**UNIVERSITÄT LEIPZIG**

**Institut für Informatik – Abteilung Datenbanken**

**Seminar - Cloud Data Management**

# Parallele Cloud-DBS – Aufbau und Implementierung

Leipzig, 24.03.2010

vorgelegt von:
Markus Weise
geb. 28.11.1983

Betreuer:
Stefan Endrullis

# Inhalt

# 1. Einleitung

In unserer heutigen Gesellschaft fallen überall Daten an. Jeder Schritt in unserem Leben hinterlässt eine digitale Spur. Diese Spuren geben Auskunft über unser Leben, unsere Hobbies und unsere Gewohnheiten.

Diese Informationen gewinnen in der heutigen Zeit immer mehr an Wert für Unternehmen. Sie werden im großen Stil gespeichert, um unter anderem den Menschen beziehungsweise Kunden für ihn maßgeschneiderte Werbung oder Kaufvorschläge unterbreiten zu können.

In sozialen Netzwerken, wie zum Beispiel Facebook, StudiVZ oder Xing, sind Millionen von angemeldeten Nutzern unterwegs und stellen hier Informationen über sich und ihr Leben online, laden Bilder hoch und kommunizieren untereinander. All diese Dinge verursachen ein riesiges Aufkommen an Daten, auf die jederzeit und zum Teil von allen Nutzern zugegriffen werden können muss.

Ein weiteres Beispiel bietet der Google Konzern. *„Ziel des Unternehmens [ist es], die gewaltige Menge an Informationen zu organisieren, die im Web verfügbar ist.“* [5] Google durchsucht das World Wide Web und sammelt dabei so viele Informationen wie möglich, um diese dem Anwender zur Verfügung stellen zu können. Der Crawling-Mechanismus zum Durchsuchen des Netzes speichert nicht nur die URL der gefundenen Webseiten sondern auch deren Inhalt und Verweise von anderen Webseiten. Weiterhin bietet Google mit seinem Email-Dienst Millionen von Nutzern mehrere Gigabyte an Speicherplatz (zurzeit 7,4 GB [6]). All die anderen Webapplikationen von Google sind nicht weniger speicherintensiv: GoogleVideo, GoogleMaps, GoogleBooks, um nur Einige zu nennen.

Amazon stellt neben seinem Onlineversandhandel unter anderem quasi unbegrenzten Speicherplatz zur Verfügung, der über das Internet erreichbar ist.

Jedes der genannten Beispiele ist zwar vom Ansatz und der Art der Anwendung her grundverschieden. Ob es nun das schlichte Chatten oder Bloggen über das Internet ist oder die Zurverfügungstellung eines Emaildienstes für den Endanwender oder das Anbieten eines Online-Datenbanksystems; alle haben doch eine Gemeinsamkeit: Es fallen riesige Mengen an Daten an, die gespeichert werden und natürlich auch in angemessener Zeit wieder abrufbar sein müssen.

Kommerzielle Datenbanksysteme sind diesen Aufgaben nicht mehr gewachsen. Die bei der Nutzung dieser Art Datenbanksystemen typischer Weise anfallenden Daten sind in der Regel zu groß, als dass sie von diesen in angemessener Zeit verarbeitet und ausgewertet werden könnten. Weiterhin benötigen solche Systeme in der Regel spezialisierte highend Hardware, die in den meisten Fällen sehr teuer ist. Die hier im Folgenden vorgestellten Clouddatabase-

Systeme zielen im Gegensatz dazu darauf ab, eine hohe Anzahl von lowcost bzw. handelsüblicher PC-Hardware zu verwenden. Diese ist schnell und billig zu beschaffen und dadurch bei Defekt schnell und effizient zu ersetzen. Neben der Kosteneffektivität spielen aufgrund der Anwendungen, die durch eine solche Art von Clouddata-Systemen betrieben werden, Latenzzeiten und Skalierbarkeit der Systeme eine wichtige Rolle.

Ein weiterer Aspekt, den unter anderem sowohl Google als auch Yahoo! hier verfolgen, ist, dass man das im eigenen Haus entwickelte System individuell auf die eigenen Anwendungsfälle spezialisieren und optimieren kann.

„And it's fun and challenging to build large-scale systems", Jeffrey Dean

# 2. Google's BigTabel

## 2.1 Einleitung

BigTabel ist ein von Google entwickeltes verteiltes Datenbanksystem, das speziell für den Umgang mit sehr großen Datenmengen geeignet ist. Die Architektur des Systems ist so angelegt, dass es jederzeit erweitert werden kann und so riesige Datenmengen verwalten kann. Google spricht hier von Datenmengen im Petabytebereich.

Google nutzt dieses System ausschließlich intern und bietet es nicht als Cloud-Service an wie zum Beispiel Amazon. Nahezu alle Webapplikationen von Google greifen auf die BigTabel zu, um die anfallenden beziehungsweise benötigten Daten zu speichern, auszuwerten oder anzuzeigen. An erster Stelle sind hier sicherlich Anwendungen wie Gmail, GoogleMaps und nicht zuletzt die Websuche zu nennen.

Die Daten werden dabei, ähnlich wie in einem relationalen Datenbankmodell, mit Hilfe von Spalten und Spalten indiziert. Wobei die Daten innerhalb der BigTable als uninterpretiertes Byte-Array gespeichert werden. Um die Daten persistent zu speichern, nutzt Google das ebenfalls selbst entwickelte *Google File System*.

Da die meisten Anwendungen, die Google im Internet bereitstellt, ihre Daten in der BigTable speichern, spielten beim Design des Systems neben der oben erwähnten Skalierbarkeit noch weitere Eigenschaften eine Rolle:

Hohe Verfügbarkeit der BigTable ist demnach von entscheidender Bedeutung für die Verfügbarkeit der Google-Apps. Ebenfalls ist die Performanz von entscheidender Bedeutung, da täglich viele Nutzer gleichzeitig zum Beispiel Googles Suchmaschine benutzen, was mehrere Zugriffe auf die Crawling-Ergebnisse (Webtable) und damit auf die BigTable erfordert.

## 2.2 Das Datenmodell

Die Daten werden mittels Zeilen- und Spaltenindizes innerhalb der Tabelle abgelegt beziehungsweise adressiert (s. Abb. 1). Die Zeilenindizes dienen dabei als eindeutiger identifier für einen Datensatz. Dieser ist ein String mit einer maximalen Länge von 64kB. Sowohl Lese- als auch Schreibzugriffe auf eine Zeile der BigTable werden atomar durchgeführt. Es spielt dabei keine Rolle, wie viel Attribute der Datensatz besitzt, auf den zugegriffen wird oder wie groß er ist.

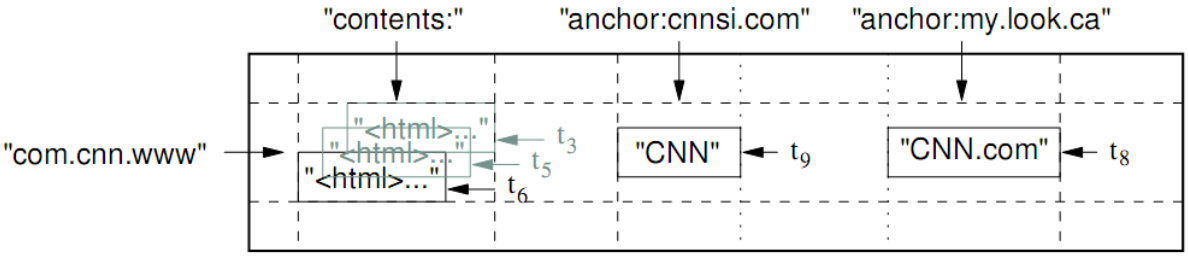


Abbildung 1 - Datenmodell am Beispiel von Google's Webtabel [1]

Da sich unter Umständen Millionen von Datensätzen in der BigTable befinden können, werden die Datensätze an Hand der Zeilenindizes alphabetisch sortiert, um die Zugriffszeiten auf einen oder mehrere Datensätze gering zu halten. Das hat zu Folge, dass Zugriffe auf einen oder einige wenige zusammenhängende Datensätze sehr effizient ausgeführt werden können und in der Regel die Kommunikation zwischen den in der Cloud vorhandenen Rechnern auf einige wenige beschränkt werden kann.

So werden zum Beispiel die aus den Crawling-Ergebnissen gewonnenen URLs in umgekehrter Reihenfolge in der Tabelle abgelegt. Zum Beispiel wird aus *maps.google.com/index.htm, com.google.maps/index.html.* Dadurch stehen zuerst alle Toplevel-Adressen zusammen, dann alle Seiten, die zu Google gehören, und so weiter.

Spalten in der BigTable sind äquivalent zu Attributen in der konventionellen relationalen Datenbank. Jedoch können verschiedene Spalten einer Spaltenfamilie (*column-family*) angegliedert sein. Daten einer Spaltenfamilie gehören typischerweise inhaltlich zusammen und sind daher in der Regel vom gleichen Typ. Bevor eine Spalte beziehungsweise ein Attribut einem Datensatz hinzugefügt werden kann, muss die Spaltenfamilie schon existieren oder neu angelegt werden. Ein Spaltenname hat somit immer die Form: *column-family:qualifier*.

Spaltenfamilien dienen auch als Einheit der Zugriffskontrolle. Das heißt, dass gleichzeitig mehrere Anwendungen Operationen auf ein und denselben Datensatz durchführen können. Im Falle von Googles Webtable, in der die Crawling-Ergebnisse gespeichert werden, bedeutet das, dass gleichzeitig Änderungen an einer Webseite gespeichert, neue Spaltenfamilien erzeugt und oder die Daten angezeigt werden können.

Um mehrere Versionen eines Datums speichern zu können, verwendet Google neben Zeilen- und Spaltennamen noch Zeitstempel, um Daten zu indizieren. Diese Zeitstempel sind 64-bit Integer, welche einerseits automatisch direkt von BigTable vergeben werden können. In diesem Fall repräsentiert dieser das Erstellungsdatum in Millisekunden. Andererseits können diese Zeitstempel auch von einem Anwendungsprogramm, welches den Datensatz, schreibt vergeben werden.

Um die Versionsverwaltung einfacher zu machen, gibt es die Möglichkeit, alte Versionen mit Hilfe einer automatischen *Garbage-Collection* zu entfernen. So kann gewährleistet werden, dass immer eine bestimmte Anzahl neuester Versionen eines Datensatzes zur Verfügung steht.

## 2.2 Tablets

Um gleichbleibend niedrige Zugriffszeiten auf die BigTable zu gewährleisten, kann diese horizontal partitioniert werden. Die so entstehenden zusammenhängenden Ausschnitte der Tabelle nennt man *Tablets.*

Diese Tablets werden auf verschiedenen, so genannten Tablet-Servern ausgeführt und verwaltet. Dabei ist das Ziel, dass ein solcher Server für ca. 100 solcher Tablets verantwortlich ist. Diese Maßnahme dient vor allem der Lastbalancierung. So können zum Beispiel mehrere Tablets, die weniger stark frequentiert sind, mit hochfrequentierten Tablets zusammen auf einem Server verwaltet werden. Sollte sich die Lastverteilung während des Betriebes ändern, so können die Tablets auch weiter auf gesplittet oder zum Beispiel zwei Tablets wieder zusammen gefügt werden. Gleiches kann auch passieren, wenn durch Veränderungen an den Datensätzen die Größe des Tablets zu stark ansteigt oder zu klein wird. Es wird eine Tabletgröße 100 bis 200MB angestrebt. Zu große Partitionen behindern die Lastbalancierung, da hier die Zugriffe auf einzelne Datensätze nicht mehr effizent genug durchgeführt werden können. Zu kleine Fragmente dagegen verursachen einen zu großen Verwaltungsoverhead. Die erreichbare Parallelisierung von Datenbankoperationen würde durch die zusätzlich anfallenden Kommunikationskosten wieder wett gemacht.

Die horizontale Partitionierung der BigTable in Tablets hat noch einen weiteren Vorteil für die Recovery-Durchführung. Obwohl augenscheinlich beim Ausfall eines Tabletservers viele Tablets wieder hergestellt werden müssen, können die anfallenden Wiederherstellungsoperationen so besser auf viele verschiedene Tabletserver verteilt werden. Fällt nun ein Server aus, so wird ermittelt, welche Server noch freie Kapazitäten haben und dann werden die Wiederherstellungen der ausgefallenen Tablets an neue Tabletserver verteilt.

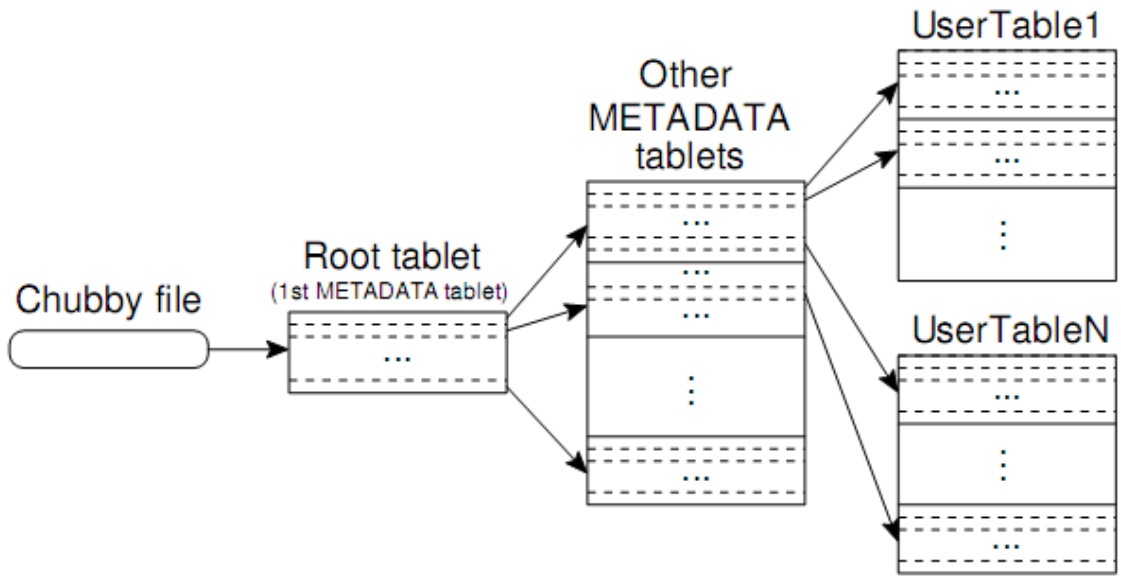


Abbildung 2 - 3-Stufiges hierachisches System zum Auffinden der BigTable-Datensätze [1]

Da, wie eingangs schon erwähnt, die Datenmengen in der BigTable sehr groß werden können, fällt eine rein sequentielle Suche auf den Tablets von vornherein aus. Daher verwendet Google hier ein dreistufiges hierarchisches System zum Auffinden der Datensätze, das ähnlich wie $B^+$-Baum funktioniert (s. Abb. 2).

Diese Daten befinden sich ebenfalls in der Cloud. Es muss daher ein fester Einstiegspunkt in die Suchstruktur gespeichert werden: Das erste Level ist demnach die Position der *Root table* (siehe Abb. 2). Dies geschieht mit Hilfe eines *Chubby*-Files. Die Root table enthält die Position aller Tablets der METADATA table. Obwohl die Root table zwar in diesem Sinne eine ganz normale Tabelle ist, wird sie jedoch gesondert behandelt. Um zu gewährleisten, dass die Datenstruktur nicht mehr als drei Ebenen bekommt, wird die Root table nie auf gesplittet. In der METADATA table werden zu jedem in der BigTable vorhanden Tablet der Zeilenindex der letzten Zeile und die Position des Tablets gespeichert. So ist sichergestellt, dass nur die Tablets aufgesucht werden müssen, die für die Bearbeitung der Suchanfrage benötigt werden. Um Zugriffe auf die Tablet-Positions-Tabellen zu sparen, werden die Standorte der Tablets in den teilnehmenden Clients gespeichert. Nur wenn der Client die Position des Tablets nicht kennt, muss in der Adresse des Tablets nachgeschlagen werden. Gleiches gilt für den Fall, dass sich die Adresse geändert hat und der Aufruf des Tablets erfolglos war.

Da sich die Adresse der METADATA table in der Regel nicht oder nur sehr selten ändert, muss bei einem erfolglosen Leseversuch nicht erneut die komplette Struktur durchlaufen werden. Es wird zuerst versucht, auf die METADATA table zuzugreifen. Erst wenn dies fehlschlägt, wird versucht, auf die Root table zu zugreifen und erst dann bei einem erneuten Fehlzugriff auf das Chubby-File.

## 2.3 Speicherung und Zugriff auf Tablets

Der persistente Zustand der Tablets und somit natürlich auch der BigTable wird innerhalb des Google File Systems gespeichert, welches in sogenannten *SSTable-Files* geschieht. Dies ist ebenfalls ein bei Google entwickeltes Datenformat, das speziell für die eigenen Ansprüche ausgelegt und optimiert ist. Hierbei werden Key-Value-Paare gespeichert, wie sie unter anderem in der BigTable auftreten (Rowkey, Columnfamily:Identifier, Timestamp) → Byte-Array. Weiterhin sind diese Dateien nicht veränderbar, um die Dauerhaftigkeit der BigTable-Daten zu schützen. Änderungen an den Datensätzen werden entweder an ein bestehendes SSTable-File angehängt oder in einer neuen Datei gespeichert.

Die aktuellsten Modifikationen der Daten, die zwar Commited sind aber noch nicht auf die Platte zurückgeschrieben sind, werden im Speicher, in der sogenannten *memtable* gespeichert. Dieser dient dabei als Datenbankpuffer. Zum Sammeln von Wiederherstellungsinformationen werden Änderungen an den Daten zusätzlich in einen Datenbank-Log geschrieben (s. Abb. 3).

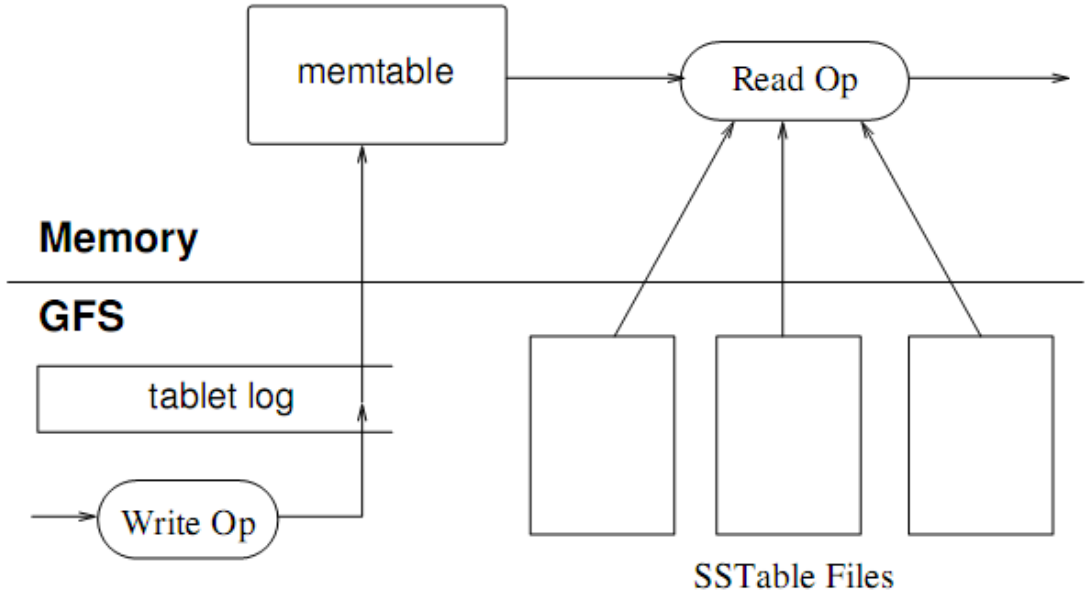


Abbildung 3 - Zugriff auf Tablets in der BigTable [1]

Eine ankommende Leseanfrage auf einen oder mehrere Datensätze eines Tablets werden zuerst auf ihre Validität überprüft. Ist dies erfolgreich durchlaufen worden, wird die Authentizität des Absenders überprüft und sichergestellt, dass dieser berechtigt ist, die angeforderten Daten zu empfangen. Dies geschieht, indem eine Liste von autorisierten Lesern in einem Chubby-File gelesen wird. Anschließend erfolgt das tatsächliche Lesen der angeforderten Daten. Da die Teile der Daten sowohl in der memtable als auch in den SSTable-Files stehen können, müssen sowohl die SSTable-Files als auch die memtable nach möglichen Treffern durchsucht werden. Die Ergebnisse der beiden Suchdurchläufe werden dann gemerged. Die Daten sind sowohl in der memtabel als auch in den SSTableFiles

alphabetisch sortiert. Dadurch können die Suche und der Merge-Vorgang entsprechend effizient durchgeführt werden.

Eine Änderungsoperation auf den Daten verläuft nach einem sehr ähnlichen Schema. Zuerst wird die eintreffende Anfrage auf ihre Validität überprüft. Danach wird eine Authentifizierung des Absenders durchgeführt. Dabei wird ebenfalls mit Hilfe eines Chubby-Files überprüft, ob der Absender berechtigt ist, die Änderungsoperation durchzuführen und die Daten zu ändern. Anschließend wird die Datenänderung im Commit-Log gespeichert. Um den Durchsatz von Änderungsanfragen zu erhöhen, ist es auch möglich, mehrere kleinere Änderungen mit Hilfe eines so genannten *Group commit* nach deren erfolgreichem Abschluss zu einem gesamten Commit zusammenzufassen. Nachdem die Änderungen im Commit-Log gespeichert sind, werden sie in die memtable übernommen.

Um die Performanz und gleichbleibend niedrige Antwortzeiten zu gewährleisten, werden je nach Bedarf zwei verschiedene Optimierungen durchgeführt: Die *minor compaction* und *major compaction*.

Mit steigender Zahl von Schreiboperationen auf die BigTable steigt die Größe der memtable stetig an. Die Anzahl an Datensätzen im Speicher wird somit immer größer. Um dies zu verhindern, wird die memtable bei Überschreiten eines Größenschwellwertes komplett ausgelagert. Dies geschieht, indem sie in ein SSTable-File umgewandelt wird und damit zurück in das Google File System geschrieben werden kann. Diese sogenannte minor compaction verfolgt im Wesentlichen zwei Ziele. Zum einen kann somit der Hauptspeicherbedarf der einzelnen Tablet-Server reduziert werden. Zum anderen wird der anfallende Arbeitsaufwand im Falle eines Serverausfalls reduziert. In diesem Fall müssen weniger Daten aus dem Commit-Log gelesen werden, um das Redo durchzuführen.

Jede minor compaction erzeugt neue SSTable-Files. Der Aufwand, der beim Lesen eines Datensatzes entsteht und damit natürlich auch die Antwortzeiten derartiger Operationen beieinflusst, hängt stark von der Anzahl dieser SSTable-Files ab. Um die Anzahl dieser SSTable-Files zu beschränken, wird die so genannte major compaction durchgeführt. Hierbei werden die Inhalte der bestehenden Dateien in periodischen Abständen zusammengeführt. Dabei werden die Inhalte von einigen SSTable-Files und der memtable gelesen. Sowohl der Inhalt dieser SSTable-Files als auch der Inhalt der memtable können verworfen werden sobald die major compaction abgeschlossen ist.

SSTable-Files, die nicht durch eine major compaction entstanden sind, können spezielle Informationen über gelöschte Daten enthalten. Diese Daten bleiben auch nach einer major compation erhalten. Die SSTable-Files, die durch eine solche major compaction entstanden sind, enthalten dagegen keine Informationen über gelöschte Datensätze. Somit ist sichergestellt, dass gelöschte Datensätze keine wichtigen Systemressourcen verbrauchen und dass diese Informationen ständig in dem System verbleiben und somit jederzeit wieder hergestellt werden können.

## 2.4 Chubby

BigTable beruht auf einem hochverfügbaren und persistent verteilten Lock Service namens *Chubby*. Chubby ist ein ebenfalls von Google entwickeltes System, welches speziell für solche lose gekoppelten Systeme, wie sie bei Cloud-Systemen üblich sind, entwickelt wurde.

Ein Chubby Prozess besteht aus einer Reihe von verschiedenen Replika, wovon eine als Masterprozess ausgewählt wird. Allein dieser Masterprozess führt die an das System gestellten Anfragen aus.

Das System wird hauptsächlich für Verwaltungsaufgaben genutzt. Hier werden zum Beispiel Schemainformationen abgelegt und die Zugriffsrechte auf die Daten verwaltet. Des Weiteren findet man hier die schon obenerwähnte METADATA table zum Auffinden der Datensätze.

Weitere systemrelevante Managementaufgaben, für die Chubby verantwortlich zeichnet, sind die eigene Konsistenzerhaltung zwischen den verschiedenen Chubby Replika und die Verwaltung der im System vorhandenen Server. Dazu zählt vor allem das Feststellen von Serverausfällen. In diesem Fall muss das Recovery der Daten des ausgefallenen Servers eingeleitet werden und die anfallenden Aufgaben auf andere verfügbare Maschinen verteilt werden. Ebenfalls wichtig ist das selbstständige Erkennen von neuen Servern im System. Diese müssen ohne Problem in die Cloud integriert werden können und zur Lastverteilung beitragen.

Da dies alles hochkritische Aufgaben sind, ist klar, dass die Funktionsweise von BigTable unmittelbar von der Funktionsweise des Chubbies-Systems abhängt. Das heißt, dass wenn Chubby nicht mehr verfügbar ist, ist auch die BigTable nicht mehr verfügbar.

# 3. Yahoos!‘s PNUTS

Ein weiter Vertreter der Clouddata management Systeme ist *Yahoo!‘s PNUTS*. PNUTS ist ein von Yahoo! entwickeltes verteiltes Datenbanksystem. Es wurde, ähnlich wie BigTable für Google, auch speziell und ausschließlich für die Verwendung bei Yahoo! selbst entwickelt. Die Nutzung des Datenbanksystems ist also allein dem entwickelnden Konzern selbst überlassen. Es wurde ebenfalls speziell dafür entwickelt, den größtmöglichen Nutzen für die verschiedensten Web-Applikationen zu bieten. Daraus ergeben sich wiederum die zwar allgemein üblichen Anforderungen für eine solche Art von Systemen, jedoch können auf Grund der Anwendungen hier und da kleine Abstriche gemacht werden.

Ganz oben auf der Anforderungsliste bei Internetanwendungen, die von vielen Nutzern gleichzeitig erreichbar und nutzbar sein sollen stehen natürlich Antwortzeiten und Verfügbarkeit. Yahoo! setzt hier stark auf ein hochverteiltes System, dass nicht nur mit schlichter Redundanz Ausfällen vorbeugen und Lastbalancierung erreichen will, sondern es werden Replika des Systems geographisch verteilt. Man versucht damit zu erreichen, dass Nutzern auf der ganzen Welt annähernd  gleiche und niedrige Antwortzeiten gewährleistet werden können. Ein weiter Nebeneffekt dieser Strategie ist, dass die Daten der Datenbank vor Katastrophen weitgehend geschützt sind.

Da Yahoo! dieses System für seine eigenen Webanwendungen nutzt, verzichtet man hier auf all zu harte Konsistenzbedingungen. Das heißt, dass man hier bewusst unrepeatable reads in Kauf nimmt. Dies geschieht hauptsächlich, um teure Kommunikation zwischen den einzelnen Datencentern zu sparen. Es ist jedoch gewährleistet, dass die Konsistenz im Großen und Ganzen immer gewahrt bleibt.

## 3.1 Datenmodell

Yahoo! bietet mit PNUTS ein vereinfachtes relationales Datenmodell. In den Tabellen werden Records (Zeilen) mit Attributen (Spalten) gespeichert. Um das System für ein breites Spektrum an Anwendungen einsetzen zu können, ist es möglich, eine ganze Reihe von Datentypen in einer solchen Datenbank zu speichern. Dies reicht von primitiven Datentypen bis hin zu Bildern oder Musik beziehungsweise *blobs*.

Darüber hinaus ist es möglich, das Schema zu jedem Zeitpunkt zu verändern. Dies bedeutet, dass während des laufenden Betriebs oder sogar während der Ausführung einer Datenbankanfrage Attribute hinzugefügt werden können. Außerdem können auch komplett leere Records angelegt werden, die in keinem ihrer Attribute Daten enthalten. Ebenfalls gibt es hier auch keine systemseitige Unterstützung für Datenintegrität.

Serialisierbarkeit im Sinne der ACID-Eigenschaften ist für derart verteilte Systeme natürlich wichtig. Daher ist die Reihenfolge der ankommenden Update-Operationen vom Datenbanksystem zu jedem Zeitpunkt sicher gestellt (s. Abb. 4).

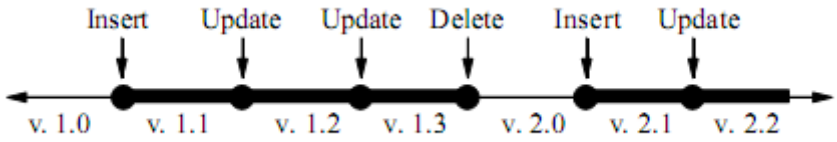


Abbildung 4 - Konsistenzerhaltung bei PNUTS [3]

Dafür gibt es für jeden Datensatz einen Master-Datensatz. Dieser wird vom System bestimmt und ist in der Regel der, mit dem meisten Workload. Ankommende Update-Anfragen werden vom User aus zunächst an das ihm am nächsten liegende Datencenter weitergeleitet und dort lokal verarbeitet. Die durchgeführten Änderungen werden dann zunächst zum Master dieses Datensatzes geschickt und von dort aus an alle weiteren Kopien des Datums weitergeleitet. Da die Masterkopie des Datensatzes sowieso den meisten Workload aufweist, besteht die relativ hohe Wahrscheinlichkeit, dass die Anfrage schon an den Master gestellt wurde und somit nicht erst dort hin weitergeleitet werden muss. Man versucht so, den Kommunikationsaufwandt möglichst gering zu halten.

Da es mit diesem Modell auch vorkommen kann, dass auf Daten zugegriffen wird, die so nicht mehr existieren, werden die Update-Anfragen im Zweifelsfall in der Masterinstanz des Datensatzes gemerged oder die Anfrage kann nicht ausgeführt werden und die Änderungen müssen zurückgenommen werden.

PNUTS stellt im Gegensatz zu konventionellen verteilten Datenbanksystemen weder SQL noch einen vergleichbaren Funktionsumfang zur Verfügung. Es sind somit auch keine höheren Datenbankoperationen wie die join-Bildung oder Group-by-Operationen möglich. Die angebotenen Funktionen sind speziell auf die geographische Verteilung angepasst und unterstützen das Daten- und Konsistenzmodell von PNUTS.

Zum Beispiel bietet PNUTS eine Reihe von Möglichkeiten, um Daten aus der Datenbank zu lesen. Die einfachste und simpelste Methode heißt *Read-any.* Diese Funktion liest einen Datensatz aus der Datenbank. Dabei wird aber nicht darauf geachtet, dass der gelesene Datensatz tatsächlich der aktuellen Version des Datensatzes in der Datenbank entspricht. Das heißt, dass es, wie oben schon erwähnt, zu unrepeatable reads kommen kann. Diese Methode ist dafür ausgelegt, eine möglichst geringe Antwortzeit zu haben. Diesen Latenzzeitgewinn erkauft man sich aber in dem Fall durch eventuell auftretende Updateanomalien. Um die Anfrage möglichst schnell beantworten zu könne, wird diese nur an das für den Nutzer nächstliegende Datencenter geleitet, um die Kommunikationskosten gering zu halten.

Im Gegensatz dazu stellt Yahoo! die Funktion *Read-Latest*, mit der sichergestellt werden kann, dass der zurückgegebene Datensatz auch tatsächlich dem jüngsten und damit aktuellstem Datenbankzustand entspricht. Im Gegensatz zu der Read-any-Methode, wird hier speziell auf Genauigkeit wertgelegt. Um sicherzustellen, dass der aktuellste Datensatz zurückgegeben wird muss, natürlich ein höherer Kommunikationsaufwand betrieben

werden. In dem Fall reicht es eben nicht aus, nur auf dem nächsten Datencenter die Daten anzufordern, sondern es müssen auch die anderen im System vorhandenen Replika auf eventuell aktuellere Versionen dieses Datums überprüft werden. Obwohl die Kommunikation zu den verschiedenen Instanzen parallel erfolgen kann, steht das Ergebnis erst zur Verfügung, wenn die langsamste Kommunikation abgeschlossen wurde.

Um einen gewissen Kompromiss zwischen Antwortzeit und Genauigkeit des zurückgegebenen Ergebnisses zu erreichen, wird die Methode *Read-critical<version>* zur Verfügung gestellt. Dieser Funktion übergibt man die Version des zu lesenden Datensatzes, und bei der Ausführung wird sichergestellt, dass der zurückgegebene Wert mindestens dieser Versionsnummer oder einer jüngeren entspricht. Somit kann im günstigsten Fall auf einen Großteil teurer Kommunikation verzichtet werden. Wenn die Version, die in dem Datencenter vorhanden ist, zu der die Anfrage geleitet wird, eben dieser Version entspricht, muss kein weiterer Aufwand betrieben werden. Es kann direkt dieser Wert zurückgegeben werden. In diesem Fall bietet diese Funktion genau die gleiche Antwortzeit, wie Read-any. Im ungünstigsten Fall jedoch, ist die Antwortzeit genau so hoch wie bei Read-Latest.

Das Schreiben von Daten mit der *write*-Methode funktioniert dagegen recht einfach. Der Datensatz der geschrieben werden soll, wird dem Datenbanksystem übergeben und dann über den Masterdatensatz im System verteilt. PNUTS sichert dabei volle ACID-Eigenschaften zu.

Eine weitere Methode, Daten zu schreiben, ist die Funktion *Test-and-set-write<Version>.* Diese Methode stellt sicher, dass der Datensatz, der geschrieben werden soll, genau der angegebenen Version entspricht. Damit ist sichergestellt, dass der Absender des Updatebefehles die Daten kennt, die er verändern will.

## 3.2 Systemarchitektur

Yahoo! unterteilt sein Clouddatebase-System in mehrere Regionen. Um möglichst allen Teilnehmern gleiche Zugriffszeiten zu ermöglichen, werden in verschiedenen Teilen der Erde Datencenter verteilt. Diese halten jeweils den kompletten Datenbestand vor.

Dabei besteht jedes Datencenter aus *Storage units*, *Tablet controllern* und *Routern*. Für die Kommunikation zwischen den einzelnen Regionen ist der Yahoo Message Broker zuständig (siehe Abb. 5).

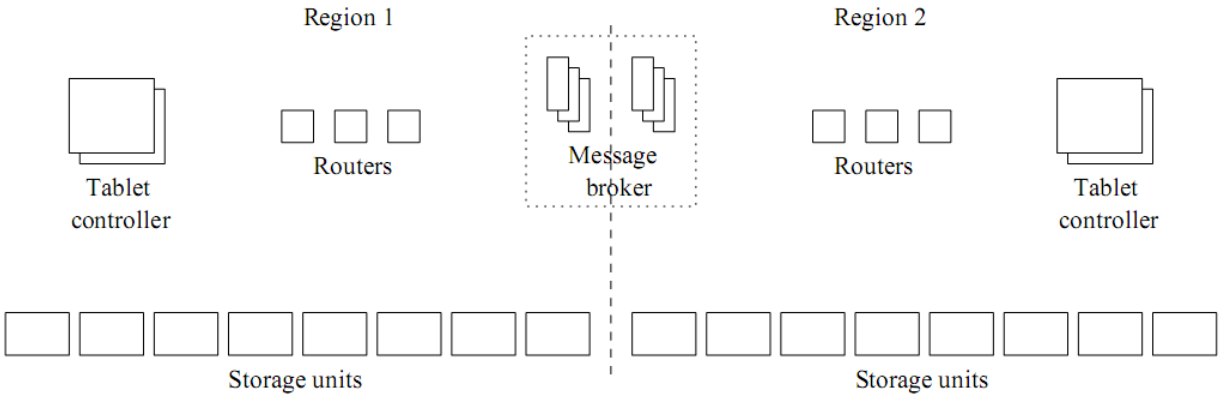


Abbildung 5 - Systemarchitektur PNUTS [3]

Ähnlich wie bei Google's BigTable werden die Tabellen bei PNUTS ebenfalls horizontal partitioniert und in einzelne Tablets zerlegt. Diese werden auf Servern in einer Region verteilt. Hierbei wird jedem Tablet genau ein Server zugeteilt. Die Zuteilung erfolgt dabei so, dass jeder Server für hunderte oder tausende dieser Tablets verantwortlich ist. Um hier ein Mittel zur Lastbalancierung zu haben, ist es möglich, diese Tabellenteile zwischen den Servern hin und her zu bewegen. So können Tablets von hoch belasteten Servern auf weniger belastete Server ausgelagert werden. Falls ein Server ausfallen sollte, ist es gleichfalls möglich, dessen Tablets auf andere oder neue Server zu verteilen.

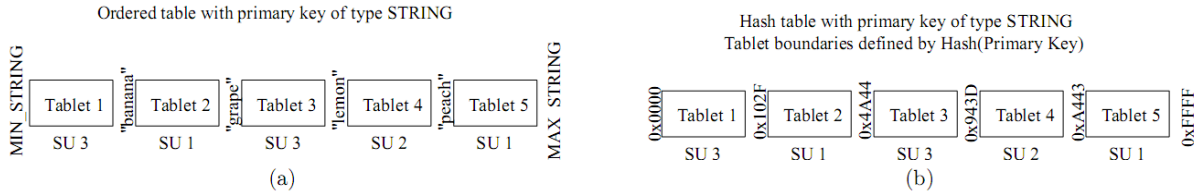


Abbildung 6 - Tablets auf Storage units [3]

Um einen Datensatz auf einer solchen Storage unit zu verändern beziehungsweise zu schreiben, muss zuerst herausgefunden werden, zu welchem Tablet der entsprechende Datensatz gehört und auf welcher Storage unit er zu finden ist (s. Abb. 6). Für beide Aufgaben ist der Router zuständig. Im Falle einer geordneten Tabelle sind die Primärschlüssel in Intervalle eingeteilt, die den Einteilungen der Tablets entsprechen. Der Router speichert dieses Mapping, um effizient auf die Daten zugreifen zu können. Dieses Mapping ist aber nur eine Kopie, die der Router vom Tablet controller gespeichert hat und in regelmäßigen Abständen wieder erneuert.

Der Tablet controller verwaltet, wie der Name schon sagt, die Tablets. Er überwacht die Auslastung der Tablet server und kontrolliert die Lastbalancierung durch Verteilung der Tablets auf die entsprechenden Tablet server. Wenn ein Tablet verschoben oder aufgesplittet wird, ist das Mapping des Routers veraltet und es wird einen Fehlzugriff geben, der dazu führt, dass das Mapping neu vom Tablet controller geladen werden muss.

## 3.3 Yahoo! Message Broker

Der Yahoo! Message Broker ist das Bindeglied zwischen den verschiedenen Datencentern in den verschiedenen Regionen. Es ist ein topicbasiertes pub/sub System (publish/subscribe), welches Nachrichten zwischen den einzelnen Datenbankreplika austauscht. Damit wird dafür gesorgt, dass die einzelnen Instanzen immer in einem konsistenten Zustand bleiben.

Die Übermittlung der Nachrichten wird garantiert. Änderungen an einem Record werden dann vom Tablet controller an den Message Broker gegeben und können damit als commited angenommen werden. Somit kommt diesem System eine zentrale Rolle in der Datenverteilung zu. Die Kommunikation und damit auch die Datenverteilung erfolgt asynchron. Aus diesem Grund kann nicht gewährleistet werden, dass alle Replika zu jedem Zeitpunkt den aktuellen Datenbankzustand repräsentieren.

Eine weitere Aufgabe, die der Yahoo! Message Broker übernehmen muss, ist das Recovery. Da PNUTS keinen Datenbank-Log für die durchgeführten Transaktionen besitzt, läuft ein Recovery auf ein schlichtes Kopieren einer anderen Instanz der Datenbank hinaus. Dieser Prozess erfolgt in drei Schritten:

Nachdem festgestellt wird, dass ein Tablet server ausgefallen ist, fordert der Tablet controller zuerst eine Kopie dieses Tablets an. In einem zweiten Schritt wird eine Checkpoint-Nachricht an den Message Broker geschickt. Damit wird sichergestellt, dass alle Updates, die während des Recovery an den ausgefallenen Tablet server gestellt werden, umgeleitet werden. Im dritten und letzten Schritt wird dann das Tablet aus einer anderen Region kopiert und damit die verloren gegangenen Daten ersetzt. Der größte Teil der Zeit beim Recovery wird mit dem tatsächlichen Kopieren verbracht. Das liegt hauptsächlich an Latenzzeiten und Bandbreitenbeschränkungen der Kommunikation. Aus diesem Grund kann es auch sinnvoll sein, neben den normalen Datencentern, die die Anfragen der Nutzer bearbeiten, auch noch sogenannte Backup-Regionen einzurichten. Diese sind dann ausschließlich dafür da, um bei einem Tabletrecovery die Kopie zur Verfügung zu stellen.

# 4. Zusammenfassung

Sowohl PNUTS als auch BigTable zeigen nach außen ein (quasi-) relationales Datenmodell. Die einzelnen Datensätze sind in Zeilen angeordnet und die Attribute der Records als Spalten. Eine horizontale Partitionierung der Datensätze ist auch in beiden Varianten möglich. Diese Teiltabellen können dann auf verschiedene Tabletserver verteilt werden, um die ankommenden Anfragen auf mehrere Server zu streuen.

Obwohl PNUTS und Google die Konsistenz ihrer Daten zusichern, kann es bei der Implementierung von Yahoo! zu unrepeatable reads kommen. Dies war eine Designentscheidung der Entwickler. Man hat versucht, trotzt der geographischen Verteilung bei Updates die Antwortzeiten auf den Daten weiterhin niedrig zu halten.

Ähnliche Unterschiede gibt es auch beim Recovery. BigTable bietet sowohl undo als auch redo-Recovery an. Diese Operationen sind bei PNUTS nicht möglich. Man hat bei Yahoo! darauf verzichtet, einen Transaktionslog zu führen. Dadurch besteht das Recovery nur aus einem Kopieren von einer anderen Datenbankinstanz. Demzufolge ist es möglich, dass Transaktionen verloren gehen.

Redundante Speicherung dient in beiden Fällen der Sicherung gegen einen Totalverlust der Daten. Google löst dieses Problem einfach durch Nutzung des Google File Systems, das von vornherein schon dafür ausgelegt ist, die Daten redundant zu speichern. Yahoo! verbindet die redundante Speicherung seiner Daten mit einer geographischen Verteilung der Kopien. Damit ist Schutz vor Katastrophen gewährleistet, und es bietet Nutzern auf der Erde gleichbleibende Latenzzeiten.

Alles in allem kann man sagen, dass die beiden vorgestellten Varianten von Google und Yahoo! sehr ähnlich sind. Es gibt zwar teilweise deutliche Unterschiede in der konkreten Implementierung der Verfahren. Das Ergebnis allerdings, das dem Nutzer präsentiert wird, ist am Ende sehr vergleichbar.

# 5. Quellen

**[1]**
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Grube:
*Bigtable: A Distributed Storage System for Structured Data.*
OSDI'06: Seventh Symposium on Operating System Design and Implementation;
Seattle, WA, November, 2006

**[2]**
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung:
*The Google File System.*
19th ACM Symposium on Operating Systems Principles;
Lake George, NY, October, 2003.

**[3]**
Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., Yerneni, R.:
*PNUTS: Yahoo!'s Hosted Data Serving Platform*
Very Large Data Base;
Auckland, NZ (2008)

**[4]**
Mike Burrows:
*The Chubby Lock Service for Loosely-Coupled Distributed Systems*
OSDI'06: Seventh Symposium on Operating System Design and Implementation;
Seattle, WA, November, 2006

**[5]**
http://www.google.de/corporate/
Stand 27.02.10

**[6]**
http://www.gmail.com/
Stand 27.02.2010