Diplomarbeit
Transaktions- und Konsistenzmodelle in Datenbanken

Daniel Marby
Matrikelnummer: 4046509
Fachhochschule Anhalt
Fakultät Informatik und Sprachen

3. März 2016

Erklärung

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema
Transaktions- und Konsistenzmodelle in Datenbanken
selbständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbständiger Arbeit gilt auch für Zeichnungen, Skizzen und graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiaterkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungswecke gespeichert wird.

Halle, den

Vorwort

Die heute verwendeten Datenbanken sind das Ergebnis jahrzehntelanger Forschung und Entwicklung. In dieser Zeit wurden sowohl Konzepte entwickelt, um Datenbanken zu beschreiben, als auch Mechanismen, um diese Konzepte umzusetzen.

Am wichtigsten dabei ist das Konzept der Serialisierung, also die Idee, dass es für alle Transaktionen, die in einer Datenbank durchgeführt werden, mindestens eine Abfolge gibt, in der diese ausgeführt werden können.

Bei der Erstellung dieses Textes habe ich anfangs versucht, eine Serialisierung des Themas Datenbanken vorzunehmen. Das Thema enthält jedoch zyklische Abhängigkeiten, so dass es mir nicht möglich ist, das Thema in einer linearen Abfolge zu präsentieren.

Daher habe ich mich entschieden, einen Top-Down-Ansatz zu wählen. Wenn man die Kapitel in aufsteigender Reihenfolge liest werden daher zuerst die allgemeinsten Konzepte vorgestellt, die folgenden Kapitel enthalten dann Informationen zu einzelnen Aspekten dieser Konzepte.

Dieser Ansatz hat den Vorteil, dass man beim Durchlesen jedes Konzept in den allgemeinen Zusammenhang einordnen kann.

Der Nachteil besteht jedoch darin, dass gerade Anfangs viele Konzepte lediglich als Black-Box genannt werden können.

Obwohl im Text auch Konzepte wie ACID und Locking genannt und erklärt werden, wird beim Leser ein gewisses Grundwissen des Themas bereits vorausgesetzt. Dazu gehören insbesondere generelle Vorkenntnisse über die Funktionsweise der heute hauptsächlich eingesetzten relationalen Datenbanken.

Inhaltsverzeichnis

1 Datenbanken und CAP

Die Entwicklung und Nutzung von Datenbanken seit den 1960er Jahren führte zu der Frage, welche Eigenschaften ein Datenbankmanagementsystem (DBMS) haben sollte. Anfang der 1980er Jahre wurden diese mit ACID (atomar, konsistent, isoliert und dauerhaft) formalisiert. Dieser Ansatz schien die Universallösung zu sein, die es Anwendern[1] ermöglichte, sich um die Verarbeitung von Daten zu kümmern, ohne Aufwand oder Code für die Speicherung und Konsistenzsicherung aufwenden zu müssen. Ebenso war die Entwicklung nebenläufiger Prozesse (Multithreading, Multiprocessing) möglich, ohne dabei Schwierigkeiten wie Locking und Serialisierung überhaupt betrachten zu müssen.

Datenbanken wurden dabei ursprünglich für die damals gebräuchlichen Einzelmaschinensysteme entworfen. Das bedeutet, dass alle Programm- und Speicherkomponenten in einem Rechner zusammengefasst waren und die Kommunikation zwischen einzelnen Systemkomponenten ohne nennenswerte Verzögerungen möglich war.

Mit der immer größer werdenden Nachfrage nach immer leistungsfähigeren Datenbanken und wachsenden Datenbeständen entstand trotz steigender Prozessorleistung die Notwendigkeit, Datenbanken als Mehrmaschinensysteme zu realisieren. Dieser Übergang dazu erhöhte zwar die Leistungsfähigkeit der Datenbanken, brachte jedoch auch die Schwierigkeit mit sich, dass die Systeme aufgrund der durch ACID definierten Forderungen synchronisiert werden müssen. Gleichzeitig entstanden Herausforderungen dadurch, dass Kommunikationsverbindungen zwischen Rechnern häufig erheblich langsamer und fehleranfälliger sind als die in einer einzelnen Maschine.

Der spätere Übergang zu dezentralen Datenbanken, bei denen die einzelnen Rechner nicht mehr im selben LAN angeordnet sind, führt zu erheblich größeren Latenzen. Weitere Schwierigkeiten entstanden, da die Kommunikationsverbindungen nicht nur eine geringere Bandbreite haben, sondern ihre Wartung und Störungsbeseitigung nicht mehr in der Hoheit des Datenbankbetreibers liegt.

[1] sofern nicht anders angegeben ist in dieser Arbeit mit Anwender immer der Entwickler von Datenbankanwendungen gemeint

1.1 CAP-Theorem und Auswirkungen

Die oben beschriebenen Entwicklungen führten zu der grundsätzlichen Frage, welche Anforderungen man an Datenbanken stellt und wie man diese sinnvoll realisieren kann.

Eric Brewer formulierte im Jahr 2000 die folgenden drei Anforderungen an verteilte Datenbanken [PODC]:

1. Konsistenz (consistency): Anfragen werden atomar bearbeitet, der Zustand der Datenbank bleibt gültig[2].

2. Verfügbarkeit (availability): Der Dienst ist mindestens so verfügbar, wie das zu Grunde liegende Netzwerk. Das bedeutet: So lange das Netzwerk verfügbar ist, steht der Dienst auch zur Verfügung. Im Speziellen sollte jede Anfrage auch eine Antwort erhalten.

3. Partitionstoleranz[3] (partition-tolerance): Anfragen werden auch dann korrekt beantwortet, wenn Nachrichten zwischen den einzelnen Knoten der Datenbank verloren gehen[4].

Im selben Vortrag stellte Brewer die Hypothese auf, dass diese Bedingungen nicht gleichzeitig erfüllt werden können. Der Beweis dieser Aussage folgte im Jahr 2002 [GiLy].

Das CAP-Theorem ist dabei so zu verstehen, dass diese Eigenschaften nicht *zum selben Zeitpunkt* erfüllt werden können. So lange in verteilten Datenbanken die Kommunikation zwischen den einzelnen Knoten gegeben ist, kann auch eine verteilte Datenbank konsistent und verfügbar sein. Lediglich in während eines Ausfalls der Kommunikation muss eine Entscheidung getroffen werden, ob Konsistenz *oder* Verfügbarkeit garantiert werden soll.

Damit lassen sich Datenbanken grundsätzlich in drei Typen unterteilen:

CA-Systeme: Diese Systeme verzichten auf Partitionstoleranz, stellen jedoch Konsistenz und Verfügbarkeit sicher. Dazu zählen alle Datenbanken, die auf Einzelmaschinensystemen arbeiten. Im weiteren Sinn kann man darunter auch verteilte Datenbanken in LAN verstehen, sofern ein Ausfall der Kommunikation mit an Sicherheit grenzender Wahrscheinlichkeit ausgeschlossen werden kann. Dies sind häufig lokale ACID-Datenbanken

CP-Systeme: Diese Systeme geben im Fall einer Partitionierung die Verfügbarkeit auf, um Konsistenz sicherzustellen. Alle verteilten ACID-Datenbanken fallen in diese Kategorie.

[2] Diese Forderung unterscheidet sich von dem durch ACID definierten Konsistenzbegriff

[3] Partitionierung beschreibt einen einen Ausfall von Kommunikationsverbindungen derart, dass es mindestens zwei Rechner gibt, die keine Daten mehr austauschen können

[4] Insbesondere dürfen beliebig viele Nachrichten über einen beliebig langen Zeitraum verloren gehen

AP-Systeme: Diese Datenbanken verzichten auf Konsistenz, stellen jedoch Verfügbarkeit sicher. Dabei können im Fall einer Partitionierung sowohl veraltete Daten zurückgegeben werden, als auch Daten, bei denen nur ein Teil der inzwischen erfolgten Änderungen sichtbar ist. Das entsprechende Konzept dazu ist BASE.

Abbildung 1.1 stellt CAP als Dreieck dar. ACID-Datenbanken sind dabei auf den Kanten am Punkt C (Konsistenz) angesiedelt.

1.2 Beispiel: Domain Name Service

Konsistenz wird generell als Maß für Zuverlässigkeit betrachtet. Das führt zu der Schlussfolgerung, dass Datenbestände zwingend zu jeder Zeit konsistent sein müssen. Eine der Optionen, die CAP aufzeigt, sind jedoch AP-Systeme, also Systeme, die Konsistenz geringer gewichten als Verfügbarkeit.

Es stellt sich die Frage, ob und in welchen Fällen es sinnvoll sein kann, Einbußen bei der Konsistenz in Kauf zu nehmen. Dazu ein Beispiel:

Eine der weltweit größten verteilten Datenbanken stellt der Domain Name Service dar. Das Internetprotokoll nutzt numerischen Adressen einer Adresslänge von 32 Bit[5]. Leichter zu merken sind jedoch Adressen in Textform. Die Aufgabe von DNS ist es, Adressen in Textform ihre nummerische Äquivalente zuzuordnen. Das Internet, so wie es sich heute dem Anwender präsentiert, wäre ohne diesen Dienst nicht denkbar.

Um zum Beispiel die IP Adresse für "www.marby.org" zu bestimmen, müssen eine Reihe von Anfragen an unterschiedliche DNS-Server versendet werden:

1. Anfrage bei einem Root-Server, welche Nameserver die TLD[6] .org bedienen

[5]Das trifft auf IPv4 zu. Momentan wird ein hybrides Adressmodell aus IPv4 und IPv6 genutzt
[6]Top-Level-Domain

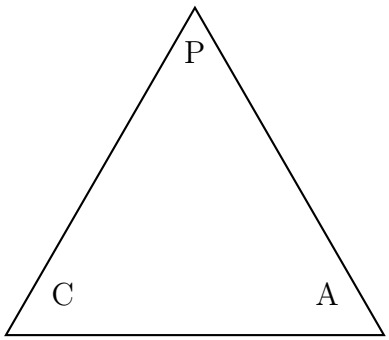

Abbildung 1.1: Das CAP-Dreieck

```
;; QUESTION SECTION:
;www.marby.org.                 IN      A

;; ANSWER SECTION:
www.marby.org.          21600   IN      A       62.108.44.35

;; AUTHORITY SECTION:
marby.org.              86400   IN      NS      dns.marby.org.
marby.org.              86400   IN      NS      dns2.marby.org.

;; ADDITIONAL SECTION:
dns.marby.org.          86400   IN      A       62.108.44.35
dns2.marby.org.         86400   IN      A       62.108.44.35
```

Abbildung 1.2: Resultat für "dig www.marby.org"

2. Anfrage bei einem dieser Server, welche Nameserver die SLD[7] marby.org bedienen
3. Anfrage bei einem dieser Server (im Beispiel 62.108.44.35), welche IP Adresse www.marby.org hat

Abbildung 1.2 zeigt das Ergebnis der Anfrage beim zuständigen Nameserver dns.marby.org.

Diese Vorgehensweise würde nicht nur zu langen Wartezeiten für die Namensauflösung führen, sondern auch erhebliche Lasten auf den Root-Servern und den Nameservern für die TLD verursachen. Um hier die Last für die Server zu senken, dürfen Einträge für eine bestimmte Zeit im Cache gehalten werden. Die entsprechende Zeitspanne wird dabei in der zweiten Spalte des Ergebnisses übermittelt. Nach einer Anfrage beim zuständigen Nameserver wird das Ergebnis auf dem Zielrechner (und zusätzlich ggf. auf Proxy-Servern, über die die Anfrage lief) gespeichert. Liegt bei einer Anfrage an einen Proxy diesem das Ergebnis bereits vor, wird keine Anfrage mehr an den zuständigen Nameserver gesendet. Der Proxy antwortet sofort mit dem vorhanden Ergebnis, reduziert dabei jedoch die Gültigkeitsdauer entsprechend.

Im oben genannten Beispiel dürfen die Einträge für die Nameserver 86400 Sekunden (24 Stunden) im Cache gehalten und als aktuell angesehen werden. Die IP Adresse für www.marby.org darf hingegen nur 21600 Sekunden (6 Stunden) im Cache gehalten werden.

Das DNS bietet hier eine sehr flexible Möglichkeit, zwischen hoher Aktualität bei hoher Serverlast und geringer Aktualität bei niedriger Serverlast einstellen zu können.

Der folgende Eintrag darf zum Beispiel nur für 30 Sekunden im Cache gehalten werden:

```
halle.ddns.marby.org.   30      IN      A       92.206.86.172
```

[7] Second-Level-Domain

Während www.marby.org eine statische IP Adresse hat, ist die IP Adresse von halle.ddns.marby.org dynamisch, d.h. die kann sich jederzeit ändern. Sobald dies geschieht, informiert der zum Anschluss gehörende Router den DNS-Server über die neue IP Adresse. Die geringe Zeitspanne, in der dieser Eintrag im Cache eines Nameservers gehalten werden darf, sorgt hier dafür, dass der entsprechende Eintrag möglichst aktuell ist.

Falls an DNS-Einträgen Änderungen vorgenommen werden sollen, ist es gängige Praxis, 24 Stunden vor der Änderung die Gültigkeitsdauer der betroffenen Einträge zu verringern. Damit wird sichergestellt, dass die Änderung mit geringer Verzögerung propagiert wird. Es ist jedoch auch möglich, den Dienst unter der alten IP-Adresse weiter zu betreiben oder auch einfach in Kauf zu nehmen, dass Nutzer eventuell keine Antwort erhalten.

DNS stellt somit ein praktisches Beispiel für eine verteilte Datenbank dar, bei der Abstriche bei der Konsistenz in Kauf genommen werden. Im Gegenzug werden damit sowohl Verfügbarkeit als auch Partitionstoleranz gewährleistet.

1.3 Ausblick auf die weiteren Kapitel

Heute sind Datenbanken zum einen fester Bestandteil nahezu aller Plattformen, zum anderen unterscheiden sie sich nicht nur in ihrer Architektur, sondern die für den Benutzer sichtbaren Unterschiede reichen bis hin zu Implementierungsdetails.

Die folgenden Kapitel sollen daher einige dieser Unterschiede näher beleuchten. Dabei sind auch Konzepte aufgeführt, die bereits seit Jahren zum Standard bei nahezu allen Implementierungen von Datenbanken gehören. Es erweist sich jedoch als sinnvoll, nicht nur neue Lösungen zu beschreiben, sondern diese den bekannten und erprobten Methoden gegenüberzustellen. In einigen Fällen dienten erkannte Schwächen bekannter Technikern als Ausgangspunkt für die Entwicklung ihrer Alternativen.

Die einzelnen Kapitel beleuchten dabei unterschiedliche Aspekte des Themas:

ACID vs. BASE stellt die beiden möglichen Grundentscheidungen Konsistenz (ACID) und Verfügbarkeit (BASE) gegenüber. Reale Datenbanken werden überlicherweise einem der beiden Typen zugeordnet, enthalten jedoch fast immer Aspekte beider Ansätze.

2-Phase-Locking vs. Multiversion-Concurrency-Control stellt die beiden am Häufigsten verwendeten Methoden zur Synchronisation nebenläufiger Prozesse gegenüber.

Zeitmodelle: klassisch vs. relativistisch soll zeigen, dass in asynchronen Systemen unser klassisches Verständnis von Vergangenheit, Gegenwart und Zukunft nicht mehr funktioniert und definiert diese Begriffe in einem kausalen Zusammenhang neu. Dies ist insbesondere wichtig, da BASE-Datenbanken ohne die bei ACID zwingende Synchronisation von Operationen arbeiten.

NoSQL Architekturen stellt danach eine Reihe von Alternativen zu dem von den meisten ACID Datenbanken bekannten relationalen Datenbankmodell vor.

verteilte Datenbanken stellt danach einige Aspekte der Verteilung von Datenbanken auf unterschiedliche Rechner vor. Diese Inhalten treffen zum großen Teil auf alle Datenbanken zu, unabhängig davon, ob es sich um ACID oder BASE Typen handelt.

Konsistenz und Konvergenz zeigt die häufigsten Isolations- und Konsistenzmodelle auf und stellt diese den in CA-Systemen verwendeten Konvergenzmethoden gegenüber.

2 ACID vs. BASE

Insbesondere die anfängliche Nutzung von Datenbanken zur elektronischen Buchhaltung bewirkte, dass die grundlegenden Eigenschaften von Datenbanken den Anforderungen an eine korrekte Buchhaltung entsprechen. Dies sind insbesondere die Forderung nach einer Vereinheitlichung der Datenspeicherung (zum Beispiel muss die Buchführung eines Unternehmens für Dritte nachvollziehbar sein) und die korrekte Abbildung von Verträgen[1].

Vor der Einführung von Datenbanken wurden Daten vorwiegend in anwendungsspezifischen Formaten gespeichert. Das hat zum einen den Nachteil, dass ein Wechsel der Software meist mit einer aufwändigen Datenkonvertierung einher ging. Daraus entstand der Wunsch, die Speicherung der Daten zu vereinheitlichen. Eine weitere Anforderung war die konfliktfreie Nutzung des selben Datenbestandes durch mehrere Benutzer. Die notwendige Logik dazu musste dabei anfangs in den entsprechenden Anwendungsprogrammen fehlerfrei implementiert sein, um eine Beschädigung des Datenbestandes zu vermeiden. Eine Verlagerung dieser Logik in das zur Speicherung verwendete System reduziert nicht nur die Anzahl der möglichen Fehlerquellen, sondern erlaubte auch eine schnellere und problemlosere Entwicklung von Software.

Verträge werden in Datenbanken durch Transaktionen abgebildet [JG01]. Ein Vertrag ist entweder geschlossen, oder er ist nicht geschlossen. Ein teilweise geschlossener Vertrag ist nicht definiert und darf nicht existieren. Sobald ein Vertrag jedoch geschlossen ist, kann er nicht annulliert werden[2] , seine Folgen können nur durch einen weiteren Vertrag (Aufhebungsvertrag) rückgängig gemacht werden.

Datenbanktransaktionen setzen dies um. Eine Transaktion ist entweder erfolgt, oder sie ist nicht erfolgt. Eine teilweise erfolgte Transaktion darf nicht existieren. Eine Transaktion kann zurückgerollt (annulliert) werden, sobald sie jedoch abgeschlossen ist, kann sie nicht rückgängig gemacht werden, sondern muss mittels einer weiteren aufgehoben werden. Dabei wird jedoch nur der Datenbestand auf seinen ursprünglichen Wert korrigiert, die erfolgten Änderungen sind im Log der Datenbank nachvollziehbar.

[1] rechtlich: zwei übereinstimmende Willenserklärungen

[2] Die einzige Ausnahme stellen Vertragshindernisse dar, die bei Abschluss des Vertrages (mindestens einer Seite) bekannt waren, da in diesem Fall der Vertrag nicht geschlossen werden dürfte.

2.1 ACID

Die eingangs genannten Anforderungen führten Ende der 1970er Jahre zur Formalisierung der heute unter der Abkürzung ACID bekannten Anforderungen an Datenbanksysteme. ACID steht hierbei für die folgenden Eigenschaften der Datenbanktransaktionen [WP01]:

1. atomar (Atomicity)
2. konsistent (Consistency[3])
3. isoliert (Isolation)
4. dauerhaft (Durability)

2.1.1 Atomicity (Abgeschlossenheit)

Eine Transaktion ist *atomar* (oder auch abgeschlossen), wenn sie zu jedem Zeitpunkt entweder vollständig oder nicht ausgeführt ist. Besteht eine Transaktion aus mehreren Operationen, müssen alle Operationen erfolgreich sein. Im Fall des Fehlschlags der n-ten Operation müssen alle vorherigen, erfolgreichen Operationen rückgängig gemacht werden.

Hierzu ein Beispiel: In einer Bank soll Geld von Konto x auf Konto y transferiert werden. Dazu müssen folgende Aktionen durchgeführt werden:

1. aktuellen Kontostand für Konto x lesen
2. neuen Kontostand für Konto x berechnen und speichern
3. aktuellen Kontostand für Konto y lesen
4. neuen Kontostand für Konto y berechnen und speichern

Weiterhin darf kein Abbruch oder Fehler zwischen den Schritten 2 und 3 erlaubt werden, da sonst nur Konto x belastet würde, ohne das eine Gutschrift auf Konto y erfolgt. Ebenso darf Schritt 2 nicht ausgeführt werden, falls in Schritt 4 der neue Kontostand für y nicht gespeichert werden kann.

[3]Hier: ACID-Konsistenz, siehe Abschnitt 2.1.2

2.1.2 Consistency (Konsistenzerhaltung)

Transaktionen auf Datenbanken gelten als *konsistent*[4], wenn folgende Bedingungen erfüllt sind:

- Eine Transaktion, die zum Zeitpunkt t startet, sieht alle Änderungen, die bis zu diesem Zeitpunkt abgeschlossen wurden.
- Die datenbankseitig hinterlegten Gültigkeitsbedingungen sind erfüllt, d.h. jede Transaktion führt zu einem gültigen Zustand, wenn die Datenbank zu Beginn in einem gültigen Zustand war. Der Datenbestand kann durch die Ausführung der Transaktion nicht ungültig werden.

2.1.3 Isolation (Abgrenzung)

Datenbanktransaktionen sind *isoliert*, wenn der durch die gleichzeitige Ausführung entstehende Zustand identisch dem Zustand ist, der bei serieller Ausführung entsteht.

Hierzu ein Beispiel: In einer Bank sollen 100 $ von Konto x auf Konto y (u_1) und 100 $ von Konto y auf Konto x (u_2) überwiesen werden. Der Kontostand vor der Transaktion betrage jeweils 100 $. Hierzu gibt es 3 Möglichkeiten:

u_1 vor u_2: Nach der Ausführung von u_1 lauten die Kontostände $x = 0\$$ und $y = 200\$$. Nach der Ausführung von u_2 lauten sie dann $x = 100\$$ und $y = 100\$$.

u_2 vor u_1: Nach der Ausführung von u_1 lauten die Kontostände $x = 200\$$ und $y = 0\$$. Nach der Ausführung von u_2 lauten sie dann $x = 100\$$ und $y = 100\$$.

u_1 und u_2 werden gleichzeitig ausgeführt: Dann ist folgende Ausführungsreihenfolge möglich (zeitlich nach Zugriffen geordnet):

1. u_1 liest $x = 100\$$
2. u_1 schreibt $x = 0\$$
3. u_1 liest $y = 100\$$
4. u_2 liest $y = 100\$$
5. u_1 schreibt $y = 200\$$
6. u_2 schreibt $y = 0\$$
7. u_2 liest $x = 0\$$
8. u_2 schreibt $x = 100\$$

[4]Konsistenz unter ACID ist ein wesentlich enger gefasster Begriff, als der unter CAP verwendete Begriff. Konsistenz im Sinne von CAP erfasst Bestandteile aller Komponenten von ACID. Im weiteren Verlauf des Textes wird der Begriff ausschließlich in der Definition gemäß CAP genutzt. Für Konsistenz in der unter ACID verwendeten Definition wird im folgenden der Begriff ACID-Konsistenz verwendet.

Die finalen Kontostände sind damit $x = 100\$$ und $y = 200\$$. Dieses Ergebnis ist mit der seriellen Ausführung nicht zu erreichen und damit unzulässig.

2.1.4 Durability (Dauerhaftigkeit)

Eine Transaktion ist *dauerhaft*, wenn die veranlassten Änderungen nach der Bestätigung der Ausführung unabhängig von externen Faktoren (z.B. Stromausfall, Absturz, Hardwareversagen) bestehen bleibt. Anders ausgedrückt bedeutet das, dass eine Transaktion, sobald sie vom DMBS bestätigt wurde, in jedem Fall Bestand haben wird.

2.2 BASE

Wie am Anfang des Kapitels erwähnt, basiert das ACID-Transaktionsmodell auf der Funktion von Verträgen. Das in Kapitel 1.2 genannte Beispiel DNS zeigt jedoch, dass die strengen Anforderungen eines Vertragsmodells nicht in allen Fällen erforderlich sind.

Tatsächlich lassen sich neben DNS viele Anwendungen nennen, in denen die Anforderungen an die Konsistenz zum Teil erheblich abgeschwächt werden können.

- Suchmaschinen: Verfügbarkeit ist hier eindeutig wichtiger als Aktualität (Konsistenz). Der Nutzer kann mit einem schnell verfügbaren und (im Bereich von Minuten) älteren Ergebnis mehr anfangen, als mit einem Ergebnis, das erst nach längerer Wartezeit erscheint.
- Webshops: Schnell verfügbare Artikellisten und -beschreibungen sind hier ebenfalls wichtiger als Aktualität. Aktuelle Daten werden erst beim Kauf (Vertragsabschluss) selbst benötigt.
- Foren: Auch hier ist es wichtiger, dem Nutzer schnell Daten zur Verfügung zu stellen. Dass ein neuer Eintrag eventuell erst wenige Minuten später erscheint, ist in nahezu allen Fällen hinnehmbar.
- Webseiten: Sowohl für den Nutzer als auch für den Betreiber ist es wichtiger, dass der Nutzer die Seite sieht, auch wenn es sich um eine (im Bereich von Minuten) ältere Fassung handelt. Ein nicht oder nur mit erheblichen Verzögerungen verfügbarer Dienst hinterlässt einen erheblich schlechteren Eindruck.

Grundsätzlich ist es bei den meisten Web-Anwendungen wichtiger, dem Nutzer Informationen zu präsentieren, als sicherzustellen, dass diese Informationen den aktuellsten Stand haben (besser fünf Minuten alte Informationen als keine).

Allgemein kann man schlussfolgern, dass in vielen Fällen Daten auf Anforderung schnell zur Verfügung gestellt werden müssen, dabei jedoch keine Aktualität (im Sinn von Konsistenz) der Daten gewährleistet werden muss.

Diese Überlegungen führen zur Formulierung von BASE: *B*asically *A*vailable, *S*oft state, *E*ventually consistent (grundsätzlich verfügbar, Soft State[5], letztendlich konsistent).

Bei BASE (dt.: Lauge) handelt es sich um ein Backronym, d.h. die Abkürzung wurde als Gegenstück zu ACID (dt.: Säure) festgelegt, erst danach wurde den einzelnen Buchstaben eine Bedeutung zugewiesen.

- grundsätzliche Verfügbarkeit (Basically Available): So lange ein Knoten der Datenbank erreichbar ist, wird jede Anfrage, die an diesen Knoten gestellt wird, innerhalb einer festgelegten Zeitspanne (beliebig, aber fest) bearbeitet.
- Soft State: Der von der Datenbank zurückgemeldete Zustand ist nicht garantiert. Für Lesezugriffe bedeutet das, dass das Ergebniss nicht notwendigerweise den aktuellen Systemzustand wiederspiegelt. Schreibzugriffe andererseits können trotz Bestätigung später im Rahmen einer Konfliktbereinigung verworfen werden.
- irgendwann konsistent (eventually consistent): Konsistenz ist kein Dauerzustand des Systems, sondern ein Zustand, der bei vorhandener Kommunikation zwischen den Knoten und ausreichend geringer Last irgendwann erreicht wird. Diese Konsistenzmodell stellt keine Anforderungen an die Zeitspanne, in der Konsistenz erreicht wird.

Damit sind die von BASE gestellten Anforderungen wesentlich flexibler, als die für ACID.

Mit dieser Flexibilisierung ist auch verbunden, dass BASE keine aus Kommandofolgen bestehenden Transaktionen, wie sie bei ACID Datendanken gebräuchlich sind, unterstützt. Lediglich die elementaren Zugriffe lesen, schreiben, ändern und löschen werden als atomare Kommandos garantiert.

Die am häufigsten verwendeten Methoden zur Konfliktbereinigung nutzen dabei Zeitstempel, Vektoruhren oder Versionsvektoren[6].

2.2.1 Eventual Consistency und Strong Eventual Consistency

Eventual Consistency ist lediglich die Garantie, dass, sofern hinreichend lang keine Aktualisierungen erfolgen, alle Knoten auf Anfrage den selben Wert zurückgeben werden. Eine Garantie, dass dies ein bestimmter geschriebener Wert ist (z.B. der zeitlich letzte) oder dass dieser Wert semantisch korrekt ist, besteht nicht.

Im Gegensatz dazu garantiert Strong Eventual Consistency, dass beliebige Knoten, sofern sie die selben Updates erhalten haben, im selben Zustand sein werden. Dies ist unabhängig von der Reihenfolge, in der diese Knoten die Updates erhalten haben.

[5] dt.: weicher (hier unsicherer) Zustand
[6] siehe Kapitel 4 auf Seite 19

Zusammen mit monotoner Versionierung[7] kann damit garantiert werden, dass keinesfalls neuere durch ältere Datenüberschrieben werden.

SEC wird dabei häufig mit konfliktfrei replizierbaren Datentypen[8] hergestellt, die es in vielen Fällen auch ermöglichen, die Semantik des Datums darzustellen.

Da die Sicherstellung von SEC gegenüber EC keine erheblich höheren Kosten (Rechenleistung, Bandbreitenbedarf) verursacht, stellen die meisten heute verfügbaren BASE Datenbanken SEC sicher.

[7] also ausschließlich aufsteigenden Versionsnummern
[8] siehe Kapitel 7.3 auf Seite 44

3 2-Phase-Locking vs. Multiversion Concurrency Control

Grundsätzlich wird bei allen Datenbanken Serialisierbarkeit gefordert. Das bedeutet, eine Ausführungsreihenfolge aller Transaktionen existiert, die den Anfangszustand der Datenbank in den neuen Zustand überführt. Hierbei ist unerheblich, ob diese Ausführungsreihenfolge eindeutig ist. Ebenso ist unerheblich, ob die Transaktionen tatsächlich in dieser Reihenfolge ausgeführt wurden, so lange der durch die tatsächliche Ausführung erzeugte Zustand auch durch eine Nacheinanderausführung der Transaktionen in geeigneter Reihenfolge erzeugt werden kann.

Weiterhin wird gefordert, dass Transaktionen atomar sind, d.h. dass unabhängig von der tatsächlichen Dauer der Transaktion ein Zeitpunkt angegeben werden kann, zu dem diese Transaktion in der oben genannten Reihenfolge ausgeführt wurde.

Die offensichtlich einfachste Realisierung hierfür besteht darin, einen einzelnen Transaktionmanager für die gesamte Datenbank zu benutzen, der jeweils nur eine Transaktion gleichzeitig bearbeitet. Genauso offentlich ist jedoch auch, dass diese Variante nicht skalierbar und anfällig für Fehler in den entsprechenden Anwendungsprogrammen ist[1].

Die sinnvollste Möglichkeit, das System skalierbar zu machen besteht daher darin, mit mehreren Transaktionsmanagern Transaktionen gleichzeitig zu bearbeiten. Dabei muss jedoch sichergestellt werden, dass auch weiterhin eine Serialisierbarkeit gegeben ist. Dazu müssen die einzelnen Transaktionen voneinander isoliert werden.

Für eine Transaktion sind grundsätzlich zwei Datenmengen interessant:

- die Menge der gelesenen Daten (R) und
- die Menge der geschriebenen, d.h. modifizierten Daten (W)

Hierbei ist üblicherweise $|R| > |W|$. Gleichzeitig ist R unter Umständen nicht offensichtlich, da nicht nur das ausgeführte Kommando, sondern auch die in der Datenbank abgelegten Gültigkeitsbedingungen (Contraints) berücksichtigt werden müssen. Ein Beispiel liefert folgendes Kommando:

[1] Bricht ein Programm ab, müsste der Transaktionsmanager die aktuelle Transaktion abbrechen. Dafür ist jedoch erforderlich, dass dieser über den Abbruch informiert wird. Meist geschieht dies erst mit erheblichen Verzögerungen durch Timeout.

```
update test set value=1 where id=1000;
```

Unterstellt man, dass es sich bei *id* um einen Primärschlüssel handelt, dann scheint diese Anweisung nur eine einzelne Zeile in der Tabelle *test* zu lesen und nur ein einzelnes Datum zu modifizieren.

Wenn es sich bei *value* jedoch um einen Fremdschlüssel handelt oder wenn *value* Teil eines durch ein Constraint abgedeckten Bereiches ist (z.B. könnte die Anzahl der entsprechenden Werte in der Tabelle limitiert sein), dann müssen zur Prüfung des Constraints aus dieser und/oder anderen Tabellen Daten gelesen werden. R kann dadurch sehr schnell sehr groß werden.

Eine andere Schwierigkeit ergibt sich, wenn das oben genannte Kommando erst spät in der Transaktion ausgeführt werden soll. In diesem Fall lassen sich die Mengen R und W erst bestimmen, wenn das Kommando der Datenbank bekannt gemacht wurde. Das bedeutet, dass erst bei einem Commit bekannt ist, welche Datenmengen für die Ausführung der Transaktion erforderlich sind. Die Implikationen werden weiter unten bei den aufgeführten Techniken beschrieben.

Man unterscheidet hier zwei Techniken: Locking, also das Sperren von benötigten Werten und Multi-Versioning, also die Möglichkeit mehrere unterschiedliche Versionen des selben Datenbestandes zu präsentieren.

Beides sind dabei Verfahren zur Synchronisierung von nebenläufigen Prozessen. Daher finden sie bei AP-Systemen keine Verwendung. In diesen Systemen werden statt dessen die in Kapitel 7 beschriebenen Konvergenzverfahren verwendet.

3.1 2-Phase-Locking

Die Grundidee eines Locks ist folgende: Wenn mehrere Prozesse auf eine gemeinsame Resource zurgreifen wollen (z.B. ein Datum in einer Datenbank), dann erhält nur ein Prozess Zugriff und im System wird vermerkt, welche Prozess Zugriff hat. Alle anderen

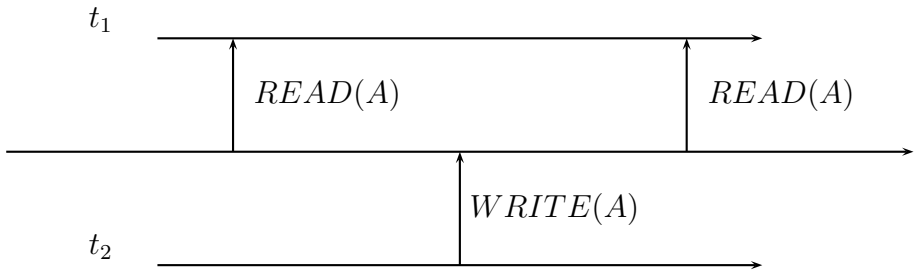

Abbildung 3.1: Beispiel gleichzeitiger Transaktionen

Prozesse müssen warten, bis diese Sperre wieder entfernt und die Resource damit wieder freigegeben wird. Um hier möglichst viele Transaktionen gleichzeitig ablaufen lassen zu können, müssen daher Locks möglichst schnell wieder entfernt werden. Das kann jedoch zu Problemen führen. Abbildung 3.1 illustriert das: Wird nach der ersten Leseoperation in t_1 das Lock entfernt, dann kann t_2 den Wert für A ändern. Eine weitere Leseoperation in t_1 liefert dann den von t_2 geschriebenen Wert. Damit ist die geforderte Isolation der beiden Transaktionen nicht mehr gegeben.

Um diese Situationen zu vermeiden, wird in Datenbanken 2-Phase-Locking eingesetzt. In diesem Modell bestehen Lock-Operationen für Transaktionen aus zwei Phasen:

1. Wachstumsphase (Expanding Phase): In dieser Phase werden Locks gesetzt
2. Schrumpfungsphase (Shrinking Phase): In dieser Phase werden Locks wieder freigegeben.

Damit werden in einer Transaktion grundsätzlich keine Locks mehr gesetzt, sobald das erste Lock wieder freigegeben wurde. Das Anfordern von Locks erfolgt damit während der Ausführung der einzelnen Kommandos der Transaktion, die Freigabe erfolgt mit Beginn der Schreibphase am Ende des Commits[2].

Locks stellen eine Form der Synchronisierung dar. Damit ergeben sich zwei Performanceprobleme:

1. Locks behindern die parallele Bearbeitung (allerdings ist das auch ihre Funktion).
2. Die Synchronisation mehrerer Maschinen (gerade in WAN Umgebungen) nimmt erheblich Zeit in Anspruch.

3.1.1 Arten von Locks

Da es kein Hinderniss darstellt, wenn mehrere Transaktionen gleichzeitig den selben Datensatz lesen, unterscheidet man zwischen zwei Arten von Locks:

Shared Locks (auch Read-Lock): Diese Locks werden zum Lesen eines Datensatzes gesetzt und können bestätigt werden, so lange kein Write-Lock gesetzt ist (insbesondere können sie auch gesetzt werden, wenn bereits ein Read-Lock auf dem Datensatz besteht).

Exclusive Locks (auch Write-Lock): Diese Locks werden zum Schreiben eines Datensatzes gesetzt. Ein Write-Lock kann nur dann gesetzt werden, wenn kein anderes Lock (egal, ob Read oder Write) vorliegt.

Grundsätzlich können auch ausschließlich exclusive Locks implementiert werden. Dann sind jedoch keine simultanen Leseoperationen mehr möglich. Die Implementierung von zwei unterschiedlichen Lock-Arten wird verständlich, wenn man berücksichtigt, dass die Lesemenge einer Transaktion häufig wesentlich größer ist, als die Schreibmenge.

[2] genauer wird diese Variante als striktes 2-Phase-Locking (TPL) bezeichnet

3.1.2 Granularität

Je nach Implementierung und Konfigurationen werden Locks auf unterschiedlichen Ebenen gesetzt:

- Datenbank: Hierbei wird die gesamte Datenbank vom Lock erfasst, unabhängig davon, ob ggf. nur ein einzelnes Datum gelesen oder geschrieben werden soll.
- Tabelle: Hierbei wird ein Lock pro betroffener Tabelle gesetzt
- Zeile: Dabei werden nur die von der Transaktion betroffenen Zeilen in den entsprechenden Tabellen mit Locks versehen
- Zelle: Es wird nur das angefragte Datum mit einem Lock versehen.

Je kleiner dabei die von den Locks betroffene Einheit ist, desto genauer kann die Menge der betroffenen Daten eingegrenzt werden und desto mehr Transaktionen können parallel bearbeitet werden.

Auf der anderen Seite werden Locks normalerweise nicht im Block, sondern nacheinander gesetzt. Je feiner dabei die Unterteilung gewählt wird, desto mehr Locks müssen gesetzt werden. Dadurch erhöht sich nicht nur der Verwaltungsaufwand, sondern auch die zum Setzen der Locks erforderliche Zeit.

Es empfiehlt sich daher, dies zu berücksichtigen und (falls möglich) die Granularität je nach Anfrage anzupassen oder das Schema so zu entwerfen, dass eine höhere Granularität verwendet werden kann[3].

3.2 Multiversion Concurrency Control

Statt die Zugriffe selbst durch Locks zu synchronisieren, kann man ebenso die Datenbank duplizieren (also eine Arbeitskopie der Datenbank erzeugen) und jede Transaktion auf einem eigenen Duplikat arbeiten lassen. Nach Abschluss der Transaktion muss diese Version dann wieder in den Master integriert werden. Damit existiert zu jeder Transaktion eine eigene Version der Datenbank.

Offensichtlich ist es hier nicht sinnvoll, den Datenbestand tatsächlich zu kopieren. Wichtig ist nur, auf eine geeignete Weise den Zustand angeben zu können, den die Datenbank zum Zeitpunkt des Starts der Transaktion hatte. Dazu wird seitens des DBMS eine Versionskontrolle bereitgestellt, mit der auch ältere Daten (mindestens alle Daten, die von laufenden Transaktionen noch gelesen werden könnten) rekonstruierbar sind. In der

[3] 1:1-Beziehungen werden in relationalen Datenbanken normalerweise mit zusätzlichen Spalten in der selben Tabelle abgebildet. Falls im speziellen Fall unterschiedliche Anwendungen disjunkte Spaltengruppen ändern, kann es sich anbieten, hier mit mehreren Tabellen zu arbeiten, die über Fremdschlüssel verknüpft werden und als Zeilen als Lockgranularität zu wählen.

Regel werden dazu entweder partielle Log-Replays verwendet (falls die Aktualisierungen noch nicht in den Master integriert wurden). Alternativ kann auch mit verzögerter Garbage-Collection gearbeitet werden, falls mit einer Read-Modify-Write-Strategie gearbeitet wird.

Die besondere Stärke von Multiversion Concurrency Control (MVCC) liegt dabei darin, dass Lese- und Schreibzugriffe auch dann korrekt bearbeitet werden können, wenn sie den Storage-Manager (SM) Out-of-Order erreichen.

Die Regeln sind dabei folgende:

1. Für jedes Datum existiert eine Menge von Zeitstempeln für Leseoperationen und eine Menge von Paaren $(t, wert)$ an Versionen.
2. Bei Anforderungen einer Leseoperation für den Zeitpunkt t_0 wird der Wert zurückgeliefert, dessen Zeitstempel am größten, jedoch kleiner als t_0 ist. Eine Leseoperation wird dabei keinesfalls abgelehnt. Der Zeitstempel t_0 wird der Menge der Lesezeitstempel hinzugefügt.
3. Bei Anforderung einer Schreiboperation für den Zeitpunkt t_1 wird geprüft, ob bereits eine Leseoperation für einen Zeitpunkt nach t_1 und vor dem nächsten Schreibzugriff mit größerem Zeitstempel bestätigt wurde. Falls ja, wird der Schreibzugriff abgewiesen. Anderenfalls wird der Schreibzugriff bestätigt und eine neue Version $(t_1, wert_1)$ erzeugt.

Abbildung 3.2 verdeutlicht das:

1. Transaktion t_1 wird zum Zeitpunkt $t = 9$ gestartet und liest a. Zurückgegeben wird $a(6)$ und der Lesezugriff wird vermerkt.
2. Danach schreibt die vorher gestartete Transaktion t_2 den Wert a für den Zeitpunkt $t = 5$. Dieser Zugriff wird bestätigt und eine neue Version $(a, 5)$ erzeugt.
3. Transaktion t_3 versucht a für $t = 7$ zu schreiben. Da für $t = 9$ bereits $a(6)$ zurückgegeben wurde, wird dieser Schreibzugriff abgewiesen. Bei einer Genehmigung hätte der Lesezugriff für t_1 $a(7)$ liefern müssen.

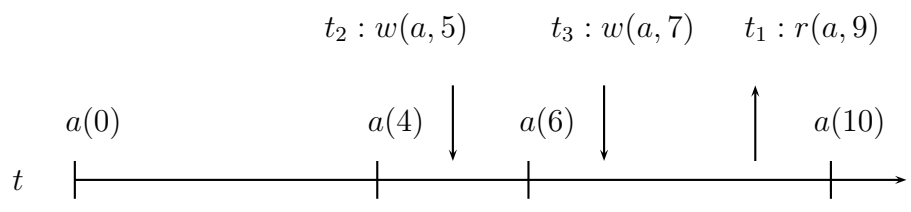

Abbildung 3.2: Beispiel MVCC

3.2.1 Spezialfall Snapshot Isolation

In vielen ACID Datenbanken ist inzwischen Snapshot-Isolation (SI) als Isolationslevel verfügbar. Dabei handelt es sich um einen Spezialfall von MVCC mit folgenden Regeln:

1. Alle Leseoperationen werden mit dem Zeitstempel des Transaktionsstarts durchgeführt.
2. Alle zu schreibenden Werte werden beim Commit darauf überprüft, ob seit dem Transaktionsstart bereits eine neue Version gespeichert wurde. Falls ja, wird die Transaktion abgebrochen. Andernfalls wird das Commit bestätigt.

Obwohl diese Isolationsstufe keinen der Fehler der Stufe Repeatable-Read aufweist und nach ANSI-SQL somit als serialisierbar gilt, ist sie das nicht. Tatsächlich gibt es einen Fehlerfall (write-skew), in dem die gleichzeitige Ausführung zweier Transaktionen, die für sich geprüft korrekt sind, zu einem inkonsistenten Zustand der Datenbank führen kann. Dieser Fall ist durch Planung im Anwendungsprogramm zu vermeiden.

Abbildung 3.3 illustriert diese Situation. Da sich die Schreibmengen von t_1 und t_2 nicht überlappen, werden beide Transaktionen ausgeführt. Allerdings gibt es hier keine mögliche Serialisierung.

1. $t_1 \rightarrow t_2$ ist nicht möglich, da t_2 dann den von t_1 modifizierten Wert hätte lesen müssen.
2. $t_2 \rightarrow t_1$ ist nicht möglich, da t_1 dann den von t_2 modifizierten Wert hätte lesen müssen.

Wenn man weiterhin annimmt, dass als Konsistenzkriterium $A \vee B$ gilt, vor Beginn der Transaktionen A und B wahr sind und t_1 und t_2 versuchen B und A auf den Wert falsch zu ändern, dann wird jede der beiden Transaktionen bestätigt, da innerhalb der Transaktion das Konsistenzkriterium erfüllt ist. Die gleichzeitige Ausführung von t_1 und t_2 führt jedoch zur Verletzung des Konsistenzkriteriums.

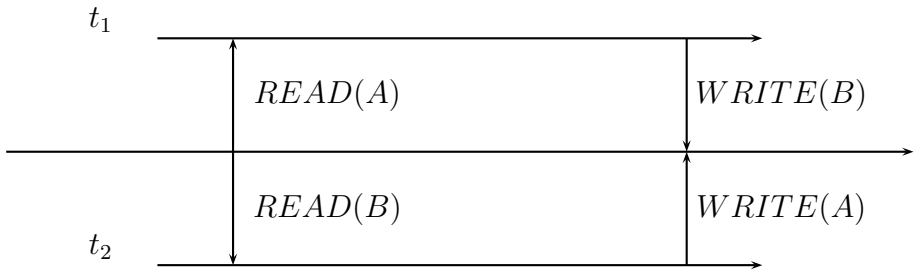

Abbildung 3.3: Write-Skew bei SI

4 Zeitmodelle

Insbesondere ACID-Datenbanken verwenden das aus dem Alltag bekannte klassiche (newtonsche) Zeitmodell (siehe Abbildung 4.1). Alles, was vor dem Zeitpunkt t stattfand, wird als Vergangenheit, alles nach dem Zeitpunkt t als Zukunft bezeichnet. t selbst stellt die Gegenwart dar. Je nachdem, wie genau man t bestimmen kann, gibt es mehr oder weniger Ereignisse, die gleichzeitig mit t stattfinden. Wenn man nun t hinreichend genau bestimmt, dann gibt es kein Ereignis, das gleichzeitig mit t stattfindet.

Durch die in ACID-Datenbanken implementierte Synchronisierung aller Zugriffe, wird dieses Verhalten für jedes Datum sichergestellt. Wie bereits erwähnt, stellt die dafür erforderliche Kommunikation nicht nur ein Performancehindernis dar, sondern kann auch dazu führen, dass eine Anfrage nicht beantwortet werden kann, falls z.B. der erforderliche Lock nicht gesetzt werden kann.

Da BASE-Datenbanken hier die Verfügbarkeit an die erste Stelle setzen, werden hier Modelle benötigt, um Abfolgen von Operationen darstellen und einordnen zu können.

Ein naiver Ansatz bestünde darin, jede Änderung mit einem Zeitstempel einer Einheitsuhr zu versehen und alle Änderungen später gemäß diesem Zeitstempel zu sortieren. Sollten dann mehrere Änderungen des selben Datums erfolgt sein, so müsste nur die letzte dieser Änderungen durchgeführt werden, um einen korrekten Zustand zu erhalten.

Abbildung 4.2 zeigt eine mögliche Abfolge von Änderungen. Ausgehend von v_0 erzeugt Prozess 1 zuerst v_1 und danach v_2. Prozess 2 erzeugt nun vor der Synchronisation aus v_0 den Wert v_3. Wenn man diese Änderungen im Zuge der Synchronisation einfach nach dem Zeitpunkt der Änderung sortiert, gehen die durch Prozess 1 gemachten Änderungen

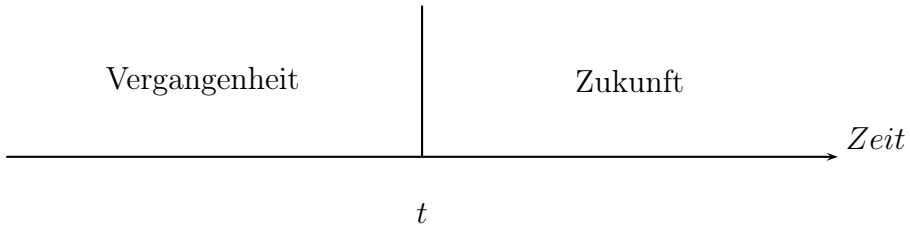

Abbildung 4.1: Das klassische Zeitmodell

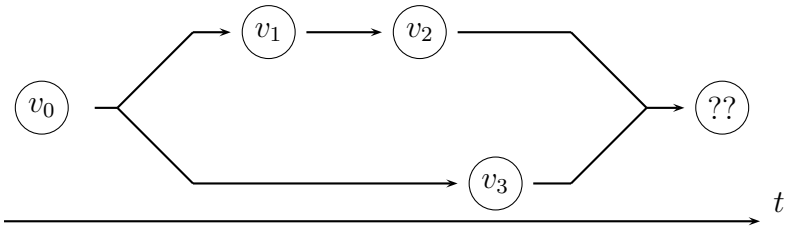

Abbildung 4.2: Abfolge von Änderungen

verloren. Es hängt dabei von der Semantik der Änderung ab, ob dieses Vorgehen korrekt ist oder nicht.

Tatsächlich spielt in Datenverarbeitungssystemen nicht der Änderungszeitpunkt, sondern der kausale Zusammenhang von Änderungen die entscheidende Rolle. Um dies zu verdeutlichen zeigt Abbildung 4.3 das hier passendere relativistische Zeitmodell. r ist dabei der Abstand zum Ort des Ereignisses e.

Da sich eine Wirkung maximal mit Lichtgeschwindigkeit ausbreiten kann, folgt, dass jedes Ereigniss einen Ereignishorizont hat (in Abbildung 4.3 dargestellt durch die von e ausgehende Diagonale). Dieser stellt die maximale Entfernung dar, in der ein Ereignis andere Ereignisse beeinflussen kann. Der Radius dieses Ereignishorizontes ist dann $r = c * (t - t_x)$.

Damit lassen sich die Begriffe Vergangenheit, Gegenwart und Zukunft rein kausal definieren:

- Die Vergangenheit eines Ereignisses x ist die Menge aller Ereignisse, die x beeinflussen können.
- Die Zukunft eines Ereignisses x ist die Menge aller Ereignisse, die von x beeinflusst werden können.
- Alle anderen Ereignisse finden relativ gleichzeitig zu x statt.

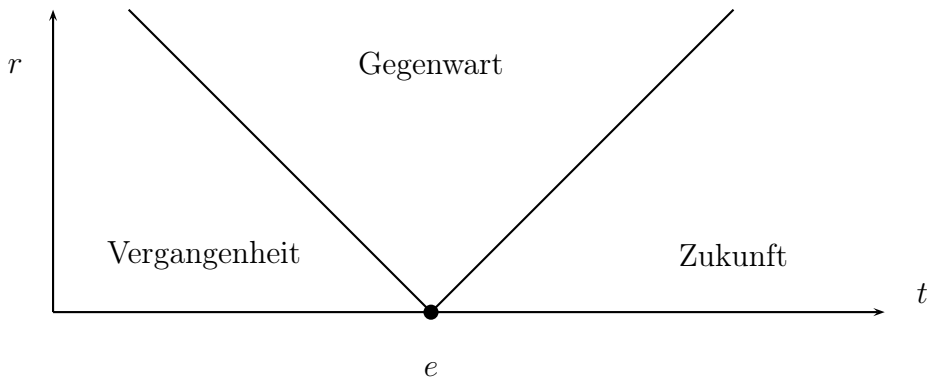

Abbildung 4.3: Das relativistische Zeitmodell

Betrachtet man unter diesem Gesichtspunkt die Situation in Abbildung 4.2, dann stellt man fest, dass die von Prozess 1 und Prozess 2 durchgeführten Änderungen gleichzeitig erfolgt sind, da keine der gemachten Änderungen den jeweils anderen Prozess bereits erreicht hat.

4.1 Happened-before-Relation

Wenn man bedenkt, dass eine Synchronisation von Rechneruhren nur bedingt möglich ist (normalerweise im Bereich von wenigen Millisekunden im Fall von NTP) und dass weiterhin die Implementierung von Rechneruhren von System zu System abweichen kann (z.B. bei unterschiedlichen Implementierungen von Schaltsekunden), so erscheint es sinnvoll, auch für Software einen Kausalitätsbegriff ohne Bezug auf eine gemeinsame Zeitbasis zu definieren[1].

Die *Happened-before* Relation ist eine strikte, partielle Ordnung mit folgenden Eigenschaften [LP78]:

- Für 2 Ereignisse a und b im selben Prozess gilt $a \rightarrow b$ genau dann wenn a im Algorithmus vor b auftritt.
- Wenn a das Senden einer Nachricht von einem an einen anderen Prozess ist und b der Empfang der selben Nachricht, dann gilt $a \rightarrow b$
- Die Relation ist transitiv, d.h. aus $a \rightarrow b$ und $b \rightarrow c$ folgt $a \rightarrow c$
- Zwei Ereignisse a und b sind gleichzeitig, wenn weder $a \rightarrow b$ noch $b \rightarrow a$ gilt.

Ein Beispiel soll verdeutlichen, welche Folgen das hat: Stellt man 2 Rechner r_1 und r_2 nebeneinander auf, ohne eine Kommunikationsmöglichkeit zwischen beiden Rechnern, dann sind aus Sicht von r_1 zu jedem Zeitpunkt alle Ereignisse auf r_2 gleichzeitig, unabhängig davon, ob sie (basierend auf einer absoluten Uhr) bereits geschehen sind oder noch geschehen werden. Aus Sicht von r_2 gilt das selbe für r_1.

4.2 Logische Uhr (Lamport Clock)

Eine einfache logische Uhr lässt sich wie folgt definieren: Sei $C : E \rightarrow N$ eine Abbildung, die jedem Ereignis eine Ganzzahl als "Uhrzeit" zuweist. Weiterhin gelte: Aus $a \rightarrow b$ folgt $C(a) < C(b)$.

Die Implementierung sieht dann folgendermaßen aus:

[1] Im Rest des Kapitels werden die Begriffe "vor", "nach" und "gleichzeitig" ausschließlich in der oben genannten kausalen Bedeutung verwendet.

- Jeder Prozess erhöht seinen eigenen Zähler vor jedem Ereignis um 1.
- Sendet ein Prozess eine Nachricht, wird der aktuelle Zähler mitgesendet.
- Erhält ein Prozess eine Nachricht, so setzt er seinen eigenen Zähler auf das Maximum aus eigenem und empfangenem Wert.

Dieses Uhrmodell erlaubt eine partielle Ordnung der Ereignisse. Der Nachteil ist, dass aus $C(a) < C(b)$ nicht geschlussfolgert werden kann, dass $a \to b$ gilt. Lediglich folgende Schlussfolgerung ist möglich: Aus $C(a) \geq C(b)$ folgt, dass nicht $a \to b$ gilt.

Anders ausgedrückt bedeutet $C(a) < C(b)$, dass entweder a vor b stattfand, oder dass beide Ereignisse relativ gleichzeitig stattfanden.

Abbildung 4.4 zeigt ein Beispiel für das Verhalten der Lamport-Clocks dreier Prozesse:

- Die Anfangsstände lauten $C(P_1) = 10$, $C(P_2) = 22$ und $C(P_3) = 16$
- P_2 erhöht seine eigene Uhr um 1 und sendet M_1 an P_3. P_3 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 17. Danach wird mit dem empfangenen Wert 23 verglichen und gemäß der oben angegebenen Regeln $C(P_3) = 23$ gesetzt.
- P_1 erhöht seine eigene Uhr um 1 und sendet M_2 an P_2. P_2 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 24. Danach wird mit dem empfangenen Wert 11 verglichen und gemäß der oben angegebenen Regeln der Wert $C(P_2) = 24$ beibehalten.
- P_3 erhöht seine eigene Uhr um 1 und sendet M_3 an P_1. P_1 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 12. Danach wird mit dem empfangenen Wert 24 verglichen und gemäß der oben angegebenen Regeln der Wert $C(P_1) = 24$ gesetzt.
- P_1 erhöht seine eigene Uhr um 1 und sendet M_4 an P_2. P_2 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 25. Danach wird mit dem empfangenen

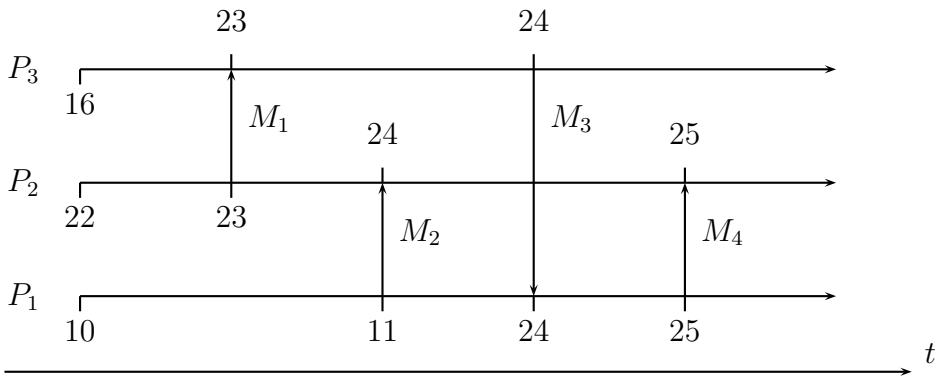

Abbildung 4.4: Beispiel Lamport Clock

Wert 25 verglichen und gemäß der oben angegebenen Regeln der Wert $C(P_2) = 25$ beibehalten.

4.3 Vektoruhr

Die oben vorgestellte Lamport-Clock erlaubt keine Aussage über Gleichzeitigkeit. Mit einer einfachen Erweiterung lässt sich diese Aussage jedoch treffen. Sei V ein Vektor der Dimension n mit folgenden Eigenschaften:

- V enthält einen Eintrag für jeden Prozess.
- Für alle x ist V_x die Lamport-Clock des entsprechenden Prozesses. Ausschließlich dieser Eintrag wird von Prozess x gemäß den oben genannten Regeln aktualisiert.
- Wird eine Nachricht versendet, übermittelt der Sender V mit an den Empfänger
- Beim Empfang einer Nachricht berechnet der Empfänger seinen neuen Vektor mit $V' = max(V, V_r)$. $max(V, V_r)$ ist hierbei das komponentenweise Maximum der Vektoruhren des Empfängers und des Senders.

Die Ordnungseigenschaft der Vektoruhr lässt sich folgendermaßen definieren: Seien $V(x)$ und $V(y)$ die Vektoruhren zweier Ereignisse. Dann gilt:

$$V(x) < V(y) \leftrightarrow \forall a(V(x)_a \leq V(y)_a) \wedge \exists b(V(x)_b < V(y)_b) \tag{4.1}$$

$V(x)$ ist also kleiner als $V(y)$, wenn alle Komponenten von $V(x)$ kleiner oder gleich den entsprechenden Komponenten von $V(y)$ sind und es mindestens eine Komponente in $V(x)$ gibt, die echt kleiner als die entsprechende Komponente in $V(y)$ ist.

Wie bei der Lamport-Clock gilt auch hier: Aus $x \to y$ folgt $V(x) < V(y)$. Im Gegensatz zur Lamport-Clock gilt hier allerdings auch die Umkehrung: Aus $V(x) < V(y)$ folgt $x \to y$.

Damit lässt sich auch eine Aussage über Gleichzeitigkeit machen. Gilt weder $V(x) < V(y)$ noch $V(y) < V(x)$, dann sind die Ereignisse x und y gleichzeitig.

Abbildung 4.5 zeigt ein Beispiel für das Verhalten der Vektoruhr:

- Zu Anfang kennt jeder Prozess nur seinen eigenen Eintrag. Damit lauten die Anfangsstände: $C(P_1) = (10, 0, 0)$, $C(P_2) = (0, 22, 0)$ und $C(P_3) = (0, 0, 16)$
- P_2 erhöht seine eigene Uhr um 1 und sendet M_1 an P_3. P_3 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 17. Danach wird das komponentenweise Maximum aus vorhandener und erhaltener Uhr gebildet und das Ergebnis $C(P_3) = (0, 23, 17)$ gesetzt.

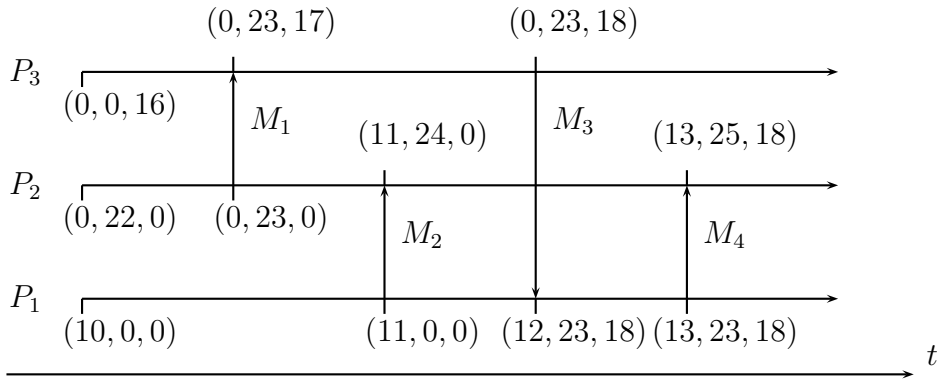

Abbildung 4.5: Beispiel Vektoruhr

- P_1 erhöht seine eigene Uhr um 1 und sendet M_2 an P_2. P_2 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 24. Danach wird das komponentenweise Maximum aus vorhandener und erhaltener Uhr gebildet und das Ergebnis $C(P_3) = (11, 24, 0)$ gesetzt. Im Gegensatz zur vorher beschriebenen Lamport-Clock wird damit der Zustand von P_1 nicht verworfen, sondern bleibt wie bereits bei M_1 als Bestandteil des Vektors erhalten.

- P_3 erhöht seine eigene Uhr um 1 und sendet M_3 an P_1. P_1 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 12. Nach der Bildung des komponentenweisen Maximums ergibt sich $C(P_1) = (12, 23, 18)$.

- P_1 erhöht seine eigene Uhr um 1 und sendet M_4 an P_2. P_2 erhöht beim Empfang der Nachricht seine Lamport-Clock um 1 auf 25. Nach der Bildung des komponentenweisen Maximums ergibt sich $C(P_2) = (13, 25, 18)$. Interessant dabei ist, dass P_2 einen Wert für P_3 kennt, obwohl nie direkt eine Nachricht von P_3 an P_2 gesendet wurde.

Über M_1, M_2, M_3 und M_4 lassen sich damit folgende Aussagen treffen:

- M_1 und M_2 wurden gleichzeitig gesendet. Da $11 > 0$ und $23 > 0$ folgt, dass weder $(0, 23, 0) < (11, 0, 0)$ noch $(11, 0, 0) < (0, 23, 0)$ gilt.

- M_2 wurde empfangen, nachdem M_1 gesendet wurde, da $(0, 23, 0) < (11, 24, 0)$ gilt.

- M_2 und M_3 wurden gleichzeitig gesendet. Da $11 > 0$, aber $0 < 23$ und $0 < 18$ ist, gilt weder $(11, 0, 0) < (0, 23, 18)$, noch $(0, 23, 18) < (11, 0, 0)$.

- M_4 wurde nach M_1 gesendet, da $(0, 23, 0) < (13, 23, 18)$

4.4 Versionsvektor

In einer verteilten Datenbank spielt die Zeitvorstellung der einzelnen Datenbankprozesse eine untergeordnete Rolle. Erheblich wichtiger sind hier die Aktualisierungszeiten einzelner Daten. Wendet man die Idee der Vektoruhr hier an, so enthält dieser je einen Eintrag pro Kopie des Datensatzes.

Wird ein Eintrag lokal aktualisiert, dann wird ausschließlich das zugehörige Element im Versionsvektor erhöht. Sobald zwei unterschiedliche Repliken synchronisiert werden, wird der Versionsvektor auf das komponentenweise Maximum geändert.

Damit lassen sich Kopien des selben Datums auf unterschiedlichen Knoten daraufhin prüfen, ob Änderungen an ihnen nacheinander oder gleichzeitig erfolgt sind[2].

[2]Genauere Informationen dazu sind in Kapitel 7.3 zu finden.

5 NoSQL-Architekturen

Seit den 1980ern versteht man unter dem Begriff "Datenbank" hauptsächlich die Kombination aus tabellenorientiertem, relationalen DBMS, ACID und SQL. Dazu hat insbesondere die leichte Erlernbarkeit und Handhabbarkeit von SQL beigetragen. Nicht umsonst kursieren Witze wie zum Beispiel:

> Wie kommt der Datenbankprogrammierer an sein Mittagessen?
> SELECT Mittagessen FROM Kantine WHERE Fleisch='Steak';[1]
>
> oder:
>
> Wie fängt der Datenbankprogrammierer einen Elefanten in Afrika?
> SELECT Elefant FROM Afrika LIMIT 1;[2]

SQL wurde nicht nur für die oben genannten relationalen DBMS und ACID entworfen, es ist auch nur dafür wirklich geeignet. Die Suche nach Alternativen zu ACID führte damit zu einer Suche nach Alternativen zu SQL und relationalen Datenbanken. Diese Alternativen werden heute üblicherweise unter der Bezeichnung *NoSQL* zusammengefasst.

Dabei bedeutet NoSQL zwar eine Loslösung von dem oben genannten strikten Verständnis einer Datenbank, es gibt jedoch eine Reihe von NoSQL-Datenbanken, die ACID-Transaktionen unterstützen. Das gleiche gilt auch für SQL. Es gibt durchaus eine Reihe von NoSQL Datenbanken, die SQL Interfaces anbieten. Meist sind diese jedoch weniger performant, als die nativen API und werden angeboten, um eine leichtere Migration bestehender SQL-basierter Anwendungen zu ermöglichen.

Die Stärke relationaler Datenbanken liegt in ihrer Fähigkeit, Daten zu verknüpfen. Dies wird jedoch in vielen Fällen nicht benötigt. Zum Beispiel werden in Web-CMS-Systemen zwar SQL Datenbanken verwendet, in vielen Fällen werden die Daten jedoch in der Form eines Key-Value-Paares gespeichert[3]. Die Abfrage erfolgt ausschließlich über den Key. Hier werden weder die Fähigkeiten relationaler DBMS zur Verknüpfung von Daten, noch

[1] Für Vegetarier lautet der Aufruf: SELECT Mittagessen FROM Kantine WHERE Fleisch IS NULL;

[2] LIMIT 1 ist hier extrem wichtig. Anderenfalls fängt man versehentlich alle afrikanischen Elefanten

[3] Als Schlüssel dient oft eine Seitennummer, der zugehörige Wert ist dann der HTML- oder PHP-Quelltext der Seite.

die Stärken von SQL bei der Abfrage dieser Daten benötigt. Im Gegenteil entsteht hier ein erheblicher und überflüssiger Overhead.

Eine ähnliche Situation liegt vor, wenn z.B. Graphen in einer relationalen Datenbank gespeichert werden sollen. Die hierfür erforderlichen rekursiven Abfragen sind mit vielen SQL Dialekten nicht möglich. In anderen Fällen sind sie eingeschränkt und/oder nur mit erheblichem Overhead möglich. In solchen Fällen behilft man sich meistens damit, lediglich die Knoten und die Adjazenzliste in der Datenbank abzulegen und die Auswertungslogik in das Anwendungsprogramm zu verlagern. Auch hier werden Daten wieder nur direkt abgefragt, ohne die Stärken des relationalen DBMS zu benötigen.

Eine Alternative für die beiden eben genannten Beispiele stellt die BerkleyDB dar. Dabei handelt es sich um einen hochperformanten Key-Value-Store [BDB1], der aufgrund geringer Systemanforderungen auch gern in embedded-Systemen verwendet wird. Neuere Versionen können (je nach Build) auch in verteilten Umgebungen verwendet werden. Dabei werden die einzelnen Datensätze (bis zu einer maximalen Länge von 4 GB pro Datensatz) unter einem beliebigen Schlüssel abgelegt und können mit Hilfe dieses Schlüssels wieder aus der Datenbank gelesen werden.

Während SQL Datenbanken üblicherweise tabellenorientiert sind, findet man im Bereich NoSQL eine Bandbreite von unstrukturiert bis hin zu hoch strukturierten Daten. Allen NoSQL Datenbanken ist dabei gemein, dass Strukturen weniger rigid gehandhabt werden, als das in SQL Datenbanken üblich ist.

Grundsätzlich sind alle NoSQL Datenbanken als Key-Value-Typen einzuordnen, da in jedem Fall ein Schlüssel zum Zugriff auf einen Datensatz benötigt wird. Es gibt jedoch erhebliche Unterschiede in den Möglichkeiten, die Werte seitens der Datenbank zu verarbeiten.

5.1 Beispiel zu speichernder Daten

Um die Unterschiede unterschiedlicher Organisationsformen von Daten in Datenbanken darstellen zu können, wird im Folgenden in jeder Datenbank die selbe Struktur gespeichert. Dabei handelt es sich um den Berechnungsbaum für den Term $(2 + 3) * 4$.

Dieser Baum kann grundsätzlich auf unterschiedliche Arten dargestellt werden:

Als Baum (siehe Abbildung 5.1).

Als Zeichenkette: $(2 + 3) * 4$

Als Knoten- und Adjazenzliste (hier zusammengesasst):

Knoten	n_1	n_2	n_3	n_4	n_5
Typ	Operator	Operator	Wert	Wert	Wert
Wert	$*$	$+$	2	3	4
Kind 1	n_2	n_3	–	–	–
Kind 2	n_5	n_4	–	–	–

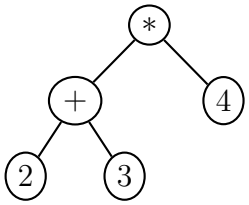

Abbildung 5.1: Berechnungsbaum für $(2 + 3) * 4$

5.1.1 Realisierung in SQL

Zum Vergleich werden im folgenden 2 mögliche Realisierungen in SQL angegeben. Da sich die Baumstruktur in SQL nicht abbilden lässt, kann je nach Anwendung entschieden werden, ob z.B. als nested List gespeichert wird oder ob die einzelnen Knoten separat in der Datenbank abgelegt werden sollen.

Für das Beispiel wird als Datenbank H2 (http://www.h2database.com) verwendet.

strukturlose Speicherung

In diesem Fall wird eine einfache Tabelle zu Speicherung der Zeichenkette angelegt und diese darin abgelegt.

```
CREATE TABLE trees (
  key INTEGER PRIMARY KEY,
  value VARCHAR
);

INSERT INTO trees
  (key, value)
VALUES
  (1, '(2+3)*4');
```

Vorteil dieser Fassung ist, dass nur eine Abfrage der Datenbank erforderlich ist. Aus dieser kann der gesamte Baum rekonstruiert werden. Der Nachteil besteht allerdings darin, dass die Daten jedesmal geparst werden müssen.

strukturierte Speicherung

Da SQL keine Polymorphie unterstützt, der Berechnungsbaum jedoch aus zwei unterschiedlichen Knotentypen (Operationen und Werten) besteht, ist eine 1:1 Übertragung

nicht möglich. Im Folgenden wird eine einfache Variante mit lediglich einer Tabelle vorgestellt:

```
CREATE TABLE nodes(
  nodeid   INTEGER PRIMARY KEY,
  parent   INTEGER REFERENCES(nodeid) ON DELETE CASCADE,
  operator CHAR,
  value    DOUBLE
);

INSERT INTO nodes (
  nodeid , parent , operator , value
) VALUES
  (1, null, '*', null),
  (2,    1, '+', null),
  (3,    2, 'v', 2),
  (4,    2, 'v', 3),
  (5,    1, 'v', 4);
```

Vorteil dieser Variante ist, dass mit einem einzigen prepared-Statement sowohl die Wurzelknoten aller Bäume bestimmt werden können, als auch die Kinder individueller Knoten:

```
SELECT FROM nodes WHERE parent=?;
```

Ebenso könnten Teilbäume gelöscht werden, indem der Wurzelknoten aus der Datenbank entfernt wird. Ebenso ist ein unter Umständen aufwändiges Parsen der Daten nicht erforderlich. Der Nachteil besteht darin, dass zur Auswertung mehrere SQL-Anfragen erforderlich sind.

5.2 Key-Value-Store

Einfache Key-Value-Stores stellen die grundlegendste Form von NoSQL Datenbanken dar. In dieser Variante wird ein Wert in der Datenbank durch einen eindeutigen Schlüssel gekennzeichnet. Der Wert selbst wird dabei von der Datenbank lediglich gespeichert. Dem DBMS liegen keine Information über Inhalt oder Struktur der gespeicherten Werte vor. Bereichsoperationen, wie sie von SQL bekannt sind[4], sind daher nicht möglich. Üblicherweise werden nur die Wörterbuchoperationen *INSERT*, *UPDATE*, *DELETE* und *FIND* unterstützt. Optional auf vielen Systemen verfügbar sind Transaktionen, Replikationsmanagement und Concurrency-Control.

[4] z.B.: SELECT ... WHERE value<x;

Der große Vorteil dieser Typen besteht in ihrer Einfachheit, Robustheit, einem extrem geringen Speicherbedarf und hoher Performance (Datendurchsatz und Reaktionsgeschwindigkeit).

5.2.1 Beispiel

Die offensichtlich einfachste Art, die Daten aus Abschnitt 5.1 zu speichern, ist die bereits vorgestellte Variante als Zeichenkette:

```
INSERT('1', '(2+3)*4')
```

Ebenso ist eine strukturierte Speicherung möglich. Als Schlüssel dafür wird die Zeichenkette $< Knotennummer > . < Feld >$. Weiterhin werden leere Felder nicht gespeichert.

```
INSERT('1.Operator', '*')
INSERT('1.Links', 2)
INSERT('1.Rechts', 5)

INSERT('2.Operator', '+')
INSERT('2.Links', 3)
INSERT('2.Rechts', 4)

INSERT('3.Wert', 2)

INSERT('4.Wert', 3)

INSERT('5.Wert', 4)
```

Im Gegensatz zu der in Abschnitt 5.1 vorgestellten SQL-Variante müssen hier zum Entfernen des Baums alle Knoten manuell gelöscht werden. Ebenso ist es erforderlich, eine Liste der Root-Knoten zu speichern.

Generell können die von SQL Datenbanken bekannten Tabellen in einem Key-Value-Store gespeichert werden, wenn zum Beispiel ein Schlüssel in der Form "Tabellenname.Zeilennummer.Spaltenname" verwendet wird. Das ist zum Beispiel bei MySQL der Fall, wenn eine BerkleyDB als Backend verwendet wird.

Beispiele für verteilte Key-Value-Stores sind Riak, Redis, DynamoDB und die bereits mehrfach erwähnte BerkleyDB [NSQL].

5.3 Wide-Column-Store

Wide-Column-Stores speichern Daten ebenfalls als Key-Value-Paare, diese werden hier jedoch als Columns bezeichnet. Eine Zusammenfassung einer beliebigen Anzahl von

Columns wird als Column Family bezeichnet [WCST]. Damit stellt jede Column Family einen eigenen Key-Value-Store dar.

Die relevante Erweiterung entsteht durch die Einführung von Super Columns. Zwar wird diese, wie auch eine Column, durch einen Schlüssel bezeichnet, enthält statt eines einzelnen Wertes jedoch beliebig viele Columns oder Super Columns. Genauso wie Columns können sie in einer Column Family enthalten sein.

Es können also Key-Value-Stores in Key-Value-Stores gespeichert werden. Das ermöglicht es, jede zyklenfreie Struktur (z.B. Records, n-dimensionale Arrays oder Bäume) direkt in der Datenbank abzubilden.

Eine Besonderheit stellt hier die Super Column Family dar. Im Gegensatz zu einer Column Family kann sie lediglich Super Columns enthalten.

Im Gegensatz zu den bei relationalen Datenbanken genutzen Tabellen sind die Wide-Column-Stores aufgrund ihrer Schemafreiheit wesentlich flexibler. Dadurch ist es auch möglich, polymorphe Datenstrukturen in einem Wide-Column-Store abzubilden.

Beispiele für verteilte Wide-Column-Stores sind Cassandra, Hadoop/HBase und SimpleDB.

5.3.1 Beispiel

Die einzelnen Knoten des Berechnungsbaums in Abbildung 5.1 werden jeweils in einer Supercolumn gespeichert. Die Super Column für ein Blatt enthält dabei eine einzelne Column mit dem Namen “Wert”. Die Super Column für einen inneren Knoten hingegen enthält eine Column mit dem Namen “Operator” und zwei Supercolumns mit den Namen “Links” und “Rechts”.

Der gesamte Berechnungsbaum wird in einer einzelnen Supercolumn mit dem Namen “Wurzel” gespeichert. Abbildung 5.2 zeigt das.

<table>
<tr><td>Name</td><td colspan="4">Wert</td></tr>
<tr><td rowspan="8">Wurzel</td><td>Name</td><td colspan="3">Wert</td></tr>
<tr><td>Operator</td><td colspan="3">’*’</td></tr>
<tr><td rowspan="5">Links</td><td>Name</td><td colspan="2">Wert</td></tr>
<tr><td>Operator</td><td colspan="2">’+’</td></tr>
<tr><td rowspan="2">Links</td><td>Name</td><td>Wert</td></tr>
<tr><td>Zahl</td><td>2</td></tr>
<tr><td rowspan="2">Rechts</td><td>Name</td><td>Wert</td></tr>
<tr><td>Zahl</td><td>3</td></tr>
<tr><td rowspan="2">Rechts</td><td>Name</td><td>Wert</td></tr>
<tr><td>Zahl</td><td>4</td></tr>
</table>

Abbildung 5.2: Beispieldaten im Wide-Column-Store

Der Knoten mit dem Wert "3" kann dann über die Schlüsselfolge: $Wurzel.Links.Rechts$ angesprochen werden.

5.4 Document-Store

Wie der Name bereits andeutet, handelt es sich bei *Document-Stores* um Key-Value-Datenbanken, in denen Dokumente gespeichert werden. Zwar unterscheiden sich die einzelnen Implementierungen in den Details ihres Verständnises des Begriffes Dokument, folgende Varianten sind jedoch meist üblich (und in der selben Datenbank realisierbar):

- unstrukturierte Daten (vergleichbar mit BLOB oder CLOB Typen in relationalen Datenbanken) oder
- strukturierte Daten, bestehend aus Metadaten+Daten, z.B. XML, JSON oder BSON[5]

Sofern strukturierte Daten gespeichert werden, können diese üblicherweise durchsucht und indiziert werden. Dabei sind sowohl Gleichheitsabfragen, als auch Bereichsabfragen möglich. Einige dieser Datenbanken unterstützen SQL als Abfragesprache um den Umstieg zu erleichtern. Die volle Leistungsfähigkeit kann damit jedoch nicht erreicht werden.

Wie alle anderen Key-Value-Datenbanken sind auch Document-Stores schemafrei, d.h. zum einen können Daten unterschiedlicher Struktur gespeichert werden, zum anderen muss die Struktur weder im Vorfeld deklariert, noch überhaupt bekannt sein.

Beispiele hierfür sind MongoDB (BSON), MarkLogic Server (JSON, XML) und CouchDB (JSON).

5.4.1 Beispiel

Der Berechnungsbaum aus Abbildung 5.1 wird als Dokument (assoziatives Array) gespeichert. Dabei werden folgende Einträge verwendet:

Name	Typ
Operator	Zeichenkette
Wert	Fließkommazahl
Links	Array
Rechts	Array

Das JSON-Dokument sieht dann folgendermaßen aus:

[5]Bei JSON handelt es sich um assoziative Arrays in Javascript Object Notation (Klartext). Jedes einzelne Element kann dabei einen primitiven Datentyp (Integer, String, etc.) oder ein weiteres assoziatives Array enthalten. Damit kann man JSON als assoziativen Baum verstehen. Bei BSON handelt es sich um eine verkürzte Binärdarstellung von JSON Daten

```
"Wurzel": {
         "Operator": "*"
         "Links":    {
                      "Operator": "+"
                      "Links":    {
                                   "Wert": 2
                                  }
                      "Rechts":   {
                                   "Wert": 3
                                  }
                     }
           "Rechts":   {
                        "Wert": 4
                       }
          }
```

Der Knoten mit dem Wert 3 kann dabei über den Schlüssel des Dokuments, gefolgt von der Folge an Subschlüsseln im Dokument angesprochen werden: $1.Wurzel.Links.Rechts$

5.5 Graphen Datenbanken

Abweichend von allen bisher vorgestellten Typen, werden in *Graphen Datenbanken* Daten in Form von Graphen gespeichert. Die primitiven Elemente sind dabei:

- Knoten, mit denen Entitäten repräsentiert werden (z.B.: Personen, Unternehmen, Konten, etc.)
- Eigenschaften, die zu diesen Knoten gehören (im Fall einer Person z.B. deren Name, Geburtsdatum und Geschlecht)
- Kanten, die die einzelnen Knoten in Relation zueinander stellen.

Die Grundidee dieser Datenbanken besteht darin, Relationen direkt zu repräsentieren. Häufig werden in den Anfragen diverse Graphenalgorithmen (Kürzeste Wege, Zusammenhangskomponenten, etc) unterstützt.

Beispiele hierfür sind InfoGrid und MapGraph.

5.6 Weitere Typen

Zwar stellen die oben genannten Typen die häuftigsten Organisationsformen von NoSQL Datenbanken dar, es gibt jedoch noch eine Vielzahl anderer Organisationsformen, die hier nur in Kürze erwähnt werden sollen:

- *Objekt Datenbanken* speichern Objekte und deren Beziehungen. Diese Datenbanken sind geeignet, wenn direkt Objektkonstrukte einer Anwendung gespeichert werden sollen.
- *XML Datenbanken* sind eine Spezialform[6] von Document Stores, die speziell für den Einsatz mit XML konzipiert wurden.
- *Multidimensionale Datenbanken* repräsentieren Daten als Multidimensionale Felder. Damit kann man sie als Erweiterung von Tabellen (2-dimensional) ansehen und in gewisser Weise auch als Vorform von Document Stores.
- *Multi-value Datenbanken* sind in der Lage, mehrere Werte pro Datenfeld (vergleichbar einer Zelle in tabellenbasierten Systemen) zu speichern.

5.6.1 Multi-Model-Datenbanken

Eine Reihe von Datenbanken unterstützen gleichzeitig mehrere Speichermodelle. Das ermöglicht dem Anwender größere Flexibilität hinsichtlich der Darstellung der Daten, ohne dass unterschiedliche Datenbanken erforderlich sind.

- ArangoDB: Document-Store, Key-Value-Store und Graphen-Datenbank
- OrientDB: Objektdatenbank, Document-Store, Key-Value-Store und Graphen-Datenbank
- AlchemiyDB: Graphen-Datenbank, relationale Datenbank, Document-Store und Key-Value-Store
- CortexDB: Document-Store, Key-Value-Store, Graphen-Datenbank, Multi-Value Datenbank und Wide-Column Datenbank

[6] z.T auch Vorläufer

6 Verteilung von Daten- und Transaktionsmanagement

Dieses Kapitel beleuchtet das Thema Verteilung zwar anhand von ACID Datenbanken, die meisten der genannten Punkte sind jedoch für alle Typen von Datenbanken relevant.

Hinsichtlich der Datenverteilung gibt es keine relevanten Unterschiede zwischen ACID und BASE.

Zwar benötigen BASE Datenbanken keinen Transaktionsmanager, diese Stelle wird jedoch von der Benutzerschnittstelle eingenommen. Im Gegensatz zu ACID Datenbanken ist bei BASE jedoch kein knotenübergreifendes Lock-Management erforderlich.

Bei der Betrachtung von Datenbanken sind 5 Elemente interessant:

1. Die Transaktionsanforderungen.
2. Der *Transaktionsmanager* (TM), der für die Verwaltung der einzelnen Transaktion verantwortlich ist.
3. Der *Lockmanager* (LM), der alle Anforderungen von Read- und Write-Locks verwaltet[1].
4. Der *Speichermanager* (SM), der den Zugriff auf den Datenspeicher kontrolliert. Die Kommunikation mit dem Transaktionmanager findet dabei über einfache Read- und Write-Kommandos statt. Weiterhin obliegt es dem Speichermanager, wie Logeinträge verwaltet werden und wann diese in den Master der Datenbank integriert werden[2].
5. Der *Datenspeicher*, der für die Ausführung der vom Speichermanager beauftragten Lese- und Schreiboperationen verantwortlich ist. Dieser entscheidet auch, in welcher Weise der Zugriff durchgeführt wird (in-situ-Änderung, read-modify-write und/oder Log-Eintrag).

Dabei sind die Punkte 2-4 Bestandteil der Datenbank. Der Datenspeicher selbst unterliegt meist der Verwaltung durch das Betriebssystem.

[1] Hinweis: Auch in Systemen mit MVCC werden üblicherweise mindestens Write-Locks verwendet, um das Auffinden potentieller Kollisionen zu erleichtern

[2] Garbage-Collection

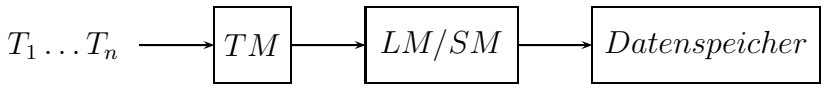

Abbildung 6.1: prinzipieller Aufbau einer Datenbank

Abbildung 6.1 zeigt die einfachste Möglichkeit, diese Komponenten zusammenzustellen. Die Transaktionsanforderungen werden in einer Warteschleife gehalten und nacheinander abgearbeitet. Auf einen Lockmanager kann in diesen Fällen verzichtet werden. Zwar funktioniert diese Variante, sie ist jedoch keinesfalls leistungsfähig und soll hier lediglich das Grundkonzept illustrieren.

6.1 Mehrere Transaktionsmanager

Auf den Transaktionsmanager entfällt ein großer Teil der effektiven Ausführungszeit einer Transaktion. Während der Bearbeitung der Transaktionen müssen zum einen Nebenbedingungen geprüft werden, zum anderen muss der Transaktionsmanager auch auf weitere Anweisungen der Anwendung warten.

Die offensichtlichste Erweiterung besteht daher darin, mehrere Transaktionsmanager einzusetzen (siehe Abbildung 6.2). Dabei ist unerheblich, ob diese auf dem selben Prozessor ausgeführt werden (time-sharing), auf anderen Prozessoren der selben Maschine oder auf anderen Maschinen laufen. Die Entscheidung, welche Variante hier gewählt wird, ist von den zu erwartenden Lasten abhängig und in den meisten DBMS konfigurierbar.

In der Regel verwenden diese Datenbanken keinen separaten Lockmanager, sondern dieser ist in den Speichermanager integriert.

Der Performancegewinn entsteht hier durch die gleichzeitige Ausführung von Transaktionen (mindestens dadurch, dass während bestehender Wartezeiten einer Transaktion andere bearbeitet werden können). Wird in einer Datenbank mit z.B. komplexen Stored-Procedures gearbeitet, kann der Einsatz mehrerer 100 Transaktionsmanager zusammen mit einem Speichermanager sinnvoll sein.

Weitere Optimierungen sind im Speichermanager möglich, da diesem eine große Anzahl von Lese- und Schreiboperationen zur Verfügung steht, die geordnet und so effizienter bearbeitet werden können (z.B. Group-Commit).

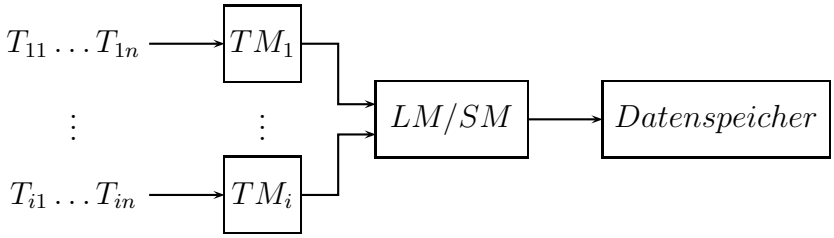

Abbildung 6.2: Datenbank mit mehreren TM und gemeinsamem Datenspeicher

Die meisten embedded Datenbanken arbeiten auf diese Weise. Dabei wird normalerweise ein Transaktionsmanager pro Datenbankverbindung (Anwendung) verwendet.

Gegebenenfalls wird eingangsseitig noch ein Load-Balancer eingesetzt, um die Transaktionen auf die einzelnen Transaktionsmanager (meist in Mehrmaschinensystemen) zu verteilen.

6.2 Mehrere Speichermanager

Ab einem gewissen Punkt genügt es nicht mehr, ein DBMS mit immer mehr Transaktionsmanagern auszustatten, da diese wiederum auf den Speichermanager warten müssen. Die einzige Lösung besteht darin, mehrere Speichermanager zu verwenden (siehe Abbildung 6.3)

Dabei gibt es grundsätzlich zwei Möglichkeiten, zu entscheiden, was mit den Daten passiert:

- Die Daten werden auf die einzelnen SM verteilt. Der Vorteil hierbei besteht darin, dass die Last für jeden einzelnen SM sinkt. Die hautpsächlichen Nachteile sind zum einen eine geringere Ausfallsicherheit (ein Ausfall eines beliebigen SM genügt bereits) und ein höherer Kommunikationsaufwand, da die TM Daten bei unterschiedlichen SM (normalerweise auf unterschiedlichen Rechnern) anfragen und speichern müssen.
- Die Daten werden zwischen den SM gespiegelt. Der Vorteil besteht darin, dass Lesezugriffe auf beliebige SM erfolgen können. Damit sinkt mindestens die durch diese Zugriffe verursachte Last. Der Nachteil ist ein größerer Aufwand zum einen für das Setzen von Locks, da zwar die Daten bei einem beliebigen SM angefragt werden können, die Locks jedoch auf allen SM gesetzt werden müssen. Weiterhin müssen Schreibzugriffe auf allen SM erfolgen.

In fast allen Datenbanken werden Kombinationen aus beiden Varianten verwendet. Dabei werden bei n Speichermanagern $m < n$ Kopien der Daten auf unterschiedlichen SM vorgehalten, um Ausfallsicherheit zu gewährleisten. Gleichzeitig werden die Daten auf unterschiedliche Speicherknoten verteilt.

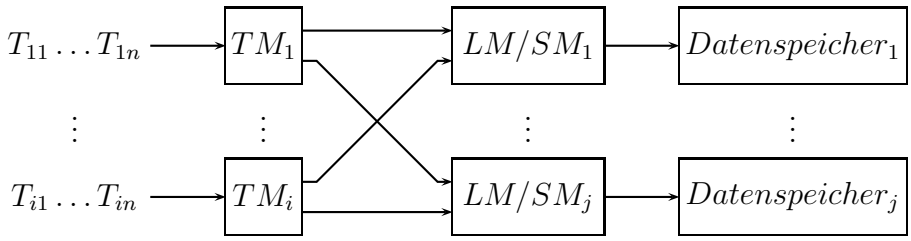

Abbildung 6.3: Datenbank mit mehreren TM und mehreren Datenspeichern

SM_1
Tabelle A
Tabelle B
Tabelle C

SM_2
Tabelle A
Tabelle C
Tabelle D

SM_3
Tabelle A
Tabelle D
Tabelle B

Abbildung 6.4: Beispiel verteilte/gespiegelte Daten

Abbildung 6.4 illustiert dies. Die Tabelle A ist dabei auf allen SM gespiegelt, die Tabellen B, C und D sind auf jeweils 2 SM zu finden. Eine derartige Verteilung wird zum Beispiel verwendet, wenn Tabelle A einen häufig benötigten Index enthält und gleichzeitig die Tabellen B-D aus Sicherheitsgründen mehrfach vorhanden sein sollen.

6.2.1 Sharding

Eine Schwierigkeit bei großen Tabellen in Datenbanken besteht darin, dass auch die Indizes mit wachsen und damit nicht nur das Suchen aufwändiger wird, sondern auch Einfüge- und Löschoperationen immer teurer.

Daher kann es unter Umständen sinnvoll sein, eine Tabelle horizontal auf unterschiedliche Server zu verteilen. Das bedeutet, dass jeweils bestimmte Blöcke von Zeilen (Shards) auf unterschiedlichen Servern liegen. Idealerweise erreicht man dabei einen Zustand, in dem entweder die abzufragende Teiltabelle sehr effizient bestimmt werden kann oder in der Suchanfragen parallel auf unterschiedlichen SM bearbeitet werden können.

Ein Beispiel soll das illustrieren:
Es ist ein Telefonbuch für Deutschland zu verwalten. Die entsprechende Tabelle enthält mehrere Millionen Einträge. Um Zugriffe schneller zu gestalten, soll diese Tabelle auf mehrere SM verteilt werden.

Es bietet sich an, die Tabelle nach Vorwahlgruppen zu verteilen, da Anfragen häufig pro Vorwahl oder Postleitzahl erfolgen und anhand der Postleitzahlen die möglichen Vorwahlen effizient bestimmt werden können.

Wenn man weiterhin davon ausgeht, dass Suchanfragen sich häufig auf die nähere Umgebung beziehen, kann man weiterhin die Daten geographisch entsprechend verteilen und damit eine große Anzahl von Anfragen auf dem geographisch nächstgelegenen Knoten bearbeiten. Damit wird nicht nur die Last für die einzelnen Knoten gesenkt, sondern auch die durchschnittliche Antwortzeit des Systems verringert.

Viele Datenbanken bieten automatisches Sharding an. Hierbei werden zum Teil Statistiken über die Zugriffe auf die Tabellen verwendet, um diese so verteilen zu können, dass Anfragen möglichst effizient beantwortet werden können.

SM_1
Tabelle A
Tabelle B
Tabelle C
Tabelle D

SM_2
Tabelle A
Tabelle B
Tabelle C
Tabelle D

SM_3
Tabelle A
Tabelle B
Tabelle C
Tabelle D

SM_4
Tabelle A
Tabelle B
Tabelle C
Tabelle D

Abbildung 6.5: Beispiel Mastercopy mit gespiegelte Daten

6.2.2 Master-Copy

Finden in einer Datenbank hauptsächlich Lesezugriffe statt (z.B. CMS), empfiehlt es sich, einen SM zur Master-Copy zu bestimmen. Alle Schreibzugriffe müssen lediglich auf dem Master erfolgen, alle anderen SM dienen als Cache für Lesezugriffe. Damit müssen auch Read-Locks maximal auf zwei SM gesetzt werden, einmal auf dem SM, von dem gelesen wird und einmal auf der Master-Copy.

Abbildung 6.5 zeigt einen möglichen Aufbau. Die Daten sind hier auf allen SM gespiegelt. Die jeweilige Master-Copy der Tabelle ist fett markiert. Schreibzugriffe auf Tabelle A müssen hierbei nur bei SM_1 angemeldet werden. Die Replikation erfolgt üblicherweise durch die SM.

Soll ein Datum aus Tabelle B von SM_4 gelesen werden, so ist neben dem Read-Lock auf SM_4 auch ein Read-Lock auf SM_2 erforderlich.

6.3 Zentraler Lock-Manager

Häufig stellen SM und LM eine Einheit dar. Da die Last für den Lockmanager jedoch verhältnismäßig gering ist (die Entscheidung, ob es möglich ist, ein Lock zu setzen, kann sehr schnell getroffen werden) und gleichzeitig ein verteiltes Lockmanagement für den Transaktionsmanger oder die Lockmanager einen erheblichen Kommunikations- und Synchronisationsaufwand bedeuten kann, besteht auch die Möglichkeit, einen zentralen Lockmanager einzusetzen, der alle Locks auf allen SM verwaltet. Dieser kann gleichzeitig auch die Transaktionsmanager darüber informieren, auf welchen SM Daten zu finden oder zu speichern sind. Abbildung 6.6 illustriert den Aufbau.

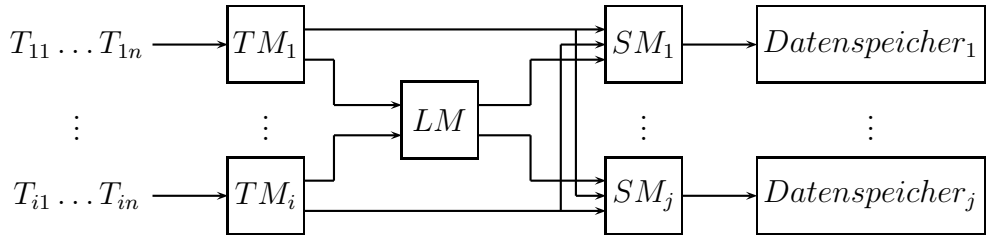

Abbildung 6.6: Datenbank mit zentralem Lockmanager

6.4 Replikationsstrategien

Sobald auf Ebene der SM Daten dupliziert (gespiegelt) vorliegen, stellt sich die Frage, wann und wie diese synchronisiert werden. Dabei werden zwei Strategien unterschieden:

eager: Hierbei werden die Daten direkt nach der Schreiboperation mit allen vorhandenen Kopien synchronisiert. Der Vorteil dabei ist, dass zum einen mit der Transaktionsbestätigung kombiniert werden kann, zum anderen liegen die Daten mit möglichst geringer Verzögerung auf allen Knoten vor.

lazy: In diesem Fall werden die Daten erst beim nächsten Lesezugriff synchronisiert. Der Vorteil hierbei liegt in einem geringeren Kommunikationsaufwand in schreibintensiven Umgebungen. Ein entscheidender Nachteil ist allerdings, dass Lesezugriffe in diesem Fall auf *alle* Kopien erfolgen müssen, um sicher zu stellen, dass die aktuellste Kopie gelesen wird.

In vielen Dankenbanksystemen sind beide Varianten auswählbar.

6.5 Serialisierung

In CP-Systemen wird Serialisierung üblicherweise hergestellt, indem Transaktionen, die zu Konflikten führen, entweder verzögert (ausschließlich bei der Verwendung von TPL) oder abgebrochen (TPL oder MVCC) werden. Das Anwendungsprogramm hat im Fall des Abbruchs dann die Möglichkeit, die Transaktion neu zu starten.

AP-Systeme verzichten auf Serialisierung. Die entsprechenden Aufgaben werden in diesem Fall durch Konvergenzstrategien abgedeckt.

7 Konsistenz und Konvergenz

Eine der Kerneigenschaften in ACID-Datenbanken ist Konsistenz. Diese wird vorwiegend durch Isolation hergestellt. Da die höchste Isolationsstufe (Serializable) jedoch mit erheblichen Performanceeinbußen verbunden wäre, wird in der Praxis mit reduzierter Isolation gearbeitet. Die Aufgabe, dies zu berücksichtigen, liegt beim Anwender.

Eine Schwierigkeit dabei besteht darin, dass selbst die einheitlich definierten Isolationslevel (Serializable, Repeatable reads, Read committed und Read uncommitted) je nach Datenbank unterschiedlich implementiert sein können. Dies erschwert nicht nur die Fehlersuche, sondern auch die Migration auf eine andere Datenbank.

Der Isolationslevel Read uncommitted ist dabei identisch mit einer vollständigen Aufgabe der Isolation, da hierbei auch Werte aus anderen, noch laufenden (und nicht notwendigerweise erfolgreich abschließenden) Transaktionen gelesen werden können.

Besondere Schwierigkeiten verursacht hierbei die in Kapitel 3.2.1 vorgestellte Snapshot Isolation. Um hier potentielle Fehler zu vermeiden, müssen alle möglichen Zugriffe in allen Anwendungen, die gleichzeitig auf die Datenbank zugreifen können, berücksichtigt werden.

Allein die Zusicherung von Konsistenz in ACID-Systemen kann dazu führen, dass der Anwender mögliche Fehlerquellen im Anwendungsprogramm übersieht.

BASE hingegen sichert nicht grundsätzlich Konsistenz zu, diese Datenbanken sind jedoch in der Lage, bestimmte Konsistenzanforderungen zu halten. In den meisten Fällen ist dies von der Implementierung und/oder von Einstellungen in der Datenbank abhängig. COPS stellt beispielsweise pro Rechenzentrum konsistente Daten zur Verfügung. Lediglich der Abgleich zwischen den Rechenzentren erfolgt mittels eines Strong Eventual Consistency Modells [COPS]. Bei Cassandra ist der Konsistenzlevel der Schreiboperationen einstellbar[1] [CASA].

Dabei bezieht sich Konsistenz hauptsächlich auf den vom Anwender wahrgenommenen Zustand der Datenbank, Konvergenz hingegen auf die Art, in der ein konsistenter Zustand hergestellt wird.

[1] Varianten: ONE, die Schreiboperation wird bestätigt, sobald ein beliebiger Knoten aktualisiert wurde; QUORUM, die Bestätigung erfolgt, sobald mehr als die Hälfte der Kopien aktualisiert wurde; ALL, die Bestätigung erfolgt erst, wenn alle Knoten aktualisiert wurden

7.1 Konsistenzmodelle

Im Folgenden werden die gebräuchlichsten Konsistenzmodelle in der Reihenfolge absteigender Stärke vorgestellt.

7.1.1 Causal Consistency (CC)

Replikationen zwischen unterschiedlichen Knoten erfolgen häufig Out-Of-Order. Gründe dafür können sowohl verlorene Nachrichten, als auch Optimierungen im Replikationscode sein. Dadurch können kausale Beziehungen zwischen einzelnen Schreibvorgängen zerstört werden. Um dies zu vermeiden, identifiziert ein System, das Causal Consistency herstellt, potentielle Abhängigkeiten zwischen Lese- und Schreibvorgängen und stellt sicher, dass die Replikation der Daten in einer Reihenfolge erfolgt, die diese Zusammenhänge berücksichtigt.

Zu diesem Zweck wird eine Relation definiert, die potentielle Zusammenhänge darstellt:

1. Sind a und b Operationen auf dem selben Wert in einem Thread, dann gilt $a \rightarrow b$ (a findet vor b statt)
2. Ist a eine Schreiboperation und b eine Leseoperation auf dem selben Datum, bei der der von a geschriebene Wert zurückgeliefert wird, dann gilt $a \rightarrow b$.
3. Die Relation ist transitiv, d.h. $\forall a, b, c[((a \rightarrow b) \wedge (b \rightarrow c))) \rightarrow (a \rightarrow c)]$

Diese Definition ist nahezu identisch der Definition der Happened-before-Relation in Kapitel 4.1. Versteht man das Schreiben und darauffolgende Lesen des selben Wertes als eine gesendete und empfangene Nachricht, dann sind die Definitionen identisch.

Bei der Replikation werden nun zusätzlich zu den eigentlichen Daten auch Informationen zur Abhängigkeit übertragen (z.B. im Fall 2 zusätzlich zum geschrieben Wert auch Name und Version des vorher gelesenen Wertes). Auf allen anderen Knoten werden die entsprechenden Schreiboperationen gepuffert und erst ausgeführt, wenn alle Abhängigkeiten erfüllt sind.

In Abbildung 7.1 ist b abhängig von a, d.h. ein Update des Wertes b darf frühestens dann erfolgen, wenn die erforderlichen Updates auf a ausgeführt wurden. Ebenso ist d von a abhängig, da das Update von d von b abhängt, das wiederum von a abhängig ist.

$$\begin{array}{lccccc} \text{Thread 1} & Read(a) & \rightarrow & Write(b) & \rightarrow & Write(c) \\ & & & \downarrow & & \\ \text{Thread 2} & & & Read(b) & \rightarrow & Write(d) \end{array}$$

Abbildung 7.1: Beispiel Causal Consistency

Diese Art der Konsistenz wird unter Anderem von COPS-GT gewährleistet[2].

Wird beispielsweise eine Website mit Newsartikeln und zugehörigem Index betrieben, kann ohne Konsistenzsicherung die Indexseite synchronisiert werden, der eigentliche Artikel liegt jedoch auf dem entsprechenden Knoten noch nicht vor. Causal Consistency stellt hier sicher, dass die Aktualisierung der Indexseite erst erfolgen kann, wenn der Artikel vorliegt, da der Index selbst vom Inhalt und Vorhandensein des Artikels abhängt.

7.1.2 Read-your-write Consistency (RC)

In Abschwächung der CC gibt RC keine Garantien für kausale Zusammenhänge. Es wird lediglich garantiert, dass nach einem Schreibvorgang eines Prozesses für das geschriebene Datum der geschriebene Wert zurückgeliefert wird. Dies gilt unabhängig davon, ob der Prozess zwischenzeitlich die Verbindung zur Datenbank beendet.

Eine Abschwächung hierzu stellt Session Consistency (SC) dar. Hierbei wird RC ausschließlich für die Dauer einer einzelnen Session garantiert.

7.2 Konvergenz durch Konfliktbereinigung

Da BASE-Datenbanken insbesondere auch gleichzeitige Schreibvorgänge auf unterschiedlichen Knoten zulassen, ist es nicht nur wichtig, solche Fälle erkennen zu können, wie dies zum Beispiel mit MVCC realisiert wird, sondern auch, diese Konflikte zu bereinigen. Dazu können unterschiedliche Strategien verwendet werden. Welche im Einzelfall sinnvoll ist, hängt dabei von der Semantik des gespeicherten Datums ab. In einigen Fällen ist es korrekt, ein Datum zu verwerfen, in anderen Fällen müssen beide Daten auf geeignete Weise miteinander verknüpft werden. Dabei ist insbesondere wichtig, dass die gewählte Strategie kovergent ist, d.h. dass unabhängig von der Reihenfolge, in der die Updates erscheinen, am Ende auf allen Knoten das selbe Ergebnis vorliegt[3].

Je nach Datenbank unterscheidet sich auch der Zeitpunkt, zu dem die Konfliktbereinigung erfolgt. Dies kann beim Zusammenfügen unterschiedlicher Fassungen (Merge)

[2]Dort wird sie als *causal+* bezeichnet.

[3]Idealerweise sollte dies auch der Fall sein, falls Updates mehrfach eingespielt werden.

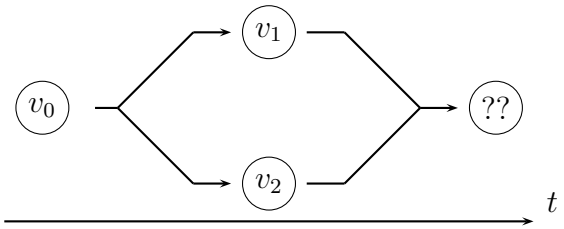

Abbildung 7.2: Merging

erfolgen, beim nächsten Lesezugriff oder unabhängig davon durch einen nebenläufigen Prozess.

7.2.1 Benutzerdefiniert

Dies ist die aus Datenbanksicht einfachste Form der Konfliktbereinigung. In diesem Fall werden die in Konflikt stehenden Versionen einer benutzerdefinierten Routine übergeben. Als Rückgabe erwartet die Datenbank das Ergebnis des Mergevorgangs.

Diese Möglichkeit der Konfliktbereinigung ist in fast allen AP-Datenbanken verfügbar.

7.2.2 Latest-Write-Wins

Sofern hinreichend synchronisierte Uhren vorhanden sind, kann eine Konfliktbereinigung erfolgen, indem das zeitlich zuletzt geschriebene Datum erhalten bleibt. Frühere Schreibvorgänge werden dabei verworfen. Diese Variante ist in fast allen Datenbanken verfügbar und in vielen die Voreinstellung.

Beispielsweise kann man in CMS-Systemen generell davon ausgehen, dass die zuletzt gemachte Änderung auch die vom Benutzer beabsichtigte finale Fassung ist. Das Verwerfen der zeitlich früheren Änderung führt daher zum korrekten Ergebnis.

7.2.3 Multi-Value-Register

Gerade die schemafreie Speicherung in NoSQL Datenbanken ermöglicht es, statt sich für einen der beiden Werte entscheiden zu müssen, beide zu speichern. In Situationen, in denen dies nicht korrekt ist, muss dann die Logik des Anwendungsprogrammes entscheiden, wie der Konflikt zu bereinigen ist. Aus Sicht der Datenbank stellt es in diesem Fall jedoch keinen Fehler dar, alle Varianten zu speichern.

Warum diese Vorgehensweise sinnvoll sein kann, lässt sich am einfachsten am Beispiel eines Terminkalenders verdeutlichen. Nehmen wir an, Alice erstellt einen Termin für 10:00 Uhr und lädt Bob und und Carol zur Teilnahme ein. Bob schlägt als neuen Termin 09:00 Uhr vor. Gleichzeitig schlägt Carol 13:00 Uhr vor. Hier ist es sinnvoll, beide Terminvorschläge zu speichern und damit Alice die Entscheidung zu überlassen, für welche Uhrzeit sie sich entscheidet.

7.3 Konvergenz durch Konfliktvermeidung

Im vorherigen Abschnitt wurden diverse Strategien zur Konfliktbereinigung vorgestellt. Eine Alternative dazu besteht darin, das Auftreten von Konflikten generell zu verhindern. Dieser Ansatz liegt konfliktfrei replizierbaren Datentypen (CRDT) zu Grunde. Damit stellen sie den vermutlich vielseitigsten und interessantesten Ansatz dar.

CRDT werden so konstruiert, dass nicht nur ein eindeutiges Ergebnis garantiert wird, sondern es kann darüber hinaus garantiert werden, dass dieses Ergebnis semantisch korrekt ist. Daher sind sie die Grundlage für Strong Eventual Consistency.

Man unterscheidet zwei Arten von CRDT:

- operationsbasierte CRDT (auch kommutativ replizierte Datentypen, CmRDT). Die Grundidee dabei ist folgende: Spielt die Reihenfolge, in der einzelne Änderungen ausgeführt werden, keine Rolle, dann genügt es bereits, wenn alle Updates jeden Knoten genau einmal erreicht haben. Jede Änderung wird also durch eine Updateoperation durchgeführt, die auf alle anderen Knoten propagiert wird.
- zustandsbasierte CRDT (auch konvergent replizierte Datentypen, CvRDT). In einer Variante davon existiert zu jedem Knoten eigenes Teildatum. Ausschließlich dieses wird vom Knoten modifiziert. Diese Änderungen werden auf alle anderen Knoten propagiert. Erst beim Lesezugriff wird aus den Teildaten das eigentliche Datum generiert. Da jeder Knoten ausschließlich sein Teildatum ändert, sind die Mengen der von unterschiedlichen Knoten modifizierten Daten immer disjunkt und es können keine Konflikte auftreten. Eine andere Variante besteht darin, lediglich momoton wachsende Mengen zu verwenden.

Während bei CmRDT häufig garantiert werden muss, dass jedes Update pro Knoten nur einmal durchgeführt wird, ist bei CvRDT die Mehrfachausführung eines Updates unkritisch.

7.3.1 operationsbasierte CRDT (CmRDT)

Sofern alle Updateoperationen auf einem Datum kommutativ sind, können operationsbasierte CRDT zur Replikation verwendet werden. Beispiele hierfür sind:

- Besuchszähler für Webseiten: Die einzige hier verwendete Operation ist ein Inkrement
- Kontostände: Die einzigen Updateoperationen hier sind Addition und Subtraktion

Um Konvergenz sicherzustellen, genügt es bereits, wenn alle Updateoperationen jeden Knoten erreicht haben. Dabei muss jedoch über geeignete Maßnahmen (Zeitstempel, Updatenummern) sichergestellt werden, dass jedes Update genau einmal ausgeführt wird, es sei denn, die Vereinigungsfunktion ist idempotent.

Am einfachsten ist das zu verstehen, wenn man betrachtet, wie das Ergebnis in den oben genannten Fällen zu Stande kommt.

Der Wert des Besuchszählers ist die Anzahl der durchgeführten Inkrement-Operationen, unabhänig von deren Reihenfolge. Wenn sichergestellt ist, dass jedes Update jeden Knoten genau einmal erreicht, dann konvergieren auch die auf den unterschiedlichen Knoten vorhanden Werte.

Ähnlich verhält es sich mit dem oben genannten Kontostand. Dieser ist eindeutig beschrieben durch die Summe der gebuchten Beträge (Last- und Gutschriften). Die Reihenfolge, in der diese durchgeführt werden, spielt für das Ergebnis keine Rolle.

7.3.2 zustandsbasierte CRDT (CvRDT)

CmRDT definieren drei Grundfunktionen:

- *query* liefert den Zustand der vorliegenden Kopie, ohne diesen zu ändern
- *update* aktualisiert den Zustand der vorliegenden Kopie unter Beachtung von Einschränkungen
- *merge* vereinigt den aktuellen Zustand mit dem einer anderen Kopie.

Folgende Eigenschaften werden für die Merge-Funktion von CmRDT gefordert:

1. kommutativ: $a \circ b = b \circ a$
2. assoziativ: $(a \circ b) \circ c = a \circ (b \circ c)$
3. idempotent: $a \circ a = a$[4]
4. monoton: Der Zustand ändert sich ausschließlich monoton steigend

Maximumbasierte CvRDT

Während bei CmRDT jedes Update an jeden Rechner gesendet werden muss, ist es bei CvRDT auch möglich, Teilergebnisse aus partiellen Updates zu einem korrekten Ergebnis zu kombinieren. Das folgende Beispiel eines monoton wachsenden Zählers soll dies illustrieren:

- Sei $Z \in N^m$ ein Vektor mit m Elementen. m entspricht dabei der Anzahl der Knoten in der verteilten Datenbank. Der Startzustand ist $Z = (0, ..., 0)$. Damit existiert für jeden Knoten ein eigenes Teildatum.
- Bei einem Query wird als Ergebnis die Summe aller Komponenten zurückgegeben, also: $q(Z) = \sum_{i=1}^{m} Z_i$. Bei Abfragen werden damit die Teildaten zum eigentlichen Datum kombiniert.
- Bei einem Update wird ausschließlich das Element des Zählers inkrementiert, das zum entsprechenden Knoten gehört, also $u(Z, i) = (Z_j + 1, \text{ falls } i = j, Z_j \text{ sonst})$. Damit sind die von unterschiedlichen Knoten modifizierten Daten disjunkt.

[4]Bei CmRDT wird dies sichergestellt, indem eine Mehrfachausführung des selben Updates verhindert wird

- Die Merge-Funktion berechnet das komponentenweise Maximum der Elemente der zu vereinigenden Vektoren X und Y, also $m(X, Y) = (\forall i : max(X_i, Y_i))$.

Dieser Zähler kann offensichtlich nur inkrementiert werden. Weiterhin ist eine neuere Version einer Komponente am größeren Eintrag im Vektor eindeutig erkennbar. Da die Änderungen der einzelnen Knoten im Vektor unabhängig voneinander sind, ist auch eine konfliktfreie Vereinigung zweier unterschiedlicher Kopien möglich. Ebenso ist der Zähler unempfindlich gegenüber der Mehrfachausführung von Updates, da die verwendete Maximum-Funktion idempotent ist.

Ein Nachteil dieses Ansatzes ist, dass sich der oben genannte Zähler nur erhöhen lässt. Eine einfache Erweiterung gestattet das Zählen in beide Richtungen:

- Seien $Z = (P, N)$, wobei P und N zwei monotone Zähler sind.
- Die Query-Funktion liefert die Differenz der Teilzähler P und N, also: $q(Z) = q(P) - q(N)$
- Die Update-Funktionen Inkrement (u_i) und Dekrement (u_d) für den Knoten x seien definiert als $u_i(Z, x) = u(P, x)$ und $u_d(Z, x) = u(N, x)$. Die Inkrement-Funktion erhöht also den Zähler P, während die Dekrement-Funktion den Zähler N erhöht.
- Die Merge-Funktion ist $m(X, Y) = (m(X_P, Y_P), m(X_N, Y_N))$. Es wird also für jeden Zähler die Merge-Funktion aus dem vorherigen Beispiel verwendet.

Damit besteht dieser Zähler aus zwei monoton wachsenden Zählern, deren Differenz den Zählerwert ergibt. Soll der Zähler erhöht werden, wird der positive Teilzähler erhöht, soll er verringert werden, wird der negative Teilzähler erhöht.

Die allgemeine Form dieser CvRDT kann man damit folgendermaßen definieren:

- Sei e ein 2-Tupel (d, t), wobei d ein Datum beliebigen Typs und t eine monoton steigende Versionsnummer ist.
- Sei $V \in (d, t)^n$ ein n-elementiger Vektor dieser Tupel, wobei n der Anzahl der Datenbankknoten entspricht.
- Als Queryfunktion kann jede beliebige Funktion verwednet werden, die aus den Teilergebnissen ein gültige Gesamtergebnis generiert.
- Die Update-Funktion für Knoten i speichert den neuen Wert d im i-ten Eintrag des Vektors und erhöht die zugehörige Versionsnummer um 1
- Die Merge-Funktion übernimmt komponentenweise den Eintrag mit dem größeren Zeitstempel.

Als Beispiel dient wieder ein Kontostand. Die Datenbank hat zwei Knoten und verwendet einen CvRDT in folgender Form:

- $e = (Kontostand, Version)$, wobei $Version$ eine Ganzzahl ist und $Kontostand$ eine Festkommazahl.
- Die Queryfunktion berechnet die Summe aller Teilkontostände.
- Die Updatefunktion ändert den zum Knoten gehörenden Wert um einen Betrag x, der sowohl positiv, als auch negativ sein kann.
- Die Merge-Funktion ist wie oben angegeben definiert.

Der anfängliche Kontostand sei $[(95, 20), (-30, 10)]$ auf beiden Knoten. Damit ergibt sich ein Guthaben von 65 auf dem Konto.

Knoten 1 bearbeitet eine Abbuchung von 50. Damit ändert sich der Kontostand zu $[(45, 21), (-30, 10)]$. Damit ist der Kontostand jetzt 45.

Gleichzeitig wird auf Knoten 2 eine Einzahlung von 100 bearbeitet, die zu folgendem Kontostand auf Knoten 2 führt: $[(95, 20), (70, 11)]$. Der Kontostand aus Sicht von Knoten 2 ist damit 165.

Die Synchronisation führt zu $[(45, 21), (70, 11)]$, also dem korrekten Kontostand von 115.

Mengenbasierte CvRDT

Die bisher vorgestellten CvRDT verwendeten die Maximumfunktion, um Monotonie sicher zu stellen. Dabei bieten sie den Vorteil, dass die Größe des Datums in der Datenbank durch die Anzahl der Datenbankknoten begrenzt ist. Dies ist jedoch gleichzeitig ein Nachteil, wenn man bedenkt, dass unter Umständen eine erhebliche Anzahl von Knoten verwendet wird, während gleichzeitig nur eine geringe Anzahl von Knoten Änderungen am Datum vornehmen wird.

Eine Alternative stellen CvRDT dar, die auf Mengenoperationen beruhen. Um das geforderte Monotoniekriterium zu erfüllen, beschränkt man sich hierbei auf die Operationen *Hinzufügen* und *Vereinigen*.

Das einfachste Beispiel hierfür ist das Grow-Set. Dieses ist wie folgt definiert:

Initialisierung		$A = \emptyset$
query	$lookup(e)$	$return(e \in A)$
update	$insert(e)$	$A = A \cup \{e\}$
merge	$R = merge(S, T)$	$R = S \cup T$

Die Merge-Funktion ist hierbei idempotent ($merge(\{1, 2\}, \{1, 2\}) = \{1, 2\}$).

Ein Nachteil des Grow-Sets ist, dass Elemente nicht entfernt werden können. Um dies zu beheben, wird das Grow-Set zum 2P-Set erweitert:

Initialisierung		$A = \emptyset, R = \emptyset$
query	$lookup(e)$	$return((e \in A) \wedge (e \notin R))$
update	$insert(e)$	$A = A \cup \{e\}$
	$delete(e)$	$if(lookup(e))\ then\ R = R \cup \{e\}$
merge	$U = merge(S, T)$	$U.A = S.A \cup T.A, U.R = S.R \cup T.R$

Ein Element e ist demnach im 2P-Set enthalten, wenn es in der Menge A, nicht jedoch in der Menge R enthalten ist. Die Insert-Operation fürgt dabei das Element der Menge A hinzu, die Delete-Operation fügt es der Menge R hinzu, sofern es in der Menge A enthalten ist. Die Merge-Operation vereinigt jeweils die A und R Mengen der zu konsolidierenden Daten. Ein Nachteil des 2P-Sets ist, das einmal gelöschte Elemente nicht wieder hinzugefügt werden können.

Hierzu ein Beispiel:

Operation	Mengen A und R	Inhalt des 2P-Sets
Ausgangszustand	$A = \{1, 2\}, R = \{2\}$	$S = \{1\}$
$insert(3)$	$A = \{1, 2, 3\}, R = \{2\}$	$S = \{1, 3\}$
$delete(4)$	$A = \{1, 2, 3\}, R = \{2\}$	$S = \{1, 3\}$
$delete(1)$	$A = \{1, 2, 3\}, R = \{1, 2\}$	$S = \{3\}$
$insert(1)$	$A = \{1, 2, 3\}, R = \{1, 2\}$	$S = \{3\}$

Das Besipiel $delete(4)$ zeigt hierbei, dass keine Änderung an den Mengen A und R erfolgt, wenn ein Element gelöscht wird, das nicht in A enthalten ist. $insert(1)$ am Ende zeigt, dass ein einmal gelöschtes Element nicht wieder hinzugefügt werden kann.

2P-Sets werden oft für Warenkörbe in Online-Shops verwendet. Dabei werden nicht nur die Artikel, sondern zusätzlich noch ein Token (Zufallszahl) gespeichert. Damit können auch gelöschte Elemente wieder eingefügt werden, sofern ein neues Token erzeugt wird.

Ein Nachteil des 2P-Sets ist (da Löschoperationen immer Vorrang haben), dass kausale Abhängigkeiten nicht berücksichtigt werden. Es ist beispielsweise möglich, dass ein Knoten ein Element löscht, während gleichzeitig ein zweiter Knoten eine davon kausal abhängige Operation ausführt. Dies kann mit dem Einsatz von OR-Sets vermieden werden.

OR steht hierbei für "observed remove", das heißt, ein Element kann nur dann gelöscht werden, wenn bereits alle Einfügeoperationen des Elements beobachtet wurden. Dabei wird ein Element auch dann eingefügt, wenn eine von diesem Element kausal abhängige Operation durchgeführt wird.

Definiert ist das OR-Set wie folgt:

Speicherformat:		$v = (e, A, R)$
Initialisierung		$S = \emptyset$
query	$lookup(e)$	$return(\exists v \in S, (v.e = e) \wedge ((v.A \backslash v.R) \neq \emptyset))$
update	$insert(e)$	$if(\neg \exists v \in S, v.e = e)\ then\ S = S \cup \{(e, \{\}, \{\})\}$
		$\exists v, v.e = e : v.A = v.A \cup \{uniqueID\}$
	$delete(e)$	$\exists v, v.e = e : v.R = v.R \cup v.A$
merge	$U = merge(S, T)$	$U = \left\{ \begin{array}{l} \forall v \in S, w \in T, v.w = w.e : \\ (v.e, v.A \cup w.A, v.R \cup w.R) \end{array} \right\}$

Statt des Datums e wird im OR-Set ein Tripel aus e und den Mengen der Einfüge- und Löschtoken (A und R) gespeichert. Die Menge dieser Tripel wird dabei mit der leeren Menge initialisiert.

Ein Element e ist dabei in der Menge S enthalten, wenn es ein Tripel (e, A, R) gibt, das e enthält und dessen A-Menge mindestens ein Element enthält, das nicht in R enthalten ist.

Ein Element wird eingefügt, wenn es der Menge S hinzugefügt wird oder wenn eine kausal davon abhängige Operation durchgeführt wird. Dazu wird ein neues Token erzeugt und dieses der A-Menge des Elements hinzugefügt. Beim erstmaligen Einfügen wird davor noch das Tripel $(e, \{\}, \{\})$ in die Menge S eingefügt.

Ein Element wird gelöscht, indem alle bekannten Einfüge-Token der Menge der Löschtoken hinzugefügt werden.

Die Merge Operation vereinigt für jedes Tripel (e, A, R) die entsprechenden A und R-Mengen.

Hierzu ein Beispiel:

Operation	Menge S	Inhalt des OR-Sets
Ausgangszustand	$\left\{ \begin{array}{l} (a, \{5,9\}, \{\}) \\ (b, \{2,3\}, \{3\}) \end{array} \right\}$	(a, b)
$insert(c)$	$\left\{ \begin{array}{l} (a, \{5,9\}, \{\}) \\ (b, \{2,3\}, \{3\}) \\ (c, \{8\}, \{\}) \end{array} \right\}$	(a, b, c)
$delete(d)$	$\left\{ \begin{array}{l} (a, \{5,9\}, \{\}) \\ (b, \{2,3\}, \{3\}) \\ (c, \{8\}, \{\}) \end{array} \right\}$	(a, b, c)
$delete(a)$	$\left\{ \begin{array}{l} (a, \{5,9\}, \{5,9\}) \\ (b, \{2,3\}, \{3\}) \\ (c, \{8\}, \{\}) \end{array} \right\}$	(b, c)
$insert(a)$	$\left\{ \begin{array}{l} (a, \{5,7,9\}, \{5,9\}) \\ (b, \{2,3\}, \{3\}) \\ (c, \{8\}, \{\}) \end{array} \right\}$	(a, b, c)

$insert(a)$ am Ende zeigt hierbei, dass im Gegensatz zu 2P-Sets auch gelöschte Elemente wieder eingefügt werden können.

7.3.3 Propagieren von Teilupdates mittels Gossip-Protokollen

Abbildung 7.3 zeigt ein komplexeres Beispiel für die Darstellung von Kontoständen. Ausgangszustand ist dabei ein konvergenter Zustand mit einem Kontostand von 150 (95-30+85). Durch Abhebungen in Höhe von 40 und 30, sowie eine Einzahlung von 100 divergiert das System. Der korrekte Kontostand nach diesen Operationen ist 180. Dieser liegt jedoch auf keinem Knoten vor (siehe Zeile "Kontostand (A)").

Danach werden zwei Updates gesendet. Das erste von Knoten 1 an Knoten 2, das zweite von Knoten 3 nach Knoten 3. Die resultierenden Kontostände sind der der Zeile "Kontostand (B)" zu sehen. Interessant hier ist, dass Knoten 3 bereits den korrekten Kontostand

Aktion	Knoten 1	Knoten 2	Knoten 3
Anfangszustand	$\begin{pmatrix}(95,20)\\(-30,10)\\(85,5)\end{pmatrix}$	$\begin{pmatrix}(95,20)\\(-30,10)\\(85,5)\end{pmatrix}$	$\begin{pmatrix}(95,20)\\(-30,10)\\(85,5)\end{pmatrix}$
Abhebung Knoten 1 (40)	$\begin{pmatrix}(55,21)\\(-30,10)\\(85,5)\end{pmatrix}$		
Einzahlung Knoten 2 (100)		$\begin{pmatrix}(95,20)\\(70,11)\\(85,5)\end{pmatrix}$	
Abhebung Knoten 3 (30)			$\begin{pmatrix}(95,20)\\(-30,10)\\(55,6)\end{pmatrix}$
Kontostand (A)	110	250	120
Update Knoten 1 → 2		$\begin{pmatrix}(55,21)\\(70,11)\\(85,5)\end{pmatrix}$	
Update Knoten 2 → 3			$\begin{pmatrix}(55,21)\\(70,11)\\(55,6)\end{pmatrix}$
Kontostand (B)	110	210	180

Abbildung 7.3: Beispiel CvRDT

zeigt, obwohl er keine Nachricht von Knoten 1 erhalten hat. Tatsächlich hat Knoten 3 das Teilergebnis von Knoten 1 von Knoten 2 erhalten, da dieses dort bereits vorlag.

Diese Möglichkeit indirekter Weitergabe von Updates ist eine der Stärken von CvRDT und ermöglicht die Verwendung von Gossip Protokollen.

Gossip bedeutet wortwörtlich Tratsch, Klatsch oder Buschfunk. Diese Begriffe beschreiben die Funktion dieser Protokolle bereits recht gut. Die Grundidee dabei ist, dass ein Knoten, der Neuigkeiten (Updates) hat, diese an eine Gruppe anderer Knoten weiterreicht. Da diese jetzt ebenfalls über neue Informationen verfügen, tratschen sie diese an andere Gruppen von Rechnern weiter. Die Menge der Knoten, die pro Tratsch-Runde über die Updates verfügen, wächst damit exponentiell. Erhält ein Knoten eine Information, die ihm bereits vorlag und die er bereits weitergereicht hatte, gibt er diese nicht mehr weiter.

Aus den Zeitstempeln der Informationen, die ein Knoten als "neu" bezeichnet, lässt sich ebenfalls schließen, welche anderen Informationen für diesen knoten neu sein könnten. Damit können Empfänger ermitteln, welche Updates auf dem Sender wahrscheinlich noch nicht vorlagen und diese weitergeben.

Im Allgemeinen versenden Gossip Protokolle immer eine geringe Anzahl von Nachrichten (z.B. 10 pro Sekunde), um anderen Knoten ihren Status mitzuteilen und um diese zu

veranlassen, Neuigkeiten zu senden.

Diese Protokolle sind nicht nur für LAN Umgebungen geeignet, sondern funktionieren auch sehr gut in Mesh-Netzwerken, in denen nicht jeder Knoten jeden erreichen kann oder in Umgebungen, in denen eine ständige Netzwerkanbindung nicht gewährleistet werden kann.

8 Fazit

Die Tatsache, dass ACID Konsistenz über Verfügbarkeit stellt, während BASE Verfügbarkeit preferiert, führt auf den ersten Blick zu der Annahme, dass diese beiden Ansätze sich gegenseitig ausschließen und Datenbanken daher nur ACID oder BASE-Systeme sein können. Tatsächlich gibt es jedoch viele Datenbanksysteme, die beide Ansätze unterstützen und sie gleichzeitig auf unterschiedichen Ebenen auch realisieren. Dies ist z.B. bei COPS-GT der Fall, wenn auf Ebene des Rechenzentrums Konsistenz und zwischen Rechenzentren Verfügbarkeit vorgezogen wird.

Der Bedarf an Datenbanken, die Verfügbarkeit sicherstellen und die Schlussfolgerung aus dem CAP Theorem, dass dies bei verteilten Datenbanken nur dann möglich ist, wenn die strikte Konsistenzanforderung von ACID-Systemen aufgegeben wird, hat zu einer neuen Welle von Datenbankkonzepten geführt, die nicht nur Verfügbarkeit sicherstellen, sondern in vielen Fällen auch performanter sind, als ACID-Systeme. Abbildung 8.1 zeigt den dabei entstehenden Trade-Off.

Die Aufgabe der strikten Forderung nach Konsistenz bedeutet jedoch nicht, dass Konsistenz nicht gewährleistet wird. Der Verzicht auf sofortige Synchronisation aller Knoten einer verteilten Datenbank führt hauptsächlich zu einer Leistungssteigerung und geringeren Antwortzeiten. Daher sind diese Datenbanken heute hauptsächlich im Bereich Big-Data zu finden.

Solange keine Kommunikationsstörung vorliegt, konvergieren BASE-Datenbanken in innerhalb weniger Sekunden. Lediglich im Fall einer Störung kann Konsistenz für eine größere Zeitspanne nicht sichergestellt werden. Diese Störungen treten jedoch nur selten auf.

Die Frage, ob im Einzelfall ein ACID oder ein BASE-System zum Einsatz kommt, lässt

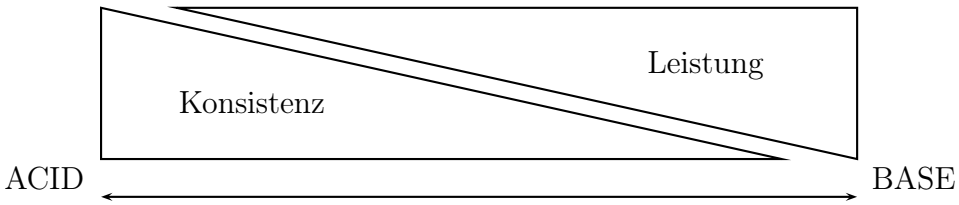

Abbildung 8.1: Spektrum ACID – Base

sich daher auf die Frage zurückführen, welche Leistung benötigt wird und welche Einschnitte in der Konsistenz dafür hingenommen werden können. Dabei müssen diese Einschnitte für den Nutzer der Datenbankanwendung nicht sichtbar sein. Vielmehr muss der Entwickler der Datenbankanwendung Vorkehrungen treffen, veraltete Daten korrekt zu behandeln.

Einer der größten Vorteile von ACID-Datenbanken ist die Standardisierung, die es dem Anwender ermöglicht, ohne großen Aufwand von einem (meist SQL basierten) System zum anderen zu wechseln. Weiterhin können erhebliche Teile des Datenmanagements in das DBMS ausgelagert werden (sortieren, suchen, indizieren, filtern, zusammenfassen).

Mit diesem Komfort sind jedoch zwei erhebliche Nachteile verbunden. Zum einen können Implementierungsdetails der DBMS zu unterschiedlichem Verhalten beim Wechsel oder der Erweiterung des DBMS führen. Zum anderen erfordert der Komfort ein erhebliches Maß an Rechenleistung, die zwar vom DBMS erbracht werden muss, vom Anwender jedoch nicht genutzt wird.

Im Gegensatz dazu sind BASE Datenbanken wesentlich diverser. Ein Wechsel von einer Datenbank zur anderen erfordert einen größeren Aufwand allein für die Einarbeitung in das entsprechende System. Dies hat jedoch gleichzeitig den Vorteil, dass der Anwender sich der Eigenschaften und möglichen Probleme des gewählten DBMS wesentlich bewusster ist.

Zwar stellen die (zumindest aktuell bestehenden) Unterschiede zwischen BASE Datenbanken ein erhebliches Hindernis für eine Migration einer Anwendung auf ein anderes DBMS dar. Dies ist jedoch, wenn auch weniger offensichtlich, auch bei ACID Datenbanken der Fall.

Die entscheidende Aufgabe bei der Entwicklung einer Datenbankanwendung für ACID Datenbanken (und meist einer der ersten Schritte beim Entwurf) besteht darin, die Abbildung der Daten in die Struktur der Datenbank festzulegen.

Bei BASE Systemen muss vorher noch festgelegt werden, welche Art der Datenorganisation mit welchem Grad an Konsistenz benötigt wird. Dies stellt zwar eine höhere Anforderung an den Entwickler dar, ermöglicht jedoch andererseits eine wesentlich größere Flexibilität sowohl in der Abbildung der Daten als auch in der Wahl der Systemperformance.

BASE und ACID stellen unterschiedliche Herangehensweisen an das Thema Datenbank dar. Beide Konzepte weisen unterschiedliche Stärken und Schwächen auf. Für den Anwender wird es entscheidend sein, diese zu kennen und je nach Anwendungsfall den geeignetsten Typ auszuwählen.

Glossar

2-Phase-Locking: Ein Verfahren zur Gewährleistung simultaner Zugriffe auf eine Datenbank mittels Sperren. Details siehe Kapitel 3.1 auf Seite 14.

2P-Set: Variante eines CvRDT, bei der Elemente hinzugefügt und gelöscht werden könne, wobei Löschoperationen immer Vorrang haben. Details siehe Kapitel 7.3.2 auf Seite 48.

ACID: Ein Satz von Anforderungen an ein DBMS, bei denen Konsistenz zu Lasten von Verfügbarkeit garantiert wird. Details siehe Kapitel 2.1 auf Seite 8.

BASE: Ein Satz von Anforderungen an ein DBMS, bei denen Verfügbarkeit zu Lasten von Konsistenz garantiert wird. Details siehe Kapitel 2.2 auf Seite 10.

CAP: Hauptsaussage des CAP Theorems: Konsistenz, Verfügbarkeit und Partitionstoleranz können nicht gleichzeitig garantiert werden. Details siehe Kapitel 1.1 auf Seite 2.

Constraint: Einschränkungen von Werten oder Werte-Tupeln, die zwingend eingehalten werden müssen. Datenbanksysteme, die die Definition von Constraints gestatten, überwachen auch deren Einhaltung. ein Beispiel hierfür stellt die in Kapitel 5.1 auf Seite 27 dar.

DBMS: siehe Datenbankmanagementsystem

Datenbankmanagementsystem: System zur Verwaltung einer Datenbank. Seine Aufgabe ist es, Daten dauerhaft und effizient zu speichern. Alle Zugriffe auf die Datenbank erfolgen über das DBMS.

Exclusive-Lock: siehe Write-Lock

Garbage-Collection: Bezeichnet die automatische Freigabe nicht mehr benötigter Speicherbereiche. Garbage-Collection ist ein Bestandteil des Storage-Managers.

Granularität: Der Wirkungsbereich eines einzelnen Locks. Details siehe Kapitel 3.1.2 auf Seite 16.

Grow-Set: Variante eines CvRDT, bei der nur Elemente hinzugefügt, jedoch nicht gelöscht werden können. Details siehe Kapitel 7.3.2 auf Seite 48.

LAN: Local Area Network. Ein Datennetz mit einer geringen geographischen Ausdehnung und üblicherweise hohem Datendurchsatz und geringen Latenzen.

Lock: Eine Sperre, die gleichzeitige Zugriffe auf das selbe Datum in der Datenbank verhindert. Details siehe Kapitel 3.1.1 auf Seite 15.

MVCC: siehe Multiversion Concurrency Control

Multiversion Concurrency Control: Ein Verfahren zur Gewährleistung simultaner Zugriffe auf eine Datenbank mittels Versionierung. Details siehe Kapitel 3.2 auf Seite 16.

NTP: siehe Network Time Protocol

Network Time Protocol: Ein netzwerkbasiertes Protokoll zur Synchronisation von Rechneruhren mit einer Referenzuhr.

NoSQL: Ein Sammelbegriff für alle Datenbanken, die nicht SQL als Hauptinterface verwenden.

OR-Set: Variante eines CvRDT, bei der nur Elemente hinzugefügt und gelöscht werden können. Kausale Abhänigkeiten können abgebildet werden. Details siehe Kapitel 7.3.2 auf Seite 49.

RDBMS: siehe relationales DBMS

Read-Lock: Eine Sperre, die den Lesezugriff auf ein Datum sichert. Details siehe Kapitel 3.1.1 auf Seite 15.

Read-Modify-Write: bezeichnet eine Änderung eines Datum, bei der der geänderte Wert in einem neuen Speicherbereich abgelegt wird. Der alte Speicherbereich wird zu einem späteren Zeitpunkt freigegeben.

SEC: siehe Strong Eventual Consistency

SI: siehe Snapshot-Isolation

SM: siehe Storage-Manager

SQL: Eine Datenbanksprache zur Definition von Datenstrukturen und Manipulation von Daten (einfügen, ändern, suchen, löschen, zusammenfassen, etc.) für RDBMS

Shared-Lock: siehe Read-Lock

Snapshot-Isolation: Bezeichnet ein von MVCC abgeleitetes Isolationsverfahren für Datenbanken. Details siehe Kapitel 3.2.1 auf Seite 18.

Sperre: siehe Lock

Storage-Manager: Verwaltet den Datenspeicher eines DBMS. Auch Speichermanager genannt. Details siehe Kapitel 6 auf Seite 35.

Strong Eventual Consistency: Die Garantie, dass ein verteiltes Datenbanksystem in einen korrekten Zustand konvergiert. Details siehe Kapitel 2.2.1 auf Seite 11.

TM: siehe Transaktions-Manager

TPL: siehe 2-Phase-Locking

Transaktion: In ACID-Datenbanken eine Folge von Anweisungen, die entweder komplett oder gar nicht ausgeführt werden.

Transaktions-Manager: Führt einzelne Transaktionen auf einer Datenbank aus. Details siehe Kapitel 6 auf Seite 35.

WAN: Wide Area Network. Ein geographisch weit reichendes Netzwerk mit im Vergleich zum LAN hohe Latenzen und geringem Datendurchsatz. Das bekannteste WAN ist das Internet.

Write-Lock: Eine Sperre, die den Schreibzugriff auf ein Datum sichert. Details siehe Kapitel 3.1.1 auf Seite 15.

relationales DBMS: Ein DBMS, in dem Daten üblicherweise in Form von Tabellen und Relationen zwischen diesen tabellen dargestellt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[WP01] https://en.wikipedia.org/wiki/ACID (01.11.2015)

[JG01] http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf (01.11.2015)

[WP02] https://en.wikipedia.org/wiki/Atomicity_%28database_systems%29 (01.11.2015)

[WP03] https://en.wikipedia.org/wiki/Consistency_%28database_systems%29 (01.11.2015)

[WP04] https://en.wikipedia.org/wiki/Isolation_%28database_systems%29 (01.11.2015)

[WP05] https://en.wikipedia.org/wiki/Durability_%28database_systems%29 (01.11.2015)

[WP06] https://en.wikipedia.org/wiki/Two-phase_commit_protocol (07.11.2015)

[WP07] https://en.wikipedia.org/wiki/Three-phase_commit_protocol (07.11.2015)

[GiLy] http://webpages.cs.luc.edu/p̃ld/353/gilbert_lynch_brewer_proof.pdf (01.11.2015)

[PCZM] http://drkp.net/papers/txcache-osdi10.pdf (01.11.2015)

[HR83] http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/HaerderReuter83.pdf (01.11.2015)

[LP78] http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf (10.11.2015)

[FI88] http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf (10.11.2015)

[WP08] https://en.wikipedia.org/wiki/Vector_clock (10.11.2015)

[BDB1] http://doc.gnu-darwin.org/intro/dbisnot.html (11.11.2015)

[NSQL] http://nosql-databases.org/ (11.11.2015)

[WCST] http://nosqlguide.com/column-store/nosql-databases-explained-wide-column-stores/ (11.11.2015)

[CaRF] https://courses.cs.washington.edu/courses/cse444/08au/544M/READING-LIST/fekete-sigmod2008.pdf (14.11.2015)

[BGYZ] http://research.microsoft.com/pubs/201602/replDataTypesPOPL13-complete.pdf (15.11.2015)

[ADH1] https://gnunet.org/sites/default/files/gossip-podc05.pdf (15.11.2015)

[COPS] https://www.cs.cmu.edu/ dga/papers/cops-sosp2011.pdf (15.11.2015)

[LAPA] http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf (15.11.2015)

[CASA] http://blog.octo.com/en/nosql-lets-play-with-cassandra-part-13/ (15.11.2015)

[FA09] http://www.inf.uni-konstanz.de/dbis/teaching/ss09/tx/Sebastian.pdf (16.11.2015)

[PODC] https://www.cs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf (12.11.2015)

[PLBD] http://bmcessen.de/wp-content/uploads/2013/06/PL-Busines-DSL-profi-SDSL-13GK01.pdf (29.11.2015)