Karlsruher Institut für Technologie

Konzeption und Entwicklung eines skalierenden Business Intelligence Systems auf Google App Engine

Bachelorarbeit
von
Johannes Rösch

An der Fakultät für
Wirtschaftswissenschaften

In dem Studiengang
Informationswirtschaft

eingereicht am 30.09.2013 beim
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
des Karlsruher Instituts für Technologie

Referent: Prof. Dr. Rudi Studer
Betreuer: Dr. Max Völkel

Zusammenfassung

Die Arbeit befasst sich mit den Möglichkeiten und Chancen, welche sich für Business Intelligence durch dem Betrieb auf einer Cloud-Plattform, wie Google App Engine, ergeben. Insbesondere bei dem Bedarf nach einem System mit einer hohen Skalierbarkeit und flexiblen Kostenstrukturen kann diese Form der Infrastruktur Vorteile mit sich bringen.

Durch die Konzeption eines Systems auf Google App Engine werden verschiedene Entwurfsmuster aufgezeigt und diskutiert. Die effiziente Einbindung der verteilten Datenbank ist für einen skalierenden Betrieb von besonderer Bedeutung. Mit einer prototypischen Umsetzung des Entwurfs, konnte die Erfüllung der gesetzten Ziele überprüft werden. Es zeigte sich, dass mit einem System auf App Engine eine hohe Skalierbarkeit bezüglich der eingehenden Datenmenge möglich ist. Die Flexibilität der Auswertungsmöglichkeiten wird von der Datenstruktur eingeschränkt. Wird diese Flexibilität nicht benötigt, ist auf Google App Engine ein skalierbares und kostenorientiertes Business Intelligence System möglich.

Abstract

The thesis deals with possibilities and opportunities for business intelligence by operating on a cloud-based platform such as Google App Engine. Particularly with regard to requirement for a system with high scalability and flexible cost structures, this infrastructure can be beneficial.

By designing a business intelligence system on Google App Engine, different design patterns are demonstrated and discussed. A efficient integration of the distributed database is importent for a scalable service. Through a prototypical implementation of the design, the aim could get verified. It shows, that a system with high scalability, according to the amount of incoming data is possible on Google App Engine. The flexibility of analisys options is limited by data structure. In case this flexibility is not required, Google App Engine allows a scalable and cost-oriented business intelligence system.

Inhaltsverzeichnis

1 Einleitung **4**
1.1 Problemstellung . . . 4
1.2 Lösungsansatz . . . 4
1.3 Aufgabenstellung . . . 4
1.4 Aufbau der Arbeit . . . 5

2 Hintergrund **6**
2.1 Business Intelligence . . . 6
2.2 Google App Engine . . . 7
2.3 Highcharts . . . 12

3 Entwurf **13**
3.1 Anforderungsanalyse . . . 13
3.2 Gesamtsystem . . . 15
3.3 Gruppierung von Ereignissen . . . 17
3.4 Persistente Datenhaltung . . . 19
3.5 Puffer und Zwischenspeicher . . . 22

4 Umsetzung **24**
4.1 Überblick . . . 25
4.2 Webschnittstellen . . . 25
4.3 Bearbeitung von Abfragen . . . 28

5 Evaluation **28**
5.1 Durchführung und Rahmenbedingungen . . . 28
5.2 Ergebnisse . . . 31
5.3 Beurteilung . . . 33

6 Verwandte Arbeiten **34**
6.1 Piwik . . . 35
6.2 Microsoft SQL Server . . . 36
6.3 Apache Hadoop . . . 37
6.4 Vergleich . . . 38

7 Zusammenfassung und Ausblick **39**
7.1 Zusammenfassung . . . 39
7.2 Ausblick . . . 40

1 Einleitung

Auf der technischen Seite geht es bei Business Intelligence um die Erhebung, Verarbeitung und Darstellung geschäftsrelevanter Daten. Auf Seite des Managements bieten diese Informationen die Grundlage aller Produktentscheidungen. Ob zur Produkt- und Prozessverbesserung oder zur Ermittlung aller wichtigen Kennzahlen im Unternehmen, die Verfügbarkeit umfangreicher Informationen über die Geschäftsbereiche bietet klare Vorteile. Aus diesem Grund ist Business Intelligence für viele Unternehmen ein unverzichtbarer Bereich geworden.

1.1 Problemstellung

Bei neuen Produkten oder neuen Produktideen können Erkenntnisse über die Kunden, sowie deren Verhalten, die weitere Produktentwicklung stark beeinflussen. Für innovative, kleine Unternehmen und Startups spielt Business Intelligence deshalb eine mindestens genauso große Rolle, wie für den Mittelstand und große Konzerne. Besonders in diesem Umfeld können sich jedoch neue Anforderungen an den Betrieb eines Business Intelligence Systems ergeben. Zum einen spielt die Skalierbarkeit des Systems eine wichtige Rolle: Ein kleines System kann bei großem Wachstum schnell an die Leistungsgrenzen kommen. Zum anderen sollten die entstehenden Kosten für das System in Relation zur Anzahl der Kunden stehen. Auch anfänglich hohe Investitionskosten für Infrastruktur können unter Umständen eine Hürde darstellen.

Wenn auch kleine Unternehmen in den Genuss umfassender Informationen über ihre Produkte kommen möchten, benötigen sie ein System, das diese Anforderungen erfüllen kann. Der Betrieb muss einfach und flexibel gehalten werden. Trotzdem muss es die Aufgaben des Business Intelligence, Daten aus dem Unternehmen in nützliches Wissen zu überführen [RSA13, S. 44], vollständig erfüllen. Es stellt sich also die Frage nach einem Business Intelligence System, bei dem alle Faktoren ausreichend Berücksichtigung finden.

1.2 Lösungsansatz

Für eine hohe Skalierbarkeit wird eine entsprechende Infrastruktur benötigt. Ein Lösungsansatz bietet die Nutzung von Cloud-Diensten. Diese bieten Möglichkeiten die Anwendungen skalierbar bzw. skalierend betreiben zu können. Durch das dort übliche Pay-per-use-Modell kann auch eine von der Nutzung abhängige Kostenstruktur erreicht werden. Die Einstiegskosten für die Cloud-Nutzung sind ebenfalls sehr gering.

Google ist einer dieser Anbieter, der den Betrieb von Anwendungen auf der Platform-as-a-Service-Ebene anbietet. Diese autonom skalierende Infrastruktur, auf der eigene Anwendungen betrieben werden können, kann die Grundlage für ein Business Intelligence System bilden. Eine bestehende open-source Implementierung[1] demonstriert die Machbarkeit der Idee. Der Kern des Systems soll aus einer Anwendung auf Google App Engine bestehen. Eine API soll Abfragen eines (exemplarisch implementierten) Frontend ermöglichen.

1.3 Aufgabenstellung

Die Aufgabe besteht in der Entwicklung und Evaluierung einer prototypischen Business Intelligence Lösung, die auf Google App Engine betrieben werden kann. Damit soll die Umsetzbarkeit von Business Intelligence auf App Engine gezeigt werden.

[1] http://xydra.googlecode.com/svn/trunk/com.sonicmetrics.core/

In der Analysephase sollen die Rahmenbedingungen und Anforderungen, die an ein Business Intelligence System gestellt werden, analysiert und abgeschätzt werden. Das Ergebnis stellt die Grundlage für den weiteren Entwicklungsprozess dar.

Bei der Umsetzung sollen die Vorteile, die sich aus dem Betrieb in der Cloud-basierten Umgebung ergeben, Berücksichtigung finden. So soll die Anwendung eine hohe Skalierbarkeit bezüglich der eingehenden Datenmenge und Datenrate aufweisen. Ebenso sollen Techniken, die eine effiziente Verarbeitung und einen schnellen Zugriff ermöglichen, eingesetzt werden. Eine besondere Herausforderung stellt die effiziente Nutzung der Eigenschaften der verteilten Datenbank sowie der speziellen Caching-Dienste von App Engine dar. Insgesamt soll dadurch ein ökonomischer Betrieb in der Cloud möglich sein.

Ein exemplarisches Frontend dient zur Darstellung der Daten. Der Zugriff erfolgt über eine von der Anwendung bereitgestellte Webschnittstelle. Für die grafische Darstellung der Ergebnisse kann die Bibliothek Highcharts[2] verwendet werden.

Abschließend gilt es den Erfolg der Umsetzung zu erfassen und zu bewerten. Anhand dieser Evaluation kann eine Abschätzung erfolgen, in wie weit der erfolgreiche Betrieb eines Business Intelligence System auf Google App Engine möglich ist.

1.4 Aufbau der Arbeit

Die Arbeit untergliedert sich in sieben Kapitel. Angefangen mit den theoretischen Grundlagen, über die Konzeption, bis hin zur Evaluation und einem Ausblick.

Im ersten Kapitel (siehe S. 4) findet eine Heranführung an das Thema statt. Die Motivation, die hinter dieser Arbeit steht, sowie die Ziele, die damit verfolgt werden, befinden sich ebenfalls in diesem Abschnitt.

Das zweite Kapitel (S. 6) zeigt den theoretischen Rahmen von Business Intelligence und Google App Engine auf. Die Schwerpunkte sind dabei auf die im Verlauf der Arbeit benötigten Unterpunkte gelegt. Des Weiteren werden die sonstigen verwendeten Programmbibliotheken kurz erläutert.

Im dritten Kapitel (S. 13) wird das Konzept des Systems erörtert und Richtwerte für die gesetzten Ziele festgelegt. Um die festgelegten Ziele zu erreichen, wird ein modular aufgebautes Gesamtsystem entwickelt. Auf die einzelnen Module sowie ihre Vor- und Nachteile wird detailliert eingegangen.

Das vierte Kapitel (S. 24) legt die programmiertechnische Umsetzung dar. Es wird geklärt welche Einstellungen im System und an der Konfiguration der verschiedenen App-Engine-Dienste vorgenommen wurden. Ebenso wird veranschaulicht, welche Besonderheiten sich bei der Umsetzung der Module ergeben haben.

Das fünfte Kapitel (S. 28) gibt Auskunft über den Erfolg der Umsetzung des Business Intelligence Systems. Zuerst wird erläutert in welchem Rahmen eine Bewertung stattgefunden hat. Anschließend werden die Ergebnisse vorgestellt und analysiert.

Im sechsten Kapitel (S. 34) findet eine vergleichende Betrachtung von weiteren Anwendungen aus dem Business Intelligence Bereich satt. Es wird versucht eine Aussage

[2] siehe Kapitel 2.3

über den Aufbau und die Umsetzung dieser Anwendungen zu erhalten. Gleichzeitig wird aufgezeigt, in wie weit diese Anwendungen die gesetzten Ziele abdecken können.

Im siebten Kapitel (S. 39), dem inhaltlichen Abschluss der Arbeit, wird eine Zusammenfassung über die gewonnenen Erkenntnisse gegeben. In einem Ausblick sollen auch weiterführende Fragestellungen aufgezeigt werden.

2 Hintergrund

Für die Konzeption eines Business Intelligence Systems auf Google App Engine ist es wichtig, sich mit den theoretischen Hintergrund der beiden Bereiche auseinander zu setzen. Zum einen um ein Rahmen für Business Intelligence aufzuzeigen, zum anderen um die technischen Grundlagen und der Aufbau von App Engine zu erläutern.

2.1 Business Intelligence

2.1.1 Definition

Unter dem Begriff „Business Intelligence“ (BI) wird heutzutage weit mehr als die einfache Übersetzung „Geschäftsinformationen“ verstanden. Im Kern jedoch, dreht sich im BI alles um diese Informationen. Eine präzisere Eingrenzung erlaubt die Definition von Chamoni und Gluchowski [KBM10, S. 3]. In dieser steht BI als Begriff „zur Kennzeichnung von Systemen [...], die auf der Basis interner Leistungs- und Abrechnungsdaten sowie externer Marktdaten in der Lage sind, das Management in seiner planenden, steuernden und koordinierenden Tätigkeit zu unterstützen“.

2.1.2 Entwicklung und Umfang

Historisch hat die Unterstützung des Managements durch Informationssysteme schon früh begonnen. Seit Beginn der elektronischen Datenverarbeitung wurde versucht aufbereitete Informationen dem Management zugänglich zu machen. In den 80er wurden diese mit dem Oberbegriff „Management Support Systems“ zusammengefasst [KBM10, S. 1]. Der Umfang beschränkte sich dabei auf einfache Reports. Schnell wurde klar, dass sich mit den ständig weiterentwickelnden Systemen weitere Auswertungen durchführen lassen. Nach und nach kamen komplexere Auswertungen auf größeren Datenbeständen hinzu [RSA13, S. 265].

Die Anwendungsbereiche von BI im heutigen Sinne sind weit fortgeschritten. Selbst bei einem engen Verständnis gehören neben verschiedenen Informationssystemen auch das Online Analytical Processing (OLAP) dazu [KBM10, S. 3]. Unter OLAP ist ein flexibles Abfragesystem für Ad-hoc-Auswertungen zu verstehen [KBM10, S. 99]. In einem analyseorientiertem Verständnis von BI sind zusätzlich Anwendungen inbegriffen, auf die bei der Entscheidungsfindung in irgendeiner Form zugegriffen wird [KBM10, S. 4]. Eine weite Auslegung von BI erlaubt den Einbezug aller direkt und indirekt beteiligten Systeme. Dazu gehören sowohl Darstellungs- und Präsentationswerkzeuge, als auch die Datenaufbereitung und Speicherung [KBM10, S. 4].

2.1.3 Zweck

Durch eine schnell wandelnde Geschäftswelt, hin zu komplexen Prozessen mit immer mehr Informationen, ist es für das Management schwierig einen umfassenden Überblick zu behalten [RSA13, S. 44]. Aus diesem Zweck werden Datenverarbeitungssysteme entwickelt, die diese

Komplexität verarbeiten können. Sie sollen eine Transparenz der Prozesse und Zusammenhänge ermöglichen. Ziel ist es dabei auf die dynamische Umgebung im täglichen Geschäft reagierten zu können [RSA13, S. 3f]. Vorteile ergeben sich nicht nur für das Management. Beispielsweise kann im Customer-Relationship-Management durch die Daten aus dem BI eine Kundensegmentierung erreicht werden. Mit einer Churn-Analyse kann so versucht werden, Kunden mit hoher Kündigungswahrscheinlichkeit zu identifizieren [KBM10, S. 88]. Durch geeignete Maßnahmen kann dann eine Kündigung vermieden werden.

2.1.4 Systemaufbau

Nach Kemper/Baars [KBM10, S. 11] lassen sich BI-Systeme in drei Schichten einteilen. Innerhalb der Schichten, des so vorgegebenen Ordnungsrahmens, findet eine unternehmensspezifische Ausgestaltung statt. Die unterste Schicht bildet die Datenbereitstellung. Im klassischen Sinne sind dies ein oder mehrere unterschiedliche Datawarehouses [KBM10, S. 11]. Eine zweite Schicht dient zur Informationsgenerierung und -distribution. Zu dieser gehören Analyseanwendungen aller Art [KBM10, S. 12f]. Darauf aufbauend befinden sich in der oberen Schicht Anwendungen oder Plattformen zum Informationszugriff. Diese werden auch Portale genannt [KBM10, S. 12]. Sie bieten eine personalisierte Darstellung über alle verfügbaren Informationen [KBM10, S. 152].

2.1.5 Datenquellen

Die Daten selbst werden von außen in das BI-System getragen. Dies geschieht in die unterste Schicht, der Datenbereitstellung. Als Datenquelle kommen dabei, neben externen Quellen über die Märkte, viele unternehmensinterne Systeme in Betracht. Zum einen sind CRM- und ERP-System für einen Datenzugriff geeignet. Des Weiteren können auch den Produktionssteuerungs- und -planungssysteme Daten für BI entnommen werden [KBM10, S. 11]. BI ist so als integrierten, unternehmensspezifischen Gesamtansatz zu verstehen [KBM10, S. 8].

2.1.6 Aktuelle Herausforderungen

Aktuell kann im Unternehmen das volle Spektrum an vorhandenen Daten für BI genutzt werden [RSA13, S. 269]. Dies führt zu immer größeren Datenbeständen. Gleichzeitig werden Auswertungen immer umfassender und tiefer gehend. Die Verarbeitung dieser „Big Data“, wie sie zusammenfassend beschrieben werden, erfordert neue Maßnahmen [RSA13, S. 15]. Als Möglichkeit diese Datenmenge zu verarbeiten, wird aktuell die parallele Berechnung von Ergebnissen betrachtet. Beginnend bei der Verwendung mehrere CPUs, über die Einbeziehung von Grafik- und Spezialprozessoren, bis hin zur Verteilung über mehreren Maschinen. Die Möglichkeiten der extremen Parallelisierung können die steigende Arbeitslast bewältigen [SM12, S. 125f]. Die Standorte der Komponenten, an denen die Verarbeitung durchgeführt wird, sind in vielen Fällen nicht mehr von Bedeutung. Dies führt dazu, dass auch BI auf Plattformen, die unter dem Begriff „Cloud Computing“ zusammengefasst werden, betrieben werden können [KBM10, S. 252].

2.2 Google App Engine

App Engine ist eine Plattform zum Betrieb von Webanwendungen [San12, S. 1]. Google gibt mit diesem Dienst die Möglichkeit eigene Webanwendungen auf der Infrastruktur von Google auszuführen. Um welchen Typ von Webanwendung es sich dabei handelt bleibt offen. So kann es sich um eine einfache statische Webseite, ein komplexes Shopsysteme oder ein Onlinespiel

handeln [San12, S. 1]. Ein BI-System mit einer Webschnittstelle zur Kommunikation und dem Datenaustausch erscheint somit ebenfalls möglich.

Dan Sanderson beschreibt App Engine in drei Teilen: Anwendungsinstanzen (1), skalierbare Speicher (2) und skalierbare Dienste (3) [San12, S. 1]. Eine Einteilung in diese Abschnitte erscheint auch für einen allgemeinen Überblick sinnvoll.

(1) **Instanzen:** Um eine Webanfrage bearbeiten zu können, wird ein sogenannter Request-Handler benötigt. Dieser bearbeitet die Anfrage und gibt eine Antwort zurück. Üblicherweise wird dieser im Zeitpunkt des Eintreffens einer Anfrage erzeugt. Nach Rückgabe der Antwort wird er wieder aufgelöst [San12, S. 118]. Da die Erzeugung eines Request-Handlers Zeit und Ressourcen benötigt, gibt es in App Engine Instanzen. Dies sind langlebige Container, welche die Request-Handler umgeben. Eine Instanz kann mehrere Request-Handler beinhalten [San12, S. 119f]. In Abbildung 1 wird beispielhaft der Aufbau einer Instanz mit drei Request-Handler dargestellt. Durch das Deaktivieren des Multithreading-Modus kann die Anzahl der Request-Handler in einer Instanz auf eins limitiert werden. Da für eine stark frequentierte Anwendung eine Instanz nicht ausreichend sein kann, können mehrere Instanzen parallel betrieben werden [San12, S. 119f].

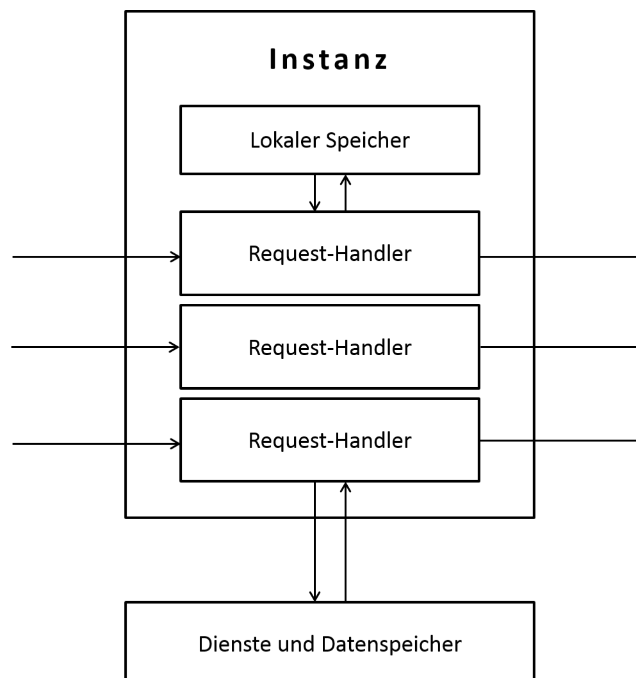

Abbildung 1: Instanz mit mehreren Request-Handlern [San12, S. 121]

(2) **Datenspeicher:** Mit dem *Datastore* stellt App Engine ein Datenspeicher zur Verfügung. Der Aufbau ist dabei nicht mit einer relationalen Datenbank zu vergleichen. Im *Datastore* unterliegen die Daten keinem festen Schema [San12, S. 5, 129]. Auch SQL als Abfragesprache wird nicht unterstützt. Da der *Datastore* als Datenspeicher eines BI-Systems eine essentielle Rolle einnimmt, wird auf diesen in einem separaten Unterkapitel genauer eingegangen.

(3) **Dienste:** In der App Engine Umgebung sind nicht alle Funktion, die bei einem herkömmlichen Webserverbetrieb möglich sind, verfügbar. Wird eine Anwendung in Java

entwickelt, so werden einige Klassen[3] auf App Engine nicht angeboten. Dies hat beispielsweise zur Folge, dass es aus Java heraus nicht möglich ist andere Webseiten als Client abzurufen. App Engine stellt diese Funktion über ein Dienst zur Nutzung bereit [San12, S. 339].

Es gibt weitere solcher Dienste, die für eine Nutzung in skalierbarer Weise zur Verfügung gestellt werden. Im engeren Sinne gehört auch der *Datastore* zu diesen. Neben einem E-Mail-Dienst wird auch ein Datenspeicher für große Dateien angeboten. Dieser ist für das Speichern und Bereitstellen von, in der Größe unbeschränkten, Dateien zuständig [San12, S. 307].

Als Zwischenspeicher kann der *Memcache* genutzt werden. Durch schnellere Zugriffszeiten eignet er sich besonders gut als Cache für den *Datastore* [San12, S. 302]. Die Organisation findet dabei als Key-Value-Store statt. Werte können dabei bis zu einem Megabyte groß sein. Bei den Schlüsseln gibt es keine Größenbeschränkung. Sind diese jedoch größer als 250 Kilobyte wird lediglich ein Hashwert verwendet [San12, S. 292]. Beim *Memcache* können Daten auch asynchron gelesen oder geschrieben werden [San12, S. 291f]. Eine fest definierte Größenbeschränkung gibt es nicht. App Engine verwaltet den Speicher selbst. So kann es vorkommen, dass alte oder wenig gelesene Daten bei Platzmangel ohne Rückfragen gelöscht werden. Allgemein wird keine Garantie über die Dauer des Vorhandenseins von Werten im *Memcache* gegeben [San12, S. 293].

Mit dem Dienst *Task Queue* können Aufgaben für eine spätere Bearbeitung in eine Warteschlange eingestellt werden. Es gibt zwei unterschiedliche Arten von Warteschlangen. Bei Push-Warteschlangen wird der Anwendung eine Aufgabe von App Engine übergeben. Über Pull-Warteschlangen kann der Zeitpunkt der Verarbeitung von der Anwendung bestimmt werden. Diese holt sich eine Aufgabe aus der Warteschlange ab [San12, S. 392]. Die First-In-First-Out-Reihenfolge wird dabei nicht garantiert [San12, S. 404]. Für eine Warteschlange stehen verschiedene Konfigurationsmöglichkeiten bereit. So kann die Größe und die Durchflussrate festgelegt werden [San12, S. 394]. Den Aufgaben können ebenfalls Einstellungen mitgegeben werden. Es ist möglich eine Aufgabe für eine bestimmte Dauer oder bis zu einem bestimmten Zeitpunkt in der Warteschlange festzuhalten. Eine Verarbeitung vor dem definierten Zeitpunkt ist dann nicht möglich [San12, S. 401].

2.2.1 Skalierbarkeit

Die Skalierbarkeit wird durch das Starten und Beenden von Instanzen erreicht. Sollten die bestehenden Instanzen ausgelastet sein, wird eine neue Instanz gestartet. Im einfachen Fall ohne Multithreading ist dies der Fall, wenn jede Instanz bereits einen Request-Handler beinhaltet [San12, S. 122]. Gibt es einige Instanzen, die längere Zeit ungenutzt sind, werden diese beendet [San12, S. 119]. Ab wann eine Instanz ausgelastet ist, unterliegt der Konfiguration. Es ist möglich eine Zeit festzulegen, wann mit der Verarbeitung einer neuen Webanfrage begonnen werden muss. Wird diese Zeit erhöht haben bestehende Instanzen mehr Zeit die Anfrage entgegenzunehmen, bevor eine neue Instanz gestartet wird [San12, S. 122]. Das Beenden ungenutzter Instanzen lässt sich ebenfalls einstellen. So kann festgelegt werden, wie viele ungenutzte Instanzen maximal ausgeführt werden dürfen [San12, S. 125]. Eine weitere Konfigurationsmöglichkeit, die eine schnelle Hochskalierung ermöglicht, liegt in der Anzahl von ungenutzten Instanzen, die mindestens ausgeführt werden. Da der Start einer Instanz Zeit benötigt, kann die Mindestanzahl an laufenden Instanzen auf eins oder mehr festgelegt werden. Es stehen somit immer

[3] https://developers.google.com/appengine/docs/java/jrewhitelist

Instanzen mit freien Ressourcen zur Verfügung [San12, S. 124]. Die Größe von Instanzen lässt sich ebenfalls einstellen. Damit können im Allgemeinen mehr Request-Handler in einer Instanz ausgeführt werden. Die Größe der Instanzen wird nicht automatisch angepasst und ist für alle Instanzen gleich [San12, S. 127].

Bei den Instanzen stößt die Skalierung theoretisch an keine Grenzen. Dies ist bei den Diensten von Google App Engine nicht uneingeschränkt zu behaupten. Auch ohne Berücksichtigung der Kosten sind gewisse Einschränkungen bei der Skalierung vorhanden. Alle Kontingente listet Google auf einer Übersichtsseite[4] auf. Dort sind diese als Tages- oder Minutenbegrenzungen angegeben. Laut Dan Sanderson werden diese von Zeit zu Zeit, an die sich ändernden technischen Gegebenheiten, angepasst [San12, S. 110]. Um nur einige Werte zur Orientierung zu nennen: Aktuell liegt das Limit der *Task Queue* bei einer Milliarde API-Aufrufen pro Tag. Die Begrenzung für den gesamten ausgehenden Datenverkehr der Anwendung, sowohl für App Engine Dienste als auch für die Antwort von Webanfragen, liegt bei rund 14 Terabyte pro Tag und zehn Gigabyte pro Minute. Für kleinere und mittlere Anwendungen sind diese Begrenzungen mehr als ausreichend. Für viel genutzte Anwendungen mit starker Auslastung eines Dienstes sollten die Nutzungsgrenzen jedoch nicht ohne Beachtung bleiben.

Eine weitere Einschränkung gibt App Engine mit der maximalen Bearbeitungsdauer von Webanfragen vor. Wird die Bearbeitung nicht innerhalb von einer Minute abgeschlossen, wird diese abgebrochen und eine Fehlermeldung zurückgegeben [San12, S. 110].

2.2.2 Datastore

Der *Datastore* ist eine Datenbank in dem Objekte abgelegt werden können. Diese Objekte können abgelegt, gelesen sowie mittels Datastore-Abfragen gefiltert geladen werden. Für Datastore-Abfragen werden im *Datastore* verschiedene Indizes geführt.

Objekte: Ein Objekt (in App Engine als „*entity*“ bezeichnet) besteht aus genau einem Schlüssel und mehreren Attributen. Ein Objekt muss keine Attribute besitzen. Die maximale Anzahl an Attributen eines Objekts wird ausschließlich durch die Gesamtgröße des Objekts beschränkt [San12, S. 130]. Die Objektgröße darf dabei ein Megabyte nicht überschreiten [San12, S. 111]. Jedes Objekt ist zudem von einem bestimmten Typ („*kind*“) [San12, S. 130].

Schlüssel: Jeder Schlüssel („*key*“) muss innerhalb des Objekttyps eindeutig sein. Ist für ein Objekt ein Schlüssel vergeben kann dieser nicht mehr geändert werden [San12, S. 131].

Attribute: Ein Attribut („*property*“) kann durch seinen Namen eindeutig identifiziert werden. Neben dem Namen hat ein Attribut mindestens ein Attributwert [San12, S. 131]. Es sind auch mehrere Werte für ein Attribut erlaubt [San12, S. 140]. Als Attributwerte können die meisten üblichen Datentypen, wie zum Beispiel Boolean, Ganz- oder Fließkommazahlen, Text und Datum, verwendet werden. Je nach Datentyp ist es möglich ein Attribut indizieren zu lassen [San12, S. 137f].

Datastore-Abfragen: Eine Datastore-Abfrage („*query*“) ermöglicht den Zugriff auf Objekten, welche geforderte Bedingungen erfüllen. Vergleiche können dabei zum Beispiel über Operatoren wie gleich, größer, größer gleich, kleiner, kleiner gleich oder ungleich durchgeführt werden. Jede Datastore-Abfrage in App Engine benötigt dabei einen Index [San12, S. 151].

[4] https://developers.google.com/appengine/docs/quotas

Datastore-Abfragen werden nicht auf dem gesamten Datenbestand ausgeführt, sondern sucht die entsprechenden Schlüssel aus dem Index. Anschließend werden die Objekte anhand des Schlüssels geladen und zurück gegeben [San12, S. 151]. Eine Datastore-Abfrage nur nach Schlüssel ist ebenfalls möglich. Als Ergebnis werden dann nicht die Objekte, sondern die Schlüssel zurückgeben [San12, S. 161].

Indizes: Eine Datastore-Abfrage kann nur auf einem Index („*index*“) ausgeführt werden. Der Index schreibt die Bedingungen, die in der Datastore-Abfrage gestellt werden können, vor. Es gibt verschiedene Arten von Indizes. Für jeden Objekttyp werden die Indizes separat geführt. Ein Index kann sich als Liste mit mindestens einer Spalte vorgestellt werden [San12, S. 161].

Die einfachste Art eines Indexes beinhaltet die Schlüssel von Objekten des gleichen Objekttyps. Diese werden beim Speichern eines neuen Objekttyps erstellt und fortlaufend aktuell gehalten. Die Schlüssel sind im Index aufsteigend sortiert [San12, S. 169].

Weiterer Indizes listen die Attributwerte. Indizierte Attribute mit gleichem Namen werden gemeinsam in einen Index aufgenommen. Enthalten ist dabei der Schlüssel des Objekts sowie der Attributwert [San12, S. 169]. Diese Indizes ermöglichen Datastore-Abfragen mit Filter nach einem Attribut. Da das Ergebnis auch in absteigender Reihenfolge erwünscht sein kann, gibt es zwei Indizes. Diese sind in auf- und absteigender Reihenfolge nach dem Attributwert sortiert.[San12, S. 168]

Es sind auch benutzerdefinierte Indizes möglich. Damit werden Datastore-Abfragen mit Bedingungen über mehr als einem Attribut durchgeführt. Ebenfalls können so Datastore-Abfragen mit mehreren priorisierten Sortierungen erfolgen. Die Erstellung eines solchen Indexes muss bei der Entwicklung der Anwendung angegeben werden [San12, S. 175].

Technisch bedingt sind bei Datastore-Abfragen nicht alle Filter- und Sortierkombinationen möglich. So ist die Anzahl der Attribute, die nicht auf Gleichheit geprüft werden, auf eins beschränkt. Weitere Filter über andere Attribute können nur über Gleichheit definiert werden [San12, S. 179].

Da im *Datastore* von App Engine kein festes Schema vorgeben ist, können verschiedene Objekte des gleichen Objekttyps unterschiedliche Attribute haben. Daraus ergeben sich einige Besonderheiten. So kann es zum Vergleich zweier Attribute kommen, die als Wert unterschiedliche Datentypen besitzen. In diesem Fall greift App Engine auf eine vordefinierte Reihenfolge der Datentypen zurück [San12, S. 184]. Es ist auch möglich, dass ein Objekt ein Attribut nicht besitzt oder es nicht indiziert ist. Da Datastore-Abfragen nur auf den Indexeinträgen ausgewertet werden, sind diese Objekte nicht im Ergebnis enthalten [San12, S. 182]. Durch mehrere Werte eines Attributes wird das gleiche Objekt im zugehörigen Index mehrfach gelistet. Die Ergebnismenge beinhaltet ein Objekt, sobald ein Attributwert über mindestens einen Filter erfasst wird [San12, S. 188f].

2.2.3 Kostenmodell

Für jede Anwendungen steht ein kostenfreies Kontingent an Ressourcen zur Verfügung. Beim Überschreiten der Freigrenzen fallen Kosten an. Zu bezahlen sind dabei nur die Ressourcen, die in Anspruch genommen wurden. Um unerwartet hohe Kosten zu vermeiden, kann ein tägliches Limit hinterlegt werden, welches die Kosten deckelt. Bei verbrauchtem Limit werden die jeweiligen Ressourcen abgeschaltet [San12, S. 112f].

Bei Instanzen werden die Kosten pro Instanzstunde abgerechnet. Eine Instanz, der kleinsten

Klasse, kostet für eine Stunde aktuell $0,08[5].

Die Kosten für den *Datastore* hängen von den API-Zugriffen ab. Jeder API-Zugriff benötigen unterschiedlich viele Datenoperationen. Es gibt drei Arten von Datenoperationen. Für diese fallen unterschiedliche Kosten an. Die Tabelle in Abbildung 2 listet die Kosten für die Datenoperationen auf. Aus der Tabelle in Abbildung 3 lassen sich die Anzahl der benötigten Datenoperationen je API-Aufruf ablesen.

Schreib-Operationen	$0,09 pro 100.000 Operationen
Lese-Operationen	$0,06 pro 100.000 Operationen
Kleine Operationen	$0,01 pro 100.000 Operationen

Abbildung 2: Kosten der Datenoperationen[5]

Objekt abrufen	1 Lese-Operation
Neues Objekt speichern	2 Schreib-Operationen + 2 Schreib-Operationen pro indiziertem Attribut + 1 Schreib-Operation pro benutzerdefiniertem Index
Datastore-Abfrage	1 Lese-Operation + 1 Lese-Operation pro Objekt
Datastore-Abfrage (nur Schlüssel)	1 Lese-Operation + 1 kleine Operation pro Schlüssel

Abbildung 3: Datenoperationen je API-Zugriff[5]

Zusätzlich fallen beim *Datastore* Entgelte für die Menge der gespeicherten Daten an. Pro Monat wird für jeden Gigabyte an Daten $0,18[5] berechnet.

2.3 Highcharts

Highcharts[6] ist eine in HTML5 und JavaScript geschriebene Grafikbibliothek. Sie bietet die Möglichkeit interaktive Diagramme in Webseiten einzubinden. Diagramme können in alle üblichen Diagrammtypen gezeichnet werden. Zur Visualisierung von Daten müssen diese zusammen mit den Darstellungsparameter in eine Grafikkonfiguration eingebunden werden. Diese muss Highcharts in JSON[7] zur Verfügung gestellt werden.

[5] https://developers.google.com/appengine/docs/billing
[6] http://www.highcharts.com
[7] JavaScript Object Notation; http://json.org

3 Entwurf

Vor der Konzeption eines BI-Systems muss sich die Frage gestellt werden, was mit dem System erreicht werden will. Der Rahmen von BI lässt hierbei eine weite Auslegung zu. Das System, welches im Rahmen dieser Arbeit entwickelt wird, setzt den Schwerpunkt auf die Analyse und Auswertung des Verhaltens der Nutzer. Kurz gesagt, ein BI-System zur Beantwortung der zentralen Fragestellung: Was wird von wem wann gemacht?

3.1 Anforderungsanalyse

Zur Beantwortung der eben aufgeworfen Frage werden detaillierte Informationen über die Nutzung der Anwendung benötigt. Diese können an verschiedenen Stellen in der Anwendung erhoben werden. Vorzugsweise werden für diesen Zweck wichtige Aufrufe oder Eingaben der Nutzer erfasst. Daraus ergibt sich ein ereignisorientiertes Protokollsystem. Die Ereignisse sollten dabei alle Daten enthalten, die für eine Auswertung von Relevanz sind. Welche Daten dies sind, kann von Fall zu Fall verschieden sein. Die Ereignisse sollten daher einen flexiblen Aufbau haben, um den verschiedensten Anwendungen gerecht werden zu können.

Um eine effiziente Verarbeitung zu gewährleisten, wird dennoch eine Struktur der Ereignisse benötigt. Durch Attribute können Ereignisse bei der Verarbeitung gefiltert und verglichen werden. Attribute setzen sich aus einem eindeutigen Schlüssel und genau einem Wert zusammen. Der Inhalt von Ereignissen wird dadurch flexibel gehalten.

Attribute, die bei jedem Ereignis vorhanden sein müssen, erleichtern ebenfalls die Auswertung. Da jedes Ereignis einen Zeitpunkt hat, an dem es ausgelöst wurde, erscheint ein *Zeitstempel* sinnvoll. Auch die Aktion, die zum Auslösen des Ereignisses geführt hat, kann als eindeutig definierte *Aktions-ID* zu jedem Ereignis hinzugenommen werden. Für eine Auswertung ist die Zuordnung eines Ereignisses zu einem bestimmten Nutzer sehr hilfreich. Für Anwendungen mit einer Wiedererkennung der Nutzer, zum Beispiel durch ein Login, ist es vorteilhaft eine eindeutige *Nutzer-ID* in den Ereignissen zu hinterlegen. Bei Fällen, in denen ein Ereignis keinem Nutzer zuzuordnen ist, kann ein vordefinierter Standardwert eingesetzt werden.

Ein Ereignis kann zu seinen obligatorischen Attributen, *Zeitstempel*, *Aktions-ID* und *Nutzer-ID* auch noch anwendungsspezifische Werte haben. Diese werden dann als Schlüssel-Werte-Paar im Ereignis erfasst. Abbildung 4 zeigt eine schematische Darstellung eines Ereignisses.

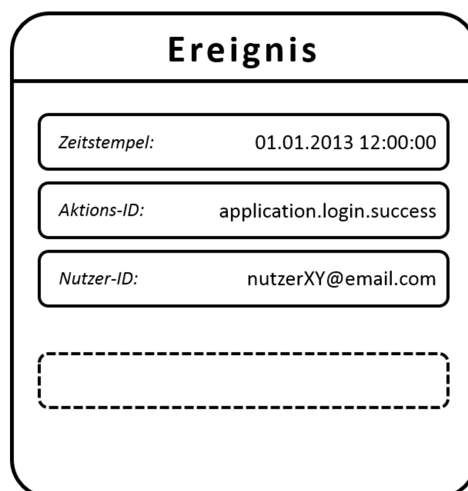

Abbildung 4: Ereignis mit Attributen

Für eine Auswertung, also eine konkrete Fragestellung, werden im Allgemeinen nicht alle gespeicherten Ereignisse benötigt. Es werden nur Ereignisse, die bestimmte Kriterien erfüllen, herangezogen. Für ein effizientes System ist es wichtig, dass es dabei möglichst keine Daten abruft, die für die Auswertung nicht benötigt werden. Dies bedeutet im Idealfall, dass mit präzisen Abfragen ausschließlich die angeforderten Ereignisse aus dem Datenspeicher geladen werden.

Es gibt eine Vielzahl verschiedene Abfragemöglichkeiten, die unterschiedlich häufig ausgeführt werden. Eine schnellere Abarbeitung einiger Abfragearten kann durch optimieren der Datenstruktur auf diese erreicht werden. Dafür ist es wichtig zu wissen, wie die häufigsten Abfragen aufgebaut sind.

Die Abfrage nach allen Ereignissen einer gewissen Aktion wird einen hohen Anteil der Abfragen ausmachen. Insbesondere durch das Verknüpfen der Daten mehrerer Abfragen dieser Art lassen sich detaillierte Auswertungen erstellen. Die zusätzliche Einschränkung des Zeitraumes kann in allen Abfragen angenommen werden. Sollte keine zeitliche Einschränkung gewünscht sein, kann der Zeitraum in der Abfrage so groß gewählt werden, dass er über den Datenbestand hinausgeht. Damit tritt keine Filterung ein.

Ereignisse, die in einem anwendungsspezifischen Attribut einen bestimmten Wert besitzen, könnten ebenfalls durch Abfragen gesucht werden. Für den Fall, dass nur Ereignisse einer bestimmten Aktion diese Attribute besitzen, kann sie durch eine in der *Aktions-ID* einschränkenden Abfrage ersetzt werden.

Im BI sind Abfragen, die sich auf alle Ereignisse eines einzelnen Nutzers beziehen, selten. Im Kontext der Ziele von BI enthalten Abfragen dieser Art keine relevanten Informationen. Bei der Datengrundlage eines Benutzers kann nicht sichergestellt werden, dass die Aussage repräsentativ das allgemeine Nutzerverhalten widerspiegelt. Für Support oder Kundenbetreuung wären Abfragen dieser Art eventuell sinnvoll, diese liegen jedoch außerhalb des Aufgabenbereichs von BI. Zumeist gibt es für diese Bedürfnisse eigene Systeme.

3.1.1 Maßstäbe

Als Ziel wurde eine hohe Skalierbarkeit festgehalten. Die Funktionen einer skalierbaren Software sollen sich, bei Veränderung der äußeren Bedingungen, nicht verschlechtern [Sdc]. Dies bedeutet, dass der Betrieb des BI-Systems mit allen Funktionalitäten sowohl für geringes Datenaufkommen als auch bei großem Datenaufkommen möglich sein soll.

Um die Leistungsfähigkeit eines Systems bestimmen zu können, müssen geeignete Maßstäbe gefunden werden. Bei einem BI-System ist es sinnvoll, dies mit einer Abhängigkeit zur verarbeitenden Datenmenge zu definieren. Eine Verarbeitung wird durch zwei relevante Funktionen eingeleitet. Zum einen ist dies ein eingehendes Ereignis, zum anderen eine eingehende Abfrage. Weitere Verarbeitungsfunktionen, wie zum Beispiel das Vorberechnen von Daten, dienen nur zur deren Unterstützung und müssen deshalb nicht selbst als Maß für die Leistungsfähigkeit definiert werden.

Für ein Maß, welches sich an der Anzahl der eingehenden Ereignissen orientiert, ist wichtig, dass es in Abhängigkeit zur Zeit steht. Um eine Million Ereignisse in einem Monat zu verarbeiten wird ein deutlich kleineres System benötigt als die gleiche Anzahl an einem Tag. Damit auch mögliche Lastspitzen Berücksichtigung finden, sollte der zeitliche Betrachtungshorizont möglichst klein sein. Um eventuelle Puffermechanismen auszuschließen, ist eine Mindestgröße sinnvoll. Ein geeigneter Zeitraum wäre somit eine Minute.

Der eben definierte Maßstab erlaubt eine Aussage über die Anzahl von Funktionsaufrufen pro Zeiteinheit. Er erlaubt eine Beurteilung der Skalierung bezüglich der Datenrate. Im zweiten Kriterium wird das Datenaufkommen bei Abfragen berücksichtigt. Ein Maßstab, der die Da-

tenrate nicht noch einmal berücksichtigt, lässt eine unabhängigere Bewertung zu. Ermöglicht wird dies durch die Betrachtung der Anzahl der Ereignisse, die als Ergebnis einer Abfrage zurückgegeben werden. Dieser Maßstab erlaubt eine Beurteilung über die Skalierung bezüglich der Datenmenge.

Durch diese beiden Maßstäbe können die Leistungsfähigkeit und damit auch das Mindestmaß an Skalierbarkeit eines BI-Systems zum Ausdruck gebracht werden. Die beiden wichtigsten Funktionen, das Entgegennehmen von Ereignissen und das Verarbeiten von Abfragen, sind dabei berücksichtigt. Ebenso werden die Skalierbarkeit bezüglich Datenrate und Datenmenge abgebildet.

3.1.2 Anforderungen

Die Anforderungen an das System können mit Hilfe der oben bestimmten Maßstäbe festgelegt werden.

Die Anzahl der eingehenden Ereignisse pro Minute kann über die Anzahl der Ereignisse eines Tages abgeschätzt werden. Diese wiederum kann als Produkt der Anzahl der täglichen Nutzer und der Anzahl der Ereignisse eines Nutzers pro Nutzungstag gesehen werden. Für die Abschätzung der Anforderungen nehmen wir die Webanwendung Mindmeister[8] als Referenz. Diese ermöglicht es im Browser Mindmaps zu erstellen und zu verwalten. Auf der Social-Media-Plattform Facebook hat Mindmeister[9] aktuell etwa 9.600 Fans. Es kann davon ausgegangen werden, dass nicht jeder Fan bei Mindmeister täglich aktiv ist. Dafür gibt es andere Anwender, die die Anwendung gelegentlich nutzen, ohne als Fan eingetragen zu sein. Für die weiteren Abschätzungen kann deshalb von 10.000 täglichen Nutzer ausgegangen werden. Die Anzahl der Ereignisse, die ein Nutzer am Tag auslöst, kann auf 100 abgeschätzt werden. Sollte eine Anwendung den Nutzer längere Zeit oder häufiger über einen Tag an die Benutzung binden können, wären auch mehr Ereignisse denkbar. In diesem Fall sollte der Detailgrad der Ereignisgenerierung angepasst werden. Die Annahme ist also, dass der Detailgrad der Ereignisgenerierung auf durchschnittlich 100 Ereignisse pro Nutzer und Tag angepasst werden kann. Beispielsweise können in Mindmeister Informationen über die Größe einer Mindmap beim Speichern und nicht beim Hinzufügen oder Löschen eines Knotens erfasst werden. Als Anforderung an die Datenrate ergibt sich daraus eine Million Ereignisse pro Tag. Verteilen sich diese über acht Stunden, ergibt dies 2.083 Ereignisse pro Minute. Da eine Gleichverteilung nicht anzunehmen ist, wird dieser Wert um 50 Prozent auf 3.125 Ereignisse pro Minute erhöht.

Bei der Anzahl der Ereignisse, die in einer Abfrage verarbeitet werden sollen, kann das Webanalysetool Google Analytics als Orientierung dienen. In der kostenfreien Nutzung liegt das Limit bei 500.000 Ereignissen. Als zahlender Kunde sind drei Millionen Ereignisse möglich[10]. Abfragen die mehr Ereignisse einbeziehen würden, werden nicht auf dem vollen Datenbestand sondern auf einer Stichprobe ausgewertet. Ziel ist es, ebenfalls drei Millionen Ereignisse in einer Abfrage einzulesen und auszuwerten.

3.2 Gesamtsystem

Um Ereignisse ins BI-System übertragen zu können, muss eine entsprechende Schnittstelle zur Verfügung stehen. Eine einfache und universelle Kommunikation erfolgt in Google App Engine

[8] http://www.mindmeister.com
[9] http://www.facebook.com/mindmeister
[10] http://www.google.com/intl/de_ALL/analytics/premium/features.html

über HTTP-Zugriffe. Die Webschnittstelle muss eingehende Ereignisse entgegennehmen und in das BI-System übertragen. Wünschenswert ist auch, dass eine Anwendung, die ebenfalls auf App Engine betrieben wird, Ereignisse nicht über die Webschnittstelle übertragen muss. Denkbar wäre eine BI-Anwendung, die direkt in die Zielanwendung eingebunden wird und Ereignisse im Programmcode erzeugen und übergeben kann. Eine Logik muss entscheiden wie eingehende Ereignisse weiterverarbeitet werden. Eine Pufferung der Ereignisse mit regelmäßiger Weiterverarbeitung oder eine sofortige dauerhafte Speicherung kann möglich sein.

Bei einer Abfrage von Ereignissen erfolgt die Verarbeitung ähnlich. Über Webaufrufe werden Abfragen gestellt und an das BI-System übergeben. Eine hohe Flexibilität ermöglicht Freiheiten bei der Art und Struktur der Abfragen. Zugleich wird so die Möglichkeit zur Erweiterung geboten. Im System werden die gesuchten Ereignisse geladen. Mechanismen zum schnelleren Zugriff auf die Daten versuchen den Ladevorgang der Ereignisse zu beschleunigen. Es bietet sich an, auf vorbereitete Teilergebnisse oder auf einen Zwischenspeicher zuzugreifen. Das Ergebnis, die ausgewählten Ereignisse, werden als Antwort der Webanfrage zurückgegeben.

3.2.1 Aufbau

Aus dem exemplarischen Ablauf der Nutzung kann der Aufbau des Gesamtsystems ableitet werden. Die Strukturen und Interaktionen der einzelnen Komponenten lassen sich in einem Diagramm (Abbildung 5) darstellen.

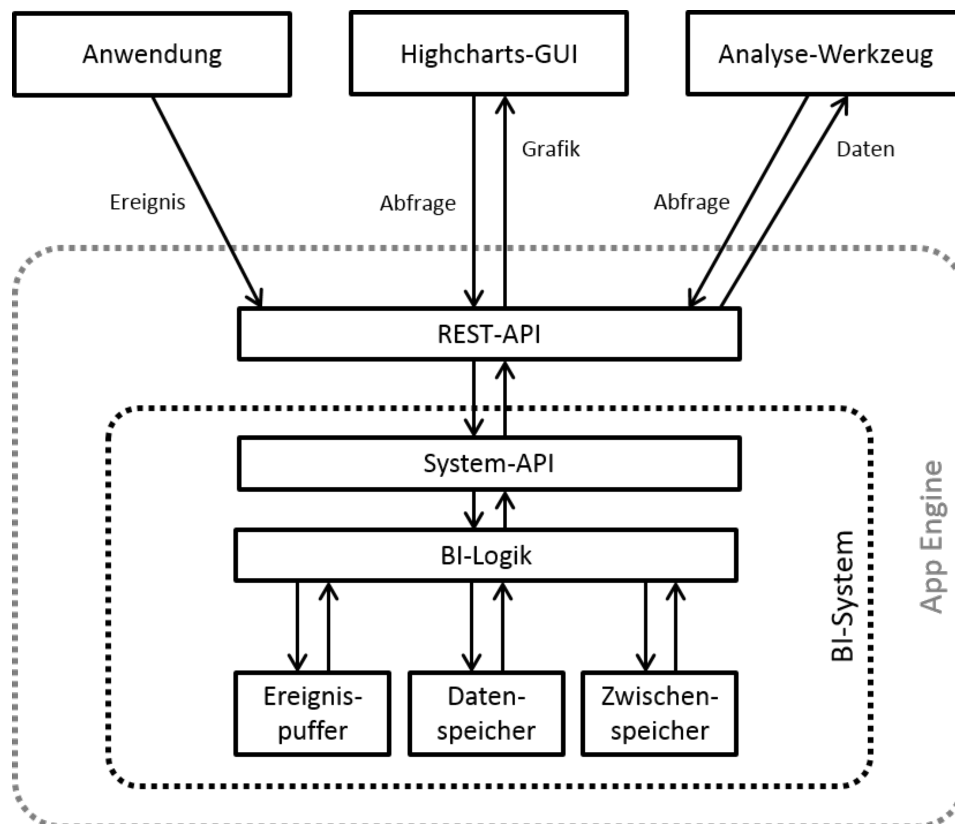

Abbildung 5: Schema des Systemaufbaus

Auf oberster Ebene befinden sich Anwendungen und grafische Oberflächen zur Darstellungen oder Weiterverarbeitung der Ergebnisse. Darunter eine zustandslose Schnittstelle zur Verarbeitung von Webanfragen. Diese kann auch von zeitlich gesteuerten Verwaltungsanfragen aus App Engine benutzt werden. Durch Verwaltungsanfragen kann beispielsweise eine Routine zur

Weiterverarbeitung gepufferte Ereignisse angestoßen werden. Das BI-System wird über eine dafür vorgesehene Schnittstelle angesprochen. Für Anwendungen, die ebenfalls in Google App Engine betrieben werden, ist ein direkter Zugriff möglich. An die Logik sind Ereignispuffer, Zwischenspeicher und persistenter Datenspeicher angebunden.

3.2.2 Datenspeicher

Für die persistente Haltung von strukturierten Daten steht in Google App Engine der *Datastore* zur Verfügung. Dieser wird genutzt, um die Ereignisse speichern zu können. Für eine hohe Flexibilität bei Anzahl und Größe der Attribute beinhaltet ein Objekt des *Datastore* genau ein Ereignis.

Durch diesen Aufbau müssen bei umfangreichen Abfragen viele Objekte geladen werden. Es ist zu erwarten, dass die in den Anforderungen festgelegten Ziele mit diesem Aufbau nicht erreichbar sind. Deshalb wird zusätzliche eine Gruppierung der Ereignisse vorgenommen.

3.3 Gruppierung von Ereignissen

Für einen schnelleren Ladevorgang der Ereignisse aus dem *Datastore*, können Ereignisse gruppiert und gemeinsam als Objekt abgelegt werden. Der Vorteil entsteht durch eine Komprimierung der Daten. Es müssen nicht mehr alle Attribute für alle Ereignisse einzeln gespeichert werden. Wenige, große Objekte können dann schnell aus dem *Datastore* geladen werden.

Um möglichst viele Ereignisse in einem Objekt ablegen zu können, werden die anwendungsspezifischen Attribute beim Gruppieren der Ereignisse verworfen. Durch die gruppierte Speicherung der Ereignisse wird eine Verbesserung der durchschnittlichen Lesegeschwindigkeit pro Ereignis um ein Vielfaches erwartet.

3.3.1 Konzept

Die Vorteile der Gruppierung zeigen sich besonders, wenn Ereignisse oft zusammen gelesen werden. Bei Abfragen, mit der Filterung nach dem Wert eines bestimmten Attributes, ist dies der Fall. Wie schon in den Anforderungen (Kapitel 3.1) erläutert, wird bei Abfragen häufig eine *Aktions-ID* angegeben, nach der gefiltert werden soll.

Ein erster Ansatz ergibt sich also daraus, Ereignisse mit der gleichen *Aktions-ID* gruppiert abzuspeichern. Da eine Abfrage auch immer mit einer zeitlichen Einschränkung verbunden ist, sollte dies bei der Gruppierung der Ereignisse Berücksichtigung finden. Diese Gruppierung von Ereignissen, welche die gleiche *Aktions-ID* aufweisen und in einem bestimmten Zeitraum liegen, wird im folgenden Cluster genannt. Beim Zugriff auf ein solches Cluster ist die *Aktions-ID* und der Zeitraum, in dem sich alle Ereignisse befinden, bekannt. Gespeichert werden diese Daten für alle Ereignisse des Clusters gemeinsam. Für die einzelnen Ereignisse müssen zusätzlich die *Nutzer-ID* sowie der genaue *Zeitstempel* festgehalten werden. Mit einer Cluster-basierten Auswertung kann nicht auf anwendungsspezifische Attribute zugegriffen werden. Da diese Form der Datenhaltung als Zusatz eingeführt wird, kann in einem Fall, in dem anwendungsspezifische Attribute benötigt werden, auf vollständige Ereignisse zurückgegriffen werden.

Der Zeitraum, der von einem Cluster abgedeckt wird, ist noch nicht genauer definiert. Er sollte so groß wie möglich sein, jedoch nicht über den Abfragezeitraum hinaus reichen. Allgemeine Aussagen über einen häufigen Abfragezeitraum können nur schwer getroffen werden. Dies erfordert Cluster mit verschieden großen Zeiträumen. Anhand der Länge des Zeitraumes ist eine Einteilung in Ebenen möglich. Jede Ebene besitzt dabei Cluster mit gleich großem Zeitraum.

Die Ereignisse werden redundant auf allen Ebenen gespeichert und können je nach Abfrage von der optimalen Ebenen gelesen werden. In Abbildung 6 wird der Aufbau der Ebenen mit ihren Clustern verdeutlicht.

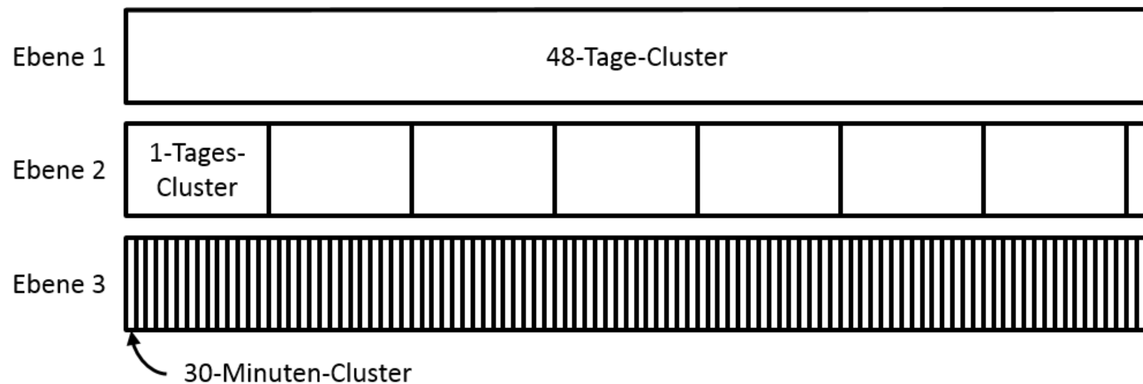

Abbildung 6: Ebenenstrukur

Den *Zeitstempel* der Ereignisse nicht explizit zu speichern bietet einen weiteren Optimierungsansatz. Dieser ist bereits durch den Abdeckungszeitraum des Clusters eingegrenzt. Ein Vorteil ergibt sich daraus erst einmal nur für den verbrauchten Speicherplatz. Für ein Ereignis wird nur noch die *Nutzer-ID* gespeichert. Wenn ein Nutzer in diesem Zeitraum mehrfach ein Ereignis mit der gleichen Aktion ausgelöst hat, kann dieses jetzt zusammengefasst werden. In diesem Fall wird nur die *Nutzer-ID* und die Anzahl der Vorkommen gespeichert. Dadurch wird eine weitere Komprimierung erreicht. Bei vielen Ereignissen mit gleicher *Aktions-ID* und *Nutzer-ID* in diesem Zeitraum können so kleinere Cluster erreicht werden. Dies wirkt sich auf die durchschnittliche Lesegeschwindigkeit pro Ereignis aus.

Diese Optimierung hat den Nachteil, dass Ereignisse nur noch einem Zeitraum zugeordnet werden können. Die Länge der Zeitspanne hängt vom Abdeckungszeitraum des Clusters ab. Es stellt sich die Frage, ob diese Unschärfe bezüglich des Zeitpunktes bei den Ereignissen vertretbar ist. Dies hängt von der jeweiligen Anwendung ab. Allgemein ist es schwer eine Antwort zu begründen. Wie oft der Fall auftritt, dass ein Nutzer eine Aktion mehrmals in einem Zeitraum aufruft, lässt sich ebenfalls nicht pauschal beantworten. Insbesondere da diese Zeiträume auch nur wenige Minuten oder Stunden umfassen können, werden die oben erläuterten Vorteile unter Umständen nur selten Anwendung finden. Um eine breitere Anwendungsbasis zu bieten, wird aktuell auf eine Umsetzung diese Optimierung verzichtet.

3.3.2 Parameter

Die Anzahl der Ebenen ist für den effizienten Zugriff auf Ereignisse von großer Bedeutung. Durch mehrere Ebenen lässt sich der Zeitraum von Abfragen in Teile zerlegen, die auf der jeweiligen Ebene schnell die gesuchten Ereignisse als Ergebnis liefern. Andererseits muss jede Ebene berechnet und gespeichert werden. Für das Zusammenführen der Ereignisse zu den Clustern müssen diese gelesen, verarbeitet und zuletzt wieder in den *Datastore* geschrieben werden. In allen drei Schritten werden Ressourcen in Anspruch genommen, für die Kosten entstehen. Da die Ereignisse als Kopie abgelegt werden, vervielfacht sich für mehrere Ebenen auch der Speicherplatzbedarf.

Für eine effiziente Verarbeitung der Abfragen sollte der Zeitraum eines Clusters möglichst den Anfangs- oder Endzeitpunkt einer Abfrage treffen. Da eine Abfrage oft mehrere Tage abdeckt,

liegt der Beginn des Abfragezeitraumes häufig auf dem Tagesbeginn. Eine Ebene mit Clustern, die alle Ereignisse eines Tages beinhaltet, wäre demnach sinnvoll.

Wie in den Anforderungen erörtert, kann ein Tag bis zu einer Millionen Ereignisse beinhalten. Für Abfragen die sich nicht nur über volle Tage erstrecken, sollte eine Ebene mit einem kürzeren Zeitraum existieren. Ebenfalls für einen zusätzlichen kürzeren Zeitraum spricht auch die Tatsache, dass die Tagesgrenze nur für eine Zeitzone gilt. Es können auch Abfragen mit einem Zeitraum, der sich an den Tagesgrenzen einer anderen Zeitzone orientiert, ausgeführt werden. Dabei entstehen durch die Verschiebung am Anfang und Ende Lücken, die mit der Ein-Tages-Ebene nicht bedient werden können. Da sich die Zeit über die verschiedene Orte der Erde in der Regel mindestens halbstündlich verschiebt[11], können diese mit einer Unterteilung in 48 30-Minuten-Cluster abgedeckt werden.

Bei Abfragen, die über mehrere Monate oder Jahre gehen, werden noch größere Cluster benötigt. Welcher Abeckungszeitraum für diese besonders vorteilhaft sind, kann im Allgemeinen nur schwer abgeschätzt werden. Je nach Anwendungsfall können die Abfragezeiträume bestimmte Zyklen aufweisen, die häufiger als Andere auftreten. In diesem Fall wählen wir mangels weiteren Informationen das gleiche Vielfache wie in den Ebenen unterhalb und definieren ein 48-Tage-Cluster.

3.3.3 Erstellungszeitpunkt

Nachdem die Techniken zur Gruppierung von Ereignissen erläutert sind, stellt sich die Frage, wann es sinnvoll ist, diese Berechnungen durchzuführen.

Eine flexible Lösung ist die Cluster bei der ersten Abfrage zu berechnen. Wird versucht mit einer Abfrage auf Cluster zuzugreifen, die nicht existieren, werden die Daten aus einer darunterliegenden Ebene geladen. Diese Ereignisse könnten nun für die Erstellung des fehlenden Clusters verwendet werden. Diese Variante hat Nachteile bezüglich der Gesamtverarbeitungszeit der Abfrage. Die Abfrage kann nicht optimal auf die Cluster aufgeteilt werden. Es müssen mehrere alternative Cluster einer niedrigeren Ebene verarbeitet werden. Des Weiteren wird zusätzliche Bearbeitungszeit für das Erstellen und Abspeichern der neuen Cluster benötigt.

Sollte die Zeit zur Erstellung der Cluster zu lang sein oder der Zugriff auf Cluster einer niedrigeren Ebene die Verarbeitung verzögern, ist mit diesem Modell auch eine zeitliche gesteuerte Erstellung möglich. Die Cluster werden durch den Abruf der Ereignisse über einen Zeitraum erstellt. Dieser Abruf kann zum Beispiel in periodischen Abständen automatisch ausgelöst werden. Idealerweise wird die Erstellung direkt nachdem alle Ereignisse eines Clusters bekannt sind angestoßen. Dies ist nach Ablauf des Abdeckungszeitraums eines Clusters.

3.4 Persistente Datenhaltung

3.4.1 Faktoren des Datenstrukturentwurfs

Die Bedingungen für einen effizienten Betrieb liegen im Entwurf der Datenstruktur. Aus den Anforderungen und Zielen ergeben sich für ein BI-System drei Faktoren, die beim Entwurf zu beachten sind: Die Betriebskosten (1), der Datendurchsatz (2) und die Abfrageflexibilität (3). Je nach Anwendungsfall sind diese von unterschiedlicher Priorität.

(1) **Betriebskosten:** Die Kosten für die Nutzung des *Datastore* setzen sich aus der Anzahl der Lese- und Schreibzugriffe und aus dem Gesamtdatenvolumen der gespeicherten

[11] Ausnahme: Nepal; vgl: http://www.zeitzonen.org/zeitzonenuebersicht.html

Daten zusammen. Durch redundantes Speichern eines Ereignisses oder durch mehrfaches Speichern von Attributen der Ereignisse können die Anzahl der Schreibzugriffe je nach Datenbankentwurf variieren. Ebenfalls sind die Kosten für einen Schreibzugriff von der Anzahl der im *Datastore* geführten Indizes abhängig. Es ist auch möglich, dass mehr Ereignisse aus dem *Datastore* gelesen werden müssen als ursprünglich angefragt. Diese werden nach dem Lesen wieder verworfen. Kosten für das Lesen dieser Ereignisse fallen dennoch an.

Die Kosten, die beim Betrieb des Systems durch den *Datastore* entstehen, hängen damit vom jeweiligen Datenbankentwurf ab.

(2) **Datendurchsatz:** Der Datendurchsatz lässt sich in den Lesedurchsatz und den Schreibdurchsatz aufteilen. Das Verhältnis der Verarbeitungszeit einer Abfrage zur Anzahl der Ereignisse, die die Abfrage als Ergebnis liefert, kann als Lesedurchsatz beschrieben werden. Je nach Entwurf können mehr oder weniger Ereignisse in einer bestimmten Zeit gelesen werden. Dies hängt unter anderem von der Anzahl und Größe der Objekte sowie von der Art des Zugriffes auf den *Datastore* ab. Da die Verarbeitungszeit einer Abfrage begrenzt ist, hängt hiervon die Anzahl der Ereignisse, die in einer Abfrage gelesen werden können, ab. Der Lesedurchsatz muss nicht für alle Abfragen gleich sein. Daher kann das Bestreben auch sein, für einen Großteil der Abfragen einen hohen Durchsatz zu erreichen.

Ereignisse werden im Allgemeinen einzeln geschrieben. Daher ist die Zeit, die zum Schreiben von Ereignissen benötigt wird, bei einem BI-System primär nicht relevant. Es besteht auch die Möglichkeit das Schreiben in den *Datastore* asynchron vorzunehmen.

(3) **Abfrageflexibilität:** Abfragen können vielfältige Bedingungen an die gesuchten Ereignisse stellen. Welche Freiheiten bei diesen Abfragen ermöglicht werden können, kann vom Entwurf der Datenstruktur abhängen. Insbesondere da die Datastore-Abfragen von App Engine strikte Einschränkungen haben, müssen alle Abfragearten beim Entwurf berücksichtigt werden. Es ist auch möglich, dass Abfragen bestimmter Art erlaubt sind, die Ausführung jedoch nicht effizient durchgeführt werden kann. Beispielsweise kann ein Filter nach einem bestimmten Wert eines Attributs auch erst nach dem Lesen aus dem *Datastore* ausgeführt werden.

Durch Abhängigkeiten der Faktoren untereinander ist es schwer alle Faktoren gleichzeitig optimal umzusetzen. Es besteht aber die Möglichkeit die Mindestanforderungen für einzelne Faktoren festzulegen oder die Faktoren nach ihrer Wichtigkeit zu priorisieren. Für die Abwägung der Faktoren gibt es keine allgemeingültige Entscheidung. Diese hängen vom jeweiligen Anwendungsfall des BI-Systems ab. Stehen diese Anforderungen und Prioritäten fest, kann der Datenbankentwurf durchgeführt werden.

Der geforderte Lesedurchsatz hängt von der Anzahl der Ereignisse pro Abfrage, wie sie in den Anforderungen (Kapitel 3.1.2) definiert sind, ab. Damit diese Anforderung erreicht werden kann, wird ein entsprechend hoher Lesedurchsatz benötigt. Wie in den Anforderungen bestimmt, treten einige Abfragearten eher selten auf. Die Flexibilität der Abfragen kann teilweise aufgegeben werden. So ist eine Filterung auf einen bestimmten Nutzer nachrangig und kann deshalb durch die bereits genannte Lösung des nachträglichen Verwerfens der nicht benötigten Ereignisse umgesetzt werden. Die Kosten für den Betrieb des *Datastore* sollen unter Einhaltung und Umsetzung der vorhergehenden Bestimmungen so gering wie möglich ausfallen.

3.4.2 Ereignis-Objekte

Ein Ereignis-Objekt ist ein Objekt des *Datastore* in dem genau ein Ereignis gespeichert wird. Der *Zeitstempel*, die *Nutzer-ID* und die *Aktions-ID* werden in den Schlüssel kodiert. Anwendungsspezifische Attribute des Ereignisses werden als Objekt-Attribute abgelegt.

Der *Datastore* von App Engine bietet die Möglichkeit einem Objekt beliebig viele Attribute anzuhängen. Naheliegend ist alle Attribute eines Ereignisses als Attribute eines Objektes anzuhängen. Dieser Ansatz würde völlige Freiheit bei den Abfragen ermöglichen. Da die Abfragemöglichkeiten eingeschränkt wurden, lassen sich jedoch effizientere Möglichkeiten finden. Priorisiert wurden Abfragen über alle Ereignisse in einem Zeitraum mit der gleichen *Aktions-ID*. Auf ein Filter nach anwendungsspezifischen Attributen wird zu diesem Zeitpunkt verzichtet.

Der Schlüssel, der jedes Objekt eindeutig identifiziert, kann auch Text enthalten. Dies ist von Bedeutung, da Informationen eines Ereignisses auch dort gespeichert werden können. Der Schlüssel bietet darüber hinaus Filtermöglichkeiten an, ohne einen weiteren Index zu benötigen. Wie in den Anforderungen (Kapitel 3.1) erklärt, wird der *Zeitstempel* in jeder Abfrage ausgewertet. Daher ist es sinnvoll diesen in den Schlüssel zu kodieren. Um eine Filterung nach der *Aktions-ID* realisieren zu können, muss die *Aktions-ID* als eigenes Attribut im Objekt hinterlegt werden. Mit einem Index über dieses Attribut kann die Ergebnismenge auf eine *Aktions-ID* eingeschränkt werden. Die *Nutzer-ID* könnte in einem nicht indizierten Objekt-Attribut abgelegt werden.

Es sind auch Datastore-Abfragen möglich, bei denen nicht die kompletten Objekte abgerufen werden, sondern nur deren Schlüssel. Für eine solche Datastore-Abfrage fallen wesentlich weniger Kosten an. Eine weitere Optimierung wird durch das Hinzufügen der *Nutzer-ID* und der *Aktions-ID* in den Schlüssel erreicht. Bei einer Abfrage nach diesen Informationen müssen dann nur die Schlüssel geladen werden. Wichtig hierbei ist die Reihenfolge. Da die Objekte in lexikographischer Ordnung der Schlüssel im Index gelistet werden, muss der Zeitstempel für eine Filterung am Anfang stehen. Ansonsten ging die Möglichkeit zur Filterung über einen Zeitraum verloren. In welcher Reihenfolge die *Nutzer-ID* und die *Aktions-ID* angehängt werden, ist nicht von Bedeutung.

Abbildung 7 zeigt den Aufbau eines Ereignis-Objekts. Die Ausdrücke in eckigen Klammer repräsentieren die jeweiligen Attribute des Ereignisses. Der Unterstrich symbolisiert dabei ein Trennzeichen, das den Schlüssel in drei Teile teilt. Um eine Filterung über *Aktions-ID* erreichen zu können, wird diese zusätzlich im ersten Attribut abgelegt. Falls im Ereignis anwendungsspezifische Attribute vorhanden sind, werden diese als weitere Attribute im Objekt gespeichert.

Schlüssel:	[*Zeitstempel*]_[*Nutzer-ID*]_[*Aktions-ID*]
1. Attribut:	[*Aktions-ID*]
weitere Attribute:	[anwendungsspezifischer Attributwert]

Abbildung 7: Aufbau eines Ereignis-Objekts

3.4.3 Cluster-Objekte

Da es ebenfalls möglich sein soll Cluster in Objekte des *Datastore* ablegen zu können, müssen auch für diese ein entsprechender Aufbau entworfen werden. Die Anforderungen bezüglich der Art der Abfragen sind hierbei identisch zu den Ereignis-Objekten.

Um eine Trennung der verschiedenen Ebenen sowie eine Abgrenzung zu den Ereignis-Objekten zu erhalten, wird jeweils ein anderer Objekttyp eingesetzt. Bei Datastore-Abfragen kann diese Trennung über den Typ der Objekte von vorn herein stattfinden.

Da mehrere Ereignisse in einem Objekt gespeichert werden müssen, unterscheidet sich der Aufbau vom Ereignis-Objekt. Datastore-Abfragen die ausschließlich die Schlüssel laden, sind ebenfalls nicht mehr möglich.

Abbildung 8 zeigt den Aufbau eines Cluster-Objektes. Über die Cluster-Nummer und der Information, auf welcher Ebene sich das Cluster befindet, kann der genaue Zeitraum, der vom Cluster abgedeckt wird, ermittelt werden. Beginnend mit dem ersten Cluster des 1. Januar 1970 sind die Cluster auf jeder Ebene fortlaufend nummeriert. Die vereinfachte Darstellungsform des Zeitraumes hat den Vorteil, dass keine zwei Daten gespeichert werden müssen und das Cluster eindeutig identifiziert werden kann.

Da im Allgemeinen nicht alle Ereignisse eines Clusters in ein Objekt überführt werden können, wird zusätzlich eine Objekt-Nummer angehängt. Da der Schlüssel eindeutig sein muss, wird es dadurch möglich, mehrere Objekte mit der gleiche Cluster-Nummer und *Aktions-ID* zu speichern.

Das erste Attribut speichert mehrere Werte. Ein Wert repräsentiert die Inhalte eines Ereignisses. Jeder Wert besteht aus der Verknüpfung von *Zeitstempel* und *Nutzer-ID*. Das zweite Attribut ist lediglich im ersten Objekt eines Clusters enthalten. Der Wert enthält die Anzahl der existierenden Objekte des Clusters.

Schlüssel:	[Cluster-Nr.]_[*Aktions-ID*]_[Objekt-Nr.]
1. Attribut:	[*Zeitstempel*]_[*Nutzer-ID*] [*Zeitstempel*]_[*Nutzer-ID*] ...
2. Attribut:	[#Objekte]

Abbildung 8: Aufbau eines Cluster-Objekts

Abfragen ohne Spezifikation eines Filters für eine *Aktions-ID* können, ähnlich zu den Ereignis-Objekten, über die Einschränkung der Schlüssel durchgeführt werden. Ist zusätzlich noch auf eine *Aktions-ID* eingeschränkt, werden die Ereignisse nicht über eine Datastore-Abfrage sondern direkt über die Schlüssel geladen. Für die gesuchten Ereignisse können alle Schlüssel berechnet werden. Nach der Auswertung des ersten Objekts eines Clusters ist die Anzahl der existierenden Objekte bekannt. Diese können dann ebenfalls über ihre Schlüssel geladen werden. Mit dem genauen *Zeitstempel* und der *Nutzer-ID* können alle Ereignisse wiederhergestellt werden.

3.5 Puffer und Zwischenspeicher

Caching, also das vorübergehende Ablegen der Daten in einen Speicher mit kürzeren Zugriffszeiten, kann auch für ein BI-System sinnvoll eingesetzt werden. Zu den Vorteilen der kürzeren Zugriffszeit kommt bei App Engine der Vorteil der Kosteneinsparung. Abgesehen von den Kosten für den Betrieb der Instanz, entstehen für die Dienste, die Caching ermöglichen, keine

weiteren Kosten. Lassen sich dabei Zugriffe auf den *Datastore* vermeiden, können die entstehenden Kosten verringert werden.

3.5.1 Dienste für Caching

Für das Zwischenspeichern von Daten können in App Engine verschiedene Dienste genutzt werden. Der übliche Weg wird in der Nutzung des *Memcache* liegen. Die Funktionen der *Task Queue* können auch für das Speichern von Daten verwendet werden. Durch das Anhängen von Nutzdaten, können diese mit dem Einreihen in die Warteschlange abgelegt werden.

Ein von App Engine unabhängiger Weg kurzzeitig Daten abzulegen, bietet die Java Umgebung. Klassenvariablen können Java-Objekte über die Dauer des Lebenszyklus einer Servlet-Instanz speichern. Bei einem Betrieb in App Engine ist die Lebensdauer einer Servlet-Instanz mit der Lebensdauer der App-Engine-Instanz gleichzusetzen.

3.5.2 Caching-Konzepte

Da alle drei Dienste verschiedene Vor- und Nachteile aufweisen, gibt es verschieden Einsatzszenarien die den Ablauf des Gesamtsystems an mehrere Stellen beschleunigen können.

Datastore-Cache: Ein erster Ansatz liegt in der Verbesserung der Lesegeschwindigkeit. Dazu werden die Cluster einer Ebene nach dem Lesen aus dem *Datastore* im *Memcache* abgelegt. Bei einem erneuten Zugriff muss nicht mehr auf den *Datastore* zugegriffen werden. Es besteht die Gefahr zu viele Daten im Memcache zu speichern. Dadurch wird ein Teil der Daten nur über kurze Zeit im Speicher gehalten. Eine genaue Größe des *Memcache* ist nicht bekannt. Um diesem vorzubeugen, wird nur eine Ebene im *Memcache* abgelegt.

Da bei einer Abfrage zuerst auf den *Memcache* zugegriffen wird, verzögert sich die Abfrage von Ereignissen, wenn diese nicht im *Memcache* enthalten sind. Bei stark variierenden Abfragen besteht somit die Gefahr, dass das System durch viele missglückte Cache-Zugriffe insgesamt langsamer wird. Die zusätzliche Zeit zum Schreiben der Ereignisse in den *Memcache* muss ebenfalls berücksichtigt werden.

Da diese Art der Nutzung des *Memcache* als Datastore-Cache üblich ist (Kapitel 2.2), kann davon ausgegangen werden, dass eine Umsetzung dennoch Vorteile bringt.

Eine Nutzung von Klassenvariablen als Datastore-Cache wäre ebenfalls denkbar. In einem skalierenden System mit mehreren Instanzen kann sich der Einsatz jedoch nachteilig auswirken. Da Klassenvariablen den Speicher der Instanz nutzen, würde jede Instanz ihren eigenen Datastore-Cache betreiben (vgl. Abbildung 1). Bei einer zufälligen Verteilung der Abfragen über die Instanzen ist nicht garantiert, dass Abfragen mit gleicher Datengrundlage von der gleichen Instanz bearbeitet werden. Die Ereignisse können somit seltener aus dem Datastore-Cache gelesen werden. Für ein System mit einer oder nur sehr wenigen Instanzen kann sich die Nutzung von Klassenvariablen, insbesondere durch die kurze Zugriffszeit, lohnen. Um den Zielen der hohen Skalierbarkeit gerecht zu werden, muss an dieser Stelle von einer Umsetzung abgesehen werden.

Ereignispuffer: Das einzelne Speichern der Ereignisse in separaten Objekten erzeugt einen großen Anteil der Kosten für den *Datastore*. In einem weiteren Ansatz wird versucht, diesen zu verringern. Durch das Ablegen eingehender Ereignisse in die *Task Queue* und das direkte Erstellen von Clustern aus diesen Ereignissen könnte auf eine separate Speicherung der Ereignisse als einzelne Objekte im *Datastore* verzichtet werden.

Hierzu wird bei eingehendem Ereignis eine Aufgabe zur Speicherung in eine Warteschlange abgelegt. Der Aufgabe werden dabei alle Informationen die das Ereignis beinhaltet angehängt. Die Konfiguration der Warteschlange zu einer Pull-Warteschlange ermöglicht eine Entkopplung. In periodischen Abständen können diese Aufgaben gesammelt abgearbeitet werden. Anstelle einer einzelnen Speicherung können aus den Ereignissen so direkt Cluster erstellt werden.

Es muss beachtet werden, dass Ereignisse mit zusätzlichen anwendungsspezifischen Attributen gesondert verarbeitet müssen. Da in der Cluster-Datenhaltung die Ereignisse aktuell ohne diese Attribute gespeichert werden, müssen diese Ereignisse im *Datastore* abgelegt werden und für die Erstellung der Cluster von dort geladen werden. In Fällen, in denen ein Großteil der Ereignisse anwendungsspezifische Attribute besitzt, werden die Verbesserungen durch das Sammeln von Ereignissen vermindert.

In Abbildung 9 wird der Ablauf bei eingehenden Ereignissen verdeutlicht.

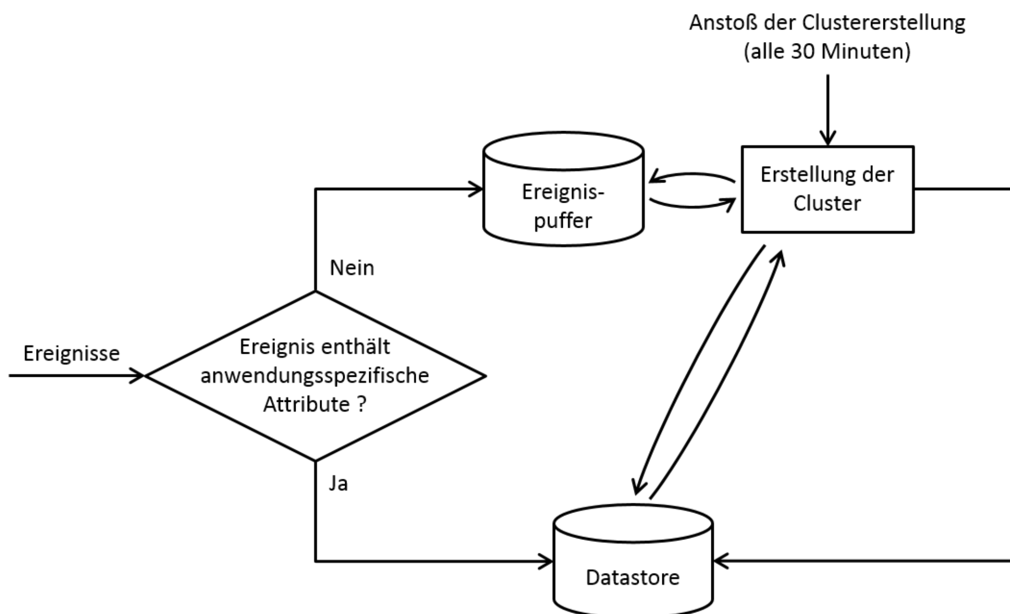

Abbildung 9: Ablauf mit Ereignispuffer

Theoretisch wird die Skalierbarkeit mit der Umsetzung des Konzeptes beschränkt. Auch wenn für die Nutzung der *Task Queue* keine Kosten entstehen, ist die Anzahl der API-Aufrufe für diesen limitiert. Die aktuelle Grenze liegt bei einer Milliarde Aufrufe pro Tag (Kapitel 2.2.1). Da Aufrufe für das Einstellen, Abholen und Löschen entstehen, dürfte dies für rund 333 Millionen Ereignisse pro Tag ausreichen. Bei Betrachtung der Anforderungen wird dieses Limit in der Praxis nicht erreicht.

4 Umsetzung

Im Rahmen der Arbeit wurde eine prototypische Umsetzung des Konzeptes in der Programmiersprache Java erarbeitet. Bei der Umsetzung wurden Java 7.0 Update 40 sowie das aktuelle App Engine SDK (1.8.4) verwendet. Der Aufbau der Klassenstrukturen orientiert sich im Wesentlichen an den im Entwurf erläuterten Komponenten.

4.1 Überblick

Als Überblick über den Aufbau der Programmstruktur dient ein UML-Klassendiagramm (Abbildung 10).

`GlobalSettings` ermöglicht eine Konfiguration des Systems. Das Verhalten des Systems kann durch das Setzen mehrerer Variablen beeinflusst werden. Verschiedene statische Methoden erlauben die Abfrage dieser Werte abgefragt.

`useEventBuffer()`: Mit einem false-Wert als Rückgabe von `useEventBuffer()`, kann der Ereignispuffer deaktiviert werden. Damit werden alle Events unverzüglich in den *Datastore* geschrieben.

`getAuthToken()`: Die Methode gibt einen String mit einem Sicherheitsschlüssel, welcher für den Zugriff auf die Highcharts-API benötigt wird, zurück.

`getClusterLevel()`: `ClusterLevel`-Objekte beschreiben eine Ebene in der Datenstruktur. Über `getClusterLevel()` werden die aktuellen Ebeneneinstellungen abgerufen. Jede Ebene besitzt einen Namen und die Länge des Abdeckungszeitraums. Ebenfalls kann ein Minimalwert für die Anzahl der im Cluster gespeicherten Ereignisse angegeben werden. Durch Setzen dieses Wertes größer Null wird ein Cluster nur geschrieben, wenn es Ereignisse beinhaltet. Der Zustand der Ebene wird durch `State` ausgedrückt. Als Werte sind definiert: `DEACTIVATED` für eine aktuell nicht genutzte Ebene, `READONLY` für deaktivierten Schreibzugriff, `ACTIVE` für Schreiben und Lesen und `CACHED` für die zusätzliche Nutzung des Caches für diese Ebene. Durch das Deaktivieren von Ebenen können diese vorübergehend oder dauerhaft außer Betrieb genommen werden. Ist dies gewünscht, kann es sinnvoll sein, übergangsweise nur noch lesend auf eine Ebene zuzugreifen. Das Ändern von Werten in `GlobalSettings` kann auch im laufenden Betrieb erfolgen.

Die Logik des Systems wird über ein Interface angesprochen. `BiController`, `BiEvent` und `BiQuery` bieten hierbei die Schnittstellen zur Kommunikation. Die verfügbaren Umsetzungen bilden die Funktionsweisen des Entwurfes ab. Es ist möglich diese durch eigene Umsetzungen zu ersetzen. Um eine durchgängige Verwendung anderer Klassen zu erreichen, werden Objekte der drei Interfaces ausschließlich von der `BiBuilder`-Klasse erzeugt. Änderungen können dadurch einfach ins gesamte System übernommen werden.

Alle für die Datenhaltung in App Engine benötigten Dienste werden in dem Unterpaket `bi.impl.gae` verwendet. Durch eine Einschränkung in der Verwendung von App Engine spezifischen Objekten auf wenige Klassen, wird eine Unabhängigkeit erreicht. Eine zusätzliche Kapselung dieser Klassen erlaubt weitere Umsetzungen. Dies ermöglicht anstelle des *Datastore* von Google App Engine andere Datenhaltungssysteme anzubinden. Beim Betrieb auf App Engine wird dies nicht ohne Weiteres unterstützt. Bei dem Betrieb auf einem Server mit entsprechender Java-Umgebung können auf diese Weise weitere Datenbanken angebunden werden. Da die Datenstruktur auf den *Datastore* von App Engine optimiert wurde, ist ein ähnlicher Datenbankaufbau wichtig.

4.2 Webschnittstellen

Die Webschnittstelle lässt sich zwei Bereiche einteilen. Zum einen wird eine direkte REST-Schnittstelle (Representational State Transfer; vgl. [Fie00]) zur Übertragung von Ereignissen angeboten. Zum anderen können zu komplexeren Fragestellungen des BI Informationen angefordert werden. Um den Zugriff auf die Nutzungsdaten zu beschränken, kann ein Authentifi-

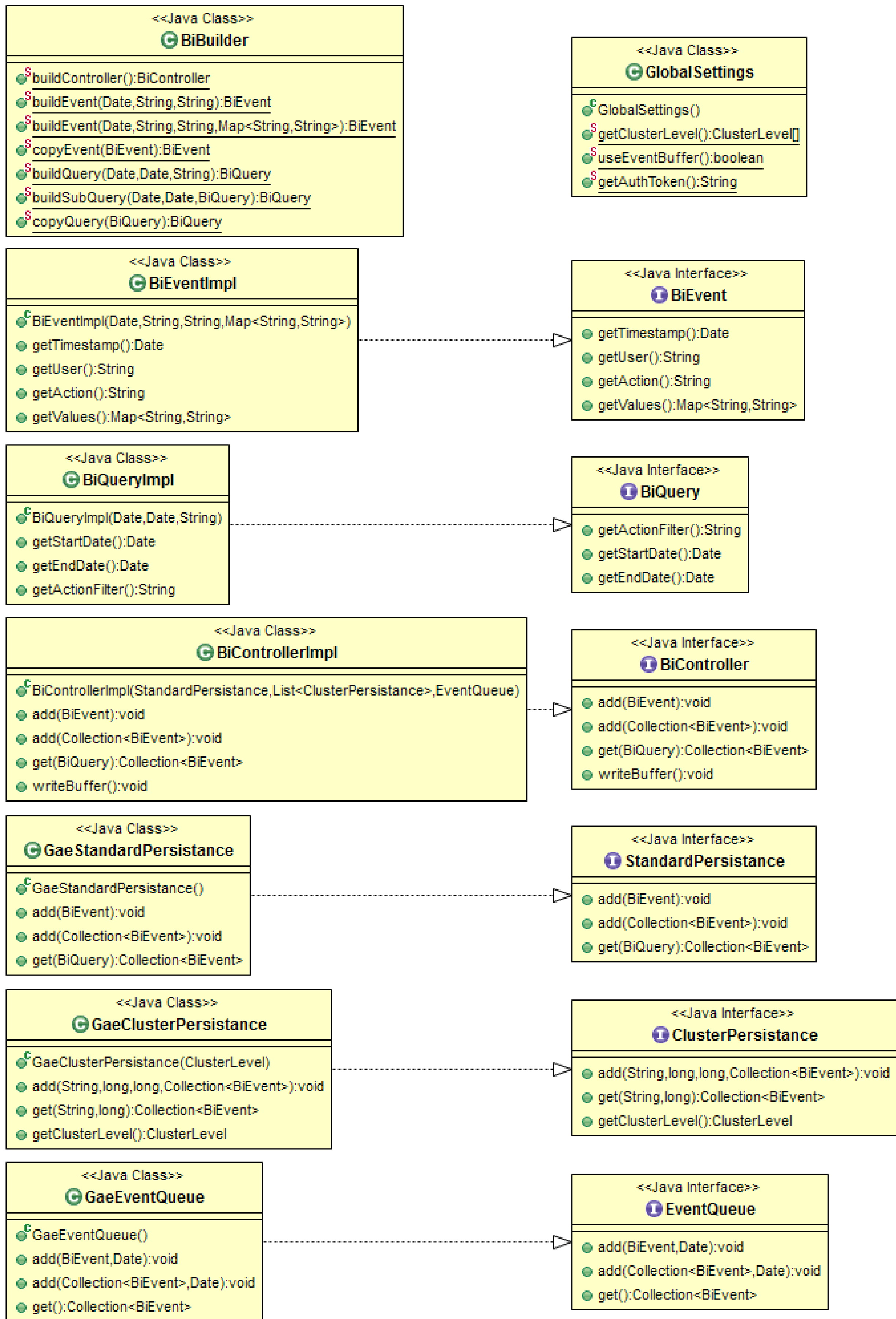

Abbildung 10: UML-Klassendiagramm

zierungstoken festgelegt werden. Ist dieses gesetzt, müssen alle Webanfragen diesen Token als Parameter mitführen. Ohne oder mit falschem Authentifizierungs-Parameter wird die Anfrage abgewiesen.

Die direkte Webschnittstelle bietet zwei Zugriffsmöglichkeiten an. Mit einer können Ereignisse ins System übertragen werden. Als Parameter werden dabei die *Aktions-ID*, die *Nutzer-ID* sowie optional der *Zeitstempel* übergeben. Wird kein *Zeitstempel* angegeben, wird der aktuelle Zeitpunkt hinzugefügt. Die zweite Zugriffsmöglichkeit erlaubt es Abfragen zu stellen. Angaben zum Zeitraum werden benötigt. Optional kann ein String zur Filterung auf eine *Aktions-ID* mitgegeben werden. Das Abfrageergebnis wird als Liste von Ereignissen in eine JSON-Struktur umgewandelt und zurückgegeben. Beide Aufrufe werden auf die äquivalenten `BiController`-Methoden abgebildet.

Für eine Visualisierung der Daten wurde eine weitere, exemplarisch umgesetzte Schnittstelle angelegt. Diese ermöglicht gleichzeitig komplexere Abfragen zu stellen. Die Ergebnisse werden nicht als Liste aus Ereignissen, sondern in einer Grafikkonfiguration für die Highcharts-Grafikbibliothek zurückgegeben. Erstellt werden diese Grafikkonfigurationen mit Wicked Charts[12], einer Java-Abstraktion der Highcharts-Bibliothek. Mit Wicked Charts lassen sich Konfigurationen erstellen und in JSON überführen. Aus der Grafikkonfiguration erstellt Highcharts ein Diagramm (Beispielgrafik in Abbildung 11). Die Art des Diagramms, sämtliche weitere Darstellungsparameter und die Diagrammwerte sind in der Grafikkonfiguration enthalten. Durch die grafische Darstellung ist es möglich einen guten Überblick über die Ereignisse zu bekommen.

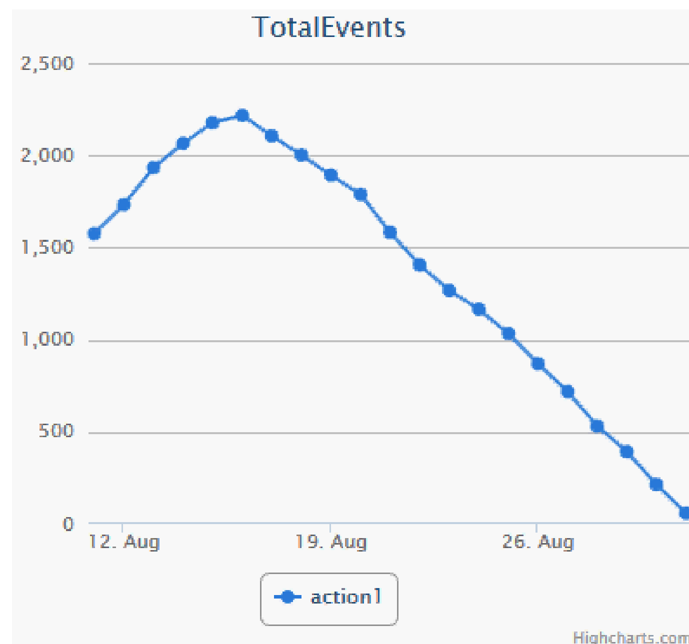

Abbildung 11: Highcharts Beispielgrafik

Die Webschnittstellen wurden mit Hilfe des Spring Frameworks[13] umgesetzt. Dieses erlaubt eine einfache und flexible Gestaltung der URLs und Parameter.

[12]https://code.google.com/p/wicked-charts
[13]http://spring.io

4.3 Bearbeitung von Abfragen

In der Methode `get(BiQuery query)` der Klasse `BiControllerImpl` muss die Abfrage entsprechend ihres Zeitraumes einer Ebene zugeordnet werden. Bei der Umsetzung gibt es dabei viele Möglichkeiten mit fehlenden oder angebrochenen Clustern umzugehen. In der vorliegenden Umsetzung wird ein Aufruf in absteigender Reihenfolge über die einzelnen Ebenen rekursiv verarbeitet. Die Aufrufe werden dazu auf eine nicht öffentliche Methode `get(BiQuery query, 0)`, wobei letzterer Parameter die höchste vorhandene Ebene darstellt, umgeleitet.

In der Methode `get(BiQuery query, int level)` wird zuerst sichergestellt, dass die Abfrage auf dieser Ebene verarbeitet werden kann. Dazu wird geprüft ob es die mit `level` beschriebene Ebene gibt. Sollte dies nicht der Fall sein, wird die Abfrage über `StandardPersistance` verarbeitet. Ist der Abfragezeitraum zu klein oder die Ebene deaktiviert, wird die Abfrage durch erhöhen der `level`-Variable an die nächst kleinere Ebene weitergegeben. In allen weiteren Fällen wird versucht die Abfrage selbst zu verarbeiten. Das Ablaufdiagramm (Abbildung 12) verdeutlicht die Vorgehensweise.

Angebrochene Cluster zu Beginn und Ende des Zeitraums werden an unterliegende Ebenen weitergeleitet. Fehlende Cluster werden ebenfalls als Unterabfrage weitergegeben. Die Ereignisse werden nach erfolgter Unterabfrage für die Erstellung des Clusters verwendet.

Wie dem Diagramm zu entnehmen ist, wird bei fehlenden Clustern stets versucht den Zeitraum nicht weiter zu zerteilen. Dies vermeidet die Ausführung extrem vieler Unterabfragen. Der Verzicht auf eine solche Routine würde die Bearbeitungszeit einer Abfrage erheblich erhöhen. Eine Abfrage über den Zeitraum von zwei Wochen, für den noch keine Cluster vorhanden sind, würde folgendermaßen verarbeitet werden: Die Abfrage wird auf 14 Teilabfragen mit einem Zeitraum von einem Tag und anschließend auf 672 Teilabfragen über eine halbe Stunde zerteilt. Da diese Cluster ebenfalls nicht vorhanden sind, werden diese Teilabfragen zur Suche von Ereignissen aus Ereignis-Objekten sequentiell an den *Datastore* gestellt.

5 Evaluation

Durch eine Evaluation können Aussagen über die Erfüllung der Anforderungen und Ziele des Systems getroffen werden. Neben einem Überblick über die im Betrieb entstehenden Kosten, soll hauptsächlich der Erfolg der Skalierbarkeit überprüft werden. Daraus abgeleitet lässt sich eine allgemeine Abschätzung über die Möglichkeiten eines BI-Systems auf Google App Engine aufstellen.

5.1 Durchführung und Rahmenbedingungen

Die Prüfung der Leistungsfähigkeit einer in der Cloud betriebenen Anwendung ist nicht einfach. Insbesondere wenn sich einer der zwei definierten Maßstäbe auf die Zugriffe pro Minute bezieht, kann ein Test ohne weitere Hilfsmittel nicht erfolgen. Aus diesem Grund wird für den wiederholten, parallelisierten Aufruf von Webressourcen ein kleines Programm eingesetzt. Diese, für diesen Fall umgesetzte, Aufrufroutine erlaubt eine zufällige Erzeugung von Ereignissen über die REST-Schnittstelle des BI-Systems. Die Anzahl der parallelen Verarbeitungsfäden kann dabei konfiguriert werden. Die Ereignisse verteilen sich über den Zeitraum von einem Monat. Wobei sich die Verteilung zur Mitte des Monats häuft. Dies ermöglicht Abfragen über Tage mit verschieden vielen Ereignissen. Des Weiteren gibt es, wie in den Anforderungen (Kapitel 3.1.2) abgeschätzt, 10.000 verschiedene Testnutzer sowie 20 verschiedene Aktionen. Während die Nutzer gleich verteilt gewählt werden, findet bei den Aktionen eine Verteilung von 50%

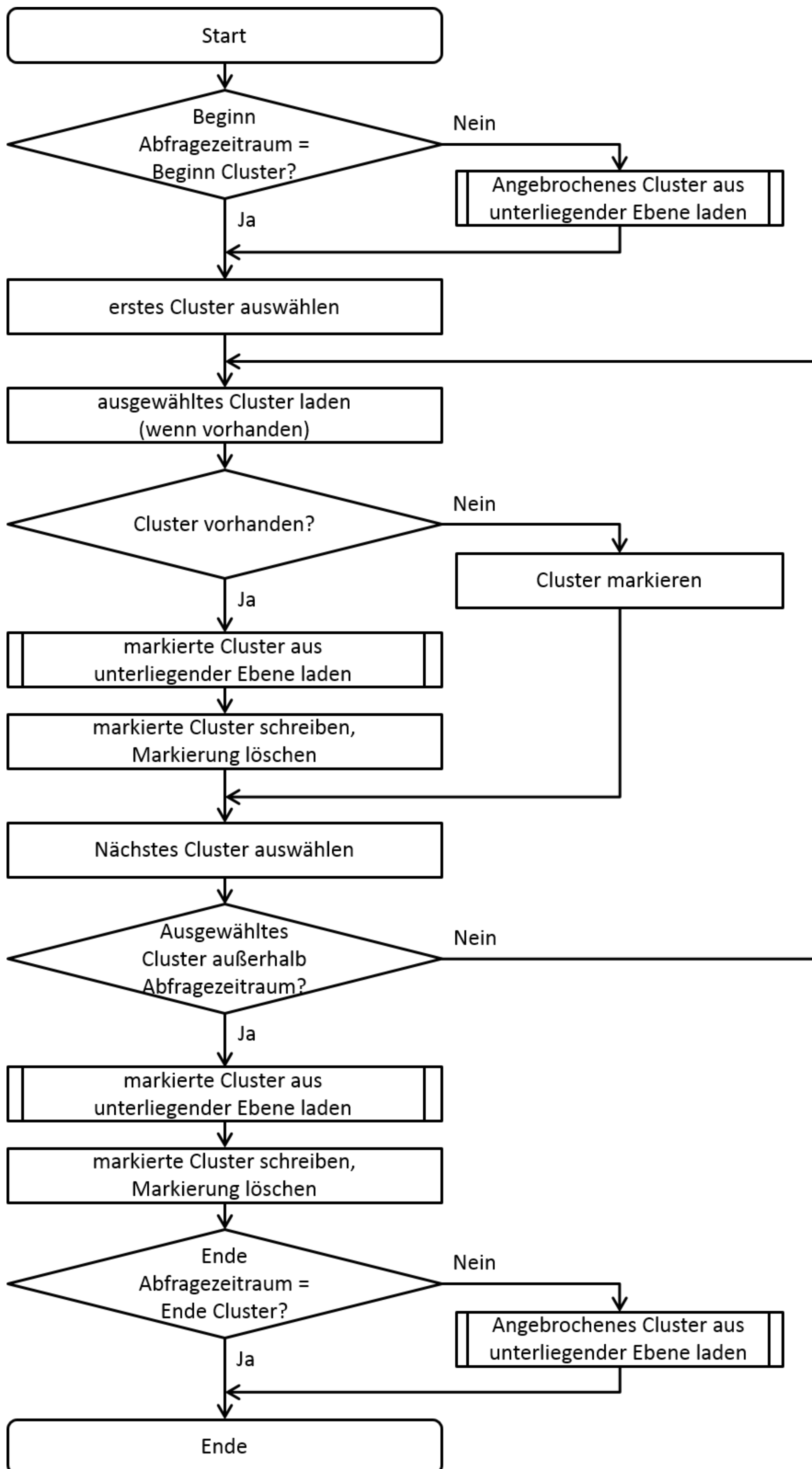

Abbildung 12: Verarbeitungsablauf Abfragen

auf die erste Aktion und 50% zu gleichen Teilen auf die restlichen Aktionen statt. Dadurch wird es ermöglicht, bei einer relativ geringen Gesamtmenge der Ereignisse, Abfragen über eine *Aktions-ID* mit vielen Ereignissen durchzuführen.

5.1.1 Eingehende Ereignisse

Da es im Einsatz unterschiedliche Anwendungsfälle geben wird, kann ein aussagekräftiges Ergebnis nur mit verschiedenen Testszenarien erreicht werden. Bei den Testszenarien werden die Ereignisse über die REST-Schnittstelle ins System übertragen. Grundlegend zu unterscheiden ist der Einsatz des Ereignispuffers. Ist dieser aktiviert, werden die Ereignisse in eine *Task Queue* abgelegt. Bei deaktiviertem Puffer werden diese sofort in den *Datastore* geschrieben. Diese unterschiedlichen Verarbeitungsarten der eingehenden Ereignisses müssen separat betrachtet werden.

Die Ereignisse werden mit mehreren Verarbeitungsfäden gleichzeitig in das System übertragen. Jeder dieser Verarbeitungsfäden erzeugt fortlaufend zufällige Ereignisse und sendet dies an das BI-System. Sobald der zugehörige Http-Aufruf abgeschlossen ist, wird ohne weitere Verzögerung ein neues Ereignis erzeugt und abgeschickt. Die Last, die am System entsteht, wird durch die Anzahl der Verarbeitungsfäden bestimmt. Entsprechend den Anforderungen sollte diese Anzahl so gewählt werden, dass mindestens 3.125 Ereignissen pro Minute, also rund 52 Ereignisse pro Sekunde, am System eintreffen.

Durch die automatische Erstellung von Ereignissen können die Ergebnisse direkt in Logdateien geschrieben werden. Relevante Informationen bei der Auswertung können neben der Anzahl der Ereignisse pro Sekunde auch die durchschnittliche Verarbeitungszeit eines Http-Aufrufs sein. Diese lässt sich ebenfalls für jede Sekunde aus den Logdateien auswerten. Über die Administrationskonsole von App Engine kann zusätzlich beobachtet werden, wie viele Instanzen aktuell aktiv sind.

Diesen Informationen sollen Aufschluss über die Skalierung des Systems geben. Durch die dichte Abfolge von eingehenden Ereignissen können mögliche Grenzen der Skalierbarkeit bezüglich der Zugriffsrate gefunden werden. Wie bereits Eingangs erläutert, findet auf Google App Engine eine Skalierung über das Starten oder Beenden von Instanzen statt. Das Starten von Instanzen benötigt Zeit. Eine neue Instanz steht somit nicht unmittelbar auf Abruf zur Verfügung. Die Zeit, die von der Anforderung einer neuen Instanz bis zu deren Betriebsbereitschaft vergeht, entscheidet über die Skalierungsgeschwindigkeit. Eine Einschätzung dieser Skalierungsgeschwindigkeit kann ebenfalls mithilfe der Ergebnisse eines solchen Testes durchgeführt werden.

5.1.2 Abfragen

Bei der Abfrage von Ereignissen gibt es ebenfalls verschiedene Szenarien. Die gestellten Abfragen unterscheiden sich im Wesentlichen durch die Anzahl der Ereignisse, die als Ergebnis zurückgeliefert werden. Für den Aufwand, der im System entsteht, gibt es weitere Faktoren. Entscheidend ist auch die Art des Zugriffes. Die Verarbeitung kann über Ereignis-Objekte oder über Cluster erfolgen. Für ersteren Fall spielt es eine Rolle, ob mit den Daten unmittelbar Cluster erstellt werden oder nicht. Beim Zugriff auf Cluster könnten diese auch aus dem Datastore-Cache gelesen werden. Auch bei der weiteren Verarbeitung der Ereignisse gibt es Unterschiede. Bei einer direkten Abfrage der Daten müssen diese in JSON umgewandelt werden. Bei einer Abfrage zur Darstellung als Grafik muss eine Highcharts-Grafikkonfiguration erstellt werden. In der Evaluation gilt es möglichst viele dieser Fälle abzudecken, um so eine umfassende Beurteilung des Systems erhalten zu können.

Als Auswertungsmaterial stehen hauptsächlich die Logdateien von App Engine zur Verfügung. Diese geben Aufschluss über die Anzahl der Ereignisse jedes Abfrageergebnisses. Neben der Gesamtverarbeitungszeit der Abfragen können auch die Zugriffszeiten auf die einzelnen Ebenen aus den Logdateien von App Engine entnommen werden. Die Ladezeiten sowie die Anzahl der Ereignisse der einzelnen Ebenen können durch eine Protokollierung innerhalb des Systems eingesehen werden.

Durch die Auswertung dieser Daten kann ein Bild von der üblichen Verarbeitungsroutine bei Abfragen erstellt werden. Diese können Rückschlüsse auf die maximale Datenmenge, die ein Aufruf verarbeiten kann, und die Zeit, welche er für die einzelnen Schritte benötigt, geben. Ziel ist es die Grenzen der Skalierbarkeit bezüglich des Verarbeitungsaufwandes eines Zugriffes abzuschätzen.

5.2 Ergebnisse

5.2.1 Eingehende Ereignisse

Bei einem Test, der in einem unerwarteten Zeitpunkt viele Zugriffe startet, zeigen sich die Skalierungseigenschaften besonders. Mit 100 Verarbeitungsfäden wurden in 80 Sekunden 14.029 Ereignisse geschrieben. Zu Beginn liefen keine Instanzen auf App Engine. Dies führte dazu, dass die Zugriffe der ersten Sekunden hohe Antwortzeiten erreichten. Durch den automatischen Start mehrere Instanzen konnten überlastungsbedingte Verzögerungen nach weniger als 30 Sekunden vermieden werden. Die Grafik (Abbildung 13) zeigt den Ablauf des Testes. Die gestarteten Zugriffe sowie die durchschnittliche Antwortzeit dieser wurden pro Sekunde erfasst. Nach 30 Sekunden liegen die Antwortzeiten der Zugriffe unter einer Sekunde. Die Anzahl der Ereignisse, die pro Sekunde ins System übertragen werden, steigt ab diesem Zeitpunkt an. Bei etwa 300 Ereignissen pro Sekunde ist die Grenze der Zugriffshäufigkeit der 100 Verarbeitungsfäden erreicht. Die Kurve steigt deshalb nicht weiter.

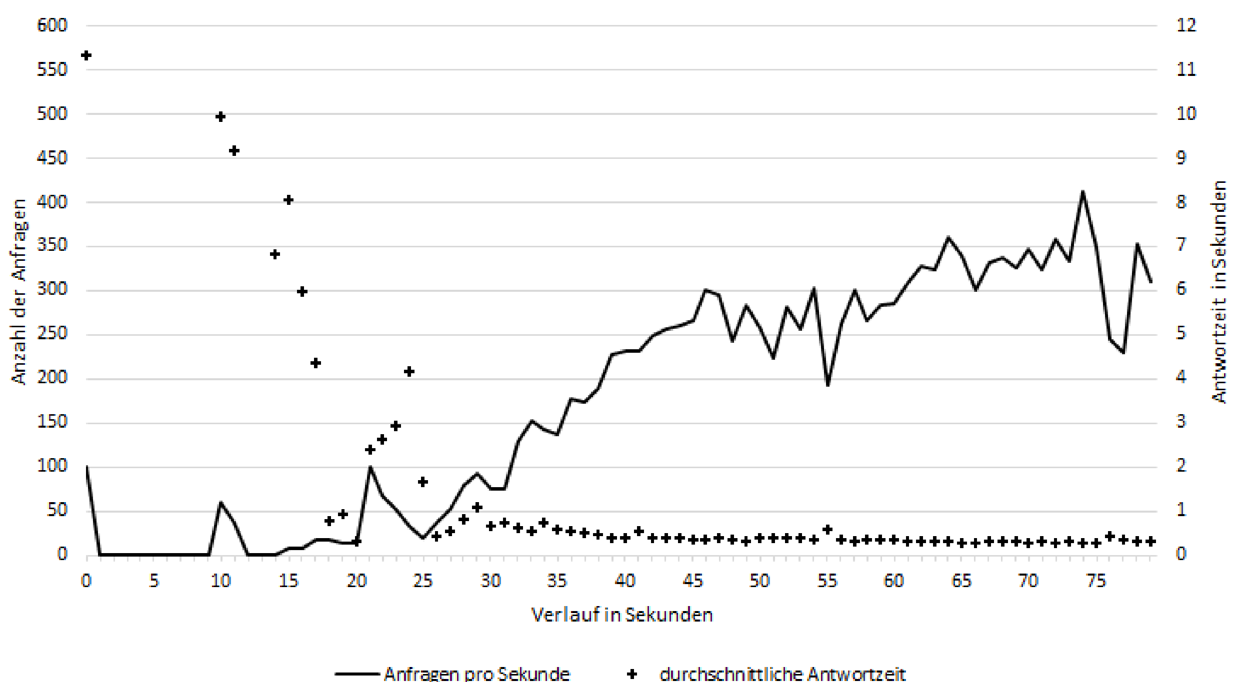

Abbildung 13: Testverlauf: Schreiben von Ereignissen

In Tests mit 300 Verarbeitungsfäden wurde über 30 Sekunden eine außergewöhnlich hohe Belastung simuliert. Mit zu Beginn wenigen laufenden Instanzen wurden von App Engine nach und nach neue Instanzen gestartet. Beim Schreiben der Ereignisse in den *Datastore* wurden 9.334 Ereignisse erreicht, mit aktiviertem Ereignispuffer wurden 11.536 Ereignisse gemessen. Die durchschnittlichen Antwortzeiten betrugen dabei eine Sekunde und 0,8 Sekunden. Abbildung 14 zeigt den Verlauf beider Tests. Die Ursache des Abfalls einer der Kurven zu Beginn kann durch die Skalierung des Systems zu diesem Zeitpunkt erklärt werden. Wie die Grafik zeigt, wurden gegen Ende über 400 Ereignisse pro Sekunde in das System übertragen.

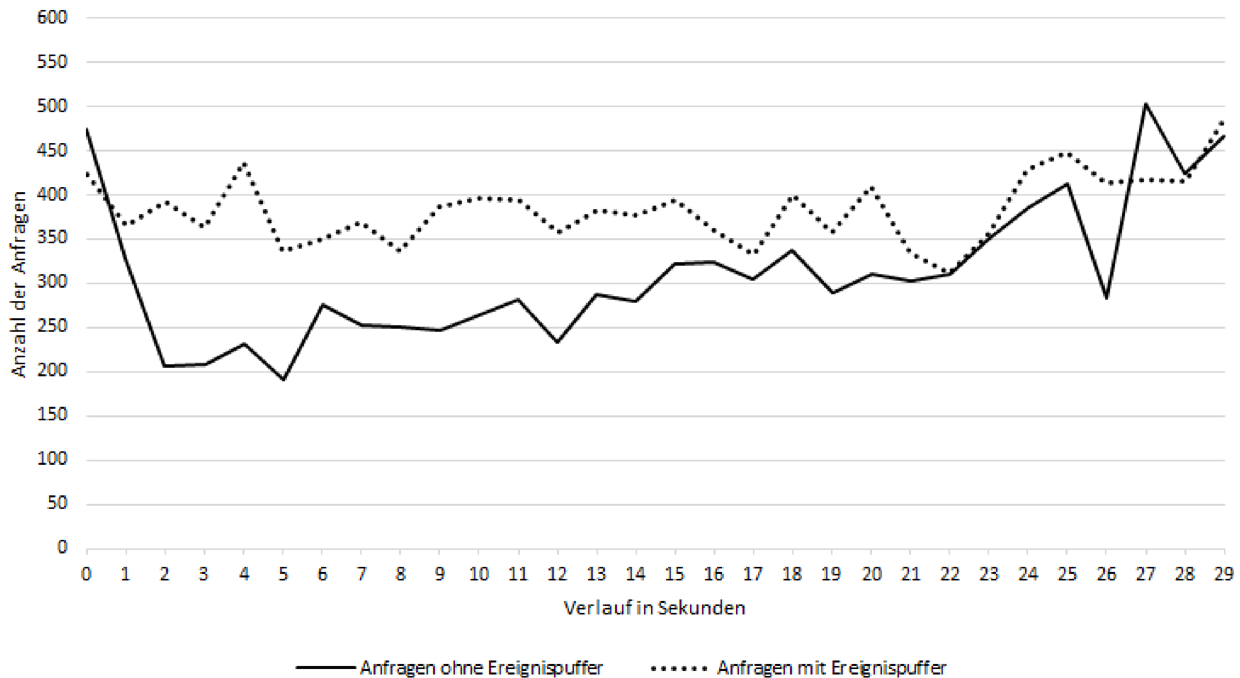

Abbildung 14: Testverlauf: Vergleich mit Ereignispuffer

Eine längere Auswertung über fünf Minuten wurde mit 20 Verarbeitungsfäden durchgeführt. Dabei wurden durchschnittlich 69 Ereignisse pro Sekunde ins System übertragen. Während des Tests zeigte das System keine Auffälligkeiten, die gegen einen dauerhaften Betrieb gesprochen hätten. Bis auf geringe Schwankungen wurden die Ereignisse mit konstanter Rate verarbeitet.

5.2.2 Abfragen

Das Abfragen von Ereignissen wurde ebenfalls über die REST-Schnittstelle durchgeführt. Bei den ersten Tests wurden Abfragen ohne Einschränkung auf eine *Aktions-ID* an das System übermittelt. Die Bearbeitung fand hierbei ohne Zuhilfenahme der verschiedenen Cluster statt. Eine Abfrage über 82.492 Ereignissen wurde in 25,9 Sekunden verarbeitet. Laut Protokollierung entfallen davon 19 Sekunden auf das Laden der Ereignisse aus dem *Datastore*. Die verbleibende Zeit wurde zur Aufbereitung der Ereignisse, in diesem Fall für das Erstellen der JSON-Struktur, benötigt. Die Antwort der Webanfrage betrug 6,4 Megabyte.

In vorigem Fall wurden die Daten lediglich gelesen. In einem weiteren Test wurden die Ebenen durch die Abfragen um fehlende Cluster ergänzt. Zur besseren Analyse beinhalten die Ebenen zu Beginn jeder Abfrage keine Cluster. Wie bisher auch, wurden die Ereignisse auch als JSON-Struktur zurückgegeben. Die Zeit um die Ereignisse aus dem *Datastore* zu laden entsprach, im Verhältnis zu der Anzahl der Ereignisse, obigem Ergebnis. Die Zeiten für das Erstellen und

Schreiben der Cluster betrugen im Test, je nach Anzahl der Ereignisse, bis zu 1,4 Sekunden je Ebene. Dieser Wert wurde für 93.885 Ereignisse gemessen. Es ist zu erwarten, dass mehr Ereignisse eine längere Zeit zur Erstellung der Cluster benötigen. Zeitliche Unterschiede bei der Erstellung der Cluster auf den verschiedenen Ebenen gibt es nur geringfügig. Bei der eben genannten Abfrage benötigte die Erstellung der Ein-Tages-Cluster insgesamt 20 Millisekunden länger als die Erstellung der 30-Minuten-Cluster. In anderen Fällen wurden die kleineren Cluster unwesentlich schneller erstellt. Die Dauer für die Erstellung und Speicherung der Cluster scheint nur von der Anzahl der Ereignisse abhängig zu sein. Die Anzahl und Größe der Cluster haben darauf keinen Einfluss.

Um das Auslesen der Ereignisse aus dem *Datastore* zu beschleunigen, wurden Ebenen und Cluster eingeführt. Die Überprüfung des so erhofften Vorteils bei der Verarbeitungsgeschwindigkeit wurde ebenfalls durchgeführt. Das Lesen von 766.513 Ereignissen dauerte durchschnittlich 21,5 Sekunden. Ob die Daten aus dem *Datastore* oder aus dem Datastore-Cache geladen wurden, wirkte sich dabei weder positiv noch negativ auf die Verarbeitungszeit aus. Die 21,5 Sekunden beziehen sich nur auf den Ladevorgang. Messungen über die weitere Verarbeitungszeit waren nicht möglich. Für das Erstellen der JSON-Struktur reichte der Speicher der Instanz nicht aus. Die JSON-Struktur allein würde hochgerechnet etwa 60 Megabyte groß sein. Es ist davon auszugehen, dass für die Erstellung zusätzlicher Speicher benötigt wird. Die Ereignisse selbst konnten deshalb nicht zurückgegeben werden.

Beim Anfordern von Highcharts-Grafikkonfigurationen werden die Ereignisse nicht einzeln zurückgegeben. Im Test mit Highcharts traten die eben beschriebenen Probleme mit zu geringer Speicherkapazität nicht auf. Abfragen an die Highcharts-Schnittstelle des Systems wurden im Schnitt in 27,6 Sekunden beantwortet. Datengrundlage waren ebenfalls 766.513 Ereignisse. Unter Abzug der Ladezeit der Daten, ergibt sich für die Kalkulation und Erstellung der Grafikkonfiguration eine Zeit von etwa 6 Sekunden.

5.3 Beurteilung

5.3.1 Skalierbarkeit

Die Geschwindigkeit beim Schreiben von Ereignissen im System kann positiv bewertet werden. Die Anforderung (siehe Kapitel 3.1.2) von 3.125 eingehenden Ereignissen pro Minute wurde deutlich übertroffen. Mit einer Ereignisrate von über 300 Ereignissen pro Sekunde wurden mehr als das Fünffache gemessen. Als Limit für die maximale Auslastung ist dies jedoch nicht zu sehen. Die Werte sind durch die Leistungsfähigkeit der Simulationsumgebung beschränkt. App Engine konnte die Last durch Verteilung auf mehrere Instanzen problemlos verarbeiten.

Bei unerwarteten Auslastungsschwankungen nach oben reagiert App Engine durch das Starten weiterer Instanzen. Nach wenigen Sekunden kann das System die erhöhte Last ohne Einschränkungen verarbeiten. In der Skalierungsphase werden Webanfragen verzögert bearbeitet. Diese Skalierungsgeschwindigkeit wird für ein BI-System in aller Regel mehr als ausreichend sein.

Das Ebenen-Konzept zur Vorberechnung von Ereignissen, die gemeinsam abgefragt werden, erscheint grundsätzlich nicht schlecht. Daten können somit deutlich schneller aus dem *Datastore* abgerufen werden. Die Lesedauer der Ereignisse konnte im Mittel auf ein Zehntel verkürzt werden. Die Verarbeitung von Abfragen gestaltet sich dennoch schwierig. Die geforderte Verarbeitung von drei Millionen Ereignisse in einer Abfrage (Kapitel 3.1.2) konnte im Test nicht erreicht werden. Die Verarbeitungszeit von einer Minute, welche durch die maximale Bearbeitungszeit eines Webaufrufes in App Engine vorgegeben ist, reichte hierfür nicht aus. Durch begrenzten Speicher wird eine Weiterverarbeitung zusätzlich erschwert.

Mit einer Anpassung der Ebenenstruktur auf die erwartete Anzahl an Ereignisse könnten in einer praktischen Anwendung besserer Werte erreicht werden. Mit der Deaktivierung des nicht erfolgreichen Zwischenspeichers kann die Verarbeitung ebenfalls beschleunigt werden. Eine Erfüllung der Anforderungen wäre damit unter Umständen möglich. Die Skalierbarkeit des Systems bleibt jedoch eingeschränkt. Bei steigendem Datenvolumen gelangen die Möglichkeiten der Datenauswertung unweigerlich an ihre Grenzen.

5.3.2 Kosten

Die Betriebskosten für die Infrastruktur hängen stark von der Nutzung ab. Die monatlichen Kosten betragen für eine Instanz mit täglich zehn Betriebsstunden um die $24. Bei einer Auslastung gemäß der Anforderungsanalyse (Kapitel 3.1.2) werden mit den Evaluationsergebnissen durchschnittlich zwei bis drei Instanzen für die eingehenden Ereignisse abgeschätzt.

Die Kosten für den *Datastore* entfallen zum Großteil auf das Speichern von Ereignissen. Mit $108 pro Monat bei täglich einer Million Ereignissen, zuzüglich der Kosten für die Erstellung der Cluster, ist zu rechnen. Durch den Einsatz des Ereignispuffers, welcher sich für diese Auslastung erfolgreich einsetzten lässt, können diese Kosten jedoch deutlich geringer ausfallen.

Für Abfragen fallen durch die wenigen und kostengünstigeren Lese-Zugriffe nur geringe Gesamtkosten an. Bei der täglichen Abfrage von zehn Millionen Ereignissen aus Clustern bewegen sich die Kosten für die Zugriffe auf den *Datastore* bei etwa $0,05 pro Monat.

Die Hochrechnungen auf Grundlage der Anforderungen geben einen Überblick über anfallenden Kosten. Durch das Pay-per-use-Modell fallen für eine geringere Nutzung des Systems auch geringere Kosten an. Die entstehenden Kosten für den Betrieb auf Google App Engine sind damit, wie vorgesehen, direkt von der Ereignis- und Abfrageanzahl abhängig.

5.3.3 Business Intelligence auf Google App Engine

Das Verarbeiten eingehender Daten lässt sich bei einem BI-System auf App Engine sehr gut skalierend betreiben. Schwierigkeiten können bei der Echtzeitverarbeitung von Daten auftreten. Die Vorberechnung von Ergebnissen in periodischen Abständen wurde als wichtiger Baustein identifiziert, um in den von App Engine gegebenen Rahmenbedingungen erfolgreich große Datenmengen verarbeiten zu können. Da die Skalierungseigenschaften bei der Verarbeitung einer Abfrage begrenzt sind, muss der Detailgrad der Vorberechnungen mit der Datenmenge steigen. Nur damit bleibt es möglich Ergebnisse mit kurzer Verarbeitungszeit zu liefern.

Die Pflicht zur Vorberechnung von Ergebnissen schränkt die Flexibilität der Abfragen ein. Die Umsetzung eines Werkzeugs zur spontanen und flexiblen Erstellung von Auswertungen erscheint auf App Engine schwierig. Liegen die Ziele in der Überwachung und Analyse der Produkt- und Marktentwicklung, ist ein erfolgreicher Betrieb eines Systems auf App Engine möglich. In diesem Fall bietet die Cloud von Google Möglichkeiten für ein leichtgewichtiges und kostenorientiertes BI-System.

6 Verwandte Arbeiten

Die Betrachtung weitere BI-Systeme erlaubt eine Einordnung der Stärken und Schwächen, die sich auf App Engine ergeben. Für den Vergleich wird ein open-source-Projekt, ein kostenpflichtiges Produkt und eine Architektur zur verteilten Verarbeitung genauer betrachtet. Der Schwerpunkt liegt dabei auf der Skalierbarkeit und der Kostenstruktur.

6.1 Piwik

Piwik ist ein System zur Analyse des Verhaltens von Besuchern auf Webseiten. Durch die umfangreichen Integrationsmöglichkeiten[14] in Content- und Shop-Managementsystemen sowie Unterstützung von vielen Programmiersprachen und Frameworks, kann es eine Alternative zum klassischen BI-System sein. Die bei dem Besuch einer Seite ausgelösten Ereignisse enthalten dabei auch die Informationen: Was wird von wem wann gemacht? Weitere benutzerdefinierte Werte sind bei Piwik ebenfalls möglich [KT11, S. 3].

6.1.1 Aufbau

Piwik entstand aus der Idee eine open-source-Alternative von Google Analytics zu entwickeln[15]. Für den Betrieb von Piwik ist im Gegensatz zu Google Analytics, welche den Dienst in der eigenen Cloud betreiben, ein eigener Webserver erforderlich [KT11, S. 5]. Auf diesem werden PHP und MySQL benötigt [KT11, S. 6]. Dies stellt auch ein entscheidendes Abgrenzungsmerkmal, auf das die Entwickler besonderen Wert gelegt haben, dar: Die erhobenen Daten bleiben auf den Servern und damit unter der Kontrolle des Betreibers[15]. Bei Google Analytics könnten die Daten über das Surf-Verhalten der Besucher an Dritte weitergegeben oder selbst zu Marketingzwecken verwendet werden [KT11, S. 5].

6.1.2 Skalierbarkeit

Datenrate oder Datenmenge sind in Piwik nur durch den Server eingeschränkt. In der Dokumentation wird ab 10.000 Seitenaufrufe pro Tag ein dedizierter Webserver empfohlen. Bis mindestens 300.000 Seitenaufrufe pro Tag läuft Piwik optimal[16]. Ein genaueres Limit ist in der Dokumentation nicht angegeben. Die Zielsetzungen (Kapitel 3.1.2) scheinen nach diesen Angaben mit Piwik nur schwer erreichbar.

Ohne Anpassung der Einstellungen arbeitet Piwik in Echtzeit. Die Vorberechnungen, die die Echtzeitabfragen ermöglichen, werden alle zehn Sekunden ausgeführt. Für größere Webseiten wird empfohlen diesen Wert auf mehrere Stunden zu erhöhen. Für Auswertungen werden immer nur die Daten bis zum Zeitpunkt der letzten Vorberechnung berücksichtigt[17].

Für den Fall, dass ein einzelner Server nicht mehr ausreicht um Piwik erfolgreich zu betreiben, besteht die Möglichkeit Piwik auf mehrere Server zu verteilen. Eine volle Skalierung über alle Komponenten kann dabei nicht erreicht werden. Die eintreffenden Ereignisse können auf mehrere Instanzen aufgeteilt werden. Auch die Abfrage der Daten kann auf mehrere Piwik-Instanzen verteilt werden. Da alle Piwik-Instanzen auf die gleichen Daten zugreifen müssen, kann die MySQL-Instanz nicht parallel betrieben werden. Hinzu kommt, dass diverse Daten, die bei einen Ein-Server-Betrieb als Dateien abgelegt werden, nun in der Datenbank gespeichert werden müssen[18]. Dadurch wird diese zusätzlich beansprucht. Die Datenbank stellt somit ein Nadelöhr beim Betrieb dar. In der Piwik-Community wurde der Vorschlag zur Abstraktion der Datenhaltungsebene bereits vorgetragen[19]. Mit diesem wäre eine Anbindung an andere Datenbanken möglich.

[14] http://piwik.org/integrate
[15] http://de.piwik.org/uber
[16] http://piwik.org/docs/optimize
[17] http://piwik.org/faq/general
[18] http://piwik.org/faq/new-to-piwik
[19] http://forum.piwik.org/read.php?3,96691,96691

6.1.3 Kosten

Piwik wird als kostenfreies Produkt angeboten. Bei der Inbetriebnahme fallen lediglich allgemeine Kosten für die Hardware an. Die laufenden Kosten beschränken sich ebenfalls auf die Betriebskosten der Infrastruktur.

6.2 Microsoft SQL Server

Der Microsoft SQL Server bietet neben der klassischen SQL-Datenbank auch viele Werkzeuge für die Datenintegration und -analyse sowie für die Visualisierung. Mit diese Anwendungen kann ein umfangreiches BI-System aufgebaut werden. Für Nutzer des Microsoft Office-Paketes bieten die Anwendungen Integrationsmöglichkeiten an, so dass zum Beispiel eine Auswertung mit den Pivot-Tabellen in Excel stattfinden kann [Sch09, S. 73].

6.2.1 Aufbau

Mit den Integration Services wird versucht verschiedene Datenquellen und Formaten eine Schnittstelle zum BI-System zu geben. Der Service bietet automatisierte Importfunktionen aus Warenwirtschaftssystemen, Datenbanken und Dateien unterschiedlichster Formate. Als Ziel können diese strukturiert in einer SQL-Server-Datenbank abgelegt werden[CF13, S. 295].

Die Analysis Services des SQL Server kann für OLAP[20] und Data Mining genutzt werden. Multidimensionale Datenwürfel können eine explorative Darstellung der Daten geben. Aufgrund von Vorberechnungen können diese Datenwürfel ohne weitere Wartezeit in einem Frontend visualisiert werden. Beim Data Mining werden zwei- und mehrdimensionale Datenmodelle unterstützt [Sch09, S. 34].

Über die Reporting Services können individuelle Berichte erstellt werden. Dies erlaubt bei einer gleichbleibenden Struktur die fortlaufende Überwachung der wichtigsten Kennzahlen im Unternehmen [CF13, S. 591].

6.2.2 Skalierbarkeit

Der komplette Datenbestand auf denen diese Dienste aufbauen, liegt in einer SQL-Server-Instanz. Eine Skalierbarkeit kann durch Lastverteilung über mehrere Berichts- und Analyseserver erreicht werden. Bis zu dem Punkt, an dem für jeden Dienst und Service ein eigener Server eingesetzt wird, ist so eine Skalierung möglich [CF13, S. 375f]. Die Datenbank selbst lässt sich nicht auf mehrere Rechner verteilen. Durch eine Spiegelung der Datenbank können aber Lesezugriffe auf mehrere Instanzen verteilt werden. Für das Abgleichen der Daten auf den Instanzen entsteht ein Synchronisationsaufwand [CF13, S. 378].

6.2.3 Kosten

Beim SQL Server von Microsoft handelt es sich um ein kommerzielles Produkt. Abgesehen von der Express Edition, welche einen eingeschränkten Funktionsumfang hat, entstehen für die Benutzung Lizenzkosten [CF13, S. 375]. Da zusätzlich auch entsprechende Hardware benötigt wird, hat dieses umfassende BI-System bei einem Einstieg hohe Investitionskosten.

[20] Online Analytical Processing (siehe Kapitel 2.1)

6.3 Apache Hadoop

Apache Hadoop ist ein open-source-Projekt der Apache Software Foundation zur Entwicklung von Software zur verteilten Verarbeitung von Daten[21].

6.3.1 Aufbau

Die Hadoop-Bibliothek besteht unter anderem aus dem Hadoop Distributed File System (HDFS)[21]. Dieses Dateisystem bildet die Grundlage von Hadoop. Das HDFS ist ein verteiltes Dateisystem und ist für den Betrieb auf kostengünstiger Standardhardware ausgelegt [Rus13]. Es zeichnet sich durch einen hohen Datendurchsatz aus und kann auch mit großen Dateien und Datenmengen umgehen. Durch ein fehlertolerantes Design, das automatische Wiederherstellungsfunktionen besitzt, wird eine hohe Verfügbarkeit erreicht [Bor07].

Verteilte Berechnungen können in Hadoop mit MapReduce ausgeführt werden. MapReduce ist ein von Google veröffentlichtes Programmiermodell um große Datenmengen parallel zu verarbeiten [SM12, S. 131]. MapReduce unterteilt sich in die zwei Ausführungsschritte „map“ und „reduce“. Beide können jeweils parallel aufgeführt werden. Im ersten Schritt werden die Daten aufbereitet, im Zweiten zu einem Gesamtergebnis zusammengeführt [Tay10, S. 2].

Es gibt zahlreiche Projekte, auch von Drittanbietern, die Hadoop um zusätzliche Funktionen erweitern oder den Zugriff auf HDFS und MapReduce vereinfachen. Die Betrachtung beschränkt sich an dieser Stelle auf HBase[22] und Hive[23], welche es ermöglichen, Daten strukturiert abzulegen und gezielt zu durchsuchen. Beide stehen ebenfalls als open-source-Projekte von Apache zur freien Benutzung.

Auf HDFS aufbauend ist Apache HBase. HBase ist eine fehlertolerante und skalierbare Datenbank. Eine Abfragesprache wie SQL wird von HBase nicht unterstützt. Bei der Entwicklung wurde die BigTable Datenbank von Google zum Vorbild genommen[Tay10, S. 3]. Google benutzt die BigTable-Technologie auch für den *Datastore* aus App Engine [Sch13, S. 137]. Als kommerzielle Erweiterung von HBase bietet Intellicus Technologies verschiedene Analysetools[24] an. Diese bieten die Möglichkeit Ad-hoc-Auswertungen in mehrdimensionalen Datenwürfeln vorzunehmen oder individuelle Berichte anzulegen. Beide Tools verfügen über verschiedene Methoden zur Visualisierung.

Für einen komfortableren Zugriff auf die Daten sorgt das Data-Warehouse Framework Apache Hive. Mit Hive können Daten über Tabellen und Spalten verwaltet werden. Zum Einsatz kommt dabei eine SQL-ähnliche Abfragesprache namens HiveQL. Da Hive-Anfragen als Jobs in MapReduce abgearbeitet werden, sind Echtzeitanfragen nicht möglich [Tay10, S. 2]. Da schnelle Anfragen nahe Echtzeit im BI eine wichtige Rolle spielen, kann Hadoop für diese Fälle um Impala[25] von Cloudera erweitert werden. Damit ist es möglich SQL-Anfragen in Echtzeit an HBase zustellen. Eine SQL-Schnittstelle kann die Anbindung weiterer Analyse- und Darstellungstools an Hadoop erheblich vereinfachen.

Durch den modularen Aufbau können die vorhandenen Systeme flexibel eingesetzt und um weitere Komponenten ergänzt werden. Einschränkende Bedingungen bei der Abfrage von Daten, wie es sie beim *Datastore* von App Engine gibt, sind bei Hadoop nicht vorhanden. Ob sich

[21] http://hadoop.apache.org
[22] http://hbase.apache.org
[23] http://hive.apache.org
[24] http://www.intellicus.com/product/features.htm
[25] http://www.cloudera.com/content/cloudera/en/products/cdh/impala.html

diese Flexibilität auf die Ausführungsdauer einer Abfrage auswirkt, kann an dieser Stelle nicht beurteilt werden.

6.3.2 Skalierbarkeit

Eine Skalierung findet bei Hadoop über eine Erweiterung des Systems um weitere Server statt. Da die Hadoop-Architektur speziell auf die verteilte Verarbeitung ausgelegt ist, dürfte eine hohe Skalierbarkeit erreichbar sein. Entstehende Lastspitzen müssen jedoch von vorn herein berücksichtigt werden, da im Betrieb keine Skalierung möglich ist. Im Vergleich zu der dynamischen Skalierung in der Cloud liegt ein Hadoop-System hier im Nachteil. Dabei sollte beachtet werden, in wie weit diese dynamische Skalierung überhaupt benötigt wird.

6.3.3 Kosten

Hadoop bietet durch die Möglichkeit des Betriebes auf kostengünstiger Hardware, Potential um ein kostengünstiges BI-System betreiben zu können. Im Vergleich zu Google App Engine wird bei Hadoop höherer Anpassungs- und Konfigurationsaufwand benötigt[Rus13]. Auch bei der Erweiterung des Systems um Speicher- und Rechenkapazitäten ist ein Mehraufwand notwendig.

Um diesen hohen Aufwand bei der Installation und Konfiguration der Hadoop-Komponenten zu verringern, kann auch auf kommerzielle Produkte verschiedener Anbieter zurückgegriffen werden. Die Technologien von Hadoop werden dort in eigene System eingebettet oder es werden Paketlösungen die eine anwendungsorientierte Benutzung erlauben angeboten. Ein Vorteil bieten diesen Produkte insbesondere bei fehlendem Knowhow, da die Anbieter im Allgemeinen auch Support anbieten [Rus13].

6.4 Vergleich

Im Vergleich zeigen sich die Stärken und Schwächen der Systeme. Die tabellarische Darstellung (Abbildung 15) bietet einen Überblick.

System/Architekur	Funktionen	Skalierbarkeit	Kosten
Piwki	0	−	+
Microsoft SQL Server	+	−	−
Apache Hadoop	+	0	−
Google App Engine	−	+	+

Abbildung 15: Bewertung der Systeme im Vergleich

Funktionen: Während Piwik alle einfachen Funktionen mit sich bringt, sind beim Microsoft SQL Server weitere umfangreiche Services in einem leicht erweiterbaren Aufbau integriert. Für Hadoop existieren, durch seinen offenen Quellcode, eine Vielzahl von zusätzlichen Anwendungen und Werkzeugen, die auf der Hadoop-Struktur aufbauen. Die

Funktionen können mit etwas Installations- und Konfigurationsaufwand erweitert werden. Auf App Engine müssen die Funktionalitäten schon beim Entwurf eines Systems spezifiziert werden.

Skalierbarkeit: Piwik und das System von Microsoft basieren jeweils auf einer SQL-Datenbank. Die Skalierbarkeit wird durch diese beschränkt. Hadoop, als verteiltes System, ist für eine extreme Leistungsfähigkeit ausgelegt. Eine Skalierung kann durch Hardware-Erweiterung erreicht werden. Darüber hinausgehende, kurzzeitige Leistungssteigerungen sind nicht möglich. App Engine ist durch seine schnelle und autonome Skalierung positiv zu bewerteten.

Kosten: Für Piwik fallen lediglich Kosten für die Hardware an. Bei Microsoft müssen zusätzlich noch Gebühren für die Nutzungslizenzen einkalkuliert werden. Da die Inbetriebnahme eines Hadoop-Systems erhebliches Knowhow erfordert, können neben Kosten für die umfangreiche Hardware auch noch Kosten für kommerzielle Erweiterungen oder Supportleistungen hinzukommen. Geringe anfängliche Investitionskosten und nutzungsabhängige Betriebskosten zeichnen ein BI-System auf App Engine aus.

7 Zusammenfassung und Ausblick

Dieses abschließende Kapitel gibt einen Überblick über die Ergebnisse der Arbeit. Im nachfolgenden Ausblick werden kurz weitergehende Fragestellungen, die sich aus der Arbeit ergeben, aufgezeigt.

7.1 Zusammenfassung

In der Einleitung wurden Motivation und Ziele der Arbeit dargelegt. Ziel ist es die Möglichkeiten eines Business Intelligence Systems auf Google App Engine über eine prototypische Umsetzung aufzuzeigen. Die Motivation liegt in der Berücksichtigung der besonderen Anforderungen, die sich bei stark wachsenden Unternehmen ergeben. Als wichtige Aspekte für ein solches System wurden Skalierbarkeit und eine variable Kostenstruktur erörtert.

Im zweiten Kapitel wurde Business Intelligence und Google App Engine vorgestellt. Business Intelligence kann als Gesamtansatz zur Unterstützung des Managements bei seiner planenden und steuernden Tätigkeit gesehen werden. Bei App Engine handelt es sich um eine Plattform zum Betrieb von Anwendungen auf der Infrastruktur von Google.

Der Aufbau des in dieser Arbeit entwickelnden Systems wird im dritten Abschnitt beschrieben. Das ereignisorientierte System lässt sich in verschiedene Bereiche unterteilen. Eingehende Ereignisse können über einen Puffer zwischengespeichert werden. Die so erreichte gemeinsame, periodische Abarbeitung aller in diesem Zeitraum angefallen Ereignisse bietet Vorteile bei der persistenten Datenhaltung. Die Datenhaltung erfolgt durch den in App Engine angebotenen *Datastore*. Mit einer Vorbrechung von Ereignis-Clustern, aus Ereignissen die oft gemeinsam abgefragt werden, wird eine schnellere Weiterverarbeitung erreicht. Über Schnittstellen können Abfragen an das System gestellt werden. Es besteht die Möglichkeit die Ergebnisse als Grafik darstellen zu lassen.

Das vierte Kapitel der Arbeit geht detailliert auf die prototypische Umsetzung des Systementwurfs ein. Die Umsetzung in Java wurde für einen flexibel konfigurierbaren Einsatz ausgelegt. Über verschiedene Webschnittstellen lassen sich Ereignisse in das System übertragen, welche

dann über Abfragen ausgewertet werden können. Die Bearbeitung solcher Abfragen erfolgt dabei verschachtelt, über mehrere Datenebenen. Eine auf App Engine optimierte Umsetzung der Datenebenen ermöglicht eine schnelle Verarbeitung der Abfragen.

Die Evaluation zeigte im simulierten Einsatz die Stärken und Schwächen des umgesetzten Systems. Als Ergebnis kann festgehalten werden, dass mit einem Business Intelligence System auf App Engine, bei eingehende Ereignisse, eine schnelle und hohe Skalierbarkeit möglich ist. Durch technische Beschränkungen in App Engine ist eine hohe Skalierbarkeit bei Abfragen schwieriger als bei eingehenden Ereignissen zu erreichen. Auswertungen auf großen Datenbeständen erfordert hier vorberechnete Teilergebnisse, welche die Abfrageflexibilität reduzieren. Durch Kenntnis der kritischen Faktoren ließen sich diese beim Entwurf des Business Intelligence Systems entsprechend berücksichtigen.

Eine vergleichende Betrachtung weitere Business Intelligence Systeme wurde im sechsten Kapitel durchgeführt. Es zeigte sich, dass der Funktionsumfang der Vergleichssysteme deutlich größer ist. Flexible Strukturen bieten mehr Anwendungsszenarien als bei einem System auf App Engine. Bei den Kosten zeigte sich das Pay-per-use-Modell von App Engine als erfolgreiches Mittel zur Erreichung einer variablen Kostenstruktur. Die schnelle und dynamische Skalierung von App Engine ist mit einem anderen System nur schwer zu erreichen.

7.2 Ausblick

In der Arbeit wurden Vorberechnungen als wichtiger Faktor für Auswertungen identifiziert. In wie weit sich die Flexibilität von Abfragen unter Verwendung dieser Erkenntnisse erhöhen lässt, ist noch offen. Ein möglicher Ansatz wäre die Umsetzung eines flexiblen, vom Anwender individualisiertes Vorberechnungsmodell. Ebenso könnte durch die Umsetzung von Stichprobenanalysen Auswertungen auf größeren Datenbeständen ermöglicht werden.

Die detaillierte Betrachtung der Eigenschaften einer Cloud-Plattform ist für den Entwurf eines Systems vorteilhaft. Die Arbeit betrachtet ein Business Intelligence System auf Grundlage der Cloud-Plattform von Google. Da es weitere Anbieter von Cloud-Plattformen gibt, wäre ein Business Intelligence System auch dort denkbar. Ein prototypische Entwicklung und Evaluation auf einer anderen Plattform, könnte weitere Lösungsansätze für ein skalierendes Business Intelligence System in der Cloud aufzeigen.

Literatur

[Bor07] Borthakur, Dhruba: *The Hadoop Distributed File System: Architecture and Design*. Technischer Bericht, Apache Software Foundation, 2007.

[CF13] Caesar, Daniel und Michael Friebel: *Schnelleinstieg Microsoft SQL Server 2012: für Administratoren und Entwickler*. Galileo Press, 2. Auflage, 2013.

[Fie00] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures*. Doktorarbeit, University of California, 2000.

[KBM10] Kemper, Hans-Georg, Henning Baars und Walid Mehanna: *Business Intelligence – Grundlagen und praktische Anwendungen: Eine Einführung in die IT-basierte Managementunterstützung*. Vieweg + Teubner, 3. Auflage, 2010.

[KT11] Karg, Moritz und Sven Thomsen: *Hinweise und Empfehlungen zur Analyse von Internet-Angeboten mit „Piwik“*. Technischer Bericht, Unabhängies Landeszentrum für Datenschutz Schleswig-Holstein, 2011.

[RSA13] Rausch, Peter, Alaa F. Sheta und Aladdin Ayesh: *Business Intelligence and Performance Management: Theory, Systems and Industrial Applications*. Springer, 2013.

[Rus13] Russom, Philip: *Integrating Hadoop into Business Intelligence and Data Warehousing*. Technischer Bericht, The Data Warehousing Institute, 2013.

[San12] Sanderson, Dan: *Programming Google App Engine*. O'Reilly, 2. Auflage, 2012.

[Sch09] Schrödl, Holger: *Business Intelligence mit Microsoft SQL Server 2008: BI-Projekte erfolgreich umsetzen*. Hanser, 2. Auflage, 2009.

[Sch13] Schwanengel, Anna: *Cloud Datenspeicher- und Datenbank-Lösungen*. Praxis der Informationsverarbeitung und Kommunikation, 36(2):133–140, 2013.

[Sdc] *Definition: scalability*. `http://searchdatacenter.techtarget.com/definition/scalability`, letzter Abfruf: 19.09.2013.

[SM12] Saecker, Michael und Volker Markl: *Big Data Analytics on Modern Hardware Architectures: A Technology Survey*, Kapitel 6, Seiten 125–149. Springer, 2012.

[Tay10] Taylor, Ronald C.: *An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics*. In: *The 11th Annual Bioinformatics Open Source Conference (BOSC)*, 2010.

Abbildungsverzeichnis

1 Instanz mit mehreren Request-Handlern 8
2 Kosten der Datenoperationen 12
3 Datenoperationen je API-Zugriff 12
4 Ereignis mit Attributen 13
5 Schema des Systemaufbaus 16
6 Ebenenstruktur 18
7 Aufbau eines Ereignis-Objekts 21
8 Aufbau eines Cluster-Objekts 22
9 Ablauf mit Ereignispuffer 24
10 UML-Klassendiagramm 26
11 Highcharts Beispielgrafik 27
12 Verarbeitungsablauf Abfragen 29
13 Testverlauf: Schreiben von Ereignissen 31
14 Testverlauf: Vergleich mit Ereignispuffer 32
15 Bewertung der Systeme im Vergleich 38

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 30.09.2013