Synchronisation von Binärbaum-indexierten, verteilten InMemory-NoSQL-Datenbanken

Abschlussarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

1. Prüfer: Prof. Dr.-Ing. Hendrik Gärtner
2. Prüfer: Jens-Peter Haack

Titel Adademischer Grad

Eingereicht von Paul Kitt

1. September 2014

Inhaltsverzeichnis

Abkürzungsverzeichniss:

EB-Baum Elastische Binär Baum

ID Identifikationsnummer

Verzeichnis aller Codebeispiele

1 Einleitung

Hintergrund, größerer Rahmen, kurze Aufgabenstellung

1. Problemstellung und Motivation
2. Zielsetzung
3. Rahmen und Aufbau der Arbeit

Mögliche Punkte: -> Motivation -> Aufgabenbeschreibung -> Inhalt und Aufbau der Arbeit

2 Grundlagen

theoretische Grundlagen, Beschreibung von Systemen(nur insoweit, als das diese Grundlagen und Beschreibungen unbedingt für das Verständnis erforderlich sind und nicht als bei studierten Informatikern vorausgesetzt werden kann)

1. Grundlagen verwendeter Algorithmen
   1. Binär Bäume
   2. Elastische Binär Bäume
2. Definition von Begriffen
   1. Verteilung
   2. Synchronisation
   3. NoSql -> CAP, ACID theoreme die meine db erfüllt
3. Grundlagen Datenbanken (Verteilung, Synchronisation)
   1. MySql
   2. NoSql
   3. Baumbasierter Indexe

2.1 Elastische Binär Bäume

Bei den „Elastischen Binär Bäumen“ handelt es sich um speziell optimierte Variante der „Binären Suchbäume“. Entwickelt wurden das Konzept von Willy Tarreau[1] im Rahmen einer Forschung zum Thema „Event-scheduling for user-space network applications“ und eignen sich daher auch speziell für Betriebssystem Scheduler, bei welchen schnelles priorisieren nach Zeit oder Dringlichkeit wichtig ist. Daten die mit Binär- oder Ganzahlen, wie Integer oder Long, indexiert sind können in dieser wenig bekannten Datenstruktur sehr effizient verwaltet werden. Dabei ist der „EB-Baum“ sehr performant wenn es zu sehr vielen den, Baum verändernden, Operationen, wie: Einfügen, Ändern, Abfragen oder Löschen kommt. Einfügeoperationen und das Abfragen von Blättern wird in O(log n) bewältigt. Löschen in O(1).

wie richtig zitieren?

Als Ausgangskonzepte dienten bei seiner Entwicklung der „balanciertem Binär Baum“und des „Radix Baumes“. Da aber Operationen wie Löschen von Blättern bei einem „balanciertem Binär Baum“ O(log n) kosten und der Baum aber gerade Operationen wie diese möglichst schnell bewältigen soll ein signifikanter Nachteil. Bei den „Radix Bäumen“, die sich laut dem Entwickler Willy Tarreau[2] im Bezug auf Geschwindigkeit sehr gut eigenen, wird aber im Betrieb das allo-

mention red black tree

[1] Tarreau, *Elastic Binary Trees - ebtree*.
[2] Tarreau, *Elastic Binary Trees - ebtree*, Absatz Introduction.

cieren von Speicher und die damit verbundene „Garbage Collection“ zum Perfomanceproblem.

Daher ist der EB-Baum eine hybride Form aus beiden um diese Mängel aus zu gleichen. Es handelt sich nicht um einen balancierten Baum, was in besonderen Fällen zu einer schlechteren Leistung als die eines „balanciertem Binär Baum“ führt. Die Blätter sind im Baum von links nach rechts aufsteigend, nach einer Binär- oder Ganzahl(e.g. Integer, Long) sortiert.

mention example

Eine weitere Besonderheit des EB-Baum ist dass seine maximale Höhe durch den Datentyp der Schlüssel bestimmt wird. So kann beispielsweise ein Baum dessen Schlüssel Werte des Datentypes „Long“ sind maximal 64 Ebenen besitzen da der Datentyp aus 64 Bit besteht. Schlüssel werden durch die Abfolge ihrer Binären Repräsentation adressiert. Der Schlüssel dient hierbei wie eine Art binäre Karte. Begonnen am höchsten gesetzten Bits des Schlüssels und wird der Baum Knoten für Knoten durch laufen und an den Knoten bei einer Null links, bei einer Eins rechts abgebogen bis dass Blatt oder falls dieses nicht vorhanden das am naheliegendste gefunden ist

2.1.1 Besonderheiten des Baumes

Durch die besondere Beschaffenheit des Baumes lassen sich viele Operationen sehr leicht umsetzen wie:

- **Abfragen**
  - das Abfragen des ersten und letzten Schlüssels
  - erlangen des nächst kleineren oder größerem Schlüssel zu einem gegeben Schlüssel
  - genaues Finden eine Schlüssels
  - das Finden des ähnlichsten Schlüssels falls dieser nicht enthalten ist
  - einfaches Abfragen von Bereichen durch ein Prefix
  - erlangen des vorherigem oder nächstem unterschiedlichen Schlüssel zu einem gegeben Schlüssel
  - mehrfach auftretende Schlüssel werden immer in ihrer eingefügten Reihenfolge zurück gegeben
- **Einfügen**
  - Einfügen mit Duplikaten: Falls ein Werte bereits existiert wird ein Duplikat angelegt
  - Einfügen von nur einzigartigen Schlüsseln: Falls der Schlüssel vorhanden wird Existierende zurück gegeben

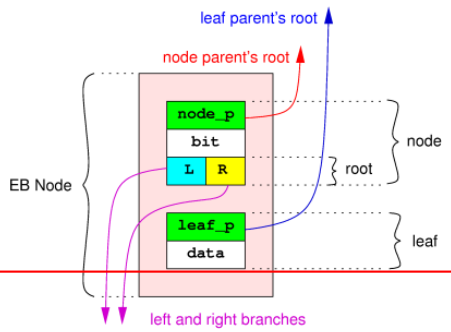
2.1.2 Aufbau des Baumes

In dem originalem von Willy Tarreau verfassten Konzept[3] werden die Daten, die Baum hält, in „EB Knoten“ gespeichert. „EB Knoten“ bestehen aus zwei Teilen:

[3] Tarreau, *Elastic Binary Trees - ebtree*, Absatz Definitions.

- Knoten: verknüpft Blätter sowie anderen Knoten
- Blatt: ist durch einen Schlüssel adressiert, hält Daten e.g. Referenz auf ein Datenobjekt

Die Aufgabe eines Blattes ist sehr simpel. Es dient nur dazu eingefügte Daten durch einen Pointer zuhalten. Dabei ist es mit einem Schlüssel adressiert und besitzt eine Referenz auf seinen Elternknoten, sowie auf seinen EB-Ursprungsknoten.



SPW fragen

Abbildung 2.1: EB-Knotenelement

Ein jedes Knotenelement besteht aus einer Referenz auf sein höher liegenden Elternknoten, dem Ebenenbit und zwei Referenzen auf seine Kinderknoten.

Darf ich das Bild benutzen?

Das Ebenenbit ist eine Zahl und repräsentiert binär die niedrigste Bitposition des Schlüssels über dem alle Bits gleich sind. Alle unter dem Knoten liegenden Schlüssel haben sich somit bis zu diesem Bit die gleich Bitfolge. Der dritte Teil des Knotens ist die Wurzel die aus zwei Referenzen besteht zu entweder weiteren Knoten oder/sowie Blättern verweist. Da es sich um einen binären Baum handelt repräsentiert je Referenz das linke Bit 0 oder das rechte 1. Nur auf Ebene 0, mit dies repräsentierendem Ebenenbit, kann der Knoten zwei Blätter als Wurzel besitzen, da der Schlüssel sich nur um einen Wert an der letzten, nullten Bitstelle unterscheiden kann.
Auch wenn Knoten sich später verschieben können diese nie unterhalb des mit ihnen angefügtem Blattes oder in einen anderen Zweig des Baumes gelangen. Falls es erlaubt ist Duplikate einzufügen werden Knoten mit negativen Ebenenbits versehen.

Eine weitere Besonderheit hier ist dass die Wurzelreferenzen nie unbelegt sind. Nur die Wurzel des Wurzelknoten des Baumes ist hier eine Ausnahme. Wenn seine zwei Wurzeln auf „null“ referenziert ist der Baum leer. Sobald der Baum befüllt wird wächst dieser an seiner linken/„0-Wurzel“. Die rechte Wurzel referenziert immer auf „null“. Dadurch lässt sich der der Wurzelknoten einfach erkennen. Sein Ebenenbit ist die höchste Ebene des Baumes und daher automatisch die binäre Länge des Schlüsseldatentyps, e.g. bei Schlüsseln des Datentyps Long ist das Ebenenbit 64.

2.1.3 Einfügen von Daten

Beim einfügen neuer Datensätze wird jeweils ein Blatt, das die Daten hält, sowie ein Knoten mit dem dass Blatt in Kombination eingefügt wird, eingefügt. Der einzige Fall in dem ein Blatt ohne Knoten angefügt wird ist wenn der Baum leer ist.

Beim Einfügen eines neuen Datensatzes wird ein gesamter „EB-Knoten“ erstellt und an der richtigen Position eingefügt. Werden später weitere Werte gespeichert wird bei deren sortiertem einfügen der Baum umstrukturiert. Dies hat zur Folge dass Knoten und Blatt eines vorher

übereinander liegenden „EB-Knoten“ auseinander gezogen werden. Da sie aber noch durch gegenseitige Referenzenverbunden bleiben wird der Baum als elastisch bezeichnet was zum Namen „EB-Baum“ führt.

logisch

Die enge Verknüpfung von Knoten und Blatt ist aber nicht zwingend notwendig. In einer, in Java implementierten, Umsetzung von Spinning Wheel, auf der später in dieser Arbeit eingegangen wird, wird ebenfalls beim Einfügen neuer Daten zu jedem Blatt ein Knoten erstellt aber später wenn der Baum sich verändert ist die Verbindung zwischen Knoten und Blatt nicht mehr nachvollziehbar.

3 Anforderungsanalyse und Vorgehen

Bewertung von theoretischen Ansätzen, Konzepten, Methoden, Verfahren; informelle Aufgabenbeschreibung, klar formulierte Zielstellung

1. Anwendungsumgebung
2. Anforderungen und Szenarios

-> Vortrag

3.1 Betriebliches Umfeld

Diese von mir verfasst Arbeit entstand in enger Kooperation mit der neue entstehenden Spinning Wheel GmbH. Das hier implementierte und getestete Datenbankteilkonzept ist ein kleiner Bestandteil eines großen Softwareprojektes mit dessen Entwicklung die Firma sich beschäftigt. So wurde ich mit Idee und Grundkonzept beauftragt und bei deren Umsetzung von Herrn Jens-Peter Haack und Gernot Sänger bei theoretischen Fragen unterstützt.

Die Spinning Wheel GmbH befasst sich mit der Entwicklung neuer Softwarelösungen für das Backend von Mobilfunkinfrastruktur. Dabei steht das verarbeiten und speichern von Mobilfunksubscriberdaten durch neue technische Möglichkeiten im Vordergrund.

3.2 Thematische Abgrenzung

Die in dieser Arbeit beschriebenen, umgesetzten und getesteten Programmkomponenten sind ein kleiner Teil eines großen neuen, verteilten Datenbankkonzeptes der Spinning Wheel GmbH und beschränken sich auf das im Speicher halten von Daten zur Laufzeit, sowie die Synchronisation zwischen Datenbanken zum Ausgleich verpasster Änderungen.

Alle weiteren Bestandteile einer Datenbank wie das Persistieren, Verarbeiten oder komplexe Abfragen der gespeicherten Daten sind nicht Teil dieser Arbeit und werden nicht oder nur am Rande behandelt.

3.3 •

andere datenbanken synch probleme
-> mysql cluster, Master slave lösung
-> NoSql: redis cluster

3.4 Problemstellung

Redundaz in verschiedenen Datenbanken secundary key probs Datenbank offline anderer Stand Datenbanken auf beiden Seiten verschieden
Packetverlust/verspätung

3.5 nicht funktionale Anforderungen

ein gewisser fehlergrad sind tolerierbar solange sie irgendwann behoben wird geschwindigkeit wichtiger daher senden der Pakete mit udp statt tcp fire and forget Dynamischer synchroansatz vs statischer komplettsynchro Synchronisation Fehlertolerant -> geht synchropaket verloren nie ein problom für den zustand der db da db dann nicht inkonsitent und bei nächsten synchronisation wird wieder der gleiche fehler gefunden

-> konzpt let it fail fehlertolerant -> Selbsheilender process

einfaches erstellen von neuen indexen

Datenbanksystem von ausen transparent

Horizontale Skalierung (scale out) Vertikale Skalierung (scale up)

ausfallsicherheit

verteiltes system

geschwindigkeit -> in memory

loadbalancing

nachteil: hoher aufwand an hardware, koordination des szstems, schwierig gut performance daher udp -> prob synchro -> synchro anforderungen

3.6 funktionale Anforderungen

einfügen updaten dbsystem löschen von datensatz synchron synchronisation Daher gilt es eine Synchronisationsmethodik zu finden die zum einen in der Lage ist schnell kleine Unterschiede als auch in einer längeren Zeitspanne Datenbanken, die sich auf völlig verschiedenen Ständen befinden, auszugleichen.

3.7 Zielstellung

Datenbanksystem aus mehreren redundanten Datenbanken mit einerfacher synchronisations Funktionalität

verschiedene datenbank zellen fuer load balancing baum indexe

Vorteile

– erhöhte Verfügbarkeit der Daten

– Beschleunigung von Lesezugriffen (bessere Antwortzeiten, Kommunikationseinsparungen)

– erhöhte Möglichkeiten zur Lastbalancierung / Query-Optimierung

Nachteile
– hoher Update-Aufwand
– erhöhter Speicherplatzbedarf
– erhöhte Systemkomplexität

-> simulation der spinning wheel datenbank
-> gesamte Idee testen
-> test wieviel sync ops bei welcher delay und fehlerrate Ziel ist es zu bestimmen wie viele SyncOps/s nötig um die Datenbanken unter einer Schranke an Unterschied bei bestimmter Last bei bestimmter Fehlerrate

3.8 Vorgehen

-> Eingrenzung auf wenige bestandteile der datenbank(halten uIDs, cID, synchro) -> Implementierung der Basis Klassen und Funktionalitaeten -> Implementierung der Simulations und Evaluations elemente -> Definieren von Tests und deren Analyse
Technisches Umfeld: (-> Scala mit Akka)
Anforderungen: ?

3.9 Testdesign

-> Test id länge node chnage id error -> Entwurf der Testdaten -> Testdesign -> test eines use cases -> Unittests: grobe Funktionstests

4 Konzept alias Definition/Entwurf

Definition: formale Darstellung der Anforderung mit Hilfe geeigneter Methoden

Entwurf: Diskussion von Lösungsansätzen, Modellierung der konzipierten Lösung

4.1 Konzept des Datenbanksystem

Um den in der Anfordrungsanalyse beschrieben Kriterien zu erfüllen ist das Datenbanksystem in verschiedene Komponenten gegliedert. Das klare Aufteilung der einzelnen Funktionen des Systems in Komponenten soll den Anspruch an Ausfallsicherheit und horizontaler, wie vertikaler Skalierung ermöglichen. So werden die Daten die das System persitieren soll in mehreren Zellen redundant abgelegt. Die Anzahl der Zellen bestimmt die Ausfallsicherheit des System. So kann mit mehr Zellen eine höhere Ausfallsicherheit gewährleistet werden aber gleichzeitig steigt damit auch die Kosten, der Aufwand das System zu betreuen und die Komplexität sowie die Koordination von System internen Ablaufen. Durch die Verwendung separater Hardware Ressourcen und einer Aufteilung in räumlich getrennten Standorten macht macht das System zudem Toleranz gegen über Hardwareausfällen oder nicht System internen Problemen wie z.B. Ausfall der Stromversorgung. Ein Partitionieren der Daten in kleine Einheiten in den Datenbankzellen wird in dieser Arbeit nicht weiter untersucht ist aber eine sehr gute Möglichkeit das System zu optimieren. So ist damit eine horizontale Skalierung des Datenbanksystem möglich da durch Unterteilung der Daten in der Zelle diese einfach auf mehrere Resourcen verteilt werden können. Durch eine Optimierung der Hardwareressourcen der Zelle oder ihrer Teile kann das System zudem auch vertikal noch oben Skaliert werden. Durch die Aufteilung der Daten auf verschiedene Instanzen ist außerdem ein verteilen der Last auf diese möglich. Dies wird durch eine Datenbankmanagmentkomponente koordiniert. Das Datenbanksystem ist nach Außen transparent. Das bedeute das alle internen Vorgänge sowie Aufbau nach Außsen hin nicht sichtbar sind. Das System liefert ein Reihe an von außen aufrufbarer Funktionen wie diese aber durch intern Vorgänge bewältigt werden oder wie der Aufbau des Datenbanksystem konzipiert ist ist nach außen hin nicht ersichtlich. Eine weitere essentielle Komponente ist der „Accesslayer“. In diesem werden für neue Daten oder Veränderungen eindeutige Identifikationsnummer (ID)s erzeugt. Jedem neuen Datensatz wird eine eindeutige Nummer zugewiesen die ihn adressiert sowie eine Nummer die seine Stand wieder gibt. Neue IDs steigen mit jeder weiterem Generierung. wird ein neuer Datensatz eingefügt oder verändert sorgt der „Accesslayer“ dafür das dieser adressiert sowie an alle Datenbankzellen gesandt wird.Um auch hier Ausfallsicherheit und eine Balancierung von Last zu ermöglichen ist es möglich das mehrere „Accesslayer“ parallel arbeiten. Die Koordination dieser ist eine weitere nicht triviale Aufgabe wird aber nicht weiter im Rahmen dieser Arbeit betrachtet.

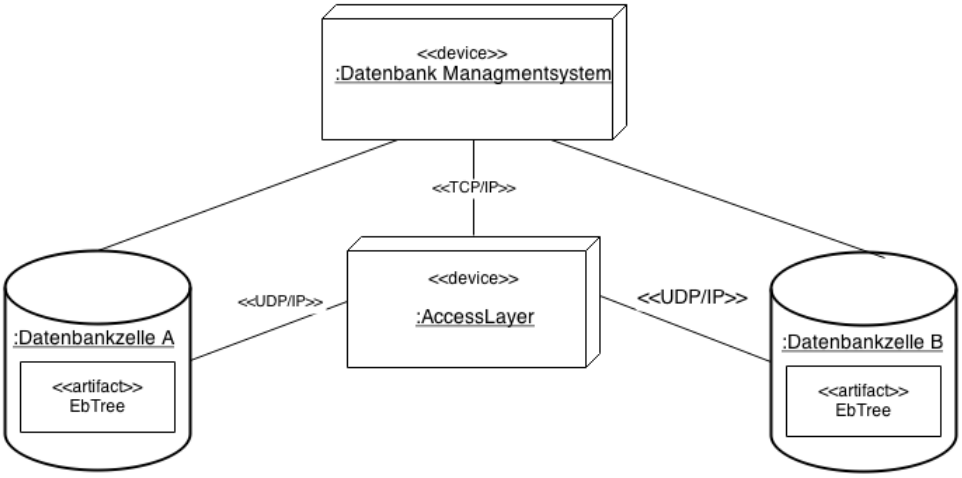


Abbildung 4.1: Synchronisationsprozess

Hinleitung zu Synchrobedarf da udp übertragung wegen performance

Datenbanken -> in memory Datenbanken system transparent uml: Komponenten diagramm Wo beschreibung der gundlegenden funktionen grundlegende konzept mehrer Datenbanken mit gleichen Daten -> bedarf von synchronisation damit Daten konsistent bleiben

4.2 Konzept der Datenbank

Die Grundlegende Idee der Datenbank ist, nicht wie in relationalen Datenbanken Daten durch Tabellen zu gruppieren, sondern durch viele verschiedene Indexe Daten je nach benötigten Kriterien zugänglich zu machen. So kann jederzeit durch bilden von neuen Indexen auf neue Anforderungen an die Datenbank reagiert werden.

Neue Datensätze erhalten bei ihrer zentralen Erzeugung im „Access Layer“ eine eindeutige, aufsteigende, ID. Die Nummer ist in allen Datenbanken gleich, verändert sich nie und adressiert den Datensatz bis dieser gelöscht wird. Eine weitere Nummer, die „Change ID“, symbolisiert den Stand des Datums. Wenn das Datum angelegt wird sind Identifikationsnummer und „Change ID“ identisch. Wird der Datensatz verändert wird diesem eine immer größer werdende „Change ID“ zu gewiesen. Somit lässt sich leicht am Wert der „Change ID“ erkennen ob Daten verändert wurden oder sich noch ihrem Ursprungszustand befinden. Besonders praktisch ist dies beim Vergleichen von Datensätzen in unterschiedlichen Datenbanken. Durch vergleichen der „Change ID“ kann so sehr schnell ermittelt werden welche von Beiden aktueller ist und der Unterschied ausgeglichen werden.

Für die ID, als auch „Change ID“, wird eine eigener Index gepflegt. Hierfür wird der sich besonders gut dafür eignende, in Abschnitt 2.1 beschriebene, EB-Baum verwendet, da dieser auf das nach Größe sortiertem Verwalten von Ganzzahlschlüsseln optimiert ist. In die Datenbank eingefügte Datensätze werden in einem Container, dem „EbTreeDataObjekt“, gekapselt. Dieses

enthält zum einen die ID, die aktuelle „Change ID“ sowie eine Referenz auf die Daten. Wird nun in die Datenbank ein Datensatz eingefügt wird dieser mit der ID in den dazu gehörigen „ID-Baum“ und mit der „Change ID“ in dem „Change ID-Baum“ eingefügt. Das Blatt jedes Baumes hält somit die entsprechende Nummer und den Container. Es lässt sich somit leicht auflösen welche „Change ID“ mit welcher ID und umgekehrt verbunden ist.

Beim Verändern eines Datensatzes wird die alte „Change ID“ aus dem „Change ID-Baum“ gelöscht und die neue „Change ID“ wieder eingefügt. Da der EB-Baum seine Schlüssel der Größe nach von links nach rechts im Baum sortiert und neu erstellte Schlüssel immer größer werden. Befinden sich neu eingefügte Elemente im „ID-Baum“ sowie neu veränderte Elemente im „Change ID-Baum“ ganz rechts in der Baumstruktur.

Soll ein Datum gelöscht werden werden nicht einfach die damit verknüpften Nummern und der Datensatz gelöscht. Beim Löschen muss sichergestellt werden dass das Datum aus allen Datenbanken entfernt wird da dieses sonst in der nachfolgend beschriebenen Synchronisationsprozess wieder hergestellt wird. Falls der Datensatz auf einer Seite der an Synchronisation beteiligten Datenbanken fehlt und auf der anderen Vorhanden ist lässt sich nicht feststellen ob dieses Ungleichgewicht durch ein verlorenes Einfügen oder Löschen zu Stande gekommen ist und ob das Element auf der einen Seite gelöscht oder auf der anderen wieder hergestellt werden soll. Daher werden nur die Daten des „EbTreeDataObjekt“ gelöscht und eine neue „Change ID“ eingefügt. Das Vollständige Löschen von ID und „Change ID“ aus beiden Bäumen sowie des „EbTreeDataObjekt“ erfolgt in einem gesonderten Löschprozess der sicherstellt das diese Veränderung an jeder Datenbank vorgenommen wird.

4.3 Synchronisation zwischen Datenbanken

Durch ein Vergleichen der IDs und „Change IDs“ der Datensätze in den sich synchronisierenden Datenbanken kann einfach festgestellt werden welche Daten fehlen und falls diese vorhanden auf welcher Seite diese aktueller sind. Ein Vergleichen der gesamten Datenbanken jedoch, gerade wenn diese unter hoher Last stehen und sehr viele Datensätze verwalten, verbraucht viele Ressourcen. Ab einem gewissen Grad von Last durch Operationen und Masse an Daten ist dies nur schwer möglich. Ein komplettes Abgleichen nimmt in diesem Fall so viel Zeit in Anspruch dass durch neue Operationen wieder ein Ungleichgewicht zwischen den Datenbanken entstehen wird.

Daher gilt es eine Synchronisationsmethodik zu finden die zum einen in der Lage ist schnell kleine Unterschiede als auch in einer längeren Zeitspanne Datenbanken, die sich auf völlig verschiedenen Ständen befinden, auszugleichen. Die „Change IDs“, die zum Vergleich während der Synchronisation gepflegt werden, werden in jeder Datenbank in einem EB-Baum gehalten. Die Möglichkeit des Vergleichen von Zweigen unterhalb jedes Knotens ist daher von großem Vorteil. Um dies Möglich zu machen wird für jeden Knoten eine den Zustand seiner beiden Kindobjekte repräsentierende Nummer, die „Node Change ID“, errechnet. Dazu werden die Nummern der Kindobjekte, „Change IDs“ falls das Kind ein Blatt oder „Node Change ID“ falls das Kind ein

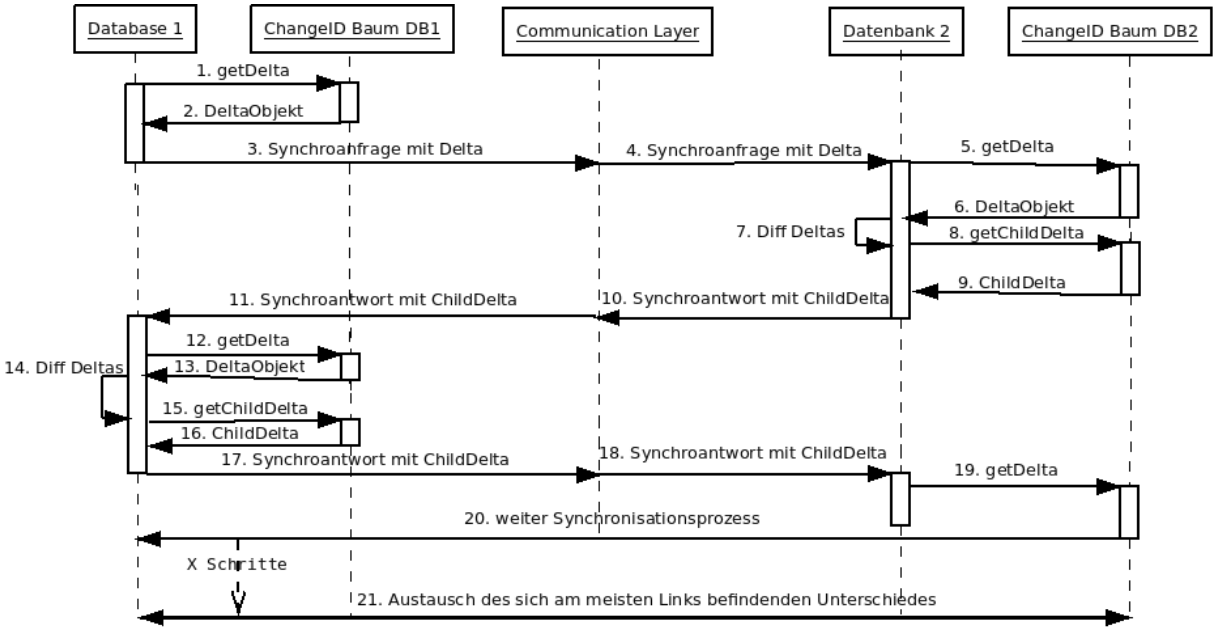


Abbildung 4.2: Synchronisationsprozess

Knoten ist, durch eine „binäre XOR Operation“ verbunden. Während des Synchronisationsprozesses senden sich beide Seiten sogenannte „Deltas“. Es werden eine die Position des Deltaknoten beschreibenen Ganzzahl, die Bitebene des Deltaknoten, der Bitebene des sich ber dem Deltaknoten befindenden Elternknoten, sowie die deren Zustand wieder gebenden Nummern seiner Kinder. Die übertragenen Werte bilden beschreiben das logisch zusammen hängende Dreieck aus Knoten und zwei Kindern. Daher wird es als „Delta“ bezeichnet.

Durch die die Position des Deltas beschreibende Ganzzahl kann der Knoten mit dem das Delta verglichen werden soll in der anderen Datenbank gefunden werden. Bei der Lokalisierung des zu vergleichenden Knoten wird am obersten Knoten des Baumes begonnen. Je nach dem ob 0 oder 1 in der Adresszahl an der binären Stelle der Bitposition des zu durchlaufenden Knoten gesetzt ist wird das linke oder rechte Kindobjekt weiter behandelt. Dies wird solange wiederholt bis die im Delta übergebene Bitebene des Deltaelternknoten unterschritten und der sich dort befindliche zu vergleichende Knoten ermittelt ist. Wird nach dem Vergleichen der Deltas festgestellt dass die„Node Change IDs“ der linken Knoten gleich sind und die der rechten Knoten ungleich. Soll ab jetzt dieser Knoten nach rechts durchlaufen werden. Dazu wird in der binären Representation des Adresswertes an der Bitstelle die der Knoten darstellt eine 1 gesetzt und von nun an bei allen weiteren Durchläufen dieses Synchronisationsprozesses an diesem Knoten das rechte Kindobjekt

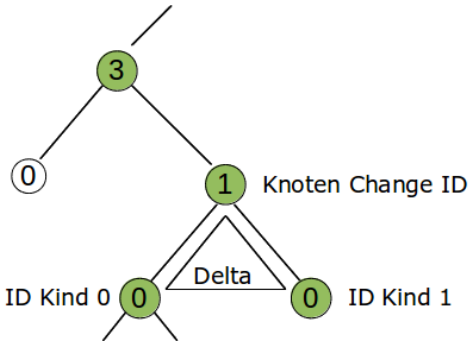


Abbildung 4.3: Deltaobjekt

behandelt. Die Synchronisation kann von jeder Datenbank angestoßen werden. Eine Möglichkeit dies zu Koordinieren ist dass jede Datenbank zufällig in einem festgelegtem Zeitintervall zu synchronisieren beginnt. Es ist Möglich das sich die Datenbank während des Synchronisationsprozess verändert. Die Synchronisation besteht aus zwei Phasen.

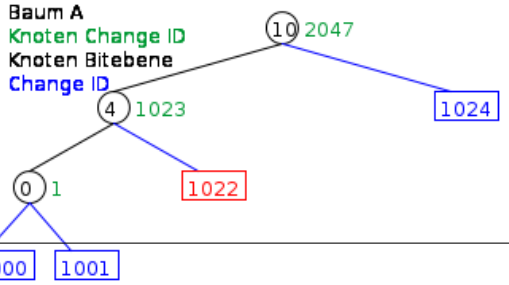
4.3.1 Synchronisation Phase 1

In der Ersten Phase gilt es den Unterschied zu finden der sich in beiden Bäumen am weitesten links befindet und somit die älteste Unterschied zwischen den Datenbanken darstellt. Da im laufenden Betrieb auf der rechten Seite des Baumes viele Veränderung statt findet soll immer der Unterschied der davon am weitesten Entfernt liegt zu erst behoben werden. Daten die sich oft Verändern sind sehr weit rechts im Baum zu finden und da somit besteht die Chance dass sich mögliche verlorene Veränderungen bei einer erneuten Veränderung selbst ausgleichen. Die mit der Synchronisation beginnende Datenbank sendet ein Delta des höchsten Knoten gekapselt in einer Synchronisationsanfrage an die andere Datenbank. Diese lokalisiert den Knoten mit dem das Delta abgeglichen werden soll und vergleicht die Nummer des linken Kindobjektes des erhaltenen Delta mit dem linken Kindobjekt des eigenen Knotens. Sind diese gleich werden die rechten Kindobjekte verglichen. Wird dabei kein Unterschied erkannt sind die Datenbanken synchron und der Synchronisationsvorgang ist abgeschlossen. Falls aber ein Unterschied lokalisiert werden kann wird mit einem Delta bestehend aus dem sich unterscheidenden Kindobjektes geantwortet. Die sich synchronisierenden Datenbanken spielen mit dem Senden der Deltas eine Art von „Ping-Pong“ bis der am weiten links liegende Unterschied gefunden ist.

Beim Abgleichen der Deltas kommt es zu unterschiedlichen Vergleichsfällen die vom Synchronisationsalgorithmus separat behandelt werden müssen:

Unterschied Links/Rechts

Die Knotenstruktur in den „Change ID-Bäumen“ beider Datenbanken ist bis zum ersten Unterschied identisch. Der Synchronisationsalgorithmus läuft Delta für Delta die Bäume entlang und navigiert dabei je nach gefundenem Unterschied nach Links oder Rechts bis ein Blatt gefunden wird. Es besteht aber nun die Möglichkeit das an der Stelle wo das Blatt gefunden wurde sich im anderen Baum ein Zweig befindet in dem das Blatt bereits vorhanden ist, Abbildung 4.5. Beim Vergleichen der „Change IDs“ wird ein Unterschied festgestellt doch dieser befindet sich im Unterzweig der anderen Datenbank. Bevor dieser aber lokalisiert werden kann wird in der Datenbank ohne den Unterzweig bereits dass Blatt gefunden. Daher wird sobald ein Blatt gefunden wird dessen ID der anderen Seite mitgeteilt und dort geprüft ob diese bereits vorhanden ist. Falls das Blatt noch nicht vorhanden ist werden nun die kompletten Daten des Blattes angefordert und eingefügt. Ist das Blatt aber bereits im Baum wird an die Position des Unterzweig gelaufen und dort das linkeste Blatt betrachtet. Ist dieses nicht das gefundene Blatt wurde der sich am weitesten Links befindende Unterschied gefunden



und der Datensatz zum Einfügen an die andere Datenbank gesandt. Ist das linkeste Blatt aber das Gefundene ist der linkeste Unterschied der nächste rechte Nachbar und kann ausgetauscht werden.

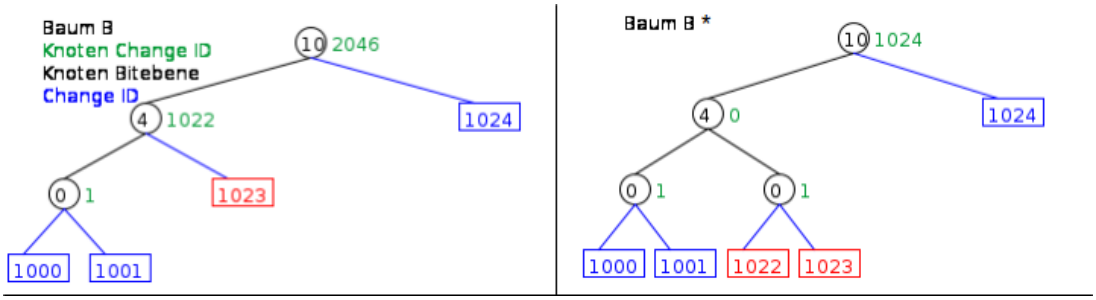


Abbildung 4.5: Baum B, B*

Behandeln von verschieden vielen Bitebenen

Je nach dem wie groß sich die Bäume unterscheiden so verschieden ist auch die Strukturen an Knoten. Das führt so folgender Problematik. Der älteste gesuchte Unterschied liegt links im Baum doch Fehlen ebenfalls andere, neuere Werte und somit auch die Knoten die mit ihnen erzeugt werden. Ein einfaches „Delta Ping-Pong“ Knoten für Knoten und dabei Bitebene für Bitebene ist daher nicht möglich. Auch ist davon Auszugehen das es Möglich ist das auf beiden Seiten Bitebenen fehlen. Dies führt zu einer Vielzahl an Möglichkeiten in denen sich bei Seiten unterscheiden können. Um die Lücken an Knoten zu erkennen und zu behandeln ist in den Deltaobjekten die Bitebene des Deltaknoten sowie die Bitebene des obehalb liegenden Elternknoten enthalten.

Zu beginn eines Synchronisationsschrittes wird im Baum der Knoten ermittelt der sich unterhalb der Bitebene des Elternknoten des erhaltenen Deltas befindet. Dazu wird der Baum mit der Adresszahl solange durchlaufen bis die Bitebene des betrachteten Knoten die Bitebene des Elternknoten des erhaltenen Deltas unterschreitet. Nun wird geprüft ob die Bitebene des gefundenen Knoten gleich der Bitebene des erhaltenen Deltaknotens ist. Ist dies der Fall wird mit der im oberen Abschnitt 4.3.1 beschriebenen Prozedur verfahren. Unterscheiden sich die Bitebenen führt dies zu zwei Möglichkeiten.
Die Bitebene des gefundenen Knotens ist kleiner als die des Deltaknoten. Zwischen den beiden Knoten befindet sich Loch in dem sich noch noch nicht synchronisierte Blätter und dazu gehörige Knoten befinden. Doch diese sollen vom Algorithmus erst später repariert werden da diese neuer und zu erst der ältesten Unterschied behoben werden soll. Daher gilt es das Loch zu überspringen. Dazu sendet die Seite mit dem tieferen Knoten ein spezielles Delta zurück welches das nächste linke Delta unterhalb des erhaltenen Delta anfordert. Dies wird solange wiederholt bis auf beiden Seiten die selbe Bitebene erreicht ist oder falls die kleine Bitebene im anderen Baum nicht vorhanden ein Blatt gefunden wird. Ist auf beiden Seiten die gleiche Bitebene erreicht kön-

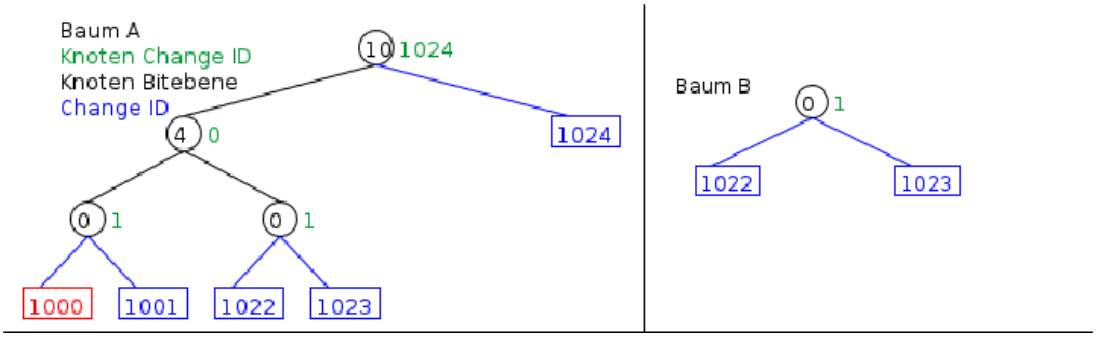


Abbildung 4.6: Bäume mit verschieden vielen Bitebenen

nen die Kinderobjekte wie im oberen Abschnitt 4.3.1 beschrieben verglichen werden. Falls aber die Bitebene des gefundenen Knoten größer ist als die des Deltaknotens wird ein Delta aus dem gefundenen Knoten gebildet und zurück gesandt. Damit werden einfach die Seiten getauscht und der gerade beschriebene Fall tritt ein. Diese Methodik wird ebenfalls genutzt um mit Synchronisation zu beginnen. Einer der Datenbanken wird ein manipuliertes Deltaobjekt gesandt. Dieses trägt als Absender die andere Datenbank. Die Werte der Bitebenen von Deltaknoten und Elternknoten sind negativ. Somit wird zuerst der oberste Knoten lokalisiert. Da dessen Ebenenbit positiv ist und somit größer als der erhaltene negative Deltawert wird der oberste Knoten als Delta an die andere Datenbank gesandt und ein Synchronisationsschritt beginnt.

Ein weiterer Sonderfall tritt ein wenn der Knoten und seine unter im liegenden Kinder des

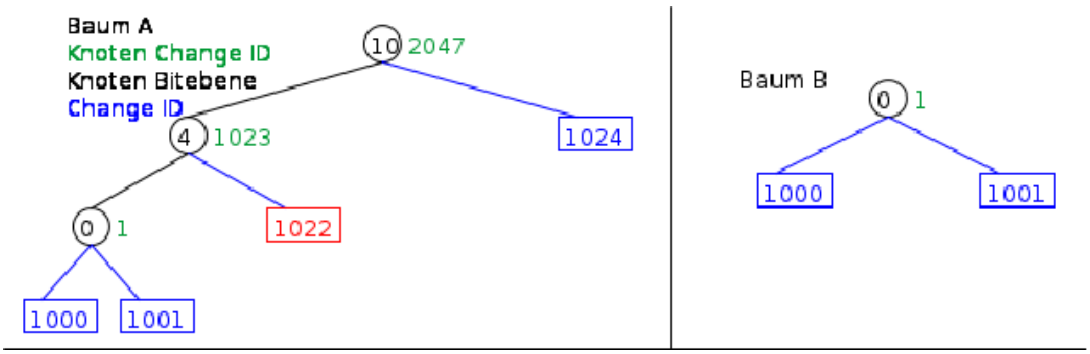


Abbildung 4.7: Bäume mit verschieden vielen Bitebenen

Baumes mit fehlenden Knoten im anderen Baum bereits vollständig enthalten ist und sich der gesuchte Unterschied oberhalb davon befindet. Während des angleichen der Bitebenen werden keine „Change IDs“ verglichen. Erst wenn auf beiden Seiten die selbe Bitebene erreicht ist wird wieder nach Unterschieden geprüft und in diesem Fall Links und Rechts keine Unterschiede gefunden werden was die Bäume als synchron erscheinen lassen wird. Daher muss immer beim Überspringen des Bitebenenlochs bevor das nächst tiefere Delta angefordert wird überprüft werden ob dieses bereits die selbe „Change ID“ hat wie der Knoten. Ist dies der Fall liegt der gesuchter Unterschied ganz links im rechten Kindobjektzweig oder ist das Kindobjekt selbst.

Special case node nicht im anderen baum

4.3.2 Synchronisation Phase 2

Nach Ablauf der ersten Synchronisationsphase ist die sich, in den „Change-ID Bäumen“, am weitesten Links befindende, unterscheidende „Change-ID“ gefunden. Nun sendet die Datenbank in der sich diese befindet das gesamte „EbTreeDataObjekt“ auf welches die „Change-ID“ referenziert an die andere Datenbank in der die „Change-ID“ nicht vorhanden ist. Durch die im „EbTreeDataObjekt“ enthaltene ID kann nun im „ID-Baum“ geprüft werden ob diese enthalten ist. Falls dies nicht der Fall ist die komplette Einfügeoperation des Datensatzes verloren gegangen und dieser muss nun nachträglich in beide Bäume eingefügt werden. Ist es Möglich die ID zu lokalisieren ist dass „EbTreeDataObjekt“ bereits in beiden Datenbanken vorhanden aber auf einem unterschiedlichem Stand. Nun gilt es durch einen Vergleich der beiden „Change-IDs“ heraus zu finden auf welcher Seite der Datensatz aktueller ist und dies auszugleichen. Ist die erhaltene „Change-ID“ größer als die Eigene wird der eigene Datensatz aktualisiert. Ist aber die erhaltene „Change-ID“ kleiner kann der eigene Datensatz als Update an die andere Datenbank gesandt werden. Der Synchronisationsschritt ist abgeschlossen. Egal ob eine veraltete oder aktuelle „Change-ID“ gefunden wird jeder Synchronisationsschritt führt zum Ausgleich eines Unterschiedes.

4.4 Architektur des Prototypen

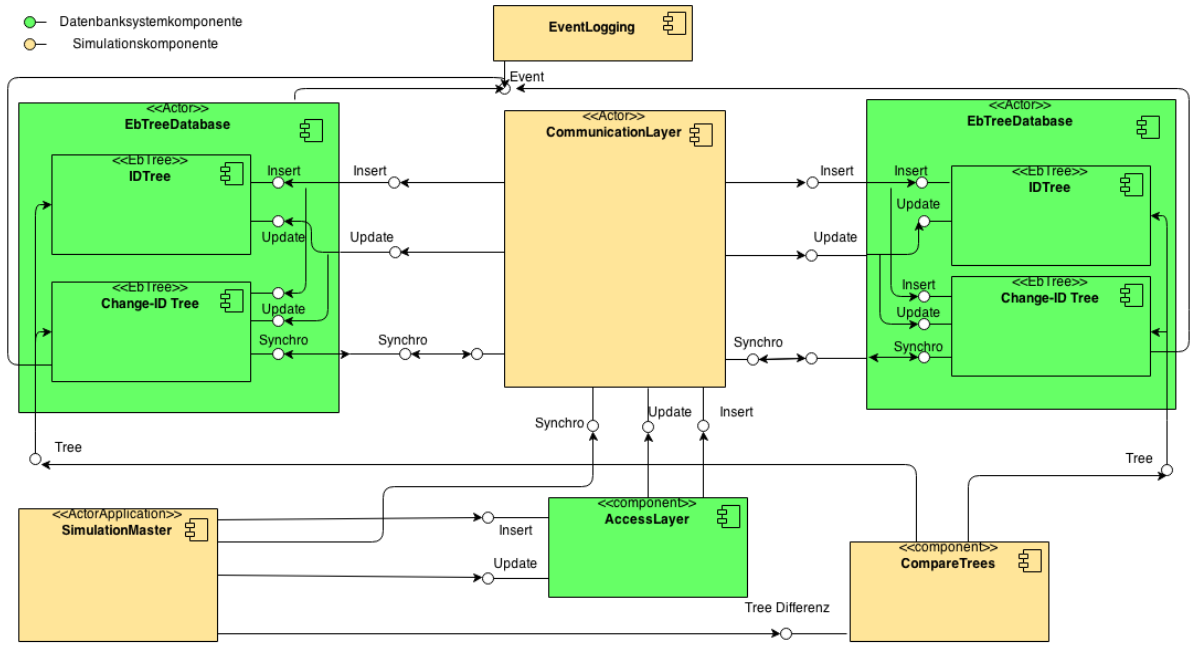


Abbildung 4.8: UML Komponentendiagramm des Prototypen

turn picture

Der in dieser Arbeit erstellte Prototyp besteht grundlegend aus zwei Teilen. Er besteht zum einen aus den Komponenten des in Abschnitt 4.2 beschriebenen Datenbankkonzeptes und der in Abschnitt 4.3 erklärten Synchronisation zwischen zwei Instanzen der Datenbank. Zum anderen einer Simulationsumgebung mit welcher die Funktionalität des Datenbanksystem getestet wer-

den kann. Die Synchronisation zwischen den einzelnen Datenbankinstanzen kann hier ebenfalls getestet und analysiert werden. Durch verschiedene Initialisierungen des Datenbanksystem und dem erzeugen von Last können verschiedene Szenarios erzeugt und die Leistung des Systems bewertet werden.

Die Applikation ist zum Teil als Actorsystem umgesetzt. So wird jede Datenbank, als auch der „CommunicationLayer“, welcher als Bindeglied zwischen den Komponenten fungiert, durch einen Aktor repräsentiert. Das Senden von Actornachrichten ähnelt dem Senden von IP-Paketen zwischen zwei Datenbanken an verschiedenen Standorten. Wann welche Nachricht und in welcher Reihenfolge bei welcher Datenbank ankommt ist nicht vorhersehbar. Verlust oder Verspätung von Paketen und die daraus resultierende Ungleichheit der Datenbanken lassen sich so gut simulieren.

Zwischen den Datenbankaktoren befindet sich, als Teil der Simulationsumgebung, ein Kommunikationsebene. In dieser können Verlust und Verspätung von Paketen gesteuert werden.

4.4.1 Datenbankkomponenten

Der Prototyp setzt einen Teil der in Abschnitt 4.1 beschriebenen Komponenten um. Die Generierung von IDs die neue Daten zu adressieren und den Stand von Aktualisierungen abbilden werden in einer Basisimplementierung des „Accesslayer“ generiert. Diese werden mit den Daten in einem „EbTreeDataObjekt“ gekapselt und alle Datenbankzellen gesandt. Jeden Datenbankzelle wird durch einen Actor repräsentiert und kann Nachrichten mit neuen Daten, Änderungen oder Synchronisationsanfragen erhalten und senden. Im Rahmen dieser Arbeit werden zwei Datenbankzellen verwendet und Operationen wie Einfügen, Aktualisieren oder Löschen sowie die Synchronisation zwischen beiden getestet. Die Datensätze werden in einer Erweiterung des klassischen EB-Baum gehalten.

4.4.2 Simulationskomponenten

Die Komponenten der Simulationsumgebung setzen auf den Datenbankkomponenten auf. Zwischen den zwei Datenbankzellen und dem „Accesslayer“ wird der „CommunicationLayer“ zwischen geschoben. Der „CommunicationLayer“ ist wie die zwei Datenbankzellen ein Actor. Durch ihn läuft die gesamte Kommunikation aller Komponenten. Werden neue Daten Eingefügt oder Aktualisiert werden diese vom „Accesslayer“ an den „CommunicationLayer“ übergeben und an die zwei Datenbanken weiter verteilt. Auch die Synchronisation zwischen den beiden Zellen läuft durch „CommunicationLayer“. Die Daten die im Datenbankystem gekapselt in „EbTreeDataObjekten“ oder „Deltaobjekten“ versandt werden werden als Nachrichten zwischen den Actoren versandt. Jede Nachricht passiert „CommunicationLayer“ und kann somit in diesem beeinflusst werden. So kann ein Prozentwert eingestellt werden mit dem Pakete verloren gehen um das Senden von UDP-Paketen zu simulieren. Auch die Verzögerung von Paketen kann hier simuliert werden. Neue Datensätze oder Aktualisierungen alter Daten werden in einer „Queue“ abgelegt. Soll ein Paket verzögert werden wird es einfach je nach dem wie stark die Verzögerung sein soll weiter hinten eingereiht und somit später versandt.Durch Verlust und Verzögerung werden die Datenbestände in beiden Datenbanken verschieden. Je nach dem wie hoch Verzögerung und Verlust

im „CommunicationLayer“ eingestellt sind können in anderen Teilen der Simulationsumgebung die Auswirkungen auf das gesamte System und die Synchronisation ermittelt werden.

Start Punkt der Simulationsumgebung als auch des gesamten Prototypen ist der „Simulation-Master“. In dieser Komponente wird beim Start des Prototypen das Actorsystem initialisiert. Die beiden Datenbankzellen, „Accesslayer“ und „CommunicationLayer“ werden instanziiert, deren Referenzen unter den Komponenten ausgetauscht und das gesamte Datenbanksystem sowie die Simulationsumgebung selbst initialisiert. Aufgabe des „SimulationMaster“ ist es zum einen das System zu initialisieren und zu starten und anschließend eine Reihe verschiedene Simulationen zu ermöglichen. Während der Simulation werden Daten ausgewertet und diese in einem einer CSV-Datei gespeichert. Welche Simulation ausgeführt werden soll kann vom Nutzer in einem einfachen „Kommandozeileninterface“ ausgewählt werden. Je nach Simulation wird nun das Datenbanksystem mit Daten initialisiert und mit der Simulation begonnen. Während der Simulation kann der Inhalt der einzelnen Zellen verändert werden um Last auf das Datenbanksystem zu simulieren. Der „SimulationMaster“ generiert dazu neue Daten oder Änderungen und sendet diese, via dem „AccessLayer“, an den „CommunicationLayer“. Dort wird nun entschieden ob das Paket Verloren geht oder Verspätet wird und anschließend dieses in der „Queue“ abgelegt. Der „SimulationMaster“ sendet in gewissen Intervallen ein „Clocksignal“ welches dafür sorgt dass das die Pakete im ersten Feld der „Queue“ versandt werden. Je nach dem wie hoch der Verlust oder die Verspätung von Paketen eingestellt ist ergibt sich ein Unterschied zwischen den Zellen der mit der Synchronisation ausgeglichen werden soll. Die Synchronisation der Datenbankzellen wird ebenfalls im „SimulationMaster“ gestartet. Dazu wird an den „CommunicationLayer“ eine Nachricht gesandt das mit der Synchronisation eines Unterschiedes begonnen werden soll. In der Nachricht ist enthalten welche Datenbank mit der Synchronisation beginnt. Der „SimulationMaster“ wartet nun bis die Synchronisation eines Unterschiedes abgeschlossen ist und fährt mit der Simulation fort.

Um den Erfolg der Synchronisation und das Entstehen von Unterschieden bestimmen zu können existiert die Komponente „TreeCompare“, die den Unterschied beider Datenbanken bestimmt werden kann. Um den Unterschied zu bestimmen werden die „ID-Bäume“ und „Change ID-Bäume“ beider Datenbanken Blatt für Blatt verglichen und gezählt um wie viele Elemente sich Unterscheiden. Beim Vergleichen der „ID-Bäume“ kann festgestellt werden um wie viele Einfügeoperationen und beim Vergleichen der „Change ID-Bäume“ wie viele Aktualisieren sich beide Datenbanken unterscheiden.

5 Implementierung

Realisierung/Umsetzung, Beschreibung der Implementierung, nicht des Programmcodes

1. Umsetzung der Systemarchitektur
2. Beschreibung und Besonderheiten der Implementierung

5.1 EBTree

Die im Rahmen dieser Arbeit erstellte Scalaimplementierung des „Elastischen Binär Baums“ orientiert sich zum einen an dem in Grundlagen 2.1 beschriebenen Entwurf von Willy Tarreau,[1] als auch auf einer mehr spezifischen Basisumsetzung der Spinning Wheel GmbH.
In dieser wird bereits die enge Koppelung von Knoten und Blättern, der ursprünglichen C Implementation, aufgebrochen.Die zuvor in „C structs“ aufgebauten Knoten und Blätter wurden durch Java Klassen ersetzt und ihre gemeinsame Eigenschaften in einem Interface definiert. Basisfunktionen des Baumes wie einfügen, löschen und durchlaufen des Baumes waren hier ebenfalls vorhanden und wurden als Teil der Arbeit in Scala neu umgesetzt.

SPW c umsetzung mit blatt+node pointerns?

Eine besondere Anforderung an den Baum ist das die sortiert, verwalteten Schlüssel einzigartig sind und ein Einfügen von Duplikaten verhindert wird. Da das Verwalten von Duplikaten an sich möglich ist aber für das Funktionieren des Baumes nicht zwingend nötig wird ein einfügen dieser, bei der folgend beschrieben Umsetzung, vermieden.

5.1.1 Grundlegender Aufbau

Der Baum selbst ist einer eigenen, generischen Klasse umgesetzt. Diese besteht aus den Funktionalitäten des Baumes, zwei „Case-Klassen“, repräsentativ für Knoten und Blätter, sowie einem „Trait“ der deren gemeinsame Eigenschaften beschreibt. Der generische Typ der Klasse wird bei deren Instanziierung übergeben und legt den Datentyp der in den Blättern gespeicherten Daten fest. Somit kann der Baum jeder Art von Daten halten ohne verändert werden zu müssen.

Der „Trait“ Child beschreibt was dass „Kind“ eines Knoten können muss. Ganz gleich ob es sich hier um ein Blatt oder einen weiteren, einen neuen Zweig öffnenden, Knoten handelt. Festgelegt ist das ein jedes „Kindobjekt“ eine Referenz auf ein „Elternobekt“ haben muss und eine Methode die die eindeutig identifizierbare ID des Objektes zurück gibt. Die ID ist bei Blättern

[1] Tarreau, *Elastic Binary Trees - ebtree*.

der Schlüssel mit dem das Blatt eingefügt wird und bei Knoten die „Node Change ID“ welche zum vergleichen von Baumzweigen anderer Datenbanken bei der Synchronisation verwendet wird.

Durch den Einsatz von „Case-Klassen“ kann durch „Pattern Matching“ beim durchlaufen des Baumes sehr einfach geprüft werden welcher Klasse Kindelemente angehören und diese verarbeitet werden.

Die Aufgabe des Blatt „Case Klasse“ ist sehr einfach. Sie hält die Referenz auf einen übergeben Datensatz und ist durch einen eindeutige Schlüssel identifizierbar. Knoten hingegen verfügen über mehr Logik. So kann dieser die in seiner Wurzel gespeicherten Kindobjekte, weitere Zweige oder Blätter, zurück geben oder neue anfügen. Durch sein Ebenenbit weiß ein Knoten auf welcher binären Ebene des Baumes er steht und kann in dem, bei einer Abfrage- oder Einfügeoperation, übergebenen Schlüssel an der entsprechenden Stelle nachschauen und seinen linkes oder rechtes Kindobjekt verarbeiten.

```
case class Node[T](myBit: Int) extends Child[T] {
    var myZero:                                  Child[T]  = _
    var myOne:                                  Child[T]  = _
    var nodeChangeID:        Long              = _

    def bitOne(uid: Long): Boolean = ((uid & (1L << myBit)) != 0) && (myBit < 64)

    def getChild(uid: Long): Child[T] = bitOne(uid) match {
      case true => myOne;
      case false => myZero
    }
    def setChild(uid: Long, child: Child[T]) = {
      bitOne(uid) match {
        case true => myOne = child
        case false => myZero = child
      }
      child.myParent = this
    }
    override def getID(): Long = nodeStateID
  }
```

Listing 1: Umsetzung eines Knoten des EB-Baum

-> klassen bild

5.1.2 Funktionalität des Baumes

Finden von Schlüsseln

Die Suche eines Schlüssel ist durch schlanke rekursive Funktion gelöst. Der zu suchende Schlüssel gibt einen Art binären Weg vor. So beginnt die Funktion am ersten Knoten des Baumes. Diesem

wird der zu suchende Schlüssel übergeben. Der Knoten weiß durch sein Ebenenbit welche Stelle er in binären Repräsentation des Schlüssels ein nimmt. Je nach dem ob das Bit an dieser Stelle 0 oder 1 ist gibt er sein linkes oder rechtes Kindobjekt zurück. Nun kann via „Pattern Matching“ unterschieden werden ob es sich bei diesem um einen weiteren Knoten handelt oder ein Blatt handelt. Ist das gefundene Objekt ein Blatt wird die Rekursion abgebrochen und das Blatt zurück gegeben. Falls ein Blatt mit dem gesuchten Schlüssel im Baum vorhanden ist wird dieses oder das Blatt mit am dem nächst kleinerem Schlüssel beim durch laufen des Rekursion gefunden. Falls das gefundene Objekt aber ein, einen weiterer einen neuen Zweig öffnenden, Knoten ist wird mit diesem die rekursive Funktion erneut aufgerufen bis ein Blatt gefunden wird.

Einfügen eines neuen Datensatzes

Beim einfügen eines neuen Blatt wird zunächst der Baum nach seinem Schlüssel durchsucht. Falls der Schlüssel bereits vorhanden ist wird der neuen Datensatz im gefundenen Blatt eingefügt und der Alte zurück gegeben. So wird verhindert das Duplikate eingefügt werden können. Falls der Schlüssel nicht vorhanden ist und sich bereits Blätter im Baum befinden wird zu nächst das Blatt gefunden dessen Schlüssel die ähnlichste binäre Repräsentation besitzt. Nun werden die Schlüssel des neuen und des gefundenem Blattes durch eine binäre Oder-Operation verknüpft und die Bitstelle des höchsten Bitunterschiedes ermittelt. Diese wird das neue Ebenenbit des Knoten der mit dem Blatt eingefügt wird. Nun muss die passende Ebene im bereits durch die Suche ermittelten Zweiges gefunden werden. Dazu wird das Ebenenbit des Elternknoten des gefundenen Blattes mit dem neue bestimmten Ebenenbit verglichen ist das Neue kleiner wurde die richtige Ebene gefunden andernfalls wird der Zweig Knoten für Knoten nach oben durchlaufen bis die passende Stelle gefunden ist. An dieser kann der neue Knoten samt Blatt eingehängt werden. Das Durchlaufen dieses Algorithmus stellt sicher das ein neuer Schlüssel stets an der richtigen Stelle eingefügt wird und der Baum aufsteigend von links nach rechts sortiert ist.

Erzeugen von Knoten Status IDs

Das vergleichen von Blättern ist durch deren einzigartige, aufsteigenden Schlüssel einfach um zu setzen. Bei dem, in dieser Arbeit umgesetzten, Synchronisationansatz, Abschnitt 4.3, muss es aber möglich sein ganze Zweige unterhalb eines Knotens zu vergleichen. Daher besitzt der Baum eine Funktion die es ermöglicht für jeden Knoten eine Nummer, die „Knoten Status ID“, zu erzeugen die den Zustand seiner beiden Kinder repräsentiert.

Wird ein Blatt angefügt oder gelöscht werden die „Knoten Status ID“ rekursiv nach oben bis zur Wurzel generiert. Hierfür werden die IDs seiner Kindobjekte einfach durch eine binäre Oder-Operation verknüpft. Ist das Kindobjekt ein Blatt wird als ID der Schlüssel verwendet. Ist das Objekt ein Knoten seine bereits berechnete „Knoten Status ID“.

Methoden zur Synchronisation

Durchlaufen des Baumes

Weitere Funktionen des Baumes

-> aendern eines elementes -> entfernen eines elementes

-> immer sortiert , Elemente werden nach einem Wert hier cID und uID eingefügt und automatisch nach rechts aufsteigend sortiert binaerlogik beschreiben

5.2 TreeActor

-> haelt 2 bäume für uID udn cID

5.3 Acceslayer

5.4 Vergleichen von Bäumen

6 Test

Testarten, Testkriterien, Testumgebung, Testergebnisse

1. Testkriterien und Szenarien
2. Demonstration der Funktionalität
3. Auswertung der Ergebnisse

Test run with different start data -> check which case gets more often check id length xor problem sync ops frequency

7 Fazit und Ausblick alias Ergebnis

Zusammenfassung, Bewertung der Ergebnisse, Vergleich mit der Zielstellung, Ausblick

- spannende basisumsetzung mit viel potenzial da erste hindernisse ueberwunden viel moeglichkeiten der optimierung
- komplexitaets berechnung vs annahme des brutforcealgo -> spannend waeren "delta vs brutforce perfomance test"
- optimize Synchro bigger deltas, abkuerzungen optimierung

die anzahl der synchronisationen selbst regulierender algorithmus -> check ob er im Baum vorwaerts nach rechts kommt und beschleinigung

Notes

8 Literatur

Buch Quellen

Schlosser: Wissenschaftliche Arbeiten schreiben mit LaTeX: Leitfaden für Einsteiger **Schlosser2011**

Joachim Schlosser. *Wissenschaftliche Arbeiten schreiben mit LaTeX: Leitfaden für Einsteiger*. Hrsg. von Joachim Schlosser. 1. Aufl. Bd. 4. Verlagsgruppe Hüthig-Jehle-Rehm, Mai 2011.

Andere Quellen

Tarreau: Elastic Binary Trees - ebtree **Tarreau**

Willy Tarreau. *Elastic Binary Trees - ebtree*. `http://1wt.eu/articles/ebtree/`. Aufgerufen: 26. Juli 2014.

9 Verzeichnisse

Glossar, Abkürzungen, Abbildungen, Tabellen

10 Anhang

technische Dokumentation, Benutzerhandbuch, Installationsbeschreibung

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, 04.02.2014
Ort, Datum

Unterschrift