HAW HAMBURG

BACHELORTHESIS

Hasan Makki

Vergleich von Cluster- und Skalierungsmechanismen in verteilen Datenbanken

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Hasan Makki

Vergleich von Cluster- und Skalierungsmechanismen in verteilen Datenbanken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Lars Hamann

Eingereicht am: 12. Januar 2025

Hasan Makki

Thema der Arbeit

Vergleich von Cluster- und Skalierungsmechanismen in verteilten Datenbanken

Stichworte

MariaDB, MongoDB, Cassandra, Cluster, Skalierbarkeit

Kurzzusammenfassung

Die vorliegende Ausarbeitung untersucht, wie sich die Datenbankmanagementsysteme MariaDB, MongoDB und Cassandra hinsichtlich ihres Clusteraufbaus und der Skalierbarkeit unterscheiden. Nach einer Einführung in die Grundlagen verteilter Datenbanksysteme, einschließlich Transaktionskonzepten und möglicher Probleme, werden die drei Systeme im Detail analysiert. Ziel ist es, Unterschiede und Gemeinsamkeiten in ihrem Aufbau und ihrer Funktionalität herauszuarbeiten. Abschließend wird bewertet, welches System in verteilten Computerumgebungen am besten geeignet ist.

Inhaltsverzeichnis

Einleitung 7

1 Problemstellung 9

1.1 Skalierbarkeit 9

1.2 Vertikale Skalierung 9

1.3 Das Mooresche Gesetz 11

1.4 Horizontale Skalierung 11

2 Grundlagen 13

2.1 Cluster 13

2.1.1 Was ist ein Cluster? 13

2.1.2 Arten paralleler Datenverarbeitung 13

2.1.2.1 Shared-Everything 14

2.1.2.2 Shared-Disk 15

2.1.2.3 Shared-Nothing 16

2.1.3 Vor- und Nachteile von Datenbankclustern 16

2.1.3.1 Replikation 16

2.1.3.2 Fragmentierung (Sharding) 18

2.1.3.3 Lastverteilung. 19

2.1.3.4 Konsistenzprobleme 19

2.1.3.5 Verfügbarkeit 20

2.1.3.6 Transparenz 20

2.3 CAP-Theorem 21

2.3.1 CAP Theorem im Detail 21

2.3.2 Drei Klassen von verteilten Datenbanksystemen 22

2.3.3 Probleme des CAP-Theorems 23

2.4 Transaktionskonzepte 25

2.4.1 Aufgaben des Transaktionskonzepts. 25

2.4.2 Maßnahmen zur Konsistenzsicherung 26

2.4.2.1 Sperren 26

2.4.2.2 Zwei-Phasen-Commit. 27

2.4.3 ACID 28

2.4.4 BASE 29

2.4.5 ACID vs. BASE. 29

5 Datenbanksysteme im Detail 30

5.1 MariaDB 31

5.1.1 Transaktionskonzept 31

5.1.2 Clusterbildung 33

5.1.2.1 MariaDB Replikation. 33

5.1.2.2 Skalierbarkeit, Verfügbarkeit und Performance. 35

5.1.3 Galera-Cluster 36

5.1.3.1 Certification-based Replikation. 36

5.1.3.2 Skalierbarkeit, Verfügbarkeit und Performance. 38

5.2 MongoDB 40

5.2.1 Transaktionskonzept 40

5.2.2 Clustermöglichkeiten 42

5.2.2.1 Replikation 43

5.2.2.2 Fragmentiertes Cluster 45

5.3 Cassandra 48

5.3.1 Transaktionskonzept 48

5.3.2 Clusteraufbau 52

5.3.2.1 Replikation 51

5.3.2.2 Skalierbarkeit, Verfügbarkeit und Performance. 52

6 Anwendungsbeispiele 53
7 Fazit 55

Literaturverzeichnis 57

Anhang 60

Erklärung 61

Glossar 62

Einleitung

Big-Data wird in der IT immer wichtiger. Die zu speichernden Daten steigen stetig. Doch das Mooresche Gesetz (siehe 1.3) bröckelt: Die prozentuale Leistungssteigerung neuer Prozessorgenerationen ist in den letzten Jahren stetig weniger geworden und die Leistungsfähigkeit eines einzelnen Systems wird vermutlich bald das Maximum erreichen. Leistungsfähigere Hardware ist daher sehr teuer und bringt keinen großen Gewinn an Schnelligkeit. Ein einzelnes System kann die enormen Datenmengen alleine nicht mehr bewältigen. Daher werden im Bereich der Datenbanken sogenannte Cluster (s. 2.1.1), also Zusammenschlüsse mehrerer Rechner verwendet. Verschiedene Datenbanksysteme bieten unterschiedliche Möglichkeiten einen solchen Rechnerzusammenschluss aufzubauen.

Diese Ausarbeitung geht der Frage nach, wie Datenbankcluster aufgebaut werden können, welche Probleme dabei auftreten, und wie die drei Systeme MariaDB, MongoDB und Cassandra die Möglichkeit der Clusterbildung umsetzen.

Zunächst werden Grundlagen vermittelt: Nachdem Kapitel 1.1 die Begriffe Skalierbarkeit, sowie die Unterschiede zwischen horizontaler und vertikaler Skalierung behandelt hat, werden in Kapitel 2.1.1 Rechnerzusammenschlüsse untersucht. Dazu wird erklärt, was ein Cluster ist, es werden Hardwarekonzepte zur parallelen Datenverarbeitung vorgestellt (s. Kap. 2.1.2) und diskutiert, welche Vorteile und Nachteile sich durch den Einsatz von Datenbankclustern ergeben (s. 2.1.3). In diesem Zusammenhang werden u.a. die datenbankspezifischen Themen Replikation (s. 2.1.3.1), Fragmentierung (s. 2.1.3.2), Konsistenz (s. 2.1.3.4), sowie die allgemeinen Themen Transparenz (s. 2.1.3.6), Verfügbarkeit (s. 2.1.3.5) und Lastverteilung (s. 2.1.3.3) behandelt. Anschließend wird in Abschnitt 2.3.1 ein Blick auf das von Eric Brewer im Jahre 2000 aufgestellte CAP-Theorem geworfen und es erfolgt eine kritische Betrachtung der Relevanz seiner Aussage für Anwendungen in der heutigen Zeit (s. 2.3.3). Um Probleme von verteilten Datenbanksystemen besser verstehen zu können, muss ein Blick auf Transaktionen (s. 2.4.1), sowie auf die unter schiedlichen Konzepte ACID (s. Kapitel 2.4.3) und BASE (s. Kap. 2.4.4) geworfen werden.

Anschließend erfolgt eine detaillierte Untersuchung der Datenbanksysteme MariaDB, MongoDB und Cassandra. In Kapitel 5.1 wird MariaDB thematisiert. Es erfolgt ein Blick auf das Transaktionskonzept (s. 5.1.1) und es werden zwei Varianten von Clustern vorgestellt. Zum einen wird erläutert, wie MariaDB das Master-Slave Replikationskonzept umsetzt (s. 5.1.2.1), zum anderen wird in Abschnitt 5.1.3 das auf synchroner Replikation basierende Galeracluster vorgestellt. Kapitel 5.2 behandelt das Datenbankmanagementsystem MongoDB. Es wird ein Blick darauf geworfen, wie

MongoDB die ACID-Kriterien Atomarität, Konsistenz, Isolation und Dauerhaftigkeit handhabt (s. 5.2.1). Danach wer den auch für MongoDB zwei Arten von Cluster präsentiert: Zum einen ein klassisches Replikationscluster (s. 5.2.2.1) auf Master-Slave Basis, zum anderen ein Cluster mit fragmentiertem Datenbestand (s. 5.2.2.2). In diesem Zusammenhang wird in Abschnitt 5.2.2.2 erläutert, nach welchen Kriterien MongoDB die Daten auf Fragmente verteilt. Das Cassandra DBMS wird in Kapitel 5.3 behandelt. Auch hier erfolgt zunächst ein Blick darauf, ob und wie Cassandra die ACID-Kriterien umsetzt (s. 5.3.1). Anschließend wird der Aufbau eines Cassandraclusters thematisiert, welches sich gänzlich von den bis her untersuchten Konzepten unterscheidet (s. Kapitel 5.3.2). In diesem Zusammenhang werden zwei Replikationsstrategien (s. Kap. 5.3.2.1) erklärt: Die SimpleStrategy und die NetworkTopologyStrategy.

Abschließend erfolgt eine Einschätzung und Bewertung, wann welches System eingesetzt werden sollte.

1. Problemstellung

Die Verfügbarkeit von Ressourcen ist sowohl bei den natürlichen Ressourcen unseres Planeten als auch bei den technischen Ressourcen von Computersystemen begrenzt. Die Leistung eines Computers wird im Wesentlichen durch drei Hauptkomponenten bestimmt: Prozessor, Arbeitsspeicher und Festplattenspeicher. Mit dem ständigen Wachstum der Datenmengen und der zunehmenden Komplexität der Anforderungen stößt jedoch jedes System früher oder später an seine Leistungsgrenzen. Um weiterhin effizient mit steigenden Datenvolumina umgehen zu können, müssen IT-Systeme regelmäßig aufgerüstet werden.

Der erste Teil dieser Arbeit behandelt die Herausforderung der Skalierbarkeit. Dabei wird zunächst der Begriff der Skalierbarkeit erklärt (Abschnitt 1.1) und die Unterschiede zwischen vertikaler (Abschnitt 1.2) und horizontaler Skalierung (Abschnitt 1.4) näher beleuchtet. Das Mooresche Gesetz (Abschnitt 1.3) verdeutlicht, warum künftige Entwicklungen besonders auf die horizontale Skalierung abzielen.

1.1 Skalierbarkeit

Skalierbarkeit bezeichnet im Allgemeinen die Fähigkeit eines Systems, seine Kapazität zu erweitern, ohne dabei größere Anpassungen vornehmen zu müssen. Dies kann entweder durch den Austausch vorhandener Hardware gegen leistungsfähigere Komponenten erreicht werden oder durch das nahtlose Hinzufügen zusätzlicher Hardware zum bestehenden System. vgl. [1]

1.2 Vertikale Skalierung

Die vertikale Skalierung bezieht sich darauf, dass ein bestehendes System durch den Austausch von Komponenten wie Prozessor oder Arbeitsspeicher leistungsfähiger gemacht wird. Mit der Installation leistungsstärkerer Hardware verbessert sich die Systemleistung, jedoch nicht proportional zu den entstehenden Kosten. Mit zunehmender Verbesserung des Systems steigen die Kosten für die Leistungssteigerung überproportional. Zudem hat dieser Ansatz natürliche Grenzen, da das System nach einer gewissen Zeit erneut seine Kapazitätsgrenze erreicht. Infolgedessen wird ein weiteres Hardware-Upgrade notwendig, was einen wiederkehrenden Zyklus von kostspieligen Aufrüstungen zur Folge hat. vgl. [1]

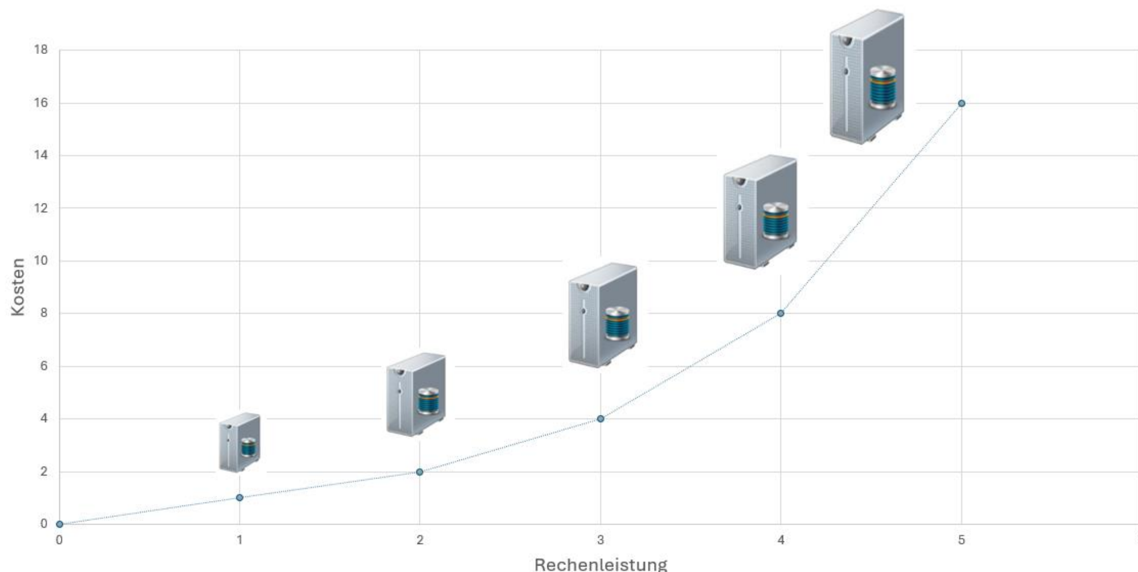


Abb. 1.1: Vertikale Skalierung: Das System wird durch Aufrüsten der Systemhardware immer besser. Ab einem Punkt steigt die Leistung nicht mehr linear an, sondern nähert sich einem Punkt an, ab dem keine größere Entwicklung mehr möglich ist.

Die **vertikale Skalierung** eines Computersystems lässt sich anschaulich mit dem Bau eines Hochhauses vergleichen. Durch den Austausch einzelner Komponenten gegen leistungsstärkere wird das System, wie ein Gebäude, in die Höhe gebaut. Mit jedem zusätzlichen Stockwerk – sprich: mit jedem Upgrade – erhöht sich die Kapazität und Leistungsfähigkeit. Doch ähnlich wie ein Hochhaus irgendwann seine maximale Höhe erreicht, stößt auch die vertikale Skalierung eines Computersystems an unüberwindbare Grenzen. Die Grafik 1.1 verdeutlicht diese Entwicklung zwischen Aufwand und Nutzen.

Der Vergleich der vertikalen Skalierung mit dem Bau eines Hochhauses verdeutlicht anschaulich, dass es physische und technologische Grenzen gibt, die durch die kontinuierliche Verbesserung einzelner Komponenten irgendwann nicht mehr überwunden werden können. Diese Grenzen sind jedoch nicht nur eine Frage der technischen Machbarkeit, sondern auch eng mit der Entwicklung von Prozessoren und der zugrunde liegenden Halbleitertechnologie verbunden.

An dieser Stelle kommt das Mooresche Gesetz ins Spiel, das die historische Grundlage für die bisherige Leistungssteigerung von Hardware bildet und zugleich die Grenzen dieser Entwicklung aufzeigt. (siehe Abschnitt 1.3)

1.3 Das Mooresche Gesetz

Gordon Moore, Mitbegründer von Intel, prognostizierte 1965 in einem Artikel der Zeitschrift „Electronics“, dass sich die Anzahl der Transistoren auf einem Prozessor jährlich verdoppeln würde, während die Hardware gleichzeitig kleiner und effizienter werde. Einige Jahre später passte Moore seine Vorhersage an und reduzierte die Verdopplungsrate der Schaltkreiskomponenten auf alle zwei Jahre. Dieses als Mooresches Gesetz bekannte Phänomen hat die Entwicklung der Halbleiterindustrie über viele Jahrzehnte hinweg stark beeinflusst. Doch es ist absehbar, dass diese Wachstumsrate nicht endlos fortgesetzt werden kann. In den letzten Jahren ist bereits spürbar geworden, dass die Leistungssteigerungen von einer Prozessorgeneration zur nächsten stagnieren oder langsamer ausfallen.

Fachleute prognostizieren, dass das Mooresche Gesetz seine Grenzen erreicht, wenn die Halbleitertechnologie einen Fertigungsprozess von etwa fünf Nanometern erreicht. Ab diesem Punkt werden Leistungssteigerungen nur noch durch größere Hardware oder durch das Stapeln von Schaltkreisen, wie es beispielsweise bei der 3D-NAND-Technologie in Flashspeichern praktiziert wird, möglich sein. Ohne bahnbrechende technologische Fortschritte, wie etwa Quantencomputer oder revolutionäre Hardware-Ansätze, wird es künftig keine signifikanten Leistungszuwächse mehr geben.

Für Datenbanksysteme hat das erhebliche Konsequenzen: Eine weitere vertikale Skalierung wird, sobald diese technologische Grenze erreicht ist, nicht mehr möglich sein, da keine leistungsfähigere Hardware verfügbar sein wird. Um diesen Einschränkungen zu begegnen, wird die horizontale Skalierung zunehmend wichtiger. Sie bietet die Möglichkeit, durch die Hinzufügung zusätzlicher Systeme die Leistung zu steigern, anstatt sich ausschließlich auf stärkere Hardware zu verlassen (siehe Abschnitt 1.4). vgl. [2]

1.4 Horizontale Skalierung

Die Grenzen der vertikalen Skalierung machen alternative Ansätze notwendig. Die **Horizontale Skalierung** bietet hier eine vielversprechende Lösung. Anstatt einzelne Komponenten zu ersetzen, wird das System durch Hinzufügen weiterer, oft identischer, aber nicht zwingend gleich leistungsfähiger Rechner erweitert. Dies gleicht dem Bau weiterer Gebäude neben einem bestehenden Wolkenkratzer, wodurch die Gesamtfläche und damit die Kapazität des Komplexes erhöht wird.

Dank Techniken wie Replikation und Loadbalancing erhöht die horizontale Skalierung nicht nur die Systemleistung, sondern auch die Ausfallsicherheit. Sollte eine Komponente ausfallen, bleibt das Gesamtsystem funktionsfähig, da die Last auf die verbleibenden Maschinen verteilt wird. Dies führt zu einer verbesserten Verfügbarkeit

und einer höheren Fehlertoleranz, was besonders in großen verteilten Systemen von entscheidender Bedeutung ist. Zudem bietet die horizontale Skalierung eine flexible und

kosteneffiziente Möglichkeit, das System je nach Bedarf zu erweitern, ohne dass teure und aufwendige Hardware-Upgrades notwendig sind.

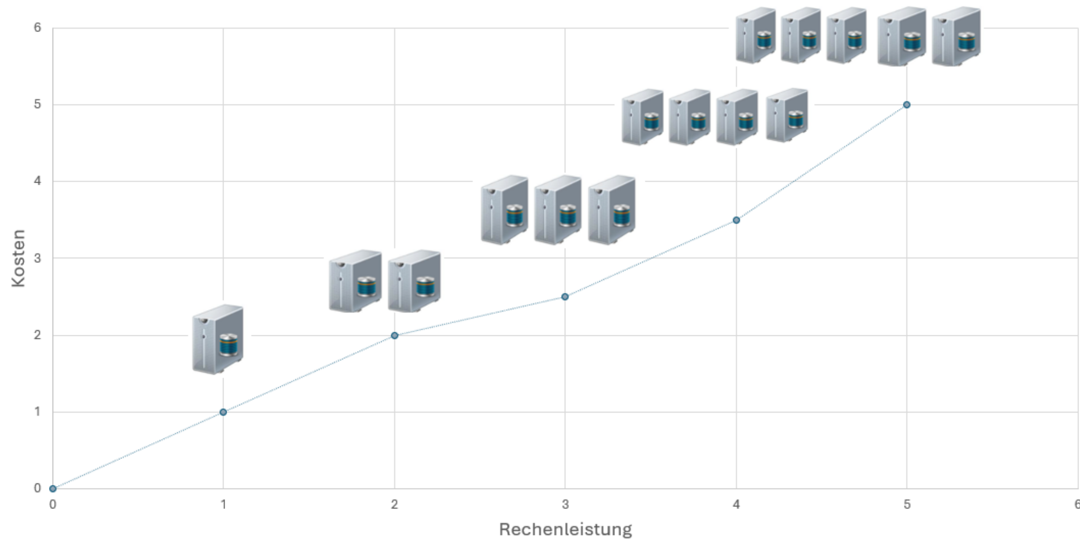


Abb. 1.2: Horizontale Skalierung: Wird mehr Rechenleistung benötigt, werden zusätzliche Rechner in das System integriert. Die Kosten steigen bei Integration zusätzlicher Hardware nahezu linear an.

Im Gegensatz zur vertikalen Skalierung, bei der die Leistungssteigerung mit zunehmenden Kosten immer weniger effizient wird, erfolgt die Leistungszunahme bei der horizontalen Skalierung im Idealfall nahezu linear. Das bedeutet, dass die Leistung mit der Hinzufügung weiterer Ressourcen proportional zunimmt, wie es in den Diagrammen 1.2 durch die Linie veranschaulicht wird. (vgl. [3], S.5).

2. Grundlagen

2.1Cluster

In Kapitel 2 werden verteilte Computersysteme in drei Schritten behandelt. Zunächst werden in Abschnitt 2.1 die Grundlagen von Netzwerken gelegt. Anschließend werden in Abschnitt 2.2 Techniken für den parallelen Datenzugriff und die Ressourcenverteilung vorgestellt. Abschließend wird in Abschnitt 2.3 der Einsatz von Clustern in Datenbanken analysiert.

2.1.1 Was ist ein Cluster?

Ein Cluster besteht aus mehreren Knoten (Rechnern), die über ein Hochgeschwindigkeitsnetzwerk miteinander verbunden sind. Diese Netzwerke lassen sich anhand verschiedener Kriterien unterscheiden, wobei die bekanntesten Kategorien das LAN (Local Area Network) und das WAN (Wide Area Network) sind.

Ein LAN ist ein lokales Netzwerk, das Computer in einem begrenzten Bereich, wie einem Gebäude, verbindet. Im Gegensatz dazu beschreibt WAN ein Netzwerk, das weiträumige Verbindungen umfasst, oft über das öffentliche Internet. Da die Übertragungsrate und die Latenz in einem LAN in der Regel besser sind als in einem WAN, befinden sich die Computer eines Clusters oft in räumlicher Nähe zueinander. Um eine optimale Leistung und Verwaltung zu gewährleisten, werden diese Systeme in der Regel in standardisierten Serverschränken (Server Racks) untergebracht und über dedizierte Netzwerkswitches vernetzt. (vgl. [3], S.37)

Ein Datenbankcluster ist ein logischer Zusammenschluss mehrerer Datenbank-Server, die gemeinsam als ein einziges System agieren. Im Gegensatz zu einem physischen Netzwerk (wie einem LAN oder WAN) bezieht sich ein Datenbankcluster auf die softwareseitige Koordination und Zusammenarbeit der beteiligten Server. Diese können sich auch an verschiedenen Standorten befinden und über VPN-Verbindungen miteinander kommunizieren. (vgl. [4], S.3)

2.1.2 Arten paralleler Datenverarbeitung

Ein Rechnernetz besteht häufig aus mehreren Computern, die nach dem Shared-Nothing-Prinzip arbeiten, wobei jeder Rechner eigene Ressourcen wie Prozessor und Speicher besitzt (siehe Kapitel 2.2.3). Parallelverarbeitung findet jedoch auch auf Einzelrechnern mit mehreren Prozessorkernen statt, die sich Hardware wie Arbeitsspeicher teilen. Dies wird als Shared-Everything-System bezeichnet (siehe Kapitel 2.2.1).

2.1.2.1 Shared-Everything

Das Shared-Everything-Prinzip charakterisiert Systeme, in denen mehrere aktive Prozesse gleichzeitig auf einen gemeinsamen Pool von Systemressourcen zugreifen. Jeder Computer mit mehr als einem Prozessorkern ist ein Beispiel für diese Architektur. Die Prozessorkerne teilen sich dabei nicht nur den Arbeitsspeicher und die Festplatte, sondern auch schnellere Cache-Speicher und die Datenübertragungswege innerhalb des Systems. Diese gemeinsame Nutzung von Ressourcen ermöglicht eine hohe Parallelität und Effizienz bei der Ausführung von Aufgaben.

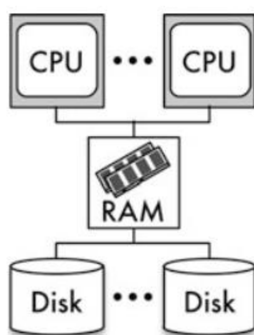


Abb. 2.1: Schematische Darstellung eines Shared-Everything Systems[1].

Die genaue Verteilung von Aufgaben auf die einzelnen Prozessorkerne wird dabei vom Betriebssystem mittels eines Prozess-Schedulers gesteuert. Für die Datenbanksoftware bleibt diese Verteilung meist im Hintergrund. Um die Vorteile eines Shared-Everything-Systems voll auszuschöpfen, müssen Datenbanksysteme jedoch spezifische Mechanismen implementieren. Dazu gehören beispielsweise Multithreading zur gleichzeitigen Bearbeitung mehrerer Aufgaben, das Speichern von Sperrtabellen im gemeinsamen Speicher zur Koordinierung von Zugriffen auf Daten und die Verwaltung von Auftragswarteschlangen zur effizienten Verteilung von Arbeitseinheiten.
(vgl. [3], S.54)

[1]Bildquelle: [entnommen aus Quelle 3, S.54, Abb. 3.7].

2.1.2.2 Shared-Disk

Shared-Disk-Systeme zeichnen sich durch die gemeinsame Nutzung eines externen Speichers aus, auf den mehrere Rechner über ein Netzwerk zugreifen können. Dieser gemeinsame Speicher wird oft in einem Storage Area Network (SAN)[1] bereitgestellt und über Netzwerkprotokolle wie NFS[2] angebunden. In einem solchen Cluster teilen sich alle Rechner denselben Datenbestand. Obwohl die einzelnen Rechner selbst häufig als Shared-Nothing-Systeme konfiguriert sind.

Die Synchronität der Daten ist in Shared-Disk-Systemen gewährleistet, da alle Änderungen am gemeinsamen Speicher unmittelbar für alle Rechner sichtbar sind. Diese Eigenschaft macht Shared-Disk-Systeme besonders gut geeignet für Anwendungen, bei denen eine hohe Leseleistung erforderlich ist, da die Last auf mehrere Rechner verteilt werden kann.

Ein weiterer Vorteil von Shared-Disk-Systemen liegt in ihrer Skalierbarkeit. Sowohl die Rechenleistung (durch Hinzufügen weiterer) als auch die Speicherkapazität (durch Erweiterung des SAN) können unabhängig voneinander erhöht werden. Dadurch können Shared-Disk-Systeme flexibel an wachsende Anforderungen angepasst werden.
(vgl. [3], S.55)

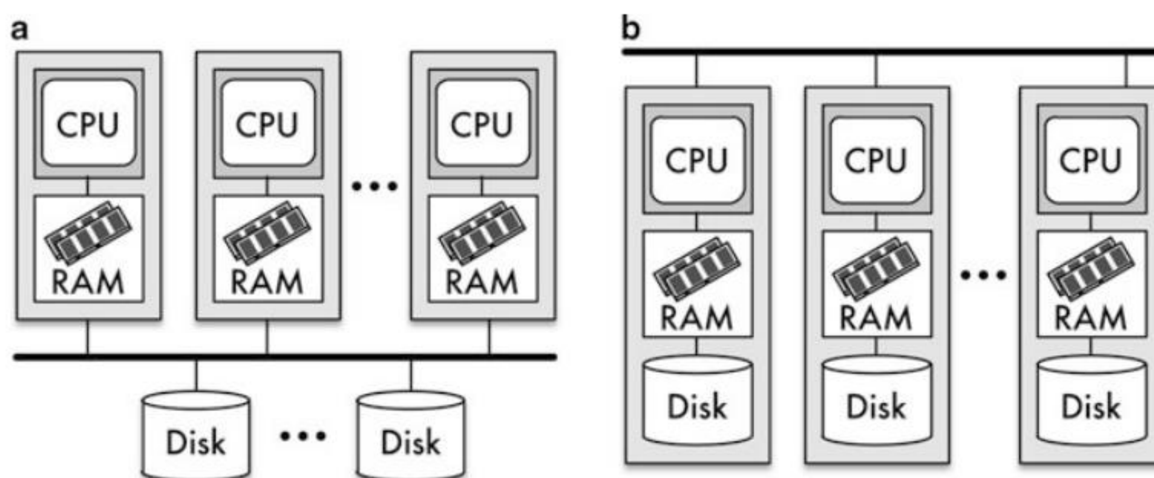


Abb. 2.2: Schematische Darstellung von Shared-Disk (a) und Shared-Nothing (b) Systemen[3].

[1]Bildquelle: [entnommen aus Quelle 3, S.56, Abb. 3.9]

2.1.2.3 Shared-Nothing

Shared-Nothing-Systeme folgen einer Architektur, bei der jeder Knoten (Rechner) über eigene, exklusive Hardware-Ressourcen verfügt. Das bedeutet, jeder Knoten besitzt einen eigenen Prozessor, eigenen Arbeitsspeicher und eigenen Festplattenspeicher. Diese Unabhängigkeit ermöglicht eine hohe Skalierbarkeit, da neue Knoten einfach hinzugefügt werden können, ohne bestehende zu beeinträchtigen.

Um die Daten zu verwalten, wird die Gesamtdatenmenge üblicherweise auf die verschiedenen Knoten aufgeteilt. Dadurch wird eine automatische Lastverteilung erreicht, da jeder Knoten nur für die Daten verantwortlich ist, die lokal gespeichert sind.

Die Kommunikation zwischen den Knoten findet über das Netzwerk statt und erfordert oft komplexe Protokolle, um die Datenkonsistenz zu gewährleisten.

Ein großer Vorteil von Shared-Nothing-Systemen liegt in ihrer horizontalen Skalierbarkeit. Durch Hinzufügen weiterer Knoten kann die Rechenleistung und Speicherkapazität des Systems nahezu linear erhöht werden. Diese Eigenschaft macht Shared-Nothing-Systeme besonders attraktiv für große, datenintensive Anwendungen. (vgl. [3], S.55)

2.1.3 Vor- und Nachteile von Datenbankclustern

Datenbankcluster bieten eine Reihe von Vorteilen, aber auch einige Herausforderungen. Ein herausragender Vorteil ist die flexible Skalierbarkeit, insbesondere durch horizontale Skalierung. Dadurch können zusätzliche Ressourcen unkompliziert und kostengünstig hinzugefügt oder entfernt werden, um den wachsenden Anforderungen gerecht zu werden, ohne dass ein komplettes System-Upgrade erforderlich ist. Diese Dynamik macht Datenbankcluster besonders anpassungsfähig. Im Folgenden werden weitere Vor- und Nachteile beschrieben.

2.1.3.1 Replikation

Die Datenbankreplikation ist ein Mechanismus, der die gezielte Schaffung von Redundanzen in einem Datenbanksystem ermöglicht. Im Gegensatz zum logischen Datenbankentwurf, der Redundanzen in der Regel vermeidet, dienen sie hier dazu, die Verfügbarkeit und Ausfallsicherheit des Systems zu erhöhen.

Die grundlegende Idee der Replikation besteht darin, eine oder mehrere Kopien eines Datenbestands oder eines Teils davon an verschiedenen Standorten zu erstellen und zu synchronisieren. Eine weit verbreitete Form der Replikation ist die Master-Slave-

Replikation. Hierbei übernimmt ein Knoten (der Master) die Rolle des primären Datenbesitzers. Alle Änderungen an den Daten werden zunächst am Master vorgenommen. Die anderen Knoten (Slaves) werden regelmäßig mit den Änderungen am Master synchronisiert.

Sollte der Master ausfallen, kann einer der Slaves zur Übernahme der Master-Rolle befördert werden. Dies gewährleistet eine hohe Verfügbarkeit des Systems, da der Betrieb ohne Unterbrechung fortgesetzt werden kann. (vgl. [3], S. 285)

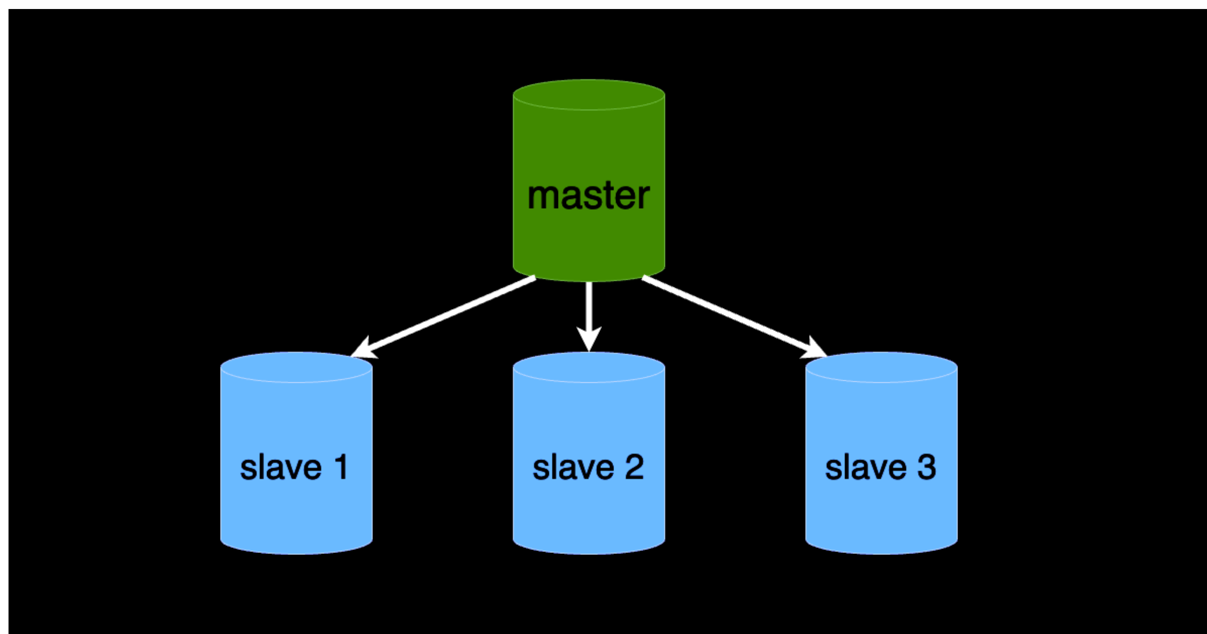


Abb. 2.3: Schematische Darstellung einer Master-Slave Replikation. Der Masterknoten repliziert seinen Datenbestand auf den Slave-knoten[1].

Synchrone-Replikation

Die synchrone Replikation gewährleistet eine hohe Konsistenz zwischen den Replikaten eines Datenbanksystems. Bei diesem Ansatz werden Änderungen an den Daten erst dann als erfolgreich betrachtet und dauerhaft gespeichert, wenn sie auf allen beteiligten Knoten des Replikationssatzes vollständig durchgeführt wurden. Dieser Mechanismus erfordert eine enge Koordination zwischen den Knoten und eine zuverlässige Kommunikation.

Um diese hohe Konsistenz zu erreichen, werden in der synchronen Replikation unterschiedliche Transaktionsprotokolle eingesetzt. Diese Protokolle definieren die Reihenfolge, in der Änderungen an den Daten vorgenommen werden und wie die Konsistenz zwischen den Knoten überprüft wird. Im Kapitel 4 werden diese Transaktionsprotokolle detailliert beschrieben. vgl. [5]

[1]Bildquelle: https://victoronsoftware.com/posts/mysql-master-slave-replication/; abgerufen am 04.11.2024

Asynchrone-Replikation

Die asynchrone Replikation ermöglicht eine schnelle Verarbeitung von Transaktionen, da Änderungen zunächst nur auf einem einzelnen Knoten (meist dem Master) vorgenommen werden. Der Benutzer erhält umgehend eine Bestätigung, auch wenn die Änderungen noch nicht auf allen Replikaten vorhanden sind. Diese asynchrone Vorgehensweise führt jedoch zu einer zeitlichen Diskrepanz zwischen den Datenständen der einzelnen Knoten.

Ein wesentlicher Vorteil der asynchronen Replikation ist die hohe Performanz und die gute Skalierbarkeit. Durch die Entkopplung der Transaktionsverarbeitung von der Replikation können hohe Durchsätze erreicht werden. Allerdings birgt diese Methode auch das Risiko von Dateninkonsistenzen, insbesondere bei Ausfällen. Wenn ein Knoten ausfällt, bevor eine Änderung auf alle Replikate übertragen wurde, können die Daten in einem inkonsistenten Zustand sein. vgl. [5]

2.1.3.2 Fragmentierung (Sharding)

Die Fragmentierung von Datenbanktabellen ist eine Technik, die in verteilten Systemen eingesetzt wird, um große Datenmengen effizienter zu verwalten und die Leistung zu verbessern. Dabei wird eine Tabelle in kleinere, unabhängige Fragmente aufgeteilt, die dann auf verschiedene Rechner eines Clusters verteilt werden können. Dabei wird zwischen horizontaler und vertikaler Fragmentierung unterschieden, wie in Bild 2.4 veranschaulicht wird.

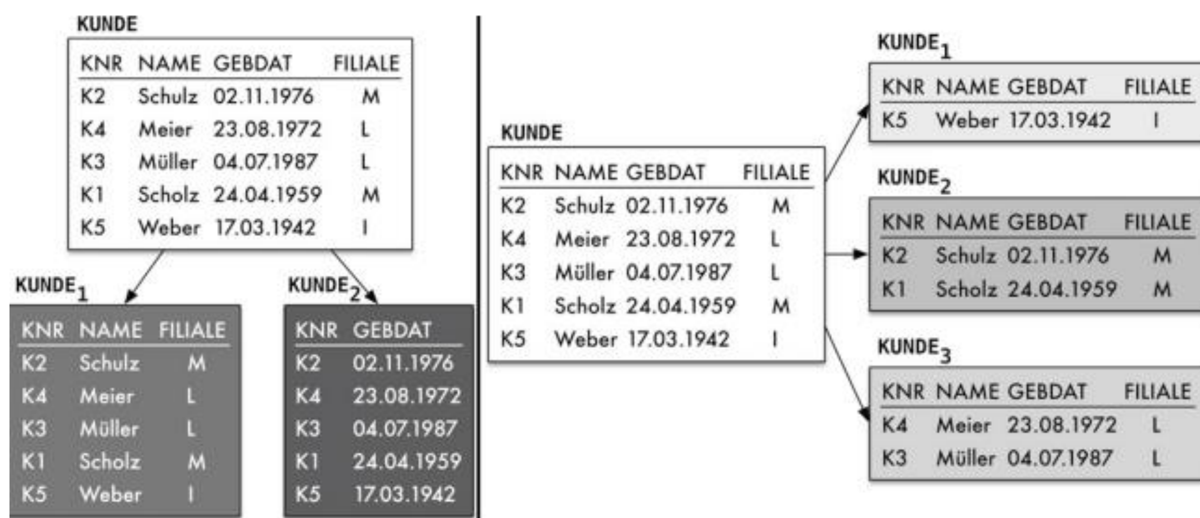


Abb. 2.4: Aufteilung einer Tabelle nach vertikaler Fragmentierung (links) und horizontaler Fragmentierung (rechts).[1]

[1]Bildquelle: [entnommen aus Quelle 3, S.95, Abb. 5.3 und S.98 Abb. 5.5].

Bei Horizontale Fragmentierung: Hierbei wird eine Tabelle entlang der Zeilen aufgeteilt. Das bedeutet, dass bestimmte Zeilen (Tupel) in ein Fragment und andere Zeilen in ein anderes Fragment verschoben werden. Die Aufteilung erfolgt in der Regel basierend auf einem bestimmten Attribut oder einer Kombination von Attributen. Beispielsweise könnten alle Kundendaten für deutsche Kunden in einem Fragment und alle Kundendaten für französische Kunden in einem anderen Fragment gespeichert werden.

Im Gegensatz dazu ist Vertikale Fragmentierung: Bei der vertikalen Fragmentierung wird eine Tabelle entlang der Spalten aufgeteilt. Das bedeutet, dass bestimmte Spalten (Attribute) in ein Fragment und andere Spalten in ein anderes Fragment verschoben werden. Diese Art der Fragmentierung ist nützlich, wenn verschiedene Benutzergruppen nur auf bestimmte Attribute einer Tabelle zugreifen müssen. Um die Daten bei Abfragen wieder zusammenführen zu können, sind Fremdschlüssel erforderlich. (vgl. [4], S.75)

2.1.3.3 Lastverteilung

Ein wesentlicher Vorteil der Datenfragmentierung liegt in der Möglichkeit zur parallelen Verarbeitung. Große, komplexe Abfragen können auf die einzelnen Fragmente aufgeteilt werden, sodass diese parallel berechnet werden können. Dies führt zu einer signifikanten Beschleunigung der Abfragebearbeitung, insbesondere bei großen Datenmengen.

Nehmen wir beispielsweise eine Datenbank, die Kundendaten speichert. Durch eine horizontale Fragmentierung nach Ländern können Abfragen, die sich nur auf Kunden aus einem bestimmten Land beziehen, auf das entsprechende Fragment beschränkt werden. Die anderen Fragmente müssen nicht beteiligt sein, was die Antwortzeit erheblich verkürzt. (vgl. [3], S.97)

2.1.3.4 Konsistenzprobleme

Die Verteilung von Daten und Berechnungen auf mehrere Knoten in einem verteilten System bietet zahlreiche Vorteile, wie beispielsweise eine verbesserte Skalierbarkeit und Verfügbarkeit. Allerdings bringt diese Verteilung auch Herausforderungen mit sich, insbesondere im Hinblick auf die Datenkonsistenz.

Wenn mehrere Knoten gleichzeitig auf dieselben Daten zugreifen und diese modifizieren, besteht die Gefahr von Inkonsistenzen. Um diese zu vermeiden, müssen Mechanismen zur Synchronisation der Knoten eingeführt werden. Ein weit verbreitetes Verfahren ist das Zwei-Phasen-Commit-Protokoll (siehe Abschnitt. 4.2.3), das sicherstellt, dass alle beteiligten Knoten eine Transaktion entweder vollständig committen oder vollständig abbrechen.

Die Frage nach der Konsistenz ist eng verknüpft mit dem ACID-Modell. Dieses Modell definiert strenge Anforderungen an die Atomizität, Konsistenz, Isolation und Dauerhaftigkeit von Transaktionen. Während einige Datenbanksysteme strikt auf die ACID-Konsistenz setzen, verfolgen andere einen flexibleren Ansatz. So gibt es das Konzept der eventuellen Konsistenz, bei dem die Konsistenz nicht sofort nach einer Änderung garantiert wird, sondern erst nach einer gewissen Zeit. (vgl. [4], S.461)

2.1.3.5 Verfügbarkeit

Datenbankcluster bieten durch ihre verteilte Architektur eine hohe Verfügbarkeit und Skalierbarkeit. Allerdings birgt diese Verteilung auch Herausforderungen, insbesondere wenn Teile des Clusters ausfallen oder die Netzwerkverbindung zwischen den Knoten unterbrochen wird. In solchen Situationen ist die Partitionstoleranz von entscheidender Bedeutung.

Partitionstoleranz beschreibt die Fähigkeit eines Systems, weiterhin korrekt zu funktionieren, auch wenn es in mehrere unabhängigen Teile (Partitionen) zerfällt.
(vgl. [4], S.12).

Weitere Details zu dieser Thematik werden in Kapitel 3.1, CAP-Theorem behandelt.

2.1.3.6 Transparenz

Die Transparenz von Datenbanken ist ein entscheidendes Kriterium für eine einfache und effiziente Nutzung. Dies bedeutet, dass ein Benutzer oder eine Anwendung, die auf eine Datenbank zugreift, sich keine Gedanken darüber machen muss, ob es sich um ein einzelnes System oder um ein verteiltes System handelt.

Um diese Transparenz in einem Datenbankcluster zu gewährleisten, werden häufig sogenannte Kontrollserver oder Steuerungsinstanzen eingesetzt. Diese Instanzen fungieren als zentrale Anlaufstelle für alle Benutzeranfragen. Wenn eine Anfrage eingeht, analysiert der Kontrollserver die Anfrage und leitet sie an die entsprechenden Knoten im Cluster weiter, auf denen die benötigten Daten gespeichert sind. Die Ergebnisse der einzelnen Knoten werden dann vom Kontrollserver zusammengeführt und dem Benutzer zurückgegeben.

Ein weiterer wichtiger Vorteil von Kontrollservern ist die Möglichkeit zur Lastverteilung. Durch die intelligente Verteilung von Anfragen auf die verschiedenen Knoten kann die Auslastung des Clusters optimiert werden. Dies führt zu einer besseren Leistung und einer höheren Verfügbarkeit. (vgl. [4], S.7).

2.3 CAP-Theorem

Kapitel 3 bietet eine umfassende Auseinandersetzung mit dem CAP-Theorem. In Abschnitt 3.1 wird die Kernidee des Theorems vorgestellt. Anschließend werden in Abschnitt 3.2 die weitreichenden Konsequenzen des CAP-Theorems für die Gestaltung und den Betrieb verteilter Systeme diskutiert. Abschließend wird in Abschnitt 3.3 kritisch reflektiert, warum das CAP-Theorem in der Fachwelt so viel Aufmerksamkeit erlangt hat und welche Bedeutung es für die Praxis besitzt.

2.3.1 CAP-Theorem im Detail

Das CAP-Theorem wurde erstmals im Jahr 2000 von dem Informatiker Eric Brewer vorgestellt. Es beschreibt ein zentrales Dilemma, dem Entwickler und Betreiber verteilter Systeme gegenüberstehen. Zwei Jahre später bestätigten die Informatiker Seth Gilbert und Nancy Lynch diese theoretische Aussage in einem formalen Beweis. Die Abkürzung CAP steht für drei grundlegende Eigenschaften, die verteilte Systeme betreffen: Konsistenz ("Consistency"), Verfügbarkeit ("Availability") und Partitionstoleranz ("Partition Tolerance").

Das Theorem besagt, dass in verteilten Systemen mit replizierten Datenbeständen immer nur zwei der drei Eigenschaften gleichzeitig erfüllt werden können. Diese Eigenschaften sind im Detail wie folgt definiert:

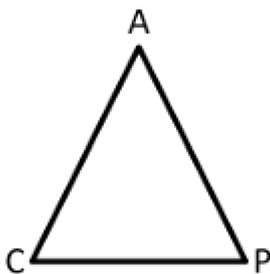


Abb. 3.1: Bildliche Darstellung des CAP-Theorems[1].

Availability (Verfügbarkeit)

Die Verfügbarkeit eines verteilten Datenbanksystems beschreibt die Fähigkeit, jederzeit und von jedem funktionsfähigen Knoten aus Anfragen zu bedienen und sinnvolle Antworten zu liefern, ohne den Benutzer mit Fehlmeldungen zu konfrontieren. Dabei ist es unerheblich, ob es sich um Lese- oder Schreibvorgänge handelt. Ein Knoten gilt dabei als funktionsfähig, wenn er mit anderen Knoten kommunizieren kann. Isolierte Knoten,

[1]Bildquelle: https://de.wikipedia.org/wiki/CAP-Theorem; abgerufen am 02.11.2024

also solche, die keine Verbindung zu anderen Knoten haben, werden als nicht funktionsfähig betrachtet.

Consistency (Konsistenz)

bedeutet, dass alle Repliken eines Datensatzes zu jedem Zeitpunkt den gleichen Wert aufweisen. Änderungen an den Daten werden sofort und synchron an alle Repliken übertragen, sodass alle Clients stets die gleiche, aktuelle Sicht auf die Daten haben. Es ist wichtig zu betonen, dass diese starke Konsistenz sich von der ACID-Konsistenz unterscheidet, die in traditionellen relationalen Datenbanken anzutreffen ist. Während ACID-Konsistenz sich auf die Atomizität, Konsistenz, Isolation und Dauerhaftigkeit von Transaktionen bezieht, fokussiert sich die CAP-Konsistenz auf die Übereinstimmung der Daten zwischen den verschiedenen Repliken eines verteilten Systems. (weiter zum Thema ACID siehe Kapitel 4).

Partition tolerance (Partitionierungstoleranz)

Partitionstoleranz beschreibt die Fähigkeit eines verteilten Systems, auch dann korrekt zu funktionieren, wenn Teile des Systems voneinander getrennt sind. Eine Partition entsteht, wenn technische Probleme wie Netzwerkausfälle dazu führen, dass bestimmte Knoten im System nicht mehr miteinander kommunizieren können. Das System wird dadurch in zwei oder mehr unabhängige Teilbereiche aufgeteilt. Ein partitionstolerantes System kann weiterhin Anfragen bearbeiten und Daten konsistent halten, auch wenn solche Partitionen auftreten.

vgl. [6] (vgl. [3], S. 354)

2.3.2 Drei Klassen von verteilten Datenbanksystemen

Das CAP-Theorem führt demnach zu drei verschiedenen Klassen von Datenbanksystemen.

AC: Verzicht auf Partitionierungstoleranz

CA-Systeme bieten eine hohe Verfügbarkeit und garantieren eine starke Konsistenz. Das bedeutet, dass jeder Lesezugriff den aktuellsten Datenstand liefert und jede Änderung sofort für alle sichtbar wird. Um diese starke Konsistenz zu gewährleisten, wird häufig ein Zwei-Phasen-Commit eingesetzt. Dieser Mechanismus sorgt dafür, dass alle beteiligten Knoten einer Transaktion entweder alle Änderungen übernehmen oder alle Änderungen rückgängig machen.

Allerdings ist ein entscheidender Nachteil von CA-Systemen ihre fehlende Partitionstoleranz. Sollte es zu einer Netzwerkpartitionierung kommen, kann das System nicht mehr garantieren, dass alle Knoten den gleichen Datenstand haben.

CP: Verzicht auf Verfügbarkeit

CP-Systeme (Konsistenz und Partitionstoleranz) stellen sicher, dass alle Knoten eines verteilten Systems immer den gleichen, aktuellen Datenstand haben. Dies wird auch als starke Konsistenz bezeichnet. Um diese Eigenschaft auch bei einer Partitionierung des Systems zu gewährleisten, müssen Maßnahmen ergriffen werden, die die Verfügbarkeit einschränken können. So kann es beispielsweise notwendig sein, eine Minderheitspartition abzuschalten oder zu sperren, um zu verhindern, dass inkonsistente Daten entstehen. Die Folge ist, dass nur noch ein Teil des Systems funktionsfähig ist und Anfragen bearbeitet.

AP: Verzicht auf Konsistenz

AP-Systeme (Availability und Partition Tolerance) zeichnen sich durch eine hohe Verfügbarkeit und Toleranz gegenüber Netzwerkpartitionierungen aus. Selbst wenn ein System in mehrere, nicht miteinander kommunizierende Teile zerfällt, bleibt jeder Teil weiterhin funktionsfähig und erlaubt Änderungen am Datenbestand. Dies führt jedoch zwangsläufig zu einer zeitweisen Inkonsistenz der Daten zwischen den einzelnen Partitionen. Diese Inkonsistenzen entstehen, da eine Synchronisierung der Daten aufgrund der fehlenden Verbindung nicht möglich ist.

(vgl. [3], S.354-359)

2.3.3 Probleme des CAP-Theorems

Das CAP-Theorem bietet eine vereinfachte Sicht auf die komplexen Trade-offs in verteilten Systemen. Es ist ein nützliches Werkzeug, um grundlegende Konzepte zu verstehen, aber es kann die Realität nicht vollständig abbilden. Eric Brewers eigene Aussage, dass er das Theorem gerne präzisieren würde, unterstreicht diese Tatsache. Viele Forscher und Praktiker haben darauf hingewiesen, dass das CAP-Theorem oft zu vereinfacht dargestellt wird und dass die Realität oft nuancierter ist.

Missverständlich

Das CAP-Theorem wird oft kritisiert, insbesondere der Begriff "Konsistenz", führt häufig zu Missverständnissen, da er leicht mit dem C im ACID-Prinzip verwechselt wird. Während das ACID-Prinzip eine starke Konsistenz innerhalb einer Transaktion garantiert, bezieht sich die Konsistenz im CAP-Theorem eher auf die lineare Abfolge von Operationen in einem verteilten System. Ein präziserer Begriff wäre hier

"Linearisierbarkeit". Diese beschreibt die Eigenschaft, dass alle Operationen so erscheinen, als wären sie in einer bestimmten sequenziellen Reihenfolge ausgeführt worden, auch wenn sie in Wirklichkeit parallel oder auf verschiedenen Knoten ausgeführt wurden. vgl. [6]

Zu unspezifisch

Die Formulierung des CAP-Theorems durch Gilbert und Lynch im Jahr 2002 stellte zwar einen wichtigen Schritt zur Präzisierung dar, doch bleiben einige Aspekte ungeklärt. Insbesondere die Definition von "Verfügbarkeit" ist vage. Wann ist ein System tatsächlich verfügbar? Ab welcher Antwortzeit gilt ein System als nicht mehr verfügbar? Das Theorem bietet keine konkreten Grenzwerte. Zudem suggeriert die Formulierung des Theorems, dass jedes verteilte System zwangsläufig zwei der drei Eigenschaften erfüllen muss. In der Praxis gibt es jedoch Systeme, die nur eine oder gar keine dieser Eigenschaften in ausreichendem Maße erfüllen.

Realitätsfern

Das CAP-Theorem bietet einen wertvollen Rahmen zur Analyse verteilter Systeme, stößt jedoch in der Praxis auf seine Grenzen. Die Annahme, dass ein System immer nur zwei der drei Eigenschaften erfüllen kann, ist eine starke Vereinfachung. Moderne Architekturen mit Lastverteilern und Replikation ermöglichen es, Systeme zu bauen, die auch bei einer Partitionierung eine hohe Verfügbarkeit und Konsistenz bieten können. Allerdings sind solche Systeme oft komplex und erfordern eine sorgfältige Planung und Konfiguration. Die Entscheidung, welche Eigenschaften priorisiert werden sollen, hängt von den spezifischen Anforderungen der Anwendung ab. vgl. [7]

2.4 Transaktionskonzepte

In diesem Kapitel befasst sich eingehend mit dem Konzept der Transaktionen. Zunächst wird in Abschnitt 4.1 der Begriff Transaktion definiert und erläutert, welche Aufgaben ein Transaktionskonzept erfüllt. Anschließend werden in Abschnitt 4.2 verschiedene Mechanismen vorgestellt, die die Datenkonsistenz in Mehrbenutzerumgebungen gewährleisten. Die folgenden Abschnitte 4.3 und 4.4 widmen sich den beiden grundlegenden Prinzipien, nach denen Transaktionskonzepte arbeiten: ACID und BASE. Diese Prinzipien stellen jeweils unterschiedliche Anforderungen an die Eigenschaften von Transaktionen und sind für verschiedene Anwendungsbereiche geeignet.

2.4.1 Aufgaben des Transaktionskonzepts

Transaktionen bilden in relationalen Datenbanken die Grundlage für die Gewährleistung der Datenintegrität und -konsistenz. Durch die Aufteilung von Daten auf mehrere Tabellen entstehen komplexe Abhängigkeiten, die durch Transaktionen koordiniert werden müssen. Eine Transaktion ist im Wesentlichen ein logischer Arbeitsabschnitt, der entweder vollständig ausgeführt oder vollständig rückgängig gemacht wird. Dies gewährleistet, dass die Datenbank immer in einem konsistenten Zustand bleibt, selbst wenn mehrere Benutzer gleichzeitig darauf zugreifen. Transaktionskonzepte regeln dabei nicht nur die Reihenfolge der Operationen, sondern auch das Verhalten im Fehlerfall. So wird verhindert, dass beispielsweise nur ein Teil einer Überweisung ausgeführt wird und damit ein inkonsistenter Zustand entsteht.

Das Transaktionskonzept ist nicht auf relationale Datenbanken beschränkt. Auch in NoSQL-Datenbanken, die oft eine andere Datenstruktur und ein anderes Konsistenzmodell haben, spielen Transaktionen eine wichtige Rolle, wenn auch in einer angepassten Form. Während relationale Datenbanken häufig versuchen, das ACID-Prinzip (Atomicity, Consistency, Isolation, Durability) einzuhalten, orientieren sich viele NoSQL-Datenbanken am BASE-Prinzip (Basically Available, Soft state, Eventual consistency). Dieses bietet eine höhere Verfügbarkeit und Skalierbarkeit, geht jedoch zu Lasten der starken Konsistenz. ACID und BASE sind keine Gegensätze, sondern setzen vielmehr unterschiedliche Schwerpunkte. Die Wahl des geeigneten Paradigmas hängt von den spezifischen Anforderungen der Anwendung ab.

2.4.2 Maßnahmen zur Konsistenzsicherung

2.4.2.1 Sperren

Um die ACID-Eigenschaften in einem Mehrbenutzerumfeld zu gewährleisten, sind Sperrverfahren unerlässlich. Diese Mechanismen dienen dazu, Konflikte bei gleichzeitigen Zugriffen auf Daten zu vermeiden und die Datenintegrität zu schützen. Durch das Setzen von Sperren wird sichergestellt, dass nur eine Transaktion zu einem bestimmten Zeitpunkt auf ein Datenobjekt schreiben kann. Dies verhindert, dass mehrere Transaktionen gleichzeitig das gleiche Datenobjekt ändern und dadurch inkonsistente Zustände entstehen.

Es gibt zwei grundlegende Ansätze für die Sperrverwaltung: pessimistische und optimistische Verfahren. Beide Ansätze haben ihre Vor- und Nachteile und die Wahl des geeigneten Verfahrens hängt von den spezifischen Anforderungen der Anwendung ab.

Sperren können in Lese- und Schreibsperren unterteilt werden. Eine Lesesperre erlaubt mehreren Transaktionen den gleichzeitigen Zugriff auf ein Datenobjekt zum Lesen. Eine Schreibsperre hingegen erlaubt nur einer Transaktion den exklusiven Zugriff auf ein Datenobjekt, um Änderungen vorzunehmen. Durch diese Unterscheidung wird sichergestellt, dass die Datenkonsistenz erhalten bleibt und dass keine Transaktion durch andere Transaktionen in ihrer Arbeit gestört wird. (vgl. [4], S. 369)

Pessimistische Sperrverfahren

Pessimistische Sperrverfahren stellen eine proaktive Methode zur Konfliktvermeidung in Datenbanktransaktionen dar. Indem Objekte bereits zu Beginn einer Transaktion gesperrt werden, wird verhindert, dass andere Transaktionen gleichzeitig auf dieselben Daten zugreifen und diese verändern. Diese Sperren bleiben in der Regel für die gesamte Dauer der Transaktion bestehen und werden erst nach erfolgreichem Abschluss freigegeben.

Diese Vorgehensweise hat jedoch auch Nachteile: Da Objekte über lange Zeiträume gesperrt werden können, kann es zu einer erhöhten Blockierungsrate kommen, d.h. andere Transaktionen müssen möglicherweise warten, bis eine Sperre aufgehoben wird. Dies kann die Gesamtleistung des Systems beeinträchtigen, insbesondere wenn häufig auf dieselben Datenobjekte zugegriffen wird. (vgl. [4], S. 369)

Optimistische Sperrverfahren

Optimistische Sperrverfahren gehen von der Annahme aus, dass Konflikte zwischen Transaktionen eher selten auftreten. Anstatt Datenobjekte proaktiv zu sperren, wird erst am Ende einer Transaktion überprüft, ob sich die Daten währenddessen geändert haben. Dazu wird jedem Datensatz ein Zeitstempel zugeordnet. Dieser Zeitstempel wird zu Beginn der Transaktion ausgelesen und am Ende erneut geprüft. Stimmen die Zeitstempel überein, bedeutet dies, dass keine andere Transaktion die Daten in der Zwischenzeit geändert hat, und die Transaktion kann erfolgreich abgeschlossen werden. Andernfalls wird die Transaktion abgebrochen und muss wiederholt werden.

Dieser Ansatz bietet den Vorteil, dass Sperren nur kurzzeitig gehalten werden und somit die Gefahr von Deadlocks reduziert wird. Zudem können mehr Transaktionen parallel ausgeführt werden, da sie nicht auf das Freigeben von Sperren durch andere Transaktionen warten müssen. (vgl. [4], S. 384)

2.4.2.2 Zwei-Phasen-Commit

Das Zwei-Phasen-Commit-Protokoll ist ein zentralisierter Algorithmus, der in verteilten Datenbank-Systemen eingesetzt wird, um die atomare Ausführung von Transaktionen über mehrere Knoten hinweg zu gewährleisten. Ein zentraler Koordinator steuert den Commit-Prozess. Zunächst leitet der Koordinator eine Anfrage an alle beteiligten Teilnehmer, um zu erfragen, ob die Transaktion lokal committed werden kann. Erhalten alle Teilnehmer eine positive Rückmeldung, sendet der Koordinator ein Commit-Signal an alle Teilnehmer. Erst dann wird die Transaktion endgültig festgeschrieben. Sollte ein Teilnehmer einen Fehler melden, wird ein Abort-Signal gesendet und die Transaktion wird rückgängig gemacht.

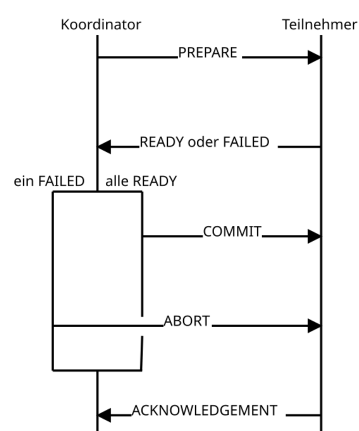


Abb. 4.1: zwei-Phasen-Commit[1]

Als Erweiterung zum Zwei-Phasen-Commit gibt es das Drei-Phasen-Commit-Protokoll. Durch zusätzliche Nachrichten zwischen den Teilnehmern und dem Koordinator wird sichergestellt, dass der Synchronisationsprozess auch dann fortgesetzt werden kann, wenn während des Ablaufs ein Teilnehmer oder der Koordinator ausfällt.
vgl. [8] (vgl. [3], S. 230)

[1]Bildquelle: https://commons.wikimedia.org/wiki/File:Zwei_phasen_commit.svg; abgerufen am 20.12.2024.

2.4.3 ACID

ACID ist ein Akronym, das die vier grundlegenden Eigenschaften einer Datenbanktransaktion beschreibt, die für die Datenintegrität unerlässlich sind:

A: Atomicity (Atomarität)

Die Atomarität einer Transaktion gewährleistet, dass diese als unteilbare Einheit betrachtet wird. Entweder werden alle Operationen einer Transaktion erfolgreich abgeschlossen und die Änderungen an der Datenbank werden dauerhaft gespeichert oder die gesamte Transaktion wird verworfen und die Datenbank verbleibt in ihrem ursprünglichen Zustand. Es gibt keine Zwischenzustände. Dieses "Alles-oder-Nichts"-Prinzip schützt die Datenbank vor Inkonsistenzen, die entstehen könnten, wenn nur ein Teil der Operationen einer Transaktion ausgeführt würde.

C: Consistency (Konsistenz)

Die Konsistenzeigenschaft einer Transaktion stellt sicher, dass diese die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt. Dabei werden alle definierten Integritätsbedingungen der Datenbank eingehalten. Dies bedeutet, dass am Ende einer erfolgreichen Transaktion die Daten in einem Zustand sind, der alle Regeln und Einschränkungen erfüllt, die für die Datenbank gelten.

I: Isolation

Die Isolationseigenschaft stellt sicher, dass jede Transaktion so ausgeführt wird, als wäre sie die einzige Transaktion, die gerade auf die Datenbank zugreift. Dies bedeutet, dass eine Transaktion nicht durch andere gleichzeitig laufende Transaktionen beeinflusst wird. Die Änderungen, die eine Transaktion an der Datenbank vornimmt, sind für andere Transaktionen erst sichtbar, wenn die betreffende Transaktion erfolgreich abgeschlossen wurde. Dadurch wird verhindert, dass sich Transaktionen gegenseitig "sehen" und inkonsistente Daten entstehen.

Durability (Dauerhaftigkeit)

Die Dauerhaftigkeit einer Transaktion gewährleistet, dass einmal erfolgreich abgeschlossene Änderungen an der Datenbank dauerhaft gespeichert werden. Das bedeutet, dass diese Änderungen auch bei Systemausfällen, Hardwareproblemen oder Softwarefehlern nicht verloren gehen. Sobald eine Transaktion erfolgreich committet wurde, sind die Änderungen fest im System verankert und können nicht mehr rückgängig gemacht werden.

vgl. [10] (vgl. [9], S136)

2.4.4 BASE

Das BASE-Modell wurde als Reaktion auf die Einschränkungen des ACID-Modells in verteilten Systemen entwickelt. Während ACID hohe Konsistenz garantiert, kann der Einsatz von Sperrmechanismen in großen Clustern zu Performance-Einbußen und reduzierter Verfügbarkeit führen. BASE priorisiert hingegen die Verfügbarkeit und ermöglicht so eine höhere Skalierbarkeit. Die einzelnen Buchstaben in BASE stehen dafür:

BA: Basically available (Grundsätzliche Verfügbarkeit)

Das System ist grundsätzlich immer erreichbar und antwortet auf Anfragen. Auch wenn Teile des Systems ausgefallen sind oder Netzwerkprobleme auftreten, kann das System weiterhin Anfragen bearbeiten. Allerdings können die zurückgelieferten Daten unter Umständen veraltet oder inkonsistent sein.

S: Soft state (Weicher Zustand)

beschreibt die Eigenschaft von Daten, sich im Laufe der Zeit ohne explizite Eingriffe zu ändern. Da Änderungen in einem verteilten System zeitlich verzögert repliziert werden, kann der Zustand eines Datensatzes in verschiedenen Repliken kurzzeitig unterschiedlich sein. Erst nach einer gewissen Zeit konvergieren alle Repliken zu einem einheitlichen Zustand.

E: Eventual Consistency (Letztendliche Konsistenz)

Eventual Consistency bedeutet, dass Änderungen an einem Datensatz zwar garantiert irgendwann in allen Repliken eines verteilten Systems sichtbar werden, jedoch nicht sofort. Es kann eine gewisse Zeit vergehen, bis alle Kopien eines Datensatzes konsistent sind. Dies führt dazu, dass während dieser Übergangsphase inkonsistente Zustände auftreten können, beispielsweise wenn ein Nutzer eine ältere Version eines Datensatzes sieht, während ein anderer Nutzer bereits eine neuere Version sieht.

vgl. [10] (vgl. [3], S.353)

2.4.5 ACID vs. BASE

Eigenschaften	ACID	BASE
Konsistenz	Starke Konsistenz	Schwache Konsistenz
Fokus	Isolation	Verfügbarkeit
Synchronisation	Pessimistische Synchronisation	Optimistische Synchronisation
Commit	Globales Commit	Lokales Commit

5 Datenbanksysteme im Detail

In den folgenden Kapiteln werden sich eingehend mit drei prominenten Datenbankmanagementsystemen auseinandersetzen: MariaDB, MongoDB und Cassandra. Diese Systeme unterscheiden sich grundlegend in ihrer Architektur, ihren Einsatzszenarien und ihren Konzepten zur Datenverwaltung.

Es wird untersucht, wie diese Datenbanken Transaktionen handhaben, welche Mechanismen zur Clusterbildung sie bieten und wie sie Daten auf mehrere Knoten verteilen. Dabei wird auch auf die Rolle von Sperrmechanismen eingehen und analysieren, inwieweit die ACID-Kriterien erfüllt werden. Ein weiterer wichtiger Aspekt ist die Skalierbarkeit: Es wird untersucht, wie die Systeme während des Betriebs erweitert werden können und welche Auswirkungen dies auf die Performance hat.

Besondere Aufmerksamkeit gilt der Ausfalltoleranz, insbesondere im Falle von Netzwerkpartitionierungen. Es wird gezeigt, wie die einzelnen Systeme mit solchen Situationen umgehen und welche Maßnahmen ergriffen werden, um die Datenkonsistenz zu gewährleisten.

Die in diesen Kapiteln präsentierten Erkenntnisse basieren nicht nur auf den offiziellen Dokumentationen der jeweiligen Systeme, sondern auch auf eigenen Beobachtungen und Erfahrungen, die im Rahmen des Studiums gesammelt wurden.

5.1 MariaDB

MariaDB ist ein leistungsstarkes, offenes und relationales Datenbankmanagementsystem, das als Fork von MySQL entstanden ist. Es wurde entwickelt, um die Zukunft von MySQL als offene Plattform zu sichern und bietet eine Vielzahl von innovativen Funktionen und Verbesserungen. Dank seiner hohen Kompatibilität zu MySQL kann MariaDB oft nahtlos als Ersatz eingesetzt werden, ohne dass umfangreiche Änderungen an bestehenden Anwendungen vorgenommen werden müssen.

Die Entwicklung von MariaDB begann im Jahr 2009, als Oracle MySQL übernahm. Um die Unabhängigkeit und die offene Entwicklung von MySQL zu gewährleisten, gründete ein Team erfahrener Entwickler unter der Leitung von Michael "Monty" Widenius das MariaDB-Projekt. Ziel war es, eine zukunftsorientierte Datenbank zu schaffen, die sowohl die Stärken von MySQL übernimmt als auch neue Innovationen ermöglicht.

Die enge Beziehung zu MySQL sorgt dafür, dass viele Anwendungen, die für MySQL entwickelt wurden, ohne größere Anpassungen auch mit MariaDB funktionieren.
vgl. [11]

In diesem Teil der Arbeit werden wir uns eingehend mit den Transaktionskonzepten in MariaDB beschäftigen, insbesondere mit dem Aspekt der Isolation (Kap. 6.1). Anschließend untersuchen wir die Master-Slave-Replikation und analysieren, inwieweit MariaDB damit die ACID-Eigenschaften erfüllt (Kap. 6.2.1). Abschließend stellen wir das Galeracluster als eine weitere Möglichkeit zur Erhöhung der Verfügbarkeit und Skalierbarkeit vor (Kapitel 6.3).

5.1.1 Transaktionskonzept

MariaDB bietet auf Einzelsystemen eine umfassende Unterstützung der ACID-Eigenschaften. Die Atomarität gewährleistet, dass Transaktionen entweder vollständig ausgeführt oder komplett rückgängig gemacht werden, wodurch die Datenintegrität geschützt ist. Die Konsistenz wird durch strenge Integritätssprüfungen sichergestellt, die verhindern, dass inkonsistente Daten in die Datenbank gelangen. Die Dauerhaftigkeit wird durch den optional aktivierbaren Binary-Log gewährleistet, der eine vollständige Historie aller Änderungen speichert und eine Wiederherstellung bei

Bildquelle: MariaDB Foundation, siehe https://mariadb.org/about/logos/; abgerufen am 18.11.2024.

Datenverlust ermöglicht. Die Isolation wird durch verschiedene Isolationsebenen reguliert:

READ UNCOMMITTED

Das Isolationsebene READ UNCOMMITTED ist die schwächste von allen. In diesem Modus werden keinerlei Sperren gesetzt, sodass alle Änderungen an Daten sofort für andere Transaktionen sichtbar werden, selbst wenn diese Änderungen noch nicht festgeschrieben (committed) wurden. Dies birgt die Gefahr von sogenannten 'dirty reads': Wenn eine Transaktion Daten liest, die gerade von einer anderen Transaktion geändert, aber noch nicht committed wurden, und diese Änderung dann rückgängig gemacht wird, hat die lesende Transaktion inkonsistente Daten erhalten.

READ COMMITTED

Die Isolationsebene READ COMMITTED bietet eine höhere Konsistenz als READ UNCOMMITTED, indem sie sicherstellt, dass nur committed Daten gelesen werden. Damit werden sogenannte 'dirty reads' verhindert, bei denen nicht gespeicherte Änderungen gelesen werden. Allerdings können bei READ COMMITTED sogenannte 'phantom reads' auftreten. Ein phantom read entsteht, wenn eine Transaktion während ihrer Ausführung neue Datensätze sieht, die von einer anderen Transaktion hinzugefügt wurden. Dies kann zu inkonsistenten Ergebnissen führen.

REPEATABLE READ

Die Isolationsebene REPEATABLE READ ist der Standard in MariaDB mit InnoDB und bietet eine hohe Konsistenz. Zu Beginn einer Transaktion wird ein konsistenter Snapshot der Datenbank erstellt. Alle Leseoperationen innerhalb dieser Transaktion greifen auf diesen Snapshot zu. Das bedeutet, dass wiederholte Lesezugriffe innerhalb derselben Transaktion immer dieselben Ergebnisse liefern, unabhängig davon, welche Änderungen andere Transaktionen währenddessen an der Datenbank vornehmen. Dies eliminiert das Problem der 'phantom reads', das bei READ COMMITTED auftreten kann.

SERIALIZABLE

Die Isolationsebene SERIALIZABLE bietet die höchste Konsistenz aller Isolationsebenen in MariaDB. Durch das Setzen von Sperren auf alle betroffenen Zeilen wird sichergestellt, dass Transaktionen so ausgeführt werden, als würden sie nacheinander (seriell) ausgeführt werden. Dies eliminiert jegliche Arten von Inkonsistenzen wie dirty reads, non-repeatable reads und phantom reads. SERIALIZABLE ist die einzige Isolationsebene, die die ACID-Eigenschaften vollständig erfüllt und somit die höchste Datenintegrität garantiert.

vgl. [12]; [13]

5.1.2 Clusterbildung

MariaDB unterstützt zwei Arten der Clusterbildung: Den Aufbau eines typischen Master Slave Replikationssets, sowie die Erstellung eines Galera Clusters.

5.1.2.1 MariaDB Replikation

MariaDB nutzt zur Replikation ein asynchrones Master-Slave Replikationsverfahren. Der Master knoten speichert alle Manipulationen an der Datenbank in einem Protokoll ab, dem sogenannten „Binary-Log“. Die Slaveknoten sollten im „read-only“ Modus betrieben werden, da sonst im Falle eines Konfliktes das Replikationsset aufgelöst wird.

Binary-Log

Der MariaDB Binary-Log ist ein detailliertes Protokoll, das jede Änderung an einer Datenbank aufzeichnet. Egal ob Sie Daten einfügen (INSERT), aktualisieren (UPDATE), löschen (DELETE) oder die Struktur einer Tabelle ändern (CREATE, ALTER), jede Aktion wird im Binary-Log festgehalten. Dabei werden nicht nur die durchgeführten Änderungen selbst, sondern auch die Reihenfolge der Änderungen aufgezeichnet. Der Binary-Log besteht aus einer Reihe von binären Protokolldateien sowie einem zugehörigen Index, der den schnellen Zugriff auf die aufgezeichneten Informationen ermöglicht. vgl. [14]

Global Transaction ID

Die GTID ist eine eindeutige Kennung, die jeder Transaktion im MariaDB Binary-Log zugewiesen wird. Diese globale Identifikation gewährleistet, dass eine Transaktion in einem gesamten Replikationsset eindeutig identifizierbar ist. Durch die Speicherung der GTID des zuletzt verarbeiteten Log-Eintrags kann ein Slave-Knoten seinen Replikationsstatus präzise bestimmen. Selbst nach einem Neustart oder einer längeren Offline-Phase kann die Replikation genau an der Stelle fortgesetzt werden, an der sie unterbrochen wurde. vgl. [15]

Einstellungsseitig kann gewählt werden, ob sie eine gesamte Datenbankinstanz, einzelne Datenbanken oder nur bestimmte Tabellen replizieren möchten. Diese Granularität ermöglicht eine maßgeschneiderte Anpassung der Replikation an die spezifischen Anforderungen der Anwendung. Da die Replikation asynchron erfolgt, arbeitet der Master unabhängig von den Slaves weiter. Dies bietet eine hohe Performance, kann jedoch zu einer gewissen Latenz zwischen Master und Slaves führen. Um eine höhere Synchronität zu erreichen, kann das semi-synchrone Replikationsverfahren eingesetzt werden. vgl. [16]

Semi-synchrones Replikationsverfahren

Die semi-synchrone Replikation in MariaDB bietet einen Mittelweg zwischen der asynchronen und vollständig synchronen Replikation. Sie gewährleistet, dass eine Transaktion auf dem Master erst als abgeschlossen gilt, wenn sie erfolgreich auf einer bestimmten Anzahl von Slaves repliziert wurde. Dadurch wird eine höhere Datenkonsistenz zwischen Master und Slaves erreicht und das Risiko von Datenverlusten im Falle eines Ausfalls des Masters verringert. Allerdings führt diese zusätzliche Sicherheit zu einer erhöhten Latenz, da der Master auf die Bestätigung der Slaves warten muss, bevor er die Transaktion committet. (vgl. [17], Folie 19)

Die MariaDB-Replikation findet insbesondere in Umgebungen mit einem hohen Anteil an Lesevorgängen und wenigen Schreibvorgängen breite Anwendung. Durch die Verteilung der Leseanfragen auf mehrere Slave-Knoten kann die Gesamtleistung des Systems erheblich gesteigert werden. Der Master-Knoten übernimmt dabei die Verarbeitung aller Schreibvorgänge, während die Slaves als Read-Replicas dienen. Diese Aufteilung ermöglicht eine effiziente Nutzung der Hardware und reduziert die Belastung des Master-Knotens. Allerdings ist zu beachten, dass die Daten auf den Slaves aufgrund der asynchronen Replikation leicht verzögert sein können. Für Anwendungen, die eine starke Konsistenz erfordern, ist das Galera Cluster eine geeignetere Wahl.

Erfüllung des ACID-Konzepts

Die asynchrone Master-Slave-Replikation in MariaDB wirft interessante Fragen hinsichtlich der Einhaltung der ACID-Eigenschaften auf. Auf dem Master-Knoten, der als zentrale Datenquelle fungiert, können die ACID-Eigenschaften in der Regel vollständig gewährleistet werden. Die Isolation von Transaktionen, die Atomarität und die Dauerhaftigkeit sind durch die Datenbank-Engine sichergestellt. Allerdings führt die asynchrone Natur der Replikation zu einer gewissen Inkonsistenz zwischen Master und Slaves.

Die Slaveknoten replizieren die Änderungen des Masters mit einer gewissen Verzögerung. Während der Master bereits eine Transaktion als abgeschlossen betrachtet, kann es sein, dass diese auf den Slaves noch nicht vollständig verarbeitet wurde. Diese sogenannte 'Slave-Lag' führt dazu, dass das Gesamtsystem nur eine eventuelle Konsistenz aufweist, wie sie im BASE-Modell definiert ist.

Obwohl die asynchrone Replikation die strikte Konsistenz des ACID-Modells auf der Ebene des gesamten Systems nicht garantiert, bedeutet dies nicht, dass die ACID-Eigenschaften auf dem Master verletzt werden.

5.1.2.2 Skalierbarkeit, Verfügbarkeit und Performance

Skalierbarkeit

Die Master-Slave-Replikation in MariaDB bietet eine hervorragende Skalierbarkeit für Leseanfragen. Durch das Hinzufügen weiterer Slave-Knoten kann die Leselast effektiv verteilt werden, was zu einer signifikanten Leistungssteigerung führen kann. Theoretisch können unbegrenzt viele Slave-Knoten in ein Replikationsset integriert werden. Allerdings ist diese Skalierbarkeit auf Lesevorgänge beschränkt, da alle Schreibvorgänge über den Masterknoten abgewickelt werden müssen.

Um einen neuen Slave in das Replikationsset aufzunehmen, sind mehrere Schritte erforderlich:

1. **Sperren des Masters:** Der Masterknoten muss gesperrt werden, um eine konsistente Sicherung zu erstellen. Dies führt zu einer vorübergehenden Einschränkung der Verfügbarkeit.
2. **Wiederherstellung der Sicherung:** Die Sicherung wird auf dem neuen Slave wiederhergestellt. Es ist nicht zwingend erforderlich, die aktuellste Sicherung zu verwenden. Ältere Sicherungen können durch das Einspielen der fehlenden Transaktionen auf den neuesten Stand gebracht werden.
3. **Konfiguration des Slaves:** Der neue Slave muss so konfiguriert werden, dass er sich mit dem Master verbindet und die Replikation beginnt.

Dieser Prozess ist relativ aufwendig und kann die Verfügbarkeit des Systems beeinträchtigen. vgl. [18]

Performance

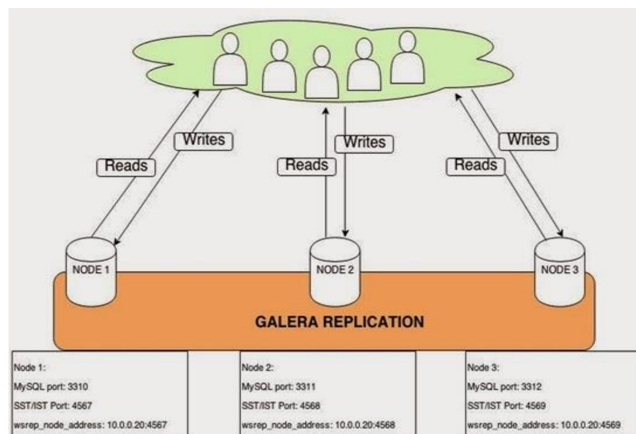
Das Master-Slave-Replikationscluster ist ideal für Anwendungen, die durch Leseanfragen dominiert werden. Die Verteilung der Lesevorgänge auf mehrere Slave-Knoten entlastet den Master signifikant und steigert die Gesamtleistung. Allerdings kann die Performance bei einer hohen Schreiblast stark beeinträchtigt werden. Da alle Schreibvorgänge zentral auf dem Master verarbeitet werden, kann es bei einem hohen Schreibaufkommen zu einem Engpass kommen. Dieser kann zu einer erhöhten Latenz und einer verminderten Verfügbarkeit führen. vgl. [18]

Verfügbarkeit

Fällt ein Slave-Knoten in einem Master-Slave-Setup aus, hat dies in der Regel keine direkten Auswirkungen auf die Verfügbarkeit des Systems für die meisten Clients. Der Lastverteiler kann die Anfragen einfach auf die verbleibenden aktiven Slaves umleiten. Nur Clients, die explizit auf den ausgefallenen Slave zugreifen, werden einen Fehler bemerken. Sollte der Masterknoten ausfallen, wird einer der Slaveknoten automatisch zum neuen Masterknoten gewählt. Dieser Vorgang ist in der Regel transparent für die Anwendung, da die Clientanwendungen das gesamte Replikationsset als Einheit betrachten.

5.1.3 Galera-Cluster

Das Galera Cluster stellt eine leistungsstarke Alternative zur traditionellen Master-Slave-Replikation in MariaDB dar. Im Gegensatz zur asynchronen Replikation des Master-Slave-Setups zeichnet sich Galera durch eine **synchronisierte Replikation** aus, die eine hohe Konsistenz und Verfügbarkeit gewährleistet.

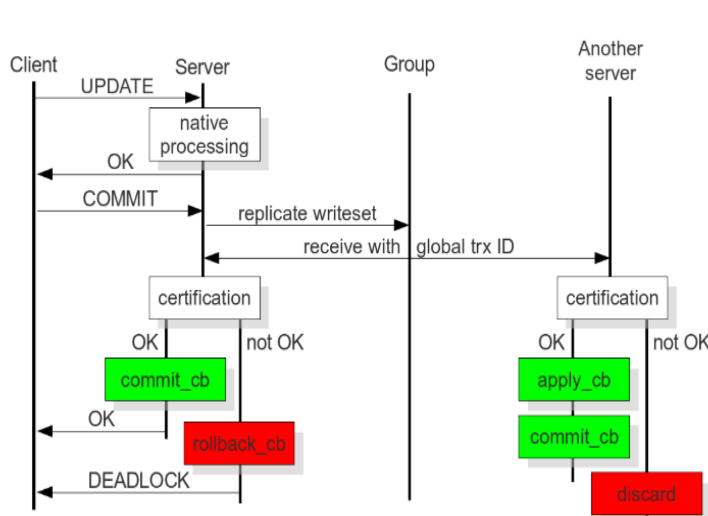


Während im Master-Slave-Modell alle Schreibvorgänge zentral auf einem Masterknoten gebündelt werden, ermöglicht das Galera Cluster einen **Multi-Master-Betrieb**. Jeder Knoten im Cluster fungiert als vollwertiger Master und kann sowohl Lese- als auch Schreibvorgänge ausführen. Diese Architektur eliminiert den Single Point of Failure des Masters und bietet eine höhere Skalierbarkeit für Schreiblasten.

Ein weiterer wesentlicher Unterschied besteht in der **Replikationstechnologie**. Während die native MariaDB-Replikation auf dem Binary-Log basiert, verwendet Galera ein **certification-basiertes Replikationsprotokoll**. vgl. [19]; [20]

5.1.3.1 Certification-based replication

Das Certification-based Replication Verfahren ist ein mehrstufiger Prozess, der sicherstellt, dass alle Änderungen an den Daten in allen Knoten eines Galera Clusters synchronisiert werden, bevor sie als endgültig betrachtet werden.



So funktioniert es:

1- **Transaktionsanfrage:** Ein Client sendet eine Transaktionsanfrage an einen beliebigen Knoten des Clusters.

2- **Lokale Ausführung:** Der Knoten führt die Transaktion lokal aus und prüft, ob sie die Integritätsbedingungen erfüllt.

3- **Zertifizierung:** Der Knoten leitet die Transaktion an alle anderen Knoten weiter und fordert eine Bestätigung (Zertifikat) an.

[1]Bildquelle: siehe https://mysqlnlinux.blogspot.com/2014/09/configuring-multiple-galera-cluster.html; abgerufen am 24.11.2024.

4- **Commit oder Rollback:**

- **Erfolgreiche Zertifizierung:** Wenn alle Knoten die Transaktion bestätigen, wird sie als committed markiert.
- **Fehlgeschlagene Zertifizierung:** Wenn mindestens ein Knoten die Transaktion ablehnt (z.B. aufgrund eines Konflikts), wird die Transaktion auf allen Knoten zurückgerollt.

5- **Rückmeldung an den Client:** Erst nachdem alle Knoten die Transaktion bestätigt oder abgelehnt haben, erhält der Client eine entsprechende Rückmeldung.

Dieser Mechanismus gewährleistet eine hohe Datenkonsistenz, da alle Änderungen nur dann dauerhaft gespeichert werden, wenn alle Knoten zustimmen. Allerdings führt dieser zusätzliche Schritt zu einer geringeren Latenz bei der Transaktionsverarbeitung im Vergleich zu einem Master-Slave-System, bei dem der Erfolg einer Transaktion direkt nach der Ausführung auf dem Master gemeldet wird. vgl. [21]

Erfüllung des ACID-Konzepts

Das Galera Cluster ist in der Lage, die ACID-Eigenschaften in hohem Maße zu gewährleisten. Die synchrone Replikation zwischen den Knoten sorgt dafür, dass alle Änderungen an den Daten konsistent und dauerhaft sind. Die Atomarität wird durch das Zwei-Phasen-Commit-Protokoll sichergestellt, das garantiert, dass eine Transaktion entweder vollständig ausgeführt oder vollständig rückgängig gemacht wird.

Die Isolation ist jedoch eingeschränkt. Während das Galera Cluster das Isolationslevel REPEATABLE-READ bietet, ist ein höheres Isolationslevel wie SERIALIZABLE in der Regel nicht realisierbar. Dies liegt daran, dass die parallele Ausführung von Transaktionen auf verschiedenen Knoten zu Phänomenen wie Phantoms führen kann. Ein strikt serielles Ausführungsmodell würde die Vorteile eines Multi-Master-Clusters, wie die parallele Verarbeitung von Transaktionen, zunichtemachen.

Insgesamt bietet das Galera Cluster ein hervorragendes Gleichgewicht zwischen Konsistenz, Verfügbarkeit und Leistung. Die Einschränkungen bei der Isolation sind in vielen Anwendungsfällen akzeptabel, insbesondere wenn die Priorität auf hoher Verfügbarkeit und Skalierbarkeit liegt. vgl. [22]

5.1.3.2 Skalierbarkeit, Verfügbarkeit und Performance

Skalierbarkeit

Die Skalierbarkeit des Galera Clusters wird durch die effizienten State Transfer-Mechanismen gewährleistet. Diese ermöglichen es, neue Knoten schnell und unkompliziert in das Cluster zu integrieren oder bestehende Knoten wieder anzuschließen.

State Snapshot Transfer (SST): Dabei wird eine vollständige Kopie des Datenbestands von einem bestehenden Knoten auf den neuen übertragen. Anschließend wird der neue Knoten dem Cluster hinzugefügt. Galera bietet verschiedene Tools wie mysqldump, rsync oder XtraBackup zur Durchführung des SST an.

Incremental State Transfer (IST): Ist ein Knoten nur vorübergehend ausgefallen, kann er durch einen IST schnell wieder in den Cluster integriert werden. Dabei werden nur die Transaktionen übertragen, die seit dem letzten gemeinsamen Zustand ausgeführt wurden. Dadurch wird die Synchronisierung deutlich beschleunigt.

Durch die Kombination von SST und IST kann ein Galera Cluster flexibel an wechselnde Anforderungen angepasst werden. Neue Knoten können schnell hinzugefügt werden, um die Kapazität zu erhöhen, und ausgefallene Knoten können ohne großen Aufwand wieder in den Cluster integriert werden. vgl. [24]

Verfügbarkeit

Der Ausfall eines einzelnen Knotens hat in der Regel keine gravierenden Auswirkungen auf den gesamten Cluster. Clients können einfach auf einen anderen Knoten ausweichen, sofern ein geeigneter Lastverteiler eingesetzt wird.

Eine größere Herausforderung stellen Netzwerkpartitionierungen dar. Wird das Cluster in mehrere Teilcluster aufgeteilt, kann es zu einer sogenannten 'split-brain'-Situation kommen. In diesem Fall gibt es keine eindeutige Mehrheit, die darüber entscheidet, welcher Teilcluster die korrekten Daten enthält. Um dieses Szenario zu vermeiden, empfiehlt es sich, Cluster mit einer ungeraden Anzahl von Knoten zu betreiben. So ist immer gewährleistet, dass es eine Mehrheitspartition gibt, die den Dienst fortsetzen kann. beispielsweise ein Cluster aus vier Knoten in zwei Teile mit je zwei Knoten verteilt werden, ist das gesamte System nicht mehr verfügbar, da keine Mehrheitspartition existiert.

Minderheitspartitionen stellen in der Regel den Dienst ein und warten auf eine Wiederherstellung der Verbindung zum Hauptcluster. Sobald die Netzwerkpartition behoben ist, synchronisieren sich die Knoten der Minderheitspartition mit der Mehrheitspartition.

Die Synchronisation erfolgt in der Regel in zwei Schritten: Zunächst wird ein vollständiger Snapshot (SST) der Daten eines Knotens erstellt und an die anderen Knoten übertragen. Anschließend wird der Inkremental-Sync (IST) durchgeführt, um die Daten seit der Erstellung des Snapshots zu synchronisieren. vgl. [23]

Performance

Aufgrund der zertifizierungsbasierten Replikation im Galera Cluster, die für eine hohe Datenkonsistenz sorgt, hat das Auswirkungen auf die Performance. Das Warten auf die Bestätigung aller Knoten, bevor eine Transaktion als committed betrachtet wird, kann insbesondere bei großen Clustern zu einer erhöhten Latenz führen. Auch bei der Skalierung eines Galera Clusters kann die Latenz weiter erhöhen, da mit jedem zusätzlichen Knoten die Anzahl der zu kontaktierenden Instanzen wächst.

5.2 MongoDB

MongoDB ist ein Datenbanksystem, das entwickelt wurde, um große Datenmengen flexibel und effizient zu verwalten. Der Name "MongoDB" leitet sich vom englischen Wort "humongous" ab, was „gigantisch" oder „riesig" bedeutet, und spielt darauf an, dass MongoDB für die Verwaltung sehr großer Datenmengen ausgelegt ist. MongoDB unterscheidet sich von klassischen relationalen Datenbanken, bei denen Daten in Tabellen mit festen Strukturen gespeichert werden. Stattdessen speichert MongoDB Daten in sogenannten „Collections" (Sammlungen), die unstrukturiert sind, was bedeutet, dass die Struktur der Daten sehr flexibel ist und sich leicht anpassen lässt.

Zuerst wird das Transaktionskonzept von MongoDB erläutert (siehe Abschnitt 7.1). Anschließend werden in den Kapiteln 7.2 und 7.2.1 die verschiedenen Clustermöglichkeiten beschrieben. vgl. [25]

5.2.1 Transaktionskonzept

MongoDB bietet keine Transaktionen im klassischen Sinn, wie sie in relationalen Datenbanken bekannt sind. An dieser Stelle könnte das Kapitel daher eigentlich enden. Trotzdem nutzt MongoDB ein Konzept, das dem ACID-Prinzip ähnelt, um den Mehrbenutzerbetrieb zu unterstützen.

Atomicity (Atomarität)

MongoDB gewährleistet Atomarität auf Dokumentenebene. Das bedeutet, dass alle Änderungen innerhalb eines Dokuments, einschließlich seiner eingebetteten Dokumente, als eine Einheit behandelt werden. Wird ein Update an einem Dokument durchgeführt, so wird entweder das gesamte Dokument aktualisiert oder gar nicht. Dies garantiert, dass die Datenkonsistenz innerhalb eines Dokuments immer erhalten bleibt.

Allerdings ist es wichtig zu beachten, dass diese Atomarität auf die Ebene des einzelnen Dokuments beschränkt ist. Werden mehrere Dokumente in einer Transaktion aktualisiert, so wird jede Änderung einzeln ausgeführt. Dies bedeutet, dass andere Clients während dieser Operationen auf die betroffenen Dokumente zugreifen und diese möglicherweise ändern können. Es besteht somit die Möglichkeit, dass ein Client eine inkonsistente Ansicht der Daten sieht, wenn er während einer solchen Mehrdokumenten-Operation liest. vgl. [26]

[1]Bildquelle: MongoDB Inc., https://www.mongodb.com/brand-resources; abgerufen am 01.12.2024.

ACID-Consistency (ACID-Konsistenz)

MongoDB bietet im Vergleich zu relationalen Datenbanken eine eingeschränkte ACID-Konsistenz. Während relationale Datenbanken in der Regel über umfangreiche Mechanismen zur Gewährleistung der Datenintegrität verfügen, wie z.B. Constraints, verlässt sich MongoDB stark auf die Implementierung solcher Prüfungen durch die Anwendung.

Die einzige integrierte Möglichkeit, die MongoDB bietet, um die Datenkonsistenz zu erhöhen, ist das Erstellen von Unique-Indizes[1]. Diese Indizes gewährleisten, dass ein bestimmter Feldwert innerhalb einer Collection nur einmal vorkommt. Dadurch können Duplikate verhindert werden, was eine grundlegende Form der Datenintegrität sicherstellt. Allerdings sind Unique-Indizes nicht ausreichend, um alle Arten von Integritätsprüfungen durchzuführen, die in relationalen Datenbanken durch Constraints abgedeckt werden. vgl. [26]; [27]

Isolation

Es existiert ein „Isolated"-Operator. Wird dieser benutzt, sind Änderungen der Daten an mehreren Dokumenten erst mit dem Abschluss der Gesamtoperation für andere Clienten sichtbar. Es findet jedoch im Fehlerfall kein Rollback statt. Der Isolated-Operator sperrt die gesamte Collection, in der Dokumente geändert werden für den Gesamtvorgang. Daher verlangsamt er das System deutlich. Der Isolated-Operator kann nicht bei fragmentierten Clustern verwendet werden und ist damit eigentlich nicht wirklich brauchbar. vgl. [28]

Durability (Dauerhaftigkeit)

MongoDB gewährleistet die Dauerhaftigkeit von Datenänderungen durch die Verwendung eines Journals. Dieses Journal ist im Wesentlichen ein Protokoll, in dem jede Änderung an der Datenbank aufgezeichnet wird. Sobald eine Schreiboperation erfolgreich abgeschlossen ist, wird sie in das Journal eingetragen. Dieses Journal wird regelmäßig auf den Datenträger geschrieben, um sicherzustellen, dass die Daten auch bei einem Systemausfall nicht verloren gehen. (vgl. [29], Absatz: durable)

5.2.2 Clustermöglichkeiten

MongoDB bietet zwei grundlegende Clustertypen, um unterschiedliche Anforderungen an Skalierbarkeit und Verfügbarkeit zu erfüllen. Zum einen stehen sogenannte Replikationssets zur Verfügung, die auf einer asynchronen Primär-/Sekundär-Replikation basieren. Zum anderen kann ein Cluster aus Fragmenten (Shards), einem Konfigurationsserver und einem Queryserver aufgebaut werden. Um diese Clusterstrukturen effektiv zu verwalten und Anfragen im gesamten Cluster zu koordinieren, stellt MongoDB zwei Hauptkomponenten zur Verfügung: mongod und mongos.

mongod

MongoDB nutzt „mongod“ als seine zentrale Anwendung, die sämtliche Funktionen bereitstellt, die für den Betrieb eines eigenständigen Datenbanksystems oder eines Replikationssets notwendig sind. Der „mongod“-Prozess ist so konzipiert, dass er in zwei spezifische Funktionsbereiche unterteilt werden kann: den Datenserver und den Konfigurationsserver.

Der Datenserver, auch als Shardserver bekannt, ist die Komponente, die die Speicherung der eigentlichen Daten verwaltet und für die Bearbeitung von Anfragen zuständig ist. Er gewährleistet, dass Daten schnell und effizient gespeichert und abgerufen werden können. Der Konfigurationsserver hingegen hat eine andere Aufgabe: Er verwaltet die Benutzerkoordination und die Verteilung der Daten in einem Cluster. Der Konfigurationsserver speichert Metadaten, die im fragmentierten Cluster Informationen über die Verteilung der Daten enthalten und sicherstellen, dass Abfragen stets zu den richtigen Shards weitergeleitet werden.

In einem einzelnen, nicht fragmentierten System übernimmt eine einzige mongod-Instanz beide Rollen: die des Datenservers und die des Konfigurationsservers. Möchte man jedoch ein fragmentiertes Cluster erstellen, in dem Daten über mehrere Server verteilt werden, empfiehlt es sich, die beiden Aufgaben auf getrennte mongod-Instanzen aufzuteilen. Diese Trennung optimiert die Leistung und erleichtert die Verwaltung in größeren und komplexeren MongoDB-Umgebungen.

mongos

Mongos ist eine eigenständige Anwendung, die unabhängig von MongoDB-Datenbankinstanzen (mongod) arbeitet und speziell für den Betrieb von fragmentierten Clustern konzipiert wurde. Die Hauptaufgabe von Mongos ist das Annehmen und Weiterleiten von Anfragen in einem verteilten MongoDB-System. Aus diesem Grund wird Mongos auch als „Queryserver“ bezeichnet. Obwohl Mongos die Anfragen der Clients entgegennimmt, bearbeitet es diese nicht direkt, da es selbst keine Daten speichert. Stattdessen leitet Mongos die eingehenden Abfragen an einen Konfigurationsserver

(eine mongod-Instanz) weiter, der die Anfragen dann an die passenden Datenspeicher-Instanzen verteilt. Auf diese Weise fungiert Mongos als Vermittler im Cluster und stellt sicher, dass Abfragen an die richtigen Datenknoten weitergeleitet werden.

vgl. [30]

5.2.2.1 Replikation

MongoDB-Replikationssets basieren auf einem asynchronen Master-Slave-Replikationsmodell. Ein Replikationsset besteht aus einem Primärknoten (Primary) und mehreren Sekundärknoten (Secondary). Der Primärknoten empfängt alle Schreibvorgänge und speichert diese in einem speziellen Log, dem sogenannten Oplog. Dieses Oplog ist ein Ringpuffer, der die letzten Änderungen protokolliert. Die Sekundärknoten lesen den Oplog regelmäßig und wenden die aufgezeichneten Änderungen auf ihre eigenen Datenbestände an. Dadurch wird sichergestellt, dass alle Knoten im Replikationsset konsistente Daten enthalten, allerdings mit einer gewissen Latenz.

Clientanwendungen können sowohl auf den Primärknoten als auch auf die Sekundärknoten zugreifen. Schreibvorgänge müssen jedoch immer an den Primärknoten gesendet werden. Lesevorgänge können dagegen auf jedem beliebigen Knoten ausgeführt werden. Die Verteilung der Lesevorgänge auf die Sekundärknoten kann dazu beitragen, die Gesamtleistung des Systems zu verbessern, da die Sekundärknoten in der Regel nicht mit Schreibvorgängen belastet sind. vgl. [31]

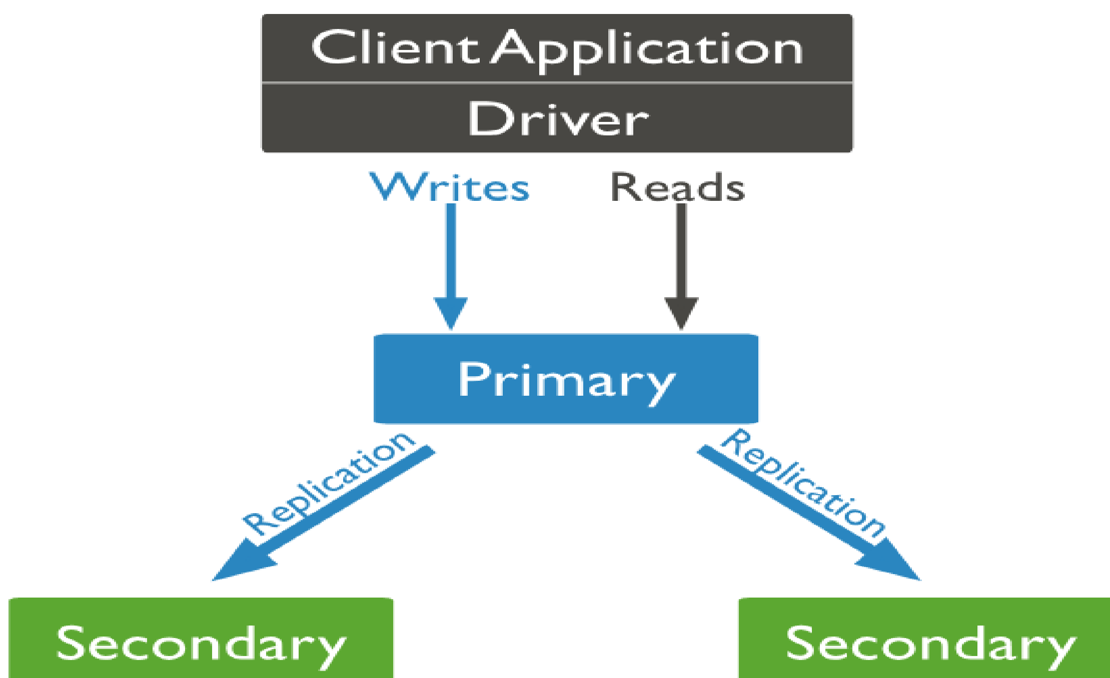


Abb. 7.2: Schema eines MongoDB Replikationssets bestehen aus einem Primär und zwei Sekundärknoten.

Skalierbarkeit, Verfügbarkeit und Performance

Skalierbarkeit

MongoDB-Replikationssets bieten eine hohe Flexibilität bei der Skalierung. Das Hinzufügen von Knoten ist relativ unkompliziert und kann je nach Bedarf durchgeführt werden.

Um ein Replikationsset zu erweitern, wird ein neuer Sekundärknoten hinzugefügt. Dieser Knoten führt zunächst eine vollständige Synchronisation mit dem Primärknoten durch. Dabei wird der gesamte Datenbestand kopiert. Sobald dieser initiale Synchronisierungsprozess abgeschlossen ist, wird der neue Sekundärknoten Teil des Replikationssets und beginnt, den Oplog des Primärknotens zu verfolgen. Dadurch bleibt der neue Knoten stets auf dem aktuellen Stand. vgl. [32]

Verfügbarkeit

MongoDB-Replikationssets bieten eine hohe Verfügbarkeit durch ihre automatische Failover-Funktionalität. Sollte der Primärknoten ausfallen, wird einer der Sekundärknoten automatisch zum neuen Primärknoten gewählt. Dieser Vorgang ist in der Regel transparent für die Anwendung, da die Clientanwendungen das gesamte Replikationsset als Einheit betrachten. Selbst wenn ein Sekundärknoten ausfällt, hat dies in der Regel keine unmittelbaren Auswirkungen auf die Verfügbarkeit, da die anderen Knoten weiterhin funktionsfähig sind. vgl. [31]

Performance

Ein wesentlicher Performance-Aspekt von MongoDB-Replikationssets ist der Schreib-Flaschenhals. Da alle Schreibvorgänge ausschließlich auf dem Primärknoten ausgeführt werden, kann die Schreibperformance nicht einfach durch Hinzufügen weiterer Knoten linear skaliert werden. Dies kann zu Leistungseinbußen führen, wenn die Schreiblast stark ansteigt.

Im Gegensatz dazu können Lesevorgänge durch die Verteilung auf mehrere Sekundärknoten parallelisiert werden. Dies ermöglicht eine signifikante Skalierung der Leseperformance. Allerdings ist zu beachten, dass die Daten auf den Sekundärknoten möglicherweise leicht veraltet sind, da die Replikation asynchron erfolgt. Daher eignen sich Replikationssets besonders gut für Anwendungen mit einer hohen Lese- und einer relativ geringen Schreiblast, bei denen eine gewisse Latenz bei den Lesevorgängen akzeptabel ist.

5.2.2.2 Fragmentiertes Cluster

Ein fragmentiertes Cluster, auch **Sharded Cluster** genannt, ist eine Datenbankarchitektur in MongoDB, und setzt sich aus drei wesentlichen Komponenten zusammen:

Shard-Server (Shards):

- Shards sind die eigentlichen Datenbankserver, auf denen die Daten gespeichert werden. Jeder Shard enthält eine bestimmte Teilmenge der Daten des gesamten Clusters.
- Shards bestehen oft aus Replikatsets, um hohe Verfügbarkeit und Datensicherheit sicherzustellen. Ein Replikatset sorgt zusätzlich für Ausfallsicherheit innerhalb eines Shards.
- Die Verteilung der Daten erfolgt basierend auf einem festgelegten „Sharding Key“. Dieser Schlüssel ist entscheidend für die Lastverteilung, da er festlegt, wie die Daten auf die Shards verteilt werden.

Konfigurationsserver (Config Servers):

- Konfigurationsserver speichern die Metadaten des Clusters, darunter Informationen über die Datenverteilung auf die einzelnen Shards.
- Diese Server fungieren als Replikatset und gewährleisten dadurch die Konsistenz und Zuverlässigkeit der Datenplatzierung im Cluster. Zudem steuern sie die Datenverteilung und sorgen dafür, dass Anfragen korrekt an die entsprechenden Shards weitergeleitet werden.

Query-Router (mongos):

- Der Query-Router, bekannt als mongos, dient als Schnittstelle zwischen Anwendungen und dem Sharded Cluster. Er nimmt die Anfragen der Clients entgegen und leitet sie an die passenden Shards weiter.
- mongos analysiert die Anfragen und greift auf die Metadaten des Konfigurationsservers zurück, um zu entscheiden, welcher Shard die gewünschten Daten enthält. Dadurch müssen Anwendungen nicht wissen, wo sich die Daten befinden – mongos übernimmt das Routing und die Weiterleitung der Anfragen.

  vgl. [33] (vgl. [34], Seite 5-17)

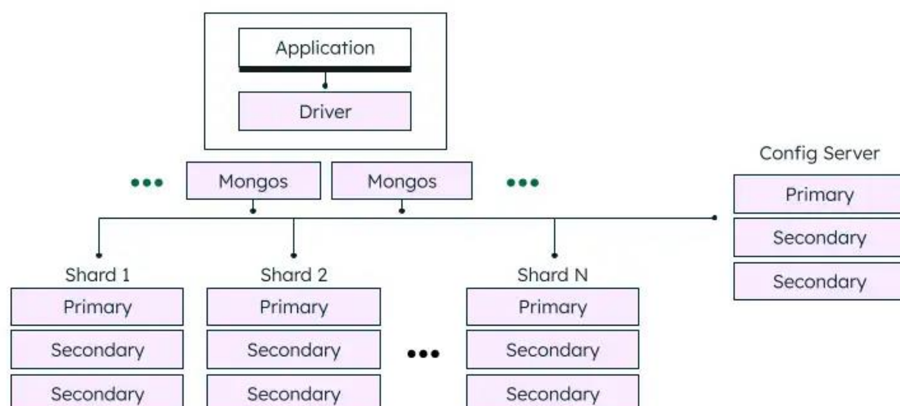


Abb. 7.3: MongoDB Cluster bestehend aus Queryserver, Konfigurationsserver und N-Fragmenten. Konfigurationssserver und die Fragmente sind als Replikationsset mit je 3 Knoten abgebildet.

Für die Fragmente gilt der Aufbau wie in Kap. 7.2.1 beschrieben. Es handelt sich also um asynchrone Master-Slave Replikationssets, bei dem Schreibanfragen vom Primär knoten verarbeitet werden. Je nach Konfiguration kann der Queryserver als Lastverteiler agieren und Leseanfragen an die Sekundärknoten der Fragmente weiterleiten.

Skalierbarkeit, Verfügbarkeit und Performance

Performance

Fragmentierte MongoDB-Cluster bieten hervorragende Möglichkeiten zur horizontalen Skalierung. Durch die Verteilung der Daten auf mehrere Fragmente kann die Last auf jeden einzelnen Server reduziert werden. Dies führt zu einer signifikanten Verbesserung der Lese- und Schreibperformance. Da MongoDB ein dokumentenorientiertes Datenbankmanagementsystem ist und keine relationalen Sperren verwendet, sind die Antwortzeiten in der Regel sehr kurz. Die asynchrone Replikation innerhalb der Fragmente kann jedoch bei hohen Schreiblasten zu einem Engpass werden, da alle Schreibvorgänge auf dem Primärserver eines jeden Replikats ausgeführt werden. Um dieses Problem zu lösen, kann die Anzahl der Fragmente erhöht werden, um die Schreiblast weiter zu verteilen. vgl. [32]; [31]

Verfügbarkeit

Die hohe Verfügbarkeit von MongoDB-Sharded Clustern wird durch die Verwendung von Replikationssets für alle wichtigen Komponenten gewährleistet. Jeder Shard sowie der Konfigurationsserver besteht aus mehreren Replikaten, die die Daten replizieren.

Sollte ein einzelnes Replikat ausfallen, übernimmt ein anderes Replikat automatisch seine Rolle. Erst wenn ein gesamtes Replikationsset offline geht, fehlt ein Teil der Daten. In diesem Fall kann der Cluster weiterhin betrieben werden, allerdings stehen die auf dem ausgefallenen Replikatsset gespeicherten Daten nicht zur Verfügung. Es liegt in der Verantwortung der Anwendung zu entscheiden, ob dieser Zustand akzeptabel ist.

Der Ausfall des gesamten Konfigurationsservers hat jedoch gravierende Folgen, da er die Metadaten des Clusters speichert. Ohne diese Informationen kann der Cluster nicht mehr ordnungsgemäß funktionieren. In diesem Fall ist das gesamte Cluster nicht mehr verfügbar. vgl. [35]; [36] (vgl. [34], S. 43)

Skalierbarkeit

MongoDB-Cluster bieten eine außergewöhnliche Flexibilität bei der Skalierung. Sowohl Replikationssets als auch Fragmente können dynamisch angepasst werden, um sich an wechselnde Anforderungen anzupassen. Das Hinzufügen oder Entfernen von Knoten in einem Replikationsset ist ein unkomplizierter Vorgang, solange die Mindestanzahl von drei Knoten eingehalten wird.

Die Skalierung von Fragmenten ist ebenfalls sehr flexibel. Ein Cluster kann theoretisch eine unbegrenzte Anzahl von Fragmenten enthalten. MongoDB verteilt die Daten automatisch auf die Fragmente, sodass eine gleichmäßige Lastverteilung gewährleistet ist. Beim Hinzufügen eines neuen Fragments werden die Daten automatisch neu verteilt. Beim Entfernen eines Fragments werden die Daten des entfernten Fragments auf die verbleibenden Fragmente umverteilt. Dieser Prozess garantiert eine kontinuierliche Verfügbarkeit des Clusters, auch während der Skalierung. vgl. [37]

5.3 Cassandra

Apache Cassandra ist eine hochskalierbare, dezentrale NoSQL-Datenbank, die für die Verwaltung großer Datenmengen optimiert ist. Als Wide-Column-Store speichert Cassandra Daten in sogenannten Column-Families, die ähnlich wie Tabellen aufgebaut sind. Innerhalb dieser Familien können nahezu unbegrenzt viele Spalten angelegt werden. Diese flexible Datenstruktur, die im Grunde auf Schlüssel-Wert-Paaren basiert, ermöglicht eine effiziente Skalierung. Cassandra ist für den Einsatz in verteilten Systemen konzipiert und kann auf Hunderten von Knoten betrieben werden, um eine hohe Verfügbarkeit und Leistung zu gewährleisten.

In Kapitel 8.1 wird zunächst das Transaktionskonzept erläutert, gefolgt von den möglichen Clusteraufbauten in Kapitel 8.2. Anschließend werden in Kapitel 8.2.1 die Replikationsmöglichkeiten innerhalb eines Cassandra-Clusters beschrieben.

5.3.1 Transaktionskonzept

Cassandra unterstützt, ähnlich wie MongoDB, keine Transaktionen im klassischen Sinne. Im Folgenden wird ein Überblick gegeben, wie der Mehrbenutzerbetrieb in Cassandra organisiert ist.

Atomicity (Atomarität)

Im Gegensatz zu relationalen Datenbanken, die in der Regel starke Atomizität garantieren, bietet Cassandra ein schwächeres Atomitätsniveau. Während das Einfügen oder Aktualisieren einer einzelnen Zeile als atomar gilt, sind komplexere Operationen, die mehrere Zeilen betreffen, eine Folge von Einzeloperationen. Dies bedeutet, dass es keine garantierte Isolation für gleichzeitige Änderungen an mehreren Zeilen gibt. Die letztendlich in der Datenbank gespeicherte Änderung ist diejenige, die zuletzt geschrieben wurde. vgl. [38]

Konsistenz

Cassandra bietet eine **einstellbare Konsistenz(**Tunable Consistency**)**, die es Benutzern ermöglicht, den Grad der Konsistenz für Lese- und Schreiboperationen anzupassen. Dies geschieht durch die Wahl, wie viele Knoten im Cluster eine Bestätigung für einen DML-Befehl (Data Manipulation Language) oder eine SELECT-Abfrage liefern müssen,

Bildquelle: Apache Cassandra Logo, siehe https://de.pinterest.com/pin/apache-cassandra-logo--80853755803115895/; abgerufen am 15.12.2024.

bevor die Operation als erfolgreich gilt. Dadurch können Benutzer einen Kompromiss zwischen Verfügbarkeit und Konsistenz eingehen. vgl. [39]

Isolation

In Cassandra gilt die Isolationsebene ausschließlich für einzelne Zeilen. Das bedeutet, dass eine Änderung an einer bestimmten Zeile erst für andere Clients sichtbar wird, wenn diese Änderung vollständig abgeschlossen ist. Solange eine Änderung noch in Bearbeitung ist, können andere Clients nicht auf die Zwischenzustände zugreifen. vgl. [40]

Durability (Dauerhaftigkeit)

Cassandra gewährleistet eine hohe Datenpersistenz durch die Verwendung eines Commitlogs. Jede Änderung an den Daten wird zunächst im Commitlog protokolliert, bevor sie auf die Festplatte geschrieben wird. Dadurch wird sichergestellt, dass im Falle eines Absturzes oder eines Stromausfalls keine Daten verloren gehen. Beim Neustart des Systems werden die im Commitlog protokollierten Änderungen wieder eingespielt, um die Datenkonsistenz wiederherzustellen. vgl. [41]

5.3.2 Clusteraufbau

Cassandra-Cluster unterscheiden sich grundlegend von denen relationaler Datenbanken wie MariaDB oder MongoDB. Anstatt einer zentralen Instanz, die alle Daten speichert, besteht ein Cassandra-Cluster aus mehreren Knoten, die horizontal fragmentiert sind. Das bedeutet, dass jede Zeile einer Tabelle einem bestimmten Knoten zugeordnet ist. Diese Zuordnung erfolgt basierend auf einem Token, das durch eine Partitionierungsfunktion berechnet wird.

Token

Ein Token in Cassandra ist eine numerische Kennung, die durch eine Hashfunktion aus dem Partitionsschlüssel berechnet wird. Der Wertebereich eines Tokens erstreckt sich von TOKEN_MIN bis TOKEN_MAX, wobei dieser Bereich von der gewählten Partitionierungsfunktion abhängt. Jeder Knoten in einem Cluster ist für einen bestimmten Tokenbereich verantwortlich. Um eine gleichmäßige Lastverteilung zu gewährleisten, wird der gesamte Tokenbereich in gleich große Segmente aufgeteilt, wobei jedes Segment einem Knoten zugewiesen wird. Die Anzahl der Token pro Knoten ergibt sich aus der Division von TOKEN_MAX durch die Anzahl der Knoten im Cluster.

Partitionierfunktion (Hashfunktion)

Die Partitionierfunktion ist eine Hashfunktion. Sie errechnet aus dem Primärschlüssel einer Zeile einen Hashwert zwischen TOKEN_MIN und TOKEN_MAX. Dieser Hashwert wird in diesem Zusammenhang Token genannt.

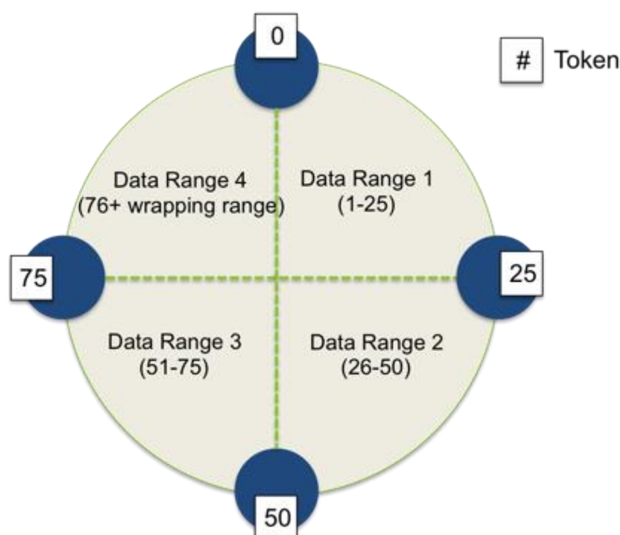


Abb. 8.2: Cassandra Cluster mit 4 Knoten. Die Zahl innerhalb der Knoten gibt das zugewiesene Token an. Die Gesamtanzahl der Token beträgt 100. TOKEN_MIN = 0 und TOKEN_MAX = 100. Jeder Knoten ist somit für 25 Token zuständig[1].

Bild 8.2 zeigt einen stark vereinfachten Aufbau eines Cassandra Clusters, bestehend aus einem einzigen Datacenter. Jeder Knoten ist zuständig für die Verwaltung von 25 Token. TOKEN_MIN ist mit 0 festgelegt und TOKEN_MAX mit 100. Wird eine neue Zeile eingefügt, errechnet die Partitionierfunktion aus dem Primärschlüssel der Zeile einen Wert. Dieser bestimmt, welcher Knoten die Zeile speichern wird.

Im Beispiel aus Bild 8.2 verwaltet Knoten 2 Token 1-25, Knoten 3 verwaltet Token 26-50, Knoten 4 verwaltet Token 51-75, und Knoten 1 verwaltet 76-100, da ihm die 0 zugeordnet ist. Jeder Knoten speichert also die Zeilen ab, deren Token zwischen der Zuordnung des Vorgängerknotens und des eigenen Tokens liegen. Grundsätzlich ist jeder Knoten in Cassandra gleichberechtigt. Anwendungen können sich mit jedem der Knoten verbinden und Daten abfragen.

vgl. [42]

[1]Grafik. angelehnt an: https://docs.datastax.com/en/archived/cassandra/1.1/_images/ring_partitions.png; abgerufen am 19.12.2025.

5.3.2.1 Replikation

Wie zu Beginn des Kapitels erwähnt, bietet Cassandra die Möglichkeit, Knoten in Racks einzuteilen, um die physische Anordnung von Servern im Rechenzentrum abzubilden. Die Zuordnung der Knoten zu Racks erfolgt durch einen sogenannten Snitch. Der Standard-Snitch ist das 'Dynamic Snitching', welches die Knoten dynamisch anhand von Latenz und Auslastung gruppiert. Diese dynamische Anpassung ermöglicht eine optimale Verteilung der Daten und eine hohe Verfügbarkeit. So können beispielsweise bei einer erhöhten Last auf einem Rack automatisch Replikate auf andere Racks verschoben werden. vgl. [43]

Replikationsfaktor

Der Replikationsfaktor in Cassandra bestimmt, wie oft jede Zeile im Cluster repliziert wird. Je höher der Replikationsfaktor, desto mehr Kopien einer Zeile werden auf verschiedenen Knoten gespeichert. Dies erhöht die Redundanz und verbessert die Verfügbarkeit, da selbst wenn mehrere Knoten ausfallen, die Daten weiterhin zugänglich bleiben. Gleichzeitig führt ein hoher Replikationsfaktor zu einer geringeren Fragmentierung der Daten.

Beispielsweise ist bei einer Clustergröße von sechs Knoten ein Replikationsfaktor von 3 eingestellt, so verfügt jeder Knoten über 50 % des Datenbestandes. Bei einem Replikationsfaktor von 6 würde jeder Knoten über sämtliche Daten verfügen.

Replikationsstrategie

Die Replikationsstrategie legt fest, auf welchen Knoten die Datenreplikate gespeichert werden. In Cassandra stehen dafür zwei Optionen zur Verfügung: die *SimpleStrategy* und die *NetworkTopologyStrategy*.

SimpleStrategy

Die SimpleStrategy ist eine einfache und effiziente Strategie zur Verteilung von Daten in einem Cassandra-Cluster, der sich auf ein einziges Datacenter beschränkt. Bei jeder Schreiboperation wird anhand des Partitionsschlüssels ein Token berechnet. Dieses Token bestimmt den primären Knoten, auf dem die Daten gespeichert werden. Anschließend werden die Replikate der Daten gemäß dem konfigurierten Replikationsfaktor auf die nächsten Knoten im Uhrzeigersinn verteilt.

NetworkTopologyStrategy

Die NetworkTopologyStrategy ist speziell für die Verteilung von Daten über mehrere Racks und Datacenter konzipiert. Sie ermöglicht eine gezielte Verteilung der Replikate auf verschiedene Racks innerhalb eines Datacenters und sogar auf andere Datacenter. Dadurch wird die Ausfallsicherheit erhöht, da ein einzelner Ausfall (z.B. ein Stromausfall in einem Rack) weniger Daten gefährdet. Cassandra versucht, die Replikate so zu verteilen, dass sie auf möglichst unterschiedlichen Racks liegen. Sollte die gewünschte Verteilung auf Rack-Ebene nicht möglich sein, fällt die Strategie auf die SimpleStrategy zurück und verteilt die Replikate im Uhrzeigersinn. vgl. [44]

5.3.2.2 Skalierbarkeit, Verfügbarkeit und Performance

Skalierbarkeit

Cassandra bietet eine flexible Skalierbarkeit, indem es das Hinzufügen neuer Knoten zu einem bestehenden Datacenter oder sogar das Hinzufügen neuer Datacenter zu einem laufenden Cluster unterstützt. Voraussetzung für das Hinzufügen eines neuen Datacenters ist die Verwendung der *NetworkTopologyStrategy* als Replikationsstrategie. Wenn ein neuer Knoten oder ein neues Datacenter hinzugefügt wird, übernimmt Cassandra automatisch die Synchronisierung dieser neuen Knoten mit dem Cluster und weist die erforderlichen Token neu zu. Dieser Synchronisierungsprozess wird als *Bootstrapping* bezeichnet und erfolgt mit Unterstützung des *seed-provider*. Der seed-provider ist ein speziell konfigurierter Knoten im Cluster, der als Quelle für die Token-Zuordnung und die initialen Informationen für neue Knoten dient. Für jedes Datacenter muss mindestens ein Knoten als seed-provider in der Konfigurationsdatei festgelegt sein. Sobald ein neuer Knoten hinzugefügt wird, erhält er vom seed-provider die für ihn bestimmten Token und lädt die relevanten Daten von den anderen Knoten im Datacenter herunter, um seine Synchronisation abzuschließen. Ansonsten fungiert der seed-provider wie ein ganz normaler Teil des Datacenters. vgl. [45]; [46]

Verfügbarkeit

Cassandra ist als AP-System konzipiert, das heißt, es priorisiert Verfügbarkeit und Partitionstoleranz. Durch die dezentrale Architektur und die robuste Replikationsstrategie gewährleistet Cassandra eine hohe Verfügbarkeit, selbst wenn einzelne Knoten oder ganze Datacenter ausfallen, kann der Client weiterhin mit anderen Knoten kommunizieren, und der Koordinator kümmert sich um die Weiterleitung der Anfragen. Bei einem Replikatausfall kann der Koordinator den Client mit einer „UnavailableException“ informieren, falls das Replikat bereits vor der Anfrage ausgefallen ist. Andernfalls wird bei einem späteren Ausfall eine „TimedOutException“ zurückgegeben. Diese Timeout-Antwort ist jedoch kein Fehler, Cassandra speichert die Anfrage und versucht, sie erneut zu senden, sobald der Replikat wiederhergestellt ist. vgl. [47]

Performance

Cassandra zeichnet sich durch eine hohe Performance aus, die auf mehreren Faktoren beruht. Zum einen verzichtet Cassandra auf komplexe Transaktionsmechanismen und Integritätsprüfungen, die in relationalen Datenbanken üblich sind. Dies ermöglicht eine schnelle Verarbeitung von Lese- und Schreibanfragen. Zum anderen erlaubt die clientseitige Konfigurierbarkeit der Konsistenzstufe eine flexible Anpassung an die Anforderungen der Anwendung. Je nach Anwendungsfall kann eine höhere Konsistenz gegen eine geringere Latenz eingetauscht werden.

6. Anwendungsbeispiele

Die folgenden Fallbeispiele zeigen, wie die zuvor analysierten Datenbanksysteme, wie MariaDB und Cassandra, in spezifischen Anwendungsbereichen eingesetzt werden. Es wird erläutert, wie die für diese Systeme charakteristischen Eigenschaften wie Replikation, Konsistenzmodelle und Skalierbarkeit praktisch umgesetzt werden.

Fallbeispiel 1: Finanztransaktionen in einer Bank

In einer modernen Bank, die Millionen von Finanztransaktionen täglich verarbeitet, sind Konsistenz und Datenintegrität von höchster Priorität. Die Bank nutzt MariaDB als primäres Datenbanksystem, um Buchungen, Kontostände und Zahlungsaufträge effizient und sicher zu verwalten. Die Entscheidung für MariaDB basiert auf der Fähigkeit der Datenbank, das ACID-Modell zu unterstützen, was für Finanzsysteme unverzichtbar ist.

Herausforderungen:
Finanztransaktionen müssen atomar und konsistent verarbeitet werden. Eine Überweisung sollte entweder vollständig durchgeführt oder vollständig zurückgerollt werden, um sicherzustellen, dass keine Gelder verloren gehen oder doppelt gebucht werden. Zudem erwartet die Bank eine hohe Verfügbarkeit ihres Systems, um auch bei Serverausfällen betriebsbereit zu bleiben.

Technische Lösung:
Die Bank setzt MariaDB in einer Galera-Cluster-Konfiguration ein. Dieser Cluster besteht aus mehreren Datenbankknoten, die synchrone Replikation unterstützen. Jeder Knoten im Cluster kann sowohl Schreib- als auch Leseanfragen verarbeiten, wodurch eine hohe Verfügbarkeit gewährleistet wird.

Ablauf einer Transaktion:

1. Ein Kunde initiiert eine Überweisung von 1.000 Euro von seinem Konto auf ein anderes.
2. Die Transaktion wird zunächst im Galera-Cluster auf dem primären Knoten verarbeitet und gleichzeitig auf die anderen Knoten repliziert.
3. MariaDB garantiert, dass die Buchung atomar erfolgt: Entweder wird der Betrag vom sendenden Konto abgezogen und dem empfangenden Konto gutgeschrieben, oder die gesamte Transaktion wird abgebrochen.
4. Sollte während der Verarbeitung ein Fehler auftreten, wie z. B. ein Knotenausfall, übernimmt automatisch ein anderer Knoten die Anfrage.

Ergebnisse:
Durch den Einsatz von MariaDB als Datenbanklösung gewährleistet die Bank strikte Konsistenz und Datenintegrität. Die synchrone Replikation minimiert das Risiko von Datenverlusten, und die Ausfallsicherheit des Clusters sorgt für eine unterbrechungsfreie Verfügbarkeit, selbst bei technischen Störungen. Im Vergleich zu NoSQL-Datenbanken wie MongoDB oder Cassandra, die in bestimmten Szenarien eine

höhere Skalierbarkeit bieten, würde deren eventual consistency für Finanztransaktionen nicht ausreichen, da sie keine strikte Konsistenz garantieren können.

Fallbeispiel 2: IoT-Datenverarbeitung in einem Smart-City-Projekt

Eine Stadtverwaltung implementiert ein Smart-City-Projekt, das Daten von Tausenden von Sensoren sammelt, um die Verkehrssteuerung, Luftqualität und Energieverbrauch in Echtzeit zu überwachen. Cassandra wird als Datenbank gewählt, da sie speziell für große, verteilte Systeme mit hohen Anforderungen an Schreibgeschwindigkeit und Verfügbarkeit entwickelt wurde.

Herausforderungen:
Das Projekt muss große Mengen an IoT-Daten schnell verarbeiten und speichern können. Gleichzeitig ist es wichtig, dass die Datenbank auch bei Ausfällen einzelner Server oder Netzwerkprobleme weiterhin verfügbar bleibt. Die Sensordaten sind stark fragmentiert, und unterschiedliche Datenquellen erfordern ein flexibles Schema.

Technische Lösung:
Cassandra wird in einem Cluster mit einer Peer-to-Peer-Architektur eingesetzt. Die Datenbank verteilt die Daten gleichmäßig über alle verfügbaren Knoten im Cluster, basierend auf einem Partitionsschlüssel und einer Token-basierten Partitionierungsfunktion. Jeder Knoten ist für einen Teil des Datenraums verantwortlich, und die Daten werden mit einem Replikationsfaktor von drei auf mehrere Knoten repliziert, um Redundanz und Verfügbarkeit zu gewährleisten.

Ablauf der Datenverarbeitung:

1. Verkehrssensoren erfassen in Echtzeit die Anzahl der Fahrzeuge an einer Kreuzung und senden die Daten an Cassandra.
2. Die Partitionierungsfunktion weist die Daten basierend auf dem Sensorstandort einem bestimmten Knoten im Cluster zu.
3. Replikation stellt sicher, dass dieselben Daten auf mindestens drei Knoten gespeichert werden, um Ausfallsicherheit zu gewährleisten.
4. Wenn eine Anfrage an Cassandra gesendet wird, z. B. zur Abfrage der Verkehrsdichte in einer bestimmten Zone, wird sie direkt an den Knoten geleitet, der für diese Daten zuständig ist, wodurch die Abfrage sehr effizient wird.

Ergebnisse:
Die Stadt profitiert von einer hochskalierbaren Datenbanklösung, die in der Lage ist, Millionen von Schreibvorgängen pro Sekunde zu verarbeiten. Selbst bei einem Ausfall mehrerer Knoten bleibt Cassandra funktionsfähig, da andere Knoten die Aufgaben übernehmen. Die Fähigkeit, Sensordaten schnell zu verarbeiten und zu speichern, unterstützt die Echtzeitentscheidungen der Stadtverwaltung und verbessert die Lebensqualität der Bürger. Im Gegensatz dazu könnten relationale Systeme wie MariaDB in diesem Szenario aufgrund der hohen Datenmengen und der intensiven Schreiblast überfordert sein, da sie primär für konsistenzorientierte Anwendungen ausgelegt sind.

7. Fazit

Datenbanksysteme sind weit mehr als die Prinzipien CAP, ACID oder BASE. Ihre Vor- und Nachteile sind stark konfigurationsabhängig, sodass Systeme je nach Einstellung zwischen diesen Konzepten variieren können. Die in dieser Ausarbeitung betrachteten Datenbanksysteme erfüllen die ACID-Anforderungen meist nur teilweise. Durch Replikationstechniken kann zudem eine Form von "letztendlicher Konsistenz" erreicht werden, wobei sich der Datenbestand in replizierten Systemen leicht verzögert an den Zustand des Master-Systems angleicht.

Die Frage, ob ein System allein aufgrund von Replikationsverzögerungen nicht als ACID-konform gilt, ist schwer eindeutig zu beantworten, da es hierzu an präzisen, offiziellen Definitionen mangelt. Ebenso lässt sich keine klare Linie zwischen ACID und BASE ziehen – diese Konzepte markieren vielmehr Endpunkte eines Spektrums, auf dem sich die meisten Datenbanksysteme bewegen. Selbst NoSQL-Systeme, die oft dem BASE-Ansatz zugerechnet werden, folgen teilweise ACID-Prinzipien oder versuchen es zumindest.

Relationale Systeme wie MariaDB versprechen ACID-Konformität, erreichen diese jedoch nur bei hohem Isolationsgrad. In verteilten Umgebungen wird dieser Grad meist vermieden, da Sperren die Parallelität einschränken und eine effiziente horizontale Skalierung behindern würden.

Die Clusterarchitekturen von MariaDB, MongoDB und Cassandra unterscheiden sich deutlich, doch die Konzepte im Hintergrund weisen einige Gemeinsamkeiten auf. So bieten sowohl MariaDB als auch MongoDB asynchrone Replikation. Cassandra hingegen verwendet je nach „Tunable Consistency“-Einstellung eine synchrone oder asynchrone Replikation.

Bei MongoDB und Cassandra liegt die Verwaltung bestimmter Datenintegritätsprüfungen – wie das Überprüfen von Referenzen oder das Erstellen fortlaufender Nummern – bei der Clientanwendung. Diese Systeme sind für typische NoSQL-Anforderungen optimiert: große Datenmengen effizient zu speichern und hohe Verfügbarkeit sicherzustellen. Integritätsprobleme, wie sie in relationalen Systemen auftreten können, entstehen hier oft gar nicht erst. Beispielsweise gibt es keine Fremdschlüsselprüfungen, da keine Fremdschlüssel vorhanden sind, und Transaktionen werden meist nicht benötigt, weil zusammenhängende Daten in MongoDB in einem Dokument und in Cassandra in einer Zeile organisiert sind. Änderungen an diesen Datenstrukturen erfolgen atomar und machen so klassische Transaktionen überflüssig.

NoSQL-Datenbanken sind daher nicht weniger leistungsfähig, nur weil sie die ACID-Prinzipien weniger strikt umsetzen – sie sind vielmehr anders aufgebaut, was die Auswahl eines geeigneten Systems je nach Anwendung erleichtert. Dabei ist es oft zweitrangig, in welche CAP-Kategorie ein System fällt; entscheidend ist die spezifische Eignung für den jeweiligen Anwendungsfall.

Hinsichtlich der Skalierbarkeit liegt die Zukunft vermutlich in einer weitergehenden Fragmentierung des Datenbestands. Hier haben MongoDB und Cassandra einen klaren Vorteil, da MariaDB von Haus aus keine Datenfragmentierung und auch keinen integrierten Lastverteiler anbietet. Meine Einschätzung ist daher: Wenn eine möglichst ACID-nahe Arbeitsweise gewünscht ist, bietet der Galera-Cluster eine praktikable Lösung. Stehen jedoch Skalierbarkeit und Lastverteilung im Vordergrund, sind MongoDB und besonders Cassandra besser geeignet.

Literaturverzeichnis:

1. Axel Feix. Skalierbare Anwendungsarchitekturen. https://www.informatik-aktuell.de/entwicklung/methoden/skalierbare-anwendungsarchitektur.html; abgerufen am 09.10.2024.
2. Angela Gruber. »Kleiner geht's nicht, Physikalische Grenze der Chip Entwicklung«. In: Spiegel Online (2023). https://www.spiegel.de/netzwelt/web/moore-s-law-die-goldene-regel-der-chiphersteller-broeckelt-a-1083468.html; abgerufen am 11.10.2024.
3. Erhard Rahm, Gunther Saake und Kai-Uwe Sattler. Verteiltes und Paralleles Datenmanagement. 1. Aufl. Springer Vieweg, 2015.
4. M. Tamer Özsu und Patrick Valduriez. Principles of Distributed Database Systems. 3.Aufl. Springer, 2011.
5. Oracle. Replication and Synchronization Modes. https://docs.oracle.com/cd/E19359-01/819-6148-10/chap2.html; abgerufen am 15.10.2024.
6. Martin Kleppmann. Please stop calling databases CP or AP. https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html; abgerufen am 02.11.2024.
7. Julian Browne. Brewer's CAP Theorem. https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/; abgerufen am 04.11.2024.
8. Oracle. Two-Phase Commit Mechanism. https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222; abgerufen am 09.11.2024.
9. Andreas Maier und Michael Kaufmann. SQL- & NoSQL-Datenbanken. 8. Aufl. Springer Vieweg, 2016.
10. Charles Roe. ACID vs. BASE: The Shifting pH of Database Transaction Processing. https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/; abgerufen am 12.11.2024.
11. MariaDB. MariaDB in Kürze: MariaDB. https://mariadb.org/de/; abgerufen am 18.11.2024.
12. MariaDB. MariaDB Dokumentation: Isolation Levels. https://mariadb.com/kb/en/set-transaction/; abgerufen am 18.11.2024.
13. Codership Ltd. Galera Dokumentation: ISOLATION LEVELS. https://galeracluster.com/library/documentation/isolation-levels.html; abgerufen am 18.11.2024.
14. MariaDB. MariaDB Dokumentation: Overview of the Binary Log. https://mariadb.com/kb/en/overview-of-the-binary-log/; abgerufen am 20.11.2024.
15. MariaDB. MariaDB Dokumentation: Global Transaction ID. https://mariadb.com/kb/en/gtid/; abgerufen am 20.11.2024.
16. MariaDB. MariaDB Dokumentation: Replication Overview. https://mariadb.com/kb/en/replication-overview/; abgerufen am 20.11.2024.
17. Oli sennhauser. MariaDB/MySQL Replication for Beginners. Foliensatz einer MariaDB Präsentation. Erhältlich unter https://de.slideshare.net/slideshow/mysql-replication-for-beginners/15658462; abgerufen am 21.11.2024.
18. MariaDB. MariaDB Dokumentation: Setting Up Replication. https://mariadb.com/kb/en/setting-up-replication/; abgerufen am 23.11.2024.
19. MariaDB. MariaDB Dokumentation: About Galera Replication. https://mariadb.com/kb/en/about-galera-replication/; abgerufen am 24.11.2024.

20. MariaDB. MariaDB Dokumentation: Was ist MariaDB Galera Cluster? https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/; abgerufen am 24.11.2024.
21. Codership Ltd. Galera Dokumentation: CERTIFICATION-BASED REPLICATION. https://galeracluster.com/library/documentation/certification-based-replication.html; abgerufen am 25.11.2024.
22. Codership Ltd. Galera Dokumentation: ISOLATION LEVELS. https://galeracluster.com/2024/09/transaction-isolation-levels-in-galera-cluster/; abgerufen am 25.11.2024.
23. Codership Ltd. Galera Dokumentation: WEIGHTED QUORUM. https://galeracluster.com/library/documentation/weighted-quorum.html; abgerufen am 27.11.2025.
24. Codership Ltd. Galera Dokumentation: State Transfers. https://galeracluster.com/library/documentation/state-transfer.html; abgerufen am 28.11.2024.
25. MongoDB. IONOS Digital Guide : Was ist MongoDB?. https://www.ionos.de/digitalguide/websites/web-entwicklung/mongodb-vorstellung-und-vergleich-mit-mysql/; abgerufen am 01.12.2024.
26. MongoDB Inc. MongoDB Dokumentation: Atomicity and Transactions. https://www.mongodb.com/docs/manual/core/write-operations-atomicity/; abgerufen am 01.12.2024.
27. MongoDB Inc. MongoDB Dokumentation: Unique Indexes. https://www.mongodb.com/docs/manual/core/index-unique/; abgerufen am 01.12.2024.
28. MongoDB Inc. MongoDB Dokumentation: Read Isolation, Consistency, and Recency. https://www.mongodb.com/docs/manual/core/read-isolation-consistency-recency/; abgerufen am 02.12.2024.
29. MongoDB Inc. MongoDB Glossar. https://www.mongodb.com/docs/manual/reference/glossary/; abgerufen am 02.12.2024.
30. MongoDB Inc. MongoDB Dokumentation: MongoDB Package Components. https://www.mongodb.com/docs/manual/reference/program/; abgerufen am 04.12.2024.
31. MongoDB Inc. MongoDB Dokumentation: Replication. https://www.mongodb.com/docs/manual/replication/; abgerufen am 07.12.2024.
32. MongoDB Inc. MongoDB Dokumentation: Replica Set Data Synchronization. https://www.mongodb.com/docs/manual/core/replica-set-sync/; abgerufen am 07.12.2024.
33. MongoDB Inc. MongoDB Dokumentation: Production Cluster Architecture. https://www.mongodb.com/docs/manual/core/sharded-cluster-components/; abgerufen am 09.12.2024.
34. Kristina Chodorow. Scaling MongoDB. 1. Aufl. O'Reilly Media, 2011.
35. MongoDB Inc. MongoDB Dokumentation: Config Servers. https://www.mongodb.com/docs/manual/core/sharded-cluster-config-servers/; abgerufen am 12.12.2024.
36. MongoDB Inc. MongoDB Dokumentation: Troubleshoot Sharded Clusters. https://www.mongodb.com/docs/manual/tutorial/troubleshoot-sharded-clusters/; abgerufen am 13.12.2024.

37. MongoDBInc. MongoDB Dokumentation: Sharded Cluster Administration. https://www.mongodb.com/docs/manual/administration/sharded-cluster-administration/; abgerufen am 13.12.2024.
38. Datastax Inc. Cassandra Dokumentation: Atomicity. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_atomicity_c.html; abgerufen am 15.12.2024.
39. Datastax Inc. Cassandra Dokumentation: Consistency. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_tunable_consistency_c.html; abgerufen am 15.12.2024.
40. Datastax Inc. Cassandra Dokumentation: Isolation. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_isolation_c.html; abgerufen am 15.12.2024.
41. Datastax Inc. Cassandra Dokumentation: Durability. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_durability_c.html; abgerufen am 16.12.2024.
42. Apache. Cassandra Dokumentation: About Data Partitioning in Cassandra. https://docs.datastax.com/en/archived/cassandra/1.1/docs/cluster_architecture/partitioning.html; abgerufen am 19.12.2024.
43. Datastax Inc. Cassandra Dokumentation: Snitches. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html; abgerufen am 20.12.2024.
44. Datastax Inc. Cassandra Dokumentation: Data replication. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html; abgerufen am 22.12.2024.
45. Datastax Inc. Cassandra Dokumentation: Replacing a dead node or dead seed node. https://docs.datastax.com/en/archived/cassandra/2.1/cassandra/operations/opsReplaceNode.html; abgerufen am 25.12.2024.
46. Datastax Inc. Cassandra Dokumentation: Adding nodes to an existing cluster. https://docs.datastax.com/en/archived/cassandra/2.2/cassandra/operations/opsAddNodeToCluster.html; abgerufen am 25.12.2024.
47. JONATHAN ELLIS. When a timeout is not a failure: how Cassandra delivers high availability. https://www.datastax.com/blog/when-timeout-not-failure-how-cassandra-delivers-high-availability-part-1; abgerufen am 26.12.2024.vv

Anhang

. Eigenständigkeitserklärung

. Glossar

Eigenständigkeitserklärung

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original

Glossar

3D-NAND-Technologie: Speicher, bei dem die Transistoren vertikal zur Chipfläche stehen und in mehreren Ebenen verbaut werden.

ACID: Steht für Atomicity Consistency Isolation Durability. Beschreibt vier Eigenschaften. Wird oft als Transaktionskonzept bezeichnet.

BASE: Steht für Basically available, Soft state, Eventual consistency. Wird oft als Transaktionskonzept bezeichnet. viele NoSQL Systeme werden dieser Kategorie zugeordnet.

Cluster: Zusammenschluss verschiedener unabhängiger Computer zur Steigerung von Rechenleistung und Ausfallsicherheit.

Collections: MongoDB spezifischer Begriff. Eine Collection ist eine Ansammlung von Dokumenten.

Column-Families: Cassandra spezifischer Begriff. Eine Column-Family ist eine Ansammlung von Zeilen mit ähnlichen Spalten.

Commit: Endgültiges Speichern einer Manipulation der Datenbank.

Datacenter: Englisch für Rechenzentrum. Ein Rechenzentrum ist ein Ort, an dem meist eine gute Anbindung an das Internet besteht, und an dem viele Server stehen. Im Zusammenhang mit Cassandra wird der Begriff als Teilkomponente eines Clusters genutzt.

Datenbankcluster: Zusammenschluss mehrerer Datenbankinstanzen zur Steigerung von Rechenleistung und Ausfallsicherheit eines Datenbanksystems.

Fragment: Zerteilung einer Datenbank in Stücke/Splitter zur besseren Verteilung der Rechenaufgaben. Jeder Fragment kennt nur bestimmte Teile der Datenbank.

Lastverteiler: Verteilt Anfragen auf verschiedene Replikationen, um diese möglichst gleichmäßig auszulasten.

Latenz: Verzögerung der Kommunikation zwischen zwei Rechnern.
Linearisierbarkeit: Linearisierbarkeit beschreibt ein Korrektheitskriterium, bei dem nebenläufig ablaufende Vorgänge mit Zugriff auf Datenstrukturen sich so verhalten, als würden sie sequentiell ausgeführt.

Loadbalancing: Verteilung der Last des Gesamtsystems auf mehrere Teilsysteme.

Master-Slave-Replikation: Strategie, bei denen paarweise Knoten in Master und Slave unterteilt werden. Der Masterknoten ist vollwertiges Mitglied im Cluster während der Slave nur unter bestimmten Umständen agiert.

mysqldump: Software zur Durchführung logischer Datensicherung von MySQL und MariaDB Datenbanken.

Partitionierungsfunktion: Cassandra spezifischer Begriff. Die Partitionierungsfunktion errechnet aus einem Eingabetext einen Hashwert.

Partitionstoleranz: Gibt an, wie tolerant ein verteiltes Datenbanksystem hinsichtlich der Isolation von Teilsystemen ist. Je nach Implementation kann das Datenbanksystem einen Verlust verkraften oder nicht.

Queryserver: Komponente eines MongoDB Clusters. Er nimmt Anfragen entgegen und leitet sie an das entsprechende Fragment weiter.

Redundanz: Mehrfaches Vorhandensein ein und der selben Information.

Replikationsset: Bezeichnet eine Anzahl von Knoten, die miteinander arbeiten und denselben Datenbestand verwalten.

rsync: Linux-Software zur Synchronisation von Dateien und Verzeichnissen zwischen zwei Rechnern.

Snitch: Cassandra spezifischer Begriff. Snitch bezeichnet eine Komponente, die eine Einteilung der Knoten in Racks vornimmt. Die Wahl des Snitch hat Auswirkung auf die Replikationsstrategie.

Token: Cassandra spezifischer Begriff. Ein Token ist ein durch die Partitionierungsfunktion errechneter Wert. Mithilfe von Tokens werden die Daten eines Cassandraclusters auf Knoten verteilt.

Transaktionskonzept: Verfahren, das den Mehrbenutzerbetrieb eines Datenbanksystems regelt.

XtraBackup: Software zur Durchführung von ACID-konsistenten Datensicherungen, ohne das System sperren zu müssen.