1

On the Spectrum of Web Scale Data Management

Liang Zhao  
University of New South Wales, Australia  
National ICT Australia  
Liang.Zhao@nicta.com.au

Sherif Sakr  
University of New South Wales, Australia  
National ICT Australia  
Sherif.Sakr@nicta.com.au

Anna Liu  
University of New South Wales, Australia  
National ICT Australia  
Anna.Liu@nicta.com.au

CONTENTS

1.1 Introduction .............................................................................. 4
1.2 NoSQL Key Systems ................................................................. 7
  1.2.1 Google: Bigtable ............................................................... 7
  1.2.2 Yahoo: PNUTS ................................................................. 8
  1.2.3 Amazon: Dynamo ............................................................... 10
1.3 NoSQL Open Source Projects .................................................. 12
1.4 Database-as-a-Service ............................................................ 14
  1.4.1 Google Datastore ............................................................. 14
  1.4.2 Amazon: S3 / SimpleDB / Amazon RDS ................................. 15
  1.4.3 Microsoft SQL Azure ....................................................... 16
  1.4.4 Challenges ....................................................................... 17
1.5 Web Scale Data Management: Trade-offs ................................. 18
1.6 Discussion and Conclusions ..................................................... 21

Over the past decade, rapidly growing Internet-based services have substantially redefined the way of data persistence and retrieval. Relational database management systems (RDBMS) have been considered as the one-size-fits-all solution for data persistence and retrieval for decades. However, ever-increasing needs for scalability and new application requirements have created high challenges for these systems. Therefore, recently, a new generation of low-cost, high-performance database software has emerged to challenge dom-
inance of RDBMS named as *NoSQL* (Not Only SQL). 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 the web scale data management. We discuss the advantages and the 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 in order to improve the current state-of-the-art.

1.1 Introduction

Over the past decade, rapidly growing Internet-based services have substantially redefined the way of data persistence and retrieval. By taking the recent advances in the web technology, content has been made easy for any user to provide and consume in any form. For example, building a personal web page (e.g. Google Sites\(^1\)), starting a blog (e.g. WordPress\(^2\), Blogger\(^3\), and LiveJournal\(^4\)) and making both publicly searchable for users all over the world have become a commodity. 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\(^5\) Flickr\(^6\), YouTube\(^7\), Zoho\(^8\), and Linkedin\(^9\) are currently leading this approach. Such applications are both *data-intensive* and very *interactive*. For example, the Facebook social network contains 500 million users\(^10\). Each user has an average 130 friendship relation. 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 professional has more than 80 million registered users. Therefore, it becomes an ultimate goal to make it easy for everybody 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

\(^1\)http://sites.google.com/
\(^2\)http://wordpress.org/
\(^3\)http://www.blogger.com/
\(^4\)http://www.livejournal.com/
\(^5\)http://www.facebook.com/
\(^6\)http://www.flickr.com/
\(^7\)http://www.youtube.com/
\(^8\)http://www.zoho.com/
\(^9\)http://www.linkedin.com/
after extensive research and development efforts and very successfully created a large market and solutions in different business domains. However, ever-increasing needs 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. Typically, three-tier approach, including the web server layer, the application server layer, and the data layer, is the most common architecture for building enterprise web applications. In practice, data partitioning [27] and data replication [22] are two well-known strategies to achieve the availability, scalability, and performance improvement goals in the distributed data management world. In particular, when the application load increases, there are two main options for achieving scalability at the database tier and make the application able to cope with more client requests (Figure 1.1):

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-per-use philosophy of cloud computing.

Recently, a new generation of database software with low-cost and high-
performance has emerged to challenge dominance of relational database management systems. An important reason for this movement, named as NoSQL (Not Only SQL), is that database requirements of web, enterprise, and cloud computing applications may vary because of different implementations (e.g., strong data consistency is not necessary for all applications). For example, scalability and high availability are essential requirements that can not be compromised for high-volume web sites (e.g., eBay, Amazon, Twitter, Facebook). For these applications, even the slightest breakdown can cause significant financial consequences and affect customer trust. The CAP theorem [6, 15] shows that a distributed database system can at most satisfy two out of three properties: Consistency, Availability and tolerance to Partitions. Therefore, most of these systems decide to compromise the strict consistency requirement. In particular, they apply a relaxed consistency policy called eventual consistency [29], which guarantees that if there is no new updates for a period of time, eventually all retrievals will return the last updated value. If no failures happen, based on factors such as communication delays, the load on the system and the number of replicas involved in the replication scheme, the maximum size of the inconsistency window can be determined [29]. In particular, these new NoSQL systems share a number of common design features such as horizontal scalability, simple interface, weak consistency model, distributed indexes and RAM and semi-structured data schema to achieve the following system goals:

- **Availability:** They must always be accessible even on the situations of having a network failure or a whole datacenter is went offline.

- **Scalability:** They must be able to serve very large databases under very high throughput 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 from overloaded servers to underloaded ones 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 need 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:** The number of nodes can be increased to hundreds or thousands in a scaling out environment, however, the homogeneous performance across nodes are hardly to guarantee due
to hardware performance degradation caused by part failures. Hence, the system should be designed to run in a heterogeneous environment and must take appropriate measures to prevent performance degrading due to parallel processing on distributed nodes [2].

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 1.2 describes the NoSQL systems, which are introduced and used internally in the big players: Google, Yahoo and Amazon respectively. Section 1.3 provides an overview of a set of open source projects, which have been designed following the main principles of the NoSQL systems. Section 1.4 discusses the notion of providing database management as a services and gives an overview of the main representative systems and their challenges. The web scale data management trade-offs and open research challenges are discussed in Section 1.5 before we conclude the chapter in Section 1.6.

1.2 NoSQL Key Systems

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

1.2.1 Google: Bigtable

Bigtable is used as a scalable, distributed storage system [8] in Google for a great number of Google products and projects such as: Google Docs\(^\text{11}\), Google Earth\(^\text{12}\), Google Finance\(^\text{13}\), Google search engine\(^\text{14}\), and Orkut\(^\text{15}\). These products can configure Bigtable for a variety of usages, supporting workloads from throughput-oriented job processing to latency-sensitive data serving, spanning servers from a handful number to thousands of commodity servers, and scaling data from a small amount to a size of petabytes.

The data model designed in Bigtable is not a relational data model, but a simple data model with dynamic control. Thus, users can change data layout and data format without being restricted by data schemas. In particular, Bigtable uses a sparse, multidimensional, sorted map to store data. Each cell in the map can be located by a row key, a column name, and a timestamp. A

\(^{11}\text{http://docs.google.com/}\)
\(^{12}\text{http://earth.google.com/}\)
\(^{13}\text{http://www.google.com/finance}\)
\(^{14}\text{http://www.google.com/}\)
\(^{15}\text{http://www.orkut.com/}\)
concrete example that reflects some of the main design decisions of Bigtable is the scenario of storing a collection of web pages. Figure 1.2 illustrates an example of this scenario where URLs are used as row keys and various web elements as column names. Values of web elements such as contents and anchors of the web page are in versioned cells under the timestamps when they were fetched.

The row keys are sorted in lexicographic order in Bigtable. Every single row key is an atomic unit of read or write. Usually, ranges of row keys, named tablets, can dynamically span multiple partitions for distribution and load balancing. Therefore, a table with multiple ranges can be processed in parallel on a number of servers. Each row can have an unlimited number of columns. Sets of them are grouped into column families for access control rights. Each cell is versioned and indexed by timestamps. The number of versions of a cell can be declared, so that only recent \( n \) versions are kept in decreasing timestamp order.

The Bigtable provides low-level APIs for following functions: creating, deleting, and changing tables and column families; updating configurations of cluster and column family metadata; adding, removing, and searching values from individual rows or a range of rows in a table. However, Bigtable does not support general transactions across row keys. Only atomic read-modify-write sequences on a single row, known as single-row transactions, are allowed.

On the physical level, the distributed Google File System (GFS) [14] is used to store Bigtable log and data files. The data is in Google SSTable file format, which offers an ordered, immutable, keys to values map for persistence. Bigtable relies on a distributed lock service called Chubby [7], which only runs if a majority of five composing replicas are accessible to each other. Among the five replicas, one of them is voted as master, initiatively serving all requests and balancing workloads across tablet servers. Each Bigtable has to be allocated to one master server and a number of tablet servers to be available. Hence, Bigtable can not work properly without Chubby, as it is necessary for, keeping master server running, as well as storing information of Bigtable, such as bootstrap locations, schemas, and access control lists.
1.2.2 Yahoo: PNUTS

PNUTS system (renamed later to Sherpa) is a scalable database system, storing tables of records with attributes to support web applications internally in Yahoo! [9]. The main goal of the system is serving data. Therefore, a list of functions is enhanced. Firstly, a simple relational model is supported, avoiding complex queries. Secondly, blob is validated as a main data type, storing arbitrary structures in a record, in addition to large binary objects like image or audio. Thirdly, the data schema of tables is enforced in a flexible way, allowing adding attributes at any time and keeping values of attributes empty in a record.

Figure 1.3 illustrates the system components of PNUTS. A region is a basic unit, which contains complement system components such as storage units, tablets, tablet controllers, and routers, as well as a full copy of tables. In practice, the PNUTS system consists of multiple geographically distributed regions. On the physical level, tables that are horizontal partitions of data tables are scattered across storage units in many servers. In each server, the number of tablets is variable, due to workloads balancing, which shifts tablets from overloaded servers to underloaded ones. Hence, hundreds to thousands of tablets can be achieved in a server. The router (Figure 1.3) can determine the location of a given record in two steps. It resolves which tablet has a given record in first place, by querying cached interval mapping, which defines tablet boundaries, and maintains mapping correlations of tablets and storage units. Then, it determines which storage unit owns a given tablet, by applying
mapping correlations to the given tablet. The *tablet controller* is the owner of interval mapping. It is also in charge of tablet management, such as moving a tablet across storage units for workload balancing or data recover, or splitting a large tablet.

As mentioned above, the system is designed for data serving that consists mainly of queries of single record or small groups of records. The query model is designed to keep simple in mind. Thus, it provides selection and projection of a single table, but join operation is too expensive to provide. It also allows updating and deleting operations only on primary key basis. Moreover, for reading multiple records, it supports a *multiget* operation to retrieving data in parallel.

PNUTS provides a consistency model that supports a variety of levels in the between of general serializability to eventual consistency [29]. The model is driven by the fact that web applications normally operate one record at a time whereas different records may be manipulated in different geographic areas. Thus, the model defines *per-record timeline* consistency as for a given record, all updates to the record are applied in the same order across replicas. Specifically, for each record, if one replica receives the most write for a specific record, the one among all replicas is elected as the master that maintains the update timeline of the record. The per-record timeline consistency model can be divided into various levels of consistency guarantees.

- **Read-any**: Read any version of the record. It is possible to return a stale version.
- **Read-critical(required version)**: Read a version of the record that is newer than, or the same as the *required version*.
- **Read-latest**: Read the latest version of the record that all writes are succeeded.
- **Write**: Write a record without reading value in advance. It may cause blind writes.
- **Test-and-set-write(required version)**: Write a record if and only if the current version is equivalent to the requirement version. It can be used as an incremental counter.

### 1.2.3 Amazon: Dynamo

As a high-volume web site, reliability is essential to Amazon because even the slightest breakdown can cause significant financial consequences and affect customer trust. Amazon Dynamo originates from Amazon, aiming to serve tens of millions customers with tens of thousands of servers geographically distributed over the world.
The Dynamo system [12] is a highly available and scalable distributed key-value based datastore implemented for internal Amazon applications. The design of Dynamo system is based on two concerns of using a relational database. On one hand, although the relational database can provide complex data schemas, in practice, many applications in Amazon only require simple primary key access. Thus, the query model of the Dynamo system is key-based single read and write operations. There is no operation that spans multiple data items. On the other hand, a relational database tends to be limited in scalability and availability according to common patterns, therefore, Dynamo system implements Dynamo ring to enhance replications.

In order to distribute workload across multiple hosts, Dynamo uses a variant of consistent hashing mechanism [21] to do partitioning. This mechanism defines a fixed circular space or ring first as the output range of a hash function. Then, a random value in the range of the space is assigned to each node, known as the "position" of the node on the ring. Hence, each data item is stored in a node that its position is clockwise closest to the data items position, which is determined by hashing the items key. Thus, each node is only in charge of the range of the ring from it to its previous node, while adding or removing a node on the ring have no impaction on other nodes except its neighbors.

In Dynamo system, each data item identified by a key $k$ is assigned to a coordinator and $(N - 1)$ clockwise successor nodes for replication where $N$ is a configurable parameter. The coordinator owns the data items falling in its range, and takes responsibility of the replication of them. As a result, each
node stores data items in the range of the ring from it to its $N^{th}$ predecessor. As illustrated in Figure 1.4, node $B$ owns a copy of the key $k$ locally, as well as replicates it at nodes $C$ and $D$. Node $D$ stores the keys within the ranges $(A, B]$, $(B, C]$, and $(C, D]$, and takes care of the keys that fall in the range of $(C, D]$.

1.3 NoSQL Open Source Projects

In practice, most of the NoSQL data management systems which are introduced by the key players (e.g. BigTable, Dynamo, PNUTS) are for their internal use and 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. These systems started to have a lot of interest from the research community. There are not much details have been published about the implementation most of these systems. In general, the NoSQL open source project 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.

- **Extensible record stores**: They provide variable-width tables (Column Families) that can be partitioned vertically and horizontally across multiple nodes.

- **Document stores**: Where the data model consists of objects with a variable number of attributes with a possibility of having nested objects.

Here, we give a brief introduction about some of these projects. For the full list, we refer the reader to the NoSQL database website\(^{16}\).

Cassandra\(^{17}\) is known as highly scalable, eventually consistent, distributed, structured key-value store [26, 25]. It is initially designed as an inbox storage service in Facebook. And then open sourced since 2008. One of its authors is also an author of Amazon’s Dynamo. Hence, Cassandra combines the distribution technology from Dynamo with the data model from Google Bigtable. This results in the system where comes with Dynamo’s eventual consistent and Bigtable’s ColumnFamily-based data model. The data model comes with four basic concepts. The basic unit of the data model is column including a name, a value and a timestamp. A column family groups multiple columns together, comparable with the table of a relational database. Column families can be composed into a keyspace, which can be considered a schema to

\(^{16}\)http://NoSQL-database.org/
\(^{17}\)http://cassandra.apache.org/
a relational database, typically, one keyspace per application. Super columns represent columns that themselves have subcolumns (e.g. Maps). Cassandra offers various levels of consistency models that are suitable for specific applications. In particular, for every read and write operation, one can choose the following consistency level: a) ONE: It ensures that at least one replica has been retrieved or committed to logs and memory before responding to the client. b) QUORUM: It ensures that a majority of replicas \((N/2 + 1)\) have been reported where \(N\) is the total number of replicas. c) ALL: It ensures that all \(N\) replicas have to be contacted. Moreover, for the write operation, two more levels are supported, namely a) ZERO: Nothing ensures in this level. The write operation is asynchronously executed in the background. b) ANY: It ensures that the data has been committed to at least one node. Starting from 0.7, two new quorum options are available that is LOCAL QUORUM and EACH QUORUM.

HBase\(^{18}\) is another project based on the ideas of BigTable system. It builds on top of the Hadoop Distributed File System (HDFS)\(^ {19}\) 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 is already considering it. However, the downside is that it becomes also constrained by the characteristics of HDFS, which 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 the corresponding server. The in-memory data storage is implemented using a distributed memory object caching system called Memcache\(^ {20}\) while the on-disk data storage is implemented as a HDFS file residing in Hadoop data node server.

HyperTable\(^ {21}\) project is designed to achieve a high performance, scalable, distributed storage and processing system for structured and unstructured data. The same as HBase, Hypertable also runs on top of HDFS that offers automatic data replication, data consistency and resiliency. In HyperTable, data model is represented as multi-dimensional tables. The system supports create, modify, and query data via low-level APIs or Hypertable Query Language (HQL). Data processing can be executed in parallel to increase the performance.

CouchDB\(^ {22}\) is a document-oriented database. A document object identified by a unique identity is the primary data unit, consisting of named fields and typed field values such as strings, numbers, dates, or even ordered lists and associative maps. Data query is access via RESTful HTTP API that offers read, update, add, and delete operations. The system is lockless and optimistic, and there is no partially edited documents saved in system. If two clients try to save the same document, an edit conflict error happens to one

---

\(^{18}\)http://hbase.apache.org/
\(^{19}\)http://hadoop.apache.org/hdfs/
\(^{20}\)http://memcached.org/
\(^{21}\)http://hypertable.org/
\(^{22}\)http://couchdb.apache.org/
client on updating. The system resolves the conflict by reopen the latest document version and reapplies all updates. The document update can either be all (succeeding entirely) or none (failing completely).

Many other variant projects are recently started to follow the NoSQL movement and support different types of data stores such as: key-value stores (e.g. Voldemort\(^{23}\), Dynomite\(^{24}\)), document stores (e.g. MongoDB\(^{25}\), Riak\(^{26}\)) and graph stores (e.g. Neo4j\(^{27}\), DEX\(^{28}\))

1.4 Database-as-a-Service

Database-as-a-service (Daas) is an emerging paradigm for data management in which a third party service provider hosts a database as a service [4, 18]. The service providers charge customers on pay-per-use basis, and in return for offering hardware and software, managing system and software upgrades, and maintaining administrative and maintenance tasks. Since the cost of an external database service are comparatively low, considering the promising reliable, scalable, and elastic data storage. It is attractive solutions for various purposes such as archive, development and test, and startup companies. 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.

1.4.1 Google Datastore

Google App Engine Datastore\(^{29}\) is not externally accessible, as it is the scalable schemaless object data storage sitting behind Google App Engine. The data object is called entities, composed of a unique identity and a number of properties where one property can hold a typed value or refer to another entities. A kind is a container of entities, analogous to the table in a relational database. However, entities are schemeless in the same kind where two entities can have different properties or even different types for the same properties.

Google App Engine datastore provides APIs in Python\(^{30}\) and Java\(^{31}\) versions. For Python interface, it includes a rich data modeling API and a SQL-like query language called Google Query Language\(^{32}\) (GQL). Figure 1.5 de-

---

\(^{23}\)http://project-voldemort.com/
\(^{24}\)http://wiki.github.com/cliffmoon/dynomite/dynomite-framework
\(^{25}\)http://www.mongodb.org/
\(^{26}\)http://wiki.basho.com/display/RIAK/Riak
\(^{27}\)http://neo4j.org/
\(^{28}\)http://www.dama.upc.edu/technology-transfer/dex
\(^{29}\)http://code.google.com/appengine/docs/python/datastore/
\(^{30}\)http://www.python.org/
\(^{31}\)http://www.java.com/
\(^{32}\)http://code.google.com/appengine/docs/python/datastore/gqlreference.html
On the Spectrum of Web Scale Data Management

![Basic GQL syntax](image)

SELECT [* | __key__] FROM <kind>
[WHERE <condition> [AND <condition> ...]]
[ORDER BY <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 1.5
Basic GQL syntax

picts the basic syntax of GQL. For Java interface, it supports two API standards for modeling and querying, namely Java Data Objects\(^{33}\) (JDO) and Java Persistence API\(^{34}\) (JPA). An entity can be retrieved with its identity or by querying its properties. A query can return 0 to maximum 1000 sorted-by-property-values results for the sake of memory and runtime limitations. In principle, join is not supported in the query.

Google App Engine datastore supports transaction. A transaction ensures that operations in a transaction succeed entirely or fail completely. A single operation of creating, updating or deleting an entity happens in a transaction implicitly. Meanwhile, a group of operations can be explicitly defined as a transaction. The datastore manages transactions in an optimistic manner.

The datastore replicates data to multiple locations. Among all replicas, one is selected as the primary to keep the view of the data consistent by replicating delta data to other locations. In the case of failures, the datastore can wait for the primary to become available, or continue accessing data from an alternative replica, depending on the selection of read policies: 1) *strong consistency* means reading from the primary. 2) *eventual consistency* [29] means reading from an alternate replica when the primary location is unavailable.

### 1.4.2 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. Each object is a container of bytes. It is identified by a URI. With the specified URI, clients are able to access via SOAP or REST-based interface remotely, for example, `get(uri)` returns an object and `put(uri, bytestream)` writes a new version of the object. Ideally, S3 can be considered as an online backup solution or for archiving large objects, which are not frequently updated.

---

\(^{33}\)[http://code.google.com/appengine/docs/java/datastore/jdo/]

\(^{34}\)[http://code.google.com/appengine/docs/java/datastore/jpa/]
Amazon has not revealed details on the implementation of Amazon S3 yet. However, Brantner et al. [5] have presented initial efforts of building web-based database applications on top of S3. They described various protocols in order to operate S3 in a manner of a relational database. In their system, the record manager component is designed to create, read, update and scan records where each record contains a key and payload data. The size of a record must be no larger than a page size, as a page is a container of records, and each page is physically stored in S3 as a single object. In addition to record manager, a buffer pool is also implemented in page manager component. It interacts with S3 like normal buffer pool in any standard database system: reading pages from S3, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated. While the page manager is mainly in charge of commit and abort transactions. Moreover, they implemented standard B-tree indexes on top of the page manager and basic redo log records. However, there are still many database-specific issues that have not been addressed yet, by this work, for example, DB-style strict consistency and transactions mechanisms. Furthermore, as addressed in the paper, more functionalities can be devised: query processing techniques (e.g. join algorithms and query optimization techniques) and traditional database functionalities (e.g. bulkload a database, create indexes, and drop a whole collection).

Similar to S3, Amazon has not published the details of its other two products: SimpleDB and RDS. Generally, SimpleDB is designed for running queries on structured data. In SimpleDB, data is organized into domains (i.e. tables) within which we can put data, get data or run queries. Each domains consist of items (i.e. records) which are described by pairs of attribute names and values. It is not necessary to pre-define all of the schema information as new attributes can be added to the stored dataset when needed. Thus, the approach is similar to that of a spreadsheet and does not follow the traditional relational model. SimpleDB provides a small group of API calls that enables the core functionality to build client applications such as: CreateDomain, DeleteDomain, PutAttributes, DeleteAttributes, GetAttributes and Select. The main focus of SimpleDB is fast reading. Therefore, query operations are designed to run on a single domain. SimpleDB keeps multiple copies of each domain where a successful write operation guarantees that all copies of the domain will durably persist. In particular, SimpleDB supports two read consistency options: eventually consistent read [29] and consistent read.

Amazon Relational Database Service (RDS) is a new service, which gives access to the full capabilities of a familiar MySQL database. Hence, the code, applications, and tools, which are already designed on existing MySQL databases 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 manage synchronizing data replication and automatic failover management.
1.4.3 Microsoft SQL Azure

Microsoft has recently released the Microsoft SQL Azure Database system\(^{35}\). Not much details has been published on the implementation this project. However, it is announced as a cloud-based relational database service, which has been built on Microsoft SQL Server technologies. So, applications can almost move whatever available operations in SQL Server to SQL Azure such as creating, accessing, and manipulating tables, views, indexes, roles, stored procedures, triggers, and functions. It can execute complex queries and joins across multiple tables. It also supports Transact-SQL (T-SQL), native ODBC and ADO.NET data access\(^{36}\). In particular, 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, the size of each hosted database can not exceed the limit of 50 GB.

1.4.4 Challenges

In general, the service level agreements (SLA) of the commercial DaaS products are focusing on providing their customers with high availability (99.99%) to the hosted databases. However, they are not providing any promises or guarantee on the performance and scalability aspects. In particular, for each hosted database, the cloud provider stores 3 replicas in the same data centre with the main guarantee that they have no single point of failure (e.g. physical server, network). However, these providers are currently not providing the management of geo-replica(s) that can be used for recovery in the case of a physical disaster to the hosting data centre. One of the hosted 3 replicas by the service providers is selected to be a primary copy which is used to serve all read and write requests while the other 2 replicas are used as a standby replicas (hot backups) which are only used to save the situation for any failure circumstances that may happen to the primary copy. These hot backups are not used for serving any read/write operations or load balancing purposes. Moreover, currently, these services are supporting very simple application-aware (non-transparent) data partitioning strategies. Such limited data partitioning and data replication strategies force the application (customer) to take care of additional responsibilities and challenges in order to achieve performance improvement and scalability goals for many condition and situation such as:

- **Achieving Performance Aspects of Defined SLAs**: In the Data centres, each physical server hosts a number of databases where each database is allocated a specific portion of the resources that server. Assuming that these allocated resources has the limit to serve a number of application requests \((L)\) according to a specific application-defined SLA performance requirement. It is the

\(^{35}\)http://www.microsoft.com/windowsazure/sqlazure/
responsibility of the application to achieve the same SLA requirements with any increasing workload that might exceed the limit \( L \) (e.g. volume spike). This problem can be solved by adding an additional replica for the hosted database and distributing the workload between the available replicas in a balanced way (scale Out). However, this action should be done in a transparent way for the application side. The same behavior should be achieved the situation where scaling down is required in order to reduce the cost where it could be possible to achieve the specified SLA requirements with less number of replicas.

- **Data Spike**: In the previous situation, replicating the whole database can deal with the volume spike situation in order to achieve the performance requirements of a specific SLA requirements. However, in the data spike situation (increasing volume to specific object or table), may require just replicating a specific shard (partition) of the database in order to tackle the problem in a more efficient, effective and economical way. Such replication of specific partition should be also done transparently to the application code.

- **Distributed Transactions**: Due to the size limit on a single database or performance requirements. It would be common to run transactions over multiple partitions (databases). Currently, these cloud database services support transactions execution in a single partition. There is no support for execution distributed transactions over different partitions even in the same data centre. It is the application responsibility to deal with such situations.

Another main challenge for the DaaS products is that the service provider needs to guarantee that the data is secure, not only being secure in results of queries, but also being secure to the data provider. Some research efforts have considered the problem of how to index and query encrypted data [3, 17, 18, 20]. However, querying encrypted data is known to be computationally expensive. Therefore, as an alternative, providing an efficient trust mechanism has emerged to be solved for turning data outsourcing a viable paradigm. Agrawal et al. [4] described a privacy preserving algorithms for data outsourcing. Instead of encryption, they distribute data to multiple data provider sites and information theoretically proven secret sharing algorithms as the basis for privacy preserving outsourcing. However, more research and development efforts are still required to find effective solution for the data security and data privacy issues in order to encourage more customers to rely on this new data management services.

1.5 Web Scale Data Management: Trade-offs

An important issue in designing large scale data management applications is to avoid the mistake of trying to be "everything for everyone". Because
Table 1.1 summarizes the design decisions of our surveyed systems. In general, it is difficult to guarantee ACID properties for replicated data over large geographic distances. The CAP theorem [6, 15] shows that a shared-data system can at most satisfy 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 networks, this essentially just leaves a system only one selection between consistency and availability. Thus, the C (consistency) part typically compromised to yield reasonable system availability [1]. Therefore, most of the cloud data management applications relax data consistency to overcome the difficulties of distributed replication. In particular, they implement various forms of weaker consistency models (e.g. eventual consistency, timeline consistency, session consistency [28]) so that all replicas do not have to agree on the same value of a data item at every moment of time. Therefore, transactional data management applications (e.g. banking, stock trading, supply chain management), which rely on the strict consistency of databases’ offering, tend to be fairly write-intensive or require microsecond precision are less obvious candidates for the cloud environment until the cost and latency
of wide-area data transfer decrease. Cooper et al. [10] discussed the tradeoffs facing cloud data management applications as follows:

- **Read performance versus write performance**: An update to a record can either attach the delta to the existing record, or completely overwrite the existing one. The former write is efficient, as it only costs the write only modified bytes. However, the former read is inefficient, which is contrary to the write, as for the former read, there is a cost of reconstruction of deltas.

- **Latency versus durability**: Synchronizing writes immediately to disk before responding success takes longer time than storing writes in memory and synchronizing later to disk. The latter approach avoids costly disk I/O operations to reduce write latency. However, the unsynchronized data could lose if system failures happen before the next synchronizing.

- **Synchronous versus asynchronous replication**: Synchronous replication keeps all replicas up to date during the time, but potentially incurs high latency on updates. Furthermore, availability of the system may be affected if synchronization is suspended due to offline of some replicas. Asynchronous replication avoids high write latency over networks but allows stale data. Moreover, data loss may occur if an updated replica goes offline before propagating data.

- **Data partitioning**: Data can be partitioned strictly on row basis or on column basis. Row-based partitioning allows efficient access to an entire record. Hence it is ideal for accessing 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.

Kraska et al. [24] have argued that finding the right balance between 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 based on the data at the transaction level but also allows 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.

Florescu and Kossmann [13] argued that in 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 what is the right number of machines to meet the performance requirements of a particular workload under an acceptable cost. Hence, performance requirements such as how fast a database workload can be executed or whether a particular throughput can be achieved is no longer the main metric any more. This argument fits well with a rule of thumb calculation which has been proposed by Jim Gray regarding the opportunity costs of distributed computing in the Internet as opposed to local computations [16]. Gray reasons that for outsourcing computing tasks, network traffic fees may outnumber savings in processing power. In principle, it is useful to involve economies into tradeoff calculation between basic computing services. This method can easily be applied to the pricing schemes of cloud computing providers (e.g Amazon, Google). Florescu and Kossmann [13] have also argued in the new large scale web applications, the requirement of providing fully read and write availability for all users has surpassed the importance of the ACID paradigm in data consistency. In this circumstance, blocking user is not ever allowed. Therefore, in order to minimize the cost of resolving inconsistencies, it is better to design a system that deals with resolving inconsistencies rather than having a system that prevents inconsistencies under all circumstances.

Kossmann et al. [23] conducted an end-to-end performance and cost evaluation on alternative cloud services (e.g. RDS, SimpleDB, S3, Google AppEngine, Azure) with OLTP workloads. The results of the experiments showed that the alternative services differed from each other greatly both in cost and performance. Most services had significant scalability issues. They confirmed the observation that public clouds lack of support to upload 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 trade-offs cloud data management systems, Cooper et al. [10] have presented the Yahoo! Cloud Serving Benchmark (YCSB) framework and a core set of benchmarks. The benchmark tool has been made available via open source\textsuperscript{37} in order to encourage expansion of cloud benchmark suites that represent different classes of applications, as well as include different cloud data management systems.

\textsuperscript{37}http://wiki.github.com/brianfrankcooper/YCSB/
1.6 Discussion and Conclusions

For more than thirty years, the relational database management systems (RDBMS) have been recognized as the dominant solution for data persistence requirements. In particular, they provide a simple but extremely powerful interface for storing and accessing data. In addition, they have shown to be wildly successful in many business domains such as: financial, business and Internet applications. However, with the new trends of web scale data management, they started to suffer from some serious limitations [11]:

- **Database systems are difficult to scale.** In practice, each database system have a maximum limit which they can not easily scale beyond it. When the application workloads hit this scalability limit, a set of time consuming and manually expensive data partitioning, data migration and load balancing tasks are required to tackle this challenge.

- **Database systems are difficult to configure and maintain.** In general, getting a good performance out of most commercial relational database systems requires highly experienced professionals. Therefore, administrative cost represents a significant fraction of the total cost of ownership of a database system.

- **Diversification in available systems complicates selection.** Recently, specialized database systems for specific types of applications have been entering the market (e.g. main memory systems for OLTP or column-stores for OLAP). Such situation makes the system selection process quite complex especially for customers with different application workloads that do not neatly class.

- **Peak provisioning leads to unneeded costs.** The workloads of large scale Web applications are often bursty and dynamic in nature. Thus, peak provisioning process is usually applied to deal with this challenge. Therefore, inefficient resources utilization during the off-peak times usually happens which consequently causes unneeded costs.

Recently, a new wave of NoSQL systems is 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, the **scale up** approach has been considered as the favorite approach to rely on rather than the **scale out** approach for achieving the scalability goal. However, with the continuous increase in the transaction rates and high availability requirements, the economic advantages of scaling out approach on commodity hardware becomes very attractive. In practice, it is not easy to scale out RDBMS on commodity clusters. Therefore, the NoSQL systems have considered the ability to expand transparently as one
of its main requirements in order to take advantage of the addition of any new nodes.

• **Less Administration**: Over the years, RDBMS vendors have introduced many manageability improvements. However, it is still very expensive to maintain high-end RDBMS without the assistance of expensive and highly trained database administrators. DBAs are intimately involved in the design, installation and ongoing tuning of high-end RDBMS systems. On the contrary, NoSQL database systems are designed from the beginning with the ability to be maintained with less expertise and effort.

• **Better Economics**: RDBMS tends to use expensive proprietary servers and storage systems. On the contrary, NoSQL systems tend to rely on clusters of cheap commodity servers on dealing with the increasing data and transaction rates. Therefore, the cost per gigabyte or transaction/second for NoSQL can be many times less than the cost for RDBMS which allows to store and process more data with a much lower price. Moreover, when an application uses data distributed across hundreds or even thousands of servers, simple economics points to using no-cost server software as opposed to 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 small changes to the schema of a large production relational database have to be carefully considered and may require downtime or degraded service levels. NoSQL databases have more relaxed (if any) data model restrictions. Therefore, any application change or database schema change can be more softly managed.

These advantages have given the NoSQL systems a lot of attractions. However, enterprises are still very cautious to rely on these systems because there are many limitations still need to be addressed such as:

• **Programming Model**: NoSQL databases offer limited support for ad-hoc querying and analysis operations. Therefore, significant programming expertise are usually required even for a simple query. In addition, 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 towards their acceptance in implementing mission critical systems.

• **Maturity**: RDBMS are well-know with their high stability and rich functionalities. In contrast, most NoSQL systems are open source projects or in pre-production stages where many key features are either not stable enough
or still under development. Therefore, enterprises are still very cautious to deal with this new wave of database systems.

- **Support**: Enterprises look for the assurance that if the system fails, they will be able to get timely and competent support. All RDBMS vendors have great experience in providing high level of enterprise support. In contrast, most NoSQL systems are open source projects. Although there are few firms offering support for NoSQL database systems, these companies are still small start-ups without the global reach, support resources or credibility of an Oracle, Microsoft or IBM.

- **Expertise**: There are millions of developers around the world who are familiar with RDBMS concepts and programming models in every business domain. On the contrary, almost every NoSQL developer is still in a learning mode. It is natural that this limitation will be addressed 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 which is centered around the right choice for implementing online transaction processing systems. RDBMS proponents think that the NoSQL camp has not spent enough time to understand the theoretical foundation of the transaction processing model. For example, the eventual consistency model is still not well-defined and there are different implementations which may significantly differ with each other. Therefore, it is the responsibility of the application developer to figure out all the inconsistency behavior that may arise which makes their task very much harder. On the other side, the NoSQL camp argues that this is actually a benefit as it provides the application developers with domain-specific optimization opportunities where they are no longer constrained by the one-size-fits-all model. However, they admit a lot of experience is required for making optimization decisions otherwise they can be very error-prone and dangerous decisions.

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 be no a single winner (one-size-fits-all) solution. However, it is more expected that different data management solutions will coexist in the same time for a single application (Figure 1.6). For example, we can imagine an application which uses different datastores for different purposes as follows:

- **MySQL** for high-value and low-volume data such as billing information or user profiles.
- **A key value store** (e.g. Hbase) for low-value and high-volume data such as log files or hit counts.
- **Amazon S3** for user-uploaded data objects such as photos, sound files and big binary files.
• MongoDB for storing the application documents (e.g. bills).

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


