Universität Leipzig

Institut für Informatik

Abteilung Datenbanken

Problemseminar NoSQL-Datenbanken

Google Megastore - eine skalierbare relationale Datenbank

Autor:
Falk Stehmann

Betreuer:
Dr. Andreas Thor

23. März 2012

Inhaltsverzeichnis

1 Einleitung **1**

2 Scalable Relational Databases **2**

2.1 Motivation 2

2.2 Eigenschaften und Einordnung 3

3 Google Megastore **5**

3.1 Design von Megastore 5

3.1.1 Datenmodell 5

3.1.2 Transaktions- und Nebenläufigkeitskontrolle 9

3.2 Verfügbarkeit und Skalierbarkeit 11

3.2.1 Replikation 11

3.2.2 Partitionierung 13

4 Zusammenfassung **16**

1 Einleitung

Der Stellenwert von Datenbanken in der heutigen Informationstechnologie und Dienstleistungsbranche steigt ständig. Durch die zunehmende Migration von Desktopanwendungen in die Cloud und die Verbreitung von sozialen Netzwerken erhöht sich das Datenaufkommen sehr stark. Um in dem schnell wachsenden Markt des User-created Content mithalten zu können, ist die rasche Entwicklung von neuen Features entscheidend. Deshalb werden nicht nur die Anforderungen an Skalierbarkeit und Performanz der Datenbanken immer größer, sondern sie sollen ebenso eine benutzerfreundliche Anwendungsentwicklung ermöglichen. Diese Eigenschaften zu vereinen bildet eine Herausforderung für das Design moderner Datenbanken.
Zu Beginn dieser Arbeit wird ein allgemeiner Ansatz vorgestellt, wie dieses Problem gelöst werden könnte. Im folgenden Kapitel wird am Beispiel von Google Megastore gezeigt, wie dieses Modell umgesetzt werden kann.

2 Scalable Relational Databases

Es gibt eine Vielzahl von Designansätzen für Datenbanken die darauf ausgelegt sind, einfache OLTP-Anwendungen über viele Server zu verteilen. Ursprünglich wurde dies durch Web 2.0 Anwendungen motiviert, in denen Tausende oder Millionen von Nutzern sowohl Update als auch Read Operationen durchführen, anders als in traditionellen DBMS und Data Warehouses.
Wir beschäftigen uns hier mit dem Ansatz der s.g. *skalierbaren relationalen Datenbanken* (SRDBMS).

2.1 Motivation

Die Migration vieler Desktop-Anwendungen in das Web stellt neue Ansprüche an die zu grunde liegenden Speichersysteme. Dienstleistungen wie Email oder soziale Netzwerke wachsen exponentiell und testen die Grenzen der aktuellen Infrastruktur. Dem Speicherbedarf dieser Anwendungen gerecht zu werden ist u.a. auf Grund der folgenden Anforderungen sehr schwierig:

1. *hohe Skalierbarkeit*
   Das Internet bringt eine Vielzahl von möglichen Nutzern mit sich, gute Skalierbarkeit ist also eine Grundvorraussetzung an das Datenbanksystem.

2. *schnelle Entwicklungszeit*
   Der Markt der Anwendungen entwickelt sich rasant weiter. Die schnelle Entwicklung neuer Features und eine kurze "time-to-market" ist wichtig, um für Nutzer attraktiv zu bleiben.

3. *kurze Latenzzeiten*
   Eine schnelle Reaktion auf Anfragen der Nutzer wird von Vielen erwartet und muss somit gewährleistet werden können.

4. *konsistente Sicht auf die Daten*
   Änderungen sollen für den Nutzer sofort sichtbar sein und dauerhaft erhalten bleiben. Das Verlorengehen von Änderungen in einem geteilten Dokument ist ein Beispiel für eine sehr unangenehme Nutzererfahrung.

5. *hohe Verfügbarkeit*
   Eine permanente Erreichbarkeit einer Dienstleistung wird ebenfalls vorrausgesetzt. Es müssen kleine Fehler, wie das Ausfallen einer Festplatte, als auch größere Fehler, die ganze Datencenter betreffen, abgefangen werden können.

Diese Anforderungen sind widersprüchlich. Einerseits werden typische Eigenschaften relationaler Datenbanken (RDBMS) benötigt. Dazu zählen die zahlreichen Features, die sie bieten, um die Anwendungsentwicklung zu vereinfachen und die globale Konsistenz. Allerdings ist es schwierig diese Datenbanken für Millionen von Nutzern skalieren zu lassen.
Andererseits werden auch Eigenschaften der NoSQL Datenbanken benötigt, welche diese hohe Skalierbarkeit bieten. Allerdings wird die Anwendungsentwicklung komplizierter, auf Grund der schwächeren Konsistenzmodelle und der begrenzten API.

2.2 Eigenschaften und Einordnung

Dieser beschriebene Konflikt soll mit den *skalierbaren relationalen Systemen* gelöst werden. Das Ziel ist es, einen Kompromiss zwischen den relationalen Datenbanken und den NoSQL Datenbanken zu finden.
Von den RDBMS werden typischerweise das vordefinierte Schema, das SQL Interface und die ACID Transaktionen übernommen, letztere jedoch evtl. mit Einschränkungen. Um eine bessere Skalierbarkeit zu erreichen, wird auf aufwändige Features wie die globale Konsistenz, eine hohe Datenunabhängigkeit und eine umfangreiche Unterstützung von Joins meist verzichtet. Da viele Web-Anwendungen nur einfache Nutzungsformen haben, im Vergleich zu OLAP o.ä., kann auf diese Eigenschaften meist verzichtet werden.
Um die Skalierbarkeit zu erhöhen wird weiterhin versucht, den Umfang einer Transaktion bzw. Operation auf einen Knoten der Datenbank zu begrenzen. Operationen/Transaktionen über viele Knoten, bespielsweise ein Join über mehrere Tabellen, können sehr ineffizient in der Ausführung sein. Bei NoSQL Datenbanken sind diese Operationen/Transaktionen teilweise gar nicht oder nur sehr schwer möglich. Für SRDBMS gibt es keinen Grund diese Funktion

nicht zu unterstützen. Der Nutzer muss lediglich den Preis dafür bezahlen, wenn die Operation/Transaktion mehrere Knoten überspannt, beispielsweise den Overhead des 2-Phasen-Commits einer Transaktion.
Weitere Vergleichspunkte sind am Beispiel von Google Megastore als SRDBMS in Abb. 1 dargestellt.

NoSQL	**Megastore**	**RDBMS**
Minimal features	Scalable features	Full-featured
Highly scalable	Highly scalable	Medium scale with effort
PK lookup and scan	Indexes, scans, physical clustering	Storage abstraction, complex query planning and execution
Limited/eventual consistency	Partitioned consistency	Global consistency

Abbildung 1: Vergleich typischer Eigenschaften von NoSQL Datenbanken und RDBMS mit Megastore. [3]

3 Google Megastore

Google Megastore ist eine skalierbare relationale Datenbank. Es vereinigt die Skalierbarkeit einer NoSQL Datenbank mit der Einfachheit einer relationalen Datenbank. Megastore benutzt synchrone Replikation um eine hohe Verfügbarkeit und eine konsistente Sicht auf die Daten zu ermöglichen. Grob gesagt bietet es serialisierbare ACID Eigenschaften über mehrere Replikate mit niedrigen Latenzzeiten. Wie diese Eigenschaften zu Stande kommen wird im Folgenden behandelt.

3.1 Design von Megastore

In diesem Abschnitt wird zuerst beschrieben, wie Daten in Megastore defniniert und gespeichert werden können. Danach wird darauf eingagangen, welche Techniken eingesetzt werden, um den logischen Einbenutzerbetrieb zu ermöglichen.

3.1.1 Datenmodell

Megastore besitzt ein Datenmodell, welches zwischen den abstrakten Tupeln einer relationalen Datenbank und den Zeile-Wert Paaren einer NoSQL Datenbank liegt.
Wie bei RDBMS besitzt das Datenmodell ein streng typisiertes Schema. Jedes Schema besteht aus einer Menge von Tabellen. Diese bestehen aus einer Menge von Entities, welche wiederum eine Menge von Eigenschaften (*Properties*) haben. Properties sind benannte und typisierte Werte. Sie können optional, notwendig (*required*) oder wiederholt (*repeated*) sein. Die repeated Eigenschaft ermöglicht eine Liste von Werten in einer einzigen Property. Das Datenmodell ist somit nicht normalisiert. Alle Entities einer Tabelle haben die gleiche Menge von Eigenschaften. Um den Primärschlüssel der Entities in einer Tabelle zu bilden, wird eine Folge von Properties festgelegt. Der Primärschlüssel muss natürlich einzigartig sein innerhalb einer Tabelle. Abbildung 2 zeigt ein einfaches Schema für eine Photodatenbank.

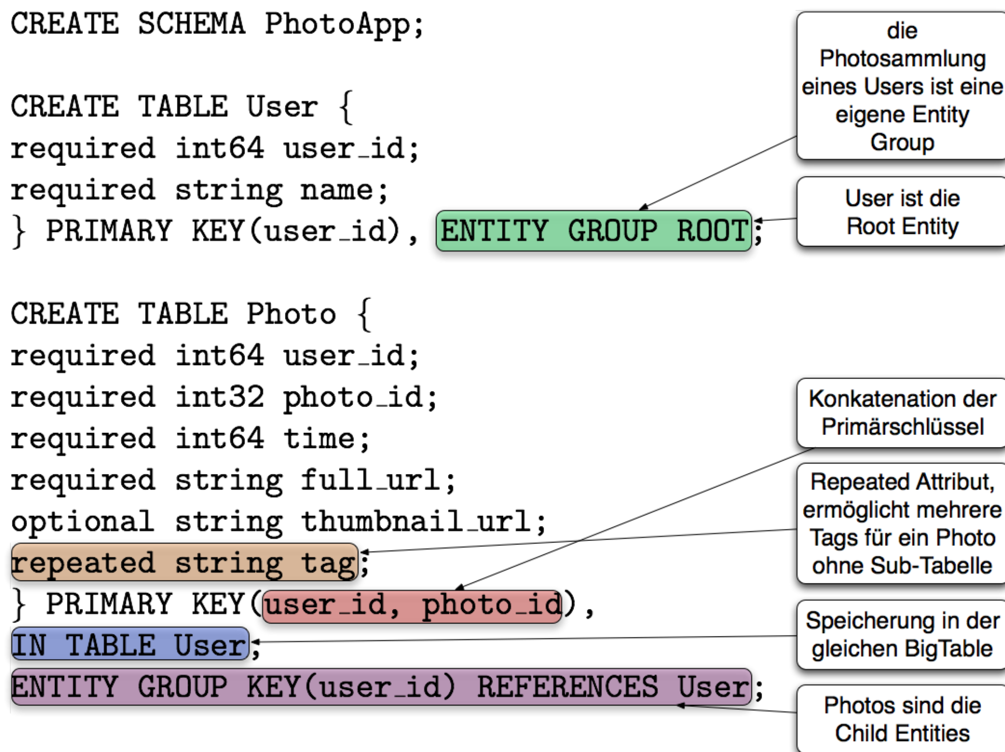

Abbildung 2: Beispielhafte Defnition eines Schemas für eine einfache Photodatenbank

Tabellen in Megastore sind entweder *Entity Group Root Tables* oder *Child Tables*. Jede Child Table muss auf genau eine Root Table (auch Root Entity genannt) verweisen, dargestellt durch die `ENTITY GROUP KEY` Annotation in Abb. 2. Eine Entity Group besteht aus einer Root Entity und allen Entitäten in Child Tables, die auf sie verweisen. Eine Megastore Instanz kann mehrere Root Tables haben, was verschiedene Klassen von Entity Groups ermöglicht. Im Beispiel in Abb. 2 ist die Photosammlung jedes Users eine eigene Entity Group. Die Root Entity ist User und die Photos sind die Child Entities. In Abb. 3 ist das in Megastore entstandene Schema dargestellt.

Indexierung Sekundäre Indexe können in Megastore über jede Liste von Properties einer Entity deklariert werden. Dabei wird zwischen zwei Arten der Indizierung unterschieden: *lokaler Index* und *globaler Index*.

user_id	name
101	John
102	Mary

(a) Root Table

user_id	photo_id	time	full_url	thumbnail_url	tag
101	501	2009	...	...	Vacation, Holiday, Paris
101	502	2010	...	...	Office, friends, Pubs
102	600	2009	...	...	Office, Picnic, Paris
102	601	2011	...	...	Birthday, Friends

(b) Child Table

Abbildung 3: Schema in Megastore mit einigen beispielhaften Werten

Ein lokaler Index dient zum Finden von Daten innerhalb einer Entity Group und wird für jede Entity Group seperat behandelt. Die Einträge des lokalen Index werden in der zugehörigen Entity Group gespeichert. Sie werden atomar und konsistent mit den Daten der Entity Group geändert. In Abb. 4(a) ist ein Beispiel für einen Lokalen Index dargestellt.
Ein globaler Index umfasst mehrere Entity Groups. Er wird genutzt um Entities zu finden, ohne vorher zu wissen zu welcher Entity Group sie gehören. Der Index `PhotosByTag` in Abb. 4(b) ist ein globaler Index und kann Photos mit einem gegebenen Tag finden, ohne zu wissen welchem User sie gehören. Globale Index Scans lesen die Daten vieler Entity Groups, aber können keine Aktualität aller Änderungen garantieren.

```
CREATE LOCAL INDEX PhotosByTimeON Photo(user_id, time);
```

(a) Lokaler Index

```
CREATE GLOBAL INDEX PhotosByTag  ON Photo(tag) STORING (thumbnail_url);
```

(b) Globaler Index

Abbildung 4: Beisielhafte Deklaration von sekundären Indexen

Megastore bietet einige weitere Features für Indexe:

Storing Um Daten über den Index zu erreichen, sind normalerweise zwei Schritte notwendig. Zuerst wird im Index nach den gesuchten Primärschlüsseln gesucht, um danach auf die Entities zuzugreifen. Mit Hilfe des Schlüsselworts `storing` können Teile der Daten von Entities direkt im Index gespeichert werden. Das ermöglicht einen wesentlich schnelleren Zugriff auf diese Daten beim Lesen, da der zweite eben beschriebene Schritt entfällt. Der Index `PhotosByTag` in Abb. 4(b) speichert die thumbnail URL, um ohne einen extra Lookup darauf zugreifen zu können.

Repeated Megastore bietet die Möglichkeit, repeated Properties zu indexieren. `PhotosByTag` ist ein repeated Index. Jeder Eintrag im Feld Tag erzeugt einen neuen Eintrag im Index für das Photo. Das ist nebenbei auch eine effiziente Alternative für Sub-Tabellen.

Inline Inline Indexe ermöglichen die Denormalisierung von Daten aus einer Quell-Entity in eine Ziel-Entity. Die Indexeinträge der Quell-Entity erscheinen in einer virtuellen Spalte ("repeated") in der Ziel-Entity. Ein Inline Index kann auf jeder Tabelle erstellt werden, die über einen Fremdschlüssel auf eine andere Tabelle verweist. Dazu wird der erste Teil des Primärschlüssels der Ziel-Entity als erster Teil des Index benutzt. Weiter müssen die Daten in der BigTable Instanz des Ziels gespeichert werden. Inline Indexes sind nützlich um Teile von Daten aus Child Entities in den Parents zu speichern, um schneller darauf zugreifen zu können.

Speicherung in Google BigTable Zur effizienten Speicherung der Daten kommt in Megastore Google BigTable als zugrundeliegende NoSQL Datenbank zum Einsatz. BigTable bietet einige Eigenschaften, die für Megastore von essenzieller Bedeutung sind, darunter die Sortiertheit und die Mehrdimensionalität.
Die Spaltennamen in BigTable ergeben sich aus der Konkatenation vom Name der Tabelle in Megastore und der Property. Dadurch können Entities ver-

schiedener Megastore Tabellen in der gleichen BigTable gespeichert werden. Dieses Verhaten kann durch die `IN TABLE` Anweisung hervorgerufen werden, zu sehen in Abb. 2.

Row Key	User.name	Photo.time	Photo.tag	Photo URL
101	John			
101,501		2009	Vacation, Holiday, Paris	...
101,502		2010	Office, friends, Pubs	...
102	Mary			
102,600		2009	Office, Picnic, Paris	...
102,601		2011	Birthday, Friends	...

Abbildung 5: Die Werte aus Beispiel in Abb. 3 in BigTable

Abb. 5 zeigt wie die Daten unserer Photodatenbank in BigTable aussehen könnten. Durch die Konkatenation der Primärschlüssel und der Sortiertheit von BigTable werden Daten die zusammengehören auch physisch geclustert. Das wird auch als *Pre-Joining* bezeichnet.
In der BigTable Row der Root Entity werden Metadaten für Transaktionen und Replikation gespeichert, einschließlich des Logs. Durch diese Speicherung der Daten innerhalb einer BigTable Row können sie atomar durch eine einzige BigTable Transaktion geändert werden.

3.1.2 Transaktions- und Nebenläufigkeitskontrolle

Jede Entity Group in Megastore funktioniert wie eine Art "mini-Datenbankünd bietet serialisierbare ACID Eigenschaften. Eine Transaktion schreibt ihr Änderungen in das *write-ahead Log* der Entity Group, welche daraufhin auf die Daten angewendet werden. BigTable ermöglicht es mehrere Werte in der gleichen Zeile/Spalte mit unterschiedlichem Zeitstempel zu speichern, indem eine Zeit-Dimension angelegt wird. Das wird genutzt um *Multiversion Concurrency*

Control (MVCC) zu implementieren.
Wenn Änderungen einer Transaktion angewendet werden, schreibt Megastore die Werte an die Position mit dem Zeitstempel ihrer Transaktion. Beim Lesen wird der Zeitstempel der letzten vollständig abgeschlossenen Transaktion benutzt, um keine partielle Änderungen zu lesen. Somit blockieren sich lesende und schreibende Zugriffe nicht. Megastore bietet drei Arten von Lesekonsistenz:

current
: Werden immer innerhalb einer Entity Group durchgeführt. Megastore warted bis alle Writes die vorher committed wurden abgeschlossen sind, und liest dann vom Zeitstempel der letzten Transaktion die commited wurde.

snapshot
: Auch diese Reads finden nur innerhalb einer Entity Group statt. Es wird vom Zeitstempel der letzten vollständig angewendeten Transaktion gelesen, auch wenn es committed Transaktionen gibt die noch nicht angewendet wurden.

inconsistent
: Hier wird der Status des Logs ignoriert und direkt der letzte bekannte Wert gelesen.

Eine schreibende Transaktion beginnt immer mit einem current Read um die nächste mögliche Position im Log zu erfahren. Die Commit-Operation sammelt alle Änderungen in einem Log Eintrag, weist ihm einen höheren Zeitstempel als dem der vorherigen Einträge zu und hängt ihn mittels *Paxos* an das Log an. Das Paxos Protokoll benutzt *optimistic concurrency*: wenn mehrere Writes auf eine Log Position zugreifen wird nur eines gewinnen. Die anderen bemerken dies, brechen die Operation ab und starten sie erneut. Eine Transaktion durchläuft also folgenden Zyklus:

1. Read
: Finde den Zeitstempel und die Log Position der letzten committed Transaktion.

2. Anwendungslogik
Lese von BigTable und sammle die Änderungen in einem Log Eintrag.

3. Commit
Nutze das Paxos Protokoll um Übereinstimmung zum Anhägen des Eintrags an das Log zu finden.

4. Apply
Schreibe Änderungen in die Entities und Indexe in BigTable.

5. Clean up
Lösche Daten die nicht mehr benötigt werden.

Eine Schreiboperation ist prinzipiell bereits nach dem Commit abgeschlossen. Es wird aber mit dem *return* an die Anwendung gewartet bis sie auf dem nähesten Replikat angewendet wurde.

3.2 Verfügbarkeit und Skalierbarkeit

Nachdem einige Features von Megastore beschrieben wurden, wird nun auf Besonderheiten der Architektur eingegangen.
Benötigt wird eine globale, zuverlässige und beliebig große Datenbank. Im Gegensatz dazu stehen die Eigenschaften der Hardware. Sie ist typischerweise geografisch in Rechenzentren verteilt, fehleranfällig und besitzt begrenzte Kapazität. Das Ziel ist es, die Schwächen der Hardware bestmöglich auszugleichen. Megastore verfolgt dazu einen zweigeteilten Ansatz:

- für **Verfügbarkeit** wird ein synchroner, fehlertoleranter Log-Replikator eingesetzt
- für **Skalierbarkeit** werden die Daten in viele kleine Datenbanken partitioniert, jede mit ihrem eigenen replizierten Log

3.2.1 Replikation

Die Daten zwischen Hosts innerhalb eines Datencanters zu replizieren erhöht die Verfügbarkeit, da Host-spezifische Fehler überstanden werden können.

Es müssen aber ebenso die Netzwerke berücksichtigt werden, welche die Datencenter mit der Außenwelt verbinden. Weitere Fehlerquelen können in der Infrastruktur liegen, die sie mit Strom versorgt, kühlt und unterbringt. Ökonomisch gebaute Anlagen unterliegen dem Risiko des Ausfalls des kompletten Datencenters und sind anfällig für regionale Katastrophen. Um die Verfügbarkeitsanforderungen für Cloud Storage zu erfüllen, müssen Daten über geographisch entfernte Datencenter repliziert werden.
Auf der Suche nach einer Replikationsstrategie für Megastore sollten Techniken vermieden werden, bei denen im Fehlerfall Daten verloren gehen. Außerdem wurden Strategien vermieden, die keine ACID Transaktionen erlauben. Des Weiteren wurden Optionen ausgeschlossen, die einen ausgezeichneten Master benötigen.

Paxos

Google entschied sich, das Paxos Protokoll einzusetzen. Es ist ein optimaler, fehlertoleranter Übereinstimmungsalgorithmus, der keinen ausgezeichneten Master benötigt. Es wird genutzt, um ein *write-ahead Log* über eine Gruppe symmetrischer Peers zu replizieren. Jeder Knoten kann dabei Lese- und Schreibvorgänge starten. Eine neuartige Änderung an Paxos erlaubt lokale Reads auf jedem aktuellen, lokalen Replikat. Eine weitere Änderung ermöglicht single-roundtrip Writes.
Trotz der Fehlertoleranz von Paxos gibt es Einschränkungen, wenn nur ein Log verwendet wird. Die geographische Verteilung der Replikate verschlechtert die Latenzzeit bei der Kommunikation und limitiert so den gesamten Durchsatz. In einer traditionellen SQL Datenbank mit nur einem, synchron replizierten Log können Unterbrechungen mit weitreichenden Auswirkungen auftreten. Um also Verfügbarkeit und Durchsatz zu verbessern, werden mehrere replizierte Logs eingesetzt, jedes verantwortlich für einen Teil des Datenbestandes (siehe Abb. 6), nämlich genau dem Teil der zu einer Entity Group gehört.

3.2.2 Partitionierung

Um den Durchsatz zu erhöhen und Unterbrechungen zu lokalisieren, werden die Daten in Entity Groups partitioniert. Jede von diesen wird synchron über große Entfernungen repliziert und in dem zu Grunde liegenden BigTable System gespeichert. Die Entities in einer Entity Group werden mit einer einphasigen ACID Transaktion geändert (siehe Abb. 6).

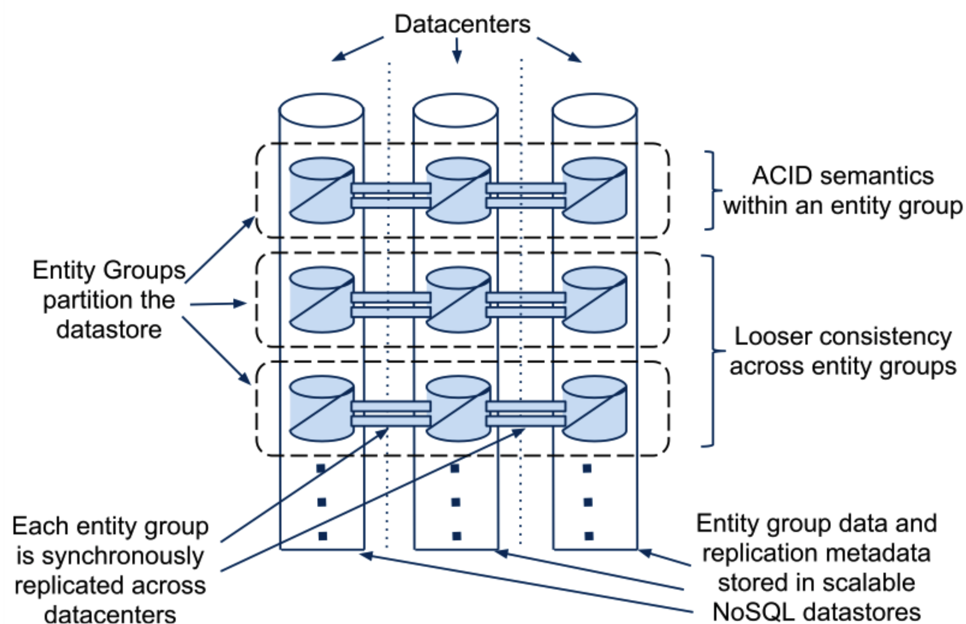

Abbildung 6: skalierbare Replikation [1]

Während die meisten Operationen innerhalb von Entity Groups stattfinden sollten, bietet Megastore verschiedene Möglichkeiten, um Operationen zwischen diesen durchzuführen. Einerseits ist das zwei-Phasen Commit möglich, aber in den meisten Fällen wird die effiziente, asynchrone Nachrichtenübertragung von Megastore benutzt. Eine Transaktion aus einer sendenden Entity Group fügt eine oder mehrere Nachrichten an eine Queue an. Transaktionen in empfangenden Entity Groups erhalten diese Nachrichten automatisch und wen-

den ausstehende Änderungen an. In Abb. 7 sind die verschiedenen Operationen zwischen Entity Groups dargestellt.
Wichtig hierbei ist, dass asynchrone Nachrichten nur zwischen logisch entfernten Entity Groups genutzt werden, nicht zwischen physisch entfernten Replikaten. Zwischen den Datencentern werden nur Replikationsoperationen ausgetauscht, welche synchron und konsistent sind.

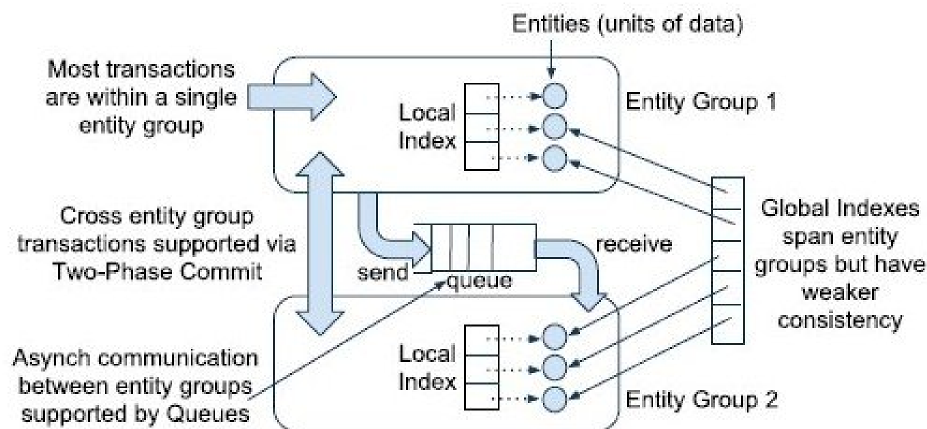

Abbildung 7: Operationen zwischen den Entity Groups [1]

Beispiele der Partitionierung in Entity Groups

Die Entity Groups definieren die a priori Grupierung der Daten in Megastore, auf denen schnelle Operationen möglich sein werden. Eine zu feine Gliederung führt zu vielen teuren Operationen, die zwischen den Entity Groups durchgeführt werden müssen. Aber wenn zu viele unnötige Daten in einer Entity Group sind, kommen unnötig teuere Read Operationen zustande, was wiederum den Durchsatz mindert. Hier sind einige Beispiele, wie Anwendungen mit diesen Grenzen umgehen können:

Email

Jeder Email Account bildet intuitiv eine Entity Group. Das bedeutet, dass Operationen innerhalb eines Accounts transaktional und konsistent sind. Ein Nutzer sieht also Änderungen an Nachrichten ebenso wie das Senden von Nachrichten sofort.

Karten

Manche Daten haben keine natürliche Granularität, zum Beispiel geographische Daten. Hier muss eine Karte in nicht überlappende Flächen aufgeteilt werden. Anders als bei Email Accounts muss hier aber auch beachtet werden, dass sich die Anzahl der Entity Groups im Verlauf der Zeit nicht ändert. Es müssen also schon zu Beginn genug Entity Groups erstellt werden, um später einen ansprechenden Durchsatz zu erreichen.

Allgemein kann gesagt werden, dass bis jetzt alle Anwendungen, die Megastore benutzen, auf natürliche Weise die Grenzen ihrer Entity Groups gefunden haben.

4 Zusammenfassung

In der Arbeit wurden die Anforderungen und Probleme dargestellt, die für Datenbanken durch das erhöhte Datenaufkommen und die ansteigenden Nutzerzahlen entstehen. Als möglicher Lösungsansatz wurden die skalierbaren relationalen Datenbanken allgemein vorgestellt. Anschließend wurde dieses Konzept am Beispiel von Google Megastore näher erläutert.

Es kann schlussfolgernd gesagt werden, dass Megastore eine skalierbare, hoch verfügbare Datenbank ist, die den Anforderungen eines interaktiven Internet Services gerecht wird. Durch die Nutzung von BigTable als zu Grunde liegendes Speichersystem wird eine schnelle Zugriffszeit auf die Daten erreicht. Um die Anwendungsentwicklung zu vereinfachen, wurden einige Features, wie ACID Transaktionen und Indexe hinzugefügt. Die Partitionierung der Daten in Entity Group Sub-Datenbanken bietet gewohnte transaktionale Eigenschaften und ermöglicht die Skalierbarkeit der Speicherung als auch des Durchsatzes.

Auf Grund der genannten Eigenschaften hat sich Megastore bei Google bereits mehrere Jahre bewährt. Es erledigt täglich ca. 3 Milliarden Schreib- und 20 Milliarden Leseoperationen. Die meisten Kunden berichten von hoher Verfügbarkeit, obwohl ständig Maschinenfehler, Netzwerkprobleme und Ausfälle von Rechenzentren auftreten.

Literatur

[1] Baker, J., Bond, C. et al. Megastore: providing scalable, highly available storage for interactive services. Conference on Innovative Data Systems Research. (2011)

[2] Cattell, R. Scalable SQL and NoSQL datastores. ACM SIGMOD Record 40, 2. (June 2011)

[3] Megastore: Providing scalable, highly available storage for interactive services.
http://muratbuffalo.blogspot.com/2011/03/megastore-providing-scalable-highly.html