HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Daniel Michel

Databases in the Cloud

HSR University of Applied Science Rapperswil
MRU Software and Systems
Advisor: Prof. Hansjörg Huser

Rapperswil
December 6, 2010

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Cloud Computing

Sometimes, it seems that it is not the capabilities that make a technological hype, but rather the name. Certainly, this holds for cloud computing as well. However, this does not mean that cloud computing has no eligibility to be a hype because of its capabilities. On the contrary: With cloud computing, the next paradigm shift — after moving from mainframes to client-server architectures and then to the internet — has begun [RosenbergEtAl10].

From a non-technical standpoint, the definition of the term cloud computing by Krissi Danielson brings it to the point nicely: «Cloud computing enables users and developers to utilize services without knowledge of, expertise with, nor control over the technology infrastructure that supports them» [Danielson08]. More technically, a cloud is defined by the following five principles [RosenbergEtAl10]:

1. «Pooled computing resources available to any subscribing users»
2. «Virtualized computing resources to maximize hardware utilization»
3. «Elastic scaling up or down according to need»
4. «Automated creation of new virtual machines or deletion of existing ones»
5. «Resource usage billed only as used»

1.2 Cloud Databases

Databases in the cloud, or just cloud databases, are any form of structured, queryable storage that is hosted in the cloud. When talking about cloud databases, people often mean different things. Some may be referring to a pay-per-use service, others may be speaking about a specific piece of software. This paper focuses on the different database services that are currently offered in public cloud environments. Also, a discussion about possible use cases for the different models is included.

Specific DB software packages such as *MongoDB* and *CouchDB* are sometimes also called cloud databases. However, they are normally used to build private cloud database solutions, so they are not discussed here.

CHAPTER

TWO

CLOUD DB SERVICES

2.1 Amazon Web Services

Amazon, being the first company to provide cloud services under the name *Amazon Web Services (AWS)* to external customers in 2006 [RosenbergEtAl10], constantly expanded their service portfolio (see *Amazon Web Services Overview* for a selection of current services). The core of *AWS* is formed by the Xen-based[1] cloud computing service called *Elastic Compute Cloud (EC2)*.

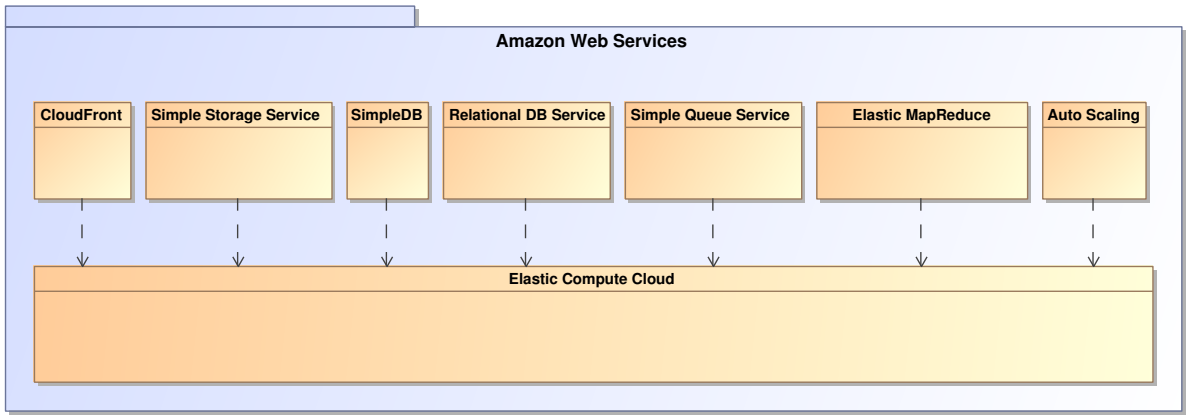

Figure 2.1: Amazon Web Services Overview
A selection of current services running on the Amazon cloud

Currently, Amazon offers two structured database services: *Amazon SimpleDB*, a distributed non-relational data store, and *Amazon Relational Database Service (RDS)*, a fully-featured relational database.

The *Amazon Simple Storage Service (S3)* is a bucket-based storage service that can be thought of as a hard disk in the cloud. As such, it is not a database. But, since *SimpleDB* is based on *S3*, it is mentioned here for completeness.

[1] The Xen hypervisor is a high-performance virtualization solution originally developed at the university of Cambridge, England.

2.1.1 Amazon SimpleDB

SimpleDB was the first structured database service offered by Amazon. It is a simple key-value store and thus not relational. Information in this chapter is based on [WebAmazonSimpleDb].

Architectural Overview

SimpleDB is optimized towards simplicity and availability. To achieve this, many sophisticated DB features were sacrificed:

1. No schema support
2. No support for relations among records
3. Only limited transaction functionality in the form of conditional put or conditional delete
4. Read consistency is not guaranteed by default[2]

Due to points 3 and 4 in the list above, *SimpleDB* is not ACID. However, these restrictions enable the database to replicate more aggressively, which makes the architecture well-suited for cloud environments.

The *SimpleDB* API can be used with a REST- or SOAP-based transport. Response messages are provided in XML.

Data Model

Data in *SimpleDB* is organized into so-called Domains. A domain is a container for items with key-value entries and forms the outermost structural element. Items are comparable to rows in traditional DB systems. An item contains key-value pairs, called attributes, where the values contain the effective data in the database. The structure of *SimpleDB* is summarized in figure *Data Model of Amazon SimpleDB*.

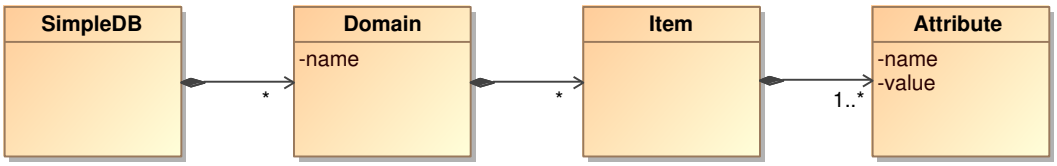

Figure 2.2: Data Model of *Amazon SimpleDB*

Note that an item may contain arbitrary attributes, so there is no predetermined structure such as columns in table-oriented databases[3]. This approach simplifies data migration over time, because there is no database schema: If, for example, an attribute "telephone number" should be added to "person" items, one just needs to modify the `Put` calls in the application where the "person" items are stored.

[2] The default behavior of *eventually consistent reads* can be changed to *consistent reads* on a per operation basis. Switching to *consistent reads* lowers the overall database performance.

[3] This is why databases like *SimpleDB* are also called *document-oriented* databases.

API

The REST- or SOAP-based API matches the design principle number one of *SimpleDB* very well: Simplicity. It consists of only four domain management and five data management operations. The following list gives a short overview over the API operations[4]:

- Domain Management
  - `CreateDomain`
  - `DeleteDomain`
  - `ListDomains`
  - `DomainMetadata`
- Data Management
  - `PutAttributes`
  - `BatchPutAttributes`
  - `DeleteAttributes`
  - `GetAttributes`
  - `Select`

The idea of `Select` is basically the same as of the `SELECT` statement in SQL databases. Note, however, that *SimpleDB* does not offer advanced features such as joins, functions, sub-queries, or other computational constructs. The `Select` operation is just a way to specify attribute selection, simple predicates, sort orders, and result size limitations.

2.1.2 Amazon Relational Database Service

After noticing that *SimpleDB* was not suitable for applications that need a certain minimum of features and robustness in the DB tier, amazon launched its second database service, called *Amazon Relational Database Service (RDS)* in late 2009 [WebAmazonRds].

Architectural Overview

Behind the scenes, *RDS* is a managed MySQL server with the InnoDB storage engine that runs on *EC2*. Thus, as a database, it offers all the advanced features that MySQL with InnoDB offers: Transactions, stored procedures and sub-queries, to name just a few. *RDS* is ACID compliant, unless read replicas are used (see *Replication*).

The API of *RDS* is twofold: The first part is an API for database instance management. It allows for creation and deletion of instances as well as scaling of CPU, memory and storage resources provided to a DB instance. This is available with SOAP or a proprietary HTTP-based transport. The second part of the API provides the interface to the data and is an ordinary

[4] The complete API reference documentation can be found at http://docs.amazonwebservices.com/AmazonSimpleDB/2009-04-15/DeveloperGuide/

MySQL socket interface. Thus, data access can be accomplished with both the MySQL tools and the ODBC/JDBC drivers for MySQL.

Replication

So, what makes *RDS* a cloud database, given it is "just" a MySQL server? This question is indeed justified, as the blog post "MySQL in Spaaaaaace – Amazon Relational Database Service (RDS)" on clouddb.info shows: «I don't see any automated replication (...) and I don't see any kind of clustering or sharding. This is not what most people would call a cloud database» [Lewis09].

Although being true at the time of writing, this statement does not hold anymore: The engineers at Amazon have improved *RDS* in the year that it has been operational. In May 2010, support for *Multi Availability Zone (Multi-AZ)* hot-standby deployments have been added, and in October 2010 support for *read replicas* followed [WebAmazonReleaseNotes].

The *Multi-AZ* deployment is an optional feature of several AWS products, not only *RDS*. If enabled, the whole database is replicated to a standby replica in a geographically separate location. In case of a DB server or whole AZ failure, the cloud performs an automated failover to the standby replica. Since replication occurs in the API layer, data updates are written to all replicas in parallel. As a consequence, the standby replica always contains the most recent data and can seamlessly take over the operation. The failover process is thus fully transparent to the database clients.

Note that *Multi-AZ* standby replicas are not accessible while the master server is working, so it is a concept that improves reliability, but not scalability.

Read replicas fill this gap by providing additional database servers for read-only access. Thus, read-intensive applications scale well with this approach. *RDS* uses the native replication mechanisms of MySQL. Thus, data distribution among the replicas may not always be synchronized. This means that reading data is only eventually consistent, as in the consistency model of *SimpleDB*.

2.2 Microsoft Windows Azure

Windows Azure is the cloud computing platform provided by Microsoft. The service offerings of *Windows Azure* are in many ways similar to those of *AWS*, but they are of course built around Microsoft technologies.

As Amazon, Microsoft offers two structured database services: The *Azure Table Service* conceptually matches *Amazon SimpleDB*, whereas *SQL Azure* forms the counterpart of *Amazon RDS*. Figure *Windows Azure Platform Overview* gives a high-level overview of the *Windows Azure* based services.

In this chapter, the focus lies on the differences of the Microsoft services compared to the respective Amazon services, since the core concepts are the same at both providers.

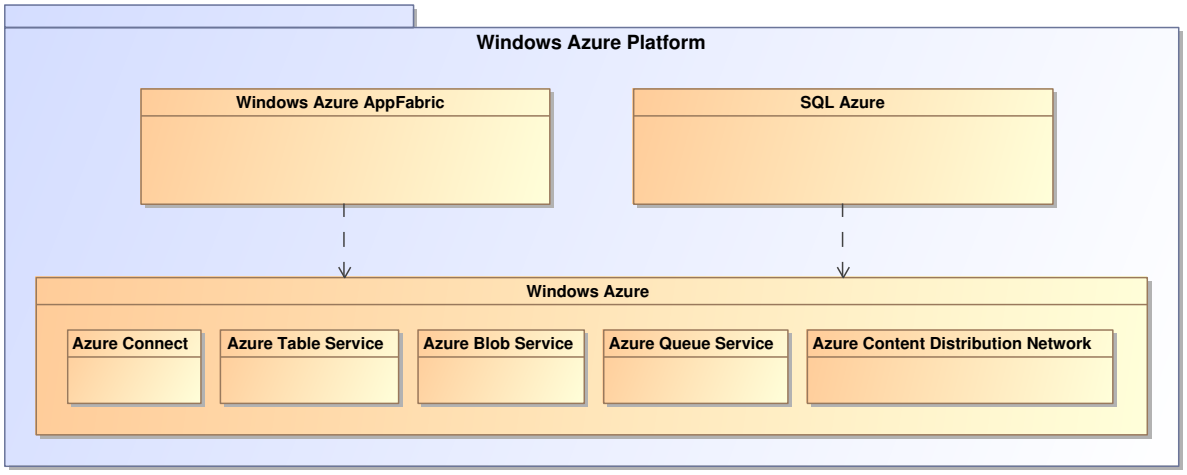

Figure 2.3: Windows Azure Platform Overview

2.2.1 Azure Table Service

Architectural Overview

The *Azure Table Service*, or *Azure Tables* for short, is a highly scalable, document-oriented database service [HaridasEtAl09]. The architecture of *Azure Tables* is almost identical to the architecture of *SimpleDB*, thus it is also not ACID compliant in general[5].

The API is HTTP-based and RESTful, and can either be used directly, or through ADO.NET Data Services.

Data Model & Scalability

The name *Table Service* is a bit misleading, because the tables are not tables in a traditional sense: They play the same role as domains in *SimpleDB* and do thus not strictly structure the data in columns: They are just a container for arbitrary items. Figure *Data Model of the Windows Azure Table Service* summarizes the data model of *Azure Tables*. The similarity to the *SimpleDB data model* is obvious: Tables are domains, entities are items, and properties are attributes.

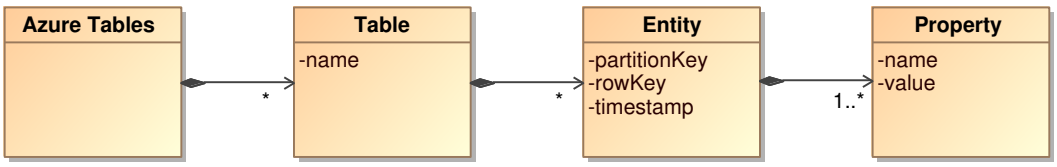

Figure 2.4: Data Model of the *Windows Azure Table Service*

However, *Azure Tables* make the horizontal partitioning capabilities more explicit than *SimpleDB*, as can be seen from the `partitionKey` and `rowKey` attributes. Together, these two

[5] For single tables, however, *Azure Tables* provides ACID guarantees, see *Consistency Model*.

keys form the primary key of an entity. The *Azure Table Service* uses the `partitionKey` to distribute entities across multiple nodes, which is known as sharding – a form of horizontal partitioning.

With this technique, *Azure Tables* can scale well. To achieve good scalability, however, it is crucial to find a good `partitionKey`. This key should partition the data uniformly regarding the requests made to entities.

API

When using .NET as application platform, access to *Azure Tables* will most likely be done through ADO.NET Data Services, which wrap the REST API and provide a somewhat cleaner implementation of the data access. There are no restrictions when using the REST interface directly, but the data access layer needs to be implemented manually.

When querying *Azure Tables*, entities are selected by specifying their `partitionKey` and `rowKey`. It is also possible to filter entities analogous to *Amazon SimpleDB*. The result set is always sorted by `partitionKey` and `rowKey`.

Consistency Model

As stated above, the *Azure Table Service* is generally not ACID. However, it can guarantee ACID-ness under the following conditions:

- Transactions are only allowed on entities that have the same `partitionKey`.
- Consistency is only guaranteed within a single table. Cross-table consistency must be ensured in the application.
- Reads are isolated from updates and can happen concurrently for single queries. Since reads of large data sets must be split up, however, snapshot isolation across continuations cannot be guaranteed.

As already mentioned under *Data Model & Scalability*, data modeling and specification of the `partitionKey` must seriously take scalability and consistency consequences into consideration. Otherwise, the resulting database layout will perform significantly worse.

2.2.2 SQL Azure

Architectural Overview

The main service concepts of *SQL Azure* are similar to those of *Amazon RDS*. *SQL Azure* is a *Microsoft SQL Server 2008* that runs on *Windows Azure*. It supports many features of an ordinary *SQL Server*, but not all. Especially, features that alter the server configuration as well as replication mechanisms and spatial data features are not available.

Apart from that, *SQL Azure* is relational, SQL-based and ACID compliant. The API is, as with *SQL Server*, a *Tabular Data Stream (TDS)* interface. Thus, the established access mechanisms like ADO.NET and ODBC/JDBC work without modifications [Microsoft09].

Replication

The *SQL Server* replication features are disabled in *SQL Azure*, as stated above. This is because replication is provided on the cloud level, which means that data and applications are transparently replicated across multiple nodes in the cloud. Thus, this concept improves the availability but leaves scalability untouched, so this type of replication is similar to the *Multi-AZ* deployments in the Amazon cloud.

If better scalability is needed, the customer needs to implement this at the application level or switch to *Azure Tables*. *SQL Azure* does not directly support a scalable replication model. However, implementing a scalable, relational data storage is easier in the cloud, because resources can be assigned in an on-demand fashion. To achieve this, Microsoft published a whitepaper[6] that discusses vertical and horizontal partitioning models in *SQL Azure* setups.

2.3 Google App Engine

Google is often named in the same breath as the other cloud providers. However, the foundation of the *Google App Engine (GAE)* is fundamentally different from other cloud infrastructures, like for example Amazons. This is because *GAE* does not provide bare VMs, but an application framework consisting of various services to build web applications. This is the reason why *GAE* is commonly termed *Platform as a Service (PaaS)*, where as the other cloud operators in this paper provide *Infrastructure as a Service (IaaS)*.

A drawback of this is that *GAE* is more restricted to certain use cases. There is no possibility to add services that cannot be implemented with *GAE*, in contrast to a VM, where the customer can install any software that possibly runs on the VM OS[7]. An advantage of this approach is that the customer does not need to care about making an application scalable: As long as the *GAE* services are used how they are intended to, the application scales across Googles infrastructure automatically[8].

Another consequence of the *PaaS* model is that the developer must pick one of the supported languages. Currently, there is a *GAE* API for Java and one for Python. Other languages cannot be used on *GAE*.

Since *GAE* in general is targeted at scalable, distributed applications, there is no relational, SQL-based database service in *GAE*. The sole database service of *GAE* is called *DataStore* and is discussed in the following section.

2.3.1 GAE DataStore

Often, people talk about *BigTable* for pesisting data in *GAE*. However, the API for structured data storage in *GAE* is called *Google App Engine DataStore*, which is built on top of *BigTable*,

[6] The whitepaper is part of the *Windows Azure* whitepaper collection and can be found at http://go.microsoft.com/?linkid=9736947

[7] It is of course possible to implement the required service outside of *GAE* and then call it from the Google App. Actually, this is a often suggested solution to circumvent *GAE* restrictions [Hoff09].

[8] However, many application developers seem to have difficulties to establish the new mind set required to build applications on *GAE*, so the application ends up with a very bad performance [Hoff09].

which itself is built on top of the *Google File System (GFS)*. With *GFS* being a distributed file system, *DataStore* is inherently distributed as well [WebGoogleAppEngine].

Architectural Overview

DataStore implements a data model similar to *Amazon SimpleDB* and *Azure Table Service*. It is a document-oriented data store with transparent horizontal partitioning across the cloud. It has transaction support (with certain restrictions, see *Consistency Model & Replication* below) and lets the user choose between *consistent reads* and *eventual consistent reads*. Unlike *SimpleDB*, *DataStore* defaults to the consistent variant, however. Thus, *DataStore* is ACID compliant only in certain cases.

DataStore imposes size limits on API calls. The maximum entity size is 1 MB, and the maximum size of API requests and responses is also 1 MB. This is comparatively low.

Data Model

Compared to *SimpleDB* or *Azure Tables*, there is no `Domain` or `Table` concept; all entities can be thought of as being in the same table. Each entity has a `kind`, which serves the purpose of distinguishing data objects of different type. Note that this is a more flexible model than building a different table for each `kind`, since it opens up the possibility to select data objects of different kind in the same query. However, `kind`-less queries are currently not supported, so the aforementioned flexibility is not achieved.

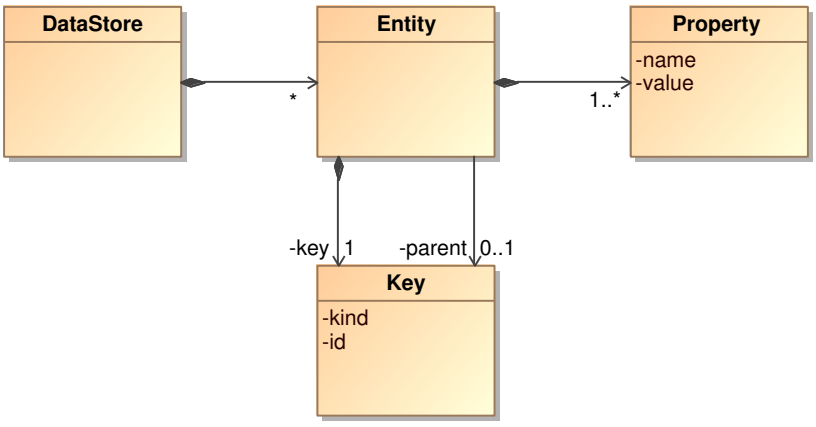

Figure 2.5: Data Model of *GAE DataStore*

Figure *Data Model of GAE DataStore* shows the basic building blocks of the *DataStore* data model. Note that the `kind` is contained in the `Key` class rather than the `Entity`. This does not make any difference though, since each `Entity` contains exactly one `Key`.

Consistency Model & Replication

DataStore is a highly replicated database. The replicas, termed secondaries in *GAE*, remain passive while the primary is working. When the primary becomes unavailable, an automatic

failover to one of the secondaries is performed. The default consistency model is *strong consistency*, which means that data is read only from the primary. But, to improve scalability, the model can be changed to *eventual consistency*, giving *GAE* the permission to distribute read operations across the replicas.

Note that reading multiple entities may still be inconsistent in the *strong consistency* model. This is because entities could change in between the read. To solve this problem, multiple entities must be read in a transaction, leading to *transactional consistency*.

Transactions are supported among so-called `EntityGroups` only. An `EntityGroup` is a set of `Entities` that share the same `parent` (see optional `parent` attribute in *Data Model of GAE DataStore*). *DataStore* uses *optimistic concurrency* to commit transactions. This means that if one transaction is modifying an `EntityGroup`, all other transactions on that `EntityGroup` fail instantly. Thus, the application should retry a transaction if it fails.

API

Google App Engine exposes all services through language APIs in Java and Python, rather than web service interfaces. This makes it easier for the developers, because no manually built gateway code is required.

The Java interface supports established standards such as *Java Persistence API (JPA)* and *Java Data Objects (JDO)*. Implementing the data access layer in a Google App is thus straightforward. There is also a low-level API for cases where an application needs to access the entities directly.

The Python API uses an object-oriented data access interface, too, however there is no data access standard in Python.

CHAPTER

THREE

USE CASES

3.1 Business Decisions

Whether an application is suited to be run in a cloud environment depends on a variety of things. From a high level point of view, the following points can be used to get an idea of why moving to the cloud could make sense [RosenbergEtAl10]:

Reasons **to run** an application in the cloud:

- Knowledge on how to run an IT infrastructure is not available. In the cloud, the virtual machine is ready to run and does not need any hardware at the customer site at all.
- Initial investments in IT infrastructures do not pay off. The reasons for this may be short application life cycles, for example.
- Reliability is too expensive to achieve in-house. Depending on the geographical location, redundant Internet connectivity may not pay off for the application in question. In the cloud, due to the cloud's virtualized nature and very good connection, reliability is cheaper.
- Flexible scalability is needed. When running a custom IT center, the infrastructure needs to be capable of handling the peak loads. This leads to infrastructure that is normally heavily unterutilized[1]. In the cloud, computing resources can be allocated dynamically, thus peak load can be handled by temporarily starting up more instances.

Reasons **not to run** an application in the cloud:

- Legacy systems may be very expensive to adapt to cloud environments. When an application even depends on some specific piece of hardware, that may even render the application incapable of running in the cloud.
- Mission-critical applications are not suited to run in the cloud. This is mainly because of the best-effort service guarantees that are common for cloud infrastructures. Also, when a critical application runs in the cloud, the Internet connectivity of the clients becomes critical, too.
- Highly confidential data is difficult to protect. A virtual machine is always in some kind exposed to its host, so even encrypting data before storing it and protecting the communication channels may not suffice.

[1] This is the fact that lead to the development of cloud computing in the first place [RosenbergEtAl10].

These reasons apply to applications as well as to database systems.

3.2 Architectural Considerations

3.2.1 Tier Location

If the cloud is the environment of choice from a business point of view, it is important to think about the architectural composition of the application. Should the application logic, the data storage and the user interface all run in the cloud? Does it make sense to keep the storage tier in-house or create a rich client that accesses the cloud application?

Of course, the answer depends on the specific case. But, in general, we can say that there should be no separation of tiers that heavily exchange data or rely on fast responses. The database of a cloud application should almost always be located in the cloud, too. This reduces delay and the chance of a service disruption.

The question whether a rich client is the way to go or some HTML interface should be used is really not relevant from a cloud perspective. Both may be suitable or not. This has to be decided by analysing the application itself.

3.2.2 Database Architecture

As it became clear in chapter two of this paper, the consistency and data models, scalability features and APIs differ considerably between the services. It is thus important to know which service suits which requirements best.

Document-Oriented vs. Relational Databases

The database services discussed above can be divided into two categories: The document-oriented databases comprise *Amazon SimpleDB*, *Windows Azure Tables* and *Google DataStore*, whereas the relational databases are *Amazon RDS* and *SQL Azure*.

In the following, we discuss three imaginary, prototypical applications where one or the other category makes more sense. This should give some directions to decide for other applications whether a document-oriented or relational approach is better.

The **Customer Relationship Management (CRM) application** has a web-based interface for quick lookups and a rich client for people that work with the system often. It organizes customers into different groups and collects statistical information. A reporting engine is used to pull the essence out of the large data collections and displays them in pleasant diagrams and figures.

> $\longrightarrow$ **Use an relational database**
>
> Since the reporting engine makes heavy use of statistical functions and the data follows a strong structure, the CRM application should use a relational database

like *SQL Azure*. The RDBMS supports the application with complex join statements, sophisticated filtering and statistical algorithms. Consistency of the data is important, but scalability is a minor issue, because the system is normally used by a well-known, rather static user group.

The **Social Marketplace Platform** combines online auctions with a social network. The application has an ever growing user base, people all around the world access the system any time a day. The whole auction platform is built around the user data that comprises all the social network connections.

⟶ **Use a document-oriented database**

This application needs the massive scalability features of a document-oriented database like *Google DataStore*. The user data is too large to fit in a single table and is thus distributed over many database systems. Records for a specific user can be stored in a DB server near to his or her location to improve performance. Consistency is important for the auctions, but can be implemented in a middle layer on top of the DB system. Replication of whole datasets ensure that even in the case of a server failure, the overall system continues to run.

The **Unified Collaboration System** is an application that brings different web services in the field of communication and collaboration together and enables the users to use email, instant messaging and other means of communication at a central place. The application itself has no data store, but needs to save some logging information.

⟶ **Use a document-oriented database**

The collaboration system needs a simple, inexpensive store for structured data records. Availability, scalability and consistency are not an issue. A basic form of a document-oriented database like *SimpleDB* suffices.

Managed vs. Unmanaged Relational Databases

As already mentioned in chapter *Cloud DB Services*, *RDS* and *SQL Azure* are managed MySQL or MS-SQL servers, respectively. This means that the customer does not need to care about setup and maintenance of the (virtual) DB server machine and server software. What is the difference to a manually built database solution on a cloud VM?

The managed servers offer several advantages. The most obvious is that no knowledge on database server software is needed on the customer side. Additionally, the completely transparent replication of a *Multi-AZ* deployment, for example, would be very involved to set up manually. Thus, I regard the replication features as the most beneficial ones of managed DB servers compared to a custom solution.

Custom database servers in the cloud may also be the solution of choice, mainly because it enables adjustment of the database to specific needs. For example, a company with broad knowledge on an other database system than MySQL or MS-SQL Server will most likely want to stay with their system when moving applications to the cloud.

For the average needs, however, I expect that the managed server is a well-suited solution, mainly because many applications do not leverage advanced database features anyway. For

these applications, it does not matter exactly which RDBMS is running, so *RDS* or *SQL Azure*, working out of the box, are just right.

3.3 Conclusion

After introducing different database services of Amazon, Microsoft, and Google, I presented several decision helpers and use cases concerning databases in the cloud. As always, there is no "best" service, but the right one has to be chosen based on the architectural requirements for the database. An additional property of the different services that might be worth being taken into consideration is the price. I do not give an overview of the pricing models here, just because they tend to change often. The service costs can be found on the respective websites easily.

Another thing that is not explained here is whether the service from one provider or the other would be better for certain cases. But, the architectural discussions in chapter *Cloud DB Services* provide a detailed description and explanation of the core concepts. They should be a good point of reference when having a concrete case at hand.

Apart from that, choosing a cloud provider is always a matter of technology preference and trust, too. A company that runs a Microsoft IT infrastructure will most likely want to stay with Microsoft as a cloud provider, for example.

BIBLIOGRAPHY

[Danielson08] Krissi Danielson, “Distinguishing Cloud Computing from Utility Computing”, SaaS Week Blog, 26.3.2008

http://www.ebizq.net/blogs/saasweek/2008/03/distinguishing_cloud_computing/

(retrieved 21.10.2010)

[HaridasEtAl09] Jai Haridas, Niranjan Nilakantan, Brad Calder, “Windows Azure Table” Whitepaper, May 2009

http://go.microsoft.com/fwlink/?LinkId=153401

[Hoff09] Todd Hoff, “Google AppEngine – A Second Look”, High Scalability Blog, 21.02.2009

http://highscalability.com/google-appengine-second-look

(retrieved 4.11.2010)

[Lewis09] Lewis C., “MySQL in Spaaaaaace – Amazon Relational Database Service (RDS)” CloudDB Blog, 27.10.2009

http://clouddb.info/2009/10/27/mysql-in-spaaaaaace-amazon-relational-database-service-rds/

(retrieved 30.10.2010)

[Microsoft09] Microsoft Corporation, “Similarities and Differences of SQL Azure and SQL Server” Whitepaper, Sept. 2009

http://go.microsoft.com/?linkid=9692818

[RosenbergEtAl10] Jothy Rosenberg, Arthur Mateos, “The Cloud At Your Service”, ISBN 978-1935182528, 28.10.2010

Chapters 1 and 3

[WebAmazonRds] Amazon Web Services LLC, “Amazon Relational Database Service (RDS)” Website

http://aws.amazon.com/rds/

(retrieved 30.10.2010)

[WebAmazonReleaseNotes] Amazon Web Services LLC, “AWS Release Notes” Website

http://aws.amazon.com/releasenotes/

(retrieved 30.10.2010)

[WebAmazonSimpleDb] Amazon Web Services LLC, “Amazon SimpleDB” Website

http://aws.amazon.com/simpledb/

(retrieved 29.10.2010)

[WebGoogleAppEngine] Google Inc., “Google App Engine – Developer’s Guide” Website

http://code.google.com/appengine/docs/

(retrieved 14.11.2010)