Konsistenz

Die etablierten relationalen Datenbankmanagementsysteme erledigen Updates in Form von **Transaktionen** nach den sogenannten ACID-Konsistenzeigenschaften, die gewährleisten, dass jede Transaktion als unteilbare Einheit die Datenbank in einen neuen, konsistenten Zustand überführt, auch dann, wenn von einer Transaktion mehrere Reihen und unterschiedliche Tabellen betroffen sind. Sie sind also atomare Operationen, die es möglich machen, den Datenbestand auf eine Weise zu manipulieren, sodass ein Nutzer zu jeder Zeit eine konsistente Datenbank hat, also Transaktionen/Updates erst dann sichtbar werden, wenn sie vollständig und korrekt ausgeführt wurden. Eine Datenbank wird somit von einem konsistenten Zustand in den nächsten überführt, ohne dass ein Nutzer zwischenzeitlich „partielle“ Updates auf den Daten zu befürchten hätte.

Das **CAP-Theorem** besagt, dass ein verteiltes System aus den Eigenschaften Consistency (Konsistenz), Availability (Verfügbarkeit) und Partition tolerance (Ausfalltoleranz) immer nur zwei Attribute erfüllen kann. Daher verzichten neuere NoSQL-Datenbanken auf **ACID**-Transaktionen und folgen dem aufgelockerten **BASE**-Modell, das Abstriche bei der strikten Konsistenz zugunsten der Verfügbarkeit macht, um dafür skalierbar zu sein.

ACID

ACID ist ein Akronym und steht für die vier Eigenschaften, die eine Transaktion in einem RDBMS (auch in jüngeren NewSQL-Systemen) zu erfüllen hat, um einen konsistenten Datenbestand zu bewahren:

- **Atomicity** (Atomizität): Eine Transaktion ist *atomar*, also eine unteilbare Einheit, die ohne Kompromisse entweder komplett ausgeführt wird oder überhaupt nicht. Alles andere würde die Daten inkonsistent machen.
- **Consistency** (Konsistenz): Eine Transaktion muss eine Datenbank von einem *konsistenten* Zustand in einen neuen *konsistenten* Zustand überführen. Entweder ist eine Transaktion erfolgreich und macht die Updates mit einem COMMIT sichtbar oder sie schlägt fehl und nimmt mit einem ROLLBACK alle bis dahin erfolgten Änderungen zurück. Zu keinem Zeitpunkt darf eine Transaktion inkonsistente Daten einlesen oder ausgeben. Dabei meint Konsistenz (und Integrität) die Vollständigkeit und Widerspruchsfreiheit der Daten.
- **Isolation**: Eine Transaktion läuft *isoliert* von anderen Transaktionen ab und wird nicht durch sie beeinflusst.
- **Durability** (Dauerhaftigkeit/Persistenz): Eine erfolgreiche Transaktion führt zu einem Ergebnis, das *dauerhaft* (auf einem Festspeicher) in einer Datenbank gespeichert wird.

BASE

BASE ist ebenfalls ein Akronym und stellt die gelockerten Konsistenzeigenschaften in NoSQL-Systemen dar, die den strengen ACID-Anforderungen der RDBMS entgegenstehen.

Mit Ausnahme der Graphdatenbanken unterstützen die meisten NoSQL-Datenbanken keine

Transaktionen. Grund dafür ist, dass es zu kompliziert wäre, volle ACID-Eigenschaften in einem Cluster zu unterstützen. Ein verteilter Datenbestand würde zu viel Kommunikationsoverhead bedeuten, der letztlich ihre Stärke, die Geschwindigkeit beim Umgang mit enormen Datenvolumen, zunichtemachen würde. Dafür verwalten diese Systeme Datensätze in Form von Aggregaten, innerhalb derer Updates atomar ausgeführt werden können, jedoch keine „Transaktionen“, die mehrere Aggregate umfassen (vgl. Sadalage/Fowler 2012: S. 19f).

BASE steht für **B**asically **A**vailable, **S**oft State, **E**ventual Consistency:

- **Basically Available**: Systeme, die auf das BASE-Modell setzen, versprechen, hochverfügbar zu sein und zu jeder Zeit auf Anfragen reagieren zu können. Dafür machen sie Abstriche bei der Konsistenz (siehe dazu auch CAP-Theorem). Das bedeutet, dass ein System immer auf eine Anfrage reagiert. Die Antwort kann aber auch ein Fehler sein oder einen inkonsistenten Datenstand liefern (Roe 2012).
- **Soft State**: Der Zustand, in dem noch nicht alle Änderungen an das gesamte System propagiert wurden, aber trotzdem auf Anfragen reagieren kann, nennt man auch *Soft State*.
- **Eventual Consistency**: Im Gegensatz zum strengen ACID-Modell wird in BASE die Konsistenz als eine Art „Übergang“ betrachtet, sodass nach einem Update nicht sofort alle Partitionen aktualisiert werden, sondern erst nach einer gewissen Zeit. Das System wird also „**letztendlich konsistent**“ (*eventually consistent*), aber bis dahin in einem inkonsistenten Übergangs-Zustand (*soft state*) sein.

Ein Beispiel für eine Situation, in der keine unbedingte Konsistenz nötig ist, sind Social-Media-Seiten. Hier ist es nicht wirklich kritisch, wenn ein Status-Update nicht sofort für alle sichtbar ist oder ob eine Freundschaftsanfrage sofort als bestätigt markiert wird oder erst wenige Sekunden später.

Dabei können einige Konsistenzmodelle gegeben sein (vgl. Vogels 2008):

- **Read-Your-Writes Consistency**: Hat ein Prozess erst einmal einen Wert geändert, wird er niemals einen älteren Wert sehen.
- **Monotonic Read Consistency**: Wenn ein Prozess einmal auf die Daten eines Objektes zugegriffen wird, wird garantiert, dass ihm bei allen folgenden Zugriffen keine älteren Daten geliefert werden.
- **Monotonic Write Consistency**: Wenn ein Client einen Wert in einem Objekt geändert hat und ihn danach nochmal ändert, wird garantiert, dass die letzte Änderung an die Kopien propagiert wird.

CAP-Theorem

Das **CAP-Theore**m wurde von Eric Brewer im Jahre 2000 vorgestellt (daher manchmal auch **Brewers Theorem** genannt) und 2002 von Seth Gilbert und Nancy Lynch bewiesen (Gilbert/Lynch 2002).

CAP steht dabei für die Eigenschaften:

Consistency (Konsistenz)

Nach einem schreibenden Zugriff haben alle verteilten Repliken auf sämtlichen Knoten immer denselben, aktuellen Datenzustand, so als ob dieser Zugriff eine atomare Operation auf einer Ein-

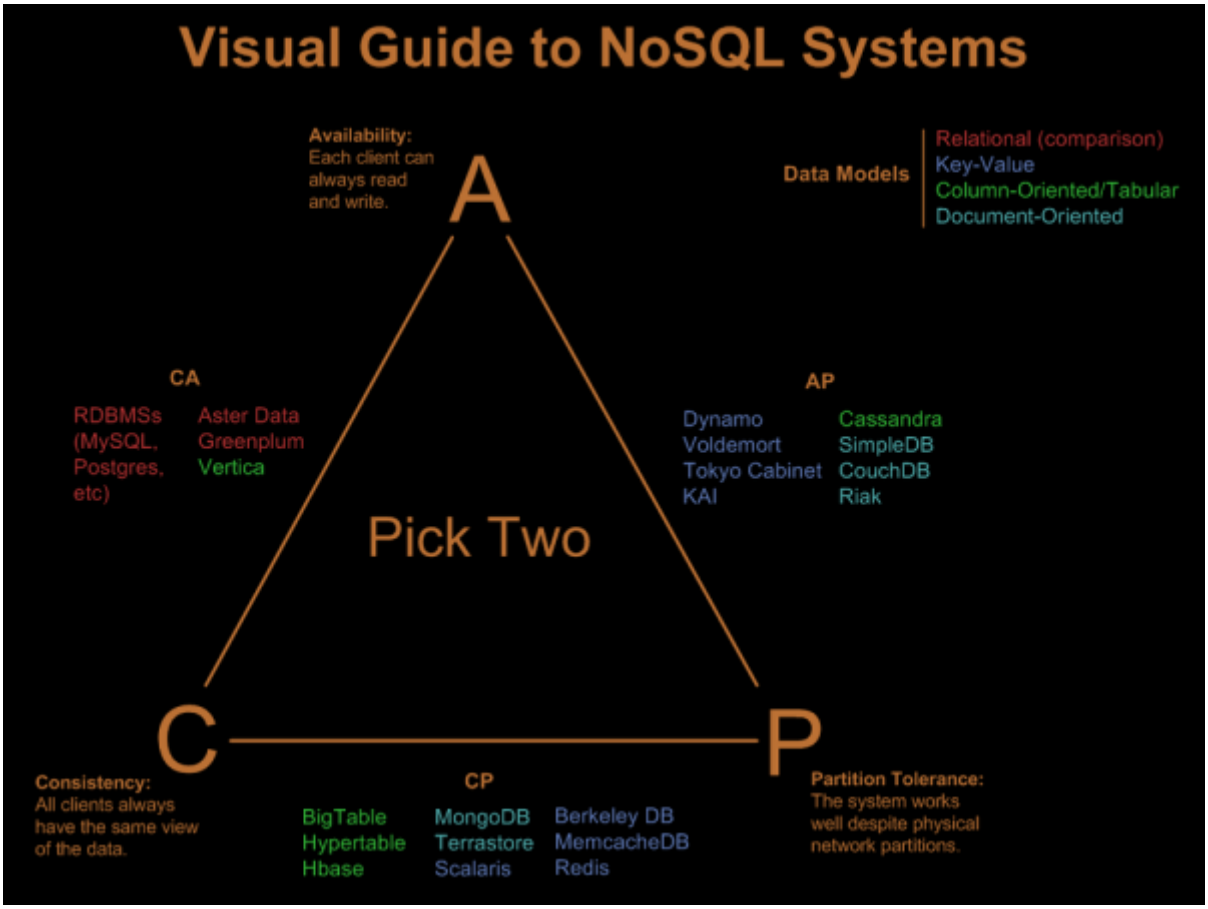
Server-Instanz wäre (Gilbert/Lynch 2002: S. 3). Wird direkt nach einem solchen Zugriff der Datenbestand abgefragt, liefert er entweder die aktualisierten Daten oder aber einen neuen Zustand, sollte in der Zwischenzeit ein erneutes Update erfolgt sein. Zu keiner Zeit aber einen älteren Datenstand.

Availability (Verfügbarkeit)

Das System muss zu jeder Zeit voll verfügbar sein, um möglichst schnell und effizient Anfragen bedienen zu können. Auf jede Anfrage an einen intakten Knoten muss auch eine Antwort erfolgen (Gilbert/Lynch 2002: S. 3). Gerade bei Internetanwendungen spielt die Reaktionszeit eine wichtige Rolle. Längere Wartezeiten (schon ab einer Sekunde) können für (potentielle) Kunden schon Grund sein, einen Webshop zu verlassen oder zur Konkurrenz zu wechseln (Borland 2013). Besonders kritisch wird das, sollte nach einer Zahlungsbestätigung eine Reaktion des Systems ausbleiben.

Partition Tolerance (Ausfallstoleranz)

Auch wenn Knoten oder Kommunikationsverbindungen ausfallen sollten (geplant oder ungeplant) und das Cluster so in Partitionen teilt, muss das Gesamtsystem weiterhin lauffähig und einsatzbereit sein und kann im Notfall Aufgaben umleiten. Um dies zu ermöglichen, werden die Daten im Cluster repliziert und redundant auf unterschiedlichen Knoten gehalten. „Außer einem totalen Netzwerkausfall darf keinerlei Fehler dazu führen, dass ein System unvorschriftsmäßig reagiert“ (übersetzt nach Gilbert/Lynch 2002: S. 4).



(Grafik-Quelle: Lee 2011)

Das Theorem besagt, dass in einem verteilten System mit Replikationen nur ***zwei*** der drei CAP-

Eigenschaften zu erreichen sind (vgl. Brewer 2012). Das bedeutet, ein System erfüllt entweder die Eigenschaften CA, AP oder CP.

Ein klassisches RDBMS setzt mit seinen ACID-Transaktionen vor allem auf allem auf Konsistenz, es ist ein CA-System. Ist es ein Ein-Server-System, so ist es verfügbar und konsistent, aber nicht ausfalltolerant. Ein einzelner Server kann nicht in Partitionen zerfallen und ist entweder verfügbar oder fällt ganz aus. Skaliert man dieses System mit Replikationen, so leidet die Verfügbarkeit darunter, da es länger dauert, bis alle Daten aktualisiert wurden. Dies wird kritischer je größer ein solches Cluster wird.

NoSQL-Systeme setzen dagegen eher ständige Verfügbarkeit, für die auch eine Ausfalltoleranz im Cluster wichtig ist. Sie sind also eher AP-Systeme, die auf Transaktionen verzichten und auf das weniger strenge BASE-Konistenzmodell bauen.

CP-Systeme sind besonders im Finanzwesen von Interesse, da hier bei Geldtransfers die Konsistenz sehr wichtig ist. Ein Betrag, der bspw. an einem Geldautomaten abgehoben/eingezahlt wurde, muss beim Kundenkonto auch abgebucht/gutgeschrieben werden, da ansonsten der betreibenden Bank und/oder dem Kunden Geld verloren geht. Dafür müssen auch Netzwerkfehler oder Knotenausfälle kompensiert werden können. Im Zweifelsfall ist der Service jedoch nicht verfügbar.

From:
https://wi-wiki.de/ - **Wirtschaftsinformatik Wiki - Kewee**

Permanent link:
https://wi-wiki.de/doku.php?id=bigdata:konsistenz

Last update: **2015/10/05 21:27**