Apache Cassandra, wieso brauche ich das?

Jan Ott, Trivadis AG

Apache Cassandra ist eine NoSQL-Datenbank und konkurriert somit nicht nur mit anderen NoSQL-Datenbanken wie zum Beispiel MongoDB, sondern auch mit traditionellen SQL-Datenbanken wie der Oracle-Datenbank.

Wieso brauche ich Cassandra? Diese Frage hat sich der Autor mit seiner 20-jährigen Oracle-Datenbank-Erfahrung gestellt. Die Antwort seiner Kollegen und auf Konferenzen war:

- Einfach horizontal skalierbar mit gängiger Hardware
- Hohe Ausfallsicherheit mittels Redundanz und die Kenntnis von Rack/Data Center
- Herausragende Performance unter anderem durch sogenannte „Tunable Consistancy“
- Kein aufwändiges Programmieren wie mit Map/Reduce; Cassandra hat eine SQL-ähnliche Sprache zur Daten-Manipulation (Cassandra Query Language, CQL)

In den letzten Jahren ist viel passiert. Cassandra findet immer mehr Verwendung. Die Apache Software Foundation und DataStax als Distributor können eine erlauchte Liste von Firmen präsentieren, die Cassandra einsetzen. Hier nur eine Auswahl: Apple, Twitter, Digg, Reddit, ING, UBS und Netflix. Apple betreibt im Moment die wohl größte Cassandra-Installation mit mehr als 75.000 Nodes.

Große Unternehmen setzen Cassandra erfolgreich ein und verarbeiten enorme Datenmengen – dies macht neugierig. Nehmen wir Netflix als Beispiel. Der Film-Streaming-Dienst hat mit einer Oracle-Datenbank angefangen. Leider sind damit mehrere Probleme aufgetreten. Netflix wurde Opfer seines Erfolgs. Die Last nahm enorm zu und die Lösung mit der Oracle-Datenbank stieß an ihre Grenzen. Ein weiteres Problem waren die Offline-Zeiten für Struktur-Änderungen,

denn die Kunden wollen zu jeder Zeit streamen können. Diese Einschränkungen widersprachen den Expansionsplänen von Netflix. Es musste eine andere Datenbank gefunden werden, die die Last tragen kann, Struktur-Änderungen ohne Ausfallzeit erlaubt und flexibel für neue Standorte ist.

Nach einer Evaluationsphase entschied sich Netflix für Cassandra. Ausschlaggebend war, dass Cassandra die Last verarbeiten und für die ehrgeizigen Expansionspläne entsprechend skalieren kann. Dies ist gewährleistet durch die horizontale Skalierung. Mit der Redundanz und dem Verteilen über mehrere Racks und Data Center gibt es keinen „single point of failure". Cassandra verkraftet sowohl den Ausfall einzelner Knoten wie auch des gesamten Data Center. Es braucht zudem kein Zeitfenster, in dem Netflix nicht verfügbar ist, um Schema-Änderungen vorzunehmen; sie können online appliziert werden. Somit stehen für die wichtigsten Problemstellungen entsprechende Lösungen bereit.

Der Streaming-Dienst betreibt mehr als 2.700 Nodes in 90 Clustern verteilt über vier Daten-Center. Cassandra kann die Last von mehr als einer Billion Operationen pro Tag verarbeiten. Wenn Netflix ein neues Land aufsetzten will, ist das in zehn Minuten erledigt. Das war im Jahr 2014 *[1, 2]*.

Woher Cassandra stammt

Cassandra wurde ursprünglich von Facebook entwickelt. Eingeflossen ist das Wissen von Avinash Lakshman und Prashant Malik. Avinash hatte zuvor an Amazon Dynamo gearbeitet. Prashant Malik war bei Microsoft angestellt und arbeitete unter anderem an P2P-Netzwerken. Im Jahr 2008 wurde der Quellcode freigegeben – Open Source. Dadurch konnten auch andere Unternehmen wie LinkedIn, Apple und Twitter zu Cassandra beitragen. Im Jahr 2009 nahm Apache das Projekt auf, 2010 wurde es zu einem „Top Level"-Projekt erklärt und 2011 kam dann die Cassandra Query Language (CQL) hinzu.

Cassandra verwendet verschiedenste Konzepte, um die Anforderungen zu erfüllen. Beginnen wir mit der kleinsten Einheit, dem sogenannten Node. Als Node bezeichnet man einen einzelnen Rechner im Cluster. Es ist zwar möglich, Cassandra auf nur einem Rechner beziehungsweise Node zu betreiben, dies ergibt in der Praxis aber wenig Sinn, da die wichtigsten Vorteile von Cassandra verloren gehen. In der Praxis werden produktive Systeme daher immer als Cluster, bestehend aus mehreren Nodes, betrieben.

Ein einzelner Node ist dabei immer nur für einen Teil der Daten verantwortlich. Diese sollen jederzeit zuverlässig geschrieben und wieder gelesen werden können. Jeder Node muss zudem für jeweils einzelne Anfragen die Aufgabe des Koordinators übernehmen. Mehr dazu später.

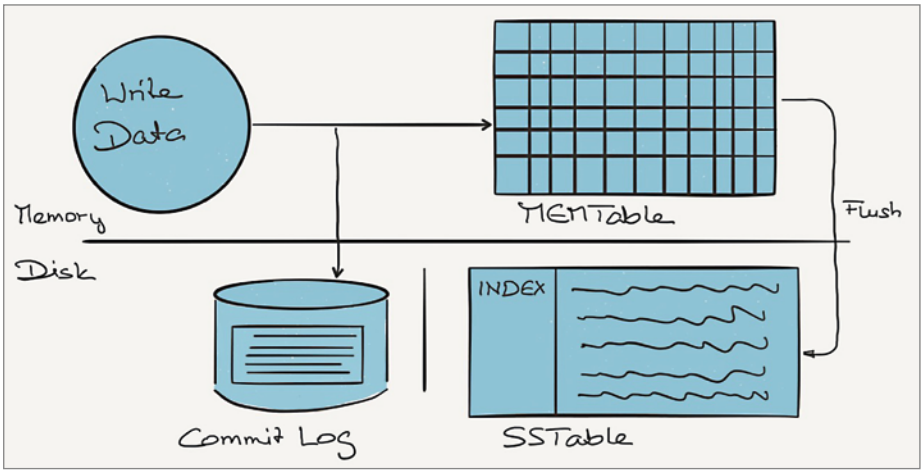

Abbildung 1: Schreiben von Daten in Cassandra

Schreiben und Lesen von Daten

Beim Schreiben von Daten führt ein Node die in *Abbildung 1* dargestellten Schritte durch:

- Schreiben der Daten auf Disk – Commit Log
- Schreiben der Daten ins RAM – MEMTable
- Bestätigung an den Client, dass die Daten geschrieben wurden

Die Daten werden dabei so lange in die MEMTable im Memory geschrieben, bis diese voll ist. Ist dies der Fall, schreibt („flushed") ein Prozess die Daten in ein File auf Disk, eine sogenannte „SSTable". Eine MEMTable wird so zu einer SSTable auf Disk. Ist die erfolgreich, können das Commit Log und MEMTable gelöscht werden. Eine einmal geschriebene SSTable ist unveränderlich.

Für das Lesen der Daten führt ein Node die in *Abbildung 2* dargestellten Schritte durch:

- Prüfen, ob die Daten im Memory sind – MEMTable
- Prüfen, ob die Daten auf den Disks sind – SSTables
- Zusammenführen der gefundenen Daten (dabei kommt ein Zeitstempel zum Zug, den jeder Datensatz und jedes Attribut enthält, wobei immer der neueste Eintrag gewinnt)
- Das Resultat wird zurückgeben

Dies kann zeitaufwändig sein, da die Daten potenziell über mehrere SSTables verteilt sein können. Cassandra führt daher regelmäßig im Hintergrund einen Prozess aus, der die Einträge aus mehreren SSTables vereint und das Resultat in einer neuen SSTable speichert. Dies wird „Compaction" genannt und beschleunigt das Lesen enorm, da eine Row nicht mehr über mehrere SSTables hinweg zusammengeführt werden muss.

Damit nicht immer die ganzen Files gelesen werden, gibt es Bloom Filter, Key Cache und mehr. Diese werden aus Platzgründen hier nicht weiter erläutert. Es reicht zu verstehen, dass ein Node für die Lese-/Schreib-Operationen optimiert ist.
Wichtig ist: Beim Schreiben von Daten führt Cassandra, neben dem Hinzufügen zur MEMTable, lediglich eine Disk-Operation auf dem Commit-Log durch, wobei Daten immer nur am Ende einer Datei angefügt („append only") werden. Sobald

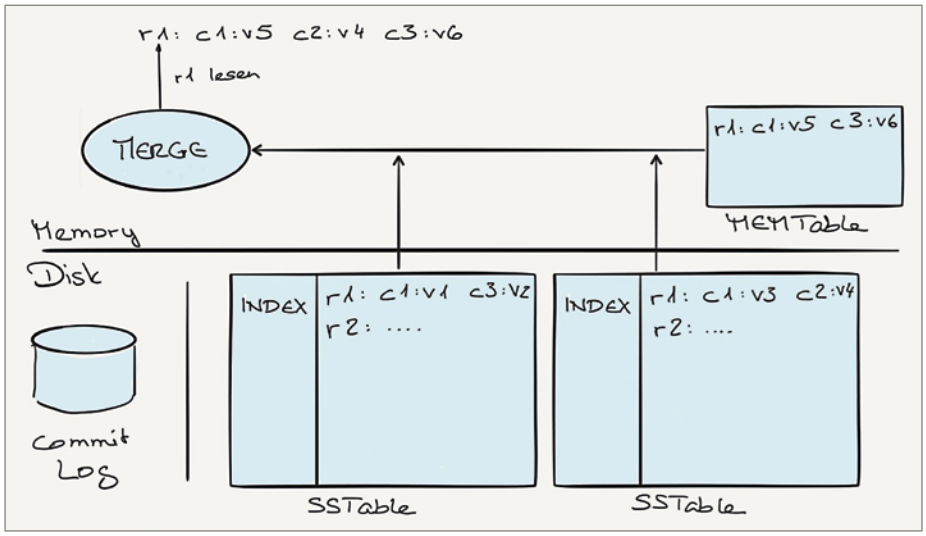

Abbildung 2: Lesen von Daten aus Cassandra

dies getan ist, wird die Bestätigung gegeben, dass alles erledigt ist. Dies ist enorm schnell und erlaubt daher konstant hohe Schreibraten in einen Cassandra Cluster.

Die Hardware

Für den Betrieb eines Nodes wird folgende Hardware empfohlen:

- RAM: 16 – 64 GB
- CPU: 8 Core (im Moment das beste Preis/Leistungs-Verhältnis)
- Disk: 500 GB – 1 TB bis zu einem Maximum von 3 – 5 TB
- Vorzugsweise SSD; HD-Spindeln sind auch möglich, wobei dabei zwingend Commit-Log und Datenfiles voneinander getrennt auf physisch unterschiedlichen Disks geschrieben werden sollen
- Disks müssen direkt mit dem Computer verbunden sein, also kein SAN – wenn es ein Netzwerk-Kabel braucht, ist etwas falsch
- LAN: mindestens 1 GB

Ein so ausgestatteter Node erreicht 3.000 bis 5.000 Schreib-/Lese-Operationen pro Sekunde und Core *[3]*. Anstatt eigene Hardware anzuschaffen, kann man Cassandra natürlich auch in der Cloud betreiben. Netflix verwendet dazu zum Beispiel Amazon Web Services (AWS) und Oracle hat auf der letzten OpenWorld die Unterstützung von Apache Cassandra auf dem Oracle Bare Metal Cloud Service angekündigt *[4]*.

Ein Node lässt sich über das Node-Tool verwalten. Damit kann er gestartet, gestoppt und aus dem Cluster entfernt werden. Zudem lassen sich damit verschiedene Informationen über den Node und Cluster abrufen. Ein Node ist vergleichbar mit einer Ameise im Ameisenhaufen. Alleine nicht sehr stark, aber im Verbund können Daten-Berge versetzt werden. Für das Zusammenspiel solcher Nodes in einem Cluster braucht es eine Organisation. In Cassandra sind die Nodes virtuell in einem Ring angeordnet *(siehe Abbildung 3)*.

Der Cassandra-Cluster

Ein Ring/Cluster ist ein Verbund von mehreren Nodes. Diese haben keinen Master, sämtliche Nodes sind gleichgestellt. Fällt einer aus, wird die Arbeit einfach von einem anderen Node übernommen.

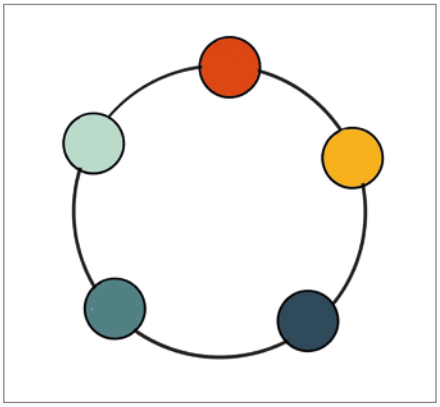

Abbildung 3: Ein Cassandra-Ring

Wie aber werden die Daten in einem Cassandra Cluster gespeichert beziehungsweise wer sorgt dafür, dass die Daten richtig verteilt werden? Diese Aufgabe übernimmt der sogenannte „Koordinator". Jeder Node in einem Cluster kann Koordinator sein, der Koordinator wird von Cassandra jeweils für jede einzelne Anfrage bestimmt und die Zuordnung gilt genau nur für die eine Anfrage. Die Wahl des Koordinators kann über verschiedene Kriterien geschehen, wie zum Beispiel der Latenzzeit des Nodes, die Daten-Lokalität je Node etc. Ist der Koordinator bestimmt, bestehen grundsätzlich die folgenden Möglichkeiten:

- Der Koordinator ist gleichzeitig der Node, der auch die Daten besitzt, er kann die Daten also direkt schreiben beziehungsweise lesen
- Der Koordinator ist nicht für diese Daten zuständig, er muss also einen anderen Node kontaktieren, der die Daten schreibt beziehungsweise liest

Auf welchem Node die Daten gespeichert sind, wird mit dem Partition Key und einer Hash-Funktion eruiert. Eine Hash-Funktion ermittelt das sogenannte „Token", das dann die Datenlokalität im Cluster bestimmt. Dieser Vorgang heißt „Consistent Hashing".

Im folgenden Beispiel wird von einem Replikationsfaktor von drei ausgegangen. Dies bedeutet, dass jeder Datensatz auf drei Nodes repliziert wird, um die Verfügbarkeit zu erhöhen. Jeder Node hält die Daten für einen bestimmten Token-Range, in unserem Beispiel in *Abbildung 4* sind dies für Node A der Token-Range 1 – 7, für Node B 8 – 14 und für Node C 15 – 21. Die Nodes C und D enthalten die Replikate von Node B. Wie auch die anderen Nodes zusätzlich noch Replikationen für andere Nodes speichern. Eine Schreib-Operation läuft wie in *Abbildung 4* dargestellt ab:

1. Der Client verbindet sich mit einem beliebigen Node, der zum Koordinator wird, und sendet ihm die Daten, die geschrieben werden sollen. In unserem Beispiel ist dies Node E.
2. Node E, der Koordinator, empfängt die Daten und bestimmt das Token über den Partition Key, hier im Beispiel ist dies 12.
3. Die Daten werden gleichzeitig an

Node B, C und D gesendet; drei Nodes deshalb, weil der Replikations-Faktor „drei“ ist.
4. Jeder der drei Nodes (B, C, und D) bestätigt dem Koordinator (E), dass er die Daten erhalten und gespeichert hat.
5. Node E, der Koordinator, sendet ein „OK“ zurück an den Client.

Beim Schritt 5 lässt sich über den Konsistenz-Level je Operation steuern, wann die Bestätigung an den Client zurückgeschickt werden soll. So könnte das „OK“ auch schon nach der Bestätigung vom ersten Node (Schritt 4) zurückgegeben werden, was die Latenzzeit für den Client wesentlich verringert, dafür ist die Daten-Konsistenz nicht unmittelbar garantiert. Dies nennt man in Cassandra „Tunable Consistancy“, mehr dazu später. Eine Lese-Operation läuft wie in *Abbildung 5* dargestellt ab:

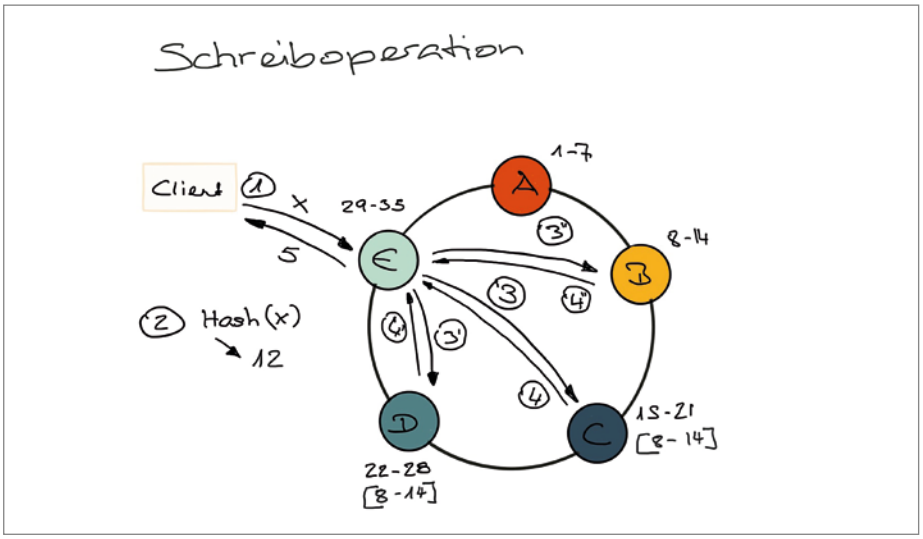

Abbildung 4: Eine Schreib-Operation

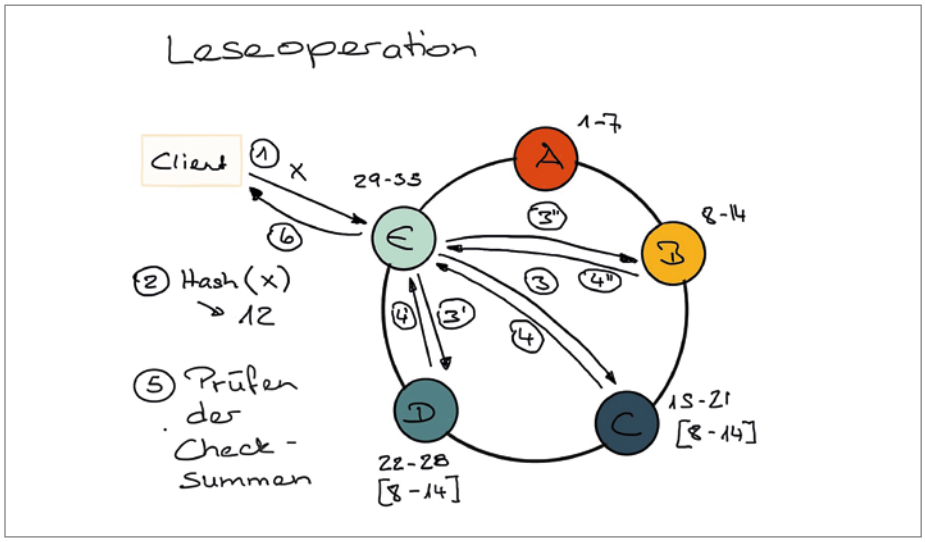

Abbildung 5: Eine Lese-Operation

1. Der Client verbindet sich mit einem beliebigen Node, der zum Koordinator wird, und schickt den Partition Key der Daten, die gelesen werden sollen. In unserem Beispiel ist dies Node E.
2. Node E, der Koordinator, empfängt den Partition Key und ermittelt über dieselbe Hash-Funktion den Token, also wiederum 12.
3. Node E macht eine Abfrage für die Daten auf B inklusive der Checksumme; er bittet Node C und D um eine Checksumme.
4. Node E erhält die Daten und die Checksummen.
5. Node E prüft, ob die Checksummen übereinstimmen.
6. Wenn alles ok ist, werden die Daten an den Client geliefert.

Ob beim Lesen tatsächlich alle drei Replikate angefragt und geprüft werden, hängt vom Konsistenz-Level ab, der bei der Read-Operation angegeben werden kann. Dies nennt man ebenfalls „Tunable Consistancy“.

Tunable Consistancy

Bei verteilten Systemen und dabei auch bei verteilten Datenbanken geht es um drei Faktoren: Consistency, Availability und Partition Tolerance – das sogenannte „CAP-Theorem“ *(siehe Abbildung 6)*. Es ist jedoch nicht möglich, alle drei Faktoren gleichzeitig zu erfüllen. In *[5]* ist das gut erklärt.

Es geht somit nicht, dass man 100 Prozent konsistent, gleichzeitig 100 Prozent verfügbar und noch partitionstolerant sein kann. Also muss ein Kompromiss gefunden werden.

Cassandra setzt die Priorität auf Availability und Partition Tolerance. Das bedeutet aber nicht, dass Cassandra nicht auch konsistent sein kann. Es ist aber dem Entwickler überlassen, wie die Gewichtung erfolgen soll. Dafür steht in Cassandra die Tunable Consistancy zur Verfügung. Hierzu zwei Beispiele: Schreiben-ONE und Schreiben-QUORUM *(siehe Abbildung 7)*:

Consistency Level = ONE:

- Der Client verbindet sich mit B, der die Aufgabe des Koordinators übernimmt.
- B schreibt an alle drei Nodes (C, D und E), die die Daten erhalten müssen.
- Wenn der erste Node die Bestätigung schickt, ist das Consistency Level erfüllt
- B schickt eine Bestätigung zurück an den Client

Consistency Level = QUORUM: Mehr als 50 Prozent der involvierten Nodes müssen bestätigen, in unserem Fall mit Replikations-Faktor „drei“ sind dies zwei Nodes:

- Der Client verbindet sich mit B, der die Aufgabe des Koordinators übernimmt.

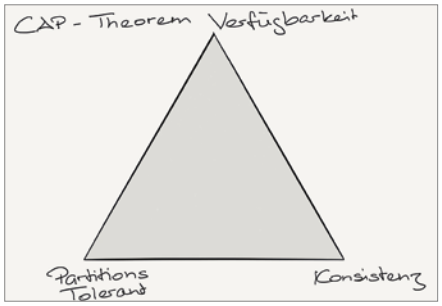

Abbildung 6: Das CAP-Theorem

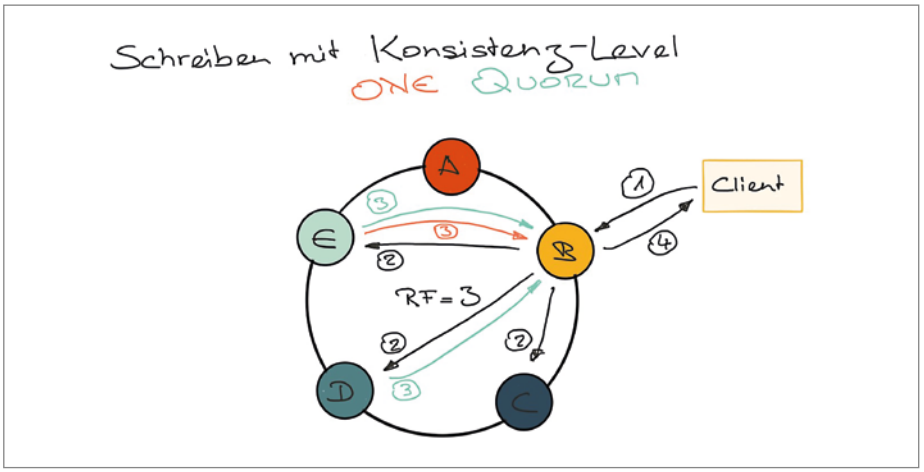

Abbildung 7: Schreiben mit Konsistenz-Level

- B schreibt an alle drei Nodes (C, D und E), die die Daten erhalten müssen.
- Wenn der zweite die Bestätigung geschickt hat, ist das Consistency Level QUORUM erfüllt
- B schickt die Bestätigung an den Client

Es gibt auch noch andere Level:

- ANY: Nur der Koordinator betätigt, dass er die Daten erhalten hat. Sie wurden aber noch nicht an die Nodes geschickt, die die Replikate speichern.
- ALL: Alle Nodes müssen bestätigen.
- ONE, TWO, THREE: Ein, zwei beziehungsweise drei Nodes müssen bestätigen.
- QUORUM: Mehr als 50 Prozent der Nodes müssen bestätigen
- LOCAL_ONE: Nur ein Node aus dem lokalen Data Center muss bestätigen.
- LOCAL_QUORUM: Mehr als 50 Prozent der Nodes des lokalen Data Center müssen bestätigen.
- EACH_QUORUM: Mehr als 50 Prozent der Nodes aus allen Data Center müssen bestätigen.

Dazu ein Beispiel: Bei Netflix ist nicht wichtig, bis zu welcher Sekunde man einen Film gesehen hat, also Level „ANY“ oder „ONE“. Die Filme, die man schon gesehen hat, sollten aber möglichst konsistent angezeigt werden, somit Level „QUORUM“. Ein Entwickler kann mit Cassandra je nach Anwendungsfall und Daten entscheiden, wie hoch die Konsistenz-Sicherheit sein muss, und dadurch die Geschwindigkeit und Verfügbarkeit des Systems beeinflussen – Tunable Consistency.

Ausfallsicherheit

Um die Ausfallsicherheit zu erhöhen, ist bei Cassandra die Unterstützung für Racks und Data Center von Haus aus eingebaut. Die Handhabung ist daher denkbar einfach. Es ist nur ein Parameter im Parameter File zu setzen, und Cassandra verteilt alles richtig über die Data Center hinweg. Im Vergleich dazu ist der Betrieb einer Oracle-Datenbank über mehrere Data Center hinweg sehr komplex. Generell sollte ein Oracle-Datenbank-Cluster wegen der Inter-Node-Kommunikation nicht mehr als ein Dutzend Nodes beinhalten. Doch wie steht das mit einem Ring von Netflix mit 2.700 Nodes? Wie kann das Cassandra handhaben?

In einem Cluster müssen Status-Mitteilungen fortlaufend ausgetauscht werden. Es wäre aber falsch, wenn nun jeder Node allen mitteilen würde, wenn er dem Ring beitritt oder diesen verlässt. Dies würde das Netzwerk zu sehr beanspruchen. Hinzu kommt, dass jeder Node auch mitteilen soll, dass er noch am Leben ist.

Die Cassandra-Entwickler haben sich ein Verhalten der Menschen zunutze gemacht – das Geschwätz – sie nennen es „Gossip“. Ein Mensch erzählt etwas seinen Freunden und Bekannten. Die erzählen es wiederum ihren Freunden und Bekannten und so weiter und so fort. Das Ergebnis ist, dass innerhalb kurzer Zeit alle alles wissen. So funktioniert Gossip auch bei Cassandra. Ein Node teilt drei Nodes mit, dass er da ist. Die drei Nodes geben die Information an je drei Nodes weiter. So erfahren alle Nodes, dass ein neuer im Ring ist. Dieses Verfahren ist sehr effizient und belastet das Netzwerk nur gering. Selbst bei einer Größe von 2.700 Nodes wie bei Netflix. Was wird dabei mitgeteilt? Informationen zum Status, zum Data Center, zu den Racks, zu den Schemata und noch weiteres. Das Ganze hat zudem einen Rhythmus, „Herzschlag“ genannt.

Somit wissen wir nun, wie Cassandra funktioniert. Wir haben ein paar der Methoden gesehen, mit denen auftretende Probleme gelöst werden können. In der nächsten Ausgabe zeigen wir, wie die Daten reingebracht sowie rausgeholt werden und wie man die Strukturen für die Daten erstellt.

Weitere Informationen

[1] Netflix, Fallstudie 2011: *http://www.datastax.com/resources/casestudies/netflix*

[2] Cassandra Summit 2014, Cassandra@Netflix: *http://www.slideshare.net/planetcassandra/cassandra-summit-2014-cassandra-netflix-building-a-house-of-cards-on-a-solid-foundation*

[3] DataStax, Wahl der richtigen Hardware: *https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architecturePlanningHardware_c.html*

[4] Oracle-Ankündigung Cassandra auf „bare metal cloud“-Service: *https://community.oracle.com/community/cloud_computing/bare-metal/blog/2016/10/24/datastax-certified-nosql-cassandra-clusters-on-bare-metal-cloud-service*

[5] CAP-Theorem, Wikipedia: *https://de.wikipedia.org/wiki/CAP-Theorem*

Jan Ott
jan.ott@trivadis.com