FernUniversität in Hagen

-

Seminar 01912

im Sommersemester 2013

Big Data Management

Thema 14

Cassandra

Referent: Jan Kristof Nidzwetzki

Inhaltsverzeichnis

Abstract

Apache Cassandra ist eine *NoSQL-Datenbank*, welche darauf ausgelegt ist, große Datenmengen auf einem Verbund von Servern zu verarbeiten. Die Ziele von Apache Cassandra sind hohe Verfügbarkeit, sowie gute Skalierbarkeit. Daten werden dazu redundant gespeichert und es existiert kein *single point of failure* in der Architektur der Software. Mittels *Tunable Consistency* kann bei jedem Lese- oder Schreibzugriff festgelegt werden, wie viele Server an diesem beteiligt sein müssen. Dies erlaubt, für jeden Zugriff zu entscheiden, ob Performance oder Konsistenz im Vordergrund stehen.

In der vorliegenden Arbeit wird die Software Cassandra vorgestellt. Ebenso wird kurz auf einige neuere Funktionen, wie die Abfragesprache *CQL – Cassandra Query Langugage* und die Anbindung an das Map-Reduce Framework *Hadoop*, eingegangen.

1 Einleitung

Die Menge der weltweit gespeicherten Daten nimmt rasant zu. Diese Datenmenge ist mit traditionellen *Relationalen Datenbank Management Systemen* (*RDBMS*) nur schwer zu verarbeiten. Seit einigen Jahren erfreuen sich sogenannte *NoSQL-Datenbanken* großer Beliebtheit. NoSQL-Datenbanken verzichten auf einige Eigenschaften, welche RDBMS bieten. Beispielsweise fehlen oft Transaktionen oder eine durchgängig konsistente Sicht auf den Datenbestand. Dafür bieten NoSQL-Datenbanken häufig bessere Skalierbarkeit und Ausfallsicherheit.

In der folgenden Arbeit wird das Datenbankmanagementsystem Cassandra vorgestellt. Diese Software gehört zu der Familie der NoSQL-Datenbanken und wird heutzutage von vielen Firmen eingesetzt, um große Mengen von Daten zu verarbeiten.

1.1 Geschichte

Das Cassandra Projekt wurde im Jahr 2007 von dem Betreiber der Social-Network Webseite *Facebook* initiiert. Facebook bietet Benutzern auf der gleichnamigen Webseite die Möglichkeit, Nachrichten auszutauschen. Mit mehr als 10 Millionen Nutzern im Jahr 2007 stieß Facebook an die Grenzen traditioneller RDBMS. Diese konnten die Nachrichten nur mit einiger Verzögerung bereitstellen und skalierten nicht sonderlich gut [LM10] [LM09].

Facebook stellte ein Team von Entwicklern zusammen, um dieses Problem zu lösen. Das Team entwickelte in den darauf folgenden Monaten die Software Cassandra. Die Software wurde im Juli 2008 in ein Projekt bei *Google Code* überführt. Der Quelltext war ab diesem Zeitpunkt frei verfügbar, jedoch konnten zunächst nur Mitarbeiter von Facebook Veränderungen vornehmen. Im Jahr 2009 wurde das Projekt an die *Apache Software Foundation* übergeben [Hew10, S. 24]. Die Software wurde in *Apache Cassandra* umbenannt und steht mittlerweile unter der Lizenz *Apache License 2.0* [WIK13].

Apache Cassandra weist eine aktive Entwicklergemeinde auf und es erscheinen fortlaufend neue Versionen der Software. Zum aktuellen Zeitpunkt, Mai 2013, steht die Version 1.2.2 auf den Seiten des Projektes zum Download bereit.

Im Jahr 2010 entwickelte Facebook sein Nachrichten-System von Grund auf neu. In dem neuen System wurde auf den Einsatz von Cassandra verzichtet. Die Ablage und das Durchsuchen der Nachrichten erfolgt nun auf der Basis der Software *HBase* [FAC13].

1.2 Grundlagen

Die Software Cassandra ist ein verteiltes Datenbankmanagementsystem. Cassandra ist mit dem Ziel entwickelt worden, große Mengen an Daten auf Standardhardware (*Knoten*) zu verarbeiten. Weitere Ziele von Cassandra sind: hohe Verfügbarkeit, Skalierbarkeit und Fehlertoleranz. Darüber hinaus ist das Konzept der *Tunable Consistency* umgesetzt: Clients können bei jeder Lese- oder Schreiboperationen festlegen, auf wie vielen Knoten diese stattfinden soll. Eine größere Anzahl von Knoten führt zu höherer Konsistenz, jedoch auch zu schlechterer Performance (siehe Abschnitt 2.3.1) [CAS13b].

Geschrieben ist Cassandra in der Programmiersprache *Java*. Die Architektur von Cassandra orientiert sich an der Software *Google Bigtable* [CDG+08], das Datenmodell an der Software *Amazon Dynamo* [DHJ+07]. Architektur und Datenmodell werden in Abschnitt 2 genauer beschrieben.

Der Autor Eben Hewitt beschreibt in seinem Buch *Cassandra: The Definitive Guide* die Software Cassandra in 50 Wörtern wie folgt [Hew10, S.14]:

> „*Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web.*“

1.3 Einsatzbereiche

Heutzutage setzen viele Unternehmen Cassandra ein. Insbesondere Unternehmen, welche große Mengen an Daten zu speichern haben oder ein großes Wachstum der Daten erwarten, setzten auf Cassandra. Hierzu zählen unter anderem [CAS13a]:

eBay: Die Internet Auktionsplattform *eBay* speichert Informationen über verkaufte Produkte in Cassandra.

IBM: Die Firma *IBM* bietet mit ihrem Produkt *BlueRunner* eine auf Cassandra basierende E-Mail Anwendung an.

Next Big Sound: Der Musikanbieter *Next Big Sound* speichert die Hörgewohnheiten seiner Nutzer in Cassandra ab.

Rackspace: Der Serverhoster *Rackspace* nutzt Cassandra zum Speichern und Auswerten von Logfiles.

Twitter: Der Kurznachrichtendienst *Twitter* setzt Cassandra zur Analyse von Nachrichten ein.

Häufig wird Cassandra in Kombination mit Hadoop eingesetzt. Cassandra übernimmt das Bereitstellen der Daten, Hadoop wird für die Auswertung der Daten eingesetzt (siehe Abschnitt 3.2).

2 Cassandra

Zu Beginn dieses Abschnitts wird das Datenmodell von Cassandra beschrieben. Anschließend wird die Architektur von Cassandra vorgestellt.

2.1 Datenmodell

Cassandras Datenmodel wird zu den *Spaltenorientierten Datenmodellen* (*Column oriented data models*) gezählt. Zum Überblick: in einem *Schlüsselraum* werden *Spaltenfamilien* definiert. Jede Spaltenfamilie besitzt eine oder mehrere *Spalten*. Zusammengehörende Werte werden in *Zeilen* mit eindeutigem *Zeilenschlüssel* zusammengefasst (Abbildung 1). In den folgenden Abschnitten werden diese Begriffe genauer beschrieben.

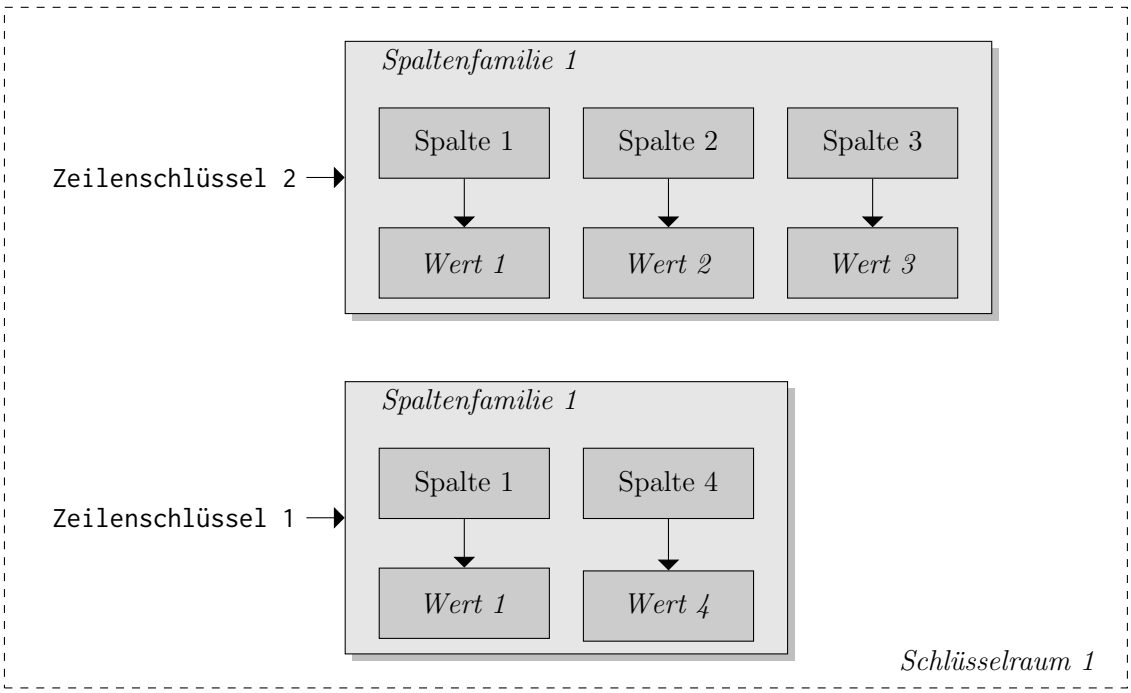


Abbildung 1: Datenmodell von Cassandra (nach [Hew10, S.44]). Im Schlüsselraum `Schlüsselraum 1` existieren zwei Zeilen mit den Zeilenschlüsseln `Zeilenschlüssel 1` und `Zeilenschlüssel 2`. Beide Zeilen gehören zur Spaltenfamilie `Spaltenfamilie 1`. Ihnen sind verschiedene Spalten mit Werten zugeordnet.

2.1.1 Schlüsselräume

Schlüsselräume (*Keyspaces*) werden in Cassandra dazu eingesetzt, unterschiedliche Daten voneinander zu trennen. Ein Schlüsselraum ist mit einer Datenbank in einem RDBMS zu vergleichen. Schlüsselräume besitzen *Metadaten*, welche das Verhalten des Schlüsselraumes festlegen. Hierzu zählen unter anderem [Hew10, S.46]:

Replikationsfaktor: Dieser Faktor legt fest, auf wie viele Knoten im Schlüsselraum gespeicherte Daten repliziert werden (siehe Abschnitt 2.2).

Platzierungsstategie für Replikate: Diese Strategie legt fest, wie Replikate auf unterschiedliche Knoten verteilt werden (siehe Abschnitt 2.2.2).

2.1.2 Spaltenfamilien

Spaltenfamilien (*Column Families*) beschreiben das Format der abgelegten Daten. Sie sind mit *Tabellen* in RDBMS zu vergleichen, weisen jedoch grundlegende Unterschiede auf. So können in Spaltenfamilien jederzeit neue Spalten eingefügt werden, ohne die bestehenden Daten zu beeinflussen. Eine Spaltenfamilie besteht aus einem Namen und einem *Comperator*, welcher festlegt, wie Daten in dieser sortiert werden sollen [DAT13a].

2.1.3 Spalten

Vergleichbar zu Attributwerten in RDBMS besitzt das Datenmodell *Spalten* (*Columns*). Diese bilden die kleinste Einheit im Datenmodel und sind für das Speichern von Name-/Wert-Paaren zuständig. Eine Spalte besteht aus einem Namen, dem dazugehörigen Wert und einen Zeitstempel (*Timestamp*).

Der Zeitstempel wird in Mikrosekunden angegeben. Er beschreibt, wann die Spalte das letzte Mal geändert worden ist. Der Zeitstempel wird zur Versionierung der Daten eingesetzt (siehe Abschnitt 2.3.3). Für Name und Wert können in Cassandra beliebige Byte-Arrays verwendet werden.

2.1.4 Zeilen

Wie in Abbildung 1 dargestellt, werden zusammengehörige Werte von Spalten in einer *Zeile* (*Row*) zusammengefasst. In RDBMS ist dies mit einem Tupel zu vergleichen. Identifiziert werden Zeilen über einen eindeutigen Zeilenschlüssel.

Beim Erzeugen einer Zeile müssen nicht alle in der Spaltenfamilie definierten Spalten mit Werten versehen werden. In Abbildung 2 sind zwei Zeilen mit den Zeilenschlüsseln `user4711` und `user0815` definiert. Beide enthalten Daten der Spaltenfamilie `Person`. Für die erste Zeile sind drei Spalten angegeben; für die zweite Zeile sind nur zwei Spalten angegeben.

Um in Cassandra auf Zeilen zuzugreifen, muss ihr Zeilenschlüssel bekannt sein. Alternativ lassen sich *Sekundärindizes* anlegen, um Zeilen mit bestimmten Eigenschaften finden zu können. Würde man für die Spalte `Nachname` in Abbildung 2 einen solchen Index anlegen, ließen sich alle Zeilen ermitteln, in denen `Nachname` z. B. den Wert „Müller“ annimmt.

2.1.5 Superspalten

Neben den vorgestellten Spalten existieren in Cassandra noch *Superspalten* (*Super Colums*). Superspalten weichen von dem bisher vorgestellten Name-/Wert-Modell ab. In Superspalten wird der Wert durch ein Array von Spaltenfamilien repräsentiert.

Superspalten werden eingesetzt, wenn Sammlungen von strukturierten Informationen gespeichert werden sollen. In der aktuellen Dokumentation von Cassandra werden Superspalten als „*Anti-Pattern*“ aufgeführt. Diese gelten als veraltet, liefern eine schlechte Performance und werden eventuell in zukünftigen Versionen nicht mehr unterstützt[1].

[1] „*Do not use super columns. They are a legacy design from a pre-open source release. This design was structured for a specific use case and does not fit most use cases. [...] Additionally, super columns are not supported in CQL 3.*“ [DAT13a]

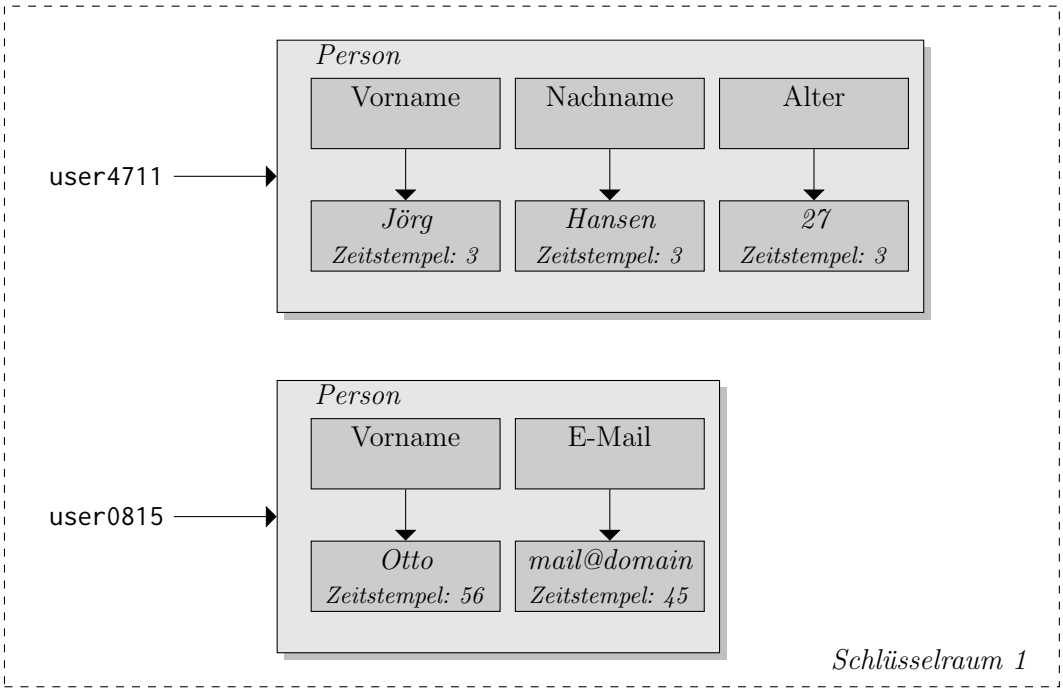


Abbildung 2: Das Datenmodell am Beispiel: Im Schlüsselraum `Schlüsselraum 1` existieren zwei Zeilen mit den Zeilenschlüsseln `user0815` und `user4711`. Beide Zeilen gehören zur Spaltenfamilie `Person`. In beiden Zeilen sind verschiedene Spalten angegeben: `Vorname, Nachname, Alter` und `E-Mail`. In jeder Spalte ist durch einen Zeitstempel vermerkt, wann diese das letzte Mal geändert worden ist.

2.2 Architektur von Cassandra

Ein Ziel von Cassandra ist es, die Verfügbarkeit von Daten auch dann sicherzustellen, wenn einzelne Knoten ausfallen. Hierzu wird ein Verbund von mehreren Knoten (ein *Cluster*) gebildet. Die Daten werden redundant auf mehreren Knoten abgelegt. Die Verteilung der Daten auf die Server wird durch einen *Partitionierer* (siehe Abschnitt 2.2.1) gesteuert.

Der Partitionierer bildet den Zeilenschlüssel einer Zeile in einen konstanten Wertebereich ab. Dieser Wertebereich wird in Cassandra in einem logischen Ring angeordnet. Zeilen werden gemäß des Partitionierers im Ring platziert. Jeder Knoten erhält beim ersten Start einen Identifizierer, den *Token*, aus dem gleichen Wertebereich. An diese Position fügt der Knoten sich in den Ring ein. Ein Knoten ist für die Werte zuständig, welche zwischen ihm und seinem Vorgänger liegen. In Abbildung 3 ist der logische Ring und die Replikation von Daten dargestellt. Partionierer können die Abbildung z. B. mit Hashfunktionen (*Consistent Hashing*) vornehmen [KLL+97]. Es wird mit einem konstanten Wertebereich gearbeitet um die Anzahl der Knoten verändern zu können ohne die Daten aller Knoten neu aufteilen zu müssen. Der Wertebereich ist in Abbildung 3 als Intervall $[0, 1]$ dargestellt.

Wird ein neuer Knoten in den Ring eingefügt, so analysiert er, welcher Knoten derzeit die meisten Daten vorhalten muss. Um diesen Knoten zu entlasten, wählt der neue Knoten einen Token, welcher zwischen diesem Knoten und seinem Vorgänger liegt. Gemäß des Wertes des Tokens fügt er sich in den logischen Ring ein und nimmt dem stark belasteten Knoten einen Teil seiner Daten ab.

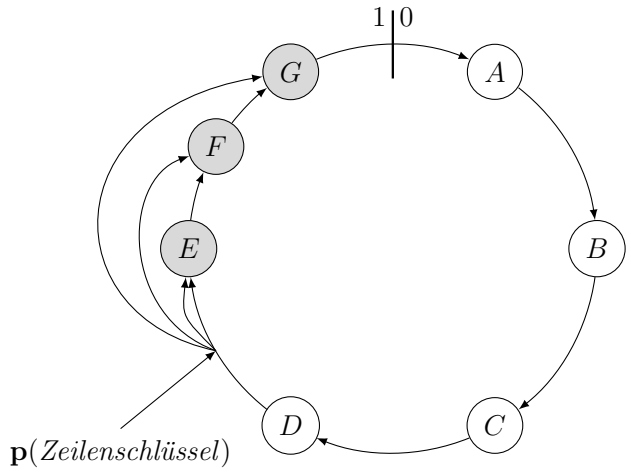


Abbildung 3: Ablage und Replikation von Zeilen: Die Knoten eines Clusters teilen die Werte des logischen Rings untereinander auf. Zeilen werden gemäß des Partionierers P im Ring abgelegt. Die Zeile mit dem Wert $p(\textit{Zeilenschlüssel1})$ wird auf den nachfolgenden Knoten E abgelegt. Zusätzlich wird ein Replikat auf den Knoten F und G abgelegt.

2.2.1 Partitionierer

Partitionierer sind dafür zuständig, aus dem Zeilenschlüssel einer Zeile die Position im logischen Ring zu berechnen. Sie legen damit fest, wie Daten auf die Knoten aufgeteilt werden. Cassandra stellt in der Version 1.1 zwei Partitionierer zur Verfügung. Ebenfalls können durch die Implementierung der Schnittstelle `org.apache.cassandra.dht.IPartitioner` eigene Partitionierer implementiert werden.

Random Partitioner: Dieser Partitionierer wird im Standardfall verwendet. Er berechnet den MD5-Hash eines Zeilenschlüssels. Hierdurch werden die Zeilen gleichmäßig über alle Knoten im Cluster verteilt. Der Wertebereich dieses Partitionierers beträgt 0 bis $2^{127} - 1$.

Byte-Ordered Partitioner: Zeilenschlüssel werden in Cassandra durch Byte-Arrays dargestellt. Dieser Partitionierer verwendet dieses Byte-Array für die Positionierung der Zeilen im Ring. Daten mit ähnlichen Zeilenschlüssel werden nah beieinander gespeichert. Dies kann sich positiv auf Abfragen auswirken, welche auf einem Bereich von Zeilenschlüssel operieren. Die benachbarte Speicherung ähnlicher Zeilen kann sich auch nachteilig auswirken, da bestimmte Bereiche im Ring stärker genutzt werden als andere. Dies führt zu sogenannten *hot spots*: Knoten die stärker belastet sind als andere.

2.2.2 Replikation

Der Partitionierer legt fest, auf welchen Knoten Zeilen primär gespeichert werden. Um Ausfallsicherheit zu erreichen, wird jede Zeile mehrfach gespeichert. Auf welchen Knoten die Replikate abgelegt werden, wird von der *Platzierungsstategie für Replikate* bestimmt. Wurde beim Anlegen des Schlüsselraums ein Replikationsfaktor von 1 angegeben, so werden Zeilen nur auf dem durch den Partitionierer bestimmten Knoten gespeichert. Wird der Replikationsfaktor auf $N > 1$ gesetzt, werden $N - 1$ Replikate auf anderen Knoten abgelegt.

Cassandra bietet verschiedene Platzierungsstategien für Replikate an. Welche Strategie verwendet werden sollte, gibt die zugrunde liegende physikalische Verteilung der Knoten vor. Grundlegend wird zwischen Knoten im gleichen Rack und Knoten im gleichen Datacenter unterschieden. Es wird davon ausgegangen, dass (i) einzelne Knoten, (ii) komplette Racks oder (iii) komplette Datacenter ausfallen können. Um diese Ausfälle kompensieren zu können, müssen die Replikate in verschiedenen Racks und in verschiedenen Datacentern abgelegt werden. In der aktuellen Version bietet Cassandra die folgenden Strategien an:

Simple Strategy: Replikate werden bei den nächsten Knoten im logischen Ring abgelegt. Die zugrunde liegende Topologie des Netzwerkes wird nicht berücksichtigt.

Old Network Topology Strategy: Es wird ein Replikat in einem zweiten Datacenter abgelegt. Alle weiteren Replikate werden über die Racks im ersten Datacenter verteilt.

Network Topology Strategy: Diese ähnelt der Old Network Topology Strategy. Hierbei wird jedoch mehr als 1 Replikat im zweiten Datacenter untergebracht.

2.2.3 Snitches

Die beiden letztgenannten Replica Placement Strategies benötigen Informationen, in welchen Racks und in welchen Datacentern sich welche Knoten befinden. Diese Informationen werden von *Snitches* bereitgestellt. Die standardmäßig von Cassandra verwendete *Simple Snitch* berechnet diese Informationen aus IP-Adressen. Knoten mit IPv4-Adressen mit gleichen Werten im ersten und zweiten Oktett befinden sich im gleichen Datacenter. Ist auch das dritte Oktett identisch, so befinden sich diese Knoten im gleichen Rack.

$$\text{Beispiel: } \overbrace{\underbrace{192.}_{Datacenter}\ \underbrace{168.}_{Datacenter}\ \underbrace{100.}_{Rack}\ \underbrace{001}_{Knoten}}^{IPv4-Adresse}$$

Um komplexere Netzwerktopologien abbilden zu können, existiert zudem eine konfigurierbare Snitch, die `PropertyFileSnitch`. In dieser können Beziehungen zwischen Knoten, Racks und Datacentern manuell hinterlegt werden. Ein Beispiel für eine solche Konfiguration ist in Listing 1 aufgeführt. In dieser Konfiguration existieren sechs Knoten, zwei Datacenter (`DC1` und `DC2`) und in jedem Datacenter zwei Racks (`RAC1` und `RAC2`).

Listing 1: Snitch Konfiguration mittels PropertyFileSnitch

```
1  # Data Center One
2  10.0.0.1=DC1:RAC1
3  10.0.0.8=DC1:RAC1
4  10.1.4.7=DC1:RAC2
5
6  # Data Center Two
7  10.5.2.1=DC2:RAC1
8  10.5.2.2=DC2:RAC1
9  10.5.3.1=DC2:RAC2
10
11 # default for unknown nodes
12 default=DC1:RAC1
```

2.2.4 Peer-to-Peer und Gossip

In vielen verteilten Systemen finden sich zwei unterschiedliche Klassen von Systemen: *Koordinatoren* und *Arbeiter*. Die Arbeiter sind für die Verarbeitung der Anfragen zuständig. Die Koordinatoren übernehmen Aufgaben wie das Verteilen von Anfragen oder das Prüfen, ob alle Mitglieder erreichbar sind. Oft stellen diese Koordinatoren einen *single point of failure* dar. Fällt der Koordinator aus, ist das gesamte System nicht mehr funktionsfähig.

In Cassandra kommt eine *Peer-to-Peer Architektur* zum Einsatz: alle Knoten nehmen die gleichen Aufgaben war. Anfragen können an jeden Knoten gestellt werden und der Ausfall eines *Knotens* sorgt höchstens für eine verringerte Leistungsfähigkeit des Systems, nicht jedoch für den Ausfall des gesamten Systems. Zudem sorgt die Architektur dafür, dass problemlos weitere Knoten in das System integriert werden können (siehe Abschnitt 2.2).

Es wird ein *Gossip protocol* [DGH+87] für die Kommunikation der Knoten untereinander verwendet. Periodisch tauschen dazu Knoten Gossip-Nachrichten aus. Aus Sicht eines *Knotens* (dem *Gossiper*) sieht die Kommunikation wie folgt aus [Hew10, S. 89]:

1. Alle n Sekunden wählt der *Gossiper* (G) zufällig einen Knoten (K) aus seiner Nachbarschaft aus und beginnt mit der Kommunikation.
2. G sendet K eine `GossipDigestSynMessage`.
3. Empfängt K die Nachricht, so bestätigt er dies mit Versand einer `GossipDigestAck Message` an G.
4. Den Empfang der Nachricht von K bestätigt G wiederum mit dem Versand einer `GossipDigestAck2Message` an K.

Erhält der Gossiper keine Antwort auf seine `GossipDigestSynMessage` geht er davon aus, dass der Knoten derzeit nicht erreichbar oder die Nachricht bei der Übertragung verloren gegangen ist. Es wird von Cassandra eine Implementation des Φ *Accrual Failure Detector* [HDYK04] eingesetzt. Dieser *Detector* sorgt dafür, dass Knoten erst nach einer bestimmten Zeit als nicht erreichbar markiert werden.

Durch die Nachrichten erhält jeder Knoten mit der Zeit Informationen über seine Nachbarn. Neben der IP-Adresse wird die Menge der gespeicherten Informationen (*Load*), sowie die Positionen im Ring (*Token*) ausgetauscht. Mit dem Programm `nodetool` lassen sich diese Informationen anzeigen. In Listing 2 sieht man einen Ring mit drei Knoten.

Listing 2: Ein Logischer Ring mit drei Knoten

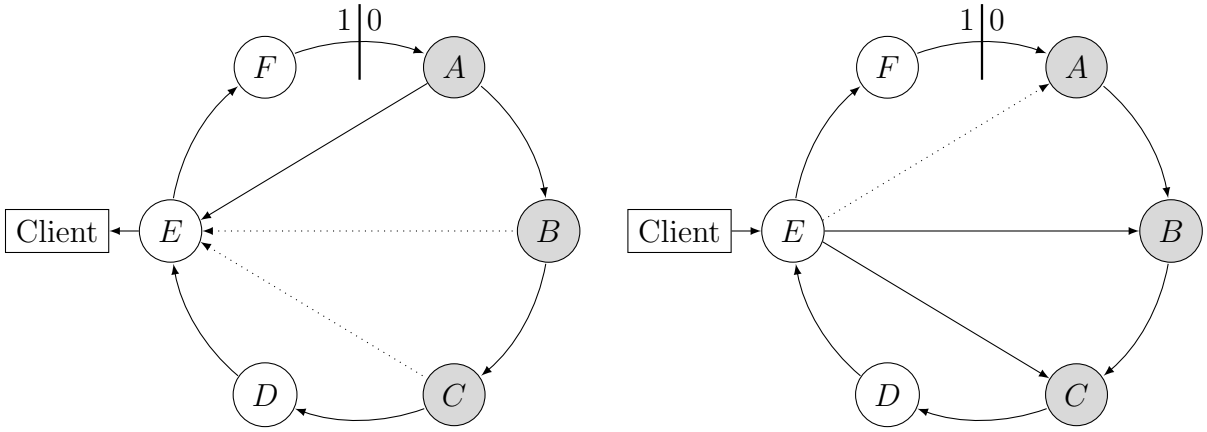
```
root@node1:~# /root/cassandra/bin/nodetool ring

Datacenter: datacenter1
==========
Address Rack  Status State   Load      Owns   Token
                                              3955191628143462120
node1   rack1 Up    Normal  93.96 KB  30.07% -8944999014129822443
node2   rack1 Up    Normal  42.59 KB  44.34% -766207061079759187
node3   rack1 Up    Normal  1.5 MB    25.59%  3955191628143462120
```

2.3 Lesen und Schreiben von Daten

In diesem Abschnitt wird beschrieben, wie Lese- und Schreibanforderungen behandelt werden. Es wird zudem die Idee der *Tunable Consistency* genauer erläutert. Darüber hinaus werden drei Konzepte zum Beheben von Inkonsistenzen betrachtet: (*i*) *Hinted Handoffs*, (*ii*) *Anti-Entropy* und (*iii*) *Read Repair*.

Um Daten zu lesen oder zu schreiben, kann sich ein Client mit jedem beliebigen Knoten verbinden. Dieser Knoten übernimmt dann die Rolle eines *Koordinierenden Knotens*. Schematisch ist dies in der Abbildung 4 dargestellt. Der Knoten E übernimmt dort die Rolle des Koordinierenden Knotens. Dieser leitet die Anfragen des Clients an die zuständigen Knoten weiter. An wie viele Knoten die Anfragen weitergeleitet werden, hängt vom gewählten *Konsistenz-Level* und dem genutzten Replikationsfaktor ab (siehe Abschnitt 2.3.1).



(a) Lesen von Daten (b) Schreiben von Daten

Abbildung 4: Lesen und schreiben von Daten. Der Client verbindet sich mit einem Knoten im Ring. An diesen Knoten sendet er seine Lese- und Schreibanforderungen. Dieser Knoten leitet die Anfragen an die zuständigen Knoten (A, B, C) weiter. Abhängig von der gewählten Konsistenz werden die Anfragen an unterschiedlich viele Knoten weitergeleitet.

2.3.1 Tunable Consistency

Clients spezifizieren bei Lese- oder Schreiboperationen den Konsistenz-Level, welchen sie für die Anfrage wünschen. Die möglichen Konsistenz-Level für das Lesen sind in Tabelle 1, die für das Schreiben in Tabelle 2, aufgeführt. Um so höher der Konsistenz-Level gewählt wird, desto mehr Knoten sind an der Anfrage beteiligt. Dies sorgt für verbesserte Konsistenz, jedoch für schlechtere Performance. Das individuelle Festlegen des Konsistenz-Level bei Anfragen wird als *Tunable Consistency* bezeichnet.

Durch eine entsprechende Wahl von Knoten kann *Read your Writes Konsistenz* [TvS07, S. 424f] erreicht werden. Dies bedeutet, dass auf eine Schreiboperation folgende Leseoperation die geschriebenen Daten sehen muss, sofern sich beide Operationen auf die gleiche Zeile beziehen. Alternativ können beim Lesen der Zeile auch neuere Daten zurückgeliefert werden, falls diese zwischenzeitlich von einem anderen Prozess aktualisiert wurde.

Konsistenz-Level	Bedeutung
ONE	Es werden die Zeilen von dem Knoten zurückgeliefert, welcher als erstes antwortet.
QUORUM	Haben $(\frac{Replikationsfaktor}{2} + 1)$ Knoten geantwortet, werden die Zeilen mit dem neuesten Zeitstempel an den Client ausgeliefert.
ALL	Verhält sich wie QUORUM, jedoch wird mit dem Ausliefern der Zeilen gewartet, bis die Zeilen von allen Knoten vorliegen.

Tabelle 1: Konsistenz-Level von Cassandra beim Lesen von Daten

Konsistenz-Level	Bedeutung
ZERO	Die Schreiboperation wird asynchron bearbeitet. Auftretende Fehler werden ignoriert.
ANY	Die Schreiboperation muss auf mindestens einem Knoten durchgeführt worden sein. Hinted Handoffs sind erlaubt (siehe Abschnitt 2.3.2).
ONE	Die Schreiboperation muss auf mindestens einem Knoten bestätigt worden sein.
QUORUM	Es müssen mindestens $(\frac{Replikationsfaktor}{2} + 1)$ Knoten die Schreiboperation bestätigen.
ALL	Die Schreiboperation muss von allen Knoten bestätigt worden sein, welche für die Daten zuständig sind.

Tabelle 2: Konsistenz-Level von Cassandra beim Schreiben von Daten

Erreicht wird dies, indem mindestens ein Knoten an beiden Operationen beteiligt ist. Dieser Knoten erhält in der Schreiboperation die geänderten Daten. Da der Knoten auch an der Leseoperation beteiligt ist, werden die Daten wieder an den Client ausgeliefert. Da immer die Daten mit dem neusten Zeitstempel an den Client ausgeliefert werden, erhält der Client mindestens den soeben geschriebenen Stand der Daten. Formal kann dies über die Ungleichung $W + R > N$ beschrieben werden. In der Ungleichung steht W für die Anzahl der Knoten, auf denen die Daten geschrieben wurden, R für die Anzahl der Knoten von den die Daten gelesen wurden und N steht für den Replikationsfaktor des Schlüsselraums.

2.3.2 Hinted Handoff

Bei einem *Hinted Handoff* handelt es sich um einen Hinweis für einen Knoten, welcher aktuell nicht erreichbar ist. Hinted Handoffs werden genutzt, um Schreibzugriffe zwischenzuspeichern und später auszuführen, sobald der Zielknoten wieder erreichbar ist. Im Konsistenz-Level ANY reicht das Erstellen eines Hinted Handoffs schon aus, um dem Client das Schreiben der Daten erfolgreich bestätigen zu können, obwohl bislang kein Replikat aktualisiert worden ist.

Beispiel: Der Client C möchte Daten auf dem Knoten A verändern. Der Knoten A ist aktuell nicht erreichbar. Der Client hat sich mit Knoten B verbunden und sendet diesem die Schreibanforderung. Als Konsistenz-Level gibt er ANY an. Dem Knoten B ist es nun erlaubt, die Schreibanforderung zu speichern und dem Client das Schreiben der Daten zu bestätigen. Der Knoten B wartet bis der Knoten A wieder erreichbar ist und sendet diesem daraufhin die Schreibanforderung.

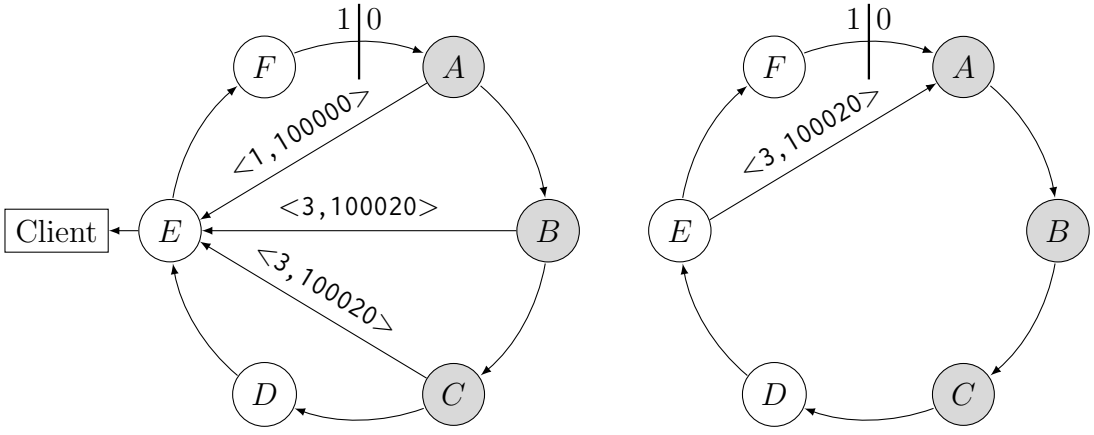
Hinted Handoffs sorgen dafür, dass Schreibzugriffe auch durchgeführt werden können, wenn Knoten nicht erreichbar sind. Zudem sorgen sie dafür, das Knoten schnell auf einen aktuellen Stand gebracht werden, sobald sie wieder erreichbar sind.

2.3.3 Anti-Entropy und Read Repair

Das Konsistenzmodell von Cassandra erlaubt vorübergehende Inkonsistenzen (*Eventual Consistency*). Neben den Hinted-Handoffs wird mit zwei weiteren Techniken gearbeitet, um Inkonsistenzen zu beheben: (*i*) *Anti-Entropy* und (*ii*) *Read Repair*.

Read Repair: Unabhängig vom gewählten Konsistenz-Level, fordert der Koordinierende Knoten bei einem Lesezugriff die Daten von allen Knoten an. Der Koordinierende Knoten überprüft, ob alle erhaltenen Zeilen den gleichen Zeitstempel aufweisen. Sofern auf Knoten veraltete Zeilen vorliegen, initiiert der Koordinierende Knoten einen Schreibzugriff, um die Zeilen zu aktualisieren (Abbildung 5). Der Konsistenz-Level bei lesenden Zugriffen gibt demnach nur an, wann der Koordinierende Knoten dem Client eine Antwort übermittelt, nicht aber, von wie vielen Knoten die Zeilen gelesen werden.

Anti-Entropy: Mittels Read Repair werden Inkonsistenzen für Daten, welche häufig gelesen werden, schnell korrigiert. Inkonsistenzen in Daten, welche nur selten gelesen werden, werden mittels Anti-Entropy korrigiert. Hierzu tauschen die Knoten untereinander *Prüfsummen* über die gespeicherten Daten aus. Verwendet werden dazu *Merkle Trees* [RCM82] um mit möglichst wenig Netzwerkverkehr große Mengen an Daten überprüfen und gegebenenfalls korrigieren zu können.



(a) Lesen einer Zeile von allen Knoten (b) Aktualisieren einer veralteten Zeile

Abbildung 5: Anwendung von Read Repair: In Abbildung (a) fordert der Knoten E eine Zeile von den Knoten A, B und C an. Dabei stellt er fest, dass der Knoten A einen veralteten Stand besitzt. Knoten B und C antworten mit dem Wert `3`, geschrieben bei Zeitstempel `100 020`. Der Knoten A antwortet hingegen mit dem Wert `1` geschrieben bei Zeitstempel `100 000`. Der Knoten wird in Abbildung (b) aktualisiert.

2.3.4 Persistenz

Dieser Abschnitt beschreibt, wie Daten lokal auf einem Knoten persistent gespeichert werden. Schreibzugriffe werden zunächst in einem *Commit Log* festgehalten. Sobald der Schreibzugriff im Commit Log steht, bestätigt der Knoten diesen als erfolgreich. Auch bei einem Programmfehler oder Neustart kann der Schreibzugriff aus dem Commit Log wiederhergestellt werden.

Nachdem der Schreibzugriff im Commit Log festgehalten ist, werden die geänderten Daten im Arbeitsspeicher, in einer *Memtable*, abgelegt. Die Daten sind dort gemäß ihres Zeilenschlüssels sortiert. Überschreitet die Memtable eine gewisse Größe, werden diese Daten auf die Festplatte ausgelagert (*flush*) und die Memtable geleert. Die auf die Festplatte ausgelagerten Daten werden sortiert als *SSTable* (*Sorted String Table*) gespeichert. Da die Daten bereits sortiert im Speicher vorliegen, können diese unverändert auf die Festplatte herausgeschrieben werden.

Sobald die Daten erfolgreich auf die Festplatte geschrieben worden sind, wird das Commit Log geleert. Die dort vermerkten Schreibzugriffe sind nun persistent in der SSTable abgelegt. SSTables sind unveränderlich. Eine einmal geschriebene SSTable kann nach dem Schreiben nicht mehr verändert werden [CAS13c].

Um Speicherplatz zu sparen und die Anzahl der zu verwaltenden SSTables zu reduzieren, werden in regelmäßigen Abständen *Compactions* durchgeführt. Dabei werden die bestehenden SSTables in eine neue SSTable überführt. Veraltete, durch einen Schreibzugriff aktualisierte, Daten werden dabei nicht übernommen. Nachdem die neue SSTable aufgebaut worden ist, werden die bestehenden SSTables gelöscht (siehe Abbildung 6). Das Konzept der Memtable und SSTables stammt aus der Architektur der Software *Google Bigtable* [CDG+08].

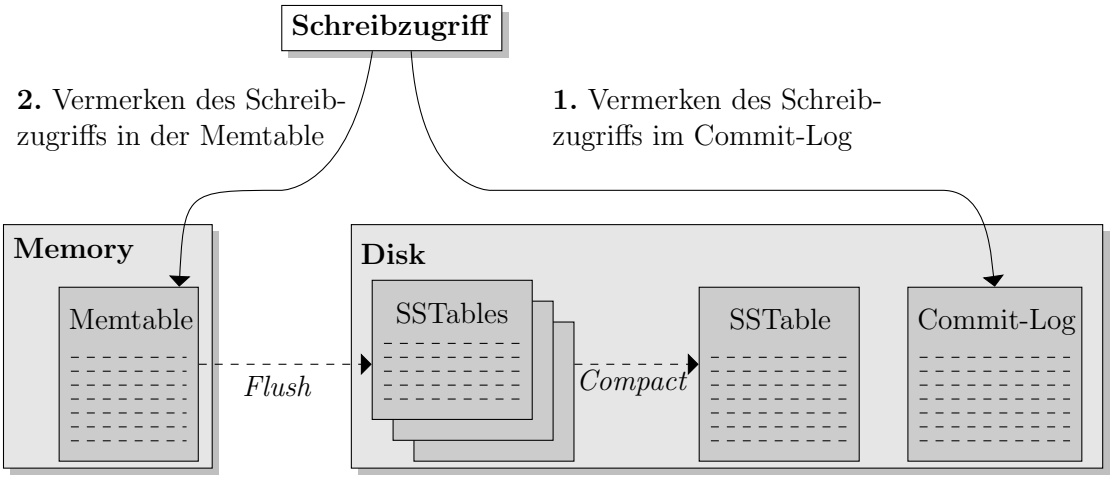


Abbildung 6: Architektur von Cassandra: Zusammenhang zwischen Commit-Log, Memtable, SSTables und Compaction.

Zugriffe auf die Festplatte sind im Vergleich zu Zugriffen auf den Arbeitsspeicher um einige Zehnerpotenzen langsamer. Soll ein Knoten Daten lesen, prüft dieser zuerst, ob die Daten im Arbeitsspeicher, in der Memtable, vorhanden sind. Liegen die Daten dort nicht vor, müssen alle SSTables nach dem neuesten Stand dieser Daten durchsucht werden.

Um diese Suche mit wenig Zugriffen auf die Festplatte durchzuführen, werden *Bloom Filter* eingesetzt [Blo70]. Ein Bloom Filter ist ein nichtdeterministischer Algorithmus, um speichersparend festzustellen, ob ein Wert in einer Sammlung von Werten auftaucht. Der Algorithmus kann zuverlässig entscheiden, ob ein Wert *nicht* Element einer Sammlung ist. Nichtzutreffende positive Antworten (*False Positives*) sind jedoch möglich.

Jeder SSTable wird ein Bloom Filter zugeordnet. Mithilfe der Filter kann festgestellt werden, in welchen SSTables die benötigten Daten *nicht* stehen. Diese SSTables müssen nicht von der Festplatte geladen werden. Hierdurch kann die Anzahl der Zugriffe auf die Festplatte erheblich reduziert werden. Nur auf SSTables, in denen die Daten womöglich stehen, muss zugegriffen werden.

2.4 Sicherheit

Im Standardfall erlaubt Cassandra den Zugriff von beliebigen Clients. Eine Anmeldung ist für den Zugriff auf die Daten nicht erforderlich. Dieser Ansatz geht davon aus, dass sich alle Knoten in einem geschützten Netzwerk befinden. Jeder der Zugriff auf dieses Netzwerk hat, darf auch auf die Daten zugreifen.

Ist dies nicht gewünscht, lässt sich eine Authentifizierung einrichten. Jeder Client, welcher auf die Daten zugreifen möchte, muss sich mit einem Benutzernamen und einem Passwort anmelden. Cassandra bringt hierzu einen `SimpleAuthenticator` mit. Dieser gleicht die Anmeldedaten mit zwei Dateien ab. In der Datei `access.properties` sind die Zugriffsberechtigungen hinterlegt. In der Datei `passwd.properties` sind die Benutzernamen und Passwörter hinterlegt.

Beispiel: In Listing 3 wird der Zugriff auf den Schlüsselraum `Keyspace1` konfiguriert. Die Benutzer `jsmith` und `Elvis Presley` dürfen auf diesen nur lesend zugreifen. Der Benutzer `dilbert` darf zudem auch Daten verändern [DAT13b].

Listing 3: Konfiguration des SimpleAuthenticator - access.properties

```
1 Keyspace1.<ro>=jsmith, Elvis Presley
2 Keyspace1.<rw>=dilbert
```

Die Passwörter für die Anmeldung an Cassandra sind im im Listing 4 angegeben.

Listing 4: Konfiguration des SimpleAuthenticator - passwd.properties

```
1 jsmith=havebadpass
2 Elvis Presley=graceland4ever
3 dilbert=nomoovertime
```

Reichen die vom `SimpleAuthenticator` angebotenen Möglichkeiten nicht aus, so lassen sich eigene *Authenticator-Module* schreiben. Diese können genutzt werden um beispielsweise Benutzer gegen eine Datenbank oder gegen einen *LDAP-Server* zu authentifizieren. Die selbst entwickelten Klassen müssen das Interface `org.apache.cassandra.auth.IAuthenticator` implementieren.

2.5 Performance

Im Jahr 2010 veröffentlichten die Autoren *Avinash Lakshman* und *Prashant Mailk* das erste Paper zu Cassandra [LM10]. In diesem Paper sind auch einige Erfahrungen mit der Performance von Cassandra bei Facebook enthalten. Dort wurde zu dieser Zeit eine Installation von Cassandra auf 150 Systemen, verteilt über zwei Rechenzentren, mit 50+ TB an Daten betrieben. Wie im ersten Abschnitt beschrieben, haben Benutzer auf der Webseite von Facebook die Möglichkeit, sich gegenseitig Nachrichten zu schicken. Diese Nachrichten wurden in dieser Cassandra-Installation gespeichert.

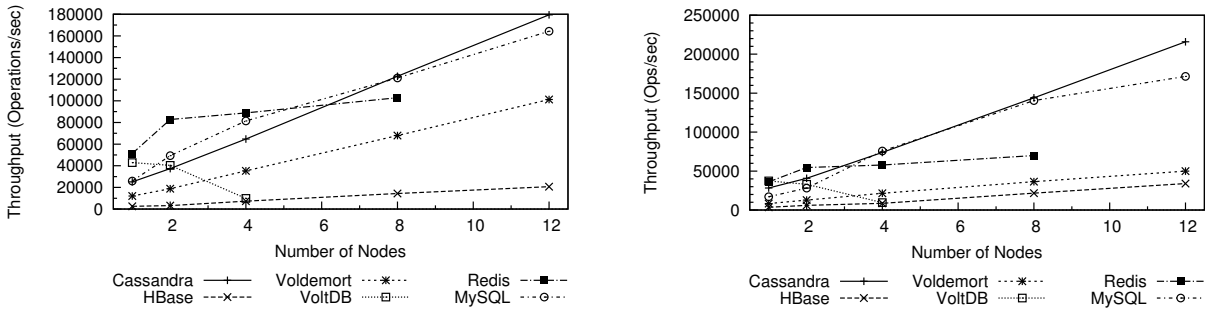
In dem Paper sind die Laufzeiten zweier Anfragen veröffentlicht (siehe Tabelle 3). Beide Anfragen greifen lesend auf die gespeicherten Daten zu. (*i*) *Search Interactions* lädt alle Nachrichten, welche ein Benutzer von einem anderen Benutzer erhalten hat. (*ii*) *Term Search* durchsucht alle Nachrichten eines Benutzers nach einem Schlüsselwort.

Latenz	**Search Interactions**	**Term Search**
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.12 ms	44.41 ms

Tabelle 3: Latenz von Anfragen des *Cassandra-Clusters* bei Facebook (nach [LM10, S. 5])

Im Jahr 2012 veröffentlichten Forscher in ihrem Paper „*Solving big data challenges for enterprise application performance management*“ einen Vergleich der Performance von verschiedenen *Datenbankmanagementsystemen* [RGVS+12]. Für den Vergleich haben die Autoren verschiedene Szenarien mit unterschiedlichen Anfragen konzipiert. Die Ergebnisse von zwei Szenarien werden im folgenden kurz vorgestellt: (*i*) *Workload R* und (*ii*) *Workload RW*.

Im ersten Szenario werden 95% lesende Operationen und 5% schreibende Operationen durchgeführt (Abbildung 7(a)). Im zweiten Szenario erfolgen 50% schreibende und 50% lesende Operationen (Abbildung 7(b)). Zwei Punkte fallen bei diesem Vergleich auf: (*i*) Cassandra skaliert fast linear hinsichtlich der Knoten und der möglichen Operationen in beiden Szenarien. (*ii*) Ab acht Knoten liegt die Performance von Cassandra über der Performance der anderen Systeme.



(a) Workload R

(b) Workload RW

Abbildung 7: Vergleich der Laufzeiten verschiedener *Datenbankmanagementsysteme* (nach [RGVS+12, S. 6f])

3 Erweiterungen von Cassandra

Cassandra wurde seit der ersten Veröffentlichung stark weiterentwickelt. Eine große Community von Entwicklern veröffentlicht alle paar Monate neue Versionen mit neuen Funktionen. Zwei dieser neueren Funktionen werden in diesem Abschnitt aufgegriffen. Dabei handelt es sich zum einen um die Abfragesprache *CQL – Cassandra Query Language* mit der, ähnlich der Sprache *SQL* (*Structured Query Language*), Anfragen formuliert werden können. Zum anderen wird die Anbindung von Hadoop kurz vorgestellt.

3.1 CQL – Cassandra Query Language

Cassandra bietet mehrere Möglichkeiten, auf Daten zuzugreifen. Neben einer Schnittstelle für Client-Bibliotheken, mittels des Protokolls *Thrift* [ASK07], lassen sich die Daten auch über ein *Command Line Interface* (*CLI*) ansprechen. In der Version 0.8 von Cassandra wurde zudem die *Cassandra Query Language* eingeführt.

Die Sprache CQL ist von der Syntax stark an SQL angelehnt. Mittels CQL können Daten gelesen, geändert oder gelöscht werden. Auch strukturelle Änderungen an Spaltenfamilien oder an Schlüsselräumen sind möglich. Zwei Beispiele zum Zugriff mittels CLI und CQL finden sich in den Listings 5 und 6.

Listing 5: Abfrage einer Zeile – Casandra CLI und CQL

```
1 # CLI
2 get People['21'];
3
4 # CQL
5 SELECT * from People WHERE key = 21;
```

Listing 6: Anlegen einer Zeile – Casandra CLI und CQL

```
1 # CLI
2 set users['jsmith'][firstname] = 'John';
3 set users['jsmith'][lastname] = 'Smith';
4 set users['jsmith'][age] = '22';
5
6 # CQL
7 INSERT INTO users (KEY, firstname, lastname, age)
      VALUES ('jsmith', 'John', 'Smith', '22');
```

CQL wurde mit dem Ziel entwickelt, eine stabile und einfache Schnittstelle zu Cassandra bereitzustellen. Zudem sollte die Sprache schnell erlernbar für Anwender mit SQL-Kenntnissen sein. Hierdurch wurde auch die Interaktion mit Anwendungen vereinfacht.

Für die Programmiersprache *Java* existiert durch das Projekt *cassandra-jdbc* [JDB13] ein JDBC-Treiber[2]. Mit diesem Treiber kann auf Cassandra mit den gleichen Methoden wie auf ein RDBMS zugegriffen werden. Zudem wurde beim Design von CQL darauf Wert gelegt,

[2] JDBC - *Java Database Connectivity*

dass nachfolgend keine großen Änderungen an der Syntax der Sprache mehr erfolgen sollen. Dies soll in Zukunft dafür sorgen, dass Cassandra den Anwendungen eine stabile Schnittstelle anbietet. Die Syntax der CLI hat in den letzten Versionen von Cassandra größere Änderungen erfahren. Programme welche per CLI auf Daten zugreifen, müssen daher fortlaufend angepasst werden.

Neben den vielen Ähnlichkeiten besitzen SQL und CQL auch grundlegende Unterschiede. So sind in CQL keine Joins implementiert. Zudem sind in CQL Schlüsselwörter für die Tunable Consistency enthalten. In vielen Anfragen lassen sich Konsistenz-Level angeben. Ein Beispiel hierfür ist in Listing 7 zu finden. In diesem Listing wird mit dem Konsistenz-Level `QUORUM` gearbeitet.

Listing 7: Anlegen einer Zeile unter Angabe eines Konsistenz-Levels

```
1 # CLI
2 consistencylevel as QUORUM;
3 set users['jsmith'][firstname] = 'John';
4 set users['jsmith'][lastname] = 'Smith';
5 set users['jsmith'][age] = '22';
6
7 # CQL
8 INSERT INTO users (KEY, firstname, lastname, age)
      VALUES ('jsmith', 'John', 'Smith', '22')
      USING CONSISTENCY QUORUM;
```

3.2 Integration von Hadoop

Cassandra Datenbanken sind oft sehr groß. Möchte man die dort gespeicherten Daten auswerten, so bietet es sich an, die Daten mittels *Map-Reduce* zu verarbeiten und auszuwerten [DG04]. Ein sehr verbreitetes Open-Source Framework hierfür ist Hadoop.

Für die Ablage von großen Datenmengen bringt Hadoop ein eigenes Dateisystem mit: *HDFS*. Dieses Dateisystem ist auf die redundante Speicherung großer Datenmengen spezialisiert. In Cassandra sind die Daten bereits redundant gespeichert. Nutzt man die klassischen Techniken von Hadoop, so müssen die Daten aus Cassandra exportiert und in HDFS importiert werden, bevor mit diesen gearbeitet werden kann. Neben einer Verdopplung des genutzten Speicherplatzes, benötigt das Kopieren der Daten ins HDFS einige Zeit. Dies sorgt bei großen Datenmengen für deutliche Verzögerungen, bis die eigentliche Auswertung der Daten beginnen kann. Ebenfalls müssen Programme entwickelt werden, welche das Kopieren der Daten übernehmen.

Ab Cassandra Version 0.6 kann Hadoop direkt auf die in Cassandra gespeicherten Daten zugriffen. Ein Export der Daten in HDFS entfällt. Ebenfalls ist es möglich, von Hadoop berechnete Ergebnisse wieder an Cassandra zu übergeben. Konkret stehen hierzu die Klassen `org.apache.cassandra.hadoop.ColumnFamilyInputFormat` und `org.apache.cassandra.hadoop.ColumnFamilyOutputFormat` zur Verfügung. Diese können in eigene Hadoop Programme eingebunden werden. Ebenfalls existiert eine Erweiterung für das *Pig* Framework [ORS+08], welche einen direkten Zugriff auf die in Cassandra gespeicherten Daten erlaubt.

4 Fazit

In dieser Arbeit wurden die Architektur, die Geschichte und einige neuere Funktionen der Software Apache Cassandra vorgestellt. Es wurde auf Themen wie Redundanz, Konsistenz, Peer to Peer und Replikation eingegangen. Ebenso wurde der im Jahr 2012 durchgeführte Performance-Vergleich verschiedener Datenbankmanagementsysteme angesprochen. Dieser bescheinigt Cassandra, in vielen Szenarien, eine höhere Performance als anderen DBMS.

Auch wenn Cassandra heute nicht mehr bei dem ursprünglichen Entwickler (Facebook) eingesetzt wird, nutzen vielen Firmen diese Software für eigene Projekte. Es ist damit zu rechnen, dass aufgrund der rasant wachsenden Datenmengen, auch in Zukunft die Nachfrage nach Cassandra und anderen NoSQL-Datenbanken nicht nachlassen wird.

Literatur

[ASK07] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[CAS13a] Apache cassandra users, 2013. http://planetcassandra.org/Company/-ViewCompany - Abgerufen am 25.04.2013.

[CAS13b] Apache cassandra website, 2013. http://cassandra.apache.org/ - Abgerufen am 25.04.2013.

[CAS13c] Apache Cassandra Wiki - MemtableSSTable, 2013. http://wiki.apache.org/cassandra/MemtableSSTable - Abgerufen am 15.04.2013.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[DAT13a] Apache cassandra documentation der firma datastax inc., 2013. http://www.datastax.com/docs/1.2/index - Abgerufen am 25.04.2013.

[DAT13b] Apache cassandra documentation der firma datastax inc. - authentication, 2013. http://www.datastax.com/docs/1.2/configuration/authentication - Abgerufen am 15.04.2013.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[DGH+87] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In Fred B. Schneider, editor, *PODC*, pages 1–12. ACM, 1987.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[FAC13] Facebook: The underlying technology of messages, 2013. https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919 - Abgerufen am 25.04.2013.

[HDYK04] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The accrual failure detector. In *SRDS*, pages 66–78. IEEE Computer Society, 2004.

[Hew10] E. Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, 2010.

[JDB13] Webseite vom cassandra jdbc-treiber, 2013. http://code.google.com/a/apache-extras.org/p/cassandra-jdbc/ - Abgerufen am 22.04.2013.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[LM09] Avinash Lakshman and Prashant Malik. Cassandra: a structured storage system on a p2p network. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 47–47, New York, NY, USA, 2009. ACM.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[RCM82] Mountain View CA Ralph C. Merkle. Method of providing digital signatures. Patent, 01 1982. US 4309569.

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, August 2012.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.

[WIK13] Apache cassandra in der wikipedia, 2013. http://en.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=545483041 - Abgerufen am 25.04.2013.