Over the past decade, rapidly growing Internet-based services such as e-mail, blogging, social networking, search, and e-commerce have substantially redefined the way consumers communicate, access contents, share information and purchase products. Relational database management systems (RDBMS) have been considered as the *one-size-fits-all* solution for data persistence and retrieval for decades. However, the ever-increasing need for scalability and new application requirements have created new challenges for traditional RDBMS. Recently, a new generation of low-cost, high-performance database software, aptly named as *NoSQL* (Not Only SQL), has emerged to challenge the dominance of RDBMS. The main features of these systems include ability to horizontally scale, supporting weaker consistency models, using flexible schemas and data models and supporting simple low-level query interfaces. In this chapter, we explore the recent advancements and the new approaches of web-scale data management. We discuss the advantages and disadvantages of several recently introduced approaches and its suitability to support certain class of applications and

end users. Finally, we present and discuss some of the current challenges and open research problems to be tackled to improve the current state-of-the-art.

9.1 INTRODUCTION

Over the past decade, rapidly growing Internet-based services such as e-mail, blogging, social networking, search, and e-commerce have substantially redefined the way consumers communicate, access contents, share information, and purchase products. In particular, the recent advances in the web technology have made it easy for any user to provide and consume content of any form. For example, building a personal web page (e.g., Google Sites*), starting a blog (e.g., WordPress,† Blogger,‡ LiveJournal§), and making both searchable for the public have become a commodity that is available for users all over the world. Arguably, the main goal of the next wave is to facilitate the job of implementing every application as a distributed, scalable, and widely accessible service on the web. Services such as Facebook,¶ Flickr,** YouTube,†† Zoho,‡‡ and LinkedIn§§ are currently leading this approach. Such applications are both *data-intensive* and very *interactive*. For example, the Facebook social network has announced that it has more than 800 millions of monthly active users.¶¶ Each user has an average of 130 friendship relations. Moreover, there are about 900 million objects that registered users interact with, such as pages, groups, events, and community pages. Other smaller scale social networks such as LinkedIn, which is mainly used for professionals, has more than 120 million registered users. Twitter has also claimed to have over 100 million active monthly users. Therefore, it becomes an ultimate goal to make it easy for every application to achieve such high scalability and availability goals with minimum efforts.

In general, relational database management systems (e.g., MySQL, PostgreSQL, SQL Server, Oracle) have been considered as the *one-size-fits-all* solution for data persistence and retrieval for decades. They have matured after extensive research and development efforts and very successfully created a large market and solutions in different business domains. However, the ever-increasing need for scalability and new application requirements have created new challenges for traditional RDBMS. Therefore, recently, there has been some dissatisfaction with this *one-size-fits-all* approach in some web-scale applications [58]. Nowadays, the most common architecture to build enterprise web applications is based on a three-tier approach: the web server layer, the application server layer, and the data layer. In practice, data partitioning [50] and data replication [40] are two well-known strategies to achieve the availability, scalability, and performance improvement goals in the distributed

* http://sites.google.com/.
† http://wordpress.org/.
‡ http://www.blogger.com/.
§ http://www.livejournal.com/.
¶ http://www.facebook.com/.
** http://www.flickr.com/.
†† http://www.youtube.com/.
‡‡ http://www.zoho.com/.
§§ http://www.linkedin.com/.
¶¶ http://www.facebook.com/press/info.php?statistics.

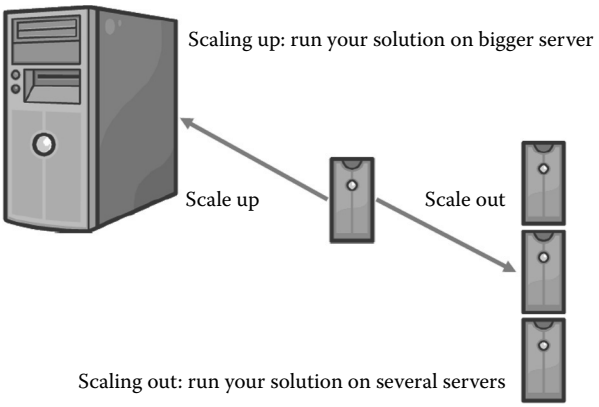

FIGURE 9.1 Database scalability options.

data management world. In particular, when the application load increases, there are two main options for achieving scalability at the database tier that enables the applications to cope with more client requests (Figure 9.1) as follows:

1. *Scaling up*: aims at allocating a bigger machine to act as database servers.
2. *Scaling out*: aims at *replicating* and *partitioning* data across more machines.

In fact, the scaling up option has the main drawback that large machines are often very expensive and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. Alternatively, it is both extensible and economical–especially in a dynamic workload environment–to scale out by adding storage space or buying another commodity server, which fits well with the new *pay-as-you-go* philosophy of cloud computing.

Recently, a new generation of low-cost, high-performance database software has emerged to challenge the dominance of relational database management systems. A big reason for this movement, named as *NoSQL* (Not Only SQL), is that different implementations of web, enterprise, and cloud computing applications have different database requirements (e.g., not every application requires rigid data consistency). For example, for high-volume web sites (e.g., eBay, Amazon, Twitter, Facebook), scalability and high availability are essential requirements that cannot be compromised. For these applications, even the slightest outage can have significant financial consequences and impacts customers' trust.

In general, the *CAP* theorem [15,34] and the *PACELC* model [1] describe the existence of direct tradeoffs between consistency and availability as well as consistency and latency. For example, the *CAP* theorem shows that a distributed database system can only choose at most two out of three properties: *Consistency*, *Availability*, and *tolerance to Partitions*. Therefore, there is a plethora of alternative consistency models, which have been introduced for offering different performance tradeoffs such as *session guarantees*, *causal consistency* [7], *causal+ consistency* [48], and *parallel snapshot isolation* [57]. In practice, the new wave of

NoSQL systems decided to compromise on the strict consistency requirement. In particular, they apply a relaxed consistency policy called *eventual consistency* [63], which guarantees that if no new updates are made to a replicated object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. In particular, these new NoSQL systems have a number of design features in common:

- The ability to horizontally scale out throughput over many servers.
- A simple call level interface or protocol (in contrast to a SQL binding).
- Supporting weaker consistency models in contrast to ACID guaranteed properties for transactions in most traditional RDBMS. These models are usually referred to as *BASE* models (*B*asically *A*vailable, *S*oft state, *E*ventually consistent) [53].
- Efficient use of distributed indexes and RAM for data storage.
- The ability to dynamically define new attributes or data schema.

These design features are made to achieve the following system goals:

- *Availability*: They must always be accessible even during network failure or a whole datacenter going offline.
- *Scalability*: They must be able to support very large databases with very high request rates at very low latency.
- *Elasticity*: They must be able to satisfy changing application requirements in both directions (scaling up or scaling down). Moreover, the system must be able to gracefully respond to these changing requirements and quickly recover its steady state.
- *Load balancing*: They must be able to automatically move load between servers so that most of the hardware resources are effectively utilized and to avoid any resource overloading situations.
- *Fault tolerance*: They must be able to deal with the situation that the rarest hardware problems go from being freak events to eventualities. While hardware failure is still a serious concern, this concern needs to be addressed at the architectural level of the database, rather than requiring developers, administrators, and operations staff to build their own redundant solutions.
- *Ability to run in a heterogeneous environment*: On scaling out environment, there is a strong trend toward increasing the number of nodes that participate in query execution. It is nearly impossible to get homogeneous performance across hundreds or thousands of compute nodes. Part failures that do not cause complete node failure, but result in degraded hardware performance become more common at scale. Hence, the system should be designed to run in a heterogeneous environment and must take appropriate measures to prevent performance degradation that are due to parallel processing on distributed nodes.

This chapter explores the recent advancements and the new approaches of the web-scale data management. We discuss the advantages and the disadvantages of each approach and its suitability to support certain class of applications and end users. Section 9.2 describes the NoSQL systems that are introduced and used internally in the key players: Google, Yahoo, and Amazon, respectively. Section 9.3 provides an overview of a set of open-source projects, which have been designed following the main principles of the NoSQL systems. Section 9.4 discusses the notion of providing database management as a service and gives an overview of the main representative systems and their challenges. The web-scale data management tradeoffs and open research challenges are discussed in Section 9.5 before we conclude the chapter in Section 9.7.

9.2 NoSQL KEY SYSTEMS

This section provides an overview of the main NoSQL systems which has been introduced and used internally by three of the key players in the web-scale data management domain: Google, Yahoo, and Amazon.

9.2.1 Google: Bigtable

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size (petabytes of data) across thousands of commodity servers [21]. It has been used by more than 60 Google products and projects such as Google search engine,* Google Finance,† Orkut,‡ Google Docs,§ and Google Earth.¶ These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

Bigtable does not support a full relational data model. However, it provides clients with a simple data model that supports dynamic control over data layout and format. In particular, a Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. Thus, clients usually need to serialize various forms of structured and semistructured data into these strings. A concrete example that reflects some of the main design decisions of Bigtable is the scenario of storing a copy of a large collection of web pages into a single table. Figure 9.2 illustrates an example of this table where *URLs* are used as row keys and various aspects of web pages as column names. The contents of the web pages are stored in a single column that stores multiple versions of the page under the timestamps when they were fetched.

The row keys in a table are arbitrary strings where every read or write of data under a single row key is atomic. Bigtable maintains the data in lexicographic order

* http://www.google.com/.
† http://www.google.com/finance.
‡ http://www.orkut.com/.
§ http://docs.google.com/.
¶ http://earth.google.com/.

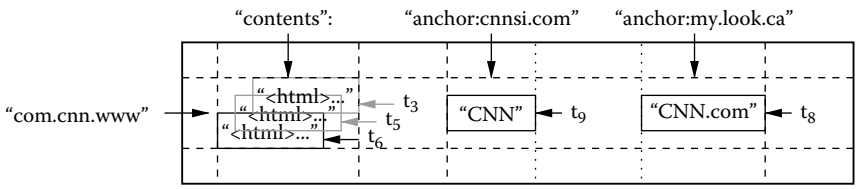

FIGURE 9.2 Sample Bigtable structure. (From F. Chang et al., *ACM Trans. Comput. Syst.*, 26, 2008.)

by row key where the row range for a table is dynamically partitioned. Each row range is called a *tablet*, which represents the unit of distribution and load balancing. Thus, reads of short row ranges are efficient and typically require communication with only a small number of machines. Bigtables can have an unbounded number of columns that are grouped into sets called *column families*. These column families represent the basic unit of access control. Each cell in a Bigtable can contain multiple versions of the same data that are indexed by their timestamps. Each client can flexibly decide the number of n versions of a cell that need to be kept. These versions are stored in decreasing timestamp order so that the most recent versions can be always read first.

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights. Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. At the transaction level, Bigtable supports only *single-row* transactions, which can be used to perform atomic read–modify–write sequences on data stored under a single row key (i.e., no general transactions across row keys).

At the physical level, Bigtable uses the distributed Google File System (GFS) [33] to store log, and data files. The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Bigtable relies on a distributed lock service called *Chubby* [17], which consists of five active replicas, one of which is elected to be the *master* and actively serves requests. The service is live when a majority of the replicas are running and can communicate with each other. Bigtable uses Chubby for a variety of tasks such as (1) ensuring that there is at most one active master at any time, (2) storing the bootstrap location of Bigtable data, (3) storing Bigtable schema information and to the access control lists. The main limitation of this design is that if Chubby becomes unavailable for an extended period of time, the whole Bigtable becomes unavailable. At the runtime, each Bigtable is allocated to one master server and many tablet servers, which can be dynamically added (or removed) from a cluster based on the changes in workloads. The master server is responsible for assigning tablets to tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations. Each tablet server manages a set of tablets. The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

9.2.2 YAHOO: PNUTS

The *PNUTS* system (renamed later to Sherpa) is a massive-scale hosted database system that is designed to support Yahoo!s web applications [25,56]. The main focus of the system is on data serving for web applications, rather than complex queries. It relies on a simple relational model where data is organized into tables of records with attributes. In addition to typical data types, *blob* is a main valid data type, which allows arbitrary structures to be stored inside a record, but not necessarily large binary objects like images or audio. The PNUTS system does not enforce constraints such as referential integrity on the underlying data. Therefore, the schema of these tables are flexible where new attributes can be added at any time without halting any query or update activity. In addition, it is not required that each record have values for all attributes.

Figure 9.3 illustrates the system architecture of PNUTS. The system is divided into regions where each region contains a full complement of system components and a complete copy of each table. Regions are typically, but not necessarily, geographically distributed. Therefore, at the physical level, data tables are horizontally partitioned into groups of records called *tablets*. These tablets are scattered across many servers where each server might have hundreds or thousands of tablets. The assignment of tablets to servers is flexible in a way that allows balancing the workloads by moving a few tablets from an overloaded server to an underloaded server.

The query language of PNUTS supports selection and projection from a single table. Operations for updating or deleting existing records must specify the primary key. The system is designed primarily for online serving workloads that consist mostly of queries that read and write single records or small groups of records. Thus, it provides a *multiget* operation that supports retrieving multiple records in parallel by specifying a set of primary keys and an optional predicate. The *router* component (Figure 9.3) is responsible of determining which storage unit needs to be accessed for a given record to be read or written by the client. Therefore, the primary-key space of a table is divided into intervals where each interval corresponds to one tablet. The router stores an interval mapping that defines the boundaries of each tablet and maps each tablet to a storage unit. The query model of PNUTS does not support join operations that are too expensive in such massive scale systems.

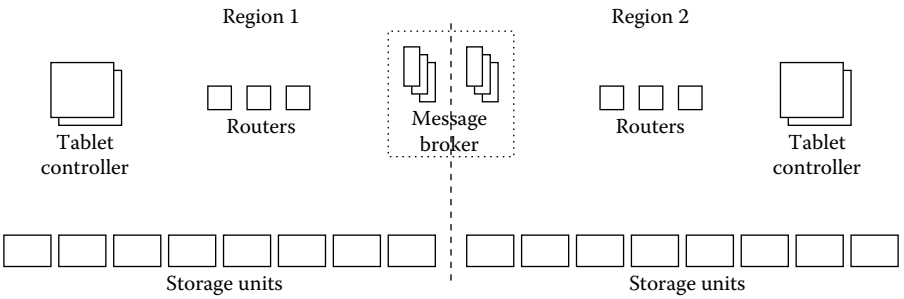

FIGURE 9.3 PNUTS system architecture. (From B. F. Cooper et al., *PVLDB*, 1, 1277–1288, 2008.)

The PNUTS system does not have a traditional database log or archive data. However, it relies on a pub/submechanism that act as a redo log for replaying updates that are lost before being applied to disk due to failure. In particular, PNUTS provides a consistency model that is between the two extremes of general serializability and eventual consistency [63]. The design of this model is derived from the observation that web applications typically manipulate one record at a time while different records may have activity with different geographic locality. Thus, it provides *per-record timeline* consistency where all replicas of a given record apply all updates to the record in the same order. In particular, for each record, one of the replicas (independently) is designated as the master where all updates to that record are forwarded to the master. The master replica for a record is adaptively changed to suit the workload where the replica receiving the majority of write requests for a particular record is selected to be the master for that record. Relying on the per-record timeline consistency model, the PNUTS system supports the following range of API calls with varying levels of consistency guarantees

- *Read-any*: This call has a lower latency as it returns a possibly stale version of the record.
- *Read-critical (required version)*: This call returns a version of the record that is strictly newer than or the same as the *required version*.
- *Read-latest*: This call returns the latest copy of the record that reflects all writes that have succeeded. It is expected that the *read-critical* and *read-latest* can have a higher latency than *read-any* if the local copy is too stale and the system needs to locate a newer version at a remote replica.
- *Write*: This call gives the same ACID guarantees as a transaction with a single write operation in it (e.g., blind writes).
- *Test-and-set-write (required version)*: This call performs the requested write to the record if and only if the present version of the record is the same as the required version. This call can be used to implement transactions that first read a record, and then do a write to the record based on the read, e.g., incrementing the value of a counter.

Since the system is designed to scale to cover several worldwide replicas, automated failover, and load balancing is the only way to manage the operations load. Therefore, for any failed server, the system automatically recovers by copying data from a replica to other live servers.

9.2.3 Amazon: Dynamo

Amazon runs a worldwide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. In this environment, there are strict operational requirements on Amazon's platform in terms of performance, reliability, and efficiency, and to support Amazon's continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust.

The Dynamo system [30] is a highly available and scalable distributed key/value-based datastore built for supporting *internal* Amazon's applications. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs among availability, consistency, cost-effectiveness, and performance. There are many services on Amazons platform that only need primary-key access to a data store. The common pattern of using a relational database would lead to inefficiencies and limit the ability to scale and provide high availability. Thus, Dynamo provides a simple primary-key-only interface to meet the requirements of these applications. The query model of the Dynamo system relies on simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (blobs) identified by unique keys. No operations span multiple data items.

Dynamo's partitioning scheme relies on a variant of consistent hashing mechanisms [39] to distribute the load across multiple storage hosts. In this mechanism, the output range of a hash function is treated as a fixed circular space or ring (i.e., the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space, which represents its position on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

In the Dynamo system, each data item is replicated at N hosts where N is a parameter configured per-instance. Each key k is assigned to a coordinator node. The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the $(N - 1)$ clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. As illustrated in Figure 9.4, node B replicates the key k at nodes C and D

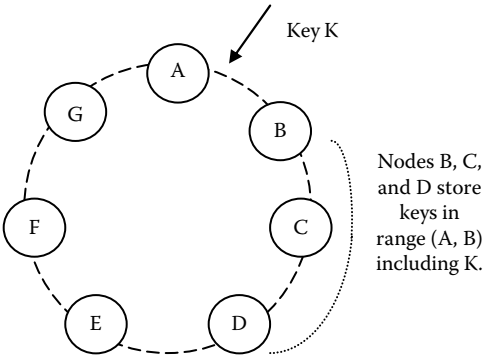

FIGURE 9.4 Partitioning and replication of keys in the Dynamo ring. (From G. DeCandia et al., Dynamo: Amazon's highly available key-value store, in *SOSP*, pp. 205–220, 2007.)

in addition to storing it locally. Node *D* will store the keys that fall in the ranges (*A*, *B*), (*B*, *C*), and (*C*, *D*). The list of nodes that is responsible for storing a particular key is called the preference list. The system is designed so that every node in the system can determine which nodes should be in this list for any particular key.

9.3 NoSQL OPEN SOURCE PROJECTS

In practice, most NoSQL data management systems that are introduced by the key players (e.g., Bigtable, Dynamo, PNUTS) are meant for their internal use only and are thus, not available for public users. Therefore, many open-source projects have been built to implement the concepts of these systems and make it available for public users [18,54]. Due to the ease in which they can be downloaded and installed, these systems have attracted a lot of interest from the research community. There are not many details that have been published about the implementation of most of these systems. In general, the NoSQL open-source projects can be broadly classified into the following categories:

- *Key-value stores*: These systems use the simplest data model, which is a collection of objects where each object has a unique key and a set of attribute/value pairs.
- *Document stores*: These systems have the data models that consists of objects with a variable number of attributes with a possibility of having nested objects.
- *Extensible record stores*: They provide variable-width tables (Column Families) that can be partitioned vertically and horizontally across multiple nodes.

Here, we give a brief introduction about some of these projects. For the full list, we refer the reader to the NoSQL database website.*

Cassandra† is presented as a highly scalable, eventually consistent, distributed, structured key-value store [44,45]. It was open-sourced by Facebook in 2008. It is designed by Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik (Facebook engineer). Cassandra brings together the distributed systems technologies from Dynamo and the data model from Google's Bigtable. Like Dynamo, Cassandra is eventually consistent. Like Bigtable, Cassandra provides a column family-based data model richer than typical key/value systems. In Cassandra's data model, *column* is the lowest/smallest increment of data. It is a tuple (triplet) that contains a name, a value, and a timestamp. A *column family* is a container for columns, analogous to the table in a relational system. It contains multiple columns, each of which has a name, value, and a timestamp, and are referenced by row keys. A *keyspace* is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e., a logical collection of tables) in RDBMS.

* http://NoSQL-database.org/.
† http://cassandra.apache.org/.

They can be seen as a namespace for ColumnFamilies and is typically allocated as one per application. *SuperColumns* represent columns that themselves have subcolumns (e.g., Maps). Like Dynamo, Cassandra provides a tunable consistency model that allows the ability to choose the consistency level that is suitable for a specific application. For example, it allows to choose how many acknowledgments are required to be received from different replicas before considering a *WRITE* operation to be successful. Similarly, the application can choose how many successful responses need to be received in the case of *READ* before returning the result to the client. In particular, every *write* operation can choose one of the following consistency levels:

a. *ZERO*: It ensures nothing. The write operation will be executed asynchronously in the system background.
b. *ANY*: It ensures that the write operation has been executed in at least one node.
c. *ONE*: It ensures that the write operation has been committed to at least 1 replica before responding to the client.
d. *QUORUM*: It ensures that the write has been executed on ($N/2 + 1$) replicas before responding to the client where N is the total number of system replicas.
e. *ALL*: It ensures that the write operation has been committed to all N replicas before responding to the client.

On the other hand, every *read* operation can choose one of the following available consistency levels:

a. *ONE*: It will return the record of the first responding replica.
b. *QUORUM*: It will query all replicas and return the record with the most recent timestamp once it has at least a majority of replicas ($N/2 + 1$) reported.
c. *ALL*: It will query all replicas and return the record with the most recent timestamp once all replicas have replied.

Therefore, any unresponsive replicas will fail the read operation. For read operations, in the *ONE* and *QUORUM* consistency levels, a consistency check is always done with the remaining replicas in the system background to fix any consistency issues.

HBase* is another project is based on the ideas of Bigtable system. It uses the Hadoop distributed filesystem (HDFS)† as its data storage engine. The advantage of this approach is that HBase does not need to worry about data replication, data consistency, and resiliency because HDFS already considers and deals with them. However, the downside is that it becomes constrained by the characteristics of HDFS, which is that it is not optimized for random read access. In the HBase architecture, data is stored in a farm of Region Servers. A *key-to-server* mapping is used to locate

* http://hbase.apache.org/.
† http://hadoop.apache.org/hdfs/.

the corresponding server. The in-memory data storage is implemented using a distributed memory object caching system called *Memcache*,* while the on-disk data storage is implemented as a HDFS file residing in the Hadoop data node server.

The HyperTable† project is designed to achieve a high performance, scalable, distributed storage, and processing system for structured and unstructured data. It is designed to manage the storage and processing of information on a large cluster of commodity servers, providing resilience to machine and component failures. Like HBase, Hypertable also runs over HDFS to leverage the automatic data replication, and fault tolerance that it provides. In HyperTable, data is represented in the system as a multidimensional table of information. The HyperTable systems provides a low-level API and Hypertable Query Language (HQL) that provides the ability to create, modify, and query the underlying tables. The data in a table can be transformed and organized at high speed by performing computations in parallel, pushing them to where the data is physically stored.

CouchDB‡ is a document-oriented database that is written in Erlang and can be queried and indexed in a MapReduce fashion using JavaScript. In CouchDB, documents are the primary unit of data. A CouchDB document is an object that consists of named fields. Field values may be strings, numbers, dates, or even ordered lists and associative maps. Hence, a CouchDB database is a flat collection of documents where each document is identified by a unique ID. CouchDB provides a RESTful HTTP API for reading and updating (add, edit, delete) database documents. The CouchDB document update model is lockless and optimistic. Document edits are made by client applications. If another client was editing the same document at the same time, the client gets an edit conflict error on save. To resolve the update conflict, the latest document version can be opened, the edits reapplied, and the update retried again. Document updates are all or nothing, either succeeding entirely or failing completely. The database never contains partially saved or edited documents.

MongoDB§ is another example of distributed schema-free document-oriented database, which is created at *10gen*.¶ It is implemented in C++ but provides drivers for a number of programming languages including C, C++, Erlang. Haskell, Java, JavaScript, Perl, PHP, Python, Ruby, and Scala. It also provides a JavaScript command-line interface. MongoDB stores documents as *BSON* (Binary JSON), which are binary encoded JSON like objects. BSON supports nested object structures with embedded objects and arrays. At the heart of MongoDB is the concept of a *document* that is represented as an ordered set of keys with associated values. A *collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table. Collections are schema-free. This means that the documents within a single collection can have any number of different shapes. MongoDB groups collections into *databases*. A single instance of MongoDB can host several databases, each of which can be thought of as completely independent. It provides eventual consistency

* http://memcached.org/.
† http://hypertable.org/.
‡ http://couchdb.apache.org/.
§ http://www.mongodb.org/.
¶ http://www.10gen.com/.

guarantees in a way that a process could read an old version of a document even if another process has already performed an update operation on it. In addition, it provides no transaction management so that if a process reads a document and writes a modified version back to the database, there is a possibility that another process may write a new version of the same document between the read and the write operation of the first process. MongoDB supports indexing the documents on multiple fields. In addition, it provides a very rich API interface that supports different batch operations and aggregate functions.

Many other variant projects have followed the NoSQL movement and support different types of data stores such as key-value stores (e.g., Voldemort,* Dynomite†), document stores (e.g., Riak‡), and graph stores (e.g., Neo4j,§ DEX¶).

9.4 DATABASE-AS-A-SERVICE

Multitenancy, a technique which is pioneered by *salesforce.com*,** is an optimization mechanism for hosted services in which multiple customers are consolidated onto the same operational system and thus the economy of scale principles help to effectively drive down the cost of computing infrastructure. In particular, multitenancy allows pooling of resources that improves utilization by eliminating the need to provision each tenant for their maximum load. Therefore, multitenancy is an attractive mechanism for both of the service providers who are able to serve more customers with a smaller set of machines, and also to customers of these services who do not need to pay the price of renting the full capacity of a server. Database-as-a-service (DaaS) is a new paradigm for data management in which a third–party service provider hosts a database as a service [3,37]. The service provides data management for its customers and thus alleviates the need for the service user to purchase expensive hardware and software, deal with software upgrades, and hire professionals for administrative and maintenance tasks. Since using an external database service promises reliable data storage at a low cost, it represents a very attractive solution for companies especially that of startups. In this section, we give an overview of the state-of-the-art of different options of DaaS from the key players Google, Amazon, and Microsoft.

9.4.1 Google Datastore

Google has released the Google AppEngine datastore,†† which provides a scalable schemaless object data storage for web application. It performs queries over data objects, known as *entities*. An entity has one or more *properties* where one property can be a reference to another entity. Datastore entities are schemaless where two entities of the same kind are not obligated to have the same properties, or use the same value types

* http://project-voldemort.com/.
† http://wiki.github.com/cliffmoon/dynomite/dynomite-framework.
‡ http://wiki.basho.com/display/RIAK/Riak.
§ http://neo4j.org/.
¶ http://www.dama.upc.edu/technology-transfer/dex.
** http://www.salesforce.com/.
†† http://code.google.com/appengine/docs/python/datastore/.

```
SELECT [* |__key__] FROM <kind>
[WHERE <condition> [AND <condition> …]]
[ORDERBY <property> [ASC|DESC] [, <property> [ASC | DESC] …]]
[LIMIT [<offset>,] <count>]
[OFFSET <offset>]

<condition> := <property> {< | <= | > | >= | = | != } <value>
<condition> := <property> IN <list>
<condition> := ANCESTOR IS <entity or key>
```

FIGURE 9.5 Basic GQL syntax.

for the same properties. Each entity also has a key that uniquely identifies the entity. The simplest key has a kind and a unique numeric ID provided by the datastore. An application can fetch an entity from the datastore using its key or by performing a query that matches the entity's properties. A query can return zero or more entities and can return the results sorted by property values. A query does not allow the number of results returned by the datastore to be very large to conserve memory and run time.

With the AppEngine datastore, every attempt to create, update, or delete an entity happens in a transaction. A transaction ensures that every change made to the entity is saved to the datastore. However, in the case of failure, none of the changes are made. This ensures consistency of data within an entity. The datastore uses optimistic concurrency to manage transactions. The datastore replicates all data to multiple storage locations, so if one storage location fails, the datastore can switch to another and still access the data. To ensure that the view of the data stays consistent as it is being updated, an application uses one location as its primary location and changes to the data on the primary are replicated to the other locations in parallel. An application switches to an alternate location only for large failures. For small failures in primary storage, such as a single machine becoming unavailable temporarily, the datastore waits for primary storage to become available again to complete an interrupted operation. This is necessary to give the application a reasonably consistent view of the data, since alternate locations may not yet have all of the changes made to the primary. In general, an application can choose between two read policies: (1) a read policy of *strong consistency*, which always reads from the primary storage location, and (2) a policy of *eventual consistency* [63], which will read from an alternate location when the primary location is unavailable.

The AppEngine datastore provides a Python* interface, which includes a rich data modeling API and a SQL-like query language called *GQL*.† Figure 9.5 depicts the basic syntax of GQL. A GQL query returns zero or more entities or keys of the requested kind. In principle, a GQL query cannot perform a SQL-like "join" query. Every GQL query always begins with either *SELECT* FROM* or *SELECT (key) FROM* followed by the name of the kind. The optional *WHERE* clause filters the result set to those entities that meet one or more conditions. Each condition compares a property of the entity with a value using a comparison operator. GQL does

* http://www.python.org/.

† http://code.google.com/appengine/docs/python/datastore/gqlreference.html.

not have an *OR* operator. However, it does have an IN operator that provides a limited form of *OR*. The optional *ORDER BY* clause indicates that results should be returned are sorted by the given properties in either ascending (ASC) or descending (DESC) order. An optional *LIMIT* clause causes the query to stop returning results after the first count entities. The *LIMIT* can also include an offset to skip the specified number of results to find the first result to be returned. An optional *OFFSET* clause can specify an offset if the no *LIMIT* clause is present. Chohan et al. [23] have presented *AppScale* as an open-source extension to the Google AppEngine that facilitates distributed execution of its applications over virtualized cluster resources, including Infrastructure-as-a-Service (IaaS) cloud systems such as Amazon EC2 and Eucalyptus.* They have used AppScale to empirically evaluate and compare how well different NoSQL systems (e.g., Cassandra, HBase, Hypertable, MemcacheDB, MongoDB, Voldemort) map to the GAE Datastore API [16].

Google Cloud SQL† is another Google service that provide the capabilities and functionality of MySQL database servers, which are hosted in Google's cloud. Although there is tight integration of the services with the *Google App Engine*, it allows the software applications to easily move their data in and out of Google's cloud without any obstacles.

9.4.2 Amazon: S3/SimpleDB/Amazon RDS

Amazon Simple Storage Service (S3) is an online public storage web service offered by Amazon Web Services. Conceptually, S3 is an infinite store for objects of variable sizes. An object is simply a byte container, which is identified by a URI. Clients can read and update S3 objects remotely using a simple web services (SOAP or REST-based) interface. For example, *get*(*uri*) returns an object and *put*(*uri, bytestream*) writes a new version of the object. In principle, S3 can be considered as an online backup solution or for archiving large objects, which are not frequently updated.

Amazon has not published details on the implementation of S3. However, Brantner et al. [14] have presented initial efforts of building web-based database applications on top of S3. They described various protocols for storing, reading, and updating objects and indexes using S3. For example, the *record manager* component is designed to manage records where each record is composed of a key and payload data. Both key and payload are bytestreams of arbitrary length where the only constraint is that the size of the whole record must be smaller than the page size. Physically, each record is stored in exactly one page, which in turn is stored as a single object in S3. Logically, each record is part of a *collection* (e.g., a table). The record manager provides functions to create new objects, read objects, update objects, and scan collections. The *page manager* component implements a buffer pool for S3 pages. It supports reading pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated. All these functionalities are implemented in a straightforward way just as in any standard database system. Furthermore, the page manager implements the commit

* http://www.eucalyptus.com/.

† https://developers.google.com/cloud-sql/.

and abort methods where it is assumed that the write set of a transaction (i.e., the set of updated and newly created pages) fits into the clients main memory or secondary storage (flash or disk). If an application commits, all the updates are propagated to S3 and all the affected pages are marked as unmodified in the clients buffer pool. Moreover, they implemented standard B-tree indexes on top of the page manager and basic redo log records. On the other hand, there are many database-specific issues that have not yet been addressed by this work. For example, DB-style strict consistency and transactions mechanisms are not provided. Furthermore, query processing techniques (e.g., join algorithms and query optimization techniques) and traditional database functionalities such as bulkload a database, create indexes, and drop a whole collection, still need to be devised.

SimpleDB is another Amazon service that is designed for providing structured data storage in the cloud and backed by clusters of Amazon-managed database servers. It is a highly available and flexible nonrelational data store that offloads the work of database administration. Storing data in SimpleDB does not require any predefined schema information. Developers simply store and query data items via web services requests and Amazon SimpleDB does the rest. There is no rule that forces every data item (data record) to have the same fields. However, the lack of schema also means that there are no data types, as all data values are treated as variable length character data. Hence, the drawbacks of a schema-less data storage also include the lack of automatic integrity checking in the database (no foreign keys) and an increased burden on the application to handle formatting and type conversions. Following the AWS' pay-as-you-go pricing philosophy, SimpleDB has a pricing structure that includes charges for data storage, data transfer, and processor usage. There are no base fees and there are no minimums. Similar to most AWS services, SimpleDB provides a simple API interface, which follows the rules and the principles for both of REST and SOAP protocols where the user sends a message with a request to carry out a specific operation. The SimpleDB server completes the operations, unless there is an error, and responds with a success code and response data. The response data is an HTTP response packet, which has headers, storing metadata, and some payload, which is in XML format.

The top level abstract element of data storage in SimpleDB is the *domain*. A domain is roughly analogous to a database table where the user can create and delete domains as needed. There are no design or configuration options to create a domain. The only parameter you can set is the domain name. All the data stored in a SimpleDB domain takes the form of key-value attribute pairs. Each attribute pair is associated with an item that plays the role of a table row. The attribute name is similar to a database column name. However different items (rows) can contain different attribute names, which give you the freedom to store different attributes in some items without changing the layout of other items that do not have the same attributes. This flexibility allows the painless addition of new data fields in the most common situations of schema changing or schema evolution. In addition, it is possible for each attribute to have not just one value (multivalued attributes) but an array of values. In this case, all the user needs to do is add another attribute to an item and use the same attribute name but with a different value. Each value is automatically indexed as it is added. However, there are no explicit indexes to maintain. Therefore, the user has

no index maintenance work of any kind to do. On the other side, the user does not have any direct control over the created indices. SimpleDB provides a small group of API calls that enables the core functionality for building client applications such as *CreateDomain*, *DeleteDomain*, *PutAttributes*, *DeleteAttributes*, and *GetAttributes.* The SimpleDB API also provides a query language that is similar to the SQL *Select* statement. Hence, this query language makes SimpleDB Selects very familiar to the typical database user that ensures a gentle learning curve. However, it should be noted that the language supports issuing queries only over the scope of a single domain (no joins, multidomain, or subselect queries).

SimpleDB is implemented with complex replication and failover mechanisms behind the scenes. Therefore, it can provide a high availability guarantee with the stored data replicated to different locations automatically. Hence, a user does not need to make any extra effort or become an expert on high availability or the details of replication techniques to achieve the high availability goal. SimpleDB supports two options for each user read request: eventual consistency or strong consistency. In general, using the option of a consistent read eliminates the consistency window for the request. The results of a consistent read are guaranteed to return the most up-to-date values. In most cases, a consistent read is no slower than an eventually consistent read. However, it is possible for consistent read requests to show higher latency and lower bandwidth on some occasions (e.g., high workloads). SimpleDB does not offer any guarantees about the eventual consistency window but it is frequently less than one second. There are quite a few limitations that a user needs to consider while using the simpleDB service such as the maximum storage size per domain is 10 GB, the maximum attribute values per domain is 1 billion, the maximum attribute values per item is 256, the maximum length of item name, attribute name, or value is 1024 bytes, the maximum query execution time is 5 seconds, the max query result is 2500, and the maximum query response size is 1 MB.

Amazon Relational Database Service (RDS) is another Amazon service that gives access to the full capabilities of the familiar MySQL, Oracle, and SQL Server relational database systems. Hence, the code, applications, and tools that are already designed on existing databases of these system can work seamlessly with Amazon RDS. Once the database instance is running, Amazon RDS can automate common administrative tasks, such as performing backups or patching the database software. Amazon RDS can also provide data replication synchronization and automatic failover management services.

9.4.3 Microsoft SQL Azure

Microsoft has recently released the Microsoft SQL Azure Database system,* which has been announced as a cloud-based relational database service that has been built on Microsoft SQL Server technologies [12]. It provides a highly available, scalable, multi-tenant database service hosted by Microsoft in the cloud. So, applications can create, access, and manipulate tables, views, indexes, referential constraints, roles, stored procedures, triggers, and functions. It can execute complex queries, joins

* http://www.microsoft.com/windowsazure/sqlazure/.

across multiple tables, and supports aggregation and full-text queries. It also supports Transact-SQL (T-SQL), native ODBC, and ADO.NET data access.* In particular, the SQL Azure service can be seen as running an instance of SQL server in a cloud hosted server, which is automatically managed by Microsoft instead of running on-premise managed server.

In SQL Azure, a logical database is called a *table group*, which can be keyless or keyed. A keyless table group is an ordinary SQL server database where there are no restrictions on the choices of keys for the tables. On the other hand, if a table group is keyed, then all of its tables must have a common column called the *partitioning key*, which does not need not to be a unique key for each relation. A *row group* is a set of all rows in a table group that have the same partition key value. SQL Azure requires that each transaction executes on one table group. If the table group is keyed, then the transaction can read and write rows of only one row group. Based on these principles, there are two options for building transaction application that can scale out using SQL Azure. The first option is to store the data in multiple groups where each table group can fit comfortably on a single machine. In this scenario, the application takes the responsibility for scaling out by partitioning the data into separate table groups. The second option is to design the database as keyed table group so that the SQL Azure can perform the scale out process automatically.

In SQL Azure, the *consistency unit* of an object is the set of data that can be read and written by an ACID transaction. Therefore, the consistency unit of a keyed table group is the row group while the consistency unit of a keyless table group is the whole table group. Each replica of a consistency unit is always fully contained in a single instance of SQL server running one machine. Hence, using the two-phase commit protocol is never required. A query can execute on multiple partitions of a keyed table group with an isolation level of read-committed. Thus, data that the query read from different partitions may reflect the execution of different transactions. Transactionally consistent reads beyond a consistency unit are not supported.

At the physical level, a keyed table group is split into partitions based on ranges of its partitioning key. The ranges must cover all values of the partitioning key and must not overlap. This ensures that each row group resides in exactly one partition and hence that each row of a table has a well-defined home partition. Partitions are replicated for high availability. Therefore, a partition is considered to be the failover unit. Each replica is stored on one server. Each row group is wholly contained in one replica of each partition that is scattered across servers such that no two copies reside in the same *failure domain*. The transaction commitment protocol requires that only a quorum of the replicas be up. A Paxos-like consensus algorithm is used to maintain a set of replicas to deal with replica failures and recoveries. Dynamic quorums are used to improve availability in the face of multiple failures. In particular, for each partition, at each point in time one replica is designated to be the primary. A transaction executes using the primary replica of the partition that contains its row group and thus is nondistributed. The primary replica processes all queries, updates, and data definition language operations. The primary replica is also

* http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx.

responsible for shipping the updates and data definition language operations to the secondary replicas.

Since some partitions may experience higher load than others, the simple technique of balancing the number of primary and secondary partitions per node might not balance the loads. The system can rebalance dynamically using the failover mechanism to tell a secondary on a lightly loaded server to become the primary, by either demoting the former primary to secondary, or moving the former primary to another server. A keyed table group can be partitioned dynamically. If a partition exceeds the maximum allowable partition size (either in bytes or the amount of operational load it receives), it is split into two partitions. In general, the size of each hosted SQL Azure database cannot exceed the limit of 50 GB.

9.5 WEB SCALE DATA MANAGEMENT: TRADEOFFS

An important issue in designing large-scale data management applications is to avoid the mistake of trying to be "*everything for everyone*." As with many types of computer systems, no one system can be best for all workloads and different systems make different tradeoffs to optimize for different applications. Therefore, the most challenging aspects in these application is to identify the most important features of the target application domain and to decide about the various design tradeoffs, which immediately lead to performance tradeoffs. To tackle this problem, Jim Gray came up with the heuristic rule of "*20 queries*" [38]. The main idea of this heuristic is that on each project, we need to identify the 20 most important questions the user wanted the data system to answer. He argued that five questions are not enough to see a broader pattern, and a hundred questions would result in a shortage of focus.

In general, it is hard to maintain ACID guarantees in the face of data replication over large geographic distances. The CAP theorem [15,34] shows that a shared-data system can only choose at most two out of three properties: *Consistency* (all records are the same in all replicas), *Availability* (a replica failure does not prevent the system from continuing to operate), and *tolerance to Partitions* (the system still functions when distributed replicas cannot talk to each other). When data is replicated over a wide area, this essentially leaves just consistency and availability for a system to choose between. Thus, the C (consistency) part of ACID is typically compromised to yield reasonable system availability [2]. Therefore, most of the cloud data management overcomes the difficulties of distributed replication by relaxing the ACID guarantees of the system. In particular, they implement various forms of weaker consistency models (e.g., eventual consistency, timeline consistency, session consistency [60]) so that all replicas do not have to agree on the same value of a data item at every moment of time. Hence, NoSQL systems can be classified based on their support of the properties of the CAP theorem into three categories:

- *CA systems*: Consistent and highly available, but not partition-tolerant
- *CP systems*: Consistent and partition-tolerant, but not highly available
- *AP systems*: Highly available and partition-tolerant, but not consistent

In principle, choosing the adequate NoSQL system (from the very wide available spectrum of choices) with design decisions that best fit with the requirements of a software application is not a trivial task and requires careful consideration. Table 9.1 provides an overview of different design decision for sample NoSQL systems.

In practice, transactional data management applications (e.g., banking, stock trading, supply chain management) that rely on the ACID guarantees that databases provide, tend to be fairly write-intensive or require microsecond precision and are less obvious candidates for the cloud environment until the cost and latency of wide-area data transfer decreases. Cooper et al. [26] discussed the tradeoffs facing cloud data management systems as follows:

- *Read performance vs. write performance*: Log-structured systems that only store update deltas can be very inefficient for reads if the data is modified over time. On the other hand, writing the complete record to the log on each update avoids the cost of reconstruction at read time but there is a correspondingly higher cost on update. Unless all data fits in memory, random I/O to the disk is needed to serve reads (e.g., as opposed to scans). However, for write operations much higher throughput can be achieved by appending all updates to a sequential disk-based log.
- *Latency vs. durability*: Writes may be synched to disk before the system returns success to the user or they may be stored in memory at write time and synched later. The advantages of the latter approach are that avoiding disk access greatly improves write latency, and potentially improves

TABLE 9.1
Design Decisions of Various Web-Scale Data Management Systems

System	Data Model	Query Interface	Consistency	CAP Options	License
Bigtable	Column Families	Low-Level API	Strict	CP	Internal at Google
Google AppEng	Column Families	Python API-GQL	Strict	CP	Commercial
PNUTS	Key-Value Store	Low-Level API	Multiple	AP	Internal at Yahoo
Dynamo	Key-Value Store	Low-Level API	Eventual	AP	Internal at Amazon
S3	Large Objects Store	Low-Level API	Eventual	AP	Commercial
SimpleDB	Key-Value Store	Low-Level API	Multiple	AP	Commercial
RDS	Relational Store	SQL	Strict	CA	Commercial
SQL Azure	Relational Store	SQL	Strict	CA	Commercial
Cassandra	Column Families	Low-Level API	Tunable	AP	Open source—Apache
Hypertable	Multidimensional Table	Low-Level API, HQL	Eventual	AP	Open source—GNU
CouchDB	Document-Oriented Store	Low-Level API	Eventual	AP	Open source—Apache

throughput The disadvantage is the greater risk of data loss if a server crashes and loses unsynched updates.

- *Synchronous vs. asynchronous replication*: Synchronous replication ensures all copies are up-to-date but potentially incurs high latency on updates. Furthermore, availability may be impacted if synchronously replicated updates cannot complete while some replicas are offline. Asynchronous replication avoids high write latency but allows replicas to be stale. Furthermore, data loss may occur if an update is lost due to failure before it can be replicated.
- *Data partitioning*: Systems may be strictly row-based or allow for column storage. Row-based storage supports efficient access to an entire record and is ideal if we typically access a few records in their entirety. Column-based storage is more efficient for accessing a subset of the columns, particularly when multiple records are accessed.

Florescu and Kossmann [32] argued that in a cloud environment, the main metric that needs to be optimized is the cost as measured in dollars. Therefore, the big challenge of data management applications is no longer on how fast a database workload can be executed or whether a particular throughput can be achieved; instead, the challenge is how many machines are necessary to meet the performance requirements of a particular workload. This argument fits well with a rule-of-thumb calculation that has been proposed by Jim Gray regarding the opportunity costs of distributed computing on the Internet as opposed to local computations [35]. Gray reasons that except for highly processing-intensive applications outsourcing computing tasks into a distributed environment does not pay off because network traffic fees outnumber savings in processing power. In principle, calculating the tradeoff between basic computing services can be useful to get a general idea of the economies involved. This method can easily be applied to the pricing schemes of cloud computing providers (e.g., Amazon, Google). Florescu and Kossmann [32] have also argued in the new large-scale web applications, the requirement to provide 100% read and write availability for all users has overshadowed the importance of the ACID paradigm as the gold standard for data consistency. In these applications, no user is ever allowed to be blocked. Hence, consistency has turned to be an optimization goal in modern data management systems to minimize the cost of resolving inconsistencies and not a constraint as in traditional database systems. Therefore, it is better to design a system that it deals with resolving inconsistencies rather than having a system that prevents inconsistencies under all circumstances.

Kossmann et al. [41] conducted an end-to-end experimental evaluation for the performance and cost of running enterprise web applications with OLTP workloads on alternative cloud services (e.g., RDS, SimpleDB, S3, Google AppEngine, Azure). The results of the experiments showed that the alternative services varied greatly both in cost and performance. Most services had significant scalability issues. They confirmed the observation that public clouds lack of support for uploading large data volumes. It was difficult for them to upload 1 TB or more of raw data through the APIs provided by the providers. With regard to cost, they concluded that Google

seems to be more interested in small applications with light workloads whereas Azure is currently the most affordable service for medium to large services.

With the goal of facilitating performance comparisons of the tradeoffs cloud data management systems, the Yahoo! Cloud Serving Benchmarks, YCSB* [26] and YCSB++† [52], have been presented as frameworks and core set of benchmarks for NoSQL systems. The benchmarking tools have been made available via open-source to allow extensible development of additional cloud benchmark suites that represent different classes of applications and to facilitate the evaluation of different cloud data management systems.

9.6 CHALLENGES OF THE NEW WAVE OF NoSQL SYSTEMS

In this section, we shed the lights on a set of novel research challenges, that have been introduced by the new wave of NoSQL systems that need to be addressed to ensure that the vision of designing and implementing successful scalable data management solutions can be achieved.

9.6.1 True Elasticity

A common characteristic of internet-scale applications and services is that they can be used by large numbers of end users and highly variable load spikes in the demand for services that can occur depending on the day and the time of year and the popularity of the application. In addition, the workload characteristic could vary significantly from one application type to another where possible fluctuations on the workload characteristics that could be of several orders of magnitude on the same business day may also occur [13]. In principle, elasticity and horizontal scalability are considered to be of the most important features that are provided by NoSQL systems [59]. In practice, both of the commercial NoSQL offerings (e.g., Amazon SimpleDB) and commercial DaaS offerings (e.g., Amazon RDS, Microsoft SQL Azure) do not provide their users with any flexibility to dynamically increase or decrease the allocated computing resources of their applications. While NoSQL offerings claim to provide elastic services of their tenants, they do not provide any guarantee that their provider-side elasticity management will provide scalable performance with increasing workloads [10]. Moreover, commercial DaaS pricing models require their users to predetermine the computing capacity that will be allocated to their database instance as they provide standard packages of computing resources (e.g., *Micro, Small, Large*, and *Extra Large* DB Instances). In practice, predicting the workload behavior (e.g., arrival pattern, I/O behavior, service time distribution) and consequently accurate planning of the computing resource requirements with consideration of their monetary costs are very challenging tasks. Therefore, the user might still tend to overprovide the allocated computing resources for the database tier of their application to ensure satisfactory performance for their workloads. As a

* http://wiki.github.com/brianfrankcooper/YCSB/.
† http://www.pdl.cmu.edu/ycsb++/index.shtml.

result of this, the software application is unable to fully utilize the elastic feature of the cloud environment.

Xiong et al. [66] have presented a provider-centric approach for intelligently managing the computing resources in a shared multi-tenant database system at the virtual machine level. The proposed approach consists of two main components:

1. The system modeling module that uses machine learning techniques to learn a model that describes the potential profit margins for each tenant under different resource allocations. The learned model considers many factors of the environment such as SLA cost, client workload, infrastructure cost, and action cost.
2. The resource allocation decision module dynamically adjusts the resource allocations, based on the information of the learned model, of the different tenants to achieve the optimum profits.

Tatemura et al. [61] proposed a declarative approach for achieving elastic OLTP workloads. The approach is based on defining the following two main components:

1. The transaction classes required for the application.
2. The actual workload with references to the transaction classes.

Using this information, a formal model can be defined to analyze elasticity of the workload with transaction classes specified. In general, we believe that there is a lack of flexible and powerful consumer-centric elasticity mechanisms that enable software application to have more control on allocating the computing resources for the database tier of their applications over the application running time and make the best use of the elasticity feature of the cloud computing environments. More attention from the research community is required to address these issues in future work.

9.6.2 Data Replication and Consistency Management

In general, stateless services are easy to scale since any new *replicas* of these services can operate completely independently of other instances. In contrast, scaling stateful services, such as a *database system*, needs to guarantee a consistent view of the system for users of the service. However, the cost of maintaining several database replicas that are always strongly consistent is very high. As we have previously described, according to the *CAP* theorem, most of the NoSQL systems overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system and supporting various forms of weaker consistency models (e.g., eventual consistency [63]). In practice, a common feature of the *NoSQL* and *DaaS* cloud offerings is the creation and management of multiple replicas (usually 3) of the stored data while a replication architecture is running behind-the-scenes to enable automatic failover management and ensure high availability of the service. In general, replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two situations is mainly reflected by the

higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication [19].

Several studies have been presented as an attempt to quantify the consistency guarantees of cloud storage services. Wada et al. [64] presented an approach for measuring time-based staleness by writing timestamps to a key from one client, reading the same key, and computing the difference between the reader's local time and the timestamp read. Bermbach and Tai [11] have tried to address a side of these limitations by extending original the experiments of [64] using a number of readers that are geographically distributed. They measure the consistency window by calculating the difference between the latest read timestamp of version n and the write timestamp of version n+1. Their experiments with Amazon S3 showed that the system frequently violates monotonic read consistency. Anderson et al. [4] presented an offline algorithm that analyzes the trace of interactions between the client machines and the underlying key-value store and reports how many violations for consistent reads are there in the trace. This approach is useful for checking the safety of running operations and detecting any violation on the semantics of the executed operations. However, it is not useful for any system that requires online monitoring for their data staleness or consistency grantees. Zellag and Kemme [67] have proposed an approach for real-time detection of consistency anomalies for arbitrary cloud applications accessing various types of cloud datastores in transactional or nontransactional contexts. In particular, the approach builds the dependency graph during the execution of a cloud application and detect cycles in the graph at the application layer and independently of the underlying datastore. Bailis et al. [8] presented an approach that provides expected bounds on staleness by predicting the behavior of eventually consistent quorum-replicated data stores using Monte Carlo simulations and an abstract model of the storage system including details such as the distribution of latencies for network links.

Kraska et al. [42] have argued that finding the right balance among cost, consistency, and availability is not a trivial task. High consistency implies high cost per transaction and, in some situations, reduced availability but avoids penalty costs. Low consistency leads to lower costs per operation but might result in higher penalty costs. Hence, they presented a mechanism that not only allows designers to define the consistency guarantees on the data instead at the transaction level, but also allows them to automatically switch consistency guarantees at runtime. They described a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible (i.e., the penalty cost is low) and raise them when it matters (i.e., the penalty costs would be too high). The adaptation is driven by a cost model and different strategies that dictate how the system should behave. In particular, they divide the data items into three categories (A, B, C) and treat each category differently depending on the consistency level provided. The A category represents data items for which we need to ensure strong consistency guarantees as any consistency violation would result in large penalty costs, the C category represents data items that can be treated using session consistency as temporary inconsistency is acceptable, while the B category comprises all the data items where the consistency requirements vary over time depending on the actual availability of an item. Therefore, the data of this category is handled with either strong or session

consistency depending on a statistical-based policy for decision making. Keeton et al. [24] have proposed a similar approach in a system called *LazyBase* that allows users to trade off query performance and result freshness. LazyBase breaks up metadata processing into a pipeline of ingestion, transformation, and query stages that can be parallelized to improve performance and efficiency. By breaking up the processing, LazyBase can independently determine how to schedule each stage for a given set of metadata, thus providing more flexibility than existing monolithic solutions. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals and to utilize resources efficiently.

In general, the simplicity of key-value stores comes at a price when higher levels of consistency are required. In these cases, application programmers need to spend extra time and exert extra effort to handle the requirements of their applications with no guarantee that all corner cases are handled, which consequently might result in an error-prone application. In practice, data replication across different data centers is expensive. Inter-datacenter communication is prone to variation in round-trip times (RTTs) and loss of packets. For example, RTTs are in the order of hundreds of milliseconds. Such large RTTs cause the communication overhead that dominates the commit latencies observed by users. Therefore, systems often sacrifice strong consistency guarantees to maintain acceptable response times. Hence, many solutions rely on asynchronous replication mechanism and weaker consistency guarantees. Some systems have been recently proposed to tackle these challenges. For example, *Google Megastore* [9] has been presented as a scalable and highly available datastore that is designed to meet the storage requirements of large-scale interactive Internet services. It relies on the *Paxos* protocol [20], a proven optimal fault-tolerant consensus algorithm with no requirement for a distinguished master, for achieving synchronous wide area replication. Megastore's replication mechanism provides a single, consistent view of the data stored in its underlying database replicas. Megastore replication semantics is done on *entity group* basis, a priori grouping of data for fast operations, basis by synchronously replicating the group's transaction log to a quorum of replicas. In particular, it uses a write-ahead log replication mechanism over a group of symmetric peers where any node can initiate reads and writes. Each log append blocks on acknowledgments from a majority of replicas and replicas in the minority catch up as they are able. Kraska et al. [43] have proposed the *MDCC* (*M*ulti-*D*ata *C*enter *C*onsistency) commit protocol for providing strongly consistent guarantees at a cost that is comparable to eventually consistent protocols. In particular, in contrast to transactional consistency two-phase commit protocol (2PC), MDCC is designed to commit transactions in a single round-trip across data centers in the normal operational case. It also does not require a master node so that apply reads or updates from any node in any data center by ensuring that every commit has been received by a quorum of replicas. It does not also impose any database partitioning requirements. The MDCC commit protocol can be combined with different read guarantees where the default configuration is to guarantee read committed consistency without any lost updates. In principle, we believe that the problem of data replication and consistency management across different data centers in the cloud environment has, thus far, not attracted sufficient attention from the

research community, and it represents a rich direction of future research and investigation. Nawab et al. [49] presented *Message Futures*, a distributed multi-datacenter transaction management system that provides strong consistency guarantees while maintaining low commit latency. It achieves an average commit latency of around one RTT. In this approach, a transaction is committed when a commit condition on mutual information is met. The commit condition is designed to be true, at any point in time, for any single object in at least one datacenter. The protocol utilizes a replicated log (RLog) [65] to continuously share transactions and state information among datacenters, which allows a datacenter to commit transactions without initiating a new wide-area message exchange with other datacenters and improves the protocol's resilience to node and communication failures.

The *COPS* system (Clusters of Order-Preserving Servers) [48] has been designed to provide geo-replicated and distributed data stores that support complex online applications, such as social networks, which must provide an *always on* facility where operations always complete with low latency. In particular, it provides causal+ consistency where it executes all *put* and *get* operations in the local datacenter in a linearizable fashion, and it then replicates data across datacenters in a causal+ consistent order in the background. COPS achieves the causal+ consistency by tracking and explicitly checking that causal dependencies are satisfied before exposing writes in each cluster.

9.6.3 SLA Management

An SLA is a contract between a service provider and its customers. *Service level agreements* (SLAs) capture the agreed upon guarantees between a service provider and its customer. They define the characteristics of the provided service including service level objectives (SLOs) (e.g., maximum response times) and define penalties if these objectives are not met by the service provider. In practice, flexible and reliable management of SLA agreements is of paramount importance for both cloud service providers and consumers. For example, Amazon found that every 100 ms of latency costs them 1% in sales and Google found that an extra 500 ms in search page generation time dropped traffic by 20%. In addition, large enterprise web applications (e.g., eBay and Facebook) need to provide high assurances in terms of SLA metrics such as response times and service availability to their users. Without such assurances, service providers of these applications stand to lose their user base, and hence their revenues.

In general, SLA management is a common general problem for the different types of software systems that are hosted in cloud environments for different reasons such as the unpredictable and bursty workloads from various users in addition to the performance variability in the underlying cloud resources [26,55]. In practice, resource management and SLA guarantee falls into two layers: the *cloud service providers* and the *cloud consumers* (users of cloud services). In particular, the cloud service provider is responsible for the efficient utilization of the physical resources and guarantee their availability for their customers (cloud consumers). The cloud consumers are responsible for the efficient utilization of their allocated resources to satisfy the SLA of their customers (application end users) and achieve

their business goals. The state-of-the-art cloud databases do not allow the specification of SLA metrics at the application nor at the end-user level. In practice, cloud service providers guarantee only the availability (uptime guarantees), but not the performance, of their services [6,10,31]. In addition, sometimes the granularity of the uptime guarantees is also weak. For example, the uptime guarantees of Amazon EC2 is on a per data center basis where a data center is considered to be unavailable if a customer cannot access any of its instances or cannot launch replacement instances for a contiguous interval of five minutes. In practice, traditional cloud monitoring technologies (e.g., *Amazon CloudWatch*) focus on low-level computing resources (e.g., *CPU speed*, *CPU utilization*, *I/O disk speed*). In general, translating the SLO of software application to the thresholds of utilization for low-level computing resources is a very challenging task and is usually done in an ad hoc manner due to the complexity and dynamism inherent in the interaction between the different tiers and components of the system. Furthermore, cloud service providers do not automatically detect SLA violation and leave the burden of providing the violation proof on the customer [10].

In the multi-tenancy environment of DaaS, it is an important goal for DaaS providers to promise high performance to their tenants. However, this goal normally conflicts with another goal of minimizing the overall running servers and thus operating costs by tenant consolidation. In general, increasing the *degree* of multi-tenancy (number of tenants per server) is normally expected to decrease per-tenant-allocated resources and thus performance, but on the other hand, it also reduces the overall operating cost for the DaaS provider and vice versa. Therefore, it is necessary, but challenging for the DaaS providers to balance between the performance that they can deliver to their tenants and the data center's operating costs. Several provider-centric approaches have been proposed to tackle this challenge. Chi et al. [22] have proposed a cost-aware query scheduling algorithm, called *iCBS*, that takes the query costs derived from the SLAs between the service provider and its customers (in terms of response time) into account to make cost-aware scheduling decisions that aims to minimize the total expected cost. *SLA-tree* is another approach that have been proposed to efficiently support profit-oriented decision making of query scheduling. SLA-tree uses the information about the buffered queries that are waiting to be executed in addition to the SLA for each query that indicates the different profits for the query for varying query response times and provides support for the answering of certain profit-oriented *what if* type of questions. Lang et al. [46] presented a framework that takes as input the tenant workloads, their performance SLA, and the server hardware that is available to the DaaS provider, and produces server characterizing models that can be used to provide constraints into an optimization module. By solving this optimization problem, the framework provides a cost-effective hardware provisioning policy and a tenant scheduling policy on each hardware resource. The main limitation of this approach is that the input information of the tenant workloads is not always easy to specify and model accurately. *PIQL* (*P*erformance *I*nsightful *Q*uery *L*anguage) [5] is a declarative language that has been proposed with a SLA compliance prediction model. The PIQL query compiler uses static analysis to select only query plans where it can calculate the number of operations to be performed at every step in their execution. In particular, PIQL extends

SQL to allow developers to provide extra bounding information to the compiler. In contrast to traditional query optimizers, the objective of the query compiler is not to find the fastest plan but to avoid performance degradation. Thus, the compiler choose a potentially slower bounded plan over an unbounded plan that happens to be faster given the current database statistics. If the PIQL compiler cannot create a bounded plan for a query, it warns the developer and suggests possible ways to bound the computation.

In general, adequate SLA monitoring strategies and timely detection of SLA violations represent challenging research issues in the cloud computing environments. Salman [10] has suggested that it may be necessary, in the future, for cloud providers to offer performance-based SLAs for their services with a tiered pricing model, and charge a premium for guaranteed performance. While this could be one of the directions to solve this problem, we believe that it is a very challenging goal to delegate the management of the fine-granular SLA requirements of the consumer applications to the side of the cloud service provider due to the wide heterogeneity in the workload characteristics, details, and granularity of SLA requirements, and cost management objectives of the very large number of consumer applications (tenants) that can be running simultaneously in a cloud environment. Therefore, it becomes a significant issue for the cloud consumers to be able to monitor and adjust the deployment of their systems if they intend to offer viable SLAs to their customers (end users). It is an important requirement for cloud service providers to enable the cloud consumers with a set of facilities, tools and framework that ease their job of achieving this goal effectively.

9.6.4 Transaction Support

A transaction is a core concept in the data management world that represents a set of operations that are required to be executed *atomically* on a single consistent view of a database [36]. In general, the expertise gained from building distributed database systems by researchers and practitioners have shown that supporting distributed transactions hinder the ability of building scalable and available systems [51]. Therefore, to satisfy the scalability requirements of large-scale internet services, many systems have sacrificed the ability to support distributed transactions. For example, most of the NoSQL systems (e.g., Bigtable, Dynamo, SimpleDB) supports atomic access only at the granularity of single keys. This design choice allows these systems to horizontally partition the tables, without worrying about the need for distributed synchronization and transaction support. While many web applications can live with single key access patterns [21,30], many other applications (e.g., payment, auction services, online gaming, social networks, collaborative editing) would require atomicity guarantee on multikey accesses patterns. In practice, leaving the burden of ensuring transaction support to the application programmer normally leads to increased code complexity, slower application development, and low-performance client-side transaction management. Therefore, one of the main challenges of cloud-hosted database systems that has been considered is to support transactional guarantees for their applications without compromising the scalability property as one of the main advantages of the cloud environments.

The *G-Store* system [29] has been presented as a scalable data store that provides transactional multikey access guarantees over non-overlapping groups of keys using a key-value store. The main idea of GStore is the *Key Group* abstraction that defines a relationship between a group of keys and represents the granule for on-demand transactional access. This abstraction allows the Key Grouping protocol to collocate control for the keys in the group to allow efficient access to the group of keys. In particular, the Key Grouping protocol enables the transfer of ownership for all keys in a group to a single-node that then efficiently executes the operations on the Key Group. At any instance of time, each key can only belong to a single group and the Key Group abstraction does not define a relationship between two groups. Thus, groups are guaranteed to be independent of each other and the transactions on a group guarantee consistency only within the confines of a group. The Key Grouping protocol ensures that the ownership of the members of a group reside with a single node. Thus, the implementation of the transaction manager component does not require any distributed synchronization and is similar to the transaction manager of any single-node relational database management systems. The key difference is that in G-Store, transactions are limited to smaller logical entities (key groups). A similar approach has been followed by the *Google Megastore system* [9]. It implements a transactional record manager on top of the Bigtable data store [21] and provides transaction support across multiple data items where programmers have to manually link data items into hierarchical groups and each transaction can only access a single group. Megastore partitions the data into a collection of *entity groups*, a priori user-defined grouping of data for fast operations, where each group is independently and synchronously replicated over a wide area. In particular, Megastore tables are either entity group root tables or child tables. Each child table must declare a single distinguished foreign key referencing a root table. Thus, each child entity references a particular entity in its root table (called the root entity). An entity group consists of a root entity along with all entities in child tables that reference it. Entities within an entity group are mutated with single-phase ACID transactions (for which the commit record is replicated via Paxos). Operations across entity groups could rely on expensive two-phase commit operations but they could leverage the built-in Megastore's efficient asynchronous messaging to achieve these operations. Google's *Spanner* [27] has been presented as a scalable and globally distributed database that shards data across many sets of Paxos state machines in datacenters that are spread all over the world. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. It supports general-purpose transactions, and provides a SQL-based query language.

Deuteronomy [47] have presented a radically different approach toward scaling databases and supporting transactions in the cloud by *unbundling* the database into two components: (1) The *transactional component* (TC) that manages transactions and their concurrency control and undo/redo recovery but knows nothing about physical data location. (2) The *data component* (DC) that maintains a data cache and uses access methods to support a record-oriented interface with atomic operations but knows nothing about transactions. Applications submit requests to the TC, which uses a lock manager and a log manager to logically enforce

transactional concurrency control and recovery. The TC passes requests to the appropriate data component (DC). The DC, guaranteed by the TC to never receive conflicting concurrent operations, needs to only support atomic record operations, without concern for transaction properties that are already guaranteed by the TC. In this architecture, data can be stored anywhere (e.g., local disk, in the cloud) as the TC functionality in no way depends on where the data is located. The TC and DC can be deployed in a number of ways. Both can be located within the client, and that is helpful in providing fast transactional access to closely held data. The TC could be located with the client while the DC could be in the cloud, which is helpful in case a user would like to use its own subscription at a TC service or wants to perform transactions that involve manipulating data in multiple locations. Both TC and DC can be in the cloud, which is helpful if a cloud data storage provider would like to localize transaction services for some of its data to a TC component. There can be multiple DCs serviced by one TC, where transactions spanning multiple DCs are naturally supported because a TC does not depend on where data items are stored. Also, there can be multiple TCs, yet, a transaction is serviced by one specific TC.

The *Calvin system* [62] has been designed to run alongside a nontransactional storage system with the aim of transforming it into a shared-nothing (near-)linearly scalable database system that provides high availability and full ACID transactions. These transactions can potentially span multiple partitions spread across the shared-nothing cluster. Calvin accomplishes this goal by providing a layer above the storage system that handles the scheduling of distributed transactions, as well as replication and network communication in the system. The key technical feature of Calvin is that it relies on a deterministic locking mechanism that enables the elimination of distributed commit protocols. In particular, the essence of Calvin lies in separating the system into three separate layers of processing:

- *The sequencing layer* intercepts transactional inputs and places them into a global transactional input sequence, which represents the order of transactions to which all replicas will ensure serial equivalence during their execution.
- *The scheduling layer* orchestrates transaction execution using a deterministic locking scheme to guarantee equivalence to the serial order specified by the sequencing layer while allowing transactions to be executed concurrently by a pool of transaction execution threads.
- *The storage layer* handles all physical data layout. Calvin transactions access data using a simple CRUD interface. Therefore, any storage engine supporting a similar interface can be directly plugged into Calvin.

Each node in a Calvin deployment typically runs one partition of each layer. It supports horizontal scalability of the database and unconstrained ACID-compliant distributed transactions by supporting both asynchronous and Paxos-based synchronous replication, both within a single data center and across geographically separated data centers.

9.7 DISCUSSION AND CONCLUSIONS

For more than a quarter of a century, the relational database management systems (RDBMS) have been the dominant model for database management. They provide an extremely attractive interface for managing and accessing data and have proven to be wildly successful in many financial, business, and Internet applications. However, with the new trends of web-scale data management, they started to suffer from some serious limitations [28]:

- *Database systems are difficult to scale*. Most database systems have hard limits beyond which they do not easily scale. Once users reach these scalability limits, time consuming and expensive manual partitioning, data migration, and load balancing are the only recourse.
- *Database systems are difficult to configure and maintain*. Administrative costs can easily account for a significant fraction of the total cost of ownership of a database system. Furthermore, it is extremely difficult for untrained professionals to get good performance out of most commercial systems.
- *Diversification in available systems complicates its selection*. The rise of specialized database systems for specific markets (e.g., main memory systems for OLTP or column stores for OLAP) complicates system selection, especially for customers whose workloads do not neatly fall into one category.
- *Peak provisioning leads to unnecessary costs*. Web-scale workloads are often bursty in nature, and thus, provisioning for the peak often results in excess of resources during off-peak phases, and thus unnecessary costs.

Recently, the new wave of NoSQL systems have started to gain some mindshares as an alternative model for database management. In principle, some of the main advantages of NoSQL systems can be summarized as follows:

- *Elastic scaling*: For years, database administrators have relied on the *scale up* approach rather than the *scale out* approach. However, with the current increase in the transaction rates and high availability requirements, the economic advantages of the scaling out approach on commodity hardware has become very attractive. RDBMS might not scale out easily on commodity clusters but NoSQL systems are initially designed with the ability to expand transparently to take advantage of the addition of any new nodes.
- *Less administration*: Despite the many manageability improvements introduced by RDBMS vendors over the years, high-end RDBMS systems cannot be maintained without the assistance of expensive, highly trained DBAs. DBAs are intimately involved in the design, installation, and ongoing tuning of high-end RDBMS systems. On the contrary, NoSQL databases are generally designed from the ground up to require less management. For example, automatic repair and the simpler data model features should lead to lower administration and tuning requirements.

- *Better economics*: While RDBMS tends to rely on expensive proprietary servers and storage systems, NoSQL databases typically use clusters of cheap commodity servers to manage the exploding data and transaction volumes. Therefore, the cost per gigabyte or transactions per second for NoSQL can be many times less than the cost for RDBMS, which allows a NoSQL setup to store and process more data at a much lower price. Moreover, when an application uses data that is distributed across hundreds or even thousands of servers, simple economics points to the benefit of using no-cost server software as opposed to that of paying per-processor license fees. Once freed from license fees, an application can safely scale horizontally with complete avoidance of the capital expenses.
- *Flexible data models*: Even minor changes to the data model of a large production RDBMS have to be carefully managed and may necessitate downtime or reduced service levels. NoSQL databases have more relaxed (if any) data model restrictions. Therefore, application changes and database schema changes can be changed more softly.

These advantages have given NoSQL systems a lot of attractions. However, there are many obstacles that still need to be overcome before theses systems can appeal to mainstream enterprises such as:*

- *Programming model*: NoSQL databases offer few facilities for ad hoc query and analysis. Even a simple query requires significant programming expertise. Missing the support of declaratively expressing the important join operation has been always considered one of the main limitations of these systems.
- *Transaction support*: Transaction management is one of the powerful features of RDBMS. The current limited support (if any) of the transaction notion from NoSQL database systems is considered as a big obstacle toward their acceptance in implementing mission critical systems.
- *Maturity*: RDBMS systems are well known with their high stability and rich functionalities. In comparison, most NoSQL alternatives are in preproduction versions with many key features either being not stable enough or yet to be implemented. Therefore, enterprises are still approaching this new wave with extreme caution.
- *Support*: Enterprises look for the assurance that if a the system fails, they will be able to get timely and competent support. All RDBMS vendors go to great lengths to provide a high level of enterprise support. In contrast, most NoSQL systems are open-source projects. Although there are few firms offering support for each NoSQL database, these companies often are small start-ups without the global reach, support resources, or credibility of the key market players such as Oracle, Microsoft, or IBM.

* http://blogs.techrepublic.com.com/10things/?p=1772.

- *Expertise*: There are millions of developers throughout the world, and in every business segment, who are familiar with RDBMS concepts and programming. In contrast, almost every NoSQL developer is in a learning mode. This situation will be addressed naturally over time. However, currently, it is far easier to find experienced RDBMS programmers or administrators than a NoSQL expert.

Currently, there is a big debate between the NoSQL and RDBMS campuses that is centered on the right choice for implementing online transaction processing systems. RDBMS proponents think that the NoSQL camp has not spent sufficient time to understand the theoretical foundation of the transaction processing model. For example, the eventual consistency model is still not well defined, and different implementations may differ significantly with each other. This means figuring out all these inconsistent behaviors lands on the application developer's responsibilities and makes their life very much harder. On the other side, the NoSQL camp argues that this is actually a benefit because it gives the domain-specific optimization opportunities back to the application developers who are now no longer constrained by a one-size-fits-all model. However, they admit that making such optimization decisions requires a lot of experience and can be very error-prone and dangerous if the decisions are not made by experts.

In principle, we believe that it is not expected that the new wave of NoSQL data management systems will provide a complete replacement of the relational data management systems. Moreover, there will not be a single winner (one-size-fits-all) solution. However, it is more expected that different data management solutions will coexist at the same time for a single application (Figure 9.6). For example, we can imagine an application that uses different datastores for different purposes as follows:

- MySQL for low-volume, high-value data-like user profiles and billing information.
- A key value store (e.g., Hbase) for high-volume, low-value data-like hit counts and logs.

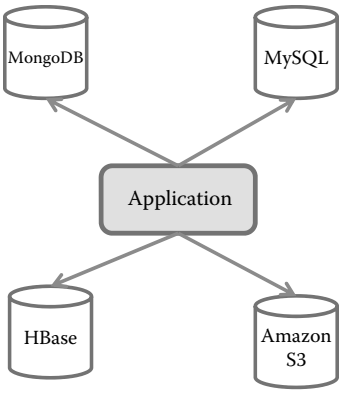

FIGURE 9.6 Coexistence of multiple data management solution in one application.

- Amazon S3 for user-uploaded assets like photos, sound files, and big binary files.
- MongoDB for storing the application documents (e.g., bills).

Finally, we believe that there is still a huge amount of required research and development efforts for improving the current state-of-the-art in tackling the current limitations in both of all campuses: NoSQL database systems, data management service providers, and traditional relational database management systems.

REFERENCES

1. Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.
2. Daniel J. Abadi. Data Management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.
3. Divyakant Agrawal, Amr El Abbadi, Fatih Emekçi, and Ahmed Metwally. Database management as a service: Challenges and opportunities. In *ICDE*, pages 1709–1716, 2009.
4. Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? In *HotDep*, 2010.
5. Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. PIQL: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.
6. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, et al. Above the clouds: A berkeley view of cloud computing, 2009.
7. Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The Potential Dangers of Causal Consistency and an Explicit Solution. In *SoCC*, 2012.
8. Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8), 2012.
9. Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.
10. Salman Abdul Baset. Cloud SLAs: Present and future. *Operating Systems Review*, 46(2):57–66, 2012.
11. David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, 2011.
12. Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kaki-vaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL server for cloud computing. In *ICDE*, pages 1255–1263, 2011.
13. Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, pages 241–252, 2010.
14. Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.
15. Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

16. Chris Bunch, Navraj Chohan, Chandra Krintz, Jovan Chohan, Jonathan Kupferman, Puneet Lakhina, Yiming Li, and Yoshihide Nomura. An evaluation of distributed datastores using the AppScale Cloud Platform. In *IEEE CLOUD*, pages 305–312, 2010.
17. Michael Burrows. The Chubby Lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, 2006.
18. Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
19. Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *SIGMOD Conference*, pages 739–752, 2008.
20. Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *PODC*, pages 398–407, 2007.
21. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
22. Yun Chi, Hyun Jin Moon, and Hakan Hacigümüs. iCBS: Incremental Costbased Scheduling under piecewise linear SLAs. *PVLDB*, 4(9):563–574, 2011.
23. Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Richard Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *CloudComp*, pages 57–70, 2009.
24. James Cipar, Gregory R. Ganger, Kimberly Keeton, Charles B. Morrey, Craig A. N. Soules, and Alistair C. Veitch. LazyBase: Trading freshness for performance in a scalable database. In *EuroSys*, pages 169–182, 2012.
25. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
26. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.
27. James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.
28. Carlo Curino, Evan Jones, Yang Zhang, Eugene Wu, and Sam Madde. Relational cloud: The case for a database service. In *CIDR*, 2011.
29. Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.
30. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
31. Dave Durkee. Why cloud computing will never be free. *Commun. ACM*, 53(5), 2010.
32. Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.
33. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, pages 29–43, 2003.
34. Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
35. Jim Gray. Distributed computing economics. Microsoft Research Technical Report MSRTR-2003-24, Microsoft Research, 2003.
36. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems, 1992.

37. Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *ICDE*, 2002.
38. Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The fourth paradigm: Data-intensive scientific discovery*. Microsoft Research, 2009.
39. David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, pages 654–663, 1997.
40. Bettina Kemme, Ricardo Jiménez-Peris, and Marta Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
41. Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD Conference*, pages 579–590, 2010.
42. Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.
43. Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.
44. Avinash Lakshman and Prashant Malik. Cassandra: Structured storage system on a p2p network. In *PODC*, page 5, 2009.
45. Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.
46. Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards multi-tenant performance SLOs. In *ICDE*, pages 702–713, 2012.
47. Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.
48. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.
49. Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*, 2013.
50. M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems, second edition*. Prentice-Hall, 1999.
51. M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 3rd edition, 2011.
52. Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In *SOCC*, 2011.
53. Dan Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008.
54. Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys and Tutorials*, 13(3):311–336, 2011.
55. Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1), 2010.
56. Adam Silberstein, Jianjun Chen, David Lomax, B. McMillan, M. Mortazavi, P. P. S. Narayan, Raghu Ramakrishnan, and Russell Sears. PNUTS in flight: Web-scale data serving at Yahoo. *IEEE Internet Computing*, 16(1):13–23, 2012.
57. Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.
58. Michael Stonebraker. One size fits all: An idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

59. Basem Suleiman, Sherif Sakr, Ross Jeffrey, and Anna Liu. On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Internet Services and Applications*, 3(2):173–193, 2012.
60. Andrew S. Tanenbaum and Maarten van Steen, editors. *Distributed systems: Principles and paradigms*. Prentice Hall, 2002.
61. Jun'ichi Tatemura, Oliver Po, and Hakan Hacigümüs. Microsharding: A declarative approach to support elastic OLTP workloads. *Operating Systems Review*, 46(1):4–11, 2012.
62. Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.
63. Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.
64. Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: The consumers' perspective. In *CIDR*, 2011.
65. Gene T. J. Wuu and Arthur J. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. *Operating Systems Review*, 20(1):57–66, 1986.
66. PengCheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüs. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, pages 87–98, 2011.
67. Kamal Zellag and Bettina Kemme. How consistent is your cloud application? In *SoCC*, 2012.