Benchmarking Web-Application Architectures in the Cloud

Master Thesis
Simon Loesing

Supervised by:
Prof. Dr. Donald Kossmann
Tim Kraska

Systems Group, http://systems.ethz.ch/
Department of Computer Science
ETH Zurich

March, 2009 - October, 2009
Abstract

Cloud computing has introduced a shift of applications and services from local data centers to remote internet servers. Providing high scalability, fault-tolerance and a pay-per-use cost model, the cloud offers attractive services which cover the entire range of data processing and data storage. While clouds become more and more popular, there are currently only few performance evaluations to verify how well cloud services perform compared to traditional architectures. Moreover, with the increasing number of cloud providers and services, it has become a challenge to compare all offerings.

To address this issue, we have distinguished the most common architectures for applications for the cloud and developed a benchmark taking the specific properties of the cloud into account. We implemented this benchmark on services of different cloud providers and carried out an extensive performance analysis. We discovered that depending on the setting and the provider, the performance as well as the cost is subject to considerable divergence. Especially, in terms of performance and scalability, cloud storage services can not reach traditional relational databases.
Acknowledgments

I would like to express my gratitude to my supervisors Prof. Dr. Donald Kossmann and Tim Kraska for their provided help, guidance and continuous support through the course of this work. I would also like to thank the Systems Group for the very friendly and helpful atmosphere. Last but not least, many thanks to all my friends who supported me during this thesis, as well as my lovely family.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Existing Benchmark Solutions</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Benchmarks and Related Work</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The TPC-W Benchmark</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>A New Cloud Benchmark</td>
<td>26</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Benchmark Configurations</td>
<td>26</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Metrics</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Benchmark Implementation</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>TPC-W Reference Implementation</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Architectures</td>
<td>32</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Relational Database running in the Cloud</td>
<td>32</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Amazon SimpleDB</td>
<td>34</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Google App Engine</td>
<td>35</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Database Library built on S3</td>
<td>36</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Sausalito XQuery Application Server</td>
<td>37</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Comparison</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Remote Browser Emulation</td>
<td>38</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Reference Implementation and Refactoring</td>
<td>38</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Dynamic Load Generation</td>
<td>40</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Measurements and Results</td>
<td>40</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Distribution and Result Collection</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Data Population</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Testing</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>Experiments and Results</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Benchmark Setup</td>
<td>45</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Experiment Setup</td>
<td>45</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Remote Browser Emulation</td>
<td>46</td>
</tr>
<tr>
<td>6.1.3</td>
<td>System under Test</td>
<td>46</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Data Storage and Scale</td>
<td>47</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Benchmark Web-Interaction Mix</td>
<td>47</td>
</tr>
<tr>
<td>6.2</td>
<td>Results and Analysis</td>
<td>47</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Data Population</td>
<td>47</td>
</tr>
<tr>
<td>6.2.2</td>
<td>MySQL Database in the Cloud</td>
<td>49</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Amazon SimpleDB</td>
<td>52</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Google App Engine</td>
<td>55</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Architecture Comparison</td>
<td>57</td>
</tr>
<tr>
<td>6.2.6</td>
<td>EC2 I/O and CPU performance</td>
<td>59</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>65</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>66</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this first chapter we would like to show the motivation behind the work of this thesis and how it contributes to the analysis and comparison of cloud computing systems.

1.1 Motivation

Cloud computing is based on the idea of providing virtually unlimited computing resources as needed. Computing resources can be allocated and disallocated on-the-fly and users only pay for the resources they use. Forthcoming, we call this the pay-as-you-go cost model. Acquiring computer resources immediately in a transparent way and at reasonable cost can be seen as a milestone in the history of IT. For the first time we can talk about “computer utilities” [29], similar to electric and telephone utilities we are using every day.

Cloud systems provide virtually unlimited scalability, high availability and fault-tolerance [8, 26]. To achieve these characteristics cloud systems are often built on top of large distributed systems in which the data is replicated within one or even across data centers. Built on these huge infrastructures cloud providers offer services to host the complete application stack in the cloud. This ranges from generic components like virtual application servers and storage services to more application specific services like for example messaging services.

Today, cloud computing is no longer a hype and driven by solid business models companies like Microsoft, Google and Amazon are building up and expanding their cloud services [44]. Following this trend the landscape of cloud computing providers has become bigger and bigger [34]. New providers have appeared on the market and have tried to convince the users of their services. All this has resulted in varying service offerings sometimes even built one top of each other [30, 1] which get harder to overlook and to compare. The examination and comparison of cloud services and providers is the main topic of this thesis.

1.2 Problem Statement

Cloud systems provide many different types of services. Trying to structure all offerings we identify two groups. First, services which allow to process data and run applications. These services range from high level managed runtime...
environments where the code has simply to be uploaded to low level infrastructure solutions in which the developer gets root-access to operating systems on virtual machines. Second, services which main purpose is the storage of data. These services differ from each other in the provided data models, access APIs and consistency guarantees.

While some cloud providers specify proprietary services which have to be used together, other offerings are loosely coupled and can be combined in several different ways. That fact makes the comparison of cloud systems a complex task.

In order to analyze the dependencies and get a general idea about comparing cloud systems, a benchmark taking the specific characteristics of cloud systems into account has to be developed. Designing such a benchmark is a topic of this thesis. Thereby, we have to analyze existing benchmark solutions and verify if the standard metrics are significant enough to provide meaningful results or if they have to be extended.

Furthermore this benchmark will provide the opportunity to test different existing cloud settings and to compare them to traditional architectures based on relational databases.

### 1.3 Contributions

First, we identify a reference architecture for applications for the cloud and explain several alternatives which scale and take advantage of individual cloud infrastructures.

In a second step we design a benchmark for cloud systems based on existing solutions. This benchmark provides metrics enabling to test common cloud properties like scalability, cost, reaction to peak-loads and fault-tolerance.

To demonstrate the relevance of these criteria, the benchmark is implemented on infrastructures of several well known cloud providers like Google App Engine [25] or Amazon Web Services [4] and additionally uses different provided data storage services. One of these implementations uses a relational database which allows a comparison with traditional enterprise architectures.

As a result the reader gets an evaluation of several ways to implement web applications in the cloud and how to persist data. To the best of our knowledge, such an extensive comparison and analysis has not been done so far.

The results of this thesis are of high practical relevance as they identify the current strengths and weaknesses of cloud systems. In particular the relation between performance and cost will have a major impact on the future success of cloud systems. Moreover we verify if the promises of cloud providers in terms of scalability and fault-tolerance are valid.

Finally this work identifies if cloud systems are a serious alternative to private data centers and therefore help organizations in their decision to move to the cloud or not.

### 1.4 Structure of this Thesis

This thesis is structured as follows:

- **Chapter 1** underlines the motivation behind this thesis and makes the reader aware of the problems in comparing different cloud architectures.
1.4 Structure of this Thesis

- **Chapter 2** introduces the reader in the world of cloud computing. We define the terminology and explain technical characteristics of cloud systems. This chapter ends with an overview of the biggest cloud providers.

- **Chapter 3** describes a reference architecture to build applications for the cloud and analyzes how far this architecture can be modified and extended. This chapter deals with topics such as scaling and load balancing.

- **Chapter 4** focuses on load-testing. We collect requirements for a cloud benchmark, analyze how well existing benchmark solutions are adequate and finally suggest new metrics for benchmarking cloud systems.

- **Chapter 5** contains details about different settings and providers on which the benchmark is implemented. Beside an overview of the architectures and their comparison the reader gets a review about implementation details as well as the challenges we had to overcome.

- **Chapter 6** shows the results of our benchmark runs and analyzes the weaknesses and strengths of each architecture. Furthermore, the analysis allows new statements about cloud computing in general.

- **Chapter 7** presents a conclusion of the thesis results and indicates the upcoming work.
Chapter 2

Cloud Computing

In this chapter we introduce the concepts and terminologies used in the following chapters. Additionally we give an overview of the big players in the cloud computing market.

2.1 What is Cloud Computing?

Cloud Computing is a new paradigm in ways of develop, deploy and access software and to provisioning of hardware.

It is difficult to find a widely accepted definition of cloud computing. Many experts have already tried to identify all aspects and tried to define what the cloud is [21]. These definitions range from specific technological aspects to business models. Lately there has been some interesting efforts to combine all the statements in order to find a common definition on how cloud computing is recognized today [54]:

Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.

This definition is extensive because it shows the implication of cloud computing on the software and hardware spheres. Indeed cloud computing must be understood as the applications delivered as a service and also as the hardware and systems software in the data centers that provide those services.

A cloud infrastructure is highly scalable. Users can acquire computing resources on demand, having the possibility of up- or down-scaling within minutes instead of years. That is why there is no need to provision resources for peak workload any more. Cloud providers hide the complexity of their physical hardware infrastructure by offering the illusion of virtually infinite computing resources. Cloud systems offer a pay-per-use cost model. This means services are generally paid on a short-term basis, usually per hour or per request independent on how many resources are used. Hence, it makes no difference using 1000 servers for one
hour or one server for 1000 hours. This flexibility, without paying for large scale, is unprecedented in the history of IT.

All these advantages lead to a new trend towards an infrastructural shift. More and more locally installed data and programs, on desktop PC or Servers are shifted to distant internet servers, into the cloud [26].

2.2 Types of Cloud Services

After the definition of cloud computing in the previous section, we now want to see how the definition can be applied to existing cloud offerings. Cloud services can be categorized in three different types: Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) (see figure 2.1) [9, 43]. Additionally clouds are called public if their services are accessible by everyone over the internet and private if they refer to internal data centers of an organization.

![Figure 2.1: Types of Cloud Services](image)

### 2.2.1 Software as a Service

Software as a Service (SaaS) refers to special-purpose software made available via the internet. These services are not suitable for building individual applications and are restricted to what the application is and can do. There is only few information published about the underlying technology. That is why these services are excluded in the further discussion.

Examples of SaaS application are public email providers (Gmail, Hotmail, etc.), the Google apps, various search engines, etc.

### 2.2.2 Platform as a Service

Platform as a Service (PaaS) provides a high-level environment, a domain-specific platform, on which developers write customized applications (e.g., Google’s App Engine is targeted exclusively at traditional web applications). The developer can focus on the main functionality of his application and program to a more or less open specification. After deployment, the service provider
2.3 Characteristics of Cloud Computing

automatically takes care of maintenance, scale out and load balancing. Unfortunately, SaaS applications are often restricted to the platforms abilities and can not be easily migrated to other providers. It would be difficult to start with a toy application on a platform provider and then switch to an infrastructure cloud (see subsection 2.2.3) when it matures. Examples of PaaS services are Google’s App Engine [25], Microsoft’s Azure platform [33] or Sausalito [1].

2.2.3 Infrastructure as a Service

Infrastructure as a Service (IaaS) provisions resources such as servers (often in form of virtual machines (VM)), network bandwidth, storage, and related tools necessary to build an application environment from scratch. The user has high flexibility and developers can still deal with low level details such as starting VMs or mapping static IP-Addresses to VMs. As a VM behaves almost similar to a physical server, virtually any web-application can be mapped to this type of service. As the management of these customizable environments is in most cases highly application-dependent, it is difficult for the providers to offer automatic scalability and fail over. Examples of IaaS services are Amazon’s Elastic Cloud Computing (EC2) [4], GoGrid [24] or Rightscale [39].

2.3 Characteristics of Cloud Computing

Cloud computing is centered around overlapping concepts of distributed, grid and utility computing. In this section we first distinguishing between these different concepts and then analyze the three central aspects of cloud computing in more details: scalability, availability and the pay-as-you-go model. Ross and Westerman define utility computing as a collection of technologies and business practices that enable computing to be delivered seamlessly and reliably across multiple computers [40]. Even if less concrete, this matches with our definition of cloud computing making utility Computing a synonym. Furthermore the name “utility” introduces the vision of computer utilities which dates back to 1969 where it was presented by Leonard Kleinrock, a chief scientist of the ARPANET project [29]. Grid computing dates back to the mid 1990s and was a first attempt to provide consumer’s computing power on demand. Foster defines grid computing as the aim to enable resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations [18]. Cloud computing has evolved out of grid computing as it shares the same ideas and the vision to increase reliability and flexibility by outsourcing computation needs to third party providers. However, taking into account the evolution of the internet in the last decade and the current trend to service-oriented applications let’s one sense that both concepts also differ in several areas. To mention some differences, grid computing is based on standard protocols, the terms of payment are more rigid, the quality on service guarantees are often best-effort only and the control decentralized [54, 19]. It is obvious that cloud computing systems are based on principles of distributed systems. The analysis of this topic is not in the scope of this thesis, for further details we refer the reader to [48].
2.3.1 Scalability

For each high-traffic internet application or computation intensive enterprise application scalability is a major issue. Companies usually have to provision resources for peak workload which is 2 to 10 times the average workload [8]. Traditional data centers therefore waste a lot of resources during idle time and in addition only provide limited scalability. Cloud providers on the other hand, offer virtually infinite scalability on-demand.

Resources can be allocated in hours or even minutes instead of years without buying hardware or investing in infrastructure. Similar to scaling up, disallocation of resources can also happen in a short period of time. This ability to scale on-the-fly is best describe by the word “elasticity”, coined by Amazon, one of the pioneers in cloud computing.

Although the idea of scalability sounds simple, on-demand computing power requires to solve several technical challenges. It requires the ability to dynamically resize resource allocation and moving customers from one physical server to another without notice. On the one hand this is handled by the virtualization of the entire infrastructure and on the other hand by a system capable of managing and maintaining itself. Large cloud providers like Google or Amazon take advantage of their already developed scalable software infrastructures (such as Google File System [22], BigTable [13] and Dynamo [16]) which give them the required expertise for the cloud business.

2.3.2 Availability

Today, users expect 100% availability from a service. To reach 24/7 uptime, each system must be capable of dynamically compensating unexpected load variations and also react to all kinds of failures.

Cloud systems introduce several mechanisms to guarantee high availability. To illustrate this behavior we take a closer look at Amazon’s Simple Storage Service (S3). S3 is a key-value storage system for all types of bytestreams accessible over a simple SOAP or REST API. Although Amazon has not published details on the S3 implementation, we take [16] as a reference for Amazon’s design principles. It seems that S3 replicates all data to several data centers. This means neither the failure of hardware (e.g., server crash or hard disk failure), nor the outage of an entire data center will have an impact on the availability of the data. In case of high increase in demand, data is simply automatically replicated to more servers. This permits to keep the average response time virtually constant.

Although several replicas increase the fault-tolerance, cloud systems are not protected against system-wide outages. In the year 2008 for example, there has been some recorded downtimes due to overload and programming errors on Amazon’s S3 and Google’s App Engine [47, 49, 57]. During the year, S3 has been unavailable for a total of 10 hours and Google App Engine for 5 hours. Nonetheless this results in an overall availability of more than 99.89% which is better than most enterprise IT infrastructures.

Meanwhile, several providers commit themselves to guarantee an uptime in their service level agreements (SLA). For example Amazon’s S3 SLA [3] ensures an uptime of 99.9%. Customers are eligible to receive a service credit if these conditions are not met.

In case of PaaS providers automatic scaling and high-availability mechanisms such
2.4 Challenges for Cloud Systems

as load-balancing are already part of the infrastructure. The developer does not have to longer take care to manually scale or distribute the application which is often an error prone and costly task. The cloud provider undertakes the task of the developer.

Similarly, the migration of systems into the cloud transfers a large amount of control and maintenance responsibilities to the cloud providers. The need for specialized know-how decreases and maintenance becomes less error prone. The general fault-tolerance therefore increases and makes the system less vulnerable to failures.

2.3.3 Pay-As-You-Go

The pay-as-you-go cost model is more simple and more transparent than traditional payment methods. The customer only pays for the resources he/she really uses. It is a widespread cost model also used by electricity and gas companies which has now reached the IT sector. There is no longer the need for the customer to pay high fixed prices for a service with an individual SLA. Instead resources are acquired just at the time needed and are as well charged on a short term basis. This means that the user can see with a short delay (usually a few hours) which resources have been consumed and how much it will cost.

Cloud providers have different cost models. PaaS providers price plans are usually based on CPU utilization whereas IaaS cost is charged on a per hour basis. Depending on the usage scenario both models have their advantages and drawbacks. If, for example an application has to only support one transaction per hour, a price model based on CPU utilization would only charge this transaction whereas for a rented virtual machine the cost is the same regardless if one or 1000 transactions are performed per hour. To give some pricing examples, the Google App Engine charges $0.10 for an hour of CPU time, a medium Amazon EC2 instance costs $0.20 per hour. The cost models are often more fine grained and take further aspects into account. Among the list of further cost elements are for example cost for network traffic or for the number of I/O operations on a storage service.

Although pay-per-use seems attractive, summarizing the cost for needed resources may result in higher operating cost than buying it’s own servers. This is obvious because cloud providers have to amortize their infrastructure and at the same time need to be profitable. Nonetheless, if we take the maintenance (e.g, server administrators) and infrastructure cost (e.g, power and cooling) into account the cost balance might already change in favor of cloud computing. Another major advantage is based on the fact that no resources must be provisioned for handling peak workloads. Data centers usually have to keep a lot of resources in reserve in order to handle peak loads and therefore waste a lot of idle computing power. This is no longer necessary since the cloud offers resources just-in-time.

2.4 Challenges for Cloud Systems

Cloud computing systems promise remarkable features. Nonetheless, they also have to face a lot of challenges.
2.4.1 Limits of Storage Services

Persistent data storage is a critical task for most applications. For the storage of structured data, we usually use relational database management systems (RDBMS) which provide strong consistency guarantees as defined by the ACID (Atomicity, Consistency, Isolation and Durability) properties [14].

In order to scale, databases have to be partitioned. Protocols like 2PL (two-phase locking) make sure the ACID guarantees are maintained across multiple database instances. Unfortunately, because of Brewer’s famous CAP Theorem [23] being consistent over many partitions has an impact on the overall availability. According to the CAP-Theorem, it is not possible to provide availability and strong consistency together in the presence of network failures.

In a cloud storage service fault-tolerance and high-availability are two of the main assets which prevent that data storage becomes a bottleneck. Consequently, most cloud providers sacrifice strong consistency guarantees for availability and offer weaker forms. The most often provided consistency level is eventual consistency [48]. Eventual consistency means that updates will eventually become visible to all clients and that the changes persist. This level of consistency is for example provided by Amazon’s S3. S3 is highly reliable and not restricted in scale-out because data is automatically distributed across several data centers.

However, in order to respond to high consistency needs Amazon provides another storage service called Elastic Block Store (EBS). EBS is a special storage service only accessible by one virtual machine and replicated in one data center. Highly consistent and very limited in scalability. Further information about S3 and EBS can be found in section 2.5.1.

Both examples show how different the guarantees and limitations of cloud storage services are. Further differentiation factors are variations in the richness of query APIs, the performance guarantees offered, and the complexity of data structures supported (e.g., schema-less blobs vs. column-oriented storage).

2.4.2 Technical Constraints

In the last subsection, we have analyzed a specific technical property of cloud storage systems. Yet, there are further technical constraints we want to mention in the following.

First, due to virtualization cloud providers offer virtually unlimited computing resources. However, this introduces the risk of performance unpredictability. Virtual Machines of a cloud computing system have to share CPU, memory and I/O Access. While the first two points are handled well, the I/O sharing is more problematic. A test of writing a 1 GB file on 75 EC2 Instances done in [8] shows a deviation of more than 16% in the mean disk write performance. This demonstrates that several virtual machines interfere in their attempt to access I/O in parallel which makes it impossible to predict the exact performance of a running virtual machine in the cloud.

Second, data centers of cloud providers have a wired connection to the internet which physically limits the data transfer rate. Garfinkel measured the bandwidth to Amazon’s S3 from different locations around the world and found out a read performance of up to 30 MB/s while accessing several large files concurrently [20]. To prevent fraudulent use of the available bandwidth, cloud providers can more or less restrict the amount of data transferred by adjusting the price per GB of
in- and outgoing data transfer. Especially in case of data intensive applications this could become an important issue for customers as the cost of transferring terabytes add up and become very expensive. Last but not least, the transitional risk, the challenge of moving applications to a new environment into the cloud must not be underestimated. To migrate a complete application architecture developers often need to master multiple programming languages and operating environments. These technological challenges can take time and consume limited resources.

2.4.3 Policy and Business Challenges

The first issue that may cross ones mind when talking about moving applications to remote internet servers are concerns about privacy and data security. In the cloud, sensitive data is externalized into an open network and no longer protected by the internal IT infrastructure. As there is no longer a physical control over the data, the fear might arise that the data is accessed by unauthorized people. Cloud providers have to react to this concern by providing guarantees to their customers (e.g., encryption of all stored data,). A similar issue is the location of the stored data. Many countries have laws requiring customer data to be kept within national borders. The same with companies who do not want governmental institutions of specific countries to be able to access their data. For example, Amazon has already reacted to this fear by introducing availability zones which allow the customers to choose if they either want to store their data in Europe or in the United States.

Another challenge is the question of software licensing. Today licensing models are usually restricted to specific hardware platforms and/or number of CPU cores. Users buy enterprise software and then pay an annual maintenance/support fee. These models are not flexible enough to match with the use-on-demand and pay-as-you-go paradigms. Therefore, we can currently see commercial software companies change their licensing structures to better fit cloud computing. For example, Microsoft and Amazon offer EC2 instances with a Windows Server or SQL Server at a cost of $0.15 per hour instead of $0.10 for the open source version. The licensing cost is in this example is directly included in the hourly usage cost.

Concluding this section, one might say that the mentioned challenges are critical obstacles to the success of cloud computing. All the more it is interesting to see how the cloud providers will deal or even overcome these problems in the near future.

2.5 Cloud Computing Providers

The first requirement for providing public cloud services are large connected data centers. Many large internet companies, including Microsoft, Google and Amazon already own such large infrastructures for their core businesses. The idea of selling idle computation power, increases revenues and at the same time amortize large investments was a prime reason in the development of cloud business models. Under these circumstances it might not be surprising that the three mentioned companies have become the big players in the cloud market.
2.5.1 Amazon Web Services

Amazon Web Services (AWS) [4] was launched in July 2002 and has since then continuously extended and improved its cloud offerings. Today AWS counts more than a dozen cloud services among which the most important will be shortly described in the following.

Simple Storage Service (S3)

S3’s main purpose is the storage of files. S3 stores objects of any bytestreams with a variable size ranging from 1 Byte to 5 GB. An object is identified by an URI and can be accessed and remotely modified through a SOAP or REST-based interface. The interface offers basic operations like `get(URI)` or `put(URI, bytestream)` but also more advanced functions like `get-if-modified-since(URI, timestamp)`. Each object is associated to a bucket. S3 provides several ways to scan through objects of a bucket. For instance, a user can retrieve all objects of a bucket or only those objects whose URIs match a specific prefix.

S3 focuses on availability and reliability and therefore replicates the data across several data centers. The provided consistency level is eventual consistency. Experiments have shown that S3 is optimized for the reading of large files [20, 10]. The latency for small files is too high to provide high bandwidth.

S3 is not free. It costs $0.15 for 1 GB of data stored per month. Additional cost is generated by the amount of data transferred and the number of `get` and `update` requests. S3 costs $0.10 to $0.18 per GB of network bandwidth consumed and $0.01 for 10,000 `get` requests or 1,000 `put` requests.

Elastic Cloud Computing (EC2)

EC2 allows to rent virtual machines by hour. The user can start VMs from a large pool of publicly available Amazon Machines Images (AMI). AMI is a special format of virtual images deployable on the EC2 infrastructure. With the provided EC2 tools each AMI can be customized and stored as an own private image. A running AMI is called an EC2 instance. With the AWS Management Console Amazon provides a web-interface which allows to easily start, terminate and monitor EC2 instances.

EC2 instances do not have a persistent storage. When an instance is stopped or crashes all data is lost. As solution Amazon provides the Elastic Block Store service. EBS allows to create virtual persistent discs of variable size. An EBS volume behaves like a hard drive and can be mounted at a specific path in one virtual machine’s filesystem. On this mounted drive all stored data becomes immediately persistent. Alternatively, the data can also be stored on S3.

EC2 instances are paid per hour, depending on the instance size (provided CPU power and memory), the price ranges from $0.10 to $0.80 per hour. AMIs with Microsoft’s Windows operating system are around 25% more expensive as the license is directly included in the per hour price. Similar to S3, the amount of data transferred to or from an EC2 instance is charged at $0.10 to $0.17 per GB. EBS costs $0.10 per GB-month of provisioned data and the same amount for one million I/O requests.

Since October 2008, Amazon EC2 is no longer in beta state and offers an SLA. The user can also choose Europe or the United States as a location to run instances.
2.5 Cloud Computing Providers

SimpleDB

SimpleDB is a service which provides simple database functions in the cloud. SimpleDB is based on a semi-structured data model which allows to create, store and retrieve data sets inside a domain. The main advantages of SimpleDB is automatic data indexing and a simple but restricted query language. Similar to S3, SimpleDB provides a SOAP and REST interface to manage the data. Meanwhile libraries for multiple programming languages which provide high level APIs to the SimpleDB functions have been published.

The cost for SimpleDB is based on machine utilization. Amazon has developed a mechanism to measure the processing time each query needs. One compute hour costs $0.14. The cost for data transfer ranges from $0.10 to $0.17 per GB. For data transfer between Amazon data centers of the same region (e.g., SimpleDB to EC2 instance in Europe) no cost is charged.

2.5.2 Google App Engine

Google App Engine is a platform to develop and host web-applications in the Google infrastructure. It was first released in April 2008 and is since then under continuous development. Currently the App Engine offers two runtime environments, a Java and a Python one. Both environments are running in what Google calls “the Sandbox”. The Sandbox is an infrastructure which hosts each application in its own secure and reliable environment. In the sandbox an application can only run code in response to a web request and has no rights to write to the filesystem. To ensure availability and scalability, the App Engine probably replicates the application across several physical servers.

All persistent data has to be stored in Google’s Datastore. The Datastore is a distributed data storage service most likely based on MegaStore [11]. It can store data objects, or entities, which have a kind and a set of properties. Entities are basically schemaless, meaning that the data structure can be freely defined and modified by the application code. Data is accessed with an SQL-like query language called GQL which has several restrictions. A main drawback is that join operations are not supported. However, the Datastore supports atomic transactions and provides strong consistency guarantees. Outside transactions the isolation level is read committed. Inside transactions the isolation level is serializable.

Like most cloud services Google App Engine uses a pay-per-use cost model. First, the user is charged for the computation power consumed in terms of CPU time. This amounts to $0.10 for one CPU-Hour. Second, 1 GB of network traffic costs between $0.10 and $0.12. Finally each GB of data stored in the Datastore is charged with $0.15 per month. To keep cost under control, the App Engine allows to define quotas to limit the cost generated per day. If the limit is reached, all services are automatically disabled.

2.5.3 Microsoft Azure

Microsoft Windows Azure was introduced in October 2008 and is a cloud service platform hosted in Microsoft’s data centers. It provides a set of services running on top of virtualized and customized Windows 2008 Server machines. Similar to Google’s App Engine, applications running on Azure can be developed locally and then deployed to the cloud. For each application Azure starts a predefined
number of virtual machines on which the application is then executed. The Azure platform automatically takes care of load balancing all entering requests to the running instances. Applications have to be developed using the .NET framework and are therefore limited to the programming languages supported by the .NET platform. Nonetheless, a Java and a Ruby SDK have been made available recently to increase the Azure platform interoperability.

Data is made persistent through a cloud storage service called Azure SQL. Azure SQL is based on the well known Microsoft SQL Server. Unlike running the database on a physical server, Azure SQL partitions and replicates all databases automatically to multiple servers. Databases become visible to the developer as virtual objects which no longer correspond to physical servers. However, the data access mechanisms remain the same as Azure SQL supports the traditional relational model and can query data with SQL and Transact-SQL.

The pricing of the Azure Platform is similar to other cloud providers. The compute hour costs $0.12, data stored in the filesystem $0.15 GB-month, and 1 GB of data transfer between $0.10 and $0.15. A database on Azure SQL ranges from $9.99 to $99.99 per month depending on the amount of storage needed.

2.5.4 Others

There are several smaller players on the cloud market covering the whole range of cloud services and types. Figure 2.1 lists some examples. The cloud market is currently in movement and in the near future we will certainly see established companies like Sun or IBM, as well as new companies entering the market and presenting innovative solutions.
Chapter 3

Cloud Architectures

This chapter defines a reference architecture for applications for the cloud and presents several ways in which this architecture can be extended and scaled. Concluding this chapter we show how this architectures can be implemented on well known cloud services.

3.1 Reference Architecture

Today, complex software systems are usually based on multi-tier architectures. Multi-tier systems follow an architectural paradigm that is based on separation of concerns. The architecture considers a vertical decomposition of functionality into a stack of dedicated software layers (tiers) [37]. Decoupling applications into several tiers brings several advantages. We gain much flexibility because the dependency between components is reduced. The system can therefore more easily be adapted to changing requirements. Additionally, all tiers can run on different servers increasing the overall system performance.

The most common architecture is the three-tier architecture consisting of a data management tier, an application tier and a client tier. In web-applications the client tier is usually an internet browser or an external application which accesses the lower tiers via a network. The application tier consists of an application server executing the business logic of the application. The business logic describes the rules, processes and algorithms that handle information exchange between a database and a user interface. Finally, the data management tier deals with data persistence. It usually consists of a relational database which manages the retrieval, storage and update of all data consistently.

For cloud systems, we introduce a five-tier architecture which we define as the reference architecture for applications running in the cloud (figure 3.1). Cloud systems are most of the time accessed remotely over the internet. Therefore we have to introduce a web tier to provide interface functionality between the client and the application layer. The web tier consists of a webserver which purpose is to handle and answer all requests sent by the clients. Additionally we separate the data management layer into two parts. On the top remains the database server or a query processing engine which goal is the management of data. Below we add a new tier for data storage. This layer describes the system in which data is physically stored. This can for example be a hard disc or a key-value store like S3. Even if a hard disc is not a software component and therefore not an
architectural layer in the proper sense, the distinction between query processing and data storage is necessary in cloud systems. For high availability and fault-tolerance cloud storage systems (e.g., Amazon S3) replicate the data at different physical locations but provide only one logical abstraction of the data to the upper tier. These systems have to be considered as an additional tier.

Of course several architecture layers can be executed together on one virtual machine. For example the architecture variant in figure 3.2 combines the web server, the application server and the database server on an EC2 instance. This architecture can be considered as a minimal setting for applications inside the cloud. Based on our reference architecture, many more settings can be defined. The next section suggests ways in which the reference architecture can be extended and scaled.

3.2 Architecture Variants

3.2.1 Horizontal and Vertical Scaling

In figure 3.2 the web tier, the application tier and the database tier are running on one server. This setting has limited scalability and quickly reaches its performance limit. To increase performance one possible solution is to add additional hardware resources like faster CPUs or more memory to the server. Taking advantage of such improvements is called scaling up or vertical scaling. In cloud environments vertical scaling would mean switching to faster virtual machines as for example offered by Amazon EC2.

Another way to scale a system and increase its performance is by adding additional servers to the architecture. This method is called horizontal scaling or scaling out. Horizontal scaling can happen in two ways. First, all architecture layers are placed on different machines. Figure 3.3 shows all architecture tiers on dedicated servers. This method of scaling is limited by the number of layers in the architecture. Once
Cloud Environment

Storage

Server / Virtual Machine

Application Server

Database Server

Web Server

Client

Client

Client

Application tier

Database tier

Web tier

Storage tier

Figure 3.2: Minimal Architecture running in the Cloud

each layer runs on one server no further scaling is possible.
The second way to scale out is by adding servers which provide the same functionality as already existing ones (figure 3.4). This means we duplicate one or more tiers of the architecture and distribute them on several servers. Scaling out can happen at every architecture tier. Usually computation extensive layers like the web tier or the application tier are the first to be duplicated in order to prevent them becoming a performance bottleneck for the application.

Cloud systems are designed to offer virtually unlimited scalability. Large cloud infrastructures enable to start new virtual machines just in time (e.g., Amazon EC2). We can therefore dynamically react to changing load and add new machines whenever needed. On the contrary, when the load decreased the system can be quickly downsized. Such a flexibility is rarely provided by traditional data centers. Because applications often need one consistent data state, scaling out can be limited in the database and storage layers. Data can either be replicated and kept synchronized between multiple servers or managed centrally by one server. Cloud storage services try to address this issue by offering one point of access to data and hiding the complexity of distributing or replicating data across several servers.

Depending on the cloud services used, the system architect might be restricted in the design choice. Some cloud services already specify the application architecture. For example the Google App Engine hosts the complete application stack and automatically scales if needed. The developer can not influence the system architecture.
3.2.2 Workload Splitting

As soon as an architectural layer has several servers providing the same functionality, we need a mechanism able to decide which request is forwarded to which server. This workload splitting is done by a component called load balancer. Figure 3.4 adds a load balancer between the client tier and the webservers. The load balancer receives all requests and forwards them to a webserver selected according to the configuration. We differentiate between two types of load splitting:

- Physical load balancing is based on the characteristics or status of the infrastructure. A simple load balancer would forward each request with a round-robin scheduling. More advanced load balancers gather information about the servers state in order to decide where to forward. Factors taken into account are for this decision are for example load on servers, availability of machines, number of active connections, etc.

- Logical load balancing takes properties of the request into account in the decision where to forward. This method is also often called session-based load balancing because all requests from a user are forwarded to the same server as this one keeps relevant in-memory session data. This method is also necessary when having several completely independent application stacks, a so called multi-tenant architecture.

A load balancer in front of the web tier is a typical usage scenario. Nonetheless, because scaling out can happen at every architecture layer, load splitting might be necessary between other tiers. We must be aware that introducing a load-balancer adds a single point of failure to the architecture. It is therefore advisable to scale out the load balancer as well.
or at least to have a backup load balancer which can immediately help out in case of failure.

### 3.2.3 Layer Combinations

The separation of all layers of the reference architecture into different processes is not carved in stone. It is possible to combine the functionality of two or more architectural layers in one software component. This especially makes sense if the components are highly depending on each other or if the combination promises strong performance improvements.

Most cloud providers do not publish technical details of their services, that is why we can not precisely say which services implement layer combinations or use physically separate software components. However, one example is Sausalito [1]. Sausalito is a PaaS service to develop web applications. It is based on an XQuery application server including a webserver, an XQuery parser and a data processing component responsible for reading and updating data. Thus the web tier, application tier and database tier are all combined in one component.

### 3.3 Existing Cloud Offerings

In section 2.5 we already gave an overview of the big cloud providers and their products. We now analyze how these offerings fit in the reference architecture. With existing cloud services almost all variations described in section 3.2 can
be realized. While some providers specify fixed architectures, others allow the
developer to flexibly choose and build their architecture. Accordingly, we identify
three general settings:

- **Google App Engine and Microsoft Azure** are PaaS providers and
define fixed architectures on which developers can exercise little control.
Applications are placed behind a load balancer and are automatically scaled.
The environments seem to combine the web tier and the application tier
on one server or virtual machine. Same picture for the persistent storage
which as well combines two tiers. Azure SQL and Googles Datastore cover
the database and storage tier.

- **Amazon EC2 and SimpleDB** combine two services from Amazon’s cloud
computing services stack. Amazon EC2 provides virtual machines which
can be individually set up and configured by the developer. For exam-
ple lightweight web servers (e.g., Apache or Tomcat) or entire application
servers (e.g., JBoss or Glassfish) can be run on EC2 instances. EC2 can be
configured to automatically start new instances and load balance incoming
application traffic. Scalability can therefore be ensured. Data is stored on
SimpleDB. Similar to Azure SQL and Google Datastore, SimpleDB com-
bines the database and storage layer in one service which can be accessed
from applications running on EC2 instances.

- **Amazon EC2 and a relation database** offer most flexibility to the de-
veloper in the design of the architecture. All components are run on EC2
instances. For example a Tomcat webserver with a Java web-application
acts as web and application layer. A relational database (e.g., MySQL) is
as well installed on EC2 instances. MySQL stores its data in the filesystem.
To guarantee persistence and fault-tolerance we can mount an external
EBS volume to store data. Thus, we keep the database and storage layer
separate.
Chapter 4

Benchmarking Cloud Systems

In the previous chapter we presented architectures for cloud systems. This chapter now answers the question how to benchmark cloud applications. First, we collect requirements to a cloud benchmark, then we have a look at existing benchmark solutions and analyze how well they cover our requirements. Finally, we suggest new measurements and metrics taking specific cloud properties into account.

4.1 Requirements to a Cloud Benchmark

4.1.1 Features and Metrics

Already mentioned several times, the main advantages of cloud systems are scalability, pay-as-you-go and fault-tolerance. Because cloud computing is still a young research topic with few best practices, it’s clear that these features are differently fulfilled by cloud providers.

Because cloud applications usually combine several cloud services, the ability to scale has to be analyzed on the whole architecture (e.g., limitations of one service could impact the scalability of the entire system). Additionally, differences in pricing models (see subsection 2.3.3) can lead to considerably differing cost. Prices of PaaS providers are based on CPU utilization whereas IaaS providers charge a price on a per hour basis. This leaves the question open for which usage scenario which model fits best. In trying to reach high availability, fault-tolerance is an important issue. Fault-Tolerance can have varying levels of granularity. An important indicator is for example the number of faulty services the system can resist before user notice. Additionally, the ability of a system to repair itself is of high relevance.

All just mentioned examples show that specific features (scalability, pay-per-use, fault-tolerance) are strongly relevant in the cloud. Consequently, a benchmark should test these aspects and provide appropriate metrics.

4.1.2 Architectures

Considering the increasing number of cloud providers, different services can be used and extended in different ways to achieve the same goal (see section 3.3).
However, some services are proprietary and can not be freely combined. It is therefore more important to know how different services play together rather than to know which provider is particularly good in just a single aspect. From this follows that a benchmark should be general enough to cover all possible architectures. Moreover, to take all types of cloud services into account and provide a real-world application scenario the benchmark should cover the complete application stack instead of micro benchmarking single services. Finally, we have to keep in mind, that cloud services are not comparable all the time. As indicated in section 2.4.1 cloud storage services provide different consistency levels. Achieving strong consistency in a highly distributed system is more expensive than weak consistency. To be generally applicable, a benchmark should consider different designs with different consistency levels and not force all solutions into a single setup.

4.2 Existing Benchmark Solutions

4.2.1 Benchmarks and Related Work

There exist multiple benchmarks to measure the performance of web applications. These benchmarks can be categorized in two groups. First, benchmarking tools which focus on generating load on a webserver. Most prominent benchmarks of this category are Apache Bench [6], httperf [36] and SPECWeb [46]. The second category specifies complete applications used to benchmark entire systems. This kind of benchmark is usually used to compare different programming environments or hardware settings. Examples are RUBiS [12] or PetStore [27]. Other benchmarks focus on evaluating transactional database systems. These benchmarks most of the time test databases in combination with other components on top (such as application servers or webservers). Most prominent examples are the various TPC benchmarks that define workload derived from different real-world application scenarios. The TPC benchmarks are standardized and widely accepted in the industry as well as in the research community. Among others, the best known TPC benchmarks are TPC-H for OLAP [53], TPC-C for OLTP [52], TPC-App for web-services [51] and TPC-W for e-commerce applications [50]. All the TPC benchmarks require that the system under test is deployed in a managed environment with a fixed configuration. Consequently, the primary metrics reflect the average performance of a static non changing system. Another important metric is the cost generated. The TPC benchmarks measure the total cost of ownership for a system over its lifetime.

A first attempt to develop a benchmark for the cloud is Cloudstone [45]. Cloudstone implements a web2.0 social events application and provides implementations for Ruby on Rails, PHP and Java. Persistent data is stored using a MySQL database. Using a relational database Cloudstone becomes dependent on virtual machines currently only provided by IaaS providers. To the best of our knowledge there is currently no known implementation using other storage mechanisms. As part of the related work concerning performance evaluation of cloud systems, there exists a study [20] which evaluates the different cloud services of Amazon in terms of cost and performance but does not provide a general benchmark. Another work [15] compares the cloud computing technology with current transactional database systems and proposes a list of comparison elements.
4.2 Existing Benchmark Solutions

4.2.2 The TPC-W Benchmark

The TPC-W benchmark fulfills most of the architectural requirements defined in section 4.1. The e-commerce application scenario is general enough to be implemented on most cloud infrastructures and covers the entire application stack. Additionally, the TPC-W is widely accepted and established for benchmarking. Because of these reasons we have decided to use it as a basis for a new cloud benchmark.

The TPC-W specifies an online bookstore consisting of 14 web interactions enabling to browse, search, display, update and order products of the store. The system under test (SUT) consists of an application server implementing the business logic responsible for answering every HTTP request and a persistent storage realized with a transactional database system.

In order to generate workload, the TPC-W specifies a remote browser emulation (RBE) system which automatically simulates a configured number of users sending requests to the SUT. The aim of the RBE is to realistically simulate the browsing behavior of multiple users. To issue different workloads to the SUT, the TPC-W benchmark defines three different mixes: browsing, shopping and ordering mix. A particular mix determines for every user session a workflow with a sequence of web interactions based on a varying ratio of browse and order operations. Browse operations only read data whereas order operations execute data updates. The workflows of each mix are specified by an $N \times N$ matrix $M$ which gives the probability $M_{ij}$ that web interaction $j$ will follow web interaction $i$. Benchmarking with different mixes shows the impact of a changing amount of update operations on the performance of the system.

![Figure 4.1: TPC-W Architecture](image-url)
Figure 4.1 gives an overview of the entire benchmark system as specified by the TPC-W. To generate high load the remote browser emulation is partitioned on several machines, the so called RBE clients. All RBE clients are managed by a central control unit. The control unit provides configuration settings (e.g., start time and rate of load increase) for each benchmark run and collects all results measured by the clients in order to generate a benchmark report. The SUT is accessed over a network and consists of one or more webservers to answer the requests send from the RBE clients. In our model the webserver also has the role of application server and executes the business logic of the bookstore. Both components could also run on separate servers. All application servers access and store data in a collective storage system (e.g., a relational database).

**Data Model**

To manage persistent data, the TPC-W benchmark specifies a relational data model. Figure 4.2 gives an overview of all tables one might expect from a bookstore. Arrows point in the direction of one-to-many relationships, dotted lines represent one-to-one relationships between non-key fields, and bold types identify primary and foreign keys. The TPC-W requires a strong consistency level. The storage system must provide the ACID properties and all updates must be stored consistently.

![TPC-W Database Entities and Relationships](image)

Additionally to the above-mentioned data model the TPC-W benchmark has to handle session data. Session data consists of all items added to the shopping cart. This data is temporary and lost if the user aborts his session (e.g., by closing the browser). Only when the order process is successfully completed the content of the shopping cart is stored persistently as part of a new order in the database.
In order to identify the session, the TPC-W specifies a shopping id propagated from page to page with a GET-parameter. It is left open to the developer how the session-data is stored.

Metrics

The primary metric used by the TPC-W is the number of web interactions per second (WIPS) that the SUT can handle. By scaling the number of emulated browsers (EBs), the number of requests and the load on the system is increased. This happens as long as 90% of the web interactions response times do not exceed a specified amount of seconds. In such a case the benchmark run is considered valid. Every benchmark run is executed with a fixed number of EBs so that the load does not change during the experiment. The minimum measurement interval is set to 30 minutes. The performance of a system is reported as the highest number of WIPS reached in a valid benchmark run. The second metric of the TPC-W is the ratio of cost and performance: $/WIPS. The price is based on the total cost of ownership of the system under test including software, hardware, maintenance and administration expenses (for 3 years). The total cost is divided by the maximum number of WIPS to calculate the amount of $/WIPS.

Limits

For benchmarking cloud systems using the TPC-W as it is reveals problematic. First, by requiring the ACID properties for data operations it becomes obvious that the TPC-W has been designed for transactional database systems. As already discussed earlier, cloud systems usually do not offer so strong consistency constraints. This is after all not necessary because most web-based applications only require lower levels of consistency [55]. As a consequence, existing TPC-W implementations for the cloud (e.g., [10]) are not conform to the specification. Second, WIPS is used as the main metric to measure the performance of a system. Although WIPS is useful in the context of a static system it is not for dynamic and scalable systems. In an ideal cloud computing setting an increasing load would always be compensated by new processing units added to the system and consequently the number of WIPS would never stop growing. This means it is not possible to report a maximum WIPS value and the main metric becomes useless for the cloud. Third, the $/WIPS metric divides the total cost by the maximum number of WIPS. For a cloud benchmark two problems arise: First, as just discussed no maximum number of WIPS exists in the context of cloud computing. Thus, there is also no fixed load for which the cost can be calculated. Second, different price plans and the lot-size problem prevent the calculation of a single $/WIPS value. Instead, the $/WIPS may vary extremely depending on the particular load. Fourth, the latest release of the TPC-W specification dates back to 2002. Considering the technical evolution of web-applications in the last years, the TPC-W is outdated and does not reflect modern access-paths such as those generated by Web2.0 like interactions (e.g., user generated content or AJAX). Finally, the TPC-W benchmark lacks of adequate metrics for measuring the features of cloud systems like scalability, pay-per-use and fault-tolerance.
4.3 A New Cloud Benchmark

The limits of the TPC-W underline that it cannot be used as it is to benchmark cloud systems. The TPC-W requires a static system to be executed. A Cloud benchmark on the other hand should analyze the ability of a dynamic system to adapt to a changing load. It is therefore unavoidable to change the number of WIPS during the benchmark execution. This opens the opportunity to determine how much a system can scale and what cost it generates.

4.3.1 Benchmark Configurations

The concept of mixes of web interaction sequences which directly impacts the ratio of read and update operations is useful. In order to recognize if the SUT is optimized for read or write operations this feature of the TPC-W is kept. Data consistency is another crucial aspect. A cloud benchmark should not require from the data storage system to provide the strong ACID guarantees for all web interactions. We therefore set the consistency level as a configuration property ranging from BASE (Basically Available, Soft-State, Eventually Consistent) [38] the weakest form of consistency to ACID (Atomicity, Consistency, Isolation, Durability) [14] the strongest form. We propose that the benchmark can choose between three different levels of consistency:

- **Low:** All web interactions use only BASE guarantees.
- **Medium:** The interactions use a mix of consistency guarantees.
- **High:** All web interactions use only ACID guarantees.

If the desired consistency level is not provided by a specific cloud provider, it either has to be implemented on top of the storage service or the benchmark cannot be executed.

4.3.2 Metrics

This section introduces new metrics which explicitly focus on the characteristics of cloud computing: Scalability, cost, peaks and fault-tolerance. These new metrics enable measuring dynamic aspects of the cloud.

**Scalability**

Ideally, cloud services scale linearly and infinitely. However technical restrictions, consistency requirements, price plans and physical limitations can prevent perfect scaling.

We suggest to test the scalability of a system by linearly increasing the number of issued web interactions over time and continuously count the number of web interactions answered in a given response time (RT). If all issued web interactions are answered in response time the system scales linearly. However, if from one moment on the gap between the number of issued WIPS and the WIPS in RT increases, meaning we have more and more web interactions out of response time, the scalability is no longer perfect. Figure 4.3 demonstrates this behavior. It is even possible, that if the number of WIPS in RT becomes constant or decreases, the system does not scale anymore at all.
Assuming that perfect scaling does not exist, we suggest to define the end of the benchmark as the time where the difference between the issued WIPS and the WIPS in RT exceeds a predefined limit. Because this could still take very long, we also define a minimum execution time necessary to get significant results.

As a metric for the scalability test, we suggest to measure the deviation between issued WIPS and WIPS in RT by using the correlation coefficient $R^2$. The correlation coefficient is a value between 0 and 1 where 0 indicates a constant behavior and 1 a perfect scaling. The value indicates how the measured scalability deviates from perfect scalability.

**Cost**

To measure the cost we keep the TPC-W metric of dollars per WIPS ($/WIPS$). But, instead of dividing the total cost of ownership by the maximum number of WIPS reached, we calculate the cost (per second) of the running SUT and divide it by the current WIPS rate. This measurement enables us to indicate the cost of a WIPS depending on the current load of the system. Figure 4.4 indicates a possible cost per WIPS evolution while the load increases over time. Ideally, the cost remains constant independently of the scaling factor. However, lot-sizes or changing price models can cause cost variations. Lot-sizes means a jump from one moment to another in the fixed cost caused by additional resources (e.g., EC2 instances).

Based on the results, we first calculate the average cost per WIPS for the entire benchmark run. Second, we also calculate the standard deviation of the cost during scaling. The standard deviation is an important indicator on how cost might vary and can help to better plan a system. A low value indicates perfect pay-as-you-go pricing whereas a bigger value corresponds to a more traditional non-cloud scenario.

**Peaks**

Peak load means a strong and rapid increase of the load for a certain amount of time. Peaks can regularly happen in real-world applications, and therefore adapting to peak load is an important property to test. Compared to the scalability experiment, the goal of a peak experiment is not to test if and how much a system...
can scale, but how fast it can scale.

To measure the peak behavior we start the benchmark run with a predefined base load which is strongly increased for a short time and then reduced afterwards as demonstrated in figure 4.5. Here again, we continuously keep track of the number of WIPS issued and the number of WIPS in RT. The difference between both values reflects the ability of a system to react to peak loads.

As a result, the ratio between issued WIPS and WIPS in RT is used to reflect the adaptability to peak loads. A value of 1 means a perfect absorption of the peak whereas a smaller value implies higher adaptation times.

This experiment has two configuration parameters. First, the base load. A too high base load might push the system to its scalability limit. Second, the pace of load increase. A slow increase should result in better adaptability. To get meaningful and comparable results, one should experiment with different base loads and define several increase rates.

Fault-Tolerance

Cloud infrastructure is built on standard hardware susceptible to failures. At any time part of the cloud infrastructure, perhaps even an entire data center, can
fail. Because cloud providers promise high availability and therefore replicate the application and the data, it is of relevance to test how a system reacts to failures. To measure the fault-tolerance we run the benchmark with a constant load and after a moment shutdown a specified percentage of the application. As a direct consequence we expect the number of WIPS in RT to drop and hope that the cloud system is able to automatically repair itself (figure 4.6).

Figure 4.6: Measurement of Fault-Tolerance

Here again the ratio between WIPS in RT and issued WIPS is calculated to indicate the systems ability to recover. In contrast to the previous metrics, the failure behavior is most likely only reportable by cloud providers and not cloud users.
Chapter 5

Benchmark Implementation

In the last chapter we have specified a benchmark for the cloud. This chapter deals with the implementation of this benchmark. Based on an existing TPC-W implementation we modify all elements to satisfy the requirements for a cloud benchmark and finally implement the benchmark on several cloud architectures.

5.1 TPC-W Reference Implementation

The choice to base our benchmark on the TPC-W allows to select an existing implementation and to adapt it to our needs. We looked for a solution which already contains the major components of the TPC-W like image generation, data population and running implementations of the RBE and the SUT. After a short evaluation of available implementations we decided to use [28] as a basis for further development.

The solution fulfills all requirements and is implemented in the Java programming language. The SUT is realized with the Java Servlet technology and can be easily deployed on a Java application server (e.g., Apache Tomcat). Persistent data is managed by a MySQL database. An existing data population program takes care of importing sample data into the database before the benchmark can be executed. The RBE is as well realized as a Java program and is optimized to work together with the SUT implementation.

In a first step we check the implementation against the official TPC-W specification to see if or how far it is fulfilled. The following aspects do not correspond to specification version 1.8 [50]:

- The buy confirm web interaction does not include an interface to an external payment gateway emulator (PGE). The PGE is specified for validating credit card data and is not implemented in the reference implementation.

- The database population program does not use the DBGEN utility for generating author names and book titles. Instead a workaround is implemented. All names and titles are generated as random strings.

- The buy process is not executed with an encrypted communication. Pages are not transferred using the HTTPS protocol and thus no SSL-Certificate is needed.
• Additionally, we modify the bestseller query. The bestseller query displays the top 50 bestseller items of a specific subject on the bestseller web interaction. The bestsellers are recomputed at every page call based on all bought items in the top 3333 latest orders. For a relational database this is a very computation intensive operation which strongly influences the performance of the whole system. As a simple workaround we display 50 random items on the bestseller page. An elegant solution would be to generate the bestsellers in regular intervals and store the items in a temporary table or an in-memory cache.

5.2 Architectures

5.2.1 Relational Database running in the Cloud

Components and Cost Factors

The first system we implement is based on a traditional architecture we transfer into the cloud. A setting composed of an application server and a relational database which usually runs on dedicated servers is now executed on rented virtual machines of an IaaS provider. Amazon’s EC2 infrastructure is used for this purpose. Figure 5.1 gives an overview of the architecture components. The figure does not dictate a fix physical implementation but only illustrates the logical components. This means that the application server as well as the database can be scaled out if technically possible. For automatic scaling Amazon EC2 provides a feature called Auto Scaling. Based on CPU utilization new instances can be automatically started or shutdown.

The famous Tomcat application server is used to execute the TPC-W web application provided by the reference implementation. The application server connects to the open source and widespread MySQL database. To guarantee persistence the database data and the binary log are stored on an EBS volume. MySQL provides strong consistency guarantees and fulfills the ACID properties.

Figure 5.1 additionally makes aware of the different cost factors. Provided that the system is not scaled out, the cost for instances are constant. The amount of data transfer and the EBS cost vary according to the load on the system.

This architecture exemplifies how existing architectures can be moved into the cloud without great effort. This is of significance because migrating large software architectures can be very expensive and time consuming. Therefore, using
5.2 Architectures

standard components (e.g., relational databases) in the cloud can help save cost and take advantage of existing know-how.

A recent study [35] has analyzed the performance impact of running database systems on virtual machines instead of physical servers. The researchers found out that the average overhead of virtual machines is less than 10%, which means that a database is slightly slower on a VM than on a physical machine.

Implementation

This architecture is already implemented in the reference implementation. However, it is necessary to refactor a part of the source code in order to reuse it in further architectures.

In a first step, the Java Servlets are refactored. The non valid HTML code is replaced by modern XHTML code valid according to the XHTML 1.0 specification. All deprecated Java SDK calls are replaced by their current alternative. The internal link generation is no longer hard coded but instead handled by an ENUM type. ENUM types allow a mapping from a key to one or several primitive types and are therefore convenient to manage the mapping from page names to URL paths for a small set of elements. Additionally, several tweaks to improve performance are implemented (e.g., use of javax.servlet.ServletOutputStream for output, singleton objects instantiated in the javax.servlet.http.HttpServlet.init() method)

In a second step the interface to the database is completely revised. As a basis for all data access, we create an interface with a method for every data transaction in the whole application. Because of this interface, implementing the access to several storage services becomes much easier. Nonetheless, in this architecture we confine ourself to implement the storage interface for the MySQL database. Much of the code can be taken over from the reference implementation. To establish and maintain the connection with the database the connection pooling feature from the Tomcat server is used.

All data accesses are realized with prepared statements. A prepared statement sets up a query once and does not need to be parsed again when repeated. That is why repeating a prepared statement a lot of times with changing parameter brings a performance advantage compared to classical queries. As well prepared statements separate data from logic and therefore make the application less vulnerable for SQL injections.

Third, a new feature for measuring performance is added to the application. We implement a generic performance class to measure time intervals for a sequence of operations inside a method. Several measurements at different places in the application can be executed concurrently. The performance class always keeps track of the running intervals by identifying the methods through the Java stacktrace. Measurements are enforced for all Servlets (page generation time) and for each executed transaction on the storage (transaction time). The measures are appended as a HTML comment to the HTTP response and can thereafter be processed by the RBE.
5.2.2 Amazon SimpleDB

Components and Cost Factors

Amazon SimpleDB is a cloud storage service and intends to combine cloud advantages with core database functions. Compared to the previous architecture, the MySQL relational database is in this setting replaced by SimpleDB (figure 5.2). The application server which takes care of the business logic remains the same but now uses the SimpleDB API to access and store data. SimpleDB provides a persistent storage, so that we no longer have to take care of manually creating and mounting EBS volumes for data storage. Additionally, the administrative effort to set up and monitor virtual machines running a database is no longer necessary. Amazon takes care of the availability and scalability of the service. However, as a counterpart, we have to accept that SimpleDB only provides eventual consistency.

Figure 5.2: Amazon SimpleDB Architecture Components and Cost Factors

SimpleDB charges all processed queries by the amount of CPU time they needed. Additionally, a small amount is charged for every GB of storage capacity used.

Implementation

The access to SimpleDB is implemented as an extension to the existing Servlet application already used for the MySQL Architecture (see subsection 5.2.1). We use an implementation of the Factory design pattern [17] to switch between different storage services. The selection of the storage service is made through a configuration entry in a properties file. Using this setting the application automatically selects the right storage system.

The relational model of the database is ported one to one to the domain based model of SimpleDB. Each relation becomes a domain in SimpleDB. SimpleDB provides a simple API with a query language similar to SQL. This significantly reduced the effort necessary to implement the storage interface for SimpleDB. However, we also encountered several obstacles.

First, every domain entry has to be identified by a unique key, the itemname. Unfortunately, Select and Query operations do not allow predicate comparison on itemnames. This makes impossible to retrieve several tuples by itemname in one API call. A workaround is to additionally add the value of the itemname as an attribute to the item and use it for predicate comparison in queries. This is
5.2 Architectures

an inconvenient solution, as the key must always be inserted twice, as itemname and as attribute. Meanwhile Amazon has fixed this problem [5].

Another restriction is that auto-increment values are not supported by SimpleDB. They can even not be implemented consistently because no locks or transactions are supported. For this reason we use UUID strings as primary key to identify items.

Additionally, SimpleDB behaves unexpected in some situations. For example a COUNT operation on a domain with several hundred thousands entries results in a HTTP timeout without any feedback, not even a server side error message.

5.2.3 Google App Engine

Components and Cost Factors

Google App Engine is a PaaS platform. This means the entire environment is managed by the provider. The availability and scalability of the web application and the stored data is handled autonomously by Google and the developer has no maintenance tasks to carry out. Internally the App Engine distinguishes between two components, the runtime environment which contains the business logic and the Datastore which is a persistent storage system probably based on MegaStore. The Datastore, although it is a distributed storage system, provides strong consistency guarantees up to the *serializable* isolation level. Transactions are supported and enable rollbacks in case of failure.

![Google App Engine Architecture Components and Cost Factors](image)

Most of the cost charged for using the App Engine is load dependent. Both, the load on the runtime environment as well as on the Datastore is charged for CPU time used during request processing. Data transfer cost is determined by the amount of data transferred to and from the Google infrastructure.

Implementation

The developer has limited access to the infrastructure which hosts the application. An App Engine application is usually developed and tested on local computers and then deployed to the infrastructure. For the deployment process Google provides an App Engine SDK. The SDK also includes a test environment and useful helper functions.
Because of our existing Java implementation of the TPC-W, a logical consequence is to use the Java version of the App Engine. With the help of the App Engine Eclipse plugin we create a new App Engine project with a predefined structure. This structure already contains all necessary configuration files (e.g., to handle versioning and data indexes) for deployment.

The Java Servlets of the reference implementation are ported one to one to the App Engine project. The storage interface is reimplemented to access the App Engine Datastore. The App Engine provides a JPA implementation which realizes an object relational mapping. With the Google Query Language (GQL), a language syntactically similar to SQL, objects can be retrieved and updated on the Datastore. However, because several query operators are not yet implemented a lot of unnecessary queries have to be send to the storage service (see section 6.2.4 for further details).

5.2.4 Database Library built on S3

Components and Cost Factors

The Database Library is a framework developed in a research project at ETH Zurich. The aim of the library is to increase the level of data consistency while placed on top of a key-value storage system which only provides eventual consistency (e.g., Amazon S3). Based on database technologies and with the help of a Queue service to manage pending updates and locks, the Database Library can provide a variable consistency guarantee ranging from eventual consistency up to serializable. Explaining the theoretical background and the implementation of the Database Library is not in the scope of this thesis, we refer the reader to [10] for further details.

In the architecture the library is placed in between the application server and the storage system. In the setting in figure 5.4 the Tomcat server and the Database Library are run on the same instance. For the storage of data we use Amazon S3. The queue service runs on a separate instance and uses an EBS volume for the persistent storage of queues and logging information. The queue server has been implemented as part of the Database Library and can be used as it is. Both the Tomcat server and the queue server can be scaled out if necessary.

A major advantage of the Database Library is that it provides a local cache for all data retrieved. This minimizes the accesses to the storage system as the library always looks in the cache first before a cost intensive read operation on S3 is initiated. Cached data is invalidated after a certain amount of time and retrieved from persistent storage the next time it is queried.

Cost is charged for all services used. The Tomcat and queue servers are run on EC2 instances which are charged per hour. The S3 and EBS cost is calculated based on the storage space used and the number of I/O operations. Last but not least, common to all architectures, we have to pay for in- and outgoing data transfer.

Implementation

The Database Library provided by the ETHZ Systems Group is implemented in C++. In order to work together with the Java Servlet implementation we have to communicate across programming languages. We decide to use Thrift [7] for this purpose. Thrift is a software framework for cross-language services development.
Thrift allows to define data types and service interfaces in a simple definition file. Taking that file as input, the compiler generates code to be used to easily build RPC clients and servers that communicate seamlessly across programming languages.

The Database Library uses its own low level page-oriented storage format. In order to store the TPC-W data, the data model with all entities and attributes has to be implemented in the Database Library. Additionally, all data access operations and the indexes have to be programmatically set. On the application server side, we implement the storage interface. Each transaction is completed by one or several RPC calls to the Database Library through Thrift.

5.2.5 Sausalito XQuery Application Server

Components and Cost Factors

Sausalito is a PaaS service that allows to implement and deploy web applications, coded entirely in XQuery [56]. XQuery is a query and functional programming language that is designed to query collections of XML data. In Sausalito everything from the request dispatching to the data access is programmed in XQuery. Similar to the Google App Engine the developer has limited access to the hosting environment and can deploy the application from the local computer to the Sausalito infrastructure. Behind the scenes, the setting is similar to the one of the previous architecture (figure 5.4). Instead of running on a Tomcat server Sausalito uses an own webserver on which runs an XQuery engine. For the storage of persistent data Sausalito uses a similar mechanism than the Database Library (section 5.2.4) and stores the collections on Amazon S3.

A specialty of Sausalito is that it runs on top of an existing cloud provider. It is a cloud service inside another cloud and can therefore be seen as another variety in our cloud architecture continuum.

Implementation

For this architecture we have to reimplement the entire TPC-W benchmark in XQuery.

Basically, the operation principles of Sausalito are very simple. A request to
a specific URL calls an XQuery function. This function contains an XQuery statement which returns the HTML code to be displayed. For more complex web applications however we should rely on established development methods. That is why we implement the TPC-W using the Model-View-Controller pattern (MVC) [17]. The View contains the HTML template, the Model handles the access to the data and the controller coordinates the processing and contains the business logic. The HTML code used in Sausalito is XHTML valid. This means it is an XML data format on which XQuery can take advantage of it’s powerful processing ability.

For further information, we refer the user to the Sausalito Development Guide [2]

5.2.6 Comparison

In this subsection we want to wrap up all implemented architectures. Table 5.1 points out the most important aspects of all architectures.

5.3 Remote Browser Emulation

5.3.1 Reference Implementation and Refactoring

The RBE is responsible for sending requests to and analyzing responses from the SUT. The goal is to stress the SUT in order to determine it’s peak performance. For a most realistic behavior, the RBE simulates the browsing behavior of real web users, also called emulated browser (EB). Each EB is implemented as a thread in the RBE program. Threads allow to run tasks concurrently on one machine and at the same time share resources. This means that the RBE can start a large amount of EBs which will run mostly independently from each other. The only shared resources in our reference implementation are objects to store the measurements. However, to prevent race conditions or even deadlocks, the access to shared resources has to be organized with great care. All accesses have to be atomic, meaning that only one thread can execute a method or a sequence of operations at a time. The easiest way to implement this behavior in Java is to add the synchronized keyword to the method declaration. This way is used by the reference implementation. The RBE instantiates all EBs during benchmark initialization. After all preparations are completed the threads are started and begin sending requests to the SUT. For the duration of the measurement the main thread is suspended. After being reactivated at the end of the measurement interval all EBs are terminated and the benchmark results generated.

All just described functionality is already provided by the reference implementation. Nonetheless, to get a deeper understanding of the functionality we refactored the code. During refactoring we structured all objects into packages, remove deprecated Java SDK calls, introduced a logging with log4j and added a lot of comments to make the source code easier to understand.

Another refactoring activity is to extract all configuration parameters into a properties file which is read at the start of the RBE program. This strongly increase flexibility as no code changes are necessary anymore to customize a benchmark run. The following list indicates the most important configuration properties of the RBE:

- MIX is the name of the web interaction mix to be executed. The mix
<table>
<thead>
<tr>
<th>Type</th>
<th>MySQL</th>
<th>Amazon SimpleDB</th>
<th>Google AppEngine</th>
<th>Database Library</th>
<th>Sausalito</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specific property</td>
<td>Tomcat and database on VMs</td>
<td>Tomcat on VM and cloud storage service</td>
<td>Cloud application and cloud storage service</td>
<td>Tomcat on VM with local cache and cloud storage service</td>
<td>A cloud service on top of another cloud</td>
</tr>
<tr>
<td>Application server</td>
<td>Tomcat</td>
<td>Tomcat</td>
<td>Java Environment</td>
<td>Tomcat</td>
<td>XQuery Environment</td>
</tr>
<tr>
<td>Storage service</td>
<td>MySQL</td>
<td>Amazon SimpleDB</td>
<td>Google Datastore</td>
<td>Amazon S3</td>
<td>Amazon S3</td>
</tr>
<tr>
<td>Consistency Level</td>
<td>Read Committed</td>
<td>Eventual Consistent</td>
<td>Snapshot Isolated</td>
<td>Snapshot Isolated</td>
<td>Eventual Consistent</td>
</tr>
<tr>
<td>Data Caching</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AppServer Scalability</td>
<td>Manual/Auto</td>
<td>Manual/Auto</td>
<td>Auto</td>
<td>Manual/Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Storage Scalability</td>
<td>Manual</td>
<td>Auto</td>
<td>Auto</td>
<td>Manual/Auto</td>
<td>Auto</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of Architectures
determines if the benchmark run is more read or more update intensive.

- **RU** is the ramp up time in seconds. This time is used for the system to warm up (e.g., load data into cache). During ramp up load is put on the SUT but the measurements are not counted.

- **MI** is the measurement interval in seconds. This property indicates how long the benchmark run is executed.

- **GETIM** is a boolean value which indicates if or not the images inside web interactions should be retrieved.

- **URLs** is a group of configuration properties which indicates how the SUT can be reached. **baseURL** indicates the root URL whereas the other properties indicate the access paths for each of the 14 TPC-W web interactions.

### 5.3.2 Dynamic Load Generation

In order to measure our new metrics defined in section 4.3.2 the RBE implementation has to be modified. The reference implementation is programmed to execute the non changing static load of the TPC-W. For the cloud, we want to dynamically adapt the load during a benchmark run. This means that while the benchmark runs, new EBs have to be created or destroyed at defined moments. We explain the implementation of this behavior with three new configuration parameters used to control it:

- **EB_MIN** is the minimum number of EBs. The benchmark run is started with **EB_MIN** threads.

- **EB_MAX** is the maximum number of EBs. This value is used to instantiate all threads before the measurement begins.

- **INCREMENTAL** defines the number of seconds between the start of two EBs. In other words a new EB is started every **INCREMENTAL seconds**. The number of EBs is increased from the start of the measurement time on until the benchmark run is over or the maximum number of EBs is reached.

Another requirement for the RBE is the ability to send requests to several web-servers. This is required if the SUT does not have a load balancer which distributes the requests across all web servers. This feature is realized by simply configuring the **baseURL** parameter with a space-separated list of the URLs of all web servers. The RBE client parses the URLs with a `java.util.StringTokenizer` and stores them in a `java.util.ArrayList`. Every time an EB is instantiated it chooses one URL randomly and keeps it during its whole existence. This guarantees a homogeneous distribution of the EBs between all SUT web servers.

### 5.3.3 Measurements and Results

During the entire benchmark run, information about each single web interaction send to the SUT is collected. This contains, among others the type of web interaction, the response time, the time the request was issued, the amount of data transferred and the think time. Some performance measurements are also made on the server side (e.g., page parse time and execution time of storage transactions).
These information are included as a comment in the HTML-Code delivered by the SUT and parsed by the RBE to be stored together with the other measurements. The reference implementation contains a restricted mechanism to generate Matlab scripts out of the measurements. To facilitate the interpretation of results, we reimplement the collection of measurements and the output of results with a more generic approach. In the new solution a result output interface is called after the completion of a benchmark run. This interface is currently implemented for the output of CSV files. CSV files can be easily imported in spreadsheet programs like Microsoft Excel. Nonetheless, if needed the interface offers the possibility to implement further output formats (e.g., write results in a database or directly generate diagrams). The following results are generated as CSV output:

- **Interactions**: This is the direct output of all collected interaction data in CSV format.

- **Throughput**: The throughput considers the number of WIPS issued, the number of WIPS in response time and the traffic generated for a small time interval (e.g., 30 seconds). This data is of great interest to see the performance evolution over time. The number of WIPS is the sum of all web interactions in a time interval divided by the size in seconds of that interval.

- **Response Time**: Based on all interactions of a benchmark run, we calculate the minimum, average, maximum and 90 percentile response time for each of the 14 web interactions of the TPC-W. Because we increase the number of EBs over time, calculating results which consider the entire run has lost most of its relevance. However, we decide to keep this information as it still might be used for benchmark runs with a fix number of EBs.

- **Cost**: Some cloud services (e.g., Amazon SimpleDB) include the amount of used CPU time in the response of each API call. The SUT appends this information to the HTML response. With this data, we can calculate parts of the cost generated during a benchmark run.

### 5.3.4 Distribution and Result Collection

In its existing implementation the RBE client is designed to be executed on a single machine. However the number of EBs a server can handle is limited. Each EB opens several streams to retrieve data from the SUT. The higher we increase the number of EBs the higher becomes the number of parallel open streams. The Linux operating system considers every open stream as a open file and refuses to open new streams if a preset number of open files is exceeded. This too many open files problem throws an IOException and prevents the EB to correctly execute a web interaction. This error has fatal consequences, as the entire benchmark results become corrupted.

As we wish to run our benchmark on systems with default settings, we decide not to change the configuration for open files but instead define a maximum number of EBs which can run on one server without causing problems. For medium EC2 instances we set this limit to 1500.

In order to further increase the load and reach a higher number of EBs, the RBE has to be executed on several servers concurrently. A distribution of the
RBE brings two new challenges. On the one hand, all RBE clients have to be synchronized in order to correctly execute the benchmark run (e.g., common start time and EB increase rate). On the other hand, all gathered results have to be either centrally collected or merged afterwards.

We solve the first challenge by adapting the RBE execution logic and introducing two new configuration parameters:

- **TIME** defines the start time of the RBE client. This value makes the benchmark run independent of the startup time of the RBE program. In order to work across servers and synchronize the start of the run, all servers require to have the same system time.

- **MULTIPLIER** indicates how many RBE clients are involved in the benchmark run. This is of relevance for a correct EB increase rate. If for example we want to start a new EB every second (parameter INCREMENTAL set to 1) while using three RBE clients, every client has to start a new EB every three seconds. \( \text{MULTIPLIER} \times \text{INCREMENTAL} \) provides the numbers of seconds after which each client starts a new EB.

The deployment of the RBE clients is handled by a script which compiles the project, creates a jar archive, copies this archive to all RBE servers using `pscp` and finally starts the RBE clients with a parallel ssh call using `pssh`.

The second challenge is solved by centrally storing all results in a database instead of local in-memory objects. We extend the RBE client to store all collected data into a MySQL database run on a remote server. For all access we use the transaction mechanism provided by the *InnoDB* storage engine. Transactions guarantee that all results can be written concurrently by all RBE clients and that data still remains consistent. After the benchmark run is completed, an additional program reads all data from the database and generates the result files according to the RBE configuration.

### 5.4 Data Population

Data Population is the process of generating and importing sample data into the data store before benchmarking. This process is handled by a stand alone Java program provided by the reference implementation. We extend this original version to import the data into all storage systems of all architectures.

To speed up the import, we modify the data population to be executed multi-threaded. This is implemented with the consumer-producer pattern. The main thread generates all data and puts it in a concurrent list. The import threads, which all have a connection to the storage system, retrieve the data from the list and send it to the store concurrently.

### 5.5 Testing

To test all TPC-W implementations on the different architectures we use Selenium [42]. Selenium is a testing framework for web applications. It provides integration tests by opening a sequence of URLs and checking if the content contains predefined patterns. Selenium provides libraries for several programming languages.

We implement the Selenium tests with Java as JUnit test cases.
5.5 Testing

The selenium test cases have the goal to check if an implementation works correctly and also to guarantee that all implementations have the same functionality. This is of importance as we want to ensure equal benchmark conditions for every architecture. In total we wrote 46 test cases to test the following functions of the TPC-W implementation:

- All pages are available and loaded correctly
- Parameter validation works
- The shopping cart works as specified. Items are correctly added, changed and removed.
- The search works and finds results
- The buy process is executed correctly and orders are stored persistently
- Items can be updated
- Promotions are available on all specified pages
- Dateformat is correct
Chapter 6

Experiments and Results

In this chapter we describe our benchmark setup and present the results collected in the benchmark runs. Because of time constraints we renounced to benchmark all implemented architectures. Instead we decided to compare the most traditional architecture using the MySQL relational database with the services of the big cloud players: Amazon SimpleDB and the Google App Engine.

6.1 Benchmark Setup

This section contains a description of the experiments and some general guidelines and common settings for all executed benchmarks.

6.1.1 Experiment Setup

Scalability

To test the scalability of different cloud architectures we linearly increment the load over time and look how many of the issued web interactions are answered in a valid response time. As described in subsection 5.3.2 the load is increased by starting new emulated browsers (EB). Every EB is a running thread which issues an unlimited number of requests to the application. After having received a response to a requested web interaction the EB waits a think time specified by the TPC-W before sending another request.

For the experiment we define a start-rate of one new EB every 0.4 seconds. This means that after one hour of benchmarking we have reached an approximate number of 9000 EBs. We therefore define 9000 EBs as the maximum load and set the default duration for a benchmark run to one hour.

The projected cost for such a long benchmark run can become very high for several architectures, we therefore decide in some exceptions to modify the experiment and opt for shorter measurements with a fixed number of EBs. In such a situation the benchmark is executed with loads of 100, 500, 1000, 2000, 4000 and 9000 EBs for 5 minutes each.

The main metric of this experiment is the number of valid WIPS. A web interaction is considered valid if it is executed in a defined response time. As a reference we use the response times defined by the TPC-W benchmark for the 90% WIRT constraint.
During the complete benchmark run we measure the throughput by counting the number of issued and the number of valid web interactions during time intervals of 30 seconds. After the first 30 seconds are over we start to count from zero on in the second 30 seconds time frame and so on until the end of the run. The current amount of WIPS is then calculated by dividing the number of valid web interactions inside each frame by the frame size.

**Cost**

Ideally, the cost per WIPS should remain constant independently of the scaling factor. However, because of lot-sizes and different pricing models the cost might vary as the load changes. As a preliminary work we first need to identify all vendor specific cost parameters for each architecture. Based on this data and the results measured in the scalability experiment we calculate the total cost per day for different amounts of WIPS and the average cost per WIPS for each architecture. Both metrics might be important indicators on how the cost varies and can help identify cost differences between the involved systems. All cost is measured in USD ($).

**6.1.2 Remote Browser Emulation**

The basis of the benchmark is the remote browser emulation (RBE) client which is responsible for stressing the SUT. In order to scale and reach a high number of EBs we execute the client concurrently on several servers. For easy scale, low cost, high bandwidth and on-demand need for resources we decided to run the RBE client in the cloud on EC2 instances. We took a publicly available AMI with a minimal installation of Ubuntu 8.04 and choose to run it as medium typed instances. Medium instances provide 1.7 GB of memory and 5 EC2 compute units. According to Amazon “one EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor”. The cost amount to $0.20 per instance-hour. On a medium instance, the RBE client is able to run up to 1500 EBs. If a request sent by an EB times out or the SUT returns an internal server error an exception is thrown and no traffic is counted. To compensate this behavior we assume that for every web interaction out of response time a static error page with a size of 5000 Byte is sent by the server. This is relevant for the traffic cost generated.

**6.1.3 System under Test**

For several architectures we have developed a benchmark implementation running on a Tomcat webservice (see section 5.2). As these architectures need or use additional AWS services (e.g., EC2 instances or SimpleDB) the Tomcat servers will run on medium EC2 instances as well. The Tomcat version used is 6.0.18 which was the most recent version at time of development. The webservice was reconfigured to listen for requests on the HTTP port 80. Moreover we have kept the default configuration which runs a maximum of 200 threads for the processing of requests and has a connection timeout of 60 seconds. Both benchmark components, the RBE server and the Tomcat webservice, even if in different data centers, are running on the AWS infrastructure. We are aware
that such a setting might not reproduce the real behavior of a web application because of a highly reduced latency. Nevertheless it enables us to push the application to its limits and prevents that bandwidth limits have an impact on the test results. In several control samples, we always reached a bandwidth higher than 30 MB/s between different Amazon data centers (in the same availability zone). Another advantage of this solution, is that we can install several benchmark components (e.g., RBE client, Tomcat server, Database Library) in one virtual machine. This increases flexibility because one AMI can now be used for multiple purposes. Additionally we can make the benchmark publicly available at a later moment.

6.1.4 Data Storage and Scale

Binary and static data like images is stored in the filesystem. All other data is stored persistently by a storage service or a database. This is of importance as for example EC2 instances do not provide a persistent storage. As soon as the instance crashes or is shutdown all changed data since startup is lost. It is obvious that such a behavior is not acceptable for practical use. To overcome the problem the data must at least be replicated across instances or stored on a persistent storage service like EBS. Because each benchmarked architecture uses a different storage service, details are given in the corresponding upcoming sections. The initial data before the benchmark start is imported as specified by the TPC-W with a scale of 10,000 items and 100 EB. This results in an amount of 10,000 items, 2,500 authors, 92 countries, 288,000 customers, 576,000 addresses, 259,200 orders and credit card transactions and 777,600 orderlines. We refer to this data scale as the default scale.

6.1.5 Benchmark Web-Interaction Mix

The benchmark web interaction mix used in the experiments corresponds to the TPC-W ordering mix (WIPS0). This mix executes to equal parts browsing and ordering operations. We choose this workload because of its high rate of update operations which we are persuaded reflects best current web applications (see subsection 4.2.2). In average 36.55% of all web interactions need at least one update operation on the storage system.

6.2 Results and Analysis

In this section we will explain the architecture setups and present the benchmark results.

6.2.1 Data Population

The data population is a prerequisite for the benchmark to run. In the default scale more than two millions data sets have to be imported or uploaded into the storage system of each architectures. All this data is generated by a specific Java program (see section 5.4) which, if possible directly inserts the data in the corresponding storage system (e.g., MySQL and SimpleDB). For Googles App Engine the data is written to CSV files first and than processed later by the App
Engine SDK. In the following we give a brief summary of the population process of each architecture.

- **Google App Engine**: Data population is executed with the *batch import* function of the App Engine Python SDK. The batch import reads CSV files and writes their content into the Datastore. Data is sent at a speed of 20 entities per second.

- **Amazon SimpleDB**: Data is imported using the *BatchPutAttributes* operation which allows to insert up to 25 data sets in one request. The population is executed using two concurrent threads which together almost double the import speed. Further increasing the number of threads results in a *ServiceUnavailable* exception which according to Amazon is thrown due to resource limitation [31]. The population is executed from an EC2 instance in order to save traffic cost.

- **MySQL Database**: MySQL is accessed over the JDBC interface. The import is executed locally on the database server with 5 concurrent threads. Because of repetitive queries we are using prepared statements to improve population speed.

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>App Engine</th>
<th>SimpleDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>10min</td>
<td>30h10min</td>
<td>1h50min</td>
</tr>
</tbody>
</table>

Table 6.1: Duration of Data Population

Table 6.1 shows that, compared to the other systems, the App Engine takes massively more time to import data. This seems obvious considering the maximum import speed of 20 entities per second. The problem becomes even worse when importing smaller data sets because the ratio of imported Bytes per second is much lower. Another important drawback is the current absence of functionalities to import larger amounts of data with the App Engine Java SDK.

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>App Engine</th>
<th>SimpleDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Hours</td>
<td>-</td>
<td>11.61</td>
<td>4.76</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>0.00</td>
<td>0.08</td>
<td>0.00</td>
</tr>
<tr>
<td>EC2 Instances</td>
<td>0.10</td>
<td>-</td>
<td>0.40</td>
</tr>
<tr>
<td><strong>Total Cost</strong></td>
<td><strong>0.10</strong></td>
<td><strong>11.69</strong></td>
<td><strong>5.16</strong></td>
</tr>
</tbody>
</table>

Table 6.2: Cost of Data Population (in $)

Table 6.2 presents the total cost for the entire data import. The cost is calculated without taking into account free quota on SimpleDB and the App Engine. Interesting is the different amount of CPU time consumed by the App Engine and SimpleDB. While the App Engine consumed 116.08 compute hours for the data import, SimpleDB only needed 33.97. An explanation could be that the App Engine sends the data to a previously deployed Python script which handles the data storage. As a consequence Google charges CPU time for processing the request and for the storage of the data itself, whereas SimpleDB only charges the second
part. The data import points out how difficult the estimation of compute hours is and underlines the differences between cloud providers in the measurement of computation time.

Data transfer is not the dominant cost factor in this population. However, when dealing with large amounts of data it must not be underestimated. Each GB of data transferred to cloud providers cost at least $0.10. Transferring GBs or even TBs to the cloud can quickly sum up to large amounts.

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>App Engine</th>
<th>SimpleDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage consumption (in GB)</td>
<td>0.81</td>
<td>4.28</td>
<td>1.32</td>
</tr>
<tr>
<td>Cost per month (in $)</td>
<td>0.10</td>
<td>0.64</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Table 6.3: Storage Consumption after Data Import

The raw size of the default scale data before import is approximately 310 MB. In table 6.3 we can observe the storage consumption after a successful import. Although we imported the same amount of data in each system, we observe some differences in the storage consumption between each storage service and also compared to the initial data size. Reasons are that the space for data indexes, metadata and data replication may be included in the calculation. Here again the cost is calculated without regard of free quota. For the MySQL database we assume that the data is stored on an EBS drive. Because EBS volumes can only be created in GB steps, we use the price of a 1 GB volume for comparison reasons. In a production environment the volume should be much larger to have enough space for future growth.

### 6.2.2 MySQL Database in the Cloud

The first benchmarked architecture uses a traditional relational database running on VMs inside the cloud.

**MySQL Setup**

Identical to the Tomcat webservers the MySQL database servers are also setup on EC2 instances. For every setting we take care that both, the tomcat and the MySQL server are launched in the same data center to make sure they are connected via a low-latency LAN network. The Tomcat servers are configured as mentioned above (see 6.1.3). The database servers use version 5.0.51 of MySQL. Although there is a newer 5.1 release of MySQL we opted for the older 5.0.x branch as this one is still provided as default by most Linux distributions. Data is stored using the transactional InnoDB storage engine. InnoDB uses repeatable read as the default transaction isolation level which we decide to keep for our benchmarks. As a reminder, repeatable read means that all consistent read operations within the same transaction read the snapshot established by the first such read in that transaction. Thus this consistency level is susceptible to phantom reads. Some MySQL configuration parameters are modified for higher performance. Especially several buffer and cache sizes are changed based on the advice of several MySQL experts [58, 41].
Benchmark Settings

To start our experiments we define a setting which consists of 7 medium-sized instances. Six Tomcat webservers to process the requests sent by the RBE clients and one dedicated MySQL server responsible for delivering and storing data. Because an EC2 instance has no persistent storage (all changes made during instance lifetime are evicted at shutdown) we mount a 30 GB EBS partition and configure MySQL to store the data and the binary log at that location. We consider 30 GB as enough because the initial data import with the default data scale only takes 0.8 GB and leaves much space for additional data.

In order to see if and how much MySQL is able to scale in terms of performance we increase the number of database servers to three in a second experiment. This scenario will use the replication functionality of MySQL. One server will have the role of master and handle all update operations, whereas the two remaining servers will become slaves and thus be read-only. Data updates are propagated automatically by the master server. Each slave server is assigned to three tomcat servers respectively. As data is replicated across several machines we are no longer obliged to persist it anymore and therefore store it in the EC2 instance filesystem. A small EBS volume is still mounted on the master server to store the binary log. The binary log guarantees the restoration of a consistent system state after a crash of the master server.

In version 4.1.3 MySQL also introduced a clustering solution. This solution uses a new storage engine which automatically synchronizes all data between several replicas. A great advantage of such a solution is that a crash of one server does not affect the general availability of the system. Unfortunately, our first performance tests were so disappointing that we no longer considered using this solution in our benchmarks.

During the benchmark runs we observed a variation in the performance of the EC2 instances. To further investigate on this behavior, we analyzed the EC2 I/O and CPU performance. The results can be found in section 6.2.6. To get meaningful and reproducible results for experiments using EC2, we decide to repeat each experiment 7 times. For each new run the database servers are freshly setup. This guarantees that all benchmark runs are executed with the same initial configuration.

Scalability

In our first experiment the dedicated MySQL server seems to perform quite well. Figure 6.1 shows the average performance of the 7 benchmark runs. We can observe that the number of WIs in response time scales linearly with the number of WIPS issued until a peak limit of 477 WIPS is reached. The linear increase is a predictable behavior as the increasing number of EBs over time directly cause a similar increase rate for the WIPS issued.

After the peak performance is reached, we observe a small but steep fall down in the WIPS in response time from 477 to 430. This is a usual behavior which indicates that a system under load has reached its performance limit. The fact that the number of valid WIPS remain more or less constant for the on going of the experiment is interesting. It shows that the MySQL server is still able to maintain a high level in query processing although it is overwhelmed with more and more requests. Only towards the end we can see that the high number of WIPS issued leads to a decrease in the number of valid WIPS.
One might ask why the number of issued WIPS does not continue to increase linearly after the peak performance is reached. This is related to the increase in the overall response time. If the response time increases every EB has to wait longer before getting a response and can therefore not issue a new request. These longer intervals between each web interaction are compensated by the still increasing number of EBs but the number of issued web interactions does not increase linearly anymore.

If we compare the results of the dedicated MySQL server with the MySQL replication on three servers in figure 6.2 one might be disappointed. Indeed the performance of this second experiment is a bit lower than the first one. We reach a peak number of 455 WIPS with one master and two slave servers. This might indicate a problem common in the scalability of traditional relational databases. The high update rate limits the performance of a multi-server database replication. Each update has to be synchronized with the slave servers generating an overhead which completely compensates the higher computation power. In case of our master/slave scenario, the master server also seems to be a bottleneck which breaks down the whole architecture.

Costs

The cost factors for this architecture consists of EC2 instance cost for the Tomcat and MySQL servers, cost for the network traffic between the RBE clients and the SUT and finally the cost generated by the use of EBS volumes. Figure 6.3 shows the total cost per day generated by all load levels in our first experiment with a dedicated MySQL server (see figure 6.1). We observe that the cost for data transfer is disproportionately high and takes a major part of all costs. The cost for the EC2 instances remains constant at $33.60 a day independently of the load and the cost for the storage and I/O requests on EBS are minimal. Compared to other storage services we do not have additional cost for
the computation time consumed. Table 6.4 underlines the generated traffic cost in relation to the number of WIPS and shows the amount of network bandwidth used.

It has to be mentioned that traffic cost is calculated with the highest possible data transfer cost. For the first 10 TB of traffic each month $0.17 per GB are charged. After having reached that limit the cost per GB is stepwise reduced from $0.17 to $0.10.

The total cost generated by the architecture using three MySQL servers is very similar. As the load increase remains the same and the performance of the database is similar, the amount of network traffic generated is approximately the same as in the single server scenario. One exception is the daily instance cost. Due to the two additional MySQL servers it now amounts to $43.20.

### 6.2.3 Amazon SimpleDB

SimpleDB is a storage service accessible through a REST- or SOAP-Interface. This service is provided by Amazon as a replacement for relational databases. SimpleDB reduces the maintenance effort because we no longer need to setup and monitor EC2 instances with databases. Data storage is handled transparently and we do not know on which servers the data is stored and how many times it is replicated.
6.2 Results and Analysis

Benchmark Setting

The SimpleDB storage service is accessed by six Tomcat webservers running on EC2 instances. For ease of development we use the Java Library for Amazon SimpleDB in the benchmark. Because we access SimpleDB from inside the Amazon infrastructure no additional traffic cost is charged. Only the network bandwidth used to deliver the web pages from the webservers to the emulated browsers is charged as usual.

Scalability

Although Amazon underlines the scalability and the speed of SimpleDB our scalability test shows that the service rapidly reaches its performance limit. We set the peak performance of SimpleDB at 128 WIPS. The peak is reached at around 1000 EBs (figure 6.4). Although the number of valid WIPS still increases afterwards, the buy confirm interaction of the web application becomes unavailable from this moment on. In our opinion, the performance increase becomes irrelevant and should not be counted if a key part of the web application does not work anymore.

In the range between 1000 and 2000 EBs about 10% of the issued WIPS are out of response time. This is due to the failure of the previously mentioned buy confirm interaction. We have discovered that SimpleDB refuses to execute updates on the item domain and therefore makes it impossible to update the stock value for each bought item. The item domain is accessed by more than 87% of all interactions. This high read access rate seems to prevent updates from being executed. SimpleDB always returns a Service Unavailable exception. In an attempt to reduce the effects of the problem we tried to recursively repeat the update operation after an interval of 200 ms. Our hope was that the storage
service would automatically adapt to the load increase after a short period of time and execute the update. Unfortunately this did not happen and the operation continued to fail.

Further increase of the load extends the update problem to other domains and causes more and more interactions to fail. Figure 6.4 impressively shows that with 4000 EBs and higher SimpleDB is no longer able to maintain the performance level and the number of delivered error pages increases linearly.

According to Amazon partitioning the domains horizontally increases the throughput and might as well increase the overall performance [31]. Nonetheless, this is a workaround and does not solve the problem. After increasing the load the maximum throughput for the new partitions will be rapidly reached again requiring another repartitioning. Considering that the maximum number of domains per SimpleDB account is set to 100, we must assert that SimpleDB can not scale infinitely.

Looking for additional reasons to explain the poor performance we stumble upon the query mechanism. The simple query language provided by SimpleDB does not support joins across domains and has several additional limits. For example the maximum number of unique attributes in a select query is set to 20 reducing the power of the IN operator. As a direct consequence much more requests have to be send to the storage service in order to display the same result. Logically the load on SimpleDB as well as the response time for each interaction increases.

Costs

The pricing model of SimpleDB is based on machine hours. Each request is charged based on the amount of machine capacity used to complete that particular request. A machine hour costs $0.14 and is according to Amazon calculated using a 1.7 GHz Intel Xeon processor as reference. Based on our scalability experiment, we have calculated the total cost per day for each load (figure 6.5). We can see
that the compute hours rapidly sum up and become a major part of the total cost. The other cost components, the traffic and the EC2 instance cost for the tomcat webservers do not have so much weight.

As mentioned in the previous section, parts of the web application do no longer work after the load exceeds about 1000 EBs. Considering the ongoing increase of cost after this limit we risk wasting money without getting the desired service. Only above 4000 EBs the cost is starting to decrease.

6.2.4 Google App Engine

Google App Engine provides a platform to host web applications in the Google infrastructure. We therefore no longer need EC2 instances to run application servers or databases. The entire application stack is deployed in the Google infrastructure and is then made available at a project specific URL. Google pretends to automatically take care of availability, replication and load balancing.

Benchmark Setting

We do not have access to the Google infrastructure and can therefore not modify the benchmark setting. The tested web application is publicly available at http://ethzbenchmark.appspot.com. To prevent all kind of external access during the benchmark runs, the URL was kept confidential until the publication of this thesis.

Scalability

Figure 6.6 presents the performance of the Google App Engine scalability test. The App Engine reaches a performance peak of around 40 WIPS. After this peak
at around 500 EBs, the WIPS in response time quickly decrease and remain at a very low level. Taking into account that our benchmark is designed to go up to 9000 EBs, the App Engine gives up at a early moment.

![Graph showing Google App Engine Scalability](image)

Figure 6.6: Google App Engine Scalability

It has to be mentioned that the App Engine Java environment used for this benchmark is currently released as a preview version and is in ongoing development. This could explain the bad performance results. It is certainly interesting to compare the results with a benchmark implementation running on the Python environment. The Python environment was the first to be made available and can be considered more mature than the Java one.

Another explanation for the disappointing performance are the current weaknesses in the data query interface. Especially the JPA/JDO API to access the Datastore still has a lot of features not supported. For example the OR, IN or COUNT operators can not be used in queries yet. This handicaps the development process and inevitably leads to a higher number of requests send to the storage service. For example, table 6.5 compares the different architectures by the number of storage accesses necessary to display fifty random items on the bestseller interaction. To correctly display the page we need to retrieve fifty items from the item domain and the fifty corresponding author names from the author domain. MySQL requires one query which executes a join on both tables to get the data. SimpleDB needs 6 requests to deliver the data. In each request 20 id comparisons are allowed by SimpleDB. And finally, because of the missing IN operator and the lack of joins, the App Engine can only perform one predicate comparison per query and has therefore to execute a total of 100 queries. Fifty for the items and authors respectively.

A solution to overcome the problem would be a local data cache which keeps a copy of unmodified data. Once the data is loaded into cache it can be directly accessed without requiring a request to the storage service. This could largely
6.2 Results and Analysis

<table>
<thead>
<tr>
<th>Queries on item domain</th>
<th>MySQL</th>
<th>SimpleDB</th>
<th>App Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Queries on author domain</td>
<td>-</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1</strong></td>
<td><strong>6</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

Table 6.5: Data Access Operations for Bestseller Interaction

reduce the load on Datastore. As part of the App Engine service stack, Google provides a service called Memcache for exactly this purpose. Memcache is a distributed in-memory cache for key/value pairs. Implementing the Memcache feature for our benchmark is part of future work.

**Costs**

The cost factors for this architecture consist of CPU time consumed, network traffic generated and storage space occupied. As the scalability in the application is limited, cost does not reach high amounts (table 6.6). The cost for the consumed CPU time is dominating the cost calculation. CPU time is composed of two elements. First, the processing of incoming requests and second, the retrieval and modification of data on Datastore. As Google provides the entire application infrastructure this seems a reasonable way to charge the operating expenses. According to Google CPU time is calculated using a 1.2 Ghz Intel processor as reference.

<table>
<thead>
<tr>
<th>Number of EBs</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time costs</td>
<td>30.96</td>
<td>73.44</td>
<td>126.24</td>
<td>130.32</td>
<td>132.16</td>
<td>117.36</td>
</tr>
<tr>
<td>Traffic costs</td>
<td>4.07</td>
<td>9.56</td>
<td>17.42</td>
<td>17.82</td>
<td>27.07</td>
<td>22.92</td>
</tr>
<tr>
<td>Storage costs</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>35.05</strong></td>
<td><strong>83.02</strong></td>
<td><strong>143.68</strong></td>
<td><strong>148.16</strong></td>
<td><strong>159.25</strong></td>
<td><strong>140.30</strong></td>
</tr>
</tbody>
</table>

Table 6.6: Google App Engine Total Cost per Day (in $)

The cost for data transfer is load specific and remains at a normal level. Google takes advantage of lower data transfer cost than Amazon. An outgoing GB of data costs $0.12. This is approximately 30% less than its business rival.

6.2.5 Architecture Comparison

Comparing the performance of all benchmarked architectures (figure 6.7), we can state that relational databases provide a far better performance than current cloud storage services. Relational databases have been developed for 30 years and are strongly optimized for query processing. But, as seen in the MySQL master/slave scenario, scalability is limited. On the other side, cloud technologies are still in their infancy and have lot of potential for performance increase. Even if the performance of cloud storage services is not convincing. They have a right to exist. Indeed, as figure 6.8 underlines, for web applications with low load, 100 EB or lower, SimpleDB and the App Engine have a lower cost per WIPS than MySQL. A relational database in the cloud suffers from constant instance cost
which increases the cost per WIPS the lower the load is. But on the contrary, the more the load increases the cheaper the WIPS in the MySQL architecture become.

The relatively constant cost per WIPS for SimpleDB and the App Engine before having reached their respective peak performance underlines the successful implementation of the pay-per-use cost model. Indeed, in an optimal situation, a linear performance increase should result in a linear cost increase and thus the cost per WIPS remain constant.

Having a look at the cost per WIPS evolution for higher load, figure 6.9 reveals that additionally to the performance crown, MySQL is in the overall the cheapest architecture. The cost generated by consumed computation time makes SimpleDB and the App Engine more expensive. The App Engine even outdistances SimpleDB because of the bad performance at a load superior to 1000 EBs.

To summarize table 6.7 presents the costs for a valid WIPS at the peak performance of each architecture. This confirms our statement of MySQL being the less expensive architecture.

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>SimpleDB</th>
<th>App Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak Performance</td>
<td>477</td>
<td>128</td>
<td>39</td>
</tr>
<tr>
<td>Costs per valid WIPS</td>
<td>6.3345</td>
<td>40.171</td>
<td>42.238</td>
</tr>
</tbody>
</table>

Table 6.7: Costs per WIPS at Peak Performance

To take advantage of cloud systems and cloud storage services, we need new approaches in the development of web applications. First, cloud applications have new requirements in terms of data modelling. Second, the limits of query languages have to be taken into account. For example, the data model of each
6.2 Results and Analysis

![Graph showing cost per valid WIPS for different applications.](image)

Figure 6.8: Comparison of Cost per valid WIPS

application should be optimized to reduce the number of necessary joins. Third, data caching such be used as much as possible. All static and session data can be cached in memory. Once the cache is loaded, the storage service is only accessed to update data and refresh the cache. This could dramatically reduce the number of accesses and would probability increase the overall application performance. The higher development effort for implementing the caching mechanisms should be quickly compensated by the reduced cost for less computation time consumed.

6.2.6 EC2 I/O and CPU performance

During our benchmark runs involving EC2 instances we have noticed varying system performance. In order to find the reason we further analyzed the I/O throughput of EBS and the CPU performance of EC2 instances.

EC2 I/O performance

Persistent storage on EC2 instances can be easily achieved with EBS. Because EBS is an external storage system, it is the first component to be suspected when dealing with performance fluctuations. To check the performance of EBS volumes we repeated write and read files with the size of 1 MB and 1 GB and compare the throughput with the local non persistent filesystem.

Technically, the performance test is executed by a shell script. The script contains an infinite loop inside which the read and write operations for both file sizes on both storage systems are executed in sequence. After a 9 minute break, the sequence is repeated. This script runs until it is manually interrupted. For all operations we measure the time to transfer 1 GB of data and thereafter calculate the throughput in MB/s. Therefore each operation with the 1 GB file size only writes or reads 1 file, whereas the 1 MB operations are repeated a 1000 times to
reach the 1 GB data transfer amount.
The script was started on an EC2 instance with a mounted 10 GB EBS volume. In the following 4 days, from Sunday to Wednesday, we collected more than 560 measurements.

All results are displayed as boxplots to see the performance spread at a glance. As an example, figure 6.10 shows the read performance for 1 MB files. In a boxplot, the box contains the middle 50% of the data. The upper edge of the box indicates the 75th percentile of the data set, and the lower one indicates the 25th percentile. The range of the middle two quartiles is known as the inter-quartile range (IQR). The line in the box indicates the median value of the data. The rhomb indicates the average value. The ends of the vertical lines or whiskers indicate the minimum and maximum data values, unless outliers are present in which case the whiskers extend to a maximum of 1.5 times the IQR. The points outside the ends of the whiskers are outliers.

Looking at the read performance, we have found that for reading small files EBS is faster than the local filesystem (figure 6.10). EBS reaches a median throughput of 170.94 MB/s whereas the local filesystem only reaches 117.92 MB/s. Additionally the local read has a larger variation compared to EBS which might result from concurrent accesses to the local hard drives by several instances simultaneously. The read of larger files shows another result. In figure 6.11 we notice that both EBS and the local filesystem reach a similar performance in reading 1 GB files. Figure 6.12 shows the write performance for 1 MB files. Here again EBS is faster. The median throughput is 89.77 MB/s compared to 66.69 MB/s for the local filesystem. For the write performance with large files, we observe an opposite result (figure 6.13). The local filesystem is faster in writing large files than EBS. Although the median write throughput for EBS is at 68.10 MB/s we have several outliers which point out that occasionally the EBS throughput for large files can
6.2 Results and Analysis

strongly drop. The median write performance of the local filesystem is at 84.30 MB/s and does never drop under 60 MB/s.
As a conclusion, EBS seems to be optimized for reading and writing smaller files.
In case of a relational database, which usually flushed data pages with the size of several dozen KBs to disc EBS is certainly the better choice compared to the local filesystem.

**EC2 CPU performance**

To verify variations in CPU performance of EC2 instances we use BYTE Magazine’s BYTEmark benchmark program. This CPU benchmark dates back to 1997 and executes a serial of algorithms to measure the performance of a systems CPU. More detailed information can be found at [32]. The benchmark is single-threaded and not designed to measure the performance gain on multi-processor or multi-core machines. However, because performance variations have an impact on every core of a machine it should be enough for our purpose. As result of a run, the benchmark outputs a value for the integer and the floating-point computation speed. Both values are a multiplier indicating how much faster the CPU is compared to a AMD K6/233 CPU.
We executed the benchmark over 850 times on 10 different EC2 instances in all 4 data centers of the US-East availability zone. The benchmarks were timed to cover almost every hour of the day and every day of the week. Considering all benchmark runs, we have obtained a standard deviation of 2.26% in the integer CPU performance and a standard deviation of 4.73% in the floating-point CPU performance. As such small variations are usual on every computer we can conclude that the CPU speed of EC2 instances is stable and that a similar performance can be expected 24 hours 7 days a week.
Concluding, the CPU and I/O tests did not provide an explanation for the performance fluctuations encountered during the benchmark runs. As part of future work, we will perform much longer measurements and extend the tests to more
Figure 6.11: Read Performance of 1 GB Files

EC2 instances and/or EBS volumes.
6.2 Results and Analysis

Figure 6.12: Write Performance of 1 MB Files

Figure 6.13: Write Performance of 1 GB Files
Cloud computing is an active and evolving market. In this context we have evaluated and benchmarked several architectures using services of different cloud providers.

We have analyzed how applications can be developed for the cloud and defined a reference architecture. Because cloud services mostly have a specific purpose and can be accessed through a public interface, not all components of an architecture must run in the cloud. Cloud services provide flexibility. They can be used to operate single application components or host the entire application stack.

We have highlighted the fact, that although there are proprietary and restricted solutions, most cloud services can be combined in multiple ways. Common to all services is their ability to scale. Either automatically or manually, scaling further increases the number of architecture variants.

In order to compare and evaluate cloud systems we have gathered requirements for a cloud benchmark and developed new metrics to measure the cloud specific properties of scalability, pay-as-you-go, fault-tolerance and peak behavior. We used the widely accepted TPC-W benchmark, which simulates an e-commerce application as basis and then adapted it to fulfill our requirements and measure our metrics. This benchmark was implemented on five architectures for the cloud, each of them characterized by a specific setup.

The experiments were conducted with a focus on the big players of the cloud market. We found out that cloud storage services like Amazon SimpleDB or Google App Engine Datastore can not reach the performance of relational databases. Traditional architectures with an application server and a database running on virtual machines currently are the most efficient way to operate applications in the cloud. However, for low load applications PaaS platforms or cloud storage services can be more cost-efficient. Especially when free quotas are provided (e.g., Google App Engine) an application can be made available for free or at low cost.

When moving to the cloud, the choice of the right cloud services has to be seriously evaluated. Even if cloud providers have similar pricing models, the cost for hosting similar applications can differ. Different standards in measuring the CPU time or slight differences in the price for data transfer can lead to large gaps in the cost.

To conclude, we observe that cloud systems are more and more used in business. They provide on-demand computing resources coupled with an innovative pricing system and simplify the developers everyday life. Although a lot of challenges
have still to be solved, it is probable that the cloud will continue its grow in near future. As a young technology cloud computing also offers a lot of research opportunities. This is reason enough to follow this interesting topic with high practical relevance.

7.1 Future Work

We identified several areas in which the work of this thesis can be continued. In our experiments we have decided to forego two architectures because of time constraints. The most obvious future work would be to benchmark these architectures as well. It is interesting to analyze how specific properties (e.g., local caching of data) of these architectures influence the system’s performance. Another area for research would be if small cloud providers like 28msec, the developer of Sausalito [1], can keep up with the big players in terms of cost and performance. Considering the large number of cloud services and possible service combinations, benchmarking further architectures would be a logical next step. A first candidate architecture to be implemented is Microsoft Azure. Azure is a PaaS service based on the .NET platform and requires a complete reimplementation of the benchmark in one of the .NET programming languages. A second candidate is the extension of the Google App Engine implementation with a local cache. Google provides a Memcache service to cache data in-memory. This implementation would help to identify the overhead generated by calling a storage service compared to retrieving data from memory. In order to get significant results, the benchmark application has to be implemented according to modern web technologies and design patterns. Considering the fast evolution of internet technologies in the last years, the TPC-W bookstore, which dates back to 2002 is outdated. To modernize the benchmark our idea is to define new interactions that resemble the access patterns of Web2.0 applications. One example would be to add a web interaction that allows users to write and read reviews of individual products. Moreover, multi-media content (e.g., flash videos) is often part of Web2.0 applications and should also be added.
Bibliography


