FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Non-functional Testing in Cloud Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Naderer-Puiu, BSc
Matrikelnummer 0625238

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Univ.Ass. Priv.Doz. Mag. Dr. Manuel Wimmer

Wien, 22.03.2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Non-functional Testing in Cloud Environments

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Naderer-Puiu, BSc
Registration Number 0625238

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Univ.Ass. Priv.Doz. Mag. Dr. Manuel Wimmer

Vienna, 22.03.2016

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Naderer-Puiu, BSc
Gisela-Legath-Gasse 5/27, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

Cloud computing affects large parts of the IT industry today. Acquiring computing resources and services on-demand not only changed how IT is structured, also new frameworks and technologies emerged in the recent years. Users of cloud services expect better scalability and application elasticity from migrating into the cloud. Instead of provisioning own hardware to applications they rely on the provider's elastic provisioning system and allocate resources if needed.

The area of cloud computing is divided into three essential service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The latter is the broadest of all models and basically includes software or web applications provided over the Internet to customers for an usage-only billing. Between IaaS and PaaS the distinction became harder with recent developments in the field. IaaS providers include more services and are no longer only virtual infrastructure providers. They enrich IaaS solutions with dynamic service components and managed software systems. On the other side PaaS solutions developed from an application-centric model to a service with advanced control over the underlying virtualized hardware components. This thesis starts with a look on preliminary progress in PaaS, shows the difficulties of migrating an existing traditional application into a popular PaaS platform, and evaluates how a PaaS-based application can be tested for the non-functional requirements scalability and application elasticity.

Google App Engine has been chosen as a representative and popular implementation of the PaaS service model. It is available since 2008 and is one of the most mature commercial platforms. With its Java sandboxed runtime it is an ideal target for JVM-based applications and therefore is open for a wide range of technologies and existing Java applications. The NoSQL Datastore will be comprehensively investigated and examined as a target data storage backend for existing relational data models taken out of SQL databases.

In a case study the scalability and elasticity of applications running on top of Google App Engine will be evaluated and assessed. A load test will be executed in a distributed setup based on immutable infrastructure created with Docker containers. The goal is to answer three distinct research questions focused on JMeter as a load test tool, App Engine's instance provisioning and load balancing, and on the 1 write per second per entity group limit exposed by App Engine's Datastore. This all leads to the overall goal of this thesis: Evaluating if App Engine is a good target for migrations from existing applications into the cloud. The work will not only cover functional requirements, but rather will focus especially on non-functional requirements. PaaS platforms require more adaptions and a deeper integration of third party services into the own applications, so risks have to be clear before a migration starts and re-evaluated after a functionally successful migration. Hence, the delivery of what a cloud provider promises has to be proven.

Kurzfassung

Cloud Computing übt heute umfassenden Einfluss auf die IT-Industrie aus. Die dynamische Bereitstellung von IT-Ressourcen veränderte dabei nicht nur das grundlegende Verständnis von IT-Infrastruktur, sondern zeigte auch Auswirkungen auf IT-Prozesse, neue Software-Frameworks und im Bereich von Datenbanken. Anstatt dem Bereitstellen von angekaufter Hardware in gemieteten oder In-House-Rechenzentren, wird zunehmend auf virtualisierte Hardware gesetzt. Diese wird dynamisch bei einem Cloud-Provider angefordert und dieser stellt nur die tatsächliche verbrauchten Einheiten in Rechnung.

Angebote im Bereich Cloud Computing lassen sich in drei Servicemodelle unterteilen: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) und Software as a Service (SaaS). Aktuell verschwimmen die Grenzen zwischen IaaS- und PaaS-Angeboten zunehmend. IaaS-Anbieter erweitern ihr Portfolio um Softwareservices, deren Wartung und laufende Betreuung ebenfalls vom Anbieter übernommen wird. PaaS-Angebote erhalten hingegen immer mehr Kontrollmöglichkeiten über die darunter liegenden Softwarekomponenten und die virtuelle Hardware. Diese Diplomarbeit beschäftigt sich mit den bisherigen Entwicklungen im Feld von PaaS, zeigt die Schwierigkeiten bei der Migration von existierenden Anwendungen in eine PaaS-Cloud auf und evaluiert wie PaaS-basierte Anwendungen hinsichtlich nichtfunktionaler Anforderungen getestet werden können.

Google App Engine wurde zur konkreten Evaluierung für diese Arbeit herangezogen, da es sich um eine ausgereifte und bereits länger am Markt verfügbare Plattform handelt. Durch die Unterstützung von Java eignet sich App Engine als Ziel für Java-Applikationen. Dies ist besonders für den Enterprise-Bereich eine wichtige Eigenschaft und lässt App Engine eine besondere Bedeutung zukommen. Weiters wird der NoSQL-basierte Google Cloud Datastore umfassend untersucht. Besonderer eingegangen wird auf die Migration von existierenden SQL-basierten Datenmodellen.

Diese Arbeit umfasst auch eine Fallstudie zur Skalierbarkeit und Elastizität von Anwendungen. Beide Eigenschaften werden durch verteilte Lasttests in mehreren Durchläufen getestet. Dabei sind drei wesentliche Forschungsfragen zu beantworten: Eignet sich JMeter als Lasttestgenerator für Cloud-Anwendungen? Wie reagiert App Engine auf eine hohe Anzahl an Zugriffen? Wie wirkt sich das Entity Group Write Limit in der Praxis aus? Die Beantwortung dieser Fragen führt zur übergeordneten Fragestellung: Eignet sich App Engine als Zielplattform für Migrationen von bestehenden Java-Applikationen in die Cloud? Dabei fokussiert sich diese Arbeit nicht nur auf die Überprüfung von funktionalen Anforderungen, sondern legt besonderen Wert auf nichtfunktionale Anforderungen und die daraus entstehenden Konsequenzen für Migrationsprozesse.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Making computing resources available via web services became one of the most influential shifts in the IT industry. Under the term *Cloud Computing* a vital and emerging business sector developed in the recent years. Instead of housing server systems or using hosting providers, customers can rent computing resources on demand with immediate provisioning. Cloud computing promises a significant reduction in IT effort, a more detailed control over IT costs, and a better concentration on a company's core business model. Services focused on the infrastructure layer are summarized under the term Infrastructure as a Service (IaaS), more integrated software solutions evolved in Platform as a Service (PaaS) offerings. Here not only the underlying hardware is abstracted for the customer, also a variety of software services and management tools are offered by the provider.

Cloud computing became one of the most important fields in commercial IT services. [1] summarizes recent research of advisory consultancies on the cloud computing market. They see a 32.8% growth from 2014 to 2015 in the global spending on IaaS services. In the IaaS economy Amazon Web Services (AWS) are the dominant service with around half of the market share. PaaS services are still seen as a growing market, but will keep significantly smaller than the IaaS market. [2] sees benefits for small and medium enterprises (SME) in building cost-effective services, improved flexibility, and lower requirements compared to self-owned IT. Expectations especially in this economic sector are high and migrations to the cloud are easier to start in the more agile SME compared to larger enterprises. Their often more innovative and disruptive business models can benefit the most from cloud computing. Moreover, adopting cloud computing enables SME to build scalable products exceeding their current business needs.

Using IaaS services also requires a certain amount of resource and configuration management. This gives more control over the software stack, but reduces the outsourcing level of sophisticated tasks. At this point PaaS offerings go a step further. The range goes from advanced operating system services to fully abstracted runtime environments. The customer typically provides only the application and the provider is responsible for the execution lifecycle and load handling.

All of these services are not standardized and vary from provider to provider. The access to detailed runtime information and middleware components can be restricted in various ways. Therefore, various concerns emerge, e.g. the outsourcing of critical parts of the software stack, compliance issues regarding data privacy and security, loss of internal know-how, vendor lock-in, and application scalability. These concerns mostly relate to non-functional requirements, since roots lie in the general quality attributes of an application.

Companies with a skilled IT staff typically have enough internal resources to run applications on top of an IaaS service. This might not be the case for SME and individuals. For them PaaS is an interesting option removing the burden of maintaining full application stacks. The basic building blocks of an application are provided by the PaaS service itself. This can include data storage, load balancing, messaging services, and support for multiple programing languages and frameworks. Only the application itself has to be deployed into the provider's cloud. The overall result should be an application that is scalable and resource efficient, but only requiring a small amount of manual intervention.

1.2 Problem Statement

In the face of the described concerns and opportunities a potential migration from an existing software stack into a PaaS cloud service has to be prepared carefully. There exists no standardized way to characterize quantitative and qualitative requirements of web applications. Popular unquantified metrics are response time, application elasticity and overall scalability.

The Java platform is one of the major platforms for enterprise software and has a sophisticated support for web applications. If narrowed to it, the migration of JEE[1]-based applications into cloud services is a current issue for many organizations. One possible target platform is Google's App Engine with its Java runtime. In the PaaS market App Engine is a major competitor according to [1]. Google opened it's own infrastructure for external customers and provides a PaaS platform (App Engine[2]) and an IaaS service (Compute Engine[3]) to the public.

App Engine is focused on running web applications and provides a rich set of services. These services can be consumed over a platform-specific SDK. App Engine promises that "applications written for App Engine scale automatically[3]". However, it does not offer a set of tools to evaluate non-functional requirements and has just rudimentary testing tools available yet. Since the initial release in 2008 the service has been through multiple stages of development and is part of Google's Cloud Platform now. App Engine became one of the leading PaaS offerings[4] and is focused on running web applications with a NoSQL-based data backend. It's Java runtime environment provides a sandboxed application execution context and is based on the Java Servlet interface.

This sandbox restriction and the general non-transparency require to measure scalability and evaluate application elasticity with an external tool. Furthermore, smooth migrations into the cloud require the identification of sources for errors and platform-specific impediments. In

[1] Java Platform, Enterprise Edition

[2] `https://cloud.google.com/products/app-engine/`

[3] `https://cloud.google.com/products/compute-engine/`

[4] `http://cloudacademy.com/blog/paas-top-players/`

proprietary systems like App Engine this demands a deep understanding of the platform and revealing the consequences of various design decisions.

1.3 Aim of the Work

Especially for software modernization projects it is important to know about the strengths and weaknesses of the potentially targeted cloud service. Profound guidance helps to simplify the migration path and to avoid obstacles during the software conversion. The main aim of this work is to provide such assistance. In an experiment the feasibility for a direct conversion of JPA-based[5] applications to App Engine's NoSQL-based Datastore is tested. This could further improve model-driven engineering techniques which transform existing relational data models into cloud-platform specific models. Additional information about the principal design of the Datastore should help to understand the specifics and implementation details. This involves a detailed look at the behavior of transactions and the resulting consistency constraints. A comparison of the JPA-based approach and Objectify, an open-source Google Datastore-only persistence API for Java, should help to choose the right persistence layer for a specific migration process.

Following this first investigation a subsequent case study will take a closer look on non-functional requirement testing. It observes the real world behavior of a sample application and executes load tests with JMeter, a server performance testing tool. Test server instances are distributed over multiple datacenter locations to get more solid results. This case study should increase the general knowledge about the scalability ability and elasticity of applications running on top of App Engine. It is designed to answer three separate questions:

1. RQ1: Is JMeter a valid testing tool for cloud-based applications?
2. RQ2: How does App Engine react on incoming traffic and what patterns can be detected during a load test?
3. RQ3: What are the effects of the throughput limit of 1 write per second per entity group for the Datastore?

The case study is influenced by the results of the persistence layer experiment: Objectify has been chosen to get a realistic App Engine setting. It aims to give suggestions for application design and data modeling, and to reveal potential general cost drivers for the PaaS model. A special focus is on the right choice for entity group boundaries, which are a very unique property of the Datastore.

[5]The Java Persistence API provides classes, methods, and tools for the management of relational data.

1.4 Methodological Approach

1. **Literature Review**

2. **Definition and Specification of Non-Functional Requirements**
   Before a system can be tested it is important to understand which non-functional requirements are influential in real world environments and how they can be measured in a quantitative way. Tests should be repeatable and return consistent results for equivalent test conditions. On cloud platforms the evolution of an application depending on the workload is a mayor decision criterion to judge if the move to the cloud is rewarding or not. It has to be considered what evolutionary qualities have notable influence on cloud-based applications.

3. **Development of Application Prototypes**
   For the persistence API experiment an application will be implemented using the JPA. This application has a simple data model based on JPA annotations. Tests accessible via an URL will trigger certain Datastore functions and try to save, read and modify entities. These tests will be executed inside the local development server provided by the SDK and in the production environment. The latter requires a full deployment to the App Engine cloud. Following this first prototype a second prototype is developed which acts as system under load for the load test. It contains multiple test cases in one single web application and will be deployed as a normal App Engine application.

4. **Testing the Prototypes on App Engine**
   In this stage the prototypes are deployed and tested on App Engine using various tools and App Engine-specific debugging services to determine the performance of web applications under real runtime environment conditions. The runtime behavior is monitored and later analyzed for bottlenecks and suboptimal coding patterns.

5. **Evaluation**
   The data gathered in the testing stage from the prototypes will now be analyzed and evaluated. The JPA tests are examined for functional equivalence between the SQL-based and Datastore implementation. The benchmark application is tested to find for weak spots in application performance and potential risks for an application's scalability. Hence, complexities of non-functional requirement testing for cloud platforms should be described and uncovered. At the end the reader should be able to see the potential advantages and disadvantages of App Engine as a PaaS services, but also where the software design needs special adaptions to run in such an restricted environment.

1.5 Structure of the Work

The subsequent chapters of this thesis are structured as follows: Chapter 2 gives a description what functional and non-functional requirements are. It provides a definition for the non-functional requirements scalability, elasticity, responsiveness, and costs with a focus on cloud computing. A brief introduction into cloud computing is given and the different characteristics of them will be discussed. Simulation-based approaches and their strengths and weaknesses are presented.

In chapter 3 related work to this thesis' topics are presented. An overview over current non-functional testing in cloud environments is provided.

Chapter 4 starts with a detailed description of App Engine and its service components. The application lifecycle is presented and the Java runtime is analyzed in more detail. Datastore entities and their characteristics will be presented in a comprehensive investigation. On this basis, the JPA implementation for App Engine is destructed in an extended experiment. It will be shown which functionally equivalent features of JPA are available on App Engine and which parts of JPA are not supported.

A case study analyzing a number of performance-related issues is the purpose of chapter 5. The problem of context-depended practical knowledge will be described. Since App Engine is a proprietary technology, the case study looks as a promising methodology to uncover new knowledge about it. The performance test setup will be unfolded in detail and the measuring arrangement with JMeter is described. At the end of the chapter the results will be discussed.

The final conclusion follows in chapter 6. Hence, potential future work building up on the prior experiment and case study is discussed.

CHAPTER 2

State of the Art

Cloud computing became a dominant computing model and changed the IT landscape in recent years. Instead of building up own datacenters, organizations can rent computational resources on-demand without long upfront times. In the scientific community this paradigm shift promoted a broad research area, starting at the different cloud service models (IaaS, PaaS, and SaaS), and covering also ethical consequences of the centralization of IT services. The following sections provide an overview over current research, even if it is only possible to present a small sample of scientific effort.

2.1 Non-functional Requirements

To describe non-functional requirements, a definition of functional requirements should be given beforehand. Functional requirements are often characterized as hard facts and properties of a software system. They describe different measurable goals and objectives and involve a defined function of the discussed system. [4] couples functional requirements with the stakeholder of a system:

> "The functional requirements are a direct extension of the stakeholder's purpose for the systems and the goals and objectives that satisfy them."

Features of a system can be formally described by languages like the Unified Modeling Language (UML), e.g. are UML use case diagrams used to describe functionalities of a system and how different actors are involved. Moreover, UML profiles to model non-functional requirements have been developed. [5] discusses challenges of using UML for modeling non-functional requirements. Examples of non-functional properties are: performance, reliability, availability, fault-tolerance, scalability, security, maintainability, costs, accessibility, legal and licensing compatibility, source code availability, testability, and many others. [6] describes the current state of the art for models used to perform non-functional requirement verification:

> "Such models are derived from software models (or selected views thereof) annotated with information specific to the property to verified. How to do such annotations was a question already addressed by researchers and practitioners. In the UML world, such annotations are done via UML profiles, which are a standard extension mechanism supported by UML editors."

One example for such an UML profile is MARTE (Modeling And Analysis Of Real-Time Embedded Systems), which is used with UML2. Another one is UMLsec for security analysis of a system, which can be used to verify security-related critical requirements and to provide security by design. But often in practice non-functional requirements are described only verbally. This makes it hard to evaluate if a system passes a requirement and if the overall system design satisfies the stakeholder needs.

Choosing the best matching cloud infrastructure often involves a high number of non-functional requirements to be fulfilled. [7] describes this as a multi-criteria selection problem and performs an experimental deployment of WordPress, which is a popular CMS and weblog publishing tool written in PHP. Potential candidate criteria for selection a cloud infrastructure a) efficiency, which consists of the sub-properties response time, CPU utilization and memory utilization; b) cost; and c) scalability. Summarizing, [7] sees non-functional requirements as a key driver for the selection process.

Scalability and Elasticity

Scalability is an important factor for the overall quality of service (QoS) of a cloud system. [8] describes a framework for scalability and performance testing, which runs in private and in public clouds and which reduces the amount of work required to test performance in the cloud. They evaluate scalability by running a load test and monitoring the instances CPU utilization, memory consumption and I/O performance. To evaluate the scalability of a cloud infrastructure, [8] used a MediaWiki installation, whereas [7] used WordPress. Another approach is to write specific benchmark tests for a given infrastructure. This might be enforced by platform restrictions, or the lack of representative open-source software to test. For example, PaaS solutions like App Engine require a lot of custom adaptions. It has proprietary APIs, so most software running on it requires customization and this makes a comparison with other cloud platforms harder.

The scalability metrics may be related to response time metrics[9]. A scalable system might not be per definition fast and show low response times, but in general it's a good indicator to have low response times. Also related to a scalability metric is the error rate of a system. Scaling up requires also keeping the total error rate low, otherwise a system is not reliable.

Elasticity is an important property of every auto-scaling cloud system. If a software cannot react to newly provided resources, it's not elastic enough to scale. Provisioned capacities have to be used by the components running on top of them, otherwise resources are wasted and lead to an inefficient use. Elasticity also has a time-based component. It's not enough to define this term only on resource utilization, since the time it takes to utilize new resources is also a crucial component in the overall system. Cloud software which is not fast enough to handle load peaks will not be able to scale and therefore is not elastic[9]. Though, not only scaling up is a requirement for an elastic system, also reducing the acquired resources if low load is detected

is an important property. Otherwise resources are wasted and a major benefit cloud computing promises is not present. Most cloud environments use a pay-per-use model to calculate the costs and typical IaaS providers calculate the usage depending on running virtual machines and not based on actual load. [10] introduces the term *agility* to describe the ability to integrate newly provisioned resources into a system. Though, they focus on simulation-based approaches to analyze cloud environments.

In [9] various solutions and system are listed, which provide mechanisms to create an overall elastic system. Current solutions include the App Engine load balancer, Elastic Load Balancing in Amazon's cloud service AWS, Rightscale and Microsoft's Azure platform. Evaluating the elasticity is often closely related to testing the actual cloud system and the general results are derived from experiments and specific benchmarks.

[11] criticize the lack of a common definition of the terms scalability, elasticity and efficiency. In their systematic literature review they try to narrow the definitions of these terms and to provide a better guidance for the usage of the terms. They see a common definition as an important factor to negotiate service level agreements and to keep a clear language between different stakeholders. Some previous approaches focused on defining metrics for benchmarking and load testing, whereas this prevents a broader acceptance of these metrics. Scalability itself is often limited to the allocation of new or fresh resources to handle an increasing workload in the cloud system. A system with an increasing workload should still be able to fulfill all SLA or QoS agreements and the underlying low-level resources should be acquired in an automatic way without any manual intervention. There is no direct link between scalability and time. The scalability only describes a steady state, not a temporal property of the system. Definitions of elasticity differ from scalability in some points, but there are also overlapping areas. Elasticity has a strong dependency to the time it takes to use the newly acquired or freed resources. It's connected with the terms over- and underprovisioning of a system, so that the resources are used in an optimized way. There exist approaches to provide temporal logic languages to formally describe elasticity of a system. A better elasticity also improves the overall efficiency of a system, whereas scalability is not directly linked to the efficiency of a system. A high scalability does not imply a better efficiency of a system, but a better elasticity always improves the efficiency. This also means that scalability and efficiency are linked together over the resource usage of the system. [11] therefore suggests the following definitions:

- *Scalability is the ability of a cloud layer to increase its capacity by expanding its quantity of consumed lower-layer services.*
- *Elasticity is the degree a cloud layer autonomously adapts capacity to workload over time.*
- *Efficiency is a measure relating demanded capacity to consumed services over time.*

Responsiveness

Most web sites and web services have a direct connection between the responsiveness of the backend and the user-facing interface. A performant backend generates HTML documents faster

or delivers API responses in a shorter time. Responsiveness of an interface is one important principle of user interaction design (UX). Users don't want to be distracted from their tasks and want to feel that they are in control of the interface they use. A highly responsive site motivates users to interact more and keeps them focused. Nielsen[12] identifies three major response-time limits:

- 0.1 seconds is the limit for an instantaneous manipulation and is perceived as direct response to an interaction.
- 1 second is the barrier for an seamless and appropriate navigation experience. The user may acknowledge that his actions need computation and further processing.
- 10 seconds is the upper bound to keep users attention. If the response takes longer, users show a high exit rate and are unsatisfied with the service.

A snappy UX helps to generate a high user engagement and increases the use time of a service. In 2009 Forrester Consulting found out in a study for Akamai[13], that potential customers of an online shop became impatient after a load time of 2 seconds. The load time also has influence on customers loyalty and poor performance influences the emotional level during a visit. In a talk in 2006, back then VP of Google, Marissa Mayer emphasized the importance of speed for a good UX. 500 milliseconds delay caused a 20 percent drop at Google[1]. Today the shift into mobile increased the performance challenges. Mobile devices have a limited bandwidth, use smaller client-side caches, offer specialized mobile browsers and often have limited hardware resources. Minimizing the data transfer and optimizing the server-side stack are important steps to provide a fluent and frictionless UX.

A key factor for high responsiveness is a fast execution on the server. If the processing on the server takes too long, other optimizations are useless. Cloud services promise a high performance execution and offer an advanced web application development stack. They integrate caching in various layers and often make caching mechanisms transparent to the developer. Seamless scaling from zero to a hundred thousands of users is a key feature, but adding users should not influence the response time performance of cloud-based applications.

Costs

The cost factor of cloud computing can be considered from two different views: as a customer paying for resource usage and as a provider who charges users of a specific service. This also leads to two contradictory goals: reducing costs for consuming computing resources on the one hand, and maximizing profit for serving a service on the other.

Consumers of cloud services expect a cost saving and better overall efficiency of their computing resources. [14] sees a cost advantage of using commercial cloud offerings for small-scale scientific research in the field of quantum chemistry. This also brings up an important further breakdown of the term cloud computing: commercial and academic clouds. Commercial clouds are normally public cloud offerings, which can be used by a variety of applications. Academic

[1] `http://glinden.blogspot.co.at/2006/11/marissa-mayer-at-web-20.html`

clouds are optimized for scientific use, which involves solving computational complex problems by powerful CPU and GPU machines. Projects like the Nimbus[2] provide IaaS clouds with a focus on scientific research. Specially the costs of current IaaS solutions like Amazon EC2 are a problem for large scale scientific computing[15]. For cloud simulations like [16] describes, modeling of costs and economic aspects is a part of the overall simulation. The case study of [17] focuses on the migration of an existing enterprise-level application into Amazon's AWS cloud based on the EC2 service. They discovered a 37 percent reduction of cost over five years, but also suggest a further look not only covering the raw infrastructure costs and include also migration costs, support staff costs and hard to quantify costs like loss of experience inside the internal IT department. Special cloud brokerage services can reduce the effective cost of cloud computing. Instead of acquiring a pool of resource, the broker service tries to find an ideal distribution of jobs. An example for deadline-constrained batch jobs is discussed in [18].

2.2 Cloud Computing

The term *Cloud Computing* has been used in the ICT and scientific community a very broad interpretation and often mixes the different service models. [19] tries to sharpen the definition of the term cloud computing. Cloud computing is a modern and uprising model to create large distributed systems using dynamically provisioned pools of resources. Cloud providers measure the consumed resources and costs are charged according to their usage only. Users expect better scalability and elasticity from moving into the cloud. Instead of provisioning own hardware to applications they rely on the provider's elastic provisioning system and allocate resources on-demand. High availability and failover management should guarantee short downtimes. In general, the availability of cloud services is hard to reach in traditional IT environments with self-managed hardware.

[19] introduces the three different approaches for provisioning sufficient computing resources to applications in cloud systems. *Provisioning for peak load* is the simplest approach of all three. Instead of having any dynamic provisioning in place, the whole system is capable of the expected traffic peaks at any time. This is also the most inefficient approach regarding the total costs of a system, since all resources have to be available throughout the lifetime of the system. *Underprovisioning* on the other hand leads to a number of users with an unsatisfying experience of the service. Therefore, dynamic changes in the provisioning of resource for the overall system is a key component of cloud services. [19] concludes that scalability is one of the key features of modern applications and that infrastructure and software must be aware of running in virtualized environments instead of real hardware servers. The authors also see a demand for change in the software licensing business, otherwise open-source software with easier licensing conditions have a big advantage over proprietary software.

[20] provides an overview over the current cloud service landscape and what are the main challenges to solve for cloud computing as a model for using shared resources over the Internet. One interesting point is the discussion of *Virtual Private Clouds* from Amazon Web Services to build a sort of private cloud with permanent VPN connections on the basis of the public cloud

[2] `http://www.nimbusproject.org`

infrastructure offered by EC2. It covers the relationships between cloud computing, service-oriented computing (SOC) and grid computing. SOC brings technologies like service description languages, e.g. WSDL, service composition, service management and service discovery to the cloud. It's an abstraction of the underlying computing model and does not deal with implementation details. Specially Google App Engine can bring existing Java web application and web services to the cloud. The cloud could be also used to test web services under load since it's easy to allocate a lot of instances and use them as workers in performance tests. The authors of [20] analyze how grids and clouds relate to each others and see *evident similarities* between them. Specially resource virtualization is a mayor topic in both areas.

Using clouds also brings new challenges. The multi-tenancy requires a strict and solid security model to protect the stored data and applications from malicious access. Security is one of the key issues which hinder cloud computing from a broader adoption. Also the costing model of the cloud should be questioned. The initial costs might be low and the infrastructure requires less intensive maintenance, but costs for computing and data transfer are influential on the total costs. Proprietary protocols and management tools make data integration harder and requires the adoption of provider-specific APIs. Service level agreements (SLAs) are a guarantee for the quality of service in the cloud, but have to be in the right granularity to be expressive enough for the required qualities. Though, [20] sees no efforts made to standardize or to improve interoperability for PaaS.

2.3 Scalability of Cloud-based Applications

The autonomic acting components in a PaaS cloud are a central component for the auto-scaling features and self-healing of the overall system. [21] searches for improvements in these autonomic components for a cloud offering. In PaaS systems automatic middleware components perform self-management of the cloud node and so of the whole cloud. This self-management functions can influence a cloud node's overall performance. Adaptive computations collect the state of managed objects and decide if adaptions are necessary. A PaaS user wants a high performance with low costs / low resource consumption, whereas a provider wants a good resource consumption without over-provisioning, which reduces costs of operations. The authors of [21] propose a control loop with the functions *monitor*, *analyze*, *plan*, *execute*. Each function can be started or stopped by various components or can be time-driven. They implemented their adaptive computation framework as middleware for PKUAS, a JEE application server. In practice the CPU and memory consumption where the two critical hardware resources between PaaS business functions (like running an application container) and the adaptive computations middleware.

Where scalability focuses primarily on the ability for growth of a system, elasticity also takes the dynamic reduction of resources into account, according to [9]. Testing for elasticity with workload generators could be performed based on historical usage data and with a test configuration database as input. [9] identifies the following metrics among others for scalability and elasticity: response time, throughput, CPU and memory usage, overhead provisioning, total capacity of the system, SLA violations, energy consumption, and availability. They also see energy consumption as one primary theme in cloud computing. Horizontal scaling on one side and vertical scaling on the other are both methods to ensure elasticity of a system. A different

approach would be migration of whole virtual machines and applications from one physical server to another to ensure a dynamic provisioning of resources. The various provisioning algorithms can be triggered in a proactive / predictive or in a reactive manner, but there are also mixtures possible to ensure a good overall provisioning with minimal over- or under provisioning.

Measuring elasticity and scalability depending on various workload scenarios is discussed in [22]. They defined numeric scores for the elasticity of a system for easier comparison and to evaluate the performance of a provisioning algorithm. JMeter is used as workload generator to generate traffic for an EC2-based server application. The TPC-W webserver benchmark, which is marked as obsolete now, was used as server application scenario.

2.4 Approaches for Scientific Computing

A lot of research in the area of cloud computing focuses on the usefulness for solving research problems and complex scientific tasks. Scientific computing involves mostly computational-intensive tasks, which require a good CPU or GPU performance and don't follow the traditional web application design patterns. So works focusing on solving scientific computing tasks in the cloud normally focus on the pure performance of each virtual server instance and not on web application technologies or fast networking. [23] focus on testing a PaaS middleware layer for cloud computing and how it works for scientific computing. It describes *Aneka*, a PaaS software platform to run distributed applications on different underlying virtual (e.g. EC2, Azure) or physical (private cloud) hardware resources. The paper compares how the *m1.small* and *c1.medium* EC2 instances perform on the *CoXCS*[3] classifier. Since the mayor difference between the two EC2 instances is the number of cores (two instead of just one), a single-threaded classifier like CoXCS does not benefit from the improved computing power. Another benchmark involves the image generation for magnetic resonance imaging workflows and compares the execution time on EC2 with jobs running in the *Grid'5000*. Here EC2 performs better at a higher load scenario then the grid version of the benchmark, but also notes that the costs of the grid might be low for scientific institutions. It summarizes that the pay per use basis is a mayor benefit for cloud users to cut costs and that this is an important advantage for scientific computing. So focusing on App Engine as a PaaS solution with a strict pay per use model, it would be only a useful platform for computational expensive tasks if it delivers enough CPU capacities for a reasonable price and if the overall pricing stays low. This also requires an efficient pricing model for storing data in the Datastore. Though, since the write operations of data are cost intensive in App Engine, solving the tasks which [23] shows in case studies might be problematic with the web application stack of App Engine. Special PaaS solutions for scientific computing look like a more promising option.

[15] states that commercial clouds are not built to support typical scientific workloads and are optimized to handle web application scenarios. This connects to the thought that App Engine is not made to perform computational expensive tasks. Average scientific clusters are built around 32 nodes and this number seems to be stable. [15] discusses IaaS offerings since "the scientific community has not yet started to adopt PaaS or SaaS solutions[15]" and simulates scientific workload scenarios inside a public cloud. Different IaaS providers and their various instances are

[3] CoXCS: A Coevolutionary Learning Classifier Based on Feature Space Partitioning

compared to each other. It showed that the effective GOPS[4] realized on EC2 for floating point and double-precision floating point are eight times lower than the theoretical GOPS. Also the for scientific calculations important double-precision float performance is not consistent over different arithmetic operations like addition or multiplication. However, the sequential I/O performance on EC2 was better than expected. For multi-machine benchmarks the performance of cloud solutions was below other scientific computing environments, specially if the number of cores was 1,024 or higher. The pricing of clouds was reasonably cheap for scientific computing, since the cloud does not require expensive hardware acquisitions, special maintenance and operations. The downside are the significantly higher computation times, which reduce the practical usefulness compared to traditional grids, but "Clouds are now a viable alternative for short deadlines[15]". An IaaS provider might not be the best solution for large long-term scientific computing, but can help to bridge short-term needs. Also new high-performance computing instance provided by the cloud services may improve the performance and bring more benefits[5].

> "Our main finding here is that the compute performance of the tested clouds is low. Last, we compare the performance and cost of clouds with those of scientific computing alternatives such as grids and parallel production infrastructures. We find that, while current cloud computing services are insufficient for scientific computing at large, they may still be a good solution for the scientists who need resources instantly and temporarily.[15]"

The authors of [24] focus on the challenges of scalable applications in the cloud. Effective load balancing algorithms are essential to provide good scalability and efficient VM allocations. Another factor is network scalability to dispatch the traffic and to provide enough bandwidth to all applications. Cloud applications should be able to set up custom networking and take advantage of smart networking services. The PaaS model allows developers to build applications in a fast manner by using predefined services and libraries. One problem for server scalability on IaaS are lacking mechanism for describing relationships and QoS requirements between provisioned VMs. Though, application providers want to be free of infrastructure management and concentrate on the actual application development. If a cloud user gets very fine-grained configuration options a scalable system might get hard to manage. Two approaches help to deal with this: a) increasing the abstraction level of APIs; or b) a higher automation degree. An elasticity controller could work on per-tier level or globally as a single controller for a whole application. An elasticity controller can be parameterized by user-defined rules, which then trigger actions. In IaaS environments the number of VMs behind a single load balancer can increase rapidly. At the presentation tier a load balancer might choose between different strategies for handling CPU-intensive or bandwidth-intensive applications. Specially enterprise applications often use hardware appliances for load balancing. Networking over virtualized resources is typically done in two ways: Ethernet virtualization or TCP/IP virtualization. To improve the quality of the network scalability a user could provide a network description to optimize resource allocation and to prevent over-provisioned networking. [24] sees PaaS providers have to find

[4] Giga Operations Per Second

[5] [15] covers IaaS providers around mid-2010.

ways to scale their whole platform according to the running applications on top of it. This is not the case for IaaS providers, since they can focus on the VM provisioning and related services, but don't have to care about the software running on top of them. A PaaS involves two core layers: the container layer where user applications run and access services; and the data storage layer to store persistent information. To improve scalability of the container layer, hard isolation between different user environments has to be implemented. Running multiple containers in parallel is one scaling strategy. This can be done automatically by the platform itself and requires no user involvement, but has to be considered by developers. Automatic scaling of containers requires stateless applications running inside of it. Stateful components should be offered by the platform and require intelligent load balancing to the right container and correct request forwarding. Caches like *memcached* act as service to share state at application level. Database scalability can be provided by distributed caching, NoSQL databases or by clustering a database. Caches speed up frequent data queries by first looking for data in the cache before performing a costly database query. For example, App Engine offers an easy-to-use distributed cache for applications. NoSQL solutions have looser consistency guarantees than traditional RDBMS, but usually provide better performance under high demand. NoSQL systems normally do not offer a transparent support for ACID-compliant transactions. Distribution of replicas of data is one problem for database clusters. Complicated or long-running transactions impact performance since data might not be visible or locked to other instances of an application. [24] describes scalability strategies for PaaS providers by looking for solutions at the application's execution container and on the database level. There is a area of tension between providing standard SQL solutions in the PaaS offering with high consistency, but on the other side NoSQL-based systems would allow a PaaS to scale easier and provide a better overall write performance.

> " It can be concluded from this section that replication of databases/components is the most important issue to consider when scaling a PaaS platform. [...] Regarding PaaS, the "ideal" scalable platform should be able to instantiate or release instances of users' components as demand changes, and transparently distribute the load among them. To ease developers tasks, the concept of session should be implemented by the platform, which requires support for (transparent) data replication. But data replication requires that component instances keep an updated copy of the data. The necessary updates consume bandwidth and time, and can lead to big delays on requests processing. An equivalent problem is found at database level. A PaaS platform should ideally give access to traditional relational databases with support for ACID transactions. But as more replicas are created to attend an increasing demand, the overload necessary to keep consistency will induce delays on requests. The ideal PaaS cloud must balance the need for powerful programming abstractions that ease developers tasks with the support for transparent scalability. [24]"

One problem for a potential cloud customer is the high number of providers with diverse pricing models and various system configurations. To find the best matching provider with the best QoS is a hard problem with multiple factors to consider. Therefor, automatic and semi-automatic tools can help in the selection. Model-driven development can help to find a proper level of automatization. SPACE4CLOUD, which is presented in [25], is a model-driven approach for the

evaluation of performance and costs of cloud systems. For a useful evaluation of different cloud offerings a general model for them is needed. The extracted meta-models (Cloud Independent Model (CIM), Cloud Provider Independent Model (CPIM) and Cloud Provider Specific Model (CPSM)) are used for a mapping to the Palladio Component Model (PCM), which then can be used to evaluate performance and costs. "Palladio has been developed in order to design and simulate the performance of web applications running on physical hardware resources.[25]" The CIM represents a component based application independent from the underlying cloud hardware. The CPIM sits between the CIM and the CPSM and represents a general structure of a cloud environment. Classes from a specific CPSM are related to abstract classes of the CPIM, e.g. a "EC2 Availability Zone" class is a subclass of an abstract "Availability Zone". A CPSM holds all details about performance metrics and cost estimations. [25] delivers with the CPIM an abstract model which can describe cloud platforms detailed because it has classes to model services, free quotas and costs, SLAs, elements / components of a platform and scaling policies. A CPSM can be seen as an instance of a CPIM and some services can be classified as IaaS and some as PaaS. The classes of the Palladio model are mapped to the CPIM classes, e.g. a *Cloud Resource* is mapped to a Palladio *Resource Container*. The authors then used the *SPECweb2005* benchmark to test an application on a real EC2 instance. For light load the estimated response times by Palladio were in the range of the real system, but the model estimates very conservative for high loads. The Layered Queue Network used by Palladio needs to be improved to work also with cloud environments.

[26] especially looks at App Engine for scientific computing. App Engine's App Master controller starts applications servers according to traffic patterns and the current load situation. Each application runs in the sandbox and has no permissions to access the underlying operating system. App Engine is also optimized to handle short running requests and terminates requests after 30 seconds of activity. CPU time is defined like it's equivalent to the time of a Intel x86 standard processor, but this doesn't reflect the actual CPU an application is using. To perform distributed scientific computing, the authors propose a master-slave framework running on App Engine. The master applications is controlled by the user and calls slave applications running on App Engine via separate URLs. The scalability is handled by App Engine and the master collects the calculated results from the slaves. There exist three types of errors: quota exceeded, offline slave applications and loss of connectivity. Development slaves use the App Engine SDK for Java, but every development server scales by starting new threads instead of new instances. To share data across multiple slaves the Datastore is used. The PaaS environment fits well to execute Monte Carlo algorithms, which employ random number generation to improve the performance of computationally expensive problems in deterministic systems. The authors used a well-known Monte Carlo approximation of the constant π. Others benchmarks include a Rank Sort and a Mandelbrot calculation. Before starting the experiment, an optimal configuration has been searched. It turned out that around 30 to 60 parallel HTTP streams where the best compromise between performance and overhead. Multiplication and division performance was quite good on App Engine, but the addition and subtraction performance was behind a local development setup. A caching benchmark showed that App Engine servers utilize a three-level cache: The L1 cache had 32 KB, L2 256 KB and L3 was 8 MB, which would fit for an Intel Nehalem processor. It was clearly visible that long-running instances had a better performance, because the just-in-time

compilation ran and the JVM was hot. One problem for the experiment was that two consecutive requests could be handled by completely different instances. The 30 seconds execution limit has been avoided by reducing the problem size of the π approximation. When the slaves got closer to the 30 seconds limit, the error rate increased. The authors of [26] conclude that traditional scientific computing and grids are still the better performing options, but clouds can be a cheap and fast available alternative.

2.5 Monitoring and Performance Testing

[27] collected performance traces, analyzed them for special time patterns and evaluated the potential migration of three applications with trace-based simulation. They focused on the longterm performance of cloud services and monitored them over two months. The raw data was extracted with a data scraper from *CloudStatus*, which provides around thirty performance indicators for AWS and App Engine. Since App Engine was in a very early stage at this time and had serious changes, e.g. introducing the high-replication Datastore and other improvements to the platforms, the numbers of the study are outdated. Though, it's notable that the type of application has a major effect on performance. This supports the proposition that App Engine is only a good choice for web applications without special requirements.

2.6 Simulating Cloud Environments

One of the core problems for every research on provisioning cloud computing systems is the lack of scale. This is the result of the costs which arise for long-term tests and which are a strong limitation for a number of distributed tests. Simulations are a way to work around this problem, but they also have a limited expressiveness for real-world cloud applications. Though, simulations can provide interesting results regarding resource allocation and scheduling tasks. [28] uses simulation for finding good QoS targets and ideal provisioning strategies. It describes a provisioning technique to adapt cloud environments to challenging workloads to offer guaranteed QoS to users. Aspects of networking itself are ignored because they are not disclosed to customers of cloud applications and the simulation is insufficient at this point. The provisioning process itself is complex, it has to take care of QoS requirements and at the same time it should be as efficient as possible by not allocating too much resources. Fulfilling QoS targets is necessary to meet SLA commitments. Potential trapdoors are: a) Estimation errors for the need of resources; b) Highly dynamic workloads, where unexpected workload peaks from customer applications occur; c) Uncertain behavior of networking. The PaaS layer of the proposed mechanism for VM provisioning in [28] includes a workload analyzer which inputs information into a load predictor and performance modeler. The predictor then passes information to the actual application provisioner which starts and destroys the VMs. Because IaaS providers do not provide detailed information about the underlying hardware, the load predictor and performance monitor cannot make assumptions about low-level hardware and networking. The simulated workload in [28] scenarios were: a) web workload based on Wikipedia usage data and b) scientific workload with computational expensive tasks. Using Wikipedia as a input for the simulation is an interesting

approach, since the web site is one of the most visited sites in the world, according to the Alexa traffic rank[6].

2.7 Security in Multi-Tenant PaaS

Securing multi-tenant PaaS platforms is important to keep user code inside the container and to prevent excessive resource allocations by malicious applications. The authors of [29] focused on two popular technologies for clouds: Java and .NET. One problem for current cloud offerings is that users perceive them as risky environments. The provider has to isolate the different application containers from each other and cannot rely on users conformance. It's necessary to force security restrictions and policies to the hosted applications. Typical PaaS applications don't run directly on the OS level and have a vendor-specific layer between. This layer ensures platform standardization and portability, abstracts complex tasks and runs interpreted languages, which are popular specially in web development. A general purpose OS also doesn't have the capabilities for detailed scheduling policies and resource management. This required restrictions can be implemented in the container management.

The Java platform has a well-known history as platform to provide applications running in containers like for Enterprise Java Beans, Servlets or OSGi. Also Java includes a special security model, sandboxed execution of code, bytecode verification and custom class loaders. A PaaS provider can control which classes the JVM is allowed to load and abort the loading if it detects malicious class repositories. Also strong access control mechanisms to limit file system access and network access are provided. But there are also limitations by running multitenant applications as Java threads. To enforce isolation three options are described:

- One JVM per application, which is very resource intensive, but separates them on process level.
- Custom class loaders, but they don't prevent problems like leaked static references and thread termination issues
- Multitasking Virtual Machines (MVMs) which implement the concept of isolation at JVM level and every task has its own memory heap and does not share objects.

MVMs may generate a considerable overhead and try to share as much resources as possible to reduce this effect. Normally static variables are shared through all threads of a JVM and in an MVM they have to be stored in a separate space per thread. There is also a Java Specification Request (JSR 121) for a Java Isolation API, but this JSR has not been implemented so far. MVMs look like the most complete solution for isolation and security problems. Another approach is KaffeOS, which isolates processes inside the runtime and manages CPU and memory resources provided to each process. Also Sun/Oracle worked on heap partitioning at the JVM level to prevent faulty software from acquiring the full heap memory and causing other applications to crash. Another problem in the JVM is the lack of resource accounting to threads, so one thread

[6] `http://www.alexa.com/topsites`

can acquire memory and CPU without limitations. With JVM profiling a basic accounting would be possible in theory, but profiling generates a lot of overhead and requires native code. One approach is bytecode rewriting, where code is injected to account resource usage and to prevent malicious access. The work on resource management and accounting APIs has lead to a new JSR 284 Resource Consumption Management API. Another security hazard is the lack of safe thread termination in Java. A malicious thread can block other threads and acquire a huge amount of memory. One approach is to inject bytecode to check threads during their execution, but this is a performance-intensive operation. A MVM prevents this problems by seamlessly stopping applications without any overhead. One problem with stopping Java threads is that if a thread stops, all monitors it holds are unlocked and released. Any object relying on the synchronization by these monitors can be in an inconsistent state at that time.

Instead of running software directly on a JVM, PaaS clouds could use container technologies on the JVM level and provide standard services. A container itself should provide a good availability (one container cannot lead into resource starvation and resource sharing policies are enforced on containers) confidentiality and integrity (applications cannot influence / modify data owned by other applications). JEE containers from the EJB specification cannot create threads on their own, cannot manipulate files, cannot modify class loaders and are not allowed to have final static fields. EJB containers enforce this through the standard Java security model. This limitations influence the programming model and do not allow many development needs. On the other hand, the Servlet specification does not provide any isolation strategies and only includes authentication and authorization.

One very prominent PaaS offering a Java container runtime is App Engine. It applies the Java security model and uses custom class loaders, uses security policies and applications cannot create threads on their own. Applications are not allowed to access a number of classes and cannot modify the filesystem. Each application has it's own JVM and so isolation is solved in this way. A long running thread is terminated after a limit of 30 seconds of request time by an exception and it can react on this exception by providing a custom response to the client or GAE will send a 500 error back and forces a thread termination. Other threads are not affected by the termination because the load balancer in front of every GAE application redirects requests to good instances. If all threads are stopped on an instance, the instance can be shut down. GAE applications have to be stateless since requests by one user might be distributed to different instances by the load balancer and state can be only shared via GAE services. This makes it easy to start and stop new instances, but developers have to be aware of this effect.

The OSGi (Open Service Gateway initiative) framework defines a Java-based modular system and a service platform. It enforces isolation through its module layer, which allows developers to specify if packages of a bundle are shared or hidden to other bundles. Some of OSGis security hazards only can be fixed at JVM level. Another platform for PaaS could be .NET by Microsoft. According to [29], the basic security mechanisms can be compared to the Java platform and are implemented by the Common Language Runtime (CLR). The CLR implements isolation with Application Domains (AD), so each application is assigned to one AD and cannot access code of other ADs. Static variables are not shared over domains and are isolated. Threads can be aborted, but this is not recommended and works quite the same as in Java. The Aneka project tries to provide PaaS containers for .NET, which provide services and management for applications.

One way to enforce security on PaaS is the consequent monitoring of the execution of external code. If code performs malicious operations, it can be stopped. These monitors can only work on external level by observing the operations, or they can inject itself into the running code. The idea of weaving trusted code into untrusted one is originally used in Aspect Oriented Programming (AOP), which provides mechanisms to define crosscutting concerns over a whole application. AOP is already used for authentication and authorization.

2.8 Performance Evaluation

We have to distinguish three different approaches to performance testing.[7] The typical performance test measures the response time, throughput and resource utilization during a predefined test procedure for a typical production load. Tester try to find inefficiencies and bottlenecks in the system before it goes live at various load levels. It gives feedback if the system's performance is ready for production or not, but it does not test for bugs. Typical problems are inefficient or improper algorithms, missing database indexes, query optimization, kernel parameter tuning and network routing improvements. Normally performance tests are white-box tests where all internal components of the system are known and monitored. A load test can be seen as part of performance tests, but has a stronger focus on the actual load a system can handle and not on meeting certain quality requirements. It tests if a system can handle a critical job which is not in the normal workload spectrum properly. Load tests search for the upper bound and are not focused on service level agreements or non-functional concessions. The results provide indicators how long a system runs stable and where a critical load starts. The third category of tests are stress tests. This tests try to expose bugs which only occur in a system under heavy load. Stress tests force systems to go to the maximum, they might break it and provoke unpredictable states. A test procedure might include a high level of randomness to find otherwise unpredictable bugs. An important result is if the system under test is able to recover from such an extreme situation. A stress test might include extreme scenarios like disconnecting necessary network interfaces, cutting hardware connections and implanting invalid internal states. It's like putting destructive evil Gremlins[8,9] into the heart of an application and see what happens.

Performance tests are system tests. Instead of testing one small unit like unit tests do, they hit an application as a whole. User interactions are rebuilt in test tools and simulated in a short period of time. There are different approaches for the test execution phase[30]. Randomized tests analyze structural properties of a system and generate automatic tests based on a simple randomized algorithm. Constraint-based testing simulates the execution of a program with symbolic input parameters to form a set of concrete and valid input values for a given program. Search-based techniques generate tests through machine learning. Invalid and valid inputs are used to enhance the machine learning task and test evolve over time, but the generated values are potentially unrealistic compared to real world executions. In contrast, model-based testing derives test cases from a given model to cover a program specification. The basic idea is to check

[7] `http://agiletesting.blogspot.ch/2005/02/performance-vs-load-vs-stress-testing.html`

[8] `http://www.imdb.com/title/tt0087363/`

[9] gremlins.js testing framework, `https://github.com/marmelab/gremlins.js/`

if the underlying code fulfills the specification and is a correct implementation of the model. Only model-based techniques can autonomously verify the output of a program. All others are focused on the input value generation, which requires no knowledge about a programs results and need another instance for verifying the output. Such an authority is called test oracle and is hard to generate in an automatic manner. Test generators may only look for an exception-free execution of the tested program and ignore the output. This avoids the presence of a proper test oracle, which is hard to generate.

App Engine is a closed source proprietary platform. Only the SDK itself is open-source software and hosted on Google Code. Since the SDK only provides a development application server with emulations of the core services, this is not relevant for a performance test working with the production environment. So to get reliable and valid results, developers need to deploy the application in the cloud and have to write HTTP-based tests.

Recent reports indicate that PaaS will be an increasing cloud model, but only a few studies have focused on it. To make rankings and evaluations possible, this paper discusses a framework to evaluate the performance of cloud providers. "It also proposes a suitable set of benchmarking algorithms that can help determine the most appropriate PaaS provider based on different resource needs and application requirements. Performance evaluations of three well-known cloud computing PaaS providers were conducted using the analytic hierarchy process and the logic scoring of preference methods.[31]" Recent studies have focused on IaaS and evaluated how good the CPU and memory performance of applications was, a lot of papers with the focus on scientific computing problems. Good performance metrics and benchmarks are an open research issue for PaaS. [31] proposes 19 different functions grouped in three main categories to evaluate PaaS solutions and compare commercial PaaS providers with two methods: Analytic Hierarchy Process (AHP) and Logic Scoring of Preference (LSP). The performance evaluation of the database / datastore technology is integrated into the general evaluation and not separated like in previous studies. The AHP method uses pairwise comparison and calculates a relative ranking from the bottom up to the final result. LSP defines for each performance variable a valid range and the elementary preference E indicates the variable satisfaction for the needs of a specific benchmark. To measure CPU performance the authors of [31] use the Whetstone benchmark to get numbers for MFLOPS and MOPS on each platform. As metric for the datastore benchmark they measured the average response time for certain operations for a single result and for multiple query results. The benchmarked PaaS services were Heroku, OpenShift and CloudFoundry. They showed that OpenShift's CPU performance is significantly lower than the other two platforms. For the database performance Heroku showed slow response times and OpenShift and CloudFoundry were about five to six times faster. Like at the CPU benchmark, Heroku and CloudFoundry delivered a good memory bandwidth and OpenShift was noticeable behind. The AHP and LSP rankings showed CloudFoundry as a clear winner, since it was ranked on the first position for all scenarios. Heroku and OpenShift alternate on the runner-up position. All three services have to improve their database stability. The authors conclude that the LSP method is more suitable for PaaS services than AHP.

[32] shows that performance tests have to distinguish between light and heavy operations. All tests were executed on the Microsoft Windows Azure cloud platform. They elucidate that e-business systems consist of a lot of light operations. However, heavy operations like submitting

orders or invoicing can also appear in a high frequency, depending on the business' scale and volume. The JMeter-based tests in [32] showed that using larger instances instead of small ones is preferable over a higher number of less powerful instances in a scalable e-business environment. They used a single desktop instance of JMeter for load testing.

[9] classifies the different experimentation types to measure elasticity of cloud applications in the following categories: experimentation, analytical modelling and simulation. These three categories also form the main area how cloud application load tests could be performed. Benchmarks are in the experiments in the real world environment, but they might be also performed in laboratory conditions with more possibilities to watch and analyze the results. Since Amazon CloudWatch is widely used together with the EC2 IaaS service, this service is also an important component to provide auto-scaling on EC2. [9] also shows that App Engine is not one of the most examined services and far behind the Amazon ecosystem. This might also be the result of the lack of support for scientific computing and PaaS nature of App Engine. They also identify the YCSB as a tool for performance comparison of cloud databases and found with Httperf, Tsung and JMeter three specific load generators for load testing.

CHAPTER 3

Related Work

Benchmarking and load testing cloud applications is a basic requirement before offering an application or web service to the public. Though, there exists no common standard or procedure to test applications in the cloud[33], but there exists a number of tools and workload generators. [33] compares five different cloud benchmarking tools: CloudCmp, CloudStone, HiBench, YCSB, and CloudSuite.

CloudCmp[34] compares public cloud providers in multiple dimensions. The chosen provider have been Amazon AWS, Microsoft Azure, Google App Engine and Rackspace CloudServers. The authors describe the internal networking of these cloud providers as one of the invisible and intransparent components of each cloud platform. To test this area, they implemented TCP-based tests revealing throughput and latency between nodes inside the same datacenter infrastructure. The CloudCmp benchmark compared different metrics. For benchmark tasks exist standard benchmark suites like the SPEC CPU2006 benchmark, but this was not applicable on a sandboxed platform like App Engine is. Therefore, the authors use the SPECjvm2008 benchmark which runs on all compared solutions. Moreover, they had to limit the runtime of the benchmark to 30 seconds and to restrict the benchmark to one single thread to fit limitations by the examined platforms. Knowing that App Engine enforced exactly that requirements at this time, the reason behind these constraints could be App Engine, but the authors don't mention explicitly this. The CloudCmp benchmark measures the runtime of the individual tests and calculates the cost per benchmark based on these runtime values. Since App Engine uses a different billing system based on CPU time and not on instances, the authors queried the App Engine billing API to retrieve cost per benchmark. To avoid any connection overhead issues during data storage benchmarks, CloudCmp's client uses a persistent HTTP connection to the different storage backends. A very interesting part of the storage benchmark part is the time to consistency. This is important in eventual consistent databases, where results of queries may not be consistent with the latest write operation. Since App Engine applications and their corresponding instances are not publicly available over an IP address, the authors used the Google frontend servers as endpoint for ping-based tests. The authors anonymized the results due legal reasons, but it's obvious that App Engine is hidden behind pseudonym *C3*, since it's the only service with CPU per hour billing

and without any information about the used cores and with only a default instance type at the rate of $0.10 per CPU hour. Also C3 had as the only service no access to the local disk and no support for multi-threading, both applies to App Engine at the time of the comparison. Since the pricing model behind App Engine and specially the Datastore changed during the last years, the results shown in [34] are not representative today. Due the PaaS nature of App Engine an extended testing is harder and comparison with IaaS services seam to be harder and only partly useful.

A more general approach to cloud benchmarking for IaaS solutions is presented in [35], which formulates challenges a cloud benchmark framework has to solve to be broadly accepted by the community. They see a disruption between traditional benchmarking techniques and those suitable for IaaS cloud environments and that solid benchmarks help to gain trust in IaaS solutions. Benchmarks are an important factor during migrations into the cloud, since they enable to see a before and after view and are testing non-functional requirements of the system. A distinction between synthetic and real-world application based benchmarks is important, since they enable different conclusions and have a different level of abstraction. Long setup times of IaaS clouds are a problem for a generic testing framework and can be adjusted with a reduced input set. Another fundamental challenge is the generation of massive workloads with multi-site benchmarking. Checking a single instance in a cloud may not be representative and only has a limited expressiveness. Costs by multi-site geographically distributed workload generators are crucial for the total cost of running a benchmark. It's also hard to isolate the real performance of benchmark jobs due the distributed environment. The workload itself has to be realistic and configurable. Using real workload traces to generate more abstract and tunable workload models is a complex task. The lack of established metrics makes it harder to obtain comparable and exchangeable results. Cloud-related metrics for elasticity and scalability exist, but are not widely accepted by the scientific community and still under development and review. The issue of a neutral and comparable cost model is not yet solved and different billing models makes it harder to agree on a set of established cost models. Resource acquisition and provisioning strategies vary on different cloud providers. Where Amazon and its EC2 require a noticeable boot and installation time for new instances, App Engine uses fast responding containers. This makes comparison of both services hard. Just-in-Time compilers for Java can improve application performance over time, so long-running App Engine applications may obtain a performance boost over time. The field of fragmented cloud solutions and very provider specific technologies is a hard to tackle problem. The identified main challenges in [35] are:

- *1. Experiment compression. (Methodological)*
- *2. Beyond black-box testing through testing short-term dynamics and long-term evolution. (Methodological)*
- *3. Impact of middleware. (Methodological)*
- *4. Reliability, availability, and related system properties. (System)*
- *5. Massive scale, multi-site benchmarking. Cloud-bursting. Co-allocation. (System)*

- *6. Performance isolation. (System)*
- *7. Realistic yet tunable models of workloads and of system performance. (Workload)*
- *8. Benchmarking performance isolation under different multi-tenancy models. (Workload)*
- *9. Beyond traditional performance. Elasticity and variability. (Metric)*
- *10. The cost issue. Relate with metrics such as utilization, throughput, and makespan. (Metric)*

Since the special nature of processing genomic datasets, [36] analyzed existing cloud platforms by Amazon and Google for their suitability for that computation task. Both are not dedicated to perform extensive genome sequencing tasks and are general purpose cloud solutions with a variety of instance types. They found that both cloud solutions are suitable to handle these kind of tasks in a cost-effective way. The clock speeds of the used CPUs have an effect on the results, so that Google Compute Engine had an advantage over Amazon's EC2. The initial data transfer into the cloud services is still a problem due the nature of large genomic datasets.

That the costs of running complex jobs in commercials clouds can be quite high is shown in [37]. The ATLAS project at CERN used Amazon AWS to calculate jobs for their experiment and got a $200,000 grant from Amazon to test this IaaS cloud. To get the enormous amount of data to Amazon's datacenters, they needed special peering between their network and EC2 datacenters in the US. This helped to reduce the costs for traffic to a reasonable amount of money. Overall the test showed that commercial clouds can compete with their own infrastructure from an economical view. In subsequent tests the ATLAS project will evaluate performance with 50,000 and 100,000 cores acquired from EC2 for tests. If these deliver positive results, the ATLAS wants to incorporate commercial cloud offerings like in this case EC2 into their standard infrastructure. CERN uses the HEP-SPEC06 benchmark, which is optimized to simulate integer and floating-point computations relevant for high energy physics (HEP). This benchmark is primarily targeting CPU performance and does not require a high amount of memory. So additional memory will not improve the results, whereas a increasing CPU speeds should. They evaluated Google Compute Engine and Amazon EC2, but also two academic clouds hosted by Compute Canada. For the results they run the benchmark 50 times inside their custom CernVM and averaged the results. Both commercial offerings resulted in similar results for comparable instance classes.

Traditional benchmarks do not fit modern cloud environments according to [38], since there are new cloud specific attributes like elasticity. The authors see cloud benchmarks as a combination of traditional benchmarks and application testing. Tests are initiated by clients which access the cloud applications and are not running on local instances anymore. Furthermore, tests simulate various workload scenarios and different activities like re-deployments and VM management operations. At the moment there exists no established and accepted standard for benchmarking clouds: "While waiting for standard benchmarks to appear and gain acceptance, we have built our own benchmark to evaluate cloud systems, along the lines described above.[38]" Instead of testing public clouds, the authors limit their approach to private clouds. They focus on user-centric metrics, which are described as representation of real-user experience. Existing

benchmarks are often based on throughput and response time as major metric. Using only average and median of these two metrics doesn't fulfill a user-centric approach, since it overlooks a large number of potential users of a system. Instead *N-th percentile of response time* offers a better metric, because it shows how many percent of users are actually experiencing a good or bad response by the system. The 90th percentile is identified as popular yardstick for benchmarks, but still N-th percentiles ignore behavior like general response time variation. Durability of transactions is another important user-facing property of a cloud solution. Transactions should not be lost and persistent. Furthermore, the availability and reliability form two additional relevant properties which are important for users. Cloud system often promise to improve these two areas compared to traditional setups. Since cloud solutions share physical resources over multiple instances, workload isolation and security are important attributes for users of cloud-based applications.

Another approach is taken in [39]. A/B testing is a difficult task in real-world environments because of interdependencies between user requests and host servers, which violate assumptions by statistical tests. [39] describes how Facebook uses benchmarking tests internally to prevent performance regressions after deployments. Their approach is based on distributed testing against a distributed environment with a lot of possible code paths and highly varying user backgrounds. A/B testing of different versions of a software can lead to false-positives, where performance bottlenecks are detected even if they don't exist, and there can be errors in the test which allow performance regressions to stay undetected in an A/B test. Both problems are the result of the complex environment, since Facebook highly customizes the results for each user and both sets in A/B testing need to include user edge cases as well. The results of [39] have been validated with Facebook's internal performance testing tool Perflab, which performs A/A test between the old and the new version of a software in an isolated non-production environment. The workload for the test has been extracted from Facebook's real traffic to be representative.

The CARE framework presented in [40] distinguishes between the raw processing time on the server of a cloud solution and the overall response time. Response times also include the client's network latency and transfer times between the client and the cloud provider, whereas the processing time is the time used to fulfill a request on the server. The presented framework supports two different load test strategies:

- **Low Stress** This strategy sends sequential requests to the server without any overlaps or parallel processing.
- **High Stress** Parallel requests are sent to the cloud platform to gain more insight in the overall architecture and scaling properties. The framework is initialized by an initial number of concurrent threads and a round counter. The number of threads depends on the current round and increases from round to round.

During the development of a common framework for cloud testing, [40] describes a major problem lying in the heterogeneous nature of current cloud architectures:

> "Providing a common reusable test framework across a number of different clouds is a very challenging research problem. This is primarily due to the large variations in

> architecture, service delivery mode, and functionality provided across various cloud platforms, including Amazon Web Services, Google App Engine, and Microsoft Windows Azure. Firstly, the service models of cloud hosting servers are different: Amazon EC2 uses the infrastructure as a service model; Google App Engine uses the platform as a service model; while Microsoft Windows Azure combines both the infrastructure as a service and platform as a service models.[40]"

Their Datastore tests had problems with the entity group write limit and long update times, but this could be a result of the principal design of the Datastore. Sequential updates on the same entity cannot be triggered in a high number. Instead, the data model should avoid the requirement of more than one update per second per entity group. During the high stress strategy test [40] discovered problems with reaching the actual App Engine application and had indicators that their requests have been blocked by the Google frontend servers as a protection. Also they ran into quota limitations, but there is no referenced to the billing model used for App Engine tests.

A main topic in the IaaS market is benchmarking and comparing different VM types and instance classes to find the best matching offering for an optimal price. [41] performs VM-level benchmarks to compare different VM types. The authors also considered the requirements raised by applications, but ignored the different cost models of cloud providers. Their goal was to find a model to predict the best matching IaaS solution and they validated their model with real workloads on the VM instances.

[42] discusses Cloud WorkBench (CWB)[1], an IaaS benchmarking service which follows the paradigm *Infrastructure as Code* (IaC). It means infrastructure of a system is described in a code file and not only provided by manual or semi-automatic configuration. This makes it easier to reproduce and test the infrastructure itself. Repeatable benchmarks also allow a continuous evaluation of the provider's offerings. In CWB a human tester describes a benchmark in a web interface using a provisioning service. The built-in IaC tools are Vagrant and Opscode Chef, both are popular in the DevOps community. A benchmark is always defined in an IaC language and requires an additional output model definition to capture the desired results. Each benchmark can be executed regularly by a scheduler, or manually by the user. The overall result are modular and repeatable benchmarks, which can be executed very flexible.

[43] provides an overview over App Engine and the Datastore. The authors provide a brief introduction to Google's cloud offerings and discuss the underlying technologies. The Datastore is a more complex NoSQL datastore than others, due it's table-like-orientation and the special key concept forming entity groups. [43] also introduce the Google Query Language (GQL) for the Datastore and shows some use cases for it.

[1] `https://github.com/sealuzh/cloud-workbench`

CHAPTER 4

Google App Engine

Google App Engine is a fully-managed web application hosting service. Instead of installing a web server environment and to manage an application's underlying services, the whole software and networking stack is provided by Google. This conforms to the PaaS cloud computing model, where a cloud provider delivers hardware and software services to its customers. A developer only has to upload an application into the App Engine service and it will be deployed, managed and executed automatically. There is no renting of a virtual server or housing of any hardware involved. The physical server setup and network routing is hidden from the developer and provided out of the box. The traditional App Engine service gave developers only a minimal influence on the infrastructure serving their application, but Google is evaluating *Managed VMs* to give back more control over the hosting environment. The Managed VM service is still in beta[1] and is not covered by any SLA policy. Therefore, most applications running inside the App Engine environment still use the fully-managed approach.

The primary goal of App Engine is to host highly scalable HTTP-based applications inside Google's cloud infrastructure. An incoming HTTP request is routed to the corresponding customer application and the response is returned to the client. Typical clients include web browsers and mobile services on smartphones. If an application is under heavy load, App Engine spins up new instances of it and automatically redirects traffic to provide a nearly uniform traffic distribution. Instead of adding more powerful hardware, App Engine follows a horizontal scaling strategy. Newly created instances should work off additional traffic. Instances itself run inside software containers, a technology to isolate applications from the host operating system and to enable true multi-tenancy per host. The raw performance of each instance is limited by its instance class. Each instance class has a different level of CPU power and limited memory. Running instances are hourly billed. So App Engine is not a good choice for number crunching or other traditional computing in the cloud. Instead it is a very sophisticated hosting service for web applications and all integrated services, e.g. Datastore, Search, and Memcache, are optimized for typical web application requirements.

[1] It was first announced in March 2014. `http://googlecloudplatform.blogspot.co.at/2014/03/bringing-together-best-of-paas-and-iaas.html`

One of the basic principles is to pay only for resources consumed by the application. There is no basic fee or other fixed costs for standard applications. Google also provides a free tier to try out App Engine and to run simple applications without a lot of load. So potential customers can evaluate App Engine if the service fits their needs before the actually move into this proprietary environment. Though, enabling billing is required to securely serve applications with HTTPS / TLS over custom domains, which should be an essential requirement for every real world website these days. Hosting applications especially in European datacenters also requires a billing account and is not available for free. Applications with the EU-hosting option are served from European data centers and the Datastore replicas will only be inside the EU. This option could be useful for applications which need to comply to local laws and which need a different data privacy standard than standard applications.

Google distinguishes three basic types of support for features and APIs. Features available under General Availability (GA) have active support and are fully covered by the SLA agreements for App Engine. Any change to these components will be implemented in a backwards-compatible manner. Beta features are not yet stable and there is no guarantee that they will become available under GA. They can be used by all App Engine users and their APIs might change with breaking changes. Alpha features are not yet stable and there will be changes until they become available under GA. Some alpha features are also limited to a smaller circle of developers, which are invited to test them before they get publicly available.

4.1 Application Modules

Each application consists of at least one or more modules. Modules form a logical part of an application and are responsible for a group of tasks. E.g. it can be useful to split up an application into a web module, which is responsible for HTML-based rendered pages, and into a mobile API as backend for apps. These two modules could be developed independently, but still live under the same application sharing a common set of data and code. Every module can be configured to use a different App Engine runtime and has its own configuration, so that auto-scaling and other settings can be modified on a per-module basis. An instance is always responsible for serving one module at exactly one version. Splitting up an application into different modules also will lead to more active instances.

It's possible to only update and deploy one single module. So if only one modules changes, the overall application does not be updated with all modules it includes. Each module is publicly available via an appspot.com URL by default. This access can be limited in a special routing configuration, called the *dispatch file*. It contains a list of URL patterns and which module is responsible for a matching pattern. The patterns can also consist of custom domains and are not limited to the appspot.com hosting. On the development server, multiple modules run all under localhost, but with different ports. The assigned ports are logged in the development server's log file at startup.

The module is also the scope for the instance class that App Engine should use to process a request. For auto-scaling modules a different set of instance classes is available than for manually and basic scaled modules. Manually and basic scaled modules have the advantage that their requests can run up to 24 hours before they time out. Automatically scaled modules have a hard

limit of 60 seconds for incoming requests and 10 minutes for cron-initiated requests. So the module concept also has deep consequences for the scaling strategy. Using the old Backend API is deprecated, which means it can be shut down in the near future, and it has been replaced by the Module API.

4.2 Request Routing and Scaling

The PaaS approach of App Engine handles requests differently than other hosting environments. The application itself does not run at the frontend edge to the user. Instead there is a separation between the application and its instances and the user-facing frontend. This enables the App Engine infrastructure to scale up frontend servers independent from the application instances.

Each incoming request starts at the Google Front End servers (GFE)[2]. These servers will be close to the client which sent the request. The GFE is connected to a local cache where it stores cacheable responses from applications and static files. If an incoming request can be fulfilled by a locally cached resource, the GFE might not forward it to the App Engine application. So developers have to carefully choose the cache headers in an application's response, otherwise the response will be cached by the GFE and not be up to date. Though, there is no guarantee that responses will be cached.

The application inside the chosen hosting runtime is running in a separated data center. Requests will be routed to it by the App Engine Frontend, which has all information about running applications and how to reach the according instances. On a single App Server can run multiple instances of different App Engine applications. The App Server is fully managed by Google, which includes software updates and hardware maintenance. It's important that each application can run in automatically scaled frontend, or in manual / basic scaling instances. Frontend instances are lightweight, autoscale, and respond very fast. Their creation is fast and cheap, whereas manually scaled instances are heavyweight. Their instances are more static, have their own instance classes prefixed with „B“, and can be created by choosing the basic or fully manual instance scaling type for a module[3].

The actual scaling strategy for instance creation is controlled by the App Master component, which has the control over the physical servers which execute different applications inside App Engine runtime environments. The App Master orchestrates the load balancing and informs the routing frontend about newly created or soon to be destroyed instances. It's also responsible for the deployment process and the version management[3]. The deployment usually is fast, but it's not atomic. So during the rollout of a new version of an application, the previous old version can be served from some instances as the default "latest" version. This behavior is important to take care of, since updates to the data model might not be visible to all active instances in the moment of the deployment. The App Master only guarantees that an instance is always serving a consistent version of an application and that all new files are in place before switching to the latest deployed version. Before a new instance gets available to the public, developers can configure warm-up request for each instance. Such requests should ensure that the application is fully

[2] A introduction is provided in this Google Developer video: `https://www.youtube.com/watch?v=QJp6hmASstQ`

[3] `https://cloud.google.com/appengine/docs/java/modules/`

responsive and ready to serve before actual requests are routed to them. They are initiated by the App Master after setting up the instance and right before the master adds an instance to the frontend routing. But there is no guarantee for the correct order of the warm-up request before any user-initiated request:

> "There are a few rare cases where an instance will not receive a warm-up request prior to the first user request even with warm-up requests enabled. Make sure your user request handler code does not depend on the warm-up request handler having already been called on the instance.[3]"

There is no API available to find out the number of running dynamically scaled instances. The Modules API provides this number only for manual or basic scaling modules. This makes it difficult to monitor an application's instances, because the number is only available in the developer console.

4.3 Hosting Environments

The developer can choose between two hosting options. The first is the sandboxed hosted environment, which is available with various language runtimes for Java, PHP, Go and Python. The developer provides the application code or executables and the App Engine container starts up with an immutable language runtime, which initializes and executes the code. The sandboxed hosting has a number of restrictions in the language runtimes. For the Java runtime Google implemented a strict sandbox, which prevents any writes to the local filesystem and terminates long-running requests before they consume too much resources. It also prevents any system-level calls from Java, disables the native JNI interface and enforces a strict class whitelist. Language features like reflection are limited to the application's own classes. This deployment option requires less configuration work from the developer and the runtime works out of the box.

App Engine sandboxed instances itself are software containers on top of Linux running inside Google's infrastructure. A software container isolates the applications it contains from the operating system. The container's underlying container engine manages a container's lifecycle and provides services to the applications. It also enforces memory isolation, CPU quotas and other restrictions to ensure strict isolation between the operating system and containerized applications. One big advantage of containers is their fast startup and destroy time. A container can be launched in seconds, whereas a full VM launch can take minutes.

The second option are Managed VMs[4], which deploy App Engine applications into Google's Compute Engine virtual machines. It's a combination of the PaaS service App Engine and the IaaS service Compute Engine. Managed VMs do not enforce any language runtime and their related restrictions, but instead of utilizing software containers the service is based on traditional virtual machines and hypervisors. This brings a lower-level security mechanism and separates the hardware from the software stack. Managed VMs can be derived from Google's standard runtime images, which provides built-in support for the App Engine APIs, or a developer can provide its own configuration. The latter provides more flexibility, but none of App Engine's

[4] `https://cloud.google.com/appengine/docs/managed-vms/`

APIs is available out of the box and services like the Datastore need to be addressed over a REST API. Developers can provide custom binaries, have normal access to the networking, can write to the local filesystem[5], and child processes and background threads are not prevented. The most fundamental difference to sandboxed runtimes is the semi-automatic auto scaling. Since a Managed VM takes minutes to boot, not only seconds like sandboxed runtimes, at least one instance must be present and healthy at any moment. Developers can further define a minimum number and a maximum limit of instances. Instances are then created following a simple CPU-based strategy. If a defined average target CPU utilization over all active instances is exceeded, new instances will be created to handle the additional demand. The CPU checks are performed in a configurable interval, but by default the instance manager checks every 60 seconds for the current CPU utilization. Health checks are regularly performed on each instance to ensure that the application stack was successfully deployed and is still healthy serving requests. Each instance of an application has to implement a health check handler[6] which has to return a simple HTTP 200 response. The interval and check timeout can be configured in the VM configuration. Managed VMs with a failing health check will be removed from the serving frontend and be restarted after a predefined consecutive number of health checks fails. Instead of just simple ping-like checks, Managed VMs can implement a full application check and therefore provide fine-grained control over instance management.

Managed VMs are still in beta and therefore they are not covered by the App Engine SLA or Google's deprecation policy. Furthermore, European Union hosting is not available and all instances will run in the US. This limits the use for customers with strict hosting compliance rules. Since instances are not as lightweight as in the sandboxed runtime, security updates on the guest operating system are performed during a controlled restart on a weekly basis. Managed VMs can host *standard runtimes* based on Google-provided Dockerfiles or *custom runtimes* configured by the developer with a self-supplied Dockerfile. The main advantage of standard runtimes is the out of the box App Engine API support, whereas custom runtimes require the use of publicly available REST APIs of Google's cloud services. The Managed VM standard runtime for Java[7] is feature-equivalent with the sandboxed runtime, so it too supports only Java 7. Its main advantage is the reduced number of restrictions and therefore the easier integration of existing libraries and additional software into the App Engine surrounding. Applications are not required to follow the specific architecture of sandboxed runtimes and gain more freedom. The practical usage is unfortunately narrowed by the missing SLAs and the expectable changes until a first stable version will be announced.

appspot.com Hosting

appspot.com is the default hosting location for applications running on top of App Engine. Every application uploaded is available via an appspot.com subdomain, in the form *application-id.appspot.com*. Even if the developer specifies a custom domain name for the application, it's

[5] The writes are only ephemeral and will be lost after an instance terminates.

[6] The reserved URL endpoint for health checks is */_ah/health*

[7] `https://cloud.google.com/appengine/docs/java/managed-vms/`

still available on AppSpot. The only way to prevent accesses via appspot.com is a manual check for the incoming request URL and a subsequent redirect.

4.4 The Java Runtime

About one year after the initial launch of App Engine, with Python as the only supported language, Google presented the Java runtime in April 2009[8]. This enabled developers to write applications in Java and other JVM-based languages, like Scala, JRuby, Groovy, or JavaScript via Mozilla Rhino. The JVM implementation for App Engine is a proprietary JVM by Google, with some additional security layers to prevent malicious code escaping the sandbox. It's not published which parts are taken from the original open source OpenJDK[9] project and where Google replaced parts from the original JDK with own components. The virtual machine identifies itself as "OpenJDK Client VM" with Google as vendor of the Java runtime environment and standard library.

To write applications for the Java runtime, Google provides a special App Engine Java SDK. The SDK integrates into the normal Java ecosystem and includes plugins for the Eclipse IDE and for Apache Ant. For Apache Maven a separate plugin is provided, which can handle the full development cycle of an application. For an easier start Google created some Maven Archtypes[10], which are skeletons for new applications and reduce the initial work to start with the development. The Java SDK ships with a local development server. This server does not include a full App Engine environment, it simulates a number of complex APIs and the security restrictions. During local testing the development server also detects queries to the Datastore API and automatically creates an index configuration file. With the AppCfg tool applications can be deployed to production and managed. AppCfg is also used to manage index configurations and to configure DoS attack prevention.

Servlet Container

The App Engine Java runtime environment implements a Servlet container. It runs web applications by invoking them using the Servlet API and takes care of the request and response handling. Servlets are components to serve HTTP-based Java applications. Since HTTP is a stateless request-response protocol, operations in such components are processed by a single method and cannot rely on other method invocations before or after its own execution. This reduces the complexity for a single request, but it requires a more verbose session management and special mechanisms like browser cookies or session parameters in URLs. Servlets build the foundation of the majority of Java web frameworks and are one of the oldest enterprise technologies in the Java Enterprise Edition.[44] Though, App Engine's Java runtime supports only version 2.5 of the specification, which has been released for JavaEE 5 and JavaSE 5 around 2005. The newer versions 3.0 and 3.1 are still not supported. App Engine provides special services to provide

[8] `http://googleappengine.blogspot.co.at/2009/04/seriously-this-time-new-language-on-app.html`

[9] `http://openjdk.java.net`

[10] `http://mvnrepository.com/artifact/com.google.appengine.archetypes`

workarounds for some features of the newer specification, like the Channel API for real time messaging between the server application and web clients.

Java-based applications have to follow the standard WAR directory structure to enable their deployment. The application code itself is stored in a `/WEB-INF` directory. Files located in this directory won't be served as static files to clients. It also contains index configurations and the standard `web.xml` configuration file. All other files and directories in the WAR archive will be served as static files starting at the application root URL. A very basic requirement of every Java-based application is the presence of App Engine's Java SDK in the classpath. Otherwise API calls will fail and the application won't work outside the development environment.

Security Architecture

The software security researchers from *Security Explorations* revealed a number of security issues in the Java runtime of App Engine in 2014 [45]. Their technical report shows sandbox bypasses and included sample code to reproduce the problems. One of the main criticisms were the modifications Google applied to central components of the JRE and the JVM.

> "The other goal is to show the very tricky nature of Java security and especially the pitfalls one can easily get into if custom Java Runtime modifications are applied to certain security sensitive Java APIs and components.[45]"

The reports describes the basic security architecture, with a lot of information not disclosed by Google. Due an agreement with Google, Security Explorations only released issues found in the Java layer, not at any lower layer they discovered. The sandbox environment has two layers. One is a native sandbox at the operating system level, which limits the visibility of operating system functions to user applications. The other is implemented on top of Google's JRE and should prevent malicious code being executed by the JVM itself.

The Java layer restricts many core functionalities and garbles standard parts of the JRE. User-provided code has read-only access to it's own base directory, but cannot read any other files from the file system. Sockets can be created, but only outbound in a non-listening way. Incoming connection sockets listening on certain ports are not possible. This also a result of the principal design of the request routing in App Engine, where a dedicated frontend server handles the request and routes it to the corresponding application instance. Outbound connections via `java.net.URL` are redirected to the URL Fetch service. Threads will be terminated at the end of the response cycle and the security layer takes care that no thread can outlive its original request. Threads exceeding the request time limit or potential deadlock threads are terminated automatically: "But, this even goes further as GAE makes sure that no user code gets executed after HTTP request has been handled. This includes all sorts of system Java handlers and finalizers in particular.[45]" Registering a shutdown hook will fail with a `java.security.AccessControlException`[11] thrown by the security layer. Applications cannot provide a custom security manager implementation, but they can supply their own custom class loaders.

[11] `https://github.com/ringo/ringojs/issues/243`

The Java runtime's class loading process involves a static class analysis step, which takes the raw input stream of the bytecode, analyzes it and then writes it back to the intended location. It inspects the class to load before it gets available in the runtime environment, like if the class conforms to the JRE class whitelist. If non-conforming code is detected, the class is rejected and an exception is thrown.

Creating new threads is possible in the Java runtime, but there exist some restraints for multi-threaded applications. Though, spawning new processes or running background threads is prohibited. As a workaround developers can choose App Engine's `ThreadManager` to create background threads for backend modules, which are independent from frontend modules.

Deployment Limitations

There are several restrictions for Java applications deployed on App Engine. They are a result of the underlying cloud infrastructure and the goal to isolate user applications from each other. Files deployed on App Engine cannot be larger than 32 megabytes and there is a limit of 10,000 files per application[3], which includes configurations, code and static resources. If larger files are needed, others hosting services like the Blobstore API or Google Cloud Storage. As discussed in the security section, the Java security layer inspects class files and intercepts class loading for security scans. As a result digitally signed JARs are not supported. The URL Fetch API can only download up to 32 megabytes, which limits its use cases compared to normal URL connections.

4.5 App Engine Datastore

The App Engine datastore is a schemaless NoSQL database, which has support for basic atomic transactions, provides strong consistency for reads and ancestor queries in the same entity group and is optimized for high availability. Even though the datastore is a key-value store, it offers a SQL-like query language for filtering and sorting data.

The underlying basic technologies are Colossus, a scalable distributed file system, and the Bigtable distributed key-value store. Colossus is the successor to the Google File System (GFS) and ensures reliable low-level storage. It offers methods to store data, but is not a database system and has no advanced query capabilities. Bigtable stores data in tables with rows in lexical order by keys. This allows fast range queries, but there is no support to search for column values. Until 2009 the datastore ran only on top of Google's Bigtable system, which supports row-level transactions, but not distributed transactions. This has been a problem together with Bigtable's column-based replication. If a transaction operates on row-level for a datastore in datacenter A, but the changes are replicated on column-level to datacenter B, inconsistencies are a problem during a switch from A to B. Bigtable gives the control over the data layout to the client and provides only the underlying replication and key-based access functionality. Internally all data is stored as a string and Bigtable clients have to convert the data into more specific data types. Google described the App Engine datastore's underlying Bigtable system in a developer article[12]. It's not clear if this is still valid for the currently used evolution of the datastore, since it has been

[12] `https://cloud.google.com/appengine/articles/storage_breakdown`

introduced at this time and the article doesn't mention any of Google's new lower-level storage systems.

Datastore used master/slave (M/S) replication from the beginning, but this sort of datastore has been deprecated and should not be used for new applications. M/S replication keeps the commit latency low compared to distributed commitment protocols, but if the master datastore fails it causes unapplied writes. Those writes got lost between the master and its slaves and have been visible to users in the datastore admin viewer. Since the slave's asynchronous replication can be delayed minutes after the master, the number of unapplied writes can be high for large-scale applications. Unapplied writes could not be processed automatically, so applications developers had to decide what to do with those writes to ensure consistency. To keep applications running during an outage, the M/S datastore could be switched into read-only mode, which happened for example in May 2010[13]. The read-only mode is also necessary to move the master from one datacenter to another, since a complete flush of outstanding writes is required and takes some minutes. So even during planned maintenance, every application relying on the M/S datastore has a short unavoidable read-only period. Unexpected short periods of errors or unavailability can occur if the underlying Bigtable needs executes internal management operations to optimize further data access. During this management operations RPC calls could fail and requests queues grow, but the overall App Engine status is not affected.

A more sophisticated storage backend is the high replication datastore (HRD). It has been introduced in January 2011 after several poor performance periods and outages of the M/S-based datastore. HRD uses synchronous replications for a majority of replicas and implements true multihoming across distributed datacenters. Majority votes require an odd number of participants, so at every write at least three independent Datastore instances are involved.

> "Cloud Datastore replicates all data to at least three places in each of at least two different data centers.[3]"

To finalize a write, all non-synchronous replicas get updates asynchronous. Traffic between datacenters can be rerouted easier than with the asynchronous M/S datastore and the reliability in case of failure is higher. Since the majority-related replication happens synchronously, a request will be only fulfilled if the data has been written into the required number of datacenters needed to ensure a permanent write. An interesting detail is that reads are done from the fastest available replica, which could be a remote and not the local datastore instance. Google promises nearly the same read-latency for the HRD datastore compared to the M/S datastore. Since typical web applications have more reads than writes, this should keep applications responsive and fast. On the other side, HRD applications have to keep the number of writes as low as possible. Another benefit of the HRD is the significantly lower error rate for reads and writes. Even if the underlying Bigtable cells start some local management operations, the applications accessing the datastore will not be affected at all. This is in contrast to the M/S datastore, where these operations will be noticed due to increased error rates.

After introducing HRD Google also changed the App Engine Service Level Agreement (SLA). It requires customers to choose HRD for their datastores to be accepted as „Eligible

[13] `https://groups.google.com/forum/#!msg/google-appengine-downtime-notify/XYzjDvolnM4/-6Y9SeZOQekJ`

Application“ for the SLA. The old M/S datastore has been deprecated and should not be used for new applications. There are no guarantees for high availability and according to the deprecation policy, the general availability of the M/S datastore is limited to „at least one year after that (deprecation) announcement“.[14]

Megastore

To gain consistent replication across datacenters, which is required for the HRD, the datastore has been switched to Megastore, a Google-internal library on top of Bigtable. Megastore is not a replacement for Bigtable, it's an additional layer which handles distributed transactions and ensures reasonable latency for synchronous writes across datacenters[46]. This allows data to be read in a consistent state from various datacenters, not relying on a single master. From the RDBMS vs. NoSQL perspective Megastore sits in between both worlds:

> "[...] we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support.[46]"

One downside is "its relatively poor write throughput"[47] since the true multihoming approach Megastore implements. App Engine itself implements a custom logic to store schema-less datastore entities inside the strict schemas required by Megastore. Megastore supports transactions on groups of entities and is not limited to row-level transactions. It allows queries to distributed storage, has ACID-like transaction support for small sets of data and requires strict schemas. To achieve this with low latency, the consistency requirements across different data partitions are relaxed.

Each Megastore entity is mapped into a single Bigtable row and each Bigtable column name is a combination of the Megastore table name and the Megastore property name[48]. The Bigtable row key is exactly the key of the Megastore entity. We know that Bigtable entities are physically stored in the order of their keys. The App Engine datastore uses the ancestor path as key, so entities sharing the same root entity in their ancestor path will be stored next to each other. If Megastore wants to change entities inside the same entity group, it can access them quickly on the Bigtable server without complex distributed storage location lookups. This is the reason why the number of entity groups per transaction is limited and why a single entity group transaction will work faster than cross-group transactions.

> "Each Megastore entity group functions as a mini-database that provides serializable ACID semantics. A transaction writes its mutations into the entity group's write-ahead log, then the mutations are applied to the data. [...] A write transaction always begins with a current read to determine the next available log position. The commit operation gathers mutations into a log entry, assigns it a timestamp higher than any previous one, and appends it to the log using Paxos.[46]"

[14] `https://cloud.google.com/terms/`

To implement a distributed consistent storage, a custom implementation of the Paxos algorithm is used. Paxos is a fault-tolerant consensus algorithm without the need of a master. As long as a majority of participants of a group has consensus on a state, a faulty minority cannot spread their incorrect states. Specially for a data backend it's important that a majority agrees on which data has been persisted and what position of a log is the last consistent state of the overall system. At its core it's similar to two-phase-commit or three-phase-commit protocols. If data is written, a majority of participating storage servers have to acknowledge the durability of the write. The other participants can be updated asynchronously to improve the overall performance. Instead of using Paxos only for master election or simpler replications of metadata and configurations, the algorithm inside Megastore is used to replicate whole entities with their user data and metadata. Google optimized its implementation of Paxos for reduced latency during cross datacenter writes, which makes it suitable for storing user data and distribute it over continents, and it even works at global scale. The key for this was reducing roundtrips between far apart from each other datacenters. Also the whole protocol does not require synchronous Paxos across different instances of the Datastore. Instead Google uses an asynchronous background replication, which ensures lower write latencies and still keeps a very high consistency during a datacenter switch.

Data Structure of Entities

Every object stored in the Datastore is called *entity*. An entity contains a key field with a number of sub-fields forming a unique identifier for each object. The Datastore groups entities into logical units called kinds. The kind is not a separated feature of an entity, it's part of the key. Assigning different kinds make it easier for application developers to assign akin entities into groups, but there is no schema enforced on the data itself. So an entity's properties are only associated with the entity itself and entities of the same kind can have completely different properties at the same time. This distinguishes the Datastore from relational databases and other NoSQL datastores.

Each entity holds data as multiple key-value based properties of different data types. Such a property is accessible via its key, sometimes also sloppily named "column name", and holds a single value or multiple values, comparable to an array of values. Property keys which are used to retrieve a property's value, should not to be confused with the surrounding entity's key.

The entity key is Datastore-wide unique. This enabled Google to place *all* entities of the whole Datastore in only one single Megastore table. This makes the management of the Datastore easier, but also introduces a single point of failure for all applications hosted on App Engine. If this single Megastore table sees some inconsistencies, all connected applications will see problems.

Entity Keys

An entity has exactly one unique key. From a developers point of view the key is constructed out of a namespace, an ancestor path, a kind and an ID. App Engine itself automatically adds the project ID of the application as prefix to each key. This makes it possible to store all entities ever created by any App Engine application into the same Megastore table. So the whole Datastore service uses only a very small number of actual Megastore tables for indexes and data, which makes maintenance easier and keeps the whole system compact. An important property of each

entity is that the key is immutable. Modifying the key and saving the entity afterwards will result in a new entity, leaving a zombie entity with the old key behind. Compared to primary keys in the traditional relational data model, Datastore keys are not constructed out of the properties of an entity. They are important for retrieving an entity by key, but they don't include any data by them self.

To allow multi-tenancy even at in-application level, applications can also add a namespace to each key. Requests and code with a programmatically attached namespace can only access entities in the same namespace. This is useful to enforce a stricter isolation between multiple tenants inside the same project. App Engine for Java supports transparent multitenancy by configuration for application instances. Developers can provide a namespace in the *web.xml* for each tenant and the datastore will automatically be isolated. Another approach is to set the namespace before the request gets processed by the application inside a servlet filter. There is no special method to set a namespace directly for a key or query, which makes it easier to enable multitenancy afterwards. If entities have a parent, they share the same namespace with their ancestors. An entity stored in namespace A is not accessible in namespace B and vice versa. The number of namespaces is unbounded, but namespace name strings have a maximum length of 100 characters.

Entity					
Key					Data
Project ID	Namespace	Ancestor Path	Kind	ID	Protobuf Serialized Properties

Table 4.1: The principal structure of an Datastore entity.

An important property of an entity's key is the ancestor path. To describe it in detail a look at the overall key space is necessary. The Datastore key space is a set containing directed tree graphs. Every single key of the Datastore is a node in such a tree. Each tree graph represents the hierarchy of entity keys which are part of the same entity group. The ancestor path itself is a single directed path inside a tree, starting at the tree's root and ending at the key it's part of. The shortest possible ancestor path for a key is the empty path, which implies that the key is the root of a tree. There is no limit how long an ancestor path can be. Ancestor paths are immutable, so keys cannot be moved out from one tree and merged into another. There is no requirement that keys which are part of an ancestor path must point to an existing entity. In practice application logic should only allow ancestor paths to be created if each element of the path represents an existing entity. Such paths are not only valid, they are also verified and consistent. Inconsistent ancestor paths occur if entities are deleted and their key is still part of an ancestor path. Such ancestor paths keep their validity, but lose their consistency. Ancestor paths are very important properties of the key if an application enters a transaction or if a query needs strongly consistent results. For example, in a school administration system a student entity could have the ancestor path *[School:exampleschool, Class:1A]*. This would mean he is part of the entity group *School:exampleschool* where *exampleschool* is the named identifier of an entity of the kind *School*. Every entity which's key has the root key from the school in its ancestor path can be part of the same transaction. The Datastore can handle up to five entity groups inside the same transactions, but not more. To retrieve a strongly consistent list of all students in class *1A* a query has to provide the class' or the school's key as ancestor.

The kind of an entity groups it together with all other entities of the same kind. It has to be an arbitrary string, whereby the double-underscore prefix "__" is reserved for Datastore-internal kinds and cannot be used. Inside the set of entities with the same kind each entity's ID has to be unique. The kind can be compared to classes in object-oriented programming, but instances of the same kind are not required to share the same properties. On one hand this makes it easier to use existing entities without expensive data model upgrades, but it also means the developer is responsible to handle the blueprint for each entity. Though, frameworks like Objectify enforce stricter requirements to entities of one kind. The primary purpose for kinds are queries. A query can search for all entities of the same kind having properties with certain values. The kind of an entity also could be seen as the last path element in its hierarchical ancestor path.

An ID uniquely identifies an entity inside it's kind. It can be either an arbitrary string, or a number of the type long, but never both. An ID using a string as identifier is called *name* of an entity, otherwise we speak only about *the ID*. If an ID is numeric, it should be generated by the Datastore's build-in number generator to avoid collisions and to ensure uniqueness. Numeric IDs also must be a non-negative number, and the zero ID is reserved for internal use. There is no hard guarantee that the automatically generated IDs are strict monotonically increasing, but in practice they do. To improve the performance of the automatic number generator, it can be instructed to acquire a contiguous range of unique IDs[15]. Already acquired ID ranges are reserved and will not be used by any other call to the sequence generator. Using an already reserved ID from a generated range will fail, since the Datastore actively prevents ID collisions in this case. Auto-generated IDs are well-distributed integer numbers up to 16 decimal digits. If a developer wants to generate IDs in a special sequence, the application has to provide them to the automatic ID generator. Those IDs will be skipped by the generator and are available for assigning to newly created entities

Properties

An entity contains properties with stored values. Each property has a property name and at least one value. A property without a value is unset, though, *null* is a normal property value. The Datastore allows multiple value per property name. Such properties are called *multiple-valued properties* and are useful for equality filters in queries or to store array-like values into a single property. Every value of a multiple-valued property can be of a different value type. The basic property value types are integers, floating-point numbers, booleans, short text strings up to 1500 bytes, long text strings up to one megabyte, dates and binary data. Other types are constructed of these basic types and are more complex, e.g. geographical points, Google account email addresses and datastore keys. Only short strings can be indexed, whereas long strings only store plain text and cannot be part of queries. There is no full-text search available for text strings, but Google provides an external search service, the "Search API", to tackle this problem.

Entities and its properties are schema-less and schemas can be only enforced by third-party frameworks like Objectify. There is no guarantee that entities with the same kind always have the same properties. And even if the name of the property is equal, it could still have a completely different value.

[15] See the *com.google.appengine.api.datastore.DatastoreService* class for details

Entity Groups

Entity groups are the basic sharding mechanism in the Datastore. Each entity group is replicated by the underlying Megastore synchronously between datacenters. They link entities by their parent-child key relationship. Child entities are always in the same entity group as their parent. Entities without a parent are called root entities. So all descendants of a root entity belong to the same entity group. If an entity has no parent and no children, then it forms an entity group for itself. This also means an entity always is part of exactly one entity group. The entity group of an entity can be looked up in its ancestor path. The beginning of the ancestor path is always an entity group's root entity.

Seen from the Megastore perspective, entities which are part of the same entity group are stored next to each other, which makes it easy to update them quickly inside a single transaction. This is a result of the lexically ordered keys inside the underlying Megastore table. Since the entity group of an entity is directly derived from its key's ancestor path, the ancestor path sorts all entities. Therefore, scanning through a whole entity group doesn't require any lookups, expensive data retrieval, or jumps. This makes optimistic locking for transactions easier to implement. Queries which define an ancestor can be strongly consistent because of the fast nature of the simple table scan. Instead of locking data or searching through the whole data structure, an ancestor-backed query starts at the first entity with the given ancestor in its ancestor path and ends where the ancestor path doesn't contain the ancestor anymore. Now it's clear why entity groups and ancestor paths are such a different concept compared to traditional relational databases.

It's important to know that entity groups are per entity instance, not per kind or class. So entities of the same kind can be part of different entity groups, and every entity is part of exactly one entity group. Kinds categorizes entities by grouping them under a name, but don't reflect any consistency or a scope of transaction.

Entity groups don't reflect a strong relationship between entities, they define the scope of a transaction and they guarantee strong consistency for ancestor queries. So using the parent-child-relationship inside an entity group for queries is reasonable from a consistency standpoint. Though, it shouldn't be treated as a logical relationship between two entities, since the ancestor path of an entity cannot be changed after the entity has been created. The developer has to choose the boundaries of each entity group at the creation time of the instance of an entity. Moving entities between entity groups by changing the ancestor path requires creating a new entity and deleting the old one manually. Very fine-grained entity group boundaries can lead a larger number of cross-group transactions. Putting a lot of entities under the same entity group reduces the throughput and often results in a high Datastore contention. So it makes no sense to put every entity in the same entity group, just to update everything in a single transaction or the perform every query with strong consistency in mind. This will break the scalability of an application, since there is a global limit around one write per second per entity group.

Transactions

Choosing the right boundaries between entity groups is an important step during the data modeling phase. The entity group relationship forms the scope of a transaction. It affects how entities can be modified inside transactions and what kind of read consistency is provided by the datastore.

Table 4.2: Example for two different entity groups inside the Megastore table storing all entities.

Project ID	Namespace	Ancestor Path	Kind	ID	Data
project1		/	school	1220vienna1	[...]
project1		/school:1220vienna/	class	1A	[...]
project1		/school:1220vienna/class:1A/	student	100000000	[...]
project1		/school:1220vienna/	class	2B	[...]
project1		/school:1220vienna/class:2B/	student	100000001	[...]
project1		/school:1220vienna/class:2B/	student	100000002	[...]
project1		/	school	1010vienna1	[...]
project1		/school:1010vienna1/	class	1A	[...]
project1		/school:1010vienna1/class:1A	student	100000003	[...]
project1		/school:1010vienna1/class:1A	student	100000004	[...]
project1		/school:1010vienna1/class:1A	student	100000005	[...]
project1		/school:1010vienna1/class:1A	student	100000006	[...]
project1		/school:1010vienna1/	teacher	100000000	[...]
project1		/school:1010vienna1/	teacher	100000001	[...]
project1		/school:1010vienna1/	teacher	100000002	[...]
project1		/school:1010vienna1/	teacher	100000003	[...]
...	...	...	...	...	...

Every transaction is limited to a range of five entity groups for the HRD and only one entity group for the deprecated M/S datastore. Entities can only join a transaction if they are in the same entity group and if the limit of involved entity groups in a cross-group transaction is not yet reached. If more entity groups are involved, the transaction has to be split up. Cross-group transactions use the asynchronous messaging APIs of Megastore. Datastore transactions are idempotent, so executing the same writes multiple times will result in a identical and consistent state.

The transaction service uses optimistic locking to provide strong serial consistency. Writes inside a transaction are immediately journaled inside the corresponding entity group on the server where the write happened. For each write on a single entity a timestamp is logged in the entity group's journal. This phase is called *Commit phase*. The journals are internally used to guarantee replication and to manage transactions on entity group level. A journal only stores diffs between changes to keep itself compact. The commit phase is successful if all involved entity groups show the identical timestamp in the journal. If timestamps are not identical, another transaction intruded the transaction before it could be committed. Both Transactions will fail with a *ConcurrentModificationException*. A possible strategy to nonetheless apply the changes is to retry the transaction with a random delay. This does not guarantee, but at least has a chance to successfully finish the commit phase. Only the owner of the transaction sees the changes made during the transaction for a read, which are stored only in the corresponding journals. All other reads are diverted to the latest persistent Bigtable row version of the entity. This enables a high read rate with low latency even if the requested entity is modified inside a transaction. "Readers and writers don't block each other, and reads are isolated from writes.[46]" Since consensus has been reached, the client will be informed that the data has been written, even if it might not be

fully applied.

After the commit phase the *Apply phase* starts. The changes are applied on the actual entity data and all index rows for the entity are updated. Since index updates are asynchronous, persistent changes on an entity might not be immediately visible in queries on indexes containing it. So queries starting immediately after a write may not see the changes even if it's guaranteed that the write will be applied and all data is consistent. The more indexes an entity is part of, as much more time is needed to update them. This is one of the reasons why developers have to choose carefully which indexes they need.

Within an entity group all write operations are serialized and consistent. If a query spawns multiple entity groups (non-ancestor queries), strong serial consistency is not possible and applications might not see the most recent updates on different entities.

A typical case where a datastore transaction fails is the reach of the request time limit. If the request takes too long, App Engine aborts the processing, stops the transaction and throws an error.

Queries

The top layer of the datastore infrastructure is the datastore API itself. It takes care of the indexes, validates the requests and provides a query interface. Traditional relational database systems support queries on each column and only require an index to improve performance. This is fundamentally different inside the App Engine datastore. Since the underlying Megastore has only support for index scans, each property of an entity which is part of a query has to be indexed. Properties without an index cannot be queried via the datastore API, but the datastore automatically creates those simple indexes on each property. More complex queries involving multiple properties have to be predefined in an application's configuration, so composite indexes for them are available before the first access. Otherwise, queries over multiple properties will fail. During the development of an application the App Engine SDK will propose composite indexes based on the detected queries and developers have to add them to the production environment configuration.

Every executed query starts with a table scan starting at the beginning of the index and searches for the first entity that matches the query. From this entity it continues the scan until the last entity that matches the query is reached. Then the Datastore returns the resulting set of entities. The cost of memory and CPU of running a query is directly proportional to the number of results returned by that query. So indexes play an important role in the Datastore for data retrieval.

If every entity has multiple associated indices, the physical storage consumed by indexes is much higher than the actual entity itself. Index updates also cost CPU time and slow down write performance, so a developer can disable the index creation on properties in the configuration. It's not possible to perform joins over datastore entities, though denormalization is an alternative strategy for this problem. Also full text search is not supported, instead the Search API has to be used to run full text queries.

Indexes

Indexes are simple data structures that power data retrieval inside the Datastore. Compared to traditional RDBMS indexes, they are not only created to improve query performance and enhance retrieval operations. A Datastore index is essential to run an application and to perform a query, even if it's only a lookup by the key of an entity. They are always in exactly one of these four states: building, serving, deleting, or error. The building state indicates that the index is not yet fully constructed. If indexes are created after a large amount of entities exist, building up a new index on its properties can take some time. During an index is in the building state, it cannot be used by the application. Queries on a building index will raise an exception. So it's important to build new indexes used in queries before deploying a new version of an application which uses those new indexes. Deleting an index can take some time and during this process it's in the deleting state. Serving is the normal state of an index and during serving it's fully functional. The error state indicates that the index is not functional and cannot be used by queries. This can happen if an index is created for an already existing property, which e.g. ignores the 500 characters property limit for indexed strings. The number of total indexes per applications is limited to 200.

For every entity kind the Datastore keeps three built-in indexes:

- **EntitiesByKind** keeps all entities with their application id, their kind name and their id. It's used to lookup an entity by key. Saving a new entity automatically adds it to this index. It's necessary to include the application identifier in the key of this index, since the indexes are not created by application, though there is only one single big Bigtable responsible for all App Engine applications and their corresponding entities.
- **EntitiesByProperty ASC** holds each indexed property of an entity in ascending order. The index entry is composed of the application id, the kind name, the key of an entity, the property's name and the value of the property. Even null values require an index entry.
- **EntitiesByProperty DESC** is constructed like the corresponding ascending index, but in the reverse order. Each indexed property of an entity will be stored in both indexes.

Storing an entity involves index writes to at least these three built-in indexes plus one write operation for each composite index of the entity. Composite indexes are composed of different properties of an entity and can explode if they are not chosen carefully. Specially if an entity's property is multi-valued, for each of its values an index write is needed. Even single-valued properties can lead to composite index explosions if they are used several times in different combinations. Index writes can also explode if the ancestor path is long, since each indexed property needs to be stored inside an index next to each of its ancestors.

Index updates are performed in parallel to the apply phase of the data of an entity. This means that a non-ancestor query might use an already updated index, find matching entities, but retrieves yet not updated entity data. To prevent this condition specifying an ancestor will append the entity group's timestamp to the query and during the data retrieval the conflict will get visible and the Datastore ensures a consistent read operation. The easiest way to prevent such problems is to

read inside a transaction with a defined ancestor, since this ensures that all changes are applied to the index and to the data.

> "In rare cases, it's also possible for changes to indexes to be applied prior to changes to entities, and for the apply phase to fail and leave committed changes unapplied until the next transaction on the entity group. If you need stronger guarantees, fetch or query entities within transactions to ensure all committed changes are applied before the data is used.
>
> A query with an ancestor filter knows its entity group, and can therefore offer the same strong consistency guarantees within transactions as fetches. But many useful queries span entity groups, and therefore cannot be performed in transactions. If it is important to your application that a result for a nontransactional query match the criteria exactly, verify the result in the application code before using it.[3]"

Write Costs

Currently Google charges for each single write operation to an index. So write costs heavily depend on the amount of writes to indexes, since the entity write itself only counts one write operation. Keeping indexes compact and small reduces the total costs of running applications on App Engine. It is possible to create an index afterwards, so starting with a minimal set of indexes does not affect the introduction of new indexes after the initial deployment.

Index Management and Configuration

Every composite index can be manually configured using a XML-based configuration file *datastore-indexes.xml*[16]. This file contains information about every index the application will use for queries. A composite index can be of exactly one kind of entity and consists of two or more properties. If the index is used for ancestor queries, it has to be explicitly declared as ancestor index. For each property the sort order has to be defined. The sort order and sequence of indexed properties is important, since each query is only a table scan and the rows of the table are sorted in lexical order. The App Engine SDK can auto-generate the XML configuration, but it can be also configured to only use manually created index definitions. In the auto-generate mode, every time the SDK detects the use of a non-existing index, it adds it to the *datastore-indexes-auto.xml* file. This file contains all auto-generated indexes over time. During the upload of an application to App Engine, the application configuration tool AppCfg uses both *datastore-indexes.xml* and *datastore-indexes-auto.xml* files to determine which indexes have to be created for the deployment. It's a general advice to move indexes from the auto-generated XML configuration into the manually configured file. This ensures that the indexes will be present during new deployments. Though, unused existing indexes will not be removed by AppCfg during a normal upload. The maintainer of an application has to remove it manually, or running *appcfg vacuum_indexes* will remove all indexes not mentioned in the XML configuration files from the production environment. Since the total number of indexes is limited to 200, this might be an issue over time, when multiple versions on an application with different queries have been deployed to production.

[16] This only applies for the Java SDK, which is the primary topic of this thesis.

Google Cloud Datastore

Since 2013 the App Engine datastore is available as standalone product called *Google Cloud Datastore*[17]. It's accessible over a HTTP API and has the same features and characteristics as the HRD. Google also separated the datastore administration from the App Engine management console into an independent Cloud Datastore console. This console provides an overview over the stored entities and allows direct changes to single entities. The shown statistics are not updated in realtime, in tests they where around one day behind the actual datastore state and showed misleading entity counts. The console also has a simple query tool to search for entities stored in the datastore.

Google Cloud SQL

Even if the Datastore is a very sophisticated and fast data backend, its complexity and scalability is not needed for every kind of application. Also migrating from a traditional SQL-based Java application to App Engine is harder since its NoSQL nature. For this Google offers Cloud SQL, which is a managed MySQL relational database running inside a dedicated virtual machine instance. Cloud SQL can be accessed with a JDBC MySQL driver and can be managed with all common database management tools. Backups can be automatically configured and run on a daily basis and the service also allows secure connections via TLS/SSL from the outside world to the MySQL instance. Databases can be exported directly to Google's Cloud Storage service or with the normal mysql dump tool. Even Cloud SQL is a very comfortable alternative to the Datastore and easier to integrate, it makes it harder to scale in a million user basis from the beginning. But if applications are not considered to do so, it might be a better data backend then the Datastore.

4.6 Java Persistence API

The main purpose of JPA is to map objects to a relational database with an object-relational mapping component (ORM). The ORM has to reflect an object's state in the persistent database and in the entity inside the application. It manages the queries needed to synchronize state between both worlds, implements caches and other optimization layers. The ORM is the core technology behind every implementation of the JPA and may different implementations will follow other strategies in their ORM.

Annotations are the common way to define mappings and other properties for the ORM. They can be applied to a whole class, methods of a class and at single field level. The configuration with annotations can be very fine-grained if a complex mapping has to be specified. There are two basic categories for annotations. Logical annotations define high-level logical relationships and the object modeling view. Physical annotations are closely related to the database engine used behind the ORM and help to perform the actual mapping. This includes concrete table and column definitions, adding constraints and validations, connection settings and other database-dependent settings. An alternative to annotation-based mappings are XML mappings, but this separates the definitions from the actual implementations, and it generates a configuration overhead. On

[17] `https://cloud.google.com/datastore/docs/concepts/overview`

the other hand changes in the XML mapping do not require a complete recompilation of the application and XML configurations can override existing annotations.[44]

DataNucleus

DataNucleus is an open-source Java persistence framework to store Java objects into a relational database. It implements the Java Data Objects (JDO) standard and serves as a layer between the actual Java code and the database backend. DataNucleus and JDO are not limited to relational databases, but they are used mainly for this purpose. Currently also map-based (HBase and Cassandra), graph-based (Neo4J), document-oriented (MongoDB), and web-based (Amazon S3, Google Storage) data backends are supported. There is even support to store data into spreadsheet-based documents like ODF and XLS. DataNucleus also supports the Java Persistence API (JPA), which is a standard part of the JEE family. Google uses DataNucleus with a dedicated plugin to provide the JPA to Java applications running on App Engine. The maintenance status of the plugin is unclear, but it is officially supported and not marked as deprecated yet. Though, the future of it is unclear and it lacks of support and updates to newer versions of DataNucleus and JPA.

Entity Managers

Entity managers are associated with exactly one persistence context which manages entity states. They are responsible for the whole lifecycle of a single entity and load or destroy them. JEE application servers provide container-managed entity managers available to all application components. The App Engine Java runtime environment provides only a servlet container and not a full-featured JEE application server. Therefore *EntityManager* instances cannot be injected with the *@PersistenceContext* annotation. Developers have to manage entity managers manually using a *RESOURCE_LOCAL* persistence context. Google recommends to use a single static instance of *EntityManagerFactory* to create new entity managers, since the creation of a factory is a resource-intensive operation.

Entity Mappings

Table Definitions

By default an entity class will be mapped by its unqualified class name, e.g. *org.foo.bar.Employee* will have the corresponding table name *Employee*. The term *table name* from the relational database model is equivalent with a Datastore *entity kind*. Uniqueness constraints are ignored and a warning message is displayed in the application log file: “The datastore does not support uniqueness constraints. The constraint definition will be ignored.”

Field Definitions and Mappings

One of the most basic tasks of every JPA implementation is to map fields of classes to a corresponding column in a database table. JPA only allows non-public fields for a mapping. This prevents modifications of an object’s state without notifying the underlying ORM of the changes.

@Table	
name	works
catalog	ignored
schema	ignored
uniqueConstraints	ignored & warning

Since the Datastore does not support tables in a traditional relational way, fields will be mapped into columns of entities. The Datastore entity is the equivalent to a relational database table. An alternative to field mappings are property mappings. Basically they are the same, but instead of directly annotate a field definition, the getter and setter methods mark the properties to persist.

A simple type of mapping can be done with the optional *@Basic* annotation. It can be applied to persistent properties or instance variables. The *optional* value is supported and the ORM will throw an exception if the field is null and marked as non-optional. If no *@Basic* annotation is supplied for a field, it will be mapped with the default values of the annotation.

Every entity in the Datastore requires a unique key to identify it. This key is of the class *com.google.appengine.api.datastore.Key* and consists of a namespace, a kind, an unique numeric or string identifier, and an optional ancestor path to parent entities. If a key's parent entity is deleted, the ancestor path is still valid, even if no parent can be retrieved anymore. Ancestor paths do not require that keys which are part of the path are connected to an entity and stay valid if entities involved in the path are deleted. The GAE/J plugin for DataNucleus converts primary keys of the Java types integer, long, and *java.lang.String*, to a Datastore key without any notice by the developer. The Datastore entity will have two columns: one for the Datastore key and one for the mapped JPA primary key. The recommended way to define a primary key for an entity is to use the generation strategy *GenerationType.IDENTITY*, which will query the Datastore for free keys by calling *DatastoreService.allocateIdRange()*.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Key key;
```

The Datastore does not support *@Column* uniqueness and length constraints, but checks not-nullable columns. If a column is annotated with a unique column constraint, the DataNucleus plugin will throw an *UnsupportedOperationException*. The same effect has a secondary table name, which is not possible in the datastore. Length constraints are ignored and the default Datastore limit counts. The column definition properties updatable and insertable are supported.

One special Java class type are enumerations. An enumeration is a collections of implicitly static and final constants. They can be represented in two different ways: as ordinal integer number or as a string. For better maintenance the string representation should be preferred, since changes in the underlying class structure will not influence its value. Ordinal numbers will change if constants are introduced or removed, so the data stored in the database will be invalid after such changes. The Datastore has problems with annotated enum classes. Using the *@Enumerated* annotation in both variants *EnumType.STRING* and *EnumType.ORDINAL* throws an exception. The concrete problem with the ordinal *EnumType* is the type conversion from long to integer

@Column	
columnDefinition	ignored
insertable	works
length	ignored
name	works
nullable	works
precision	ignored
scale	ignored
table	exception
unique	exception
updatable	works

during the read from the datastore. Enumerations have to be used as is[18], without any annotations, or with a special converter for the enum class. Using a custom converter has the advantage to take control over the representation in the datastore, but also introduces redundancy.

```
public class VoteChoiceEnumConverter
    implements AttributeConverter<VoteChoice, String> {

    @Override
    public String convertToDatabaseColumn
        (VoteChoice voteChoice) {
        // VoteChoice is an enum class
        switch (voteChoice) {
            case NO_OBJECTION: return "N";
            case OBJECTION: return "O";
            case SERIOUS_OBJECTION: return "S";
            default: throw new IllegalArgumentException();
        }
    }

    @Override
    public VoteChoice convertToEntityAttribute(String choice) {
        switch(choice) {
            case "N": return VoteChoice.NO_OBJECTION;
            case "O": return VoteChoice.OBJECTION;
            case "S": return VoteChoice.SERIOUS_OBJECTION;
            default: throw new IllegalArgumentException();
        }
    }
}
```

[18] Like in Google's JPA test case for enums. `http://goo.gl/rVOsWy`

A special column type are temporal objects to store time and date information. JPA provides a *@Temporal* annotation to define a mapping strategy. In theory this maps fields of the types *java.util.Calendar* and *java.util.Date*, but the GAE/J plugin for the Datastore only allows the latter. If an entity uses a calendar, the date's timestamp and a corresponding timezone have to be stored. Older versions of DataNucleus required a two column representation for this task. This is not supported by the App Engine plugin and will throw a *PersistenceException*. In the newer version of DataNucleus a one column representation has been implemented as default, but it's not compatible with the older GAE/J plugin.[19]

@Temporal	
TemporalType.DATE	works only for `java.util.Date`
TemporalType.TIMESTAMP	works only for `java.util.Date`
TemporalType.TIME	works only for `java.util.Date`

Relationships

To put two ore more entities into a relationship and referencing each other, JPA provides a number of annotations. Every side of the relationship fits into a role, either owner object, or owned target object. Relationships in the relational database model can be unidirectional or bidirectional. The Datastore also defines two distinct types of relationships:

1. **Unowned Relationships** store a simple reference to other entities. The involved entities are not part of the same Datastore entity group and can exist independently. The GAE/J plugin does not monitor or manage the relationship.

2. **Owned Relationships** connect two entities which are part of the same entity group. This kind of relationship should be used if one entity is dependent on the other for its existence.[20]

Persisting relationships using JPA annotations are owned relationships by default. After an entity has been persisted, it's primary key cannot be changed anymore. Any modification of the key will throw an exception. Since the Datastore does not support joins, all features related to joins like specifying join columns are not supported by the GAE/J plugin. Using joins will throw exceptions indicating an unsupported operation has been started. The only fetch type possible is lazy loading, so collections will be only filled if they are accessed, otherwise they are empty.

One-To-One

One-to-One relationships work without any problems. They can be defined with different cascade types, which all work as expected.

```
@OneToOne(cascade = CascadeType.PERSIST)
```

[19] `http://www.datanucleus.org/products/datanucleus/jpa/types.html`
[20] `https://groups.google.com/forum/#!topic/google-appengine-java/7Xa8lNDZrOE`

```
private Person boss;

@OneToOne(cascade = CascadeType.ALL)
private Document passport;
```

One-To-Many

The add and remove operations for a one-to-many relationship worked without any problems. But if a parent entity is deleted in an owned relationship, the *CascadeType.PERSIST* cascade value is ignored and the owned child is also deleted. This is a result of the strong parent-child-relationship the Datastore enforces through it's key-value strategy, which is in contrast to the relational database primary-foreign-key relationship model.

```
// CascadeType.PERSIST is ignored
// This will behave like CascadeType.ALL in owned relationships
@OneToMany(cascade = CascadeType.PERSIST)
private Set<Person> friends = new HashSet<Person>();

@OneToMany(cascade = CascadeType.ALL)
private Set<Leaf> treeLeaves = new HashSet<Leaf>();
```

Many-to-One

This relationship type works in the same way as one-to-many, so the cascade problems are the same. Instead of using concrete classes, a workaround would be to use raw Datastore keys and manage the persisting of the according entities manually.

Many-to-Many

Many-to-Many relationships are only supported if they are unowned. The minimal cascade types required are *CascadeType.PERSIST* and *CascadeType.MERGE*. Persisting an owned many-to-many relationship will throw an exception:

```
HTTP ERROR 500
Problem accessing /ManyToMany. Reason:
Detected attempt to establish Employee(no-id-yet) as the parent
of Employee(6368371348078592)/Project(4960996464525312) but
the entity identified by Employee(6368371348078592)/
Project(4960996464525312) is already a child of
Employee(6368371348078592).  A parent cannot
be established or changed once an object has been persisted.
```

Embedded Entities

Embedding an object with the *@Embeddable* annotation into another is the strongest form of dependency. The existence of an embedded entity is tied to the existence of the entity holding it.

Instead of creating a one-to-one relationship from the owning to the related object, both are stored into the same physical location of the database. For the relational database model embedding solves the problem of a mismatch between the object-oriented model and the database model. Since the App Engine Datastore is a NoSQL database without the concept of a relational database table, embedding objects into it's strong parent performs a denormalization. Denormalizing the relational data model helps to keep the performance of the Datastore high because queries execute faster and expensive queries to join entities are not unusual.

```
@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Key key;

   private String name;

   @Embedded
   private Address mainAddress;
}

@Embeddable
public class Address {
    private String street;
    private String zipCode;
    private String city;
    private String country;
}
```

Collection Mappings

Collections are data structures to store objects in a single container object. They are not just an improved version of primitive arrays. In fact collections provide a rich set of APIs to manipulate the underlying data structure and store objects in an efficient way. Collections are a fundamental part of the object-oriented model and so they need a representation in the relational data model. The JPA has mechanisms to connect this two models. The *ElementCollection* annotation allows a collection field to be persisted.

The GAE/J has one limitation for storing lists of embeddable objects. Instead of storing them with their real class fields, it supports only storing them as string values. Also embeddable classes have to be optimized by the DataNucleus enhancer, otherwise a *PersistenceException* is thrown. It's possible to persist *List<String>*, *Set<String>*, and *Map<String, String>* collections into the datastore.

```
@Entity
public class Person {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key key;

    private String name;

    // Workaround to persist a list of Address objects
    @ElementCollection
    private List<String> addresses;

}

@Embeddable
public class Address {
     private String street;
     private String zipCode;
     private String city;
     private String country;
}
```

Entity Group Management

SQL-based relational databases are arranged around unique primary keys for each entity. The primary key uniquely identifies a row inside a table and is indexed to allow fast queries and range scans. Entities in related tables use foreign key fields to store the primary key of the referenced row. Creating a relationship between two entities with a foreign key has no effect on transactions and on the general scalability of an application. This is different in the App Engine datastore. Defining a primary key with JPA annotations on a Datastore key field has far reaching consequences on related entities: It can enforce the semantics of an entity group relationship.

An entity group relationship cannot be dissolved or changed after it has been established. Each entity belongs to exactly one entity group. To transfer an entity from one entity group to another, it has to be deleted in the old group and recreated in the new one. Strong consistency when reading is only possible for entities in the same entity group. With cross-group transactions (XG transactions) ACID transactionality is ensured for up to five participating entity groups in one transaction. Another aspect is that smaller entity groups have a better write performance. Constructing entity group relationships is an important task in the data model design, whereas in relational database systems foreign keys do not entail such consequences. If a single transaction involves too many entity groups, an *IllegalArgumentException* will be thrown.

Owned Circular Relationships

If an entity class has an owned relationship, it will participate in an entity group relationship. Building a circular owned relationship between two entities is not possible. The GAE/J plugin tries to modify an entity's key after it has been persisted and will throw an exception: "Detected

attempt to establish B(no-id-yet) as the parent of A(5629499534213120) but the entity identified by A(5629499534213120) has already been persisted without a parent. A parent cannot be established or changed once an object has been persisted." A workaround is to store Datastore keys instead of entities and thus create unowned relationships.

```
@Entity
class A {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key key;

    // Does not work -> Exception
    @OneToMany(cascade = CascadeType.ALL)
    private Set<B> collection = new HashSet<>();
}

@Entity
class B {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Key key;

    @Unowned // Workaround. Without -> Exception
    @OneToMany(cascade = CascadeType.ALL)
    private Set<A> collection = new HashSet<>();
}
```

Primitive Types as Primary Key

Using primitive data types like *long* as primary key is not recommended for the App Engine datastore. Primitive datatypes are not able to model the complex key of a Datastore entity. *String* fields as primary key have similar shortcomings. The *@GeneratedValue* annotation can be used for primitive *long* values and *Long* objects, but not for *String*: "Cannot have a null primary key field if the field is unencoded and of type String. Please provide a value or, if you want the Datastore to generate an id on your behalf, change the type of the field to Long." If an application creates an owned relationship between two entities, an exception is thrown, indicating that the child object cannot have a primitive type as primary key. Developers using primitive types or strings without a generated value are responsible for uniqueness of the keys. Assigning an already existing value for a key will override the existing entity without any warning or exception.

DataNucleus Extensions

The GAE/J plugin for DataNucleus comes with a custom extension to encode App Engine proprietary Datastore keys into arbitrary strings. This makes it easier to migrate from the

Datastore back to a traditional SQL-based database: If an application is migrated to a RDBMS, the ancestor key can be treated as foreign key to a parent entity. The extension uses the *KeyFactory* class' methods *keyToString(Key key)* and *stringToKey(String encoded)* to translate Datastore key into standard strings. It's important to know that the *KeyFactory* methods are not equivalent to the *Key.toString()* method and the later one should be used for debugging only. Using the string-encoded primary key requires also a string-encoded ancestor key field. Otherwise the mapping is invalid and will throw an exception at runtime. The ancestor key has to be populated by the GAE/J plugin and cannot be manipulated by the user. It is not possible to mix *long* or *Long* with *String* encoded ancestor keys. By the definition of an entity group relationship only one ancestor key can be encoded per entity, so that an entity has exactly one ancestor.

```
@Entity
class Child {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Extension(
        vendorName = "datanucleus",
        key = "gae.encoded-pk",
        value = "true"
    )
    private String thisKey;

    @Extension(
        vendorName = "datanucleus",
        key = "gae.parent-pk",
        value = "true"
    )
    private String ancestorKey;

    ...
}

@Entity
class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Extension(
        vendorName = "datanucleus",
        key = "gae.encoded-pk",
        value = "true"
    )
    private String thisKey;

    // Adding a Child object enforces an owned relationship
```

```
    @OneToMany(cascade = CascadeType.ALL)
    private Set<Child> children = new HashSet<>();

    ...
}
```

Queries

SQL is a common way to execute queries in relational database systems. Since the object-oriented nature of Java and the special requirements of the ORM layer, SQL is not an option as query language for JPA. Instead, JPA provides the Java Persistence Query Language (JPQL) to query for objects in the underlying database. JPQL is focused on the object model and does not provide the full features of SQL. It hides the underlying tables and columns used by the ORM to generate a SQL query. The GAE/J has the following limitations:

- Subqueries not supported by Datastore and throw a *PersistenceException*.
- Joins between two different classes are only supported when all filters are *equals* filters. Filters like > or < don't work and throw a *PersistenceException*.
- The documentation states that "Aggregation queries not supported", but they work in the local development server and in the deployed cloud application. Instead of performing the aggregations in the Datastore engine, the running application instance will compute these functions in-memory after the full result set is returned.

```
// working aggregation queries
SELECT SUM(c.employeeCount) FROM Company c
SELECT MIN(c.employeeCount) FROM Company c
SELECT MAX(c.employeeCount) FROM Company c
SELECT AVG(c.employeeCount) FROM Company c
SELECT c.name, COUNT(c.name) FROM Company c GROUP BY c.name

// working and correct
SELECT c.name, SUM(c.employeeCount)
   FROM Company c
      GROUP BY c.name HAVING COUNT(c) = 2
```

Although the previous queries worked as expected, the following query throws an *PersistenceException* saying *SIZE* is an unsupported method in the *WHERE* clause. Using *SIZE* in the select is possible and works.

```
// Throws an exception
SELECT p FROM Person p WHERE SIZE(p.children) > 10

// Works as expected
SELECT p, SIZE(p.children) FROM Person p
```

Named Queries

Named queries remove the actual query definition from the Java code into an annotation on the entity level. They are predefined and unchangeable during runtime, which makes JPQL query strings better maintainable. Because named queries are precompiled, query parameters have to be set like in prepared SQL statements. This prevents injection attacks and improves the execution performance. Providing different parameters does not change the underlying JPQL named query. Entity managers are responsible for storing, optimizing and executing named queries. The GAE/J plugin does support named queries.

Native Queries

The App Engine Datastore does not support SQL or any other native query language. Creating a native query with *createNativeQuery()* results in an *IllegalArgumentException*.

Inheritance

Inheritance represents an "is-a" relationship between two classes. A class derives fields and methods from another class. This makes the derived class behave like the base class. It provides specialized functionalities which are not available in the parent class. Inheritance is a core concept of object-oriented languages like Java and lets express a complex relationship between objects. This is not the case for relational databases, which store data in two-dimensional tables. A relationship between two records is expressed inside columns of the participating tables. This makes a relationship in a relational database static. On the other side, inheritance is a dynamic concept and objects are created and modified at runtime.

JPA implements inheritance for relational databases and provides a special *@Inheritance* annotation. This annotation lets define the inheritance strategy which the ORM layer should use. There are three different strategies available:

- *InheritanceType.JOINED* uses a separate table for every class in the inheritance hierarchy. To create a new instance, the ORM has to join all involved class tables into one single object representation. This approach is the most normalized one and avoids redundant data. The Datastore does not support joins, so this strategy is not available.
- *InheritanceType.SINGLE_TABLE* stores the complete inheritance hierarchy into one table. This is a denormalized approach and does not requires joins. This way would not conflict with the Datastore entity model. According to the documentation, this strategy has not been implemented by the GAE/J plugin, but it works using version 2.1.2.
- *InheritanceType.TABLE_PER_CLASS* assigns every class it's own table, so each table includes all fields of the full inheritance hierarchy. This type is implemented and supported by the GAE/J plugin.

@MappedSuperclass is a special annotation for entities. If the base class is abstract and only derived classes are stored into the database, this strategy is an efficient way and is supported by the GAE/J plugin. Using an unsupported inheritance strategy will throw an *PersistenceException*.

4.7 Objectify

Objectify is an abstraction of the low-level App Engine datastore API. It's specially designed for the NoSQL-nature of the datastore and does not aim to support any RDBMS-like features. Instead of the pseudo-relational approach of the DataNucleus GAE/J plugin, it's focused on persisting and retrieving denormalized data. Like JPA, Objectify provides annotations to define entities and relationships. Developers familiar with JPA or JDO will notice the similarities between JPA and Objectify annotations. Though, Objectify directly operates on the datastore low-level APIs instead of using an intermediate persistence layer. People familiar with the App Engine datastore should find it's core concepts adopted in the Objectify API. Instead of porting primary-foreign-key relationships into entity groups, it directly operates on datastore keys and takes care of the parent-child-relationships between entities. Objectify does not aim to be database or datastore agnostic, it is a specialized API for the App Engine datastore. It also enforces a stricter data model, since Java fields are typed. This is not true for datastore entities created with the low-level API. Every property can hold different types and it's not required to have the same properties for all entities of a type. So using a framework like Objectify helps to keep a stricter data model on top of the very flexible datastore.

Instead of agreeing on a minimum set of features, Objectify's goal is to implement the whole datastore API. "Objectify provides a level of abstraction that is high enough to be convenient, but low enough not to obscure the key/value nature of the datastore. It is intended to be a Goldilocks API - not too low level, not too high level, just right."[21]

Simple Mappings

One of the shortcomings of the low-level datastore API is the non-extensible *Entity* class. Creating instances of *Entity* and setting properties creates verbose code and involves no strict type checking. This is one of the reasons to use a third-party library to abstract this tasks, and to keep code compact and readable. Objectify mediates between datastore objects and Java objects. This makes it easier to store and retrieve data in App Engine applications. Since the framework is datastore-only and only takes the available datastore data types into account, it has no support to store the following primitive types or classes:

- *java.lang.Character*, primitive *char*, and *char[]*
- *java.math.BigInteger*
- *java.math.BigDecimal*[22]

Entities in Objectify are simple POJOs with annotations. They might look like arbitrary JPA or JDO annotations, but only share a common naming and are in their own package. Fields in such a POJO are mapped to a column in the datastore table by their field name. There is no way to redefine the field's name mapping like in the JPA. Changing a field name will change the column

[21] `https://code.google.com/p/objectify-appengine/wiki/Introduction`

[22] An optional translator for this class is available, but it only transforms *BigDecimal* into a standard *Long*. This is inappropriate for large values.

name in the datastore table. Objectify also ignores getter and setter methods, static or final fields, or fields marked with the *@Ignore* annotation. Each Objectify entity has exactly one *Long*, *long*, or *String* field annotated with *@Id*. To use the datastore service id allocator, the id field has to be of the type *Long*. Otherwise the developer has to manually assign values to the id field. Every entity needs an valid assigned id before it can be used as parent of another entity. Objectify also takes care of the datastore's limits. If a string gets longer than 500 characters, it automatically converts the datastore field into an unindexed *Text*. Fields of the type *byte[]* are mapped to a *Blob* datastore column.[23] Dates can be stored in simple *java.util.Date* objects, *java.util.Calendar* is not supported. If a more sophisticated library for date and time handling is needed, Objectify provides translators for Joda-Time. Such translators have to be enabled before the POJO classes are registered, otherwise an *IllegalArgumentException* is thrown. Objectify also supports the App Engine specific Java data classes like *Rating* or *GeoPt* for geographical points.

Datastore Field Indexing

Datastore indexes are created or updated only during a save operation on an entity. Per default all fields of an Objectify entity are unindexed. This improves the write performance and keeps indexes small. Developers have to annotate which fields they will use in queries, or if the are part of a multi-property index. This can be done for a whole class or a single field by adding an *@Index* annotation. If the *@Index* annotation is added after entities have been persisted to the datastore, the unindexed entities have to be touched with another save operation to update the corresponding indexes. It is also possible to define conditional indexing. A property will be only added to the index if it fulfills a certain condition. Developers can provide custom conditions to create fine-grained conditions. If a query involves multiple properties, a multi-value index has to be defined in the *datastore-indexes.xml*. There is no way to define multi-value indexes with Objectify annotations.

```
public class SomeClass {
  // This field is unindexed
  private int balance;

  @Index
  private String street;

  // Conditional indexing:
  // only if not null and not an empty string
  @Index({IfNotNull.class, IfNotEmpty.class})
  private String district;
}
```

[23] https://code.google.com/p/objectify-appengine/wiki/Entities

Building Entity Group

Entity groups are a central concept of the App Engine datastore. They define the scope of transactions and directly refer to the underlying Bigtable data model. Instead of using a non-transparent ancestor linking mechanism with the distinction between owned and unowned relationships, Objectify has a special *@Parent* annotation. This marks a parent field and will establish an entity group relationship. Developers have to remember that changing an entity's parent and saving it will duplicate the entity. Any changes to the key create a new entity. This is different to the behavior of any relational database, where changing a primary key field will not duplicate the row. So datastore keys are de facto immutable. If the *@Parent* field is a *Key* and points to a non-existing datastore entity, it has no consequences. The ancestor path of an entity is only important for its entity group membership, but not for its existence. Deleting a parent has no cascading effects on any child or it's predecessors.

```
@Entity
class Comment {
  @Id
  Long id;

  @Parent
  Ref<Story> parent;
}
```

Queries

Objectify does not provide a query language like JPA does with JPQL. Instead it offers an API to create queries based on the low-level datastore API. Datastore queries are always executed inside an index scan, which also defines the functional boundaries of them. Since the default behavior of Objectify is to disable indexing on all entity properties, developers have to add the *@Index* annotation to all properties the want to use in a query. Every query starts with a *Loader* returned by the Objectify instance with the *load()* method. The *Loader* is the top-level element of the query chain and needs a *type()* restriction to define the query's entity type. Otherwise it has to be a key query. Since each key is also of an entity type, a query is always strongly typed.

Queries are chainable. Typical parts of a query chain are filters. A filter has a condition string, which consists of a property name and an operator, and a condition value. Filters cannot query parent properties, since the parent is part of the entity's key and not a normal property. To filter by parent, the *ancestor()* method has to be used. Using *ancestor()* will only return entities with the given ancestor in their full ancestor path. The result of a query can be ordered ascending or descending with the *order()* method. To limit the number of results, a query chain can be extended with the *limit()* method. It's important to know that the *offset()* method will skip results after querying them and each skipped entity costs one *small datastore calls*[24]. Faster skips are possible with datastore cursors, which are the recommended strategy to implement paging over a large result set.

[24] Google charged $0.01 per 100.000 small calls until March 2014, they are free now.

```
Query<Author> query = ofy().load().type(Author.class)
    // Authors with a name staring with B
    .filter("name >=", "B").filter("name <", "C")
    // wrote exactly 20 books
    .filter("bookCount", 20);

for (Author a : query) {
   // do something
}
```

Collections

Different to traditional ORMs Objectify doesn't know annotations to define mappings, because the datastore is not aware of special relationships between two entities except the entity group. Instead it stores collections and arrays as embedded datastore entities. Embedded entities can be instances of arbitrary Java classes and don't need a datastore key. The live inside the containing entity and are not indexed. A set collection is stored as a list of embedded entities. To create something like a one-to-many mapping, the collection has to store references to the owned entities.

```
@Entity
public class Team {
  @Id
  private Long id;

  private ArrayList<Ref<Player>> players =
     new ArrayList<Ref<Player>>();
}
```

Caching

Every Objectify instance created by *ObjectifyService.ofy()* keeps track of entities under its control and caches those entities in-memory. This "session cache"[25] prevents expensive datastore calls for simple operations, like loading a single entity by key multiple times, by returning a thread-local cached result instead. This implementation make the session cache not thread-safe, so Objectify instances should never be shared through threads. Objectify also provides a global cache based on the App Engine *memcache* service, which is shared over all instances of an application. Every entity annotated with *@Cache* will use the global cache for get-by-key retrieval, but not for queries. This improves the general read performance and reduces costs, since the number of direct datastore is reduced to a minimum. Objectify stores only field values in the global cache, not serialized objects. The memcache expiration time for cached entities can be defined directly in the annotation. Per default there is no expiration time set. It's possible to deactivate the global cache per call with *ofy().cache(false).load(...)*.

[25] https://code.google.com/p/objectify-appengine/wiki/Caching

Transactions

Normal batch operations are executed as a series of single little transactions and not as one transaction per batch. If no explicit transaction is started, every *save()* will form it's own single transaction.

Objectify requires a servlet filter wrapped around each request. The *ObjectifyFilter* takes care of dirty transaction contexts and blocks until all asynchronous operations are completed. It's main responsibility is to update the memcache-based global cache, since the low-level datastore APIs will take care of pending asynchronous datastore operations.

The transaction API has been changed between Objectify 3 and 4. Instead of using a JDO/JPA-like approach with explicit transaction handling, the current API is influenced by EJB[26] transactions. It's recommended to avoid queries inside transactions and to only perform updates inside of it. The reason is the different query behavior between the low-level datastore API and the Objectify approach. Reading entities from the datastore inside a transaction will not reflect any changes performed inside the transaction on them. Instead the unchanged values will be returned until the transaction is committed. Though, changed property values of an entity are visible inside an Objectify transaction. The reason behind this is the internal session cache. Objectify starts with a new cleared session cache at the beginning of a transaction, so that every read will read a fresh entity from the datastore. As the transaction continues, entities will be fetched from the cache for *get* operations. After the commit the transaction's session cache is merged into the previous session cache to apply changes to the *old* cache too. Newly created entities, which haven't been committed to the datastore yet, are never visible to any queries. Updated entities retrieved by a query will be fetched from the session cache, so their updates are immediately visible.

Objectify encapsulates each transaction into a special unit of work, which is executed by the *ofy().transact()* method. Each unit of work is idempotent, so the result of the execution is independent of the number of times it's executed. This means if a transaction fails with a *ConcurrentModificationException*, Objectify will try to repeat the unit of work as often as it takes to successfully finish it. The API provides a method to limit the number of retries, but per default it's unbounded[27]. Every transaction is per default a cross-group transaction and can handle up to five entity groups. Calling another transaction context from an already existing context will inherit the existing context. To suspend an ongoing transaction and to start a new one, Objectify provides a *transactNew()* method.

[26] Enterprise JavaBeans

[27] The upper limit is only theoretically bounded by *Integer.MAX_VALUE*.

CHAPTER 5

Performance Case Study

The developer documentation for App Engine and specially the Datastore are comprehensive. Both cover a wide range of topics and provide a deep insight into the structure and mechanisms behind these cloud services. Nevertheless they only provide information filtered by the operator and do not take a neutral stance. This motivates to perform experimental research to confirm or disprove such promises. Cloud services try to solve complex computing problems like high scalability and elasticity, which need a broader approach and more intensive observations. For a sufficient insight it's important to avoid small toy examples in the field of cloud computing. The goal is to gain empirical evidence for different non-functional qualities of App Engine and the Datastore. This can lead to new hypotheses as a starting point for further research around these cloud services.

5.1 The Case Study as Empirical Research Method

Studying a managed proprietary cloud service leads to context-depended practical knowledge. The case study methodology provides tools and assistance to find such knowledge. It helps to find truth or falsity for proposition and presumptions inside a certain context. The parameters and the concrete path of the study are defined during the study process, since case studies are flexible design studies.[1] Since each cloud service forms a very specific context for applications or other hosted services, following the case study methodology looks natural and promising.

Proprietary environments cannot be deconstructed or analyzed at source code level. Their official documentation, tutorials, and best practice documents are not sufficient for a deep analysis. Instead of taking a cloud system into an isolated context, it seems more promising to study it in a real world situation with realistic scenarios. This can form the basis for further profound decisions. Seeing a cloud service as a whole keeps them as general and abstract as their principal idea promises. The customer or user of a cloud service takes a top-down approach, which keeps the physical infrastructure abstract and intangible. A case study can reflect this approach and has

[1] [49] describes the flexibility as key difference to traditional surveys and experiments.

enough flexibility to incorporate such restrictions. A big drawback is the reduced level of control. Results might not be reproducible in the same way they would be in an isolated environment. This is also a result of the lack of knowledge over the variables and conditions inside a cloud service, or to be more specific, the lack of insight in the concrete data center design and management.

> "The three definitions agree on that case study is an empirical method aimed at investigating contemporary phenomena in their context. [...] The case study methodology is well suited for many kinds of software engineering research, as the objects of study are contemporary phenomena, which are hard to study in isolation. Case studies do not generate the same results on e.g. causal relationships as controlled experiments do, but they provide deeper understanding of the phenomena under study.[49]"

For a case study it's important to collect the data in a valid and incorruptible way. Observed facts have to be recorded in a consistent manner. Otherwise the results could be questionable and easy to disprove. The collected data couples the propositions from the beginning with the conclusion of a case study. This linking of results to the propositions has to be strong to keep the *Chain of Evidence*[50] intact. It's important to identify patterns in the resulting data and to isolate special edge cases. If not, rare edge cases will question the whole conclusion of the study and make it vulnerable for critics. If running a benchmark is part of the case study, it has to show degree of correlation between the input and output variables, if they are not independent.

The lack of mathematical and statistical techniques makes it hard to use quantitative methods in real-world cloud environments. A test can measure response times or throughput, but still this needs interpretation. What was the state of the cloud environment during the test? Where all parts of the cloud system available, or reported the provider some functional issues like service outages or reduced quality of service? Cloud services don't have a clean and ready state since their complexity is much higher than the complexity of a single computer system. Their state is always in a flow. At the best we can distinguish between an *all services normal*, *some services affected* and a total outage stage.[2] And even if all services are reported as functioning normal, it's sill open what the term *all services* covers, and if *normal* is a 100 percent success rate, or is there an acceptable tolerated error rate. During a case study of a cloud computing environment this has to be kept in mind. Strictly applying "hard research strategies"[51] for cloud-based systems might lead to the wrong conclusions. Transferring results of a closed-world experiment into the real-world environment, where we lack of control and a system-wide overview, is at least questionable.

Other Empirical Research Methods

Methods like the controlled experiment are not suitable, since it's hard to apply quantitative metrics for non-functional characteristics of given software. Also researchers cannot reconstruct a whole data center inside a clean lab environment. Modern data centers operate at a scale not feasible for a single experiment. Energy management, high-throughput networking, monitoring

[2] Google reports incidents and status updates at `https://status.cloud.google.com/`

and real-time reporting, failover strategies and interconnected cluster designs are just some aspects which have to be considered during the design process of a lab environment. Though, even if all components would be available, the experiment still has to show that it replicates the real-world environment with strong confidence. Researchers instead can choose simulations and abstract models to gain insights into the operations of a large-scale data center. This also reduces the complexity, abstracts the context, involves no hardware investments and is less time consuming.

Surveys would be another research method to gain insights about a certain cloud service. They are good in investigating an issue affecting a high number of individuals or users. Though, it's hard to find a representative cross section over the total customer base of a cloud service. A survey also relies on the given answers to a limited set of predefined questions. Researchers have to avoid subjective questioning and need a statistical sound evaluation of the given answers. Nevertheless, a survey could be a good approach for the cloud service provider to gain a deeper insight into its customer base.

Analyzing existing artifacts of a software project is not a valid method for the evaluation of proprietary cloud services. The providers don't provide internal documents at a fine-grained level. It's hard to understand the internal processes and decisions made during the development of a service from the outside. Also such an analysis is very specific to the analyzed project and would not allow to generalize the results.

Summarizing the research methods discussed before, it's clear that a case study is a very profound method to gain new insights in a closed cloud environment like Google App Engine and the Datastore. A qualitative approach requires an isolated context, which we cannot establish for managed cloud services. Though, self-hosted cloud software like OpenStack or AppScale would allow such isolation inside a lab environment. For these cloud software the discussed methodologies are valid options to gain a deeper understanding.

5.2 Starting Point and Initial Considerations

This case study analyzes Java-based applications using Google Cloud Endpoints as frontend technology running on Google App Engine in combination with the Google Cloud Datastore as data storage backend. The focus is on the performance of the Datastore during various load test scenarios. It's designed as *holistic, single-case study* and only looks at one specific service, namely App Engine. There is no comparison with other cloud services under the same context. An exploratory design approach enables changes during the whole process, since it's open what problems and challenges will occur during the load testing stage. We already looked at two different persistence frameworks for the Datastore: JPA in section 4.6 and Objectify in section 4.7. All concrete tests are implemented based on the knowledge gained in the previous sections.

The study describes how different load scenarios are handled by App Engine, how the computing instances react, and what are the effects on the read and write performance to the Datastore. It should show why App Engine starts or stops instances and what might be the factors influencing this load handling algorithm. If possible, a model should be developed which describes the used performance metrics in relation to the incoming load scenario.

On the App Engine homepage[3] Google promises: "Let Google worry about database administration, server configuration, sharding and load balancing." This thought is a starting point for an initial hypotheses: Every application running on top of App Engine automatically scales, regardless the incoming traffic. There is no special need to route traffic or to perform application-wide load balancing. This brings a principal data model design consideration into place. Entity groups are the basic partitioning strategy of the Datastore. Inside an entity group additional capabilities are provided, like strong consistency during transactions. Though, writes to a single entity groups should be designed to not exceed 1 write operation per second. This brings up another hypotheses: Writes to a single entity group are limited to 1 write per second. These two hypotheses form the foundation for the following case study. On one side there is an apparently continuous scalability, on the other is a very specific limitation for the throughput. Based on the emerged hypotheses the following research questions are asked:

- **RQ1: Is JMeter an adequate tool to create meaningful load tests for cloud environments?**
- **RQ2: How does App Engine react on incoming traffic and what patterns can be detected during a load test? What are the effects of different write strategies for the Datastore?**
- **RQ3: What are the effects of the throughput limit of 1 write per second per entity group? According to the documentation this is only a soft limit and should be the design goal for an application's data model.**

The case study follows a first degree method by observing App Engine reacting on different HTTP-based load tests. The data is collected by remote JMeter instances, which are controlled by different scripts. This makes it easy to scale the test and to broaden the tests for more sound results. Creating the tests, applying them on all remote instances and collecting the results is a time consuming task. Though, it provides a deeper understanding of the behavior of the real world application and the results can be verified easier than by using existing data and studies.

Choosing a Persistence Layer

In practice there are two persistence layers available: JPA, which has been discussed in section 4.6, and Objectify, discussed in section 4.7. It's clear that the JPA implementation on top of DataNucleus, with its roots in the relational data model, has not been designed for the Datastore. Instead it tries to bring ORM-concepts into the schemaless NoSQL world. This suggests the assumption that it adds some performance overhead to various operations. On the other side, JPA also has advanced capabilities like interception of updates. As previously discussed, the data modelling has to follow traditional concepts and there is no full support for building entity groups. Though, it has to be shown that JPA really adds a performance overhead and that Objectify is the better choice to build the test cases on. A short benchmark running for two minutes has

[3] `https://cloud.google.com/appengine/`

been implemented for JPA and Objectify. It writes to the Datastore in a similar manner and tries to provide equivalent write procedures. To trigger a write a test web application has been created, which provided a simple REST interface to a HTTP-based client. Each handled request recorded two timestamps: one for the start of the write operation and one for the end. These two timestamps were returned to the client and exported into a simple log file. This avoids any effects by the network connection between the application and the benchmark client. To further ensure equal conditions, the written entities contained only field types which are supported by both frameworks. To avoid any optimizations by the Datastore API itself, all written values were random and generated by a random string and number generator. The collected results and according percentiles are summarized in table 5.1.

	avg	**0.50**	**0.10**	**0.90**	**0.99**
DataNucleus / JPA	76.5	76	53	98	114
Objectify	50.6	50	44	60	69

Table 5.1: Performance comparison between JPA and Objectify for simple writes, results are in milliseconds per request.

Looking at the results we see a certain gap between requests using the JPA interface and the ones using the Objectify backend. JPA is around 50 percent slower than Objectify in the average and median. Even at the 10th percentile Objectify is still significantly faster than JPA. Especially for cloud services with a large user base the 90th and 99th percentiles are interesting. Here we see a clear performance win for Objectify, which expresses in a slower increase and a significantly lower processing time. Keeping the overall performance per request in mind, this is a clear indicator to prefer Objectify over JPA if there is a choice. Though, in real-world projects other factors influence the selection of the underlying persistence framework. One of the most important cost drivers for applications running on App Engine are writes to the Datastore and this should also be considered in the selection of the right persistence framework. This particular micro-benchmark only is important for the following test cases during the data collection and should not be generalized.

A third option would be to use the plain Datastore Java SDK API. This API does not abstract any calls to the Datastore and manipulates raw entity objects, which then will be stored during a write. It's not a very convenient interface and is more verbose than Objectify. Feature wise Objectify is equivalent to the Java SDK and uses the SDK for all operations between applications and the Datastore backend. Also Objectify ensures type safety for all operations and has a modern transaction model. So the following study relies on Objectify as persistence layer.

5.3 Data Collection

To make the data collection process as open and transparent as possible, the following sections describe the used tools and followed procedures. It shows the load test infrastructure and how the overall test execution environment looks like.

JMeter Load Tests

JMeter is a Java-based open-source testing tool. It runs automated functional performance tests against a large number of targets. The general design of JMeter is modular to enable extensions. There exists a large number of plugins to enhance the core functionalities. For example, the *Cross-Layer Multi-Cloud Real-Time Application QoS Monitoring and Benchmarking As-a-Service Framework* (CLAMBS) framework[52] used JMeter to implement its benchmarking component using HTTP and JDBC SQL. To run a test the user can choose between a GUI-based executor and a command-line tool, which takes JMeter test files as input and runs the test without any user interaction. One very useful feature is the support for scripting languages like the BeanShell scripting language and JavaScript. This enables the tester to react on the current test environment and every request parameter can be set through a pre-executed script. JMeter runs multiple threads in parallel to generate test traffic. The number of threads can be chosen manually or in automatic mode, where as much threads are created as the test client can handle. The basic elements of each test are:

- **Samplers**
  A sampler executes a request to a target, which can be web applications using HTTP, FTP servers, active JDBC connections, raw TCP requests, and many other targets which can generate test results.
- **Listeners**
  They listen to results of samplers and display them in the GUI mode. If the non-GUI mode is active, most listeners are ignored, especially if they draw results onto a graph.
- **Thread Groups**
  To execute threads in parallel a thread group has to be defined. It collects logic elements, samplers and listeners to execute tests. The number of threads used in a single group can be configured. Each thread in a thread group acts independent from the other threads, so they cannot influence each other directly. It's possible to define a ramp-up period to slow start tests instead of shooting with all threads from the beginning. All elements of a test must be under a thread group.

One aim of this case study is to use automated tests to execute repeatable load test scenarios against an App Engine web application over HTTP. For this, JMeter looks promising and provides the features needed to generate a realistic load scenario. Though, to get more representative results, the load tests have to run at larger scale than possible with a single JMeter client instance. For this the non-GUI mode of JMeter can be used. For a realistic test a larger number of JMeter instances have to generate load in parallel against the App Engine application.

Testing on Virtual Infrastructure

During the development of a load test it is sufficient to use a standard Internet connection to write JMeter tests. In this stage it's only important to find bugs in the tests and to create good traffic scenarios. A developer or software tester can ignore side effects caused by slow connections, and

also high network latency is not a problem. But once the tests should generate real world traffic patterns, it's necessary to also distribute the load generators across different servers and in the best case even utilize multiple data center locations. Though, a hosted load tester running on dedicated or virtual hardware in a data center is not a perfect imitation of a cloud service user. But since a benchmark or load test should have a sound setup and a solid hosting environment, this seems to be a good approach to obtain valid results.

Today exist an impressive number of widely used IaaS providers to choose from to host load generators. For this case study DigitalOcean was the primary choice since its support to run Docker containers from a base image. DigitalOcean provides *droplets* as base virtual machine images to run in its cloud. Each droplet has a special purpose, e.g. there are droplets for running Redis services, complete web applications like GitLab, or simple bare OS droplets just running an operation system like Ubuntu. DigitalOcean claims a 1 GBit connection for each server hosted in its data centers. Even if Datastore load tests are not data-intensive tasks, the included one terabyte of traffic in the basic option gives enough room to run them without worrying about traffic limits. The service provides an API to check the status of each running droplet and also allows to automatically controlling the start and termination of them. This could be useful to further automate load tests, even if this is not required for this case study. Another difference from DigitalOcean to other IaaS services is the very reduced set of features. It offers the hosting of virtual machines and services around the management of them, but has no built-in cluster solution or load balancing. Running load tests doesn't require these features. One of the biggest advantages is the hourly billing of consumed resources. For load tests requiring a big amount of memory the underlying droplet instance could allocate up to 64 gigabyte of memory with 20 CPUs, costing around 1 US dollar per hour[4]. The simple but powerful REST API of DigitalOcean makes it easy to increase the automation of load tests. Instead of starting and stopping droplets manually, it's possible to perform these actions with simple HTTP POST requests. The requests have to provide an active API token and a droplet identifier, which can be easily done in standard *curl*[5] calls. For a better distribution of the request origins different US-based data center regions have been selected, shown in table 5.2. Five droplets were running on the east coast New York data centers, five droplets in the west coast San Francisco data centers.

Table 5.2: Chosen data center regions for the DigitalOcean droplets.

Region	Droplets
NYC2	1
NYC3	4
SFO1	5

Docker Containers as Test Execution Environment

Docker is an open source platform to start and manage containers, which itself are designed to run one specific task or application. Containers are like relatively lightweight virtual machines[53],

[4]This is the highest option on the DigitalOcean droplet list from April 2015.

[5]A command-line URL retrieval client. `http://curl.haxx.se/`

but more agile and faster than hypervisor-based machines. Here, the specific task for a container is to provide a stable JMeter client instance and to run the given JMeter test against the App Engine web application. Docker manages so-called images, which are the foundation for each container. An image contains the operating system, the software stack, and installed software packages in a static read-only format. Each image itself can be derived from another image, which might be also a downloaded image from the web. A Dockerfile defines if an image has such a public base image and which other components should be installed in the current image. Every Docker container itself is just a running instance of an image.

Instead of installing JMeter and the full benchmark environment on each droplet, Docker containers can be used to run the actual JMeter test in an isolated environment. Docker containers are built around the concept of *immutable infrastructure*. This means the software stack and the virtual environment running inside a container don't change after multiple executions. Every time the container starts up again, it will be at the same entrypoint as in all previous and following executions. Only the environment around it can change. This environment provides the JMeter test file, important environment variables for the execution and a startup script for the JMeter headless client. Depending on the test to execute, only the environment has to be modified. The DigitalOcean droplets and the Docker images itself are not touched and can be reused without big effort. This is a huge difference to existing frameworks and approaches, where a full runtime environment is configured and started for each test. Every change in the underlying system can lead to unwanted consequences in the higher levels of the software stack. Inside the Docker container a reusable and stable execution context is guaranteed.

Using Docker containers also avoids drifts between different servers. After creating a droplet with DigitalOcean's Docker/Ubuntu image no changes are made to a new droplet. Each droplet of the load test cluster has the exact same configuration. Every droplet's Docker images are created from the same Dockerfile, which is transferred at the beginning of each test run over a simple secure copy (SCP) command. This makes it easy to manage a fleet of test servers, since each one of them will be configured the same way. Instead of defining a dedicated framework like [8], most complicated parts are already handled by the DigitalOcean API and the Docker container.

So the droplet, which is the lowest level of the testing stack, is only responsible for booting up a standard Linux system and provides the Docker runtime installed. The Docker runtime manages the different virtual images which are needed to run a JMeter test on top of them. And the Docker environment is present at every test run and contains the actual JMeter tests. This separation makes it easy to validate the tests on a local development machine with App Engine's SDK server. On the other side it's easy to deploy the test on multiple machines without complex tools and special cluster management. Instead of using big physical hardware, this approach enables a distributed load test on many small lightweight virtual machines. This is closer to the real world, where traffic doesn't come from a limited number of sources, but rather has many different sources.

5.4 Implementing Tests with Cloud Endpoints

The following tests have been implemented using the Google Cloud Endpoints SDK[6] and always have a corresponding JMeter HTTP-based test request. Cloud Endpoints make it easy to access an API over HTTP and it takes care of the request and response handling. Most of the glue code between the servlet container and the actual implementation is provided by Cloud Endpoints. So we only describe the topic of the test and their basic input parameters and not how their implementation looks in every detail.

Cloud Endpoints is a collection of tools, libraries and services to create a HTTP-based API backend on top of App Engine. These backends can provide their capabilities to standard HTTP clients, whereby Google promoted the service especially for iOS and Android app developers as easy to use cloud data backend. Authentication is possible via OAuth 2.0, but not needed for this case study. An important restriction for applications running behind Cloud Endpoints is the missing support for custom domains, so all requests have to point to the AppSpot domain. HTTP client code could be auto-generated by the Cloud Endpoints SDK, but this is not necessary for the JMeter-based tests in this study.

Each test is implemented as a public method in the *DatastoreBench* class. This class has a static initialization block which calls the Objectify service for the initial configuration. The different tests are annotated with the *@ApiMethod* annotation for finer configuration. By default all non-annotated public non-static methods of this class would be also accessible via the endpoint API, but since the better readability the tests use explicit annotations. Tests which only query the Datastore are implemented with the GET HTTP method, the others are all implemented using POST. To make it clearer which Java test method maps to a certain JMeter test, all API methods also define a explicit path under which they are accessible.

datastore.writeLightweightEntity

This test creates a very lightweight entity with just three unindexed string fields. All string fields are limited to keep their size small. To prevent any optimizations from the Java runtime the method takes a string as input. This string is the base for the two other fields, which also contain a randomly[7] created long number. To get results less influenced by the runtime or network, this method repeats saving an entity five times. Each entity is saved synchronized without an explicit transaction. The entities itself are not part of a larger entity group and have not parent.

datastore.writeHeavyweightEntity

Instead of writing a relatively compact entity, this method creates a heavyweight entity containing five large string fields and also nine indexed fields. At the creation of a new entity a long random string is generated and stored in the large string fields. One goal was to fill up the entity as close as possible to the one megabyte size limit the Datastore has. Since this limit is based on the real entity size with metadata, only an approximation was possible. In real world applications such

[6] `https://cloud.google.com/appengine/docs/java/endpoints/`

[7] Using *java.util.Random* since the better performance compared to random UUIDs, which internally use the cryptographically strong *SecureRandom* number generator.

entities should not be used and other data storage options should be considered. Though, the goal of this test is to find out how the Datastore handles such heavy entities and what are the consequences for the response times.

datastore.writeEntityGroup

The first two tests do not involve any entity group building. Since entity groups define the scope of each transaction and are important for strongly consistent queries, it's important to test how the Datastore handles their creation and if there are any differences between normal non-grouped entities and entities which are part of an entity group. For this the test has a special entity called *GroupedEntity*. Each grouped entity can have another grouped entity as parent. This makes it possible to create an entity group in a very easy name. To identify every entity inside its group, each has a simple field which holds their nesting level. Another string property also shows what their position inside the whole entity group. The benchmark itself creates the root entity and adds three levels of children. This makes a total number of 40 entities inside each entity group. The whole group is constructed inside a single transaction.

datastore.writeEntityGroupWithoutTransaction

This test is based on the previous *datastore.writeEntityGroup* test. The key difference is the missing transaction, so each entity of the group is created immediately. A goal is to show if this approach is faster or slower than the other one.

datastore.writeRetrievableEntity

This test method creates an entity which contains a string and a number. Both fields can be used to query for specific entities. The first approach is to fill the string with a four characters long random sequence. The same sequence generator which is used to generate the query string in the following tests. It's not guaranteed that each possible string combination exists, since there are over half a million and also the generation of the entities is random. The second approach is to create a random number between 1000 and 9999 and to store it into the entity. Since the Yahoo YCSB can generate random integer sequences, we can incorporate its generators in the follow-up retrieval test.

datastore.readRetrievableEntity

The test is based on the *RetrievableEntity*, which contains a string and a number. In this case the integer number stored in the entities is used to lookup the first matching entity with the given number. This should involve a normal index table scan for the first matching entity. It's important to distinguish between key-based lookups, which are free operations in the Datastore and should be even faster, and index scans, which involve two steps: looking for an entry in the index and then retrieving the corresponding entity from the entity Megastore table.

datastore.readRetrievableEntityList

Instead of querying for a single *RetrievableEntity* based on its number, this test looks for a list of entities sharing the same random string sequence. The result is returned as a *CollectionResponse* list, which is a special Java class to return a collection as result via a Cloud Endpoint.

datastore.writeNamedKeyEntity

The previous tests had no guarantees that the actual key space has been filled up with entities. This allowed also empty results from the endpoint. For a more specific test another entity, namely the *NamedKeyEntity*, has been defined. This entity doesn't use a random long number as ID part for the key, instead a string property is used as unique identifier. This makes it easier to limit the possible IDs for a benchmark and enforces a deterministic benchmark strategy, where it's guaranteed to retrieve an entity. To enable also index-based lookup for entities, every *NamedKeyEntity* mirrors the key's identifier in a property called *indexedIdentifier*. This property can be used to lookup an entity with an index query. This makes key-based lookups and index-based lookups directly comparable. This test method only creates one single entity. The JMeter test is responsible for filling up the key space and should only be executed once.

datastore.readNamedKeyEntityByKey

Based on the filled key space from the previous test, this test performs a lookup by key. This should be the fastest possible lookup in the Datastore, since it only involves a single table scan for the actual entity by key. This simple read operations are free of charge and therefore a cost-effective data model should include key-based lookups instead of using normal reads from indexes. That's why this test is important to compare key-based lookups to index-based lookups.

datastore.readNamedKeyEntityByIndex

This test enables a direct comparison between key-based and index-based entity lookups. It's based on the *NamedKeyEntity*'s indexed identifier property, which holds a copy of the key's identifier, but in a string field. Instead of querying for a given key, the test creates an equality query for the indexed string field.

datastore.createTreeGroup

The *writeEntityGroup* tests focused on the creation of new entities inside the same entity group. Though, this test creates entity groups to study the update behavior inside an entity group. It focuses on the one update per second per entity group limit[8] which Google recommends as maximum update rate for the high replication Datastore. For this a tree with four levels is created. Each entity in the tree shares the same root entity, so it's in the same entity group as all other entities with the same root node. During the creation it's important to set correct keys for each entity of the tree. Each parent entity's key has to contain the full path up to the root. Also each

[8] `https://cloud.google.com/appengine/docs/java/datastore/`

parent entity has to be persisted before it can be used inside the ancestor path of a key. Otherwise it would not be known as a valid entity.

datastore.updateTreeGroup

This test is build upon the *createTreeGroup* test, which creates a large entity group. After the group is created, this test selects a random entity out of the group and changes the text field string of the *TreeEntity*. To see if the write limit is reached and to detect update contention, which occurs if an entity group is updated too rapidly, the number of retries for each update is logged to the application log. So after running the test the potential contention is visible in the logs and can be analyzed further.

5.5 Executing the Load Tests

Each of the described tests runs isolated from the others. For this we terminate all running instances of the project before the actual test run. App Engine will be as close to a fresh environment as possible. Though, it's unknown how the instance scheduler in the App Manager takes care of historic data and specially what are the consequences of running a sequence of different load tests on instance creation and termination. The load test application uses the default module for all tests. It has been configured without any explicit scaling settings, so instances will be of the default F1 instance class and are all auto-scaling. There is no manual scaling configuration in place, since a primary target of the case study is the behavior of the App Master during a load test and how the master handles a high number of requests during a relatively short period of time. Each test consists of the following steps:

1. Ensure all old containers are stopped and deleted from the DigitalOcean droplets.
2. Upload the Dockerfile and the container context with the latest JMeter test to each droplet via SCP.
3. Build the Docker images and run the actual container with the JMeter load test.
4. Download all results from the shared file service where all containers upload the test result at the end of their execution.
5. Stop all droplets.

Using auto scaling F1 instances also brings a very limited amount of memory and only a 600 MHz equivalent CPU limit. The load tests do not require a faster CPU, since they are relatively simple and do not perform any expensive computations. Though, the memory limit is a much more restricting limitation. In order to fit into this boundary, the load tests have to avoid a heavy class loading and keep their framework simple. Since all tests are based on Cloud Endpoints, first experiments showed that the amount of memory is sufficient for this load test. However, this must not be true for more complex real world applications, which require large and memory-expensive frameworks to load. Table 5.3 shows other instance classes which were available at the time.

Table 5.3: Available instance classes for automatic scaling at the time of test.

Name	Memory Limit	CPU Limit	Costs / Hour
F1	128 MB	0.6 GHz	$0.05
F2	256 MB	1.2 GHz	$0.10
F4	512 MB	2.4 GHz	$0.20
F4_1G	1024 MB	2.4 GHz	$0.30

Comparing Lightweight and Heavyweight Entities

The first test writes very lightweight entities into the Datastore. This should be a very common scenario for Web applications, where the actual amount of data is low and where larger binary data is stored on separate services. During the test one of the ten JMeter containers did not respond to the test, so the remaining nine executed it. Every JMeter client sent 10,000 requests to the application, parallelized in ten separate threads. This means a total amount of 90 requests could reach the application at the same time.

At the peak App Engine handled this load with 14 instances, of which each consumed around 100 megabytes of memory. Running at full steam around 220 requests per second were executed. To confirm the numbers, the test also has been repeated with just one droplet. It has been shown that the requests per second scale in a nearly linear manner and that the end of the maximum parallel write has not been reached. Though, it's clearly visible that the high-replication model of the Datastore has a poor write performance compared to other storage backend. Storing a small entity took around 50 ms per entity, where the write performance improved a little bit after the start of the test. Write-intensive applications need to choose a faster storage backend, if they need faster responses or require more throughput.

The heavyweight entity test is built the same way the lightweight entity test is, but only with an entity close to the Datastore's size limit for entities. As expected the write performance was not as good as for the lightweight entity, but there has been an unexpected effect on scaling the instances. At the beginning of the test a lot of outliners in the requests per second and in the response time occurred. It looks like App Engine has to find the right balance to handle these entities and the associated larger payload. The transactions per second dropped dramatically at this outliners, also the response time increased. Where the full lightweight entity test took 7 minutes and 45 seconds, the execution of the heavyweight entity test spanned nearly 44 minutes. The longer runtime and the lower overall throughput is visible in figure 5.2. Also the error rate[9] was 12 times higher than in the lightweight load test and some request came close to request timeout of 60 seconds. The median time per Datastore write operation laid at 180 milliseconds, which is significantly longer than for small entities. Where the lightweight test continued to have excellent response times at the 90th percentile, the heavyweight test shows a much poorer performance and a clear decrease of performance beyond the 95th percentile. Summarizing this results, a clear recommendation to keep entities small and avoid any binary data like images inside them can be issued. The Datastore's design is made for text data and not for storing larger objects. Also from the costs storing data in the Datastore is more expensive than in the Cloud

[9]Errors reported by JMeter, which are not directly connected to real Datastore put operation errors.

Storage option. One hundred gigabyte in the Datastore cost $18.00 per month for raw data storage, whereas the Cloud Storage equivalent is $2.60 per month at the highest durability level.

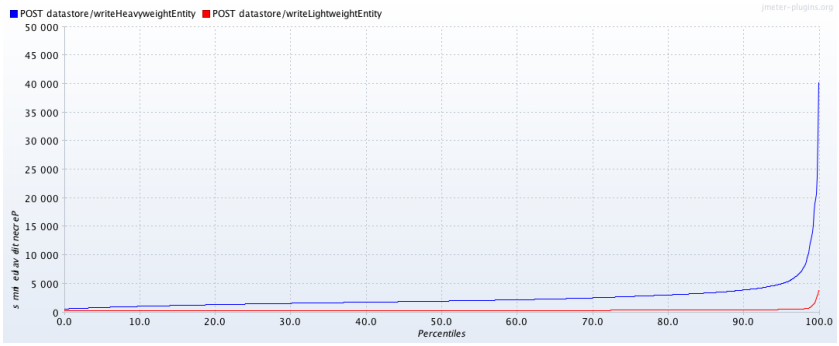

Figure 5.1: Response times percentiles for the lightweight entity test (lower red) and the heavyweight entity test (higher blue).

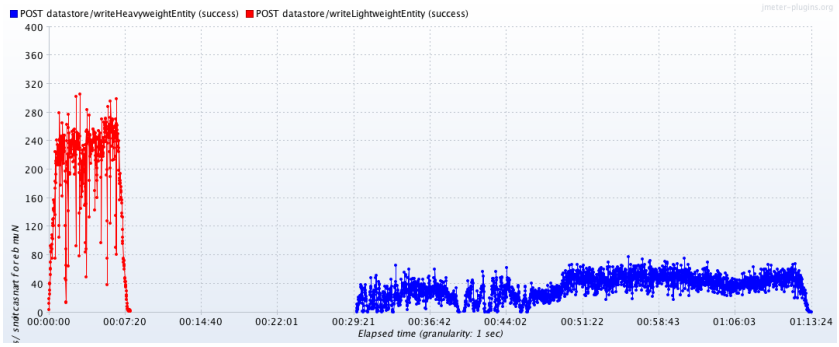

Figure 5.2: Throughput over time. The left side shows the requests per second for lightweight, the right side for heavyweight entities.

Writing Entity Groups

Creating large entity groups should be an expensive task by the fundamental design of the Datastore. Each write requires a RPC call and a majority vote to guarantee the durability of the write and the consistency of the entity. Both tests, *datastore.writeEntityGroup* and *datastore.writeEntityGroupWithTransaction*, create large entity groups and write them to the Datastore. The only difference is that once it happens inside a large transaction and in the other case every write happens without an explicit transaction handling.

The first test was without explicit transaction handling. Since the lack of a fine-grained instance startup and termination reporting, only a rough number of the instance count can be given. At the peak 8 instances were reported in the Cloud Console dashboard, which means around one instance per write per second for this test. This sounds reasonable, since that number of instances would be likely necessary to handle all incoming requests without filling up the

request queue with waiting requests, which will emit writes to the Datastore. Though, after some time the number of instances reduces to 2. It looks like the App Master discovered an over-provisioning situation, since the overall workload per instances is not high and memory is wasted. This could be a consequence of the relatively easy structure of this test, which only writes a high number of entities to the Datastore and the problem is the latency of the writes and not the volume of incoming requests. The test itself is also not CPU-intensive and most of the request contains on waiting for the Datastore writes to be acknowledged. In the second run with a transaction around the creation of the entity group a similar number of active instances has been noticed. Though, due the coarse-grained reporting only a very general conclusion can be given. Using transactions has no observable effect on the number of instances, but provisioning optimizations have been noticed during the execution of the test.

Regarding the overall performance, a clear picture displayed after the load test. Creating large entity groups without a transaction catching all write operation into one single unit is noticeable slower and ineffective. Write operations which belong logically together are much faster inside a common transaction context. It looks like the Datastore writes are bundled together and are only executed at the commit of the transaction. This requires that all auto-generated IDs inside the transaction are valid and will be unique at the commit time. So even if a transaction fails, the IDs have to be allocated from the auto-generator to guarantee a consistent entity group tree in this particular test. The boxplot in figure 5.3 clearly shows that keeping the creating of the entity group inside a transaction was a good guarantee to respond in less then 500 milliseconds. This is an acceptable timeframe for such a large entity group, which won't occur very often in real world applications. On the other side the transaction-less test took around 2,000 milliseconds and never came close to the other test. Two seconds is a perceptible delay in an user-facing application and should be avoided. So App Engine applications should always try to write to the Datastore in a consolidated well-packaged unit of work, which prevents long-running requests and retains the overall responsiveness.

Querying Entities

The *datastore.readRetrievableEntityList* test loads a list of entities and returns it. The query criterion is a simple text-based query searching for a string match. During the benchmark App Engine used 23 instances with an average memory consumption of 98 megabytes at the peak of the load test. The average throughput was at 1,214 requests per second with a total count of 777,892 requests executed. During the test three irruptions happened: at 3.30 minutes, 4.25 minutes, and at 7.30 minutes. All of them are not initiated by the JMeter client, they are results from the App Engine application and are clearly visible in figure 5.4. This is also backed by the response times over time graph, which clearly shows three short increased response times at this time frames. Though, the effect was not dramatic and it could be a result of re-provisioning by the App Master because of the increased load situation. It's not possible to perform a more in-depth analysis since the closed nature of App Engine's provisioning component for auto-scaled instances. Looking at all percentiles we see a very flat curve with a slight increase at the 70th percentile and a significant, but tolerable further increase at the 85th percentile. Though, at the 99th percentile the response time is only exactly 1 second, which is a satisfying performance for a query. Each query also returned the execution time and the number of returned results inside the

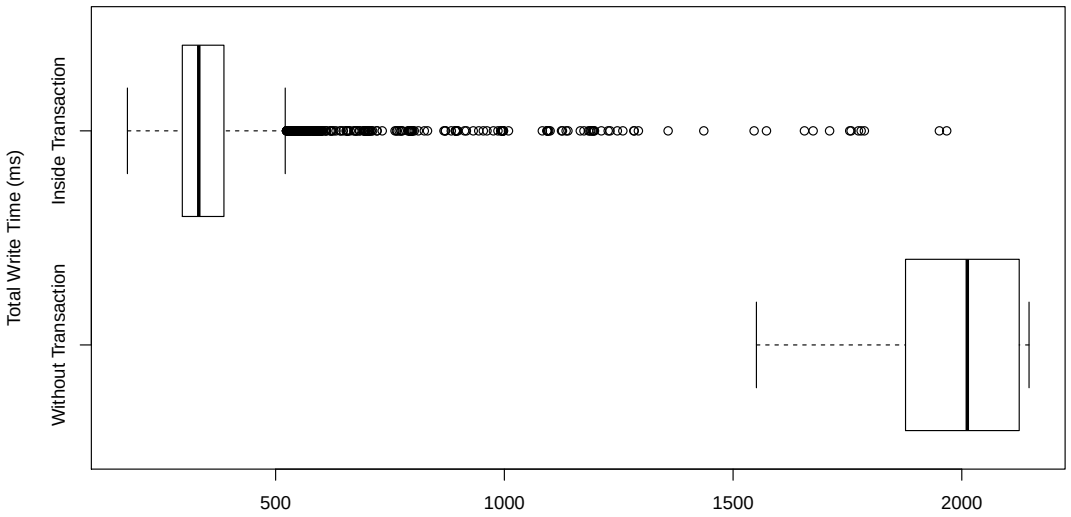

Figure 5.3: Raw Datastore write times for the creation of a large entity group inside a single explicit transaction (top) vs. implicit separated single-write transactions (bottom).

Java endpoint method. Table 5.4 shows a slight increase for each entity which has been returned by the Datastore. This could be the result of the additional entity data fetch which is needed after spotting an entity in the index table during an index scan.

Table 5.4: Result Set Size and Retrieval Time

Results	Retrieval Time
0	12 ms
1	19 ms
2	24 ms
3	25 ms

Randomized queries

Instead of querying the Datastore via the *datastore.readRetrievableEntity* method by an equality criteria on a string property, a variation of the tests takes an integer number as input and queries for the first matching entity with the given number. Using an integer number makes it easy to generate a random sequence of numbers with the YCSB sequence generators. Each generator from the YCSB benchmark[54] uses a different strategy to return a differently distributed range of integers. This allows to simulate different load test scenarios by selecting a different randomly distributed range of numbers. The used strategies are described in [54]:

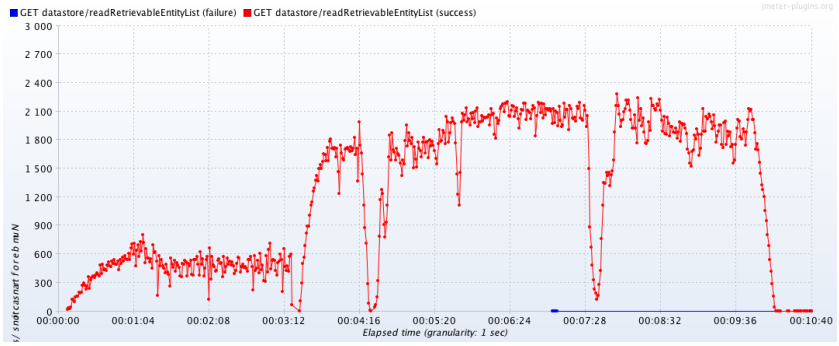

Figure 5.4: Throughput over time for the *datastore.readRetrievableEntityList* test.

- Uniform: Choose an item uniformly at random. For example, when choosing a record, all records in the database are equally likely to be chosen.
- Zipfian: Choose an item according to the Zipfian distribution. For example, when choosing a record, some records will be extremely popular (the head of the distribution) while most records will be unpopular (the tail).
- Latest: Like the Zipfian distribution, except that the most recently inserted records are in the head of the distribution.

In addition to these standard generators this load test will also use the *Hotspot Generator*. This generator returns in the chosen configuration an integer sequence which ensures that one percent of the possible data items will be accessed by 99.9 percent of all incoming requests. This could be a very common scenario in social media applications were a small number of the total entries goes viral and will be accessed by a high number of the total incoming requests.

There has been no significant difference in the load test results between the the various sequence generators used to generate the query criteria. All four generated a similar boxplot and their quartiles are distributed around the same measured response times, visible in figure 5.5.

During the execution of the different tests App Engine started between 35 and 62 instances to handle the incoming traffic. Over the 10 minutes long load test an average of 1,600 requests per second were handled by the application. A very interesting number is the 99th percentile where the response time still was around 300 milliseconds, a very competitive number for index lookups. Even the 99.9th percentile lies at one second per request, which is still a fast response for the huge majority of users. So read-intensive applications can benefit from the principal design of the Datastore and how entities are read. As seen in previous tests, there are some spikes in the response times over time which might indicate an intervention by the App Master to start new instance for a better distribution of the load and to prevent an under-provisioned application. During these spikes the response times kept in an acceptable range around 600 milliseconds and disappeared after 30 seconds.

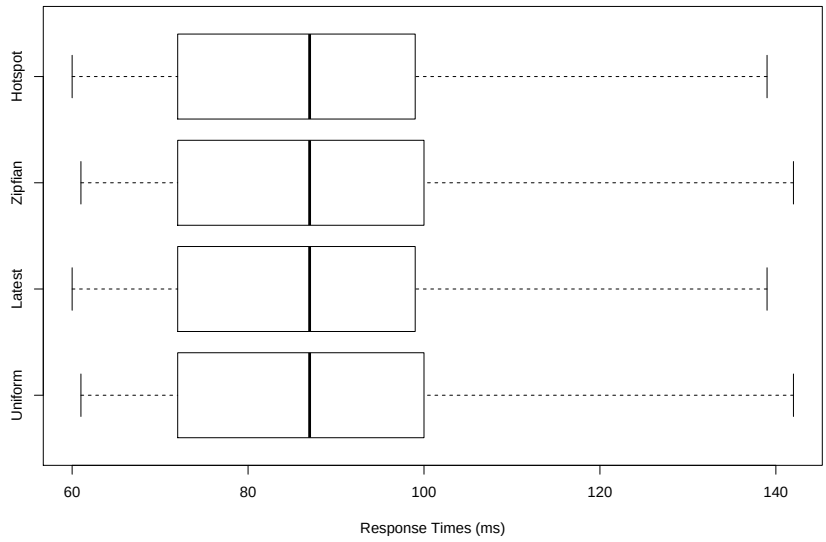

Figure 5.5: Response time results using four different ID generators for simple index queries.

Retrieving Entities by Key

This test involved a total number of 3,389,730 requests and had a maximum throughput of 3,574 requests per second. Even the average throughput was at a high number with 2,566 requests per second on the application. The main goal was to find out if there are any differences between a lookup by key or a lookup over an index scan. The result shows that both approaches produce a similar result at the location of the quartiles and the median in figure 5.6 of both lies at exactly 89 milliseconds per request. The slight difference in the upper quartiles might be a result of a non-optimal provisioning at the lookup by key. A peak number of 162 active instances were displayed in the Cloud Console during this test. So to handle this large amount of traffic on quite small and low-memory instances, App Engine needed to start up a high number of these F1 instances. An interesting behavior was shown after the test. App Engine kept a lot of instances running and the only way to remove the instances from the application was to manually delete them instance by instance.

But even if the performance numbers of both operations are quite the same, there is an important difference: key-only queries are free of charge, but index-based queries cost one read operation. Additionally, every returned entity also costs one more read operation. At the time of the benchmark every 100,000 read operations were billed with $0.06 by Google. So executing the index-based test cost $2.00, whereas the key-based test charged the billing account with roughly $1.00. Over the lifetime of an application this will be a perceptible difference in costs for an application.

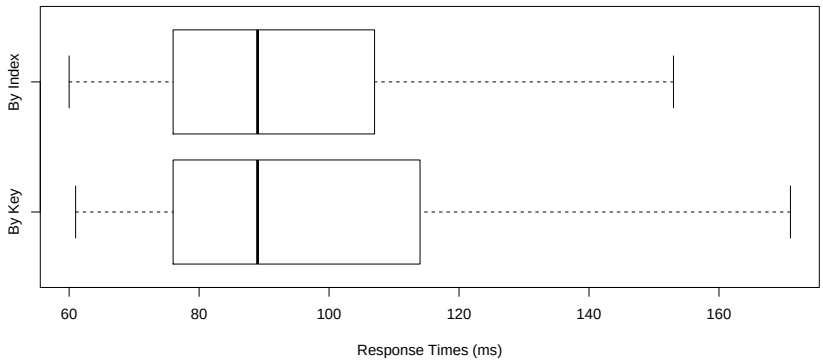

Figure 5.6: Response time results using key-based and index-based queries.

Testing the Entity Group Write Limit

During the execution of this test a quite unexpected number of parallel writes to the same entity group has been reported by JMeter. Due the distributed nature and the complex Paxos algorithm to ensure a durable write and overall consistent state, the Datastore had a soft write limit of one write per entity group per second. This limit was not replicable in the load test, instead a very constant throughput of around 45 requests per second was visible. If a transaction failed due to high Datastore contention this will result in a high number of re-attempts initiated by the Objectify framework. This number of re-attempts has been also reported in the test result, so we can se the overall number in figure 5.7. Instead of heaving a lot of transaction stuck, only a low number needs a then quite high number of re-attempts to complete. But as the response times percentiles in figure 5.8 show, around 70 percent of all requests where processed in a very good time under 300 milliseconds and even the median lies at 129 milliseconds per request.

Since the entity group was large containing over 1,000 entities and updates on them were triggered randomly, this high throughput was not expected. If a modified version of this test was executed on only one single entity, the write rate even reached 90 writes per second. Google emphasized that the write limit is primarily a design goal and that the SLA only provides a guarantee for this number of writes, in real-world applications it seems to be possible to exhaust this limit over at least some time. For this study the benchmark only ran 10 minutes due to the high write costs of the Datastore updates. Therefore, applications violating the SLA's limit over a longer time may be penalized by the Datastore or by a manual intervention by Google. Though, this is only an assertion based on documentation pages[10] and blog articles by Google[11,12]. No reports have been found about the probable consequences in the case of a misbehaving application.

[10] `https://cloud.google.com/appengine/docs/java/datastore/`
[11] `https://cloud.google.com/appengine/articles/scaling/contention`
[12] `https://cloud.google.com/appengine/articles/scalability`

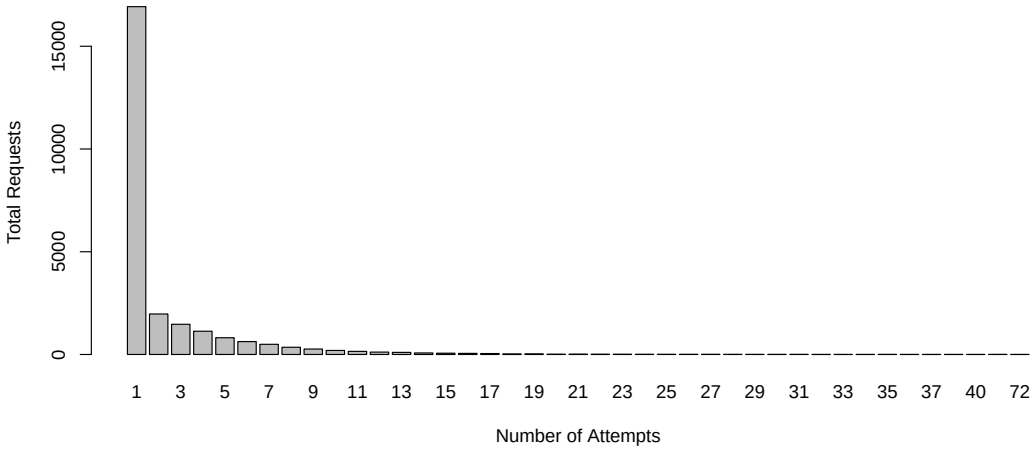

Figure 5.7: Attempts necessary to update an entity of an entity group inside a transaction. Note that empty attempt groups have been skipped due better readability.

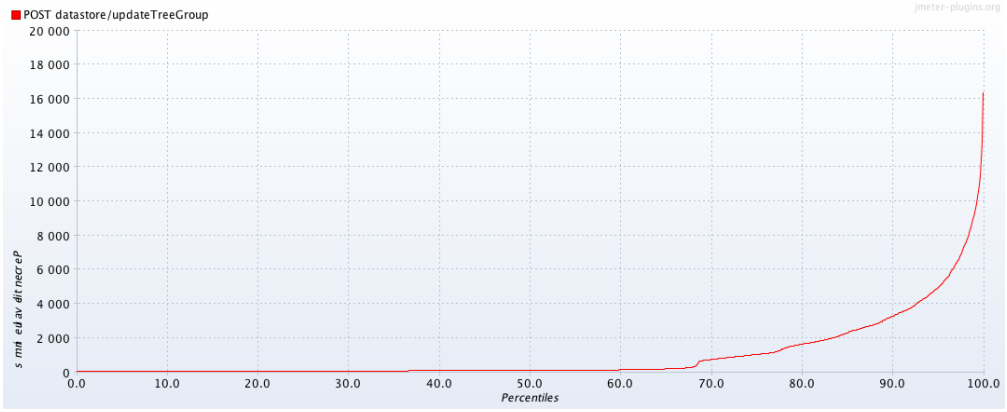

Figure 5.8: Response times percentiles for the parallel entity group update test.

5.6 Threats to Validity

The presented case study has been carefully implemented. During the design and realization of the prototypical benchmark code the attention was on clean and isolated test methods. Although this approach should avoid misleading results, there still remain internal and external threats to validity like described in [49]. Such threats are omnipresent in empirical studies.

Internal Threats

The selection of test cases for this study is based on previous experience with App Engine and is not derived from other studies. Other researches might choose different focus areas and other variations of tests. Each load test has been executed for one full round and then repeated to confirm the gained result at least once. Since Google only publishes a rudimentary status information and minor events are not disclosed, the detailed results may vary from execution to execution. All load tests have been executed over a limited number of time. Longer executions especially for the write-intensive tests and the entity group write limit would be interesting, but the high costs prohibited an extension of test execution times.

Google does not disclose the exact parameters which are considered for the App Master's instance provisioning. From this evolves a threat to the internal validity of the study. It is not possible to detect any learnings the App Master gains from previous test executions. It might adapt its provisioning strategy based on experiences and this would also change the measured results. This possible effect could be reduced by recreating an application's project container every time before the first execution of a test. Another strategy could be to leave the sandboxed runtime and switch to Managed VMs. They provide more control over the auto scaling component and are more transparent in their configuration. It would be also easier to log instance startups, restarts and termination.

To prevent distortion by network effects the DigitalOcean droplets have been split 1 to 1 between the east and west coast datacenters. This should bring the JMeter test machines closer to the respective Google Frontend service endpoints and reduce network latency. As project primary hosting location the US datacenters have been chosen, so the App Engine instances were likely to run in Google's data center in Council Bluffs, Iowa, and Berkeley County, South Carolina.

External Threats

The study is focused on App Engine and the Datastore and results cannot be generalized to other NoSQL datastores and further PaaS services. The runtime results are limited to the Java sandboxed runtime and might differ from runtime to runtime. Managed VMs have not been tested. Future generations of App Engine, which are expected to rely on containers instead of sandboxed runtimes, might show different results and especially the autoscaling behavior will be completely different.

The response time results may not be generalizable to other similar load tests with App Engine. They are highly dependent on the execution context. Also choosing other instance classes instead of the low-end F1 instances can lead to different results. Even if other instance classes are linearly faster than F1, an extrapolation to other classes should be avoided.

The test methods contained artificial code which has been developed for the performance case study only. It cannot be generalized to real world code and only focuses on a certain detail which is the subject of the test. Some methods use patterns which should be avoided in production-level code, like tracking the number of attempts with a thread-local variable.

5.7 Conclusion

The very different nature of App Engine and the PaaS approach makes it harder to measure different low-level metrics. Non-functional properties are dominated by the provided cloud environment and not by the applications itself. Therefore, this study focused on three research questions that directly relate to the benchmark and which are not contentual depended to the App Engine platform itself.

RQ1: Is JMeter an adequate tool to create meaningful load tests for cloud environments?

JMeter turned out as a very sophisticated load testing tool and a simple to use workload generator. The only problem was the high memory use during long running requests, which brought the memory of the droplets to an upper bound and it was required to find the right thread count per test to get a good request output, but also a smoothly running container inside the droplet. A very useful addition was the ability to use sequence generators from the YCSB. This enabled more realistic testing scenarios and also moved the test in this study closer to other load tests which were published for other NoSQL databases. Due the open source nature of JMeter it was also possible to look up internals like the exact generation of the result files and the supported formats. The ability to export the results as CSV files made it easier to merge them after the tests, since every instance created its own result file. And the resulting merged CSV could be imported in Excel and R for further analyzes and an in-depth look on the performance.

The DigitalOcean droplets performed well, but their limited memory was a constraint for the test runs. 512 megabyte of memory are not sufficient to execute a lot of JMeter threads in parallel and often the limit was at 20 threads per instance for write tests. This also limited the total generated traffic, however for short-living read tests a number of 50 threads per instance has been achieved. Using the DigitalOcean API to start and stop droplets was a very comfortable way to reduce the total runtime and therefore also kept the costs lower than in traditional virtual server environments. Each droplet started in seconds and was ready for further commands in under a minute. The preinstalled Docker worked as expected and made it easy to test the JMeter tests on the local development machine first. After it was clear that the test worked, it was executed by the Docker runtimes installed on the droplets. From time to time it was necessary to manually kill a container which didn't stopped properly. Summarizing the setup, it was a good choice to use Docker containers and DigitalOcean droplets, since both technologies were easy to automate and scriptable. With the help of some helper bash scripts it was straightforward to manage the whole system of ten virtual machines.

Though, the costs of the testing were noticeable, but within the expectations. The DigitalOcean droplets resulted in expenses of around $50 for the whole development of the tests and the execution. A very cost intensive factor were Datastore writes and read operations and the consumed instance hours. Specially tests with a high throughput resulted in a higher number of running instances and they consumed more instance hours. Real-word applications have to look for a cost effective entity model, which includes an optimal low of indexed properties, preferring free operations like key-based lookups over paid operations and setting the right balance between fast responses to high load situation and lower costs over time. The total costs of the used App

Engine applications were $30. So with a very reasonable amount of money a comprehensive test of various components of App Engine were possible.

RQ2: How does App Engine react on incoming traffic and what patterns can be detected during a load test? What are the effects of different write strategies for the Datastore?

Tracing the real active instance count was not possible. App Engine has no detailed enough logging for a fine-grained breakdown. With the instance count metric in the Cloud Console it was at least possible to get an impression how the App Master scheduler reacts on different incoming requests. A general characterization is that a high number of writes blocks the request processing due the relatively long commit phase. Therefore, the App Master assigns more instances to the application. The reaction time on incoming traffic was quite short. During seconds the number of instances could go from zero to over 50 instances. A peak of 162 instances has been reached during the retrieval test. This means that App Engine easily spins up a high number of standard F1 instances to serve incoming read requests in parallel. After the test run finished, the number of instances sticked high over a longer time and a manual intervention was necessary. It's likely that App Engine is very active in avoiding underprovisioning and as a consequence is more conservative in destroying instances after a high traffic situation occurred.

Writing large entities to the Datastore will increase the retrieval time and has a negative effect on the overall performance. Even if it's possible to store byte arrays in an entity, it should be avoided to store anything else then text-based or numeric data.

The read performance of the Datastore was impressive and App Engine provided enough instances to handle the high load situations with constant response times and without any errors. Though, normal read operations which required an index lookup were costly since every lookup and every retrieved entity cost money. Using the memcache and the built-in support for transparent caching in Objectify can prohibit costly operations. Transactions are not only useful to ensure consistent data, they also have a positive effect on the write performance. Instead of putting every single write in a small implicit transaction, it looks like App Engine collects all writes and prepares all needed resources to persist the transaction. Persisting a transaction requires a complex write backed by the Paxos protocol, so the unit of work inside of it should be small and compact. All time consuming computations should be performed before the actual start of a transaction. This also avoids write contention, which can hinder transactions from persisting changes to an entity group. There is no noticeable difference in query time between various access strategies. The Datastore makes no difference between old data and newer records. The index scans are in both cases fast and reliable. The number of results only has a small effect on the query performance and can be neglected in real-world applications. Key-based lookups can avoid costs since they are free, on the other side it is recommended to avoid the keys as lookup property and use normal properties instead. In this point the right balance between saving costs and more flexible queries has to be found. In general queries were executed fast and the overall performance of the load test was a little bit under the targeted number of 3,000 requests per second for reads, but at least at the peak the throughput was clearly over that target. Nevertheless, increasing the number of droplets or using more powerful droplet configurations with more memory is definitely

possible and it's undoubtedly possible to scale up for an even higher throughput.

RQ3: What are the effects of the throughput limit of 1 write per second per entity group?

During the load test a constant throughput of 45 requests per second per entity group has been detected. Though, some test requests needed a high number of attempts to save a single entity. In user-facing applications this would punish some users with high response times and exceptions, whereas the majority will still get a quite acceptable experience. So this recommendation can be seen as a soft limit and a general data model design advice. Badly designed applications will see a lower general throughput and and increased number of concurrency exceptions for writes. An important additional information is that batch put operations or write grouped into transactions count as single write for the throughput limit. So developers can increase their throughput of single entity manipulations by grouping those write together in an atomic bundle. Due the quite simple design of the test and the relatively small amount of test data, the peak write throughput per second per entity group might be lower than found in this case study.

The Datastore in combination with App Engine is highly optimized for web applications and their principal data structure. Getting the entity group boundaries right and keeping the groups compact is elementary for good application performance and to reduce write contention. Even if the write limit per entity group per second seems to be only a design guideline, the overall write performance per entity group is still far behind other NoSQL databases.

Final Thoughts

This case study showed some exemplary tests how App Engine works together with the Datastore. It provided insights what operations perform good and what patterns should be avoided. The Datastore emerged as a scalable data storage backend for distributed applications, but its transaction logic enforces some unique constraints. Even due the cost limitation for this study, some guidelines can be derived. One important outcome is the demonstration that JMeter is a valid test tool and works well together with Docker containers in a distributed test environment. It's now clearer for which type of applications App Engine and the Datastore are worthwhile targets and what kind of applications would not profit from the PaaS environment.

CHAPTER 6

Conclusion & Future Work

The range of covered topics includes both functional and non-functional properties of a cloud-based application. An experiment focused on the functional features of JPA-based data models preceded a case study focused on performance aspects, scalability and application elasticity. The following conclusion summarizes the results of both parts and provides an outlook for possible further research.

6.1 Conclusion

Today's PaaS services are heterogenous and common standards for provisioning, deployment and application packaging have not yet emerged. This makes it hard to choose the right service for an application, especially if an existing application should be modernized and migrated to the cloud. Not only the fulfillment of all functional requirements is critical for a successful migration, also non-functional requirement testing has an important role for the overall success. The dominant non-functional requirements for cloud-based applications are scalability, elasticity, responsiveness, and costs. It became clear that scalability and elasticity are the two primary qualities a software system optimized for the cloud should prove. Otherwise the main benefit of dynamically provisioned computing resource falls flat and a migration might not be rewarding under this viewpoint. Dynamic allocation of resources is the main characteristic scientific computing experiments looked for. It enables them to handle peak loads and to allocated a large number of computing resources without delay times until installation and deployment. Studies around scientific computing assumed functional equivalence of migrated software and performed an in-depth analysis on non-functional parameters and raw computing performance. The latter is a popular benchmark for IaaS clouds, but not directly applicable in the PaaS environment.

Looking at App Engine as a popular PaaS service and the underlying NoSQL-oriented Datastore revealed some interesting details. From the functional requirement viewpoint an elaborated analysis of the GAE/J plugin has been realized. This was necessary to show how well existing JPA features are supported and where workarounds are possible. The different nature

of Datastore entities compared to relational data has been shown as major problem for every migration. The key of a Datastore entity has different semantic meanings than a primary key in a relational database. It forms the scope of a transaction and a parent-child relationship with other entities, not only a unique identifier. This makes transformations from JPA data models to GAE/J JPA data models complex and entails several potential pitfalls. A good example is the relationship management between entities, which becomes more complicated due the missing advanced primary/foreign key semantics. Relationships in the Datastore are relatively weak and not managed by the database engine. Each query in the Datastore requires a predefined index, otherwise it fails. In contrast traditional RDBMS implement relational operations[1] as a standard feature, have support for declarative consistency constraints and allow queries on non-indexed data. Hence, migrations to App Engine and the Datastore require a knowledge about the target platform and its distinguishing characteristics. After an application has been successfully migrated the overall system is highly specific to the PaaS environment. The level of integration required to consume the different services makes it hard to revert the selection of a cloud provider. Applications following the Datastore design patterns not only gain most from the platform, they will also have cost advantages and higher throughput.

Objectify pointed out to be a solid alternative to the JPA implementation of the GAE/J plugin. Instead of copying RDBMS-like feature to the NoSQL-based Datastore it is designed to work with the Datastore only. This reduces the complexity of the object mapping, makes entity group management easier and reduces the level of abstraction from the raw entity data structure. Instead of providing a subset of the full JPA feature range, Objectify implements the whole Datastore set of features. It makes Datastore operations type-safe, which is an important property for programming in Java. The built-in memchache layer provides transparent configurable caching on application-wide scale. This increases read performance and reduces the need for direct reads. The transactional logic orients at the EJB standard and represents an expedient abstraction of native Datastore transactions. Each Objectify transaction is a single unit of work and must be idempotent, so repeated executions of it have to result in a consistent state. This concept is different to the JPA transaction model and much closer to what Datastore transactions are. Even due the fact Objectify is not an official library, Google promotes it over the JPA plugin in tutorials[2], recent articles, and on the frontpage of the Google Cloud Datastore product site[3]. Consequently, the framework can be recommended as a standard component for Java-based applications on App Engine.

Cloud Computing and Empirical Research Methods

The case study has been chosen as methodological approach due to the possibility for investigation of a cloud system in a production environment. Other methods have been rejected due a number of problems. Three considered methods have been a survey, a controlled experiment, and analyzing existing artifacts. Survey-based research would require to find a representative sample of Google's

[1] This includes joins, aggregations, referential integrity constraints, cardinalities, identifying or non-identifying relationships.

[2] `https://cloud.google.com/appengine/docs/java/gettingstarted/usingdatastore`

[3] `https://cloud.google.com/datastore/`

customers. Since there exists no information about who is using App Engine for what purpose, this would be an effortful approach. Furthermore, a controlled experiment requires a certain level of isolation and closure of the execution runtime. App Engine provides no way to execute and study an application in a separate experimental setting. Google does not provide any artifacts of App Engine except API definitions and basic runtime helpers. This makes it impossible to look at proprietary software artifacts and defeats this research method. Therefore, any investigation has to study effects during runtime in the cloud deployment. For this the case study is an established and reasonable approach.

At the beginning of the case study the basic principals and promises of App Engine have been reflected. From this two hypotheses emerged: App Engine allows applications to scale regardless the incoming traffic; and Datastore writes to a single entity group are limited to 1 write per second. Based on the emerged hypotheses three separate research questions have been answered using distributed load tests.

JMeter and Distributed Load Testing

Performance and load testing is a common method to show the scaling behavior of a system. The range of performance testing starts with incremental load tests to spot limits, involves stress testing and in it's extreme can even be performed in a destructive manner. For this work a incremental approach has been chosen. This enabled the test client to identify scaleable designs and performance bottlenecks.

Using container technology with Docker made it easy to define a load testing server configuration using a simple configuration file. As a result it was easy to scale the load tests to a sufficient amount of test instances. Every test instance then started a headless JMeter instance and executed the assigned load test. The API of DigitalOcean to start and stop virtual server instances further increased the level of automatization. One limitation was the missing monitoring per test instance. A workaround was checking the App Engine console in a regular interval. This has been sufficient for the scope of this test, but could be problematic with a significant higher number of running instances. Furthermore, one could fully automate the instance handling and start up more servers until a target load test pressure. However, this would be out of focus of this work and could be a separate study for its own.

Scalability, Elasticity and Overall Performance of App Engine

App Engine had a limited set of instances suitable for auto-scaling. These frontend instance classes[4] reflect the focus on serving web applications. Their memory limit is quite low with a default of 128 megabyte and a maximum of 1 gigabyte. Here the AWS Auto Scaling[5] service is more flexible offering the full range of instance types, keeping in mind that is based on the EC2 IaaS service and is not a PaaS offering. Where AWS requires at least some manual configuration, App Engine's auto scaling works out of the box and is extremely responsive to the current traffic. The App Master automatically started new containers in seconds and made them available

[4] Named F1 to F4_1G

[5] `https://aws.amazon.com/autoscaling/`

for serving requests. The storage link to the Datastore service handled even high amount of concurrent writes and showed no problems to process thousands of parallel reads.

Application elasticity has been tested with computational or memory expensive benchmarks. These benchmarks tried to consume as much resources of a single instance as possible. Due the limited monitoring capabilities only the following estimation based on the cloud console can be given: Whenever the application became CPU or memory intensive, App Engine detected the critical situation and started new instances. No situation occurred where the overall application behaved unresponsive or was unavailable. During intensive phases of instances creation minor changes in the response times and throughput have been detected.

Summarizing the results a general upper limit for read performance has not been detected. During higher load App Engine spins up new instances and processes the requests in parallel. Writes to the Datastore take more computing resources, but also no limit has been reached. Only writes to the same entity group lead to Datastore contention. This has also been reported in the application logs. App Engine tries to avoid a bad experience for a majority of users and defers only a small portion of the incoming requests.

6.2 Future Work

An important limiting factor of this work has been the available budget for the test application in the production environment. Running write-intensive performance tests on App Engine is a costly operation. Costs have been dominated by writes to the Datastore, not by reads or instance hours. Bringing this work a step further would require a defined budget or a special free credit funding by Google. The latter would question the independence of a case study like described in chapter 5. Before the execution of a load test the expected cost should be estimated. This will prevents over-quota errors and test abortions during long-time executions, which occurred during some write-intensive testing.

The impact of Objectify on the read and write performance has been neglected in favor of a broader range of tests. It has been omitted to look if there are significant performance differences between raw Datastore Java APIs and the Objectify-based approach. Since Objectify provides a higher abstraction level of Datastore operations and an EJB-like transaction support, more information about this area would be useful. However, the effect of the object-to-datastore mapping layer on the overall performance has been proven to be insignificant in the overall picture. Efficient caching and the relatively poor write performance of the Datastore are dominant factors, not the chosen persistence framework.

An omitted topic has been the shift to container-based services and Managed VMs. App Engine offers a number of auto-scaling language runtimes, but also runs on top of semi-automatic Managed VMs[6]. PCWorld[7] speculated that "Some Google App Engine languages may not get updates" and that the future generation of applications will exclusively work on top of this hosting service. Newer SDK versions[8] warn developers if the do not use Java 7 to create, test and deploy applications, ignoring that Oracle stopped the public support for Java 7 in April

[6] `https://cloud.google.com/appengine/docs/managed-vms/`

[7] `http://www.pcworld.com/article/2938352/`

[8] 1.9.28 was the latest available SDK at the time of writing.

2015. Java 8 is only available if App Engine is used together with Managed VMs. This is also an indicator for the steady transition to a different runtime hosting environment. Updates and additions to this thesis should consider this development and look closer at Managed VMs and the consequences for scalability and application migration strategies. Managed VMs support manual and automatic scaling[9]. Though, compared to the sandboxed language runtimes a Managed VM takes significantly longer to start, typically minutes compared to seconds. Furthermore, Instead of integrated API services provided by the runtime environment, some cloud services have to be accessed via their public REST APIs. These drawbacks become abolished by the less restrictive software environment: Managed VMs have am ephemeral writeable local disc, allow background threads and child processes, feature a full networking stack, and custom binaries can be installed. This makes the Managed VM runtime more interesting as a migration target since the lowered restrictions and the more traditional approach for building a software stack. Subsequent experiments and studies could improve the knowledge about this new deployment target, even if Managed VMs are still in beta and not covered by any SLA.

[9]Compared to the sandboxed runtimes Managed VMs need more configuration for automatic scaling. The developer has to provide a minimum and maximum number of instances and and optional target average CPU utilization over all instances. The level of CPU utilization is the only parameter taken into account for scaling. At least one instance has to be active at any time, compared to none in the sandboxed environment.

Appendix

Google App Engine Details

Java System Properties

```
java.specification.version -> 1.7
java.vendor -> Google Inc.
line.separator ->
java.class.version -> 51.0
java.util.logging.config.file -> WEB-INF/logging.properties
com.google.appengine.runtime.version -> Google App Engine/1.9.21
java.specification.name -> Java Platform API Specification
java.vendor.url -> http://wiki.corp.google.com/twiki/...
java.vm.version -> 1.7.0
os.name -> Linux
java.version -> 1.7.0
java.vm.specification.version -> 1.7
com.google.appengine.application.version -> 1.384854464320720
user.dir -> /base/data/home/apps/s~gaebenchapp/1.384854464320720
java.specification.vendor -> Oracle Corporation
java.vm.specification.name -> Java Virtual Machine Specification
com.google.appengine.application.id -> seesternstachelhaut
java.vm.vendor -> Oracle Corporation
file.separator -> /
path.separator -> :
java.vm.specification.vendor -> Oracle Corporation
java.vm.name -> OpenJDK Client VM
file.encoding -> ANSI_X3.4-1968
com.google.appengine.runtime.environment -> Production
```

Performance Case Study Source Code

All sources related to this master's thesis are available at Github:
`https://github.com/botic/gae-performancecasestudy`

List of Figures

List of Tables

Bibliography

[1] Louis Columbus. *Roundup Of Cloud Computing Forecasts And Market Estimates Q3 Update, 2015*. 2015. URL: http://www.forbes.com/sites/louiscolumbus/2015/09/27/roundup-of-cloud-computing-forecasts-and-market-estimates-q3-update-2015/ (visited on 11/05/2015).

[2] Katsantonis Konstantinos, Filiopoulou Evangelia, Michalakelis Christos, and Nikolaidou Mara. "Cloud computing and economic growth". In: *PCI '15 Proceedings of the 19th Panhellenic Conference on Informatics*. New York: ACM, 2015, pp. 209–214. DOI: 10.1145/2801948.2802000.

[3] Dan Sanderson. *Programming Google App Engine with Java: Build & Run Scalable Java Applications on Google's Infrastructure*. O'Reilly Media, 2015.

[4] KevinMacG Adams. "Introduction to Non-functional Requirements". In: *Nonfunctional Requirements in Systems Analysis and Design*. Vol. 28. Topics in Safety, Risk, Reliability and Quality. Springer International Publishing, 2015, pp. 45–72. DOI: 10.1007/978-3-319-18344-2.

[5] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. "UML-Based Modeling of Non-Functional Requirements in Telecommunication Systems". In: *The Sixth International Conference on Software Engineering Advances (ICSEA 2011)* (2011), pp. 213–220.

[6] Dorina C Petriu. "Challenges in Integrating the Analysis of Multiple Non-Functional Properties in Model-Driven Software Engineering". In: *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*. WOSP '15. New York, NY, USA: ACM, 2015, pp. 41–46. DOI: 10.1145/2693561.2693566.

[7] Ronaldo Gonçalves Junior, Tiago Rolim, and Aplicada Ppgia. "A Multi-Criteria Approach for Assessing Cloud Deployment Options Based on Non-Functional Requirements". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York: ACM New York, 2015, pp. 1383–1389. DOI: 10.1145/2695664.2695923.

[8] Martti Vasar, Satish Narayana Srirama, and Marlon Dumas. "Framework for monitoring and testing web application scalability on the cloud". In: *Proceedings of the WICSA/ECSA 2012 Companion Volume* (2012), pp. 53–60. DOI: 10.1145/2361999.2362008.

[9] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Rego, Danielo Gonçalves Gomes, and Jose Neuman de Souza. "Elasticity in Cloud Computing :

A Survey". In: *annales des télécommunications* (2014). DOI: 10.1007/s12243-014-0450-7.

[10] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. "SMICloud: A Framework for Comparing and Ranking Cloud Services". In: *2011 Fourth IEEE International Conference on Utility and Cloud Computing* Vm (2011), pp. 210–218. DOI: 10.1109/UCC.2011.36.

[11] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. "Scalability, Elasticity, and Efficiency in Cloud Computing: A Systematic Literature Review of Definitions and Metrics". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '15. New York: ACM, 2015, pp. 83–92. DOI: 10.1145/2737182.2737185.

[12] Jakob Nielsen. *Website Response Times*. 2010. URL: http://www.nngroup.com/articles/website-response-times/ (visited on 10/29/2014).

[13] *eCommerce Web Site Performance Today*. Tech. rep. Forrester Consulting on behalf of Akamai Technologies, Inc., 2009.

[14] Russell Thackston and Ryan C Fortenberry. "The performance of low-cost commercial cloud computing as an alternative in computational chemistry". In: *Journal of Computational Chemistry* 36.12 (2015), pp. 926–933. DOI: 10.1002/jcc.23882.

[15] A Iosup, S Ostermann, M N Yigitbasi, R Prodan, T Fahringer, and D H J Epema. "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 931–945. DOI: 10.1109/TPDS.2011.66.

[16] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50. DOI: 10.1002/spe.995.

[17] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville. "Cloud migration: A case study of migrating an enterprise it system to iaas". In: *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE. 2010, pp. 450–457. DOI: 10.1109/CLOUD.2010.37.

[18] Min Yao, Peng Zhang, Yin Li, Jie Hu, Chuang Lin, and Xiang Yang Li. "Cutting Your Cloud Computing Cost for Deadline-Constrained Batch Jobs". In: *Web Services (ICWS), 2014 IEEE International Conference on*. 2014, pp. 337–344. DOI: 10.1109/ICWS.2014.56.

[19] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), p. 50. DOI: 10.1145/1721654.1721672.

[20] Tharam Dillon, Chen Wu, and Elizabeth Chang. “Cloud Computing: Issues and Challenges”. In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications* (2010), pp. 27–33. DOI: 10.1109/AINA.2010.187.

[21] Ying Zhang, Gang Huang, Xuanzhe Liu, and Hong Mei. “Tuning Adaptive Computations for Performance Improvement of Autonomic Middleware in PaaS Cloud”. In: *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 732–733. DOI: 10.1109/CLOUD.2011.66.

[22] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. “How a consumer can measure elasticity for cloud platforms”. In: *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering - ICPE ’12*. New York, New York, USA: ACM Press, 2012, p. 85. DOI: 10.1145/2188286.2188301.

[23] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. “High-Performance Cloud Computing: A View of Scientific Applications”. In: *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks* (2009), pp. 4–16. DOI: 10.1109/I-SPAN.2009.150.

[24] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. “Dynamically scaling applications in the cloud”. In: *ACM SIGCOMM Computer Communication Review* 41.1 (2011), p. 45. DOI: 10.1145/1925861.1925869.

[25] Davide Franceschelli, Danilo Ardagna, Michele Ciavotta, and Elisabetta Di Nitto. “SPACE4CLOUD”. In: *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds - MultiCloud ’13*. New York, New York, USA: ACM Press, 2013, p. 27. DOI: 10.1145/2462326.2462333.

[26] Radu Prodan and Michael Sperk. “Scientific computing with Google App Engine”. In: *Future Generation Computer Systems* 29.7 (2013), pp. 1851–1859. DOI: 10.1016/j.future.2012.12.018.

[27] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. “On the Performance Variability of Production Cloud Services”. In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2011, pp. 104–113. DOI: 10.1109/CCGrid.2011.22.

[28] Rodrigo N. Calheiros, Rajiv Ranjan, and Rajkumar Buyya. “Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments”. In: *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 295–304. DOI: 10.1109/ICPP.2011.17.

[29] Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Adrian Muresan, and Frédéric Desprez. “Building safe PaaS clouds: A survey on security in multitenant software platforms”. In: *Computers & Security* 31.1 (2012), pp. 96–108. DOI: 10.1016/j.cose.2011.10.006.

[30] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. “WebMate: Generating Test Cases for Web 2.0”. English. In: *Software Quality. Increasing Value in Software and Systems Development SE - 5*. Ed. by Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann. Vol. 133. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2013, pp. 55–69. DOI: 10.1007/978-3-642-35702-2.

[31] Gültekin Ataş and Vehbi Cagri Gungor. “Performance evaluation of cloud computing platforms using statistical methods”. In: *Computers & Electrical Engineering* 40.5 (2014), pp. 1636–1649. DOI: 10.1016/j.compeleceng.2014.03.017.

[32] Kristina Kolic. “Performance Analysis of a New Cloud e-Business Solution”. In: *MIPRO 2015 - 38th International Convention* (2015).

[33] C Vazquez, R Krishnan, and E John. “Cloud Computing Benchmarking: A Survey”. In: *The 2014 World Congress in Computer Science, Computer Engineering, and Applied Computing*. 2014.

[34] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. “CloudCmp : Comparing Public Cloud Providers”. In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010. DOI: 10.1145/1879141.1879143.

[35] Alexandru Iosup, Radu Prodan, and Dick Epema. “IaaS Cloud Benchmarking: Approaches, Challenges, and Experience”. In: *Cloud Computing for Data-Intensive Applications*. New York: Springer New York, 2014, pp. 83–104.

[36] Seyhan Yazar, George E C Gooden, David A Mackey, and Alex W Hewitt. “Benchmarking undedicated cloud computing providers for analysis of genomic datasets.” In: *PloS one* 9.9 (2014). DOI: 10.1371/journal.pone.0108490.

[37] Ryan P Taylor, Frank Berghaus, Franco Brasolin, Cristovao Jose, Domingues Cordeiro, Ron Desmarais, Laurence Field, Ian Gable, Domenico Giordano, Alessandro Di Girolamo, John Hover, Matthew Leblanc, Peter Love, Michael Paterson, and Randall Sobie. *The Evolution of Cloud Computing in ATLAS*. Tech. rep. The ATLAS collaboration, 2015. URL: http://cds.cern.ch/record/2016817/files/ATL-SOFT-PROC-2015-049.pdf?subformat=pdfa.

[38] Bin Sun, Brian Hall, Hu Wang, Da Wei Zhang, and Kai Ding. “Benchmarking Private Cloud Performance with User-Centric Metrics”. In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. Boston, MA, 2014, pp. 311–318. DOI: 10.1109/IC2E.2014.74.

[39] Eytan Bakshy and Eitan Frachtenberg. “Design and Analysis of Benchmarking Experiments for Distributed Internet Services”. In: *WWW ’15 Proceedings of the 24th International Conference on World Wide Web*. Geneva, Switzerland, 2015, pp. 108–118. DOI: 10.1145/2736277.2741082.

[40] Liang Zhao, Sherif Sakr, Anna Liu, and Athman Bouguettaya. *Cloud Data Management*. Cham: Springer International Publishing, 2014. DOI: 10.1007/978-3-319-04765-2.

[41] Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai, and Adam Barker. "Cloud Benchmarking for Performance". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science Cloud*. 2014, pp. 1–6. DOI: 10.1109/CloudCom.2014.28.

[42] Joel Scheuner, Jürgen Cito, Philipp Leitner, and Harald Gall. "Cloud WorkBench: Benchmarking IaaS Providers Based on Infrastructure-as-Code". In: *Proceedings of the 24th International Conference on World Wide Web Companion*. WWW '15 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015, pp. 239–242. DOI: 10.1145/2740908.2742833.

[43] I Shabani, A Kovaci, and A Dika. "Possibilities Offered by Google App Engine for Developing Distributed Applications Using Datastore". In: *Computational Intelligence, Communication Systems and Networks (CICSyN), 2014 Sixth International Conference on*. 2014, pp. 113–118. DOI: 10.1109/CICSyN.2014.35.

[44] Mike Keith and Merrick Schincariol. *Pro JPA 2: Mastering the Java™ Persistence API*. 2009.

[45] Security Explorations. *Google App Engine Java security sandbox bypasses*. Tech. rep. 2014. URL: http://www.security-explorations.com/en/SE-2014-02-details.html.

[46] Jason Baker, Chris Bond, James C Corbett, J J Furman, Andrey Khorlin, James Larson, L Jean-michel, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore : Providing Scalable , Highly Available Storage for Interactive Services". In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234.

[47] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally-Distributed Database". In: *Proceedings of OSDI 2012* (2012), pp. 1–14. DOI: 10.1145/2491245.

[48] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable : A Distributed Storage System for Structured Data". In: *OSDI'06: Seventh Symposium on Operating System Design and Implementation*. Seattle, 2006.

[49] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2008), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[50] Steve Perry, Dewayne E. and Sim, Susan Elliott and Easterbrook. “Case Studies for Software Engineers”. In: *Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China: ACM, 2006, pp. 1045–1046. DOI: 10.1145/1134285.1134497.

[51] Brian Fitzgerald and Debra Howcroft. “Towards dissolution of the IS research debate: From polarization to polarity”. In: *Journal of Information Technology* 13.4 (1998), pp. 313–326. DOI: 10.1057/jit.1998.9.

[52] Khalid Alhamazani, Rajiv Ranjan, Prem Prakash Jayaraman, Karan Mitra, Chang Liu, Fethi Rabhi, Dimitrios Georgakopoulos, and Lizhe Wang. “Cross-Layer Multi-Cloud Real-Time Application QoS Monitoring and Benchmarking As-a-Service Framework”. In: (2015). arXiv: 1502.00206.

[53] Charles Anderson. “Docker”. In: *IEEE Software* 32 (2015). DOI: 10.1109/MS.2015.62.

[54] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*. New York, New York, USA: ACM Press, 2010, p. 143. DOI: 10.1145/1807128.1807152.