Large Scale and Big Data

Processing and Management

Edited by

Sherif Sakr

Cairo University, Egypt and
University of New South Wales, Australia

Mohamed Medhat Gaber

School of Computing Science and Digital Media
Robert Gordon University

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20140411

International Standard Book Number-13: 978-1-4665-8150-0 (Hardback)

Library of Congress Cataloging-in-Publication Data

Large scale and big data : processing and management / editors, Sherif Sakr, Mohamed Medhat Gaber.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4665-8150-0 (hardback)
1. Database management. 2. Big data. I. Sakr, Sherif, 1979- II. Gaber, Mohamed Medhat.

QA76.9.D3L3667 2014
005.7--dc23 2014002749

Contents

Preface

Information from multiple sources is growing at a staggering rate. The number of Internet users reached 2.27 billion in 2012. Every day, Twitter generates more than 12 TB of tweets, Facebook generates more than 25 TB of log data, and the New York Stock Exchange captures 1 TB of trade information. About 30 billion radio-frequency identification (RFID) tags are created every day. Add to this mix the data generated by the hundreds of millions of GPS devices sold every year, and the more than 30 million networked sensors currently in use (and growing at a rate faster than 30% per year). These data volumes are expected to double every two years over the next decade. On the other hand, many companies can generate up to petabytes of information in the course of a year: web pages, blogs, clickstreams, search indices, social media forums, instant messages, text messages, email, documents, consumer demographics, sensor data from active and passive systems, and more. By many estimates, as much as 80% of this data is semistructured or unstructured. Companies are always seeking to become more nimble in their operations and more innovative with their data analysis and decision-making processes, and they are realizing that time lost in these processes can lead to missed business opportunities. In principle, the core of the Big Data challenge is for companies to gain the ability to analyze and understand Internet-scale information just as easily as they can now analyze and understand smaller volumes of structured information. In particular, the characteristics of these overwhelming flows of data, which are produced at multiple sources are currently subsumed under the notion of Big Data with 3Vs (volume, velocity, and variety). *Volume* refers to the scale of data, from terabytes to zettabytes, *velocity* reflects streaming data and large-volume data movements, and *variety* refers to the complexity of data in many different structures, ranging from relational to logs to raw text.

Cloud computing technology is a relatively new technology that simplifies the time-consuming processes of hardware provisioning, hardware purchasing, and software deployment, therefore, it revolutionizes the way computational resources and services are commercialized and delivered to customers. In particular, it shifts the location of this infrastructure to the network to reduce the costs associated with the management of hardware and software resources. This means that the cloud represents the long-held dream of envisioning computing as a utility, a dream in which the economy of scale principles help to effectively drive down the cost of the computing infrastructure.

This book approaches the challenges associated with Big Data-processing techniques and tools on cloud computing environments from different but integrated perspectives; it connects the dots. The book is designed for studying various fundamental challenges of storing and processing Big Data. In addition, it discusses the applications of Big Data processing in various domains. In particular, the book is divided into three main sections. The first section discusses the basic concepts and tools of large-scale big-data processing and cloud computing. It also provides an overview of different programming models and cloud-based deployment models. The second section focuses on presenting the usage of advanced Big Data-processing techniques in different practical domains such as semantic web, graph processing, and stream processing. The third section further discusses advanced topics of Big Data processing such as consistency management, privacy, and security.

In a nutshell, the book provides a comprehensive summary from both of the research and the applied perspectives. It will provide the reader with a better understanding of how Big Data-processing techniques and tools can be effectively utilized in different application domains.

Sherif Sakr
Mohamed Medhat Gaber

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.

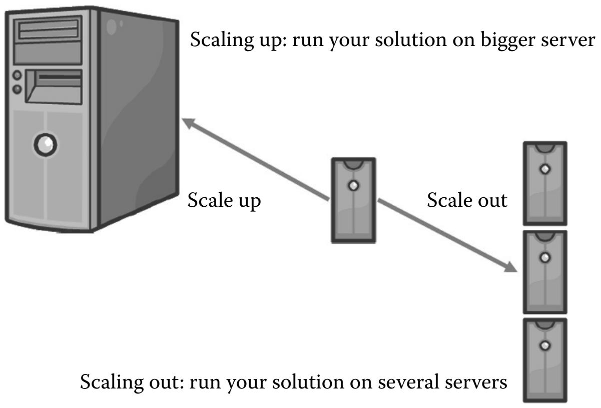
Scaling up: run your solution on bigger server
Scale up
Scale out
Scaling out: run your solution on several servers

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/.

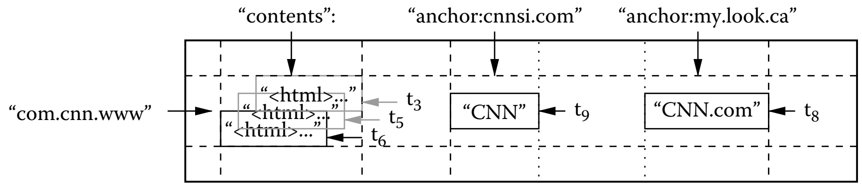

FIGURE 9.2 Sample Bigtable structure. (From F. Chang et al., *ACM Trans. Comput. Syst.*, 26, 2008.)

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.

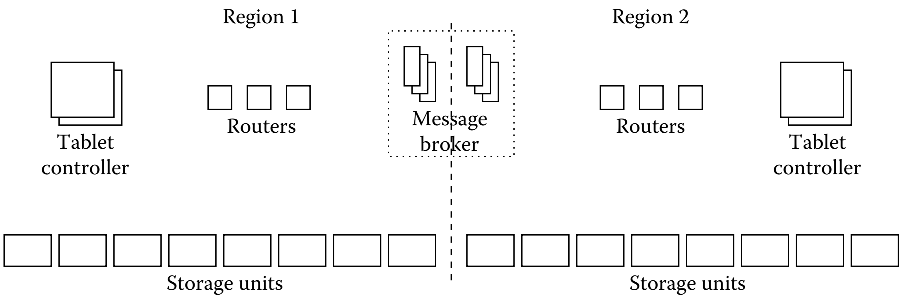

FIGURE 9.3 PNUTS system architecture. (From B. F. Cooper et al., *PVLDB*, 1, 1277–1288, 2008.)

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

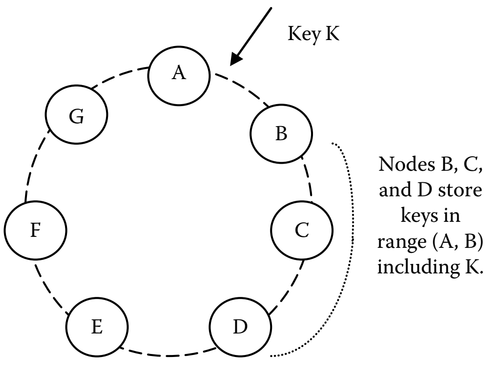

FIGURE 9.4 Partitioning and replication of keys in the Dynamo ring. (From G. DeCandia et al., Dynamo: Amazon's highly available key-value store, in *SOSP*, pp. 205–220, 2007.)

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/.