Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder

Grundlegende Aktualisierungsprobleme beim Constraint-basierten Datenbank-Caching

Diplomarbeit

von
Julia Thiele

Betreuer:
Prof. Dr.-Ing. Dr. h. c. Theo Härder
Dipl.-Inf. Joachim Klein

Tag der Anmeldung:	12. März 2007
Tag der Abgabe:	11. September 2007

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Kaiserslautern, den 10. September 2007

Inhaltsverzeichnis

1 Motivation **1**
1.1 Allgemeines Beispielszenario 3
1.2 Gliederung der Arbeit 3

2 Constraint-basiertes Datenbank-Caching **5**
2.1 Grundlagen des CbDBC 5
2.2 Eigenschaften 8
2.3 Funktionen 9
2.4 Weitere Definitionen 11
2.5 Prototyp *ACCache* 12

3 Grundsätzliche Problemstellungen **15**
3.1 Annahmen 15
3.2 Grundlegende Beobachtungen 16
3.2.1 Daseinsberechtigung von Datensätzen 17
3.2.2 Nachladen von Datensätzen 19
3.2.3 Gültigkeit der Cache Constraints 21
3.2.4 Akzeptanz von Änderungsoperationen im Cache 22
3.3 Änderungsoperationen im Backend 23
3.3.1 Backend hat sehr wenig Wissen 24
3.3.2 Backend hat eingeschränktes Wissen 25
3.3.3 Backend hat volles Wissen 27
3.4 Änderungsoperationen im Cache 29
3.4.1 Ausführung von Änderungsoperationen 30
3.4.2 Probleme bei der Ausführung von Änderungsoperationen 31

4 Synchronisierung von Replikaten **35**
4.1 Problemstellungen der Synchronisierung 35
4.2 Kategorien der Synchronisierungsverfahren 36
4.3 Architektur des verwendeten Gesamtsystems 39
4.3.1 Änderungsoperationen auf Replikaten 41
4.3.2 Isolationsgrade von Datenbanksystemen 42
4.4 Ziele für das CbDBC 44
4.5 Synchronisierung, die SI garantiert 45
4.5.1 Snapshot Isolation (SI) 46
4.5.2 Basisalgorithmus SRCA 47
4.5.3 Einsatzmöglichkeit in CbDBC 51
4.6 Primary-Copy-Ansatz für das CbDBC 53

5 Synchronisierung im CbDBC **55**
5.1 Grundlegende Annahmen 55
5.2 Verwendetes Kostenmodell 56
5.2.1 Selektivitätsfaktor 56
5.2.2 Cache-Selektivität 59
5.2.3 Nachrichten des 2-Phasen-Commit-Protokolls 59
5.2.4 Zusammenfassung des Kostenmodells 60
5.3 Grundidee 61
5.4 Basisalgorithmus 62
5.4.1 Leseoperationen 62
5.4.2 Änderungsoperationen 62
5.4.3 Commit 64
5.4.4 Kosten des Basisalgorithmus 66
5.5 Lösungen der konkreten Problemstellungen 67
5.5.1 Wissensstand des Backend 67
5.5.2 Konflikterkennung und Konfliktbehandlung 71
5.5.3 Deadlocks 76
5.5.4 Anomalien 80
5.5.5 Nachladen von Datensätzen 83
5.5.6 Probleme bei der Ausführung von Änderungsoperationen im Cache 88
5.6 Fazit 90

6 Zusammenfassung und Ausblick **93**

A Anomalien des Mehrbenutzerbetriebs **97**

B Pseudocode für ROpWAn **99**
B.1 Code für das Backend 99
B.2 Code für den Cache 100
B.3 Weiterleitung von Write Sets 101
B.3.1 Backend hat eingeschränkte Wissensbasis 101
B.3.2 Backend hat vollen Wissensstand 101
B.3.3 Berechnung der Anzahl nachzuladender Datensätze 103

Kapitel 1

Motivation

Caching ist ein weit verbreitetes Konzept, um die Verfügbarkeit und Performanz von Anwendungen zu verbessern. Die von einer Anwendung benötigten Daten werden in einem zentralen Datenbank-Server gespeichert, auf den viele unterschiedliche Benutzer zugreifen. Sind die Nutzer über die Welt verstreut, ergeben sich dadurch sehr lange Datenwege, die durch den Einsatz von Caches verkürzt werden können.

Die Verwendung von Caches ist von Vorteil, wenn Nutzer wiederholt auf dieselben Daten des entfernten zentralen Server (Backend) zugreifen. Aufgrund der verkürzten Datenwege können folgende Verbesserungen erreicht werden [Bü06]:

- Verringerung der Latenzzeit für die Abfrage der Daten
- Reduktion der Kommunikationskosten
- Entlastung des entfernten Server
- Steigerung der Skalierbarkeit und der Verfügbarkeit des Gesamtsystems

Die wohl bekannteste Anwendung des Caching-Konzepts ist das sogenannte Web-Caching. Hier werden statische sowie dynamische, von der Anwendungslogik generierte Dokumente oder deren Fragmente im Cache gespeichert. Die dynamischen Dokumentfragmente werden zunehmend aus Daten, die in Datenbanken gehalten werden, generiert. Dieser Caching-Ansatz wird in vielen unterschiedlichen Einsatzbereichen angewendet, wie beispielsweise im E-Commerce. Allerdings ist der Web-Cache aufgrund seiner Konzeption nur von begrenztem Nutzen, da hier zwei Fragmente mit verschiedenen Identifikatoren als unterschiedlich angesehen werden, selbst wenn sie gleiche Daten, möglicherweise auf unterschiedliche Art und Weise, darstellen [Bü06].

Aus diesem Grund nutzt das Datenbank-Caching einen Ansatz, der sich auf die Beziehung zwischen der Anwendungslogik und der Datenbank konzentriert. Im Cache wird eine Teilmenge der Backend-Daten abgebildet, die erst kürzlich angefragt wurde. Dabei ist zu beachten, dass diese Datenspeicherung nicht wie im Datenbankpuffer auf Seitenebene erfolgt, sondern in Form von Anfrageergebnissen. Weiterhin sollte der Cache transparent für den Benutzer sein, so dass es sinnvoll ist, den Cache als Erweiterung eines vollständigen Datenbankverwaltungssystems zu realisieren [Bü06]. Wird der Cache-Manager als Datenbank-Server realisiert, steht die volle Funktionalität einer Datenbank bereits im Cache zur Verfügung. Da der Cache lediglich eine Teilmenge aller Daten aus dem

zentralen Datenbank-Server speichert, muss der Cache-Manager zur Laufzeit entscheiden, ob die Anfrage lokal im Cache ausgewertet werden kann oder auf dem zentralen Datenbank-Server durchgeführt werden muss. Zur Realisierung eines Datenbank-Cache wurden beispielsweise folgende Ansätze vorgestellt:

- Der konzeptionell einfachste Caching-Ansatz bildet ausgewählte Backend-Tabellen komplett im Cache ab [Cor04]. Probleme entstehen hierbei durch sehr große Tabellen, sofern darauf sehr viele Datenänderungen durchgeführt werden. Dadurch entstehen hohe Replikations- und Wartungskosten der Tabellen, die nicht durch die Vorteile einer schnelleren Anfrageausführung ausgeglichen werden können.

  Weiterhin besteht die Möglichkeit, diesen Ansatz aufbauend auf einer feineren Granularität, wie beispielsweise auf Objektebene, umzusetzen. In diesem Fall kann der Objektzugriff über einen Identifikator erfolgen. Wird allerdings eine deklarative Anfrage an den Cache gestellt, so kann dieser nicht entscheiden, ob eine Anfrageauswertung ohne eine Weiterleitung an das Backend möglich ist.

- Materialisierte Sichten repräsentieren Tabellen oder Sichten aus dem entfernten Datenbank-Server, die durch Selektions- oder Projektions-Anfragen spezifiziert werden [BDD+98, GL01, LMSS95]. Aus diesem Grund kann dieser Ansatz als Caching von Anfrageergebnissen bezeichnet werden. Jede materialisierte Sicht wird als separate Tabelle im Cache abgebildet. Dadurch besteht die Gefahr, dass aufgrund überlappender Sichten Redundanzen auftreten. Ein erhöhter Speicherplatzbedarf sowie die Konsistenzerhaltung der einzelnen Sichten, sobald Daten verändert werden, stellen Probleme dar. Zusätzlich können mit Hilfe dieser Sichten lediglich Anfragen beantwortet werden, die durch die Sichtdefinitionen subsumiert werden [BDD+98].

- Die Anwendung *DBProxy* verwaltet Sichten im Cache so, dass eine Datenreplikation dort vermieden werden kann [APTP03]. Um redundanzfreie Sichten erstellen zu können, werden Zusammenhänge zwischen verschiedenen Sichten ausgenutzt, wie z. B. deren Referenz auf gleiche Tabellen. Der Ausschluss von Duplikaten wird durch eine Anfrageerweiterung um die Attribute des Primärschlüssels erreicht. Werden mehrere Sichten auf der gleichen Tabelle definiert, so werden die entsprechenden Attribute in einer gemeinsamen Sicht abgebildet. Die Datensätze erhalten gegebenenfalls „unechte“ Nullwerte für Attribute, die in der betrachteten Sicht nicht definiert wurden. Auf diese Weise können Datenreplikate verhindert, aber dennoch alle Anfragen beantwortet werden, welche die Sichtspezifikationen subsumieren. Mit Hilfe dieses Ansatzes kann der Speicherplatzbedarf im Gegensatz zu sich überlappenden materialisierten Sichten reduziert werden. Allerdings ist eine Erweiterung der Anfrageauswertung notwendig, da hier unechte Nullwerte von echten unterschieden werden müssen. Bei Datenänderungen müssen ebenfalls die Sichten im Cache aktualisiert werden, sofern betroffene Daten hier abgebildet sind.

Alle Lösungen haben gemeinsam, dass das Füllen des Cache nicht selektiv und adaptiv erfolgt, da es weitestgehend vorgeplant werden muss. Teilweise verlangen diese Ansätze eine ständige Überwachung durch die Datenbank-Administra-

tion. Dadurch kann es zu Leistungseinbußen des Gesamtsystems kommen, die von außen erkannt und behoben werden müssen.

Ein weiterer Datenbank-Caching-Ansatz stellt das Constraint-basierte Datenbank-Caching dar. Hier wird unter anderem eine adaptive Vorgehensweise unterstützt, indem sogenannte *Cache Constraints* das Verhalten des Cache steuern. Dieser Ansatz wird im nächsten Kapitel detailliert vorgestellt und bildet die Grundlage für alle weiteren Überlegungen.

Ein verwandtes Konzept des Constraint-basierten Datenbank-Caching stellt der Ansatz von *Oracle TimesTen* [Tea02] dar. Bei diesem Ansatz werden die Tabellen im Cache mit Hilfe von Constraints verbunden, die den Primärschlüssel/ Fremdschlüssel-Beziehungen im Backend entsprechen. Das Konzept *DBCache* [ABK+03] verwendet Cache Groups, die auf der Idee von *TimesTen* basieren. Allerdings liegt hier der Fokus auf der Unterstützung deklarativer Anfragen durch den Cache.

1.1 Allgemeines Beispielszenario

Folgendes Beispiel, welches bereits in [Kle06] verwendet wurde, dient zur Darstellung der grundlegenden Eigenschaften des Gesamtsystems und beschreibt typische Anwendungen eines weltweit agierenden Handelsunternehmens.

Ein Handelsunternehmen entwickelt Produkte, die mit Hilfe von Zweigstellen in mehreren Ländern vertrieben werden. Um die Produkte weltweit besser verkaufen zu können, werden speziell an die Gegebenheiten des Landes angepasste Anwendungen entwickelt. Hierbei müssen diese Anwendungen auf einen zentralen Datenbank-Server zugreifen, der alle notwendigen Daten bereithält, wie z. B. Produktliniendaten oder Stammdaten. Durch den zentralen Server entstehen große Entfernungen. Daraus resultierende lange Antwortzeiten der Anwendungen werden durch eine Datenhaltung in Anwendungsservern verringert. Da nicht alle Daten des zentralen Datenserver für alle Anwendungsserver von Interesse sind, werden beispielsweise länderspezifische Daten in einem Cache-System repliziert. Dieses Cache-System soll möglichst adaptiv, je nach aufkommender Anfragelast, benötigte Teilmengen der Daten vorhalten, wobei nur solche Daten zwischengespeichert werden, die mit hoher Wahrscheinlichkeit in naher Zukunft nochmals angefragt werden. In Abbildung 1.1 wird die Struktur dieses Handelsunternehmens dargestellt.

Das Cache-System wird mit Hilfe eines vollständigen Datenbank-Server realisiert. Dadurch kann im Cache auf die, in einer Datenbank zur Verfügung gestellte, Anfrageverarbeitungslogik zurückgegriffen werden. Weiterhin besteht für die Caches die Anforderung, dass eine volle Transparenz für die Anwendungen erhalten bleibt. Dies bedeutet, dass keine Änderungen hinsichtlich des Zugriffs auf die Daten über die Anwendungsschnittstelle notwendig sind. Die Anwendung soll folglich nicht erkennen, ob die Daten vom zentralen Datenserver oder vom Cache-System bezogen werden.

1.2 Gliederung der Arbeit

Zunächst wird das Constraint-basierte Datenbank-Caching vorgestellt, mitsamt dessen Anforderungen, Grundlagen und Eigenschaften. Danach werden grund-

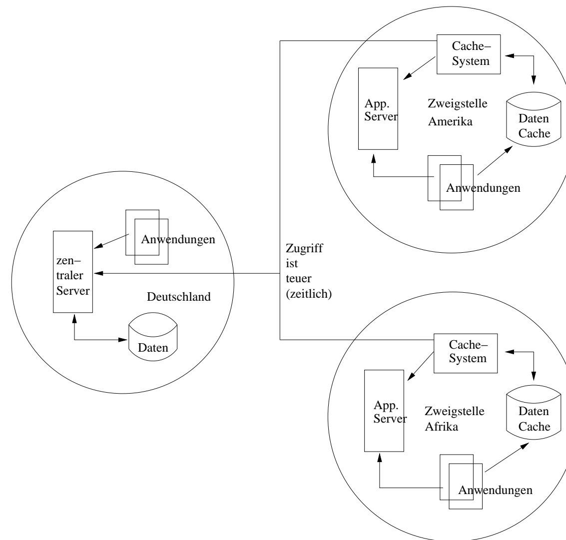


Abbildung 1.1: Allgemeines Beispielszenario

sätzliche Problemstellungen beschrieben, die bei auszuführenden Änderungsoperationen im Cache oder auf dem Datenbank-Server auftreten können. Ergänzt werden diese Beschreibungen mit Ansätzen zur Synchronisierung von Replikaten. Diese Synchronisierungsverfahren werden bezüglich ihrer Einsatzmöglichkeiten im Constraint-basierten Datenbank-Caching analysiert. Darauf folgend wird ein Synchronisierungsverfahren vorgestellt, das auf einem Primary-Copy-Ansatz basiert und an die Anforderungen des Constraint-basierten Datenbank-Caching angepasst wird. Abschließend wird eine Zusammenfassung und Bewertung des vorgestellten Verfahrens gegeben. Zusätzlich wird beschrieben, welche weiteren Arbeiten bezüglich der Umsetzung des Verfahrens notwendig sind.

Kapitel 2

Constraint-basiertes Datenbank-Caching

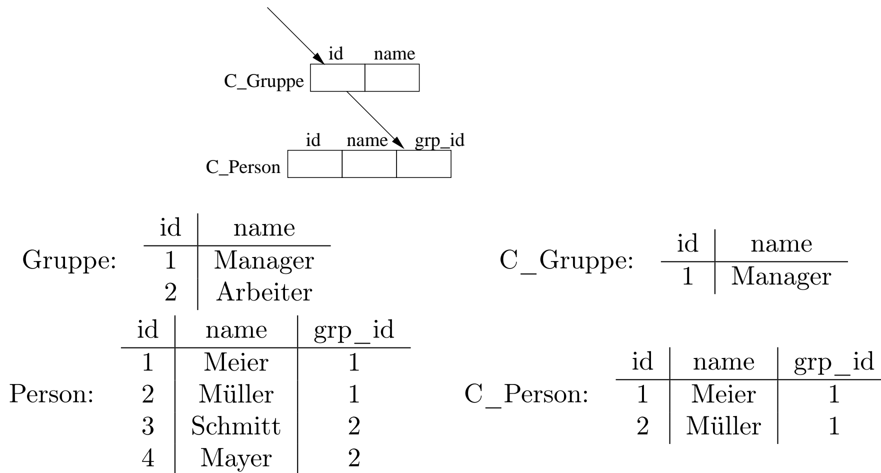
In diesem Kapitel wird das Constraint-basierte Datenbank-Caching (CbDBC) vorgestellt, das die Grundlage für alle weiteren Überlegungen bildet. Die Hauptidee dieses Caching-Ansatzes liegt darin, dass im Cache alle Datensätze gespeichert werden, die vordefinierte Bedingungen erfüllen. Dadurch wird eine adaptive Verwaltung des Cache-Inhalts möglich, der sich an den in der Vergangenheit nachgefragten Daten orientiert.

In den weiteren Betrachtungen wird die Verwendung eines relationalen Datenbanksystems angenommen, das SQL als Anfragesprache verwendet. Die grundlegenden Prinzipien des Constraint-basierten Datenbank-Caching können aber ebenfalls auf andere Datenmodelle übertragen werden [Bü06].

Zur besseren Erläuterung dieses Caching-Ansatzes werden in Kapitel 2.1 zunächst grundlegende Begriffsdefinitionen vorgestellt, welche das Verhalten im Cache beschreiben. So können z. B. folgende Fragestellungen beantwortet werden: Wie wird der Cache mit Daten gefüllt? Wie können mit Hilfe des Cache-Inhalts möglichst flexibel Anfragen beantwortet werden? In den folgenden Abschnitten werden die im Constraint-basierten Datenbank-Caching verwendeten Funktionen und Eigenschaften vorgestellt. Danach werden ergänzende Begriffe definiert, die zum Verständnis der nachfolgenden Erläuterungen beitragen.

2.1 Grundlagen des CbDBC

Im Cache werden sogenannte *Cache Groups* festgelegt. Zusätzlich werden Bedingungen definiert, die beispielsweise das Laden relevanter Daten im Cache auslösen. Eine Cache Group bildet einen statischen Ausschnitt der Tabellen des zentralen Datenbank-Server, dem Backend, in den Cache ab [HB07]. Diese Teilmenge der Backend-Tabellen wird mit den entsprechenden Tabellendefinitionen im Cache erzeugt. In den Cache-Tabellen werden die kompletten Tupel aus dem Backend gespeichert, ohne jedoch die Fremdschlüsselbeziehungen abzubilden. Weiterhin werden in einer Cache Group sogenannte *Cache Constraints* definiert, welche das Cache-Verhalten steuern. Cache Constraints beschreiben die Inhalte, sowie die Abhängigkeiten zwischen den Inhalten der Cache-Tabellen und definieren die gültigen Zustände der Caches. Darüber hinaus werden die



Gruppe:

id	name
1	Manager
2	Arbeiter

C_Gruppe:

id	name
1	Manager

Person:

id	name	grp_id
1	Meier	1
2	Müller	1
3	Schmitt	2
4	Mayer	2

C_Person:

id	name	grp_id
1	Meier	1
2	Müller	1

Abbildung 2.1: Beispiel zur Darstellung der Wirkungsweise eines RCC

Constraints genutzt, um zu entscheiden, ob eine Anfrage im Cache beantwortet werden kann. Ziel der Cache Constraints ist eine Unterstützung bestimmter Anfragetypen sowie die Gewährleistung einer leichten Entscheidung bezüglich der Beantwortbarkeit von Anfragen. Zu diesen Anfragetypen gehören vor allem Gleichheitsprädikate. Es können folglich nur Projektionen, Joins und Selektionen mit Gleichheitsprädikaten ausgewertet werden, sofern entsprechende Datensätze im Cache existieren. Folgende Cache Constraints werden unterschieden:

- Referential Cache Constraint (RCC)
- Füllspalten
- Kontrollspalten

Referential Cache Constraint (RCC)

Ein Referential Cache Constraint $S.a \rightarrow T.b$ zwischen einer Quellspalte $S.a$ und einer Zielspalte $T.b$ ist genau dann erfüllt, wenn alle Werte v in $S.a$ wertvollständig[1] in $T.b$ sind [HB07].

Ein RCC beschreibt eine Beziehung zwischen zwei Tabellen in einer Cache Group. Die Quell- und Zielspalte können entweder einer Tabelle oder verschiedenen Tabellen angehören. Ein RCC kann als *ein-* oder als *ausgehender* RCC für eine Tabellenspalte bezeichnet werden. Ist die betrachtete Tabellenspalte die Quellspalte eines RCC, so ist dieser ein ausgehender RCC. Ist die Tabellenspalte die Zielspalte eines RCC, so spricht man von einem eingehenden RCC.

Zur Verdeutlichung der Funktionen eines RCC dient Abbildung 2.1. Im Backend existieren unter anderem die Tabellen *Gruppe* und *Person*. Diese beiden Tabellen werden mit Hilfe einer Cache Group im Cache abgebildet und haben

[1] Ein Wert w heißt wertvollständig in einer Spalte $S.c$ genau dann, wenn alle Sätze $\sigma_{c=w}S$ in C_S vorhanden sind [HB07] (vgl. Kapitel 2.2).

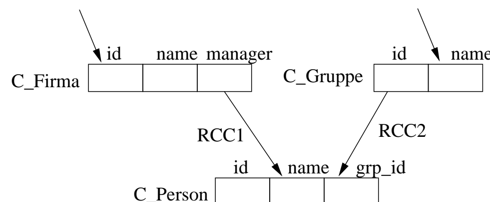


Abbildung 2.2: Gegenbeispiel zu: RCCs gelten in beiden Richtungen

in Abbildung 2.1 die Bezeichnungen *C_ Gruppe* und *C_ Person*[2]. Weiterhin ist im Cache zwischen diesen beiden Tabellen der RCC

$$\text{C_Gruppe.id} \rightarrow \text{C_Person.grp_id}$$

definiert. Aufgrund des RCC können folgende Aktionen unterschieden werden:

- Wie im Beispiel dargestellt, wird in die Tabelle *C_ Gruppe* der Datensatz mit $id = 1$ geladen. Daraufhin müssen aufgrund des definierten RCC alle Datensätze mit $grp_id = 1$ in die Tabelle *C_ Person* nachgeladen werden.

- Weiterhin können mit Hilfe des RCC z. B. Joins mit Gleichheitsprädikaten über die Quell- und Zielspalte ausgeführt werden. Es wird z. B. folgende Anfrage an den Cache gestellt:

  ```
  select p.id, p.name, g.name from C_Person p, C_Gruppe g
  where g.id = p.grp_id and g.id = 1
  ```

  Der Cache kann diesen Join aufgrund des definierten RCC im Cache beantworten. Dies folgt daraus, dass sobald der Wert $id = 1$ in der Tabelle *C_ Gruppe* existiert, befinden sich alle Join-Partner aufgrund der Wertvollständigkeit für diesen Wert im Cache.

Es ist zu beachten, dass die Vollständigkeit der Werte nur in der Richtung des RCC gelten. Folglich kann der Cache-Manager nicht davon ausgehen, dass ein Wert aus der Spalte *C_ Person.grp_ id* ebenfalls in der Spalte *C_ Gruppe.id* existiert. In dem Fall von Abbildung 2.1 trifft dies zwar zu, allerdings lassen sich diverse Gegenbeispiele konstruieren, indem auf einer Tabelle mehrere eingehende RCCs definiert werden. Ein solches Gegenbeispiel wird in Abbildung 2.2 veranschaulicht. Hier werden in die Tabelle *C_ Person* Datensätze aufgrund von *RCC1* und *RCC2* geladen. Nun wird folgender Join durchgeführt:

```
select p.id, p.name, g.name from C_Person p, C_Gruppe g
where p.grp_id = g.id and p.grp_id = 1
```

Hier besteht die Möglichkeit, dass die Datensätze mit den Werten $grp_id = 1$ aufgrund von *RCC1* in den Cache geladen wurden und der entsprechende Datensatz mit $id = 1$ nicht in der Tabelle *C_ Gruppe* existiert. In diesem Fall kann der angefragte Join nicht ausgewertet werden, da der Cache-Manager keinerlei Informationen über die Wertvollständigkeit der Spalte *C_ Gruppe.id* hat.

[2]Aus Gründen der Transparenz müssen die Tabellen im Cache mit den gleichen Schemadefinitionen wie im Backend erstellt werden. Lediglich die Fremdschlüsseldefinitionen werden im Cache nicht realisiert. Zur besseren Veranschaulichung und einfacheren Unterscheidung werden die Tabellennamen im Cache zusätzlich zum eigentlichen Namen mit *C_* markiert.

Füllspalten

Eine Füllspalte ist eine spezielle Spalte einer Cache Group, über welche das Füllen der Cache Group mit Backend-Datensätzen gesteuert wird [HB07]. Diese Spalte enthält einen eingehenden RCC, bei dem keine Quellspalte existiert. Die Werte in diesen Füllspalten werden als *Füllwerte* bezeichnet.

Der Cache wird aufgrund angefragter Daten gefüllt. Es wird z. B. folgende *Select*-Anfrage an den Cache gestellt:

```
select * from C_Gruppe where id = 3
```

Wenn die Spalte *id* eine Füllspalte ist, werden alle Datensätze mit dem Wert $id = 3$ aus der Tabelle *Gruppe* in den Cache geladen. Sind zusätzlich ausgehende RCCs auf der Tabelle *C_ Gruppe* definiert, so wird für jeden dieser ausgehenden RCCs die Wertvollständigkeit der RCC-Zielwerte sichergestellt.

Kontrollspalte

Eine Spalte $S.a$ ist eine Kontrollspalte von $T.b$, wenn ein RCC der Form $S.a \rightarrow T.b$ im Cache definiert ist [HB07].

Zur einheitlichen Behandlung aller Tabellenspalten im Cache wird vom Cache-Manager für jede existierende Füllspalte eine *künstliche Kontrollspalte* erzeugt. Diese stellt eine Tabelle mit lediglich einer Spalte dar, welche die Werte der Füllspalte ohne Duplikate enthält. Aufgrund dieser künstlichen Spalten erhalten alle vorhandenen RCCs eine Quell- und eine Zielspalte, wodurch die Betrachtung der RCCs vereinfacht wird.

2.2 Eigenschaften

Die Hauptidee des Constraint-basierten Datenbank-Caching basiert auf der Vollständigkeit von Werten festgelegter Spalten. Aufgrund dieser Eigenschaft kann der Cache selbstständig entscheiden, ob alle Datensätze, die ein Prädikat erfüllen, dort vorhanden sind. Folgende Formen der Vollständigkeit werden für den Cache definiert [HB07]:

- *Wertvollständigkeit*

  Ein Wert w heißt wertvollständig in einer Spalte $S.c$ genau dann, wenn alle Sätze $\sigma_{c=w}S$ in C_S vorhanden sind. Wertvollständigkeit bedeutet folglich, dass alle Datensätze des Backend, die in Spalte $S.c$ den Wert w enthalten, im Cache vorhanden sind.

- *Spaltenvollständigkeit*

  Eine Tabellenspalte ist spaltenvollständig, wenn jeder dort auftretende Wert wertvollständig ist.

- *Prädikatvollständigkeit*

  Eine Tabellenmenge heißt bezüglich eines Prädikats P prädikatvollständig, wenn alle Datensätze $\sigma_P S$ in C_S vorhanden sind. Dies bedeutet, dass im Cache alle Datensätze aus dem Backend existieren, welche das Prädikat P erfüllen.

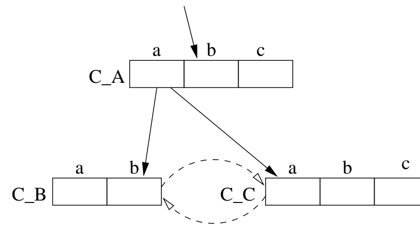


Abbildung 2.3: Beispiel für induzierte Vollständigkeit

- *Intervallvollständigkeit*

  Ein Intervall r ist in einer Cache-Spalte $S.c$ genau dann intervallvollständig, wenn alle Datensätze $\sigma_{c \in r} S$ in C_S enthalten sind.

- *Induzierte Bereichsvollständigkeit*

  Durch das Zusammenspiel von mehreren Constraints erfolgt eine Bereichsvollständigkeit in einer Spalte. Existieren die in Abbildung 2.3 dargestellten Tabellen und definierten RCCs, so sind die beiden Spalten $C_B.b$ und $C_C.a$ induziert vollständig. Dies folgt daraus, dass diese beiden Spalten nur aufgrund der Spalte $C_A.a$ gefüllt werden und sonst aufgrund keines anderen RCC Datensätze in die Tabellen C_B und C_C geladen werden.

Weiterhin gibt es sogenannte *Cache Keys*, welche auf Spalten verweisen, für die eine Spaltenvollständigkeit erzwungen wird. Eine *Kandidatenmenge* beschreibt alle Werte w, die bei einer Referenz $f = w$ aus dem Backend in den Cache geladen und dort wertvollständig gemacht werden.

Bei der Verwendung der RCCs ist zu beachten, dass die Transitivität nicht gilt. Dies kann mit Hilfe des negativen Caching, das in Kapitel 2.3 erklärt wird, verdeutlicht werden. Existieren beispielsweise folgende RCCs:

$$\text{C_S.a} \rightarrow \text{C_T.b und C_T.b} \rightarrow \text{C_U.c}$$

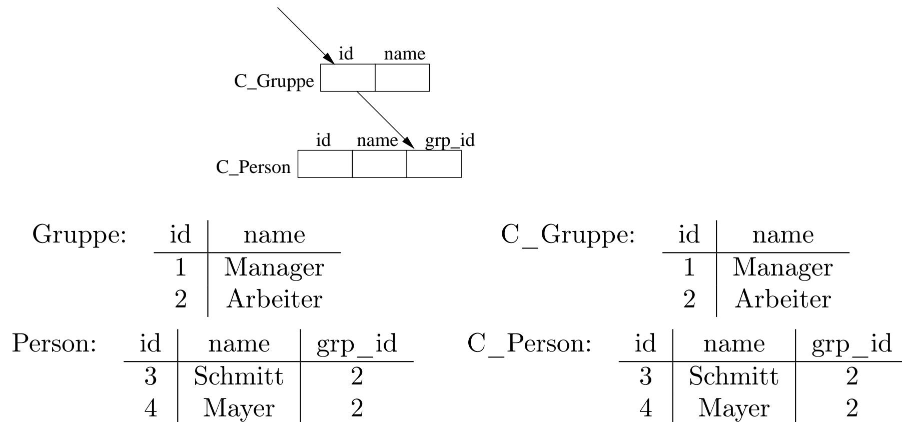
Sobald ein Wert w in der Spalte $C_S.a$ existiert, muss dieser in Spalte $C_T.b$ wertvollständig gemacht werden. Unter der Annahme, dass der Wert w in T.b nicht existiert, ist dieser Wert in $C_T.b$ wertvollständig. Allerdings wird durch diese Nichtexistenz von w kein Nachladen entsprechender Datensätze in $C_U.c$ erzwungen. Somit wurde gezeigt, dass die Transitivität für RCCs nicht gilt.

2.3 Funktionen

In diesem Abschnitt werden die wichtigsten Funktionen des Constraint-basierten Datenbank-Caching vorgestellt. Hierbei handelt es sich um die Sondierung und um das Löschen von Datensätzen im Cache.

Das Stellen von Testanfragen, die im Cache die Existenz von Werten überprüfen, wird *Sondierung* genannt [HB07]. Um das Sondierungsverfahren leichter erklären zu können, soll im Cache folgender Equi-Join ausgewertet werden:

```
select A.a, B.c from A, B where A.a = B.c and A.a = 3
```



Gruppe:

id	name
1	Manager
2	Arbeiter

C_Gruppe:

id	name
1	Manager
2	Arbeiter

Person:

id	name	grp_id
3	Schmitt	2
4	Mayer	2

C_Person:

id	name	grp_id
3	Schmitt	2
4	Mayer	2

Abbildung 2.4: Beispiel für negatives Caching

Das Sondierungsverfahren prüft, ob der Wert $A.a = 3$ im Cache wertvollständig ist. Hierfür wird für jeden eingehenden RCC der Spalte $A.a$ überprüft, ob der Wert 3 in der RCC-Quellspalte existiert. Ist dies der Fall, so ist dieser Wert in der Spalte $A.a$ wertvollständig und aufgrund des existierenden RCC $A.a \rightarrow B.c$ auch in der Spalte $B.c$. Die Sondierung wird somit erfolgreich abgeschlossen, da sich alle benötigten Datensätze zum Auswerten dieser Anfrage im Cache befinden. Kann hingegen keine Aussage über die Wertvollständigkeit des Wertes 3 in der Spalte $A.a$ getroffen werden, so schlägt die Sondierung fehl und die Anfrage kann nicht im Cache ausgewertet werden.

Zur Verbesserung der Sondierung wird das *negative Caching* angewendet. Als negatives Caching wird der Fall bezeichnet, dass die Informationen, die im Cache zwischengespeichert werden, das Nicht-Vorkommen von Objekten im Backend ausdrücken [HB07]. Zur Verdeutlichung dieses Verfahrens dient das Beispiel in Abbildung 2.4. In diesem Beispiel wurden der Gruppe *Manager* bisher noch keine Personen zugeordnet. Da keine solchen Datensätze im Backend existieren, werden auch keine Personen mit dem Identifikator eines Managers in den Cache geladen. Aufgrund des Datensatzes in der Tabelle *C_ Gruppe* und dessen Wertvollständigkeit kann der Cache-Manager folgern, dass alle notwendigen Datensätze im Cache vorhanden sind. In diesem Fall kann ein Join zwischen den beiden Tabellen im Cache ausgeführt und eine Anfrage korrekt beantwortet werden, wie z. B.

```
select p.id, p.name, g.name from C_Person p, C_Gruppe g
where g.id = p.grp_id and g.id = 1
```

Diese Anfrage gibt zwar eine leere Ergebnismenge zurück, ist aber dennoch korrekt.

Bei der Entfernung von Prädikatextensionen aus dem Cache können ähnliche Testanfragen wie im Sondierungsverfahren verwendet werden. Eine Prädikatextension beschreibt die Menge aller Datensätze, die für eine Auswertung eines Prädikats P benötigt wird. Prädikatextensionen werden gelöscht, um die Datenmenge im Cache in Grenzen zu halten. Weiterhin kann dadurch der Aufwand verringert werden, welcher notwendig ist, um die Daten im Cache nach

Änderungsoperationen aktuell zu halten. Eine optimale Entfernungsmethode für Prädikatextensionen ist nicht leicht zu finden, da sich mehrere Prädikatextensionen überlappen können und somit Datensätze mehreren Extensionen angehören können. Es stehen folgende Alternativen zum Löschen von Prädikatextensionen zur Verfügung [Bü06]:

- Der erste Ansatz löscht solche Prädikatextensionen aus dem Cache, die nicht weiter im Cache bestehen müssen. Wenn sich alle Prädikatextensionen überlappen und somit voneinander abhängig sind, dann ist es möglich, dass der gesamte Cache-Inhalt gelöscht wird. Dies ist der radikalste Ansatz und ein unerwünschter Weg, weil ohne Datensätze im Cache dessen Verwendungsvorteile nicht ausgenutzt werden können.

- Ein zweiter Ansatz geht in kleineren Schritten vor, um dadurch den Speicherplatz flexibler freizugeben. Prädikatextensionen können einzeln oder in kleineren Gruppen entfernt werden. Hierbei wird ständig geprüft, dass keine andere Prädikatextension verletzt wird und somit alle Cache Constraints eingehalten werden. Es werden Datensätze einer zu löschenden Extension identifiziert, die keiner anderen Prädikatextension angehören, die im Cache verbleiben soll. Um dies umsetzen zu können, wird ein etwas abgeändertes Sondierungsverfahren angewendet, das im Folgenden anhand einer Tabelle C_S beschrieben wird. In dieser Tabelle werden alle Spalten mit eingehenden RCCs bezüglich ihrer Werte überprüft, z. B. $C_S.a = w$. Ein erfolgreiches Sondieren im Cache beschreibt die Situation, dass sich die Extension des getesteten Prädikats im Cache befinden muss. Bei einem erfolglosen Sondieren kann der Datensatz aus dem Cache gelöscht werden, weil er keiner Extension angehört.

2.4 Weitere Definitionen

Nachdem die wichtigsten Grundlagen und Eigenschaften beschrieben wurden, werden diese noch um weitere Begriffsdefinitionen ergänzt. Durch die Definition mehrerer RCCs innerhalb einer Cache Group besteht die Möglichkeit, dass eine Menge dieser RCCs einen Zyklus bildet. Zyklen implizieren die Gefahr, dass hierdurch gegebenenfalls so lange Datensätze in den Cache geladen werden, bis alle Datensätze aus den betroffenen Backend-Tabellen in den Cache geladen wurden. Es werden folgende Zyklen unterschieden:

a Ein Zyklus ist *homogen*, wenn pro Tabelle immer nur eine Spalte dem Zyklus angehört (Abbildung 2.5 (a)).

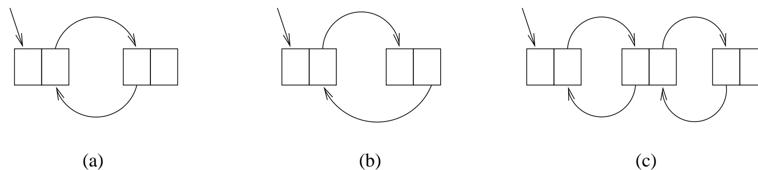


Abbildung 2.5: Zyklenarten

b Ein Zyklus ist *heterogen*, wenn in mindestens einer Tabelle mehrere Spalten im Zyklus involviert sind (Abbildung 2.5 (b)). Diese Zyklenart wird normalerweise im Constraint-basierten Datenbank-Caching nicht erlaubt, da beispielsweise das Füllen der am Zyklus beteiligten Tabellen oft unerwünscht viele Tupel betrifft.

c Ein Zyklus ist *isoliert*, wenn keine Tabelle des ersten Zyklus an einem zweiten Zyklus beteiligt ist. In Abbildung 2.5 (a,b) sind isolierte Zyklen dargestellt, Abbildung 2.5 (c) enthält hingegen keine isolierten Zyklen.

Eine *atomare Zone* enthält eine oder mehrere Cache-Tabellen, die während des Nachladens innerhalb einer Transaktion gefüllt werden müssen [Mer05]. Dies trifft z.B. bei Zyklen zu. Sobald in eine Tabelle des Zyklus Datensätze eingefügt werden, müssen in den anderen Tabellen des Zyklus Datensätze nachgeladen werden. Hierbei wird allerdings gegebenenfalls der RCC verletzt, welcher den Zyklus schließt und somit auf die Tabelle verweist, in die die Datensätze ursprünglich eingefügt wurden. Folglich müssen alle Tabellen eines Zyklus innerhalb einer Transaktion gefüllt werden, so dass nach dieser Transaktion alle Cache Constraints gültig sind. Mit Hilfe dieser atomaren Zonen kann die Cache Group als zyklenfrei angesehen werden.

Innerhalb einer Tabelle kann des Weiteren eine sogenannte *Schmugglerbeziehung* zwischen zwei Spalten a und b existieren ($a \Rightarrow b$) [HB07]. Diese Beziehung beschreibt den Einfluss der Werte von Spalte a auf die Werte von Spalte b, bezogen auf die Einträge, die aus der Backend-Tabelle in den Cache geladen werden. Eine solche Beziehung wird in Abbildung 2.6 dargestellt. In dieser Abbildung werden beispielsweise Datensätze aufgrund von *RCC1* in die Cache-Tabelle geladen. Aufgrund dieser neuen Datensätze besteht die Möglichkeit, dass neue Werte in der Spalte c auftreten. Diese neuen Werte lösen gegebenenfalls ein erneutes Nachladen von Datensätzen über *RCC2* aus. Ein solches Verhalten wird als Schmugglerbeziehung beschrieben.

2.5 Prototyp *ACCache*

Eine konkrete Implementierung des Constraint-basierten Datenbank-Caching stellt der Prototyp *ACCache* (Adaptive Constraint-based Cache) dar. Hierbei sind die Caches und das Backend in einer Sterntopologie angeordnet. Dies bedeutet, dass alle Caches nur mit dem Backend und nicht untereinander kommunizieren. Im Backend werden alle Datensätze gespeichert, während in den

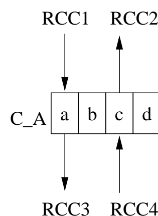
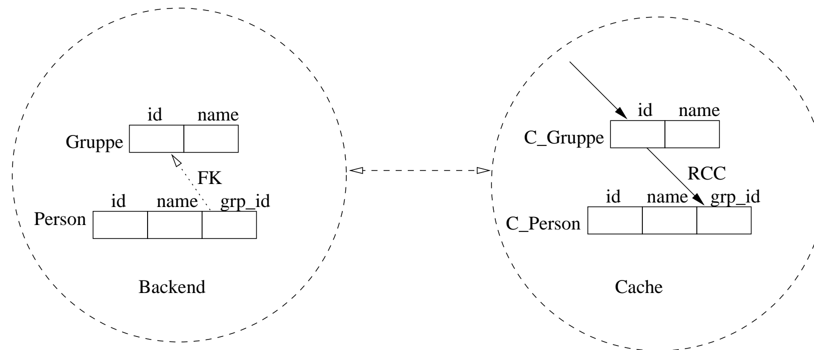


Abbildung 2.6: Beispiel einer Schmugglerbeziehung



Gruppe:

id	name
1	Manager
2	Arbeiter

C_Gruppe:

id	name
1	Manager

Person:

id	name	grp_id
1	Meier	1
2	Müller	1
3	Schmitt	2
4	Mayer	2

C_Person:

id	name	grp_id
1	Meier	1
2	Müller	1

Abbildung 2.7: Beispieltabellen im Backend und Cache

Caches lediglich Teilmengen der Daten mit Hilfe von Cache Groups abgebildet werden.

Bei dieser Implementierung ist zunächst das Laden von Datensätzen in den Cache von Interesse. Es gibt zwei Aktionen, die das Laden von Datensätzen in den Cache auslösen.

- Es werden aufgrund von Datenanfragen Datensätze in den Cache geladen. Dies geschieht, wenn die Prädikate der Datenanfragen auf Füllspalten basieren.
- Es werden Datensätze in den Cache nachgeladen, wenn neue Werte in RCC-Quellspalten eingefügt werden.

Um betroffene Datensätze in den Cache zu laden, werden die RCCs top-down ausgehend von den Kontrollspalten analysiert. Dies kann basierend auf den Kontrollspalten und den Quellspalten der RCCs erfolgen, da hierdurch die Werte festgelegt werden, die in der Cache Group vorhanden sein müssen. Die Analyse stoppt, sobald keine Datensätze mehr zum Nachladen in den Cache identifiziert werden. Bei der zuletzt betrachteten Tabelle wird dann mit dem Nachladen der Datensätze begonnen. Danach werden bottom-up die Datensätze in den Cache geladen. Ein Beispiel wird in Abbildung 2.7 veranschaulicht. Unter dieser Voraussetzung soll folgende Datenanfrage ausgeführt werden:

```
select * from C_Gruppe where id = 1
```

In diesem Fall ist die Spalte *C_ Gruppe.id* eine Füllspalte, so dass das Laden von Datensätzen in den Cache ausgelöst wird. Es wird zunächst der Datensatz

mit der $id = 1$ in der Tabelle *Gruppe* zum Laden in den Cache identifiziert. Danach wird der ausgehende RCC überprüft und es müssen die Datensätze mit $id = 1, 2$ der Tabelle *Person* in den Cache geladen werden. Beim Laden in den Cache werden zunächst die Datensätze der Tabelle *Person* in die Cache-Tabelle *C_ Person* eingefügt und darauf folgend der zuvor identifizierte Datensatz in die Tabelle *C_ Gruppe*.

Das Laden wird als Bottom-up-Ansatz durchgeführt, weil dadurch das sehr lange Halten von Schreibsperren vermieden werden kann [Mer05]. Würde man top-down Datensätze in den Cache laden, müssten die Sperren auf einer Tabelle so lange gehalten werden, bis alle referenzierten Tabellen ebenfalls gefüllt sind, so dass die Cache Constraints gültig bleiben. Beim Bottom-up-Ansatz ist dies nicht notwendig, da für jede dieser Tabellen bereits alle referenzierten Tabellen gefüllt wurden und somit die Cache Constraints gültig sind.

Aufgrund der vorhandenen RCCs erhalten die Datensätze im Cache ihre Daseinsberechtigung. Dies bedeutet, dass diese Datensätze im Cache vorhanden sein müssen, damit alle RCCs in der Cache Group gültig sind. Dadurch besteht die Möglichkeit, dass der Cache überfüllt wird. Aus diesem Grund wird periodisch überprüft, ob eine spezifizierte Füllmarke der Cache-Datenbank erreicht wurde [Mer05]. Wird diese Füllmarke überschritten, so führt der Garbage Collector einen oder mehrere Löschvorgänge durch. Hierbei werden Werte aus den Kontrollspalten gelöscht sowie rekursiv entlang der ausgehenden RCCs absteigend Datensätze entfernt. Allerdings muss dabei beachtet werden, dass keine Cache Constraints verletzt werden und somit keine Datensätze gelöscht werden, die noch von anderen RCCs referenziert werden. Bei diesen Löschvorgängen nehmen Zyklen eine besondere Rolle ein. Aufgrund der Referenzierungen im Zyklus dürfen keine Datensätze entfernt werden, da sonst Cache Constraints verletzt würden. In diesem Fall hilft erneut die Betrachtung der Zyklen in atomaren Zonen. Wird der Löschvorgang innerhalb einer Transaktion durchgeführt, so können keine ungültigen Cache-Zustände entstehen. Allerdings ist bei dem Löschen von Werten im Zyklus ebenfalls zu beachten, dass ein Wert aufgrund eines weiteren, nicht zum Zyklus gehörenden, RCC referenziert werden kann und somit nicht gelöscht werden darf.

In diesem Kapitel wurde das Constraint-basierte Datenbank-Caching vorgestellt. Diese Ausführungen basieren zunächst lediglich auf *Select*-Anfragen. Da jedoch Änderungsanfragen nicht ausgeschlossen werden dürfen, werden diese im Folgenden im Detail betrachtet. Hierbei werden Problemstellungen aufgrund der Änderungsoperationen identifiziert sowie notwendige Aufgaben bezüglich der Synchronisierung von Änderungen im Backend und in den Caches.

Kapitel 3

Grundsätzliche Problemstellungen

In diesem Kapitel werden Problemstellungen im Constraint-basierten Datenbank-Caching diskutiert, die aufgrund der Weiterleitung von Änderungsoperationen auftreten. Diese Probleme ergeben sich aus der Tatsache, dass Änderungen vom Backend an die Caches, oder umgekehrt, weitergeleitet werden müssen. Um einen Überblick und ein Verständnis des grundsätzlichen Verhaltens zu erlangen, werden in unterschiedlichen Szenarien die auftretenden Probleme erörtert.

Zur besseren Verständlichkeit werden zunächst die Annahmen beschrieben, welche getroffen werden, um die auftretenden Problemstellungen im Detail erläutern zu können. Danach werden grundlegende Beobachtungen aufgezeigt, die aus der Durchführung von Änderungsoperationen im Constraint-basierten Datenbank-Caching resultieren. Anschließend werden mögliche Vorgehensweisen vorgestellt, wie Änderungsoperationen im Backend und im Cache durchgeführt werden und welche Probleme bei deren Weiterleitung auftreten.

3.1 Annahmen

Um die auftretenden Probleme leichter beschreiben zu können, wird zunächst die Betrachtung kompletter Transaktionen vernachlässigt. Dies bedeutet, dass in diesem Kapitel nur einzelne Änderungsoperationen betrachtet werden, nicht jedoch die Transaktionen, in welchen sie ablaufen. Eine Analyse der kompletten Transaktionen bezüglich Serialisierbarkeit und Synchronisierung folgt in den Kapiteln 4 und 5.

Eine einzelne Änderungsoperation als kleinste Einheit einer Transaktion muss wie die Transaktion selbst gemäß der ACID[1]-Eigenschaften durchgeführt werden. Dies gilt ebenfalls für den verteilten Fall. Allerdings sind nicht notwendigerweise alle Tupel im Cache vorhanden, die von einer Änderungsoperation betroffen sind. So besteht die Möglichkeit, die Menge aller geänderten Datensätze einzeln (datensatzweise) oder als zwei Mengen (mengenweise) zu betrachten.

[1] ACID steht für Atomarität (atomicity) - Konsistenz (consistency) - Isoliertheit (isolation) - Dauerhaftigkeit (durability)

- „Datensatzweise“: Bezieht sich die Änderung auf eine Menge von Datensätzen, wird diese in Teiländerungen bezüglich der einzelnen Datensätze aufgespalten.
- „Mengenweise“: Bei der mengenweisen Betrachtung wird die Änderungsoperation hinsichtlich folgender zwei Datensatzmengen betrachtet: Einerseits die Menge aller Datensätze, die im Cache vorhanden sind und andererseits die Menge aller Datensätze, die nicht im Cache existieren.

Der Vorteil der mengenweisen Betrachtung liegt lediglich darin, dass man nicht mehrere kleinere Änderungsoperationen ausführen muss, sondern dies für eine Menge von Datensätzen durchführen kann. Um zu überprüfen, ob die von einer Änderungsoperation betroffenen Datensätze im Cache verbleiben müssen oder entladen werden können, müssen die geänderten Werte analysiert werden. Es ist allerdings zunächst einfacher, dies für einen Datensatz zu prüfen. Aus diesem Grund wird in den folgenden Überlegungen dieses Kapitels die datensatzweise Betrachtung gewählt. Dies erleichtert die Analyse für das notwendige Vorgehen bei einer Änderungsoperation, da eine solche Änderung auf einem einzigen Datensatz besser überblickt werden kann. Sind mehrere Datensätze von der Änderung betroffen, kann das beschriebene Verfahren für jeden dieser Datensätze durchgeführt werden.

Für den Austausch von Änderungsoperationen zwischen Backend und den Caches besteht die Möglichkeit, statt der Änderungsoperation ein Write Set zu verschicken. Ein Write Set sollte mindestens folgende Informationen für jedes, von der Änderungsoperation betroffene, Datenobjekt bereitstellen:

- Identifikator für das Datenobjekt
- Wertänderungen, die für das Datenobjekt durchzuführen sind, z. B. Attribut = neuer Wert

Um zu überprüfen, ob die korrekte Version des Datenobjekts vorliegt, könnte zusätzlich der alte Wert des zu ändernden Attributs im Write Set gespeichert werden. Da der Cache nur eine Teilmenge der Backend-Daten abbildet, kann der Cache nicht zwingend alle Änderungsstatements auswerten und somit auch nicht ausführen. In diesem Fall ist es sinnvoller, ein Write Set vom Backend an die Caches zu schicken, so dass der Cache die Änderungsoperation ohne weiteres Wissen ausführen kann. Weiterhin besteht die Möglichkeit, dass die Nachricht des Write Set deutlich kleiner ist, als die Nachricht mit dem kompletten Änderungsstatement.

3.2 Grundlegende Beobachtungen

In diesem Abschnitt werden grundlegende Beobachtungen bezüglich der Durchführung einer Änderungsoperation im Cache vorgestellt. Hierbei steht nicht der Ablauf einer Operationsverarbeitung im Vordergrund, sondern zu überprüfende Eigenschaften während der Änderungsausführung.

Für die im Cache definierten Cache Groups und deren Cache Constraints wird die allgemeine Tabellenrepräsentation aus Abbildung 3.1 gewählt. Im Ca-

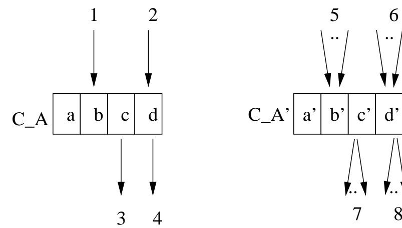


Abbildung 3.1: Allgemeine Form einer Tabelle im Cache

che müssen vor und nach den Datenänderungen alle vorhandenen RCCs erfüllt sein. Um dies zu erreichen, sind gegebenenfalls zusätzliche Aktionen notwendig, wie beispielsweise das Nachladen von Datensätzen. Die Tabellenrepräsentation aus Abbildung 3.1 veranschaulicht alle Varianten ein- und ausgehender RCCs, die in einer Cache-Tabelle auftreten können. Des Weiteren können mit Hilfe dieser Abbildung alle auftretenden Problemstellungen veranschaulicht werden. Wird eine Änderungsoperation basierend auf einem Datensatz ausgeführt, so **bezieht sich** diese Änderung immer **nur auf eine Tabelle**. Folglich muss zunächst nur diese eine Tabelle betrachtet werden. Um nachzuschauen, welche Werte in der betrachteten Tabelle wertvollständig sind, werden andere Cache-Tabellen benötigt. Bei einem, aus der Änderungsoperation resultierendem, Nachladen von Datensätzen sind ebenfalls weitere Tabellen von Interesse. Die Nachfolgeaktionen, die anhand der ein- und ausgehenden RCCs erkannt werden können, sind:

- die Überprüfung der Daseinsberechtigung eines Datensatzes (vgl. Kapitel 3.2.1)
- das Nachladen von Datensätzen (vgl. Kapitel 3.2.2)

Weiterhin muss sichergestellt werden, dass vor und nach den Änderungsoperationen die Cache Constraints gültig sind. Die RCC-Gültigkeit wird in Kapitel 3.2.3 im Detail diskutiert. Darüber hinaus wird in Kapitel 3.2.4 darauf hingewiesen, unter welchen Bedingungen eine Änderungsoperation im Cache akzeptiert wird und welche Probleme daraus resultieren können.

3.2.1 Daseinsberechtigung von Datensätzen

Eine Tabelle im Cache enthält mindestens einen eingehenden RCC. Dies folgt daraus, dass entweder eine Beziehung zwischen zwei Tabellen oder eine Beziehung zwischen einer Kontrollspalte und einer Tabelle existiert. Eine Tabelle ohne eingehende RCCs (siehe Abbildung 3.2) ist somit im Cache nicht erlaubt, da in

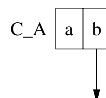


Abbildung 3.2: Tabelle ohne eingehende RCCs (für den Cache ausgeschlossen)

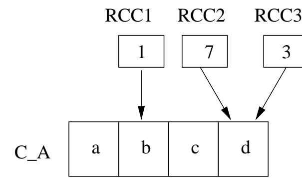


Abbildung 3.3: Beispiel zur Verdeutlichung der Daseinsberechtigung

diesem Fall für die komplette Tabelle keine Daseinsberechtigung im Cache existiert. Aufgrund der Existenz eingehender RCCs erhalten Datensätze im Cache ihre Daseinsberechtigung und es kann folgende Definition der Daseinsberechtigung abgeleitet werden:

Seien a_i die Attribute eines Datensatzes und w_i die Werte der entsprechenden Attribute. Zusätzlich seien r_j die eingehenden RCCs, die auf der Tabelle des betrachteten Datensatzes definiert sind. Ein Datensatz DS erhält seine Daseinsberechtigung Ex unter folgender Bedingung:

$$Ex_{DS} \Leftrightarrow \forall r_j \; \exists i, \text{ so dass}$$
$$a_i \text{ ist Zielspalte von } r_j \text{ und } w_i \in \text{Quellspalten-Werte von } r_j$$

Dies bedeutet, dass ein Datensatz eine Daseinsberechtigung für den Cache genau dann besitzt, wenn mindestens ein eingehender RCC dessen Existenz im Cache fordert.

Eine Überprüfung der Daseinsberechtigung findet beispielsweise bei Datensätzen statt, die von einer Änderungsoperation betroffen sind. Für einen geänderten Datensatz wird folglich geprüft, ob mindestens ein eingehender RCC die Existenz des geänderten Datensatzes im Cache fordert. Für die Überprüfung der Daseinsberechtigung aktualisierter Datensätze stehen folgende Ansätze zur schnelleren Ausführung zur Verfügung:

- Existiert mindestens ein eingehender RCC, dessen Zielspalte nicht von der Änderungsoperation betroffen ist, so existiert auch weiterhin die Daseinsberechtigung für den geänderten Datensatz.

  Beispielsweise wird folgende Änderungsoperation im Szenario von Abbildung 3.3 ausgeführt:

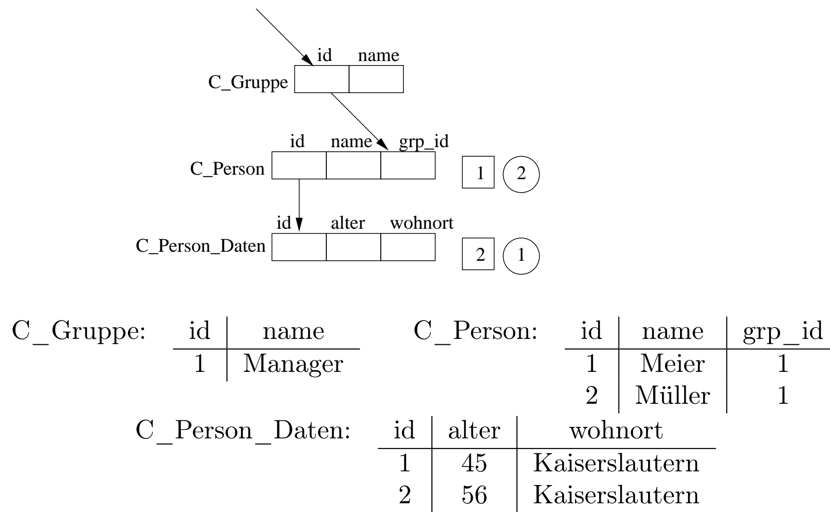
  ```
  update C_A set a = 2 where c = 3
  ```

  In diesem Fall behält der geänderte Datensatz seine Daseinsberechtigung, da sich die Zielspalten-Werte der drei vorhandenen RCCs nicht verändert haben.

- Existieren auf einer Spalte mehrere eingehende RCCs, so können die Werte dieser RCCs zusammengefasst werden.

  Es wird z. B. folgende Änderungsoperation im Cache von Abbildung 3.3 ausgeführt:

  ```
  update C_A set b = 2, d = 3 where c = 4
  ```



C_Gruppe:

id	name
1	Manager

C_Person:

id	name	grp_id
1	Meier	1
2	Müller	1

C_Person_Daten:

id	alter	wohnort
1	45	Kaiserslautern
2	56	Kaiserslautern

Abbildung 3.4: Beispieltabellen im Backend und im Cache

In diesem Fall können die Werte der Quellspalten von RCC2 und RCC3 in einer Wertmenge zusammengefasst und die betrachteten RCCs als ein RCC angesehen werden. Für die betrachtete Änderungsoperation muss somit der zusammengefasste RCC (RCC2, RCC3) und RCC1 überprüft werden, um eine Entscheidung bezüglich der Daseinsberechtigung des geänderten Datensatzes treffen zu können.

3.2.2 Nachladen von Datensätzen

Das Nachladen von Datensätzen wird ausgelöst, sobald neue Werte in die RCC-Quellspalten hinzugefügt werden. Damit die Cache Constraints gültig bleiben, müssen die neuen Werte der RCC-Quellspalten in den entsprechenden RCC-Zielspalten wertvollständig gemacht werden. Dies geschieht, indem in die RCC-Zieltabellen notwendige Datensätze nachgeladen werden. Existiert auf einer Spalte mindestens ein ausgehender RCC (vgl. Abbildung 3.1: Spalte c oder c') und es wird ein Nachladen von Datensätzen initiiert, so muss für jeden einzelnen ausgehenden RCC das Nachladen der Datensätze ausgelöst werden.

Aufgrund der Realisierung des Nachladens im Prototyp *ACCache* können je nach Vorgehensweise Dateninkonsistenzen auftreten. Dieses Problem wird anhand des folgenden Beispiels verdeutlicht. Es wird top-down ausgehend von den Kontrollspalten analysiert, welche Datensätze in den Cache geladen werden müssen. Allerdings werden diese beim eigentlichen Laden der Datensätze bottom-up in den Cache geladen. Zur Veranschaulichung des Nachladeproblems dient Abbildung 3.4. Es wird beispielsweise folgender Datensatz in die Tabelle *Gruppe* neu eingefügt:

```
insert into C_Gruppe values (3, 'Programmierer')
```

Sobald der Cache diese Änderung ausgeführt hat, wird das Nachladen von Datensätzen initiiert. Hierbei werden die Wertvollständigkeiten der Zielspalten aller

ausgehenden RCCs ausgehend von Tabelle *C_ Gruppe* überprüft. Die entsprechende Reihenfolge wurde in der Abbildung 3.4 in den quadratischen Kästchen markiert. Durch diese Analyse wird festgestellt, dass in den Tabellen *C_ Person* und *C_ Person_ Daten* Datensätze nachgeladen werden müssen. Das Nachladen beginnt in diesem Beispiel bei der Tabelle *C_ Person_ Daten*. Sobald hier alle Datensätze nachgeladen wurden, wird mit dem Nachladen der Datensätze in der Tabelle *C_ Person* begonnen. Diese Nachladereihenfolge wurde in der Abbildung mit Kreisen markiert. In diesem Beispiel wird parallel zum Nachladeprozess folgende *Select*-Anfrage an den Cache gestellt:

```
select * from C_Person where grp_id = 3
```

Zunächst wird angenommen, dass die Select-Anfrage parallel zum Nachladen ausgewertet werden darf. In diesem Fall geht der Cache davon aus, dass er diese Anfrage beantworten kann, weil der Datensatz mit der $id = 3$ in der Tabelle *C_ Gruppe* vorhanden ist und der Cache aufgrund des RCC annimmt, dass dieser Wert ebenfalls in der Tabelle *C_ Person* wertvollständig ist. Unter der Annahme, dass das Nachladen der Datensätze noch nicht beendet ist, führt die Auswertung der SelectAnfrage zu einem falschen Cache-Verhalten. Die *Select*-Anfrage wird falsch beantwortet, da noch nicht alle benötigten Datensätze in der Tabelle *C_ Person* im Cache vorhanden sind. Da solch ein falsches Cache-Verhalten unerwünscht ist, stehen folgende Lösungsansätze zur Verfügung:

- Wird die Durchführung der Änderungsoperation und das Nachladen von Datensätzen innerhalb einer Transaktion ausgeführt, so entspricht dies einem synchronen Ansatz. Parallel zu dieser Transaktion sind keine Lese- oder Änderungsoperationen auf den von der Transaktion geänderten Datenelementen möglich. Somit können die soeben beschriebenen Probleme nicht auftreten.

- Wird die Änderungsoperation durch Commit abgeschlossen, bevor das Nachladen der Datensätze beendet ist, so spiegelt dies einen asynchronen Ansatz wider. Allerdings können in diesem Fall die soeben beschriebenen Dateninkonsistenzen auftreten. Um diese zu verhindern, könnten folgende Ansätze realisiert werden:

  - Eine Möglichkeit besteht in der Invalidierung aller ausgehenden RCCs, deren Quellspalten-Werte sich geändert haben. Dies hat allerdings zur Folge, dass die invalidierten RCCs nicht weiter verwendet werden können und somit auch nicht die Informationen, die mit diesen RCCs verbunden sind.

    Dieser Fall wird anhand von Abbildung 3.5 verdeutlicht. Nachdem z. B. RCC1 invalidiert wurde, wird folgende Anfrage an den Cache gestellt:

    ```
    select A.b, C.b from A, B, C
    where A.b = B.a and B.a = C.a and A.b = 5
    ```

    Diese Anfrage kann im Cache nicht ausgeführt werden, da der benötigte RCC1 nicht nutzbar ist. Somit können die Daten der Tabellen B und C dieser Anfrage nicht zugeordnet werden, obwohl gegebenenfalls alle benötigten Datensätze im Cache existieren.

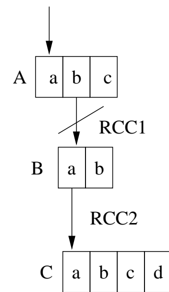


Abbildung 3.5: Beispiel für die Invalidierung von RCCs

- Aus dem soeben beschriebenen Beispiel folgt ein weiterer Lösungsansatz. Man könnte die RCCs nur für die Werte invalidieren, aufgrund derer ihre Gültigkeit verloren geht. Dies hätte zur Folge, dass nur Joins mit gültigen Werten durchgeführt werden dürfen. Allerdings muss hierfür die Implementierung der Join-Durchführung angepasst werden, um sicherzustellen, dass ungültige Werte für einen Join ausgeschlossen werden.
- Eine weitere Möglichkeit besteht darin, in den Cache-Tabellen zusätzliche Informationen bezüglich der Werte der RCC-Quellspalten zu speichern. Dies bedeutet, dass für jeden Wert einer RCC-Quellspalte im Datensatz selbst gespeichert wird, ob entsprechende Datensätze in den RCC-Zieltabellen vorhanden sind. Dies hätte allerdings wiederum zur Folge, dass die Implementierung der Join-Durchführung angepasst werden müsste.

Da die Realisierung des asynchronen Ansatzes sehr aufwändig ist und diesbezüglich weitere Analysen notwendig sind, wird der synchrone Ansatz für die folgenden Betrachtungen angenommen.

3.2.3 Gültigkeit der Cache Constraints

Wie bereits erwähnt, müssen die Cache Constraints zu jeder Zeit gültig sein. Allerdings haben Änderungsoperationen gegebenenfalls Auswirkung auf die Wertvollständigkeit der RCC-Quellspalten. Aus diesem Grund wird im Folgenden gezeigt, dass die Wertvollständigkeit und die Gültigkeit der RCCs vor und nach der Änderung erhalten bleiben.

- Soll ein neuer Datensatz eingefügt werden, muss zunächst die Daseinsberechtigung für den Cache geprüft werden. Aufgrund der RCC-Quellspalten-Werte und der RCC-Eigenschaft kann festgestellt werden, welche Werte im Cache wertvollständig sind. Wird nun ein neuer Datensatz aufgrund einer existierenden Daseinsberechtigung im Cache eingefügt, ist die Wertvollständigkeit aller Werte auch nach der Änderung gegeben. Dies folgt daraus, dass durch die ausgeführte Änderungsoperation wieder alle Datensätze, die im Backend mit diesem Wert existieren, im Cache vorhanden sind. Erhält der neue Datensatz keine Daseinsberechtigung im Cache,

so ist dieser nicht notwendig, um eine Wertvollständigkeit der Werte im Cache zu garantieren, und wird folglich nicht in den Cache geladen.

- Soll ein Datensatz gelöscht werden, so wird dies immer im Cache ausgeführt, sofern dieser Datensatz dort existiert. Für alle RCC-Quellspalten-Werte galt vor dem Löschen, dass diese Werte wertvollständig im Cache waren. Dies gilt ebenfalls nach dem Löschen des Datensatzes, da dieser in Zukunft nicht mehr existiert und somit die Datenmenge für die notwendige Wertvollständigkeit lediglich verkleinert wurde. Folglich gilt vor und nach dem Löschen die Wertvollständigkeit der RCC-Quellspalten-Werte.

  Soll eine Prädikatextension im Cache gelöscht werden, so dürfen nur solche Datensätze dieser Extension gelöscht werden, die keiner anderen Prädikatextension angehören. Aufgrund dieser Voraussetzung gelten auch nach der Änderung noch alle RCCs. Ein mögliches Vorgehen zur Löschung einer Prädikatextension wurde in Kapitel 2.3 beschrieben.

- Soll ein Update ausgeführt werden, so kann dies als eine Kombination der ersten beiden Fälle (*Delete* und *Insert*) betrachtet werden. Dementsprechend gelten vor und nach einer solchen Änderung die Wertvollständigkeiten der RCC-Quellspalten-Werte.

Diese Erläuterungen können auf alle Arten der Vollständigkeiten aus Kapitel 2.2 des Constraint-basierten Datenbank-Caching übertragen werden.

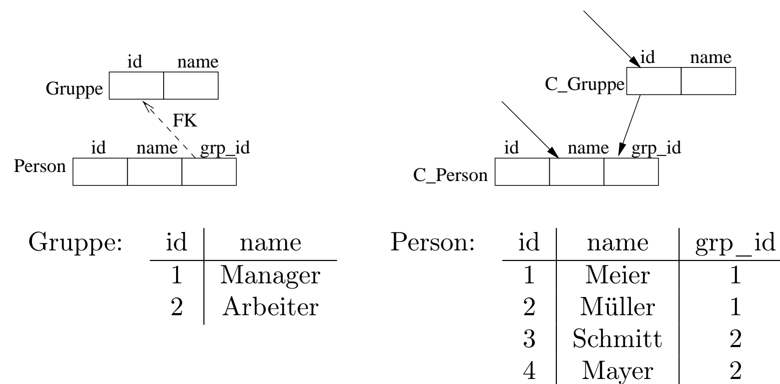
3.2.4 Akzeptanz von Änderungsoperationen im Cache

Wird eine *Select*-Anfrage an den Cache gestellt, so wird zunächst mit Hilfe der Sondierung geprüft, ob alle betroffenen Datensätze im Cache existieren. Diese Überprüfung muss ebenfalls für Änderungsoperationen im Cache erfolgen. Es kann mit derselben Sondierung geprüft werden, ob die Änderung im Cache ausgeführt werden kann oder ans Backend weitergeleitet werden muss. Da die Sondierung lediglich das Vorhandensein von Datensätzen überprüft, dürfen im Cache gegebenenfalls Änderungen ausgeführt werden, die im Backend aufgrund von Constraint-Verletzungen zurückgewiesen werden. Da z. B. Integritätsbedingungen nur im Backend definiert werden, kann der Cache nicht eigenständig entscheiden, ob eine Änderung diesen Bedingungen genügt. Ein Beispiel für eine solche Integritätsverletzung wird mit Hilfe der Abbildung 3.6 beschrieben. Es wird beispielsweise folgende *Insert*-Anweisung an den Cache gestellt:

```
insert into person (id, name, grp_id) values (5, 'Schmitt', 3)
```

Unter der Annahme, dass im Cache ebenfalls die beiden Tabellen *Gruppe* und *Person* abgebildet werden, akzeptiert der Cache diese Änderung. Im Backend wird diese Änderung allerdings zurückgewiesen, weil der neue Datensatz der Fremdschlüsselbeziehung zwischen *Person.grp_id* und *Gruppe.id* widerspricht, da keine Gruppe mit $id = 3$ definiert wurde.

Eine Lösung für dieses Problem wäre es, im Cache zusätzlich die vorhandenen Fremdschlüsselbeziehungen abzubilden. Dies würde für das Beispiel bedeuten, dass zusätzlich zum RCC zwischen *C_Gruppe.id* und *C_Person.grp.id* die Fremdschlüsselbeziehung C_Person.grp.id → C_Guppe.id abgebildet wird. Dadurch resultieren allerdings weitere Probleme, z. B. bezüglich des Ladens



Gruppe:

id	name
1	Manager
2	Arbeiter

Person:

id	name	grp_id
1	Meier	1
2	Müller	1
3	Schmitt	2
4	Mayer	2

Abbildung 3.6: Beispieltabellen im Backend

von Datensätzen. Wie bereits in Kapitel 2.3 beschrieben wurde, werden die Datensätze bottom-up in den Cache geladen, damit die Cache Constraints nicht verletzt werden. Werden allerdings ebenfalls Fremdschlüsselbeziehungen im Cache abgebildet, werden diese beim Laden von Datensätzen verletzt. Dies resultiert daraus, dass hier die Datensätze top-down in den Cache geladen werden müssten. Dies widerspricht allerdings der Vorgehensweise beim Füllen der Cache Group. Zusätzlich wären aufgrund der Fremdschlüsselbeziehungen die RCCs nicht weiter notwendig und damit auch nicht die Cache Groups.

Aufgrund dieser Analyse resultiert, dass Fremdschlüsselbeziehungen auf keinem Fall in der Cache-Datenbank angelegt werden. Es besteht jedoch die Möglichkeit diese Beziehungen im Cache zu speichern, ohne sie auf der Datenbank zu definieren. So könnten diese Bedingungen zusätzlich zur Sondierung geprüft werden.

Im Folgenden wird auf das Problem des Datenaustauschs zwischen Backend und Cache eingegangen. Hierfür wird zunächst betrachtet, wie Änderungsoperationen behandelt werden sollten, wenn sie im Backend ausgeführt werden. Danach werden diese Problemstellungen analysiert, sobald die Änderungsoperationen über den Cache durchgeführt werden.

3.3 Änderungsoperationen im Backend

In diesem Abschnitt werden alle notwendigen Konsequenzen einer Änderungsoperation ausgehend vom Backend analysiert. Diese Änderungsoperationen müssen an alle vorhandenen Cache Groups weitergeleitet werden, sofern diese von der Änderungsoperation betroffen sind. Um eine Entscheidung bezüglich der Weiterleitung der Änderung treffen zu können, kommt es darauf an, wieviel Wissen auf dem Backend bezüglich der Cache Groups vorhanden ist.

Damit die verschiedenen Wissensstände klarer unterschieden werden können, werden folgende Begriffe definiert:

- Eine *Cache-Group-Definition* beschreibt die, in einem Cache vorhandenen, Tabellen sowie die dort definierten RCCs.

- Bei dem *Cache-Group-Inhalt* sind zu den vorhandenen RCCs die entsprechenden Werte der RCC-Quellspalten bekannt.

Im Backend können folgende Wissensstände unterschieden werden:

1. Das Backend weiß lediglich von der Existenz der Caches. Es existiert allerdings weder Wissen über die *Cache-Group-Definition* noch über den *Cache-Group-Inhalt*. Das Backend kann somit nicht entscheiden, welche Tabellen oder Datensätze im Cache vorhanden sind. Dieser Ansatz wird im Abschnitt *Backend hat sehr wenig Wissen* (Kapitel 3.3.1) im Detail betrachtet.

2. Das Backend kennt die vorhandenen Caches sowie deren *Cache-Group-Definition*. Allerdings hat das Backend kein Wissen über den *Cache-Group-Inhalt*. Aufgrund des Wissens, welche Tabelle in welchem Cache abgebildet ist, kann das Backend entscheiden, ob eine Änderungsoperation an den Cache weitergeleitet werden muss. Ob in dieser Cache-Tabelle von der Änderungsoperation betroffene Datensätze existieren, kann das Backend allerdings nicht herausfinden. Dieser Ansatz wird im Abschnitt *Backend hat eingeschränktes Wissen* (Kapitel 3.3.2) analysiert.

3. In diesem Szenario kennt das Backend alle vorhandenen Caches, inklusive ihrer *Cache-Group-Definition* und ihrem *Cache-Group-Inhalt*. Aufbauend auf dieser Grundlage kann das Backend feststellen, ob von einer Änderungsoperation betroffene Datensätze im Cache vorhanden sind. In diesem Szenario sind differenzierte Entscheidungen bezüglich der Weiterleitung einer Änderungsoperation bereits im Backend möglich, die im Abschnitt *Backend hat volles Wissen* (Kapitel 3.3.3) erläutert werden.

Eine detaillierte Analyse dieser Szenarien wird in den folgenden Abschnitten durchgeführt. Hierzu wird zunächst das jeweilige Szenario mit dem vorhandenen Wissensgrad vorgestellt. Danach wird analysiert, welche Auswirkungen Änderungsoperationen haben und wie die Daten in den Caches aktuell gehalten werden können. Abschließend werden für jedes Szenario deren Vor- und Nachteile abgewägt. Welches dieser Szenarien im Synchronisierungsverfahren des Constraint-basierten Datenbank-Caching ausgewählt wird, wird bei dessen Vorstellung ab Kapitel 5.3 vorgestellt.

3.3.1 Backend hat sehr wenig Wissen

Unter der Annahme, dass das Backend lediglich die vorhandenen Caches kennt, wie dies in Abbildung 3.7 veranschaulicht wird, muss jede Änderungsoperation an alle Caches weitergeleitet werden. Dies ist notwendig, da das Backend keine Informationen über die Cache-Group-Definition und den Cache-Group-Inhalt besitzt. Somit kann das Backend keine Entscheidungen darüber treffen, ob eine Änderung für den Cache interessant ist oder nicht.

Im Backend ist es irrelevant, ob eine *Insert*-, *Delete*- oder *Update*-Änderung durchgeführt werden muss. Nach der Ausführung der Änderungsoperation erzeugt das Backend ein entsprechendes Write Set, das an die Caches weitergeleitet wird. Hierbei ist zu beachten, dass dieses Write Set an alle vorhandenen Caches weitergeleitet werden muss, da das Backend aufgrund seines geringen Wissens keine Auswahl treffen kann. Sobald die Änderungsoperation den Cache

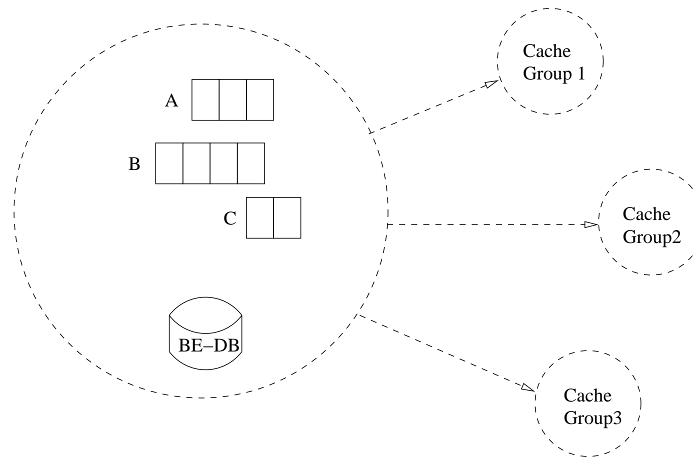


Abbildung 3.7: Wissensstand des Backend (sehr wenig Wissen)

erreicht, wird zunächst überprüft, ob von der Änderung betroffene Datensätze im Cache existieren. Ist dies der Fall, so wird die Änderungsoperation auch im Cache ausgeführt. Andernfalls wird die Änderung im Cache verworfen.

Die Vorteile dieses Szenarios liegen darin, dass das Backend lediglich das Vorhandensein der unterschiedlichen Cache Groups speichern muss. Es müssen keine Informationen bezüglich der Cache-Group-Definition und des Cache-Group-Inhalts gespeichert werden. Weiterhin muss das Backend keinerlei detaillierte Entscheidung bezüglich der Weiterleitung der Datenänderungen treffen, da alle Änderungsoperationen an alle Caches propagiert werden müssen.

Folgende Nachteile wiegen allerdings deutlich schwerer: Ein Hauptnachteil dieses Szenarios liegt im Datenaustausch. Dieser Datenaustausch basiert einerseits auf sehr langen Datenwegen zwischen Backend und Cache und andererseits auf einer schnellen Übertragungsrate. Allerdings resultiert aus dieser Kombination eine hohe Latenz, so dass dies den Datenaustausch „langsam“ werden lässt. Dadurch, dass alle Datenänderungen aus dem Backend an alle Caches weitergeleitet werden, müssen jedesmal die Nachrichten über den „langsamen“ Datenaustausch vom Backend zum Cache erfolgen. Zusätzlich wird die Weiterleitung von Änderungsoperationen teilweise unnötig getätigt, da die Caches lediglich Datenausschnitte repräsentieren und somit nicht alle Tabellen und Datensätze enthalten. Deshalb wird die Datenänderung in all den Fällen durch den Cache verworfen, in denen die Tabelle oder relevante Datensätze nicht im Cache vorhanden sind.

3.3.2 Backend hat eingeschränktes Wissen

In diesem Szenario speichert das Backend zusätzlich zu den existierenden Caches die entsprechenden Cache-Group-Definitionen. Dieser Wissensstand wird in Abbildung 3.8 veranschaulicht. Das Backend kann in diesem Szenario mit den existierenden Informationen zwar noch nicht entscheiden, ob relevante Datensätze im Cache vorhanden sind, allerdings ist bereits eine differenzierte Weiterleitung der Änderung an die Caches möglich.

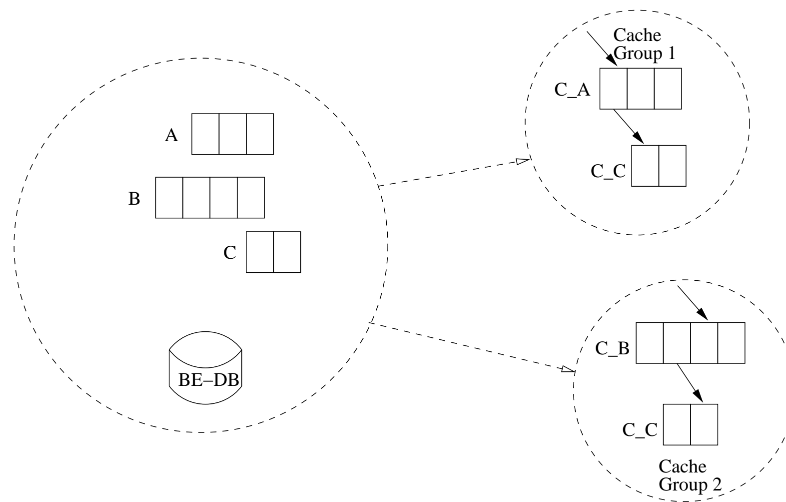


Abbildung 3.8: Wissensstand des Backend (eingeschränktes Wissen)

Sobald eine Datenänderung auf dem Backend ausgeführt wird, muss das Backend entscheiden, an welche Caches diese weitergeleitet werden muss. Bei dieser Entscheidung ist interessant, in welcher Tabelle die Datensätze geändert werden sollen. Existiert die betroffene Tabelle in einem Cache, so ist dieser Cache ein Kandidat für die Weiterleitung der Änderungsoperation. Sobald diese Entscheidung getroffen wurde, wird für die Änderungsoperation ein entsprechendes Write Set erzeugt. Dieses Write Set wird dann an die zuvor identifizierten Caches weitergeleitet. Im Cache muss dann noch überprüft werden, ob von der Änderungsoperation betroffene Datensätze vorhanden sind. Ist dies der Fall, wird die Änderungsoperation bearbeitet, andernfalls verworfen.

Die Vorteile dieses Szenarios liegen in der selektiven Auswahl der Caches, an welche die Datenänderungen vom Backend aus weitergeleitet werden. Somit muss der „langsame" Datenaustausch zwischen Backend und Caches (vgl. Kapitel 3.3.1) nur in den Fällen genutzt werden, in denen die Tabelle im Cache vorhanden ist und dort auch potentiell betroffene Datensätze existieren.

Ein Nachteil dieses Szenarios ist, dass das Backend zusätzlich die Informationen über die Cache-Group-Definitionen speichern muss. Unter der Annahme, dass dieser Mehraufwand im Vergleich zu der geringeren Nutzung des „langsamen" Datenaustauschs gerechtfertigt ist, wiegt dieser Nachteil allerdings nicht so schwer. Es kann weiterhin vorkommen, dass der „langsame" Datenaustausch unnötigerweise genutzt wird. Es ist zwar eine selektive Auswahl der Caches möglich, allerdings kann das Backend nicht feststellen, ob von der Änderungsoperation betroffene Datensätze im Cache vorliegen. Existieren diese nicht im Cache, so müsste die Datenänderung nicht an den Cache weitergeleitet werden. Um ein unnötiges Weiterleiten der Änderung zu verhindern, müssten weitere Informationen gespeichert werden, so dass das Backend analysieren kann, welche Datensätze in den Cache geladen wurden.

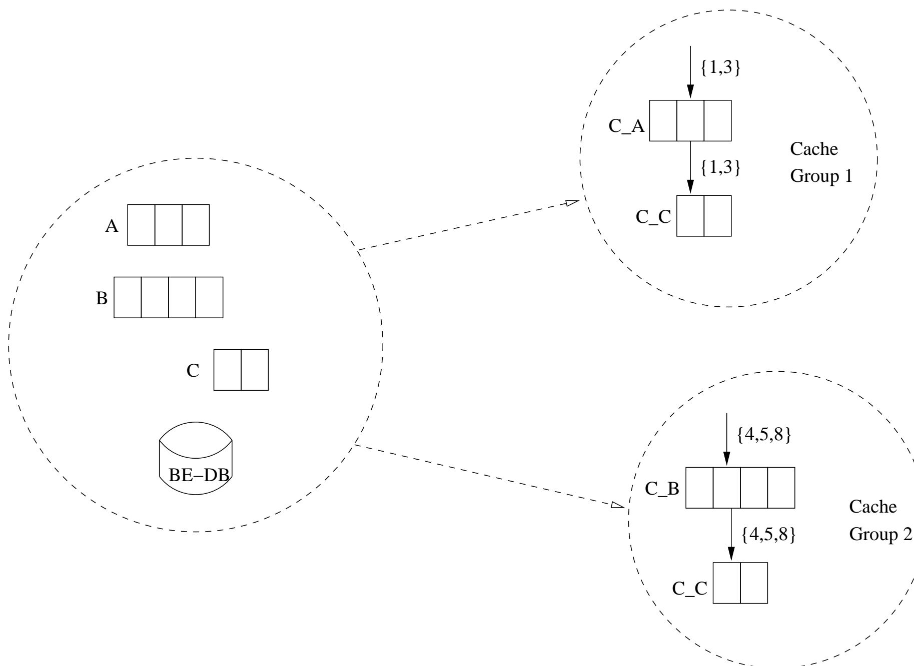


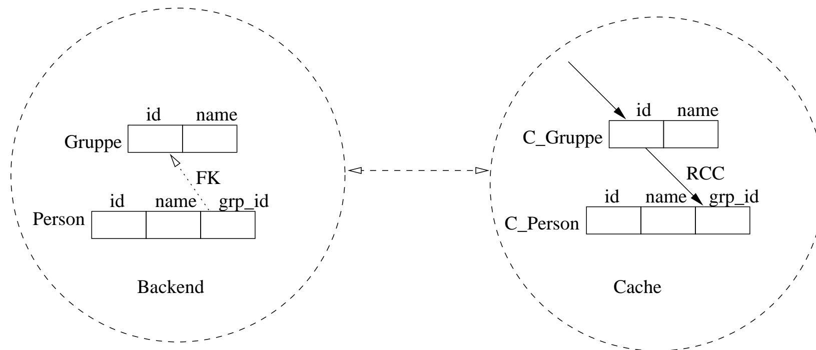
Abbildung 3.9: Wissensstand des Backend (volles Wissen)

3.3.3 Backend hat volles Wissen

In diesem Szenario besitzt das Backend detailliertes Wissen über die vorhandenen Caches. Es kennt alle vorhandenen Caches, deren Cache-Group-Definition sowie den Cache-Group-Inhalt, wie dies in Abbildung 3.9 veranschaulicht wird. Mit Hilfe dieser Informationen kann das Backend feststellen, welche Datensätze im Cache vorhanden sind. Dies ist für Spalten mit ein- oder ausgehenden RCCs einfach, da hierfür die Quellspalten-Werte der RCCs direkt gespeichert werden. Allerdings ist eine Analyse für RCC-freie Spalten deutlich schwieriger, da keine direkten Aussagen über betroffene Datensätze getroffen werden können. Eine mögliche Vorgehensweise zur Identifikation der betroffenen Datensätze kann indirekt über die Werte der RCC-Quellspalten erfolgen. Bei dieser Vorgehensweise, die anhand von Abbildung 3.10 veranschaulicht wird, werden die betroffenen Datensätze aus dem Cache im Backend als solche markiert. Unter der Annahme, dass alle Entscheidungen im Backend getroffen werden, soll folgende Änderung ausgeführt werden:

```
A1: update Person set name = 'Mueller' where name = 'Müller'
```

In diesem Fall muss das Backend zunächst entscheiden, ob von der Änderungsoperation betroffene Datensätze im Cache vorhanden sind. Da bei dieser Änderung lediglich eine RCC-freie Spalte betroffen ist, muss das Backend beispielsweise alle Datensätze markieren, die aufgrund der RCC-Quellspalten-Werte in den Cache geladen wurden. Im Beispiel sind das die beiden Datensätze aus der Tabelle *Person* mit $id = 1$ und $id = 2$. Wurde der von der Änderung $A1$ betroffene Datensatz bei der vorherigen Analyse markiert, so muss die Änderung



Gruppe:

id	name
1	Manager
2	Arbeiter

C_Gruppe:

id	name
1	Manager

Person:

id	name	grp_id
1	Meier	1
2	Müller	1
3	Schmitt	2
4	Mayer	2

C_Person:

id	name	grp_id
1	Meier	1
2	Müller	1

Abbildung 3.10: Beispieltabellen im Backend und Cache

an den Cache weitergeleitet werden. Andernfalls befindet sich dieser Datensatz nicht im Cache und muss dort nicht aktualisiert werden. Im Beispiel wurde der betroffene Datensatz markiert und muss im Cache aktualisiert werden. Diese Analyse kann folglich nur indirekt über die RCC-Quellspalten-Werte erfolgen und ist somit deutlich aufwändiger als eine Analyse auf Spalten mit ein- oder ausgehenden RCCs.

Trifft eine Änderung im Backend ein, so wird diese zunächst dort verarbeitet. Für den nächsten Schritt stehen folgende Alternativen zur Verfügung:

- Das Backend leitet die Änderung an alle zuvor ausgewählten Caches weiter. In den Caches wird dann, wenn möglich, die Änderung ausgeführt. Wird die Entscheidung über die Weiterleitung der Änderungsoperation nur aufgrund der vorhandenen Cache-Group-Definition im Backend getroffen, so entspricht dieser Ansatz dem in Kapitel 3.3.2 beschriebenen Szenario. Darin enthält das Backend lediglich ein eingeschränktes Wissen über den Cache.

- Das Backend trifft zusätzlich zur Cacheauswahl alle weiteren detaillierten Entscheidungen bezüglich der Weiterleitung der Änderungsoperation. Dies bedeutet, dass das Backend die im Cache vorhandenen RCC-Quellspalten-Werte analysiert und entscheidet, ob der geänderte Datensatz im Cache eine Daseinsberechtigung erhält. Ist diese gegeben, kann das Backend die Änderung an den Cache weiterleiten. Andernfalls weiß das Backend, dass dieser Datensatz aus dem Cache entladen werden kann und schickt die entsprechende Nachricht anstatt der Änderungsnachricht. Das Backend überprüft ebenfalls die ausgehenden RCCs, so dass entschieden werden

kann, ob Datensätze in den Cache nachgeladen werden müssen. Ist dies der Fall, so kann das Backend diese benötigten Datensätze zusammen mit der Änderungsnachricht schicken, wodurch die Anzahl der Nachrichten zwischen Backend und Cache reduziert werden kann. Da dies alles durch das Backend entschieden werden kann, muss der Cache anschließend nur noch die Änderung ausführen.

Die Vorteile dieses Szenarios liegen darin, dass das Backend ein sehr detailliertes Wissen über alle vorhandenen Caches enthält. Das Backend kann für jeden Cache und jede Änderungsanweisung im Detail überprüfen, ob und wie diese Änderungsoperation an den Cache weitergeleitet werden muss. Beispielsweise können direkt mit der Änderungsanweisung die Datensätze mitgeschickt werden, welche in den Cache nachgeladen werden müssen. So kann der Nachrichtenaufwand über den „langsamen" Datenaustausch zwischen Backend und Cache reduziert werden.

Ein Nachteil dieses Szenarios liegt darin, dass das Backend entsprechend viele Informationen über alle vorhandenen Caches speichern muss. Weiterhin ist eine sehr detaillierte Anfrageanalyse durch das Backend notwendig, wodurch im Backend weiterer Aufwand entsteht. Zusätzlich ist eine optimale Speicherung dieser Zusatzinformationen notwendig, damit die Entscheidungen bezüglich der Änderungsweiterleitung effizient getroffen werden können.

3.4 Änderungsoperationen im Cache

In diesem Abschnitt werden alle notwendigen Aktionen einer Änderung ausgehend vom Cache analysiert. Dies bedeutet, dass Änderungsoperationen von der Anwendung direkt an den Cache gestellt werden und von dort aus weitergeleitet werden müssen. Ein möglicher grundsätzlicher Wissensstand eines Cache wird in Abbildung 3.11 veranschaulicht. Da dieser Wissensstand die Änderungspropagierung beeinflusst, sind unterschiedliche Szenarien möglich. In allen Fällen wird die Änderungsoperation zunächst an den Cache gestellt, welcher diese weiterleiten muss. Somit fungiert der Cache als Koordinator für alle Änderungen.

- Der Cache kennt nur das Backend. In diesem Fall kann der Cache die Änderungsoperation nur an das Backend weiterleiten. Folglich muss das Backend die Änderungsoperation an alle anderen existierenden Caches propagieren.

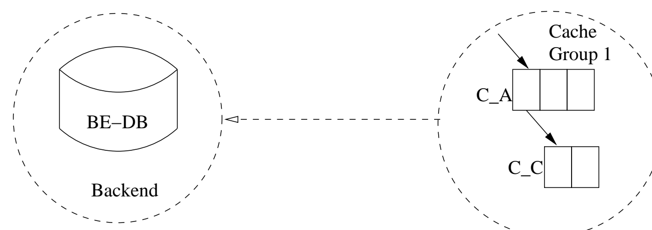


Abbildung 3.11: Wissensstand des Cache

- Die Caches sind im Cluster angeordnet. In diesem Fall kennt der Cache neben dem Backend weitere Caches. Sobald eine Änderungsoperation an den Cache gestellt wird, kann dieser Cache die Änderungsoperation an die ihm bekannten Caches sowie an das Backend weiterleiten.

Weiterhin muss unterschieden werden, wo Änderungsoperationen ausgeführt werden dürfen. Bezüglich dieser Unterscheidung wird angenommen, dass die Caches nicht im Cluster angeordnet sind. Es können folgende Alternativen unterschieden werden.

- Der Cache darf die Änderungsoperation ausführen, sofern von der Änderung betroffene Datensätze im Cache existieren. Danach wird in jedem Fall die Änderungsoperation an das Backend weitergeleitet. Dies entspricht einem indirekten Primary-Copy-Ansatz, da die Änderungen ebenfalls im Backend durchgeführt werden und von dort aus alle restlichen Caches aktualisiert werden.

- Der Cache kann lediglich *Select*-Anfragen beantworten und muss durchzuführende Änderungsoperationen an das Backend weiterleiten, wo sie ausgeführt werden. Nachdem die Änderungsoperationen bearbeitet wurden, werden alle Caches vom Backend aus aktualisiert. Dies entspricht dem Primary-Copy-Ansatz.

In der folgenden Analyse bezüglich der Weiterleitung von Änderungsoperationen wird angenommen, dass die Caches lediglich das Backend kennen und keinerlei Beziehungen zu anderen Caches haben, wie dies bereits im Prototyp *ACCache* realisiert ist. Weiterhin wird davon ausgegangen, dass der Cache die Änderungsoperationen ausführt, sofern die benötigten Datensätze im Cache vorhanden sind.

3.4.1 Ausführung von Änderungsoperationen

Im weiteren Verlauf wird die grundsätzliche Änderungsdurchführung im Cache betrachtet. Bei diesem Vorgehen wird die datensatzweise Betrachtung angenommen, um die notwendigen Aktionen genau beschreiben zu können. Weiterhin wird unterstellt, dass der Cache selbstständig entscheiden kann, ob relevante Datensätze vorhanden sind. Dies bedeutet, dass die Änderungen beispielsweise in ihrer *Where*-Klausel nur auf Tabellen verweisen, die auch im Cache vorhanden sind.

Update

Ist ein von einer Änderungsoperation betroffener Datensatz im Cache vorhanden, muss die Daseinsberechtigung des geänderten Datensatzes sowie deren Folgeauswirkungen überprüft werden.

1. Zur Überprüfung der Daseinsberechtigung werden zunächst die eingehenden RCCs betrachtet.
   (a) Ist eine Daseinsberechtigung für den geänderten Datensatz gegeben, so muss die Änderung im Cache ausgeführt werden.
   (b) Ist keine Daseinsberechtigung für den geänderten Datensatz gegeben, so stehen folgende Alternativen zur Auswahl:

  - Die Änderungsoperation wird dennoch im Cache ausgeführt.
  - Die Änderungsoperation wird nicht im Cache durchgeführt und der von der Änderungsoperation betroffene Datensatz wird aus dem Cache geladen.

2. Wird die Änderungsoperation ausgeführt und existieren ausgehende RCCs auf Spalten, deren Werte sich geändert haben, so wird der Prozess des Nachladens initiiert.

Insert

Soll ein neuer Datensatz im Cache eingefügt werden, muss zunächst dessen Daseinsberechtigung für den Cache geprüft werden. Sobald mindestens ein RCC den Datensatz im Cache verlangt, muss dieser dort eingefügt werden. Andernfalls kann der Cache diese Änderungsoperation verwerfen. Wurde die Änderungsoperation durchgeführt, so kann es aufgrund ausgehender RCCs notwendig sein, dass neue Datensätze in den Cache nachgeladen werden müssen.

Delete

Wird eine *Delete*-Anweisung an den Cache gestellt, so muss dieser zunächst überprüfen, ob ein von der Änderungsoperation betroffener Datensatz im Cache vorhanden ist. Ist dies der Fall, muss der Cache diese Änderungsoperation ausführen. Zusätzlich könnten zu dem betroffenen Datensatz alle Datensätze aus dem Cache entladen werden, welche aufgrund dieses gelöschten Datensatzes ihre Daseinsberechtigung verloren haben. Ist der Datensatz nicht im Cache vorhanden, kann der Cache diese Änderung verwerfen.

Nachdem die Änderungsoperation (Insert, Update oder Delete), sofern möglich, im Cache bearbeitet wurde, muss sie an das Backend weitergeleitet werden. So kann das Backend diese Änderungsoperation durchführen und an alle notwendigen Caches weiterleiten.

3.4.2 Probleme bei der Ausführung von Änderungsoperationen

Dadurch, dass der Cache lediglich Datenausschnitte aus dem Backend abbildet, können folgende Probleme auftreten:

- Da der Cache beispielsweise keine Fremdschlüsselbeziehungen abbildet und lediglich Datenausschnitte speichert, kann er nicht erkennen, ob eine Änderung den Integritätsbedingungen des Backend genügt. So besteht die Möglichkeit, dass der Cache eine Änderung ausführt, das Backend diese jedoch zurückweist, weil beispielsweise eine Fremdschlüsselbeziehung nach dieser Änderung nicht erfüllt werden kann.

  Obwohl die Fremdschlüsselbeziehungen nicht im Cache abgebildet werden können, besteht dennoch die Möglichkeit diese Beziehungen im Cache zu speichern, jedoch nicht in der Cache-Datenbank abzubilden. Diese Bedingungen müssten dann zusätzlich überprüft werden, bevor eine Änderungsoperation ausgeführt werden kann.

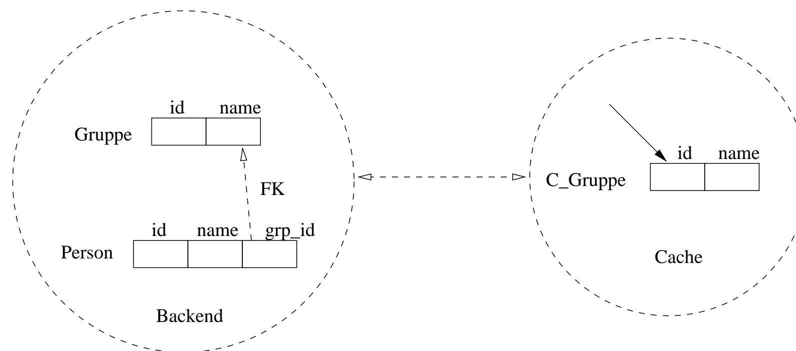


Abbildung 3.12: Beispielszenario zu: Eine Anfrage kann aufgrund der Where-Klausel nicht im Cache ausgewertet werden

- Ein weiteres Problem stellen Anfragen dar, die in ihrer *where*-Klausel auf Tabellen verweisen, die nicht im Cache abgebildet werden. Als Beispiel dient Abbildung 3.12 und es soll folgende Änderung ausgeführt werden:

  ```
  delete from Gruppe where Gruppe.id not in (
      select distinct(Person.grp_id) from Person)
  ```

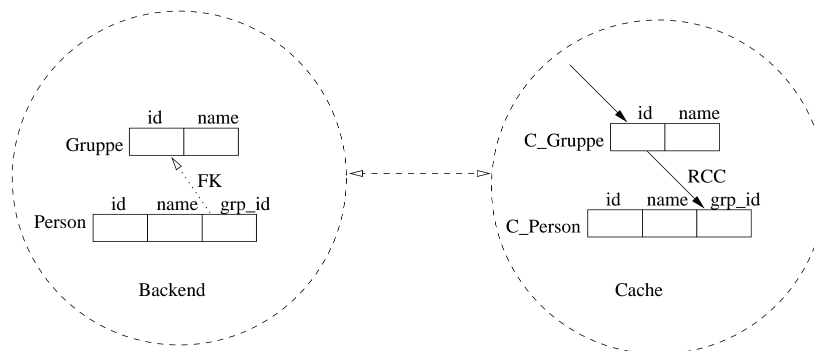
  In diesem Fall wird die Tabelle *Person* nicht im Cache abgebildet, während die Tabelle *Gruppe* dort existiert. Allerdings kann der Cache anhand dieser Änderungsoperation nicht erkennen, ob betroffene Datensätze vorliegen, da er keine Informationen bezüglich der Tabelle *Person* enthält. Somit muss die Änderung zunächst an das Backend weitergeleitet werden. Dort wird ein entsprechendes Write Set erstellt, das durch den Cache ausgewertet werden kann.

- Weiterhin stellen kaskadierende Änderungsoperationen ein Problem für den Cache dar. Aufgrund einer solchen Änderungsoperation sind Folgeänderungen notwendig, wenn zum Beispiel ein *on delete set null/on delete cascade* auf einem Fremdschlüssel definiert wurde. Diese Auswirkungen kennt der Cache nicht und kann sie folglich nicht ausführen. Aufgrund des im Backend erstellten Write Set dieser Änderung kann der Cache erkennen, welche Datensätze neben der bereits durchgeführten Änderungsoperation betroffen sind.

- Es besteht die Möglichkeit, dass Änderungsoperationen auf Datensätze verweisen, die nicht im Cache existieren, allerdings aufgrund dieser Aktualisierung und den Cache Constraints in den Cache geladen werden müssen. Ein Beispiel hierfür wird in Abbildung 3.13 dargestellt. Es liegen die aufgezeigten Datensätze in den Tabellen von Cache und Backend vor. Unter dieser Voraussetzung wird folgende Änderung ausgeführt:

  ```
  update person set grp_id = 1 where id = 3
  ```

  Die Änderung kann nicht vom Cache selbstständig ausgeführt werden, da der relevante Datensatz dort nicht existiert. Zudem ist aufgrund dieser



Gruppe:

id	name
1	Manager
2	Arbeiter

C_Gruppe:

id	name
1	Manager

Person:

id	name	grp_id
1	Meier	1
2	Müller	1
3	Schmitt	2
4	Mayer	2

C_Person:

id	name	grp_id
1	Meier	1
2	Müller	1

Abbildung 3.13: Beispieltabellen im Backend und Cache

Änderung der RCC und damit die Wertvollständigkeit des Wertes 1 der Spalte *grp_id* nicht erfüllt, da jetzt ebenfalls der Datensatz mit der $id = 3$ aus der Tabelle *Person* im Cache vorhanden sein müsste.

Um diese Probleme umgehen zu können, wäre es die einfachste Möglichkeit, alle Änderungsoperationen nur im Backend ausführen zu lassen. Allerdings würde dies bedeuten, dass lediglich Leseoperationen im Cache ausgewertet werden können und somit nicht alle Nutzungsvorteile des Cache zur Geltung kommen.

Die in diesem Kapitel beschriebenen Problemstellungen werden in den folgenden Kapiteln wieder aufgegriffen und durch weitere Problemstellungen aus dem Bereich der Synchronisierung von Transaktionen (Kapitel 4) ergänzt. In Kapitel 5 wird ein konkretes Synchronisierungsverfahren für das Constraint-basierte Datenbank-Caching vorgestellt sowie Lösungsansätze für die in diesem Kapitel vorgestellten Problemstellungen.

Kapitel 4

Synchronisierung von Replikaten

Durch das entfernte Zwischenspeichern von Datensätzen entstehen Replikate, die innerhalb von Transaktionen synchronisiert werden müssen. Dadurch können die Daten auf allen Replikaten aktuell gehalten werden. Darüber hinaus kann durch die Synchronisierung der Replikate garantiert werden, dass Anwendungen auf korrekte Datenzustände zugreifen.

In diesem Kapitel werden zunächst Problemstellungen der Synchronisierung vorgestellt, die aufgrund einer verteilten Ausführung von Transaktionen resultieren. Darauf folgend werden Kategorien möglicher Verfahren vorgestellt, die zur Synchronisierung verwendet werden können. Diese Verfahren werden danach bezüglich ihren Einsatzmöglichkeiten im Constraint-basierten Datenbank-Caching analysiert. Hierzu wird zunächst die Architektur des Gesamtsystems betrachtet, die durch Ziele der Synchronisierung ergänzt werden. Abschließend werden konkrete Synchronisierungsverfahren vorgestellt.

4.1 Problemstellungen der Synchronisierung

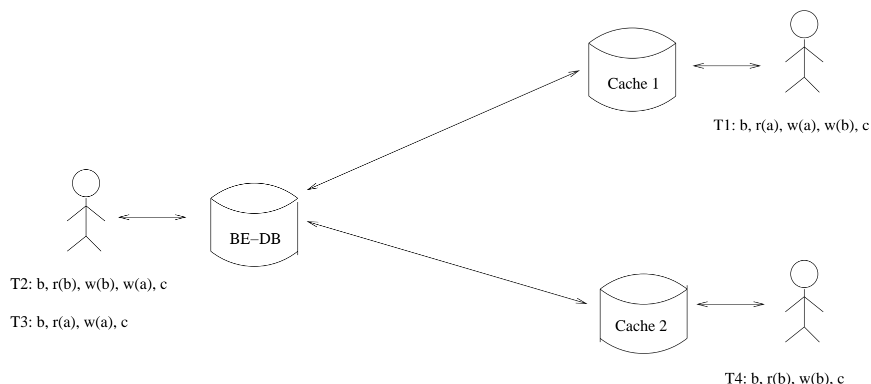


Abbildung 4.1: Grundaufbau des Gesamtsystems

In Abbildung 4.1 wird ein Grundaufbau des Gesamtsystems veranschaulicht, mit dessen Hilfe Probleme der Synchronisierung im verteilten Fall erörtert werden. *Cache1* und *Cache2* stehen hier repräsentativ für alle vorhandenen Caches. Zu beachten ist jedoch, dass jeder Cache lediglich mit dem Backend und dem jeweiligen Benutzer kommunizieren kann. Ein Benutzer führt eine Transaktion immer nur über einen Cache durch. Dies bedeutet, dass alle Operationen einer Transaktion nur über einen Cache und nicht verteilt über mehrere Caches ausgeführt werden.

Um z. B. auf Konflikte zwischen Transaktionen klarer hinweisen zu können, werden in den Transaktionsabfolgen folgende Standardformalien verwendet: $r_1(a)$ entspricht einem Lesezugriff auf dem Datenelement a, der in Transaktion T1 ausgeführt wird. $w_2(a)$ ist analog ein Schreibzugriff auf dem Datenelement a der Transaktion T2.

Mit Hilfe des in Abbildung 4.1 dargestellten Grundaufbaus werden Problemstellungen aufgezeigt, die bei der Synchronisierung von Replikaten zu erkennen und zu lösen sind. Folgende Problemstellungen und grundsätzliche Fragestellungen bezüglich der Synchronisierung können identifiziert werden:

1. Welche Informationen einer Transaktion werden zwischen Backend und Cache ausgetauscht? Beispielsweise kann es sinnvoll sein, lediglich Änderungsoperationen weiterzuleiten. Zusätzlich ist von Interesse, in welcher Form Änderungsoperationen ausgetauscht werden, z. B. die unveränderten Statements oder die entsprechenden Write Sets.

2. Auf welchen Knoten dürfen Änderungsoperationen ausgeführt werden? Diese Fragestellung bezieht sich im Speziellen auf die Replikationskontrolle.

3. Wann werden Transaktionen synchronisiert? Werden Replikate vor oder nach dem Commit einer Transaktionen aktualisiert?

4. Weitere Problemstellungen ergeben sich durch den Mehrbenutzerbetrieb, da hierdurch sogenannte Anomalien[1] auftreten können. Diese Anomalien können sein: *Dirty Read/Write, Inconsistent Read, Phantom Problem, Lost Update, Read Skew* oder *Write Skew* [HR01, BBG+95].

5. Zusätzlich besteht die Möglichkeit, dass Deadlocks zwischen Transaktionen auftreten. In Abbildung 4.3 wird diese Möglichkeit durch die Transaktionen $T1$ und $T2$ angedeutet. Je nachdem, in welcher Reihenfolge die Operationen dieser beiden Transaktionen ausgeführt und welche Synchronisierungsverfahren angewendet werden, besteht die Gefahr eines Deadlock.

4.2 Kategorien der Synchronisierungsverfahren

Da durch das entfernte Speichern von Datensätzen Replikate entstehen, liegt im Folgenden der Fokus auf Prinzipien der Replikationskontrolle. Zu den Vorteilen einer Verwendung von Replikaten gehört die höhere Verfügbarkeit und ein effizienter Zugriff auf die Datenelemente. Allerdings führen Veränderungen

[1] Eine Erklärung der Anomalien befindet sich in Anhang A.

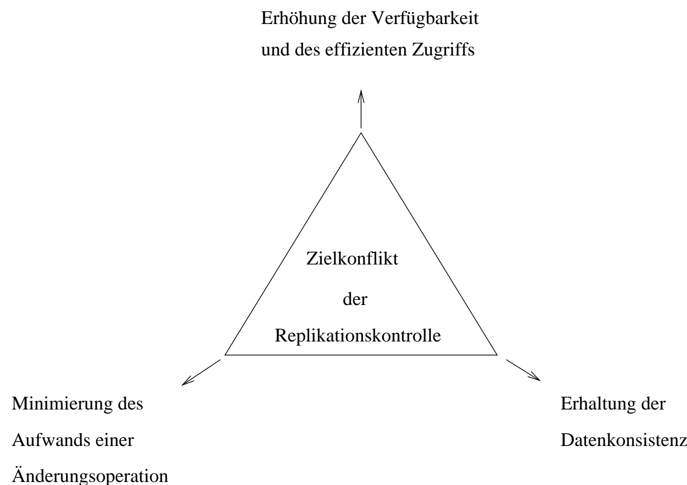


Abbildung 4.2: Zielkonflikt der Replikationskontrolle

von Objekten zu einem höheren Kommunikations- und Verwaltungsaufwand, da das Objekt in der Regel auf mehreren Knoten aktualisiert werden muss [BD96]. Basierend auf diesen Vor- und Nachteile können die in Abbildung 4.2 dargestellten Ziele der Replikationskontrolle definiert werden, die zu einem Zielkonflikt führen. Synchronisierungsverfahren versuchen diese Ziele weitestgehend umzusetzen und können wie folgt klassifiziert werden:

- Welche Konfliktarten werden erkannt und welches Korrektheitskriterium wird verwendet? ⇒ Syntaktische vs. semantische Verfahren
- Wie und wann werden Konflikte erkannt? ⇒ Optimistische vs. pessimistische Verfahren
- Wann werden Konflikte zwischen Transaktionen behandelt? ⇒ Synchrone vs. asynchrone Verfahren

Bei syntaktischen Verfahren wird die Korrektheit eines Zugriffs über die Reihenfolge der zugreifenden Transaktionen bestimmt [BD96]. Eine syntaktische Konflikterkennung wird durch die meisten Synchronisierungsverfahren unterstützt.

Semantische Verfahren hingegen nutzen die Semantik der Transaktionen aus, um z. B. mit Hilfe der Kommutativität der Operationen oder der Semantik der Datenbankdaten die Anzahl zulässiger Ausführungsreihenfolgen zu erhöhen. Semantische Verfahren sind allerdings nicht so universell einsetzbar wie syntaktische Verfahren, da nicht jede Anwendung die notwendigen Voraussetzungen, wie z. B. kommutative Datenmanipulationsoperationen, erfüllt. Allerdings sind diese Verfahren besonders für verteilte Systeme attraktiv, da hier der Kommunikations- und Synchronisierungsaufwand reduziert werden kann [BD96].

Bei pessimistischen Verfahren werden bereits während der Transaktionsausführung Konflikte erkannt. Ein Beispiel für ein pessimistisches Verfahren stellt das *2-Phase-Locking* dar. Hierbei werden die Sperren der Datenelemente in einer

ersten Phase angefordert und in einer zweiten Phase wieder freigegeben. Durch die Sperranforderungen können frühzeitig Konflikte zwischen Transaktionen erkannt werden. Wird das 2-Phase-Locking nur auf einem Replikat angewendet, können Mehrbenutzeranomalien und sogar Deadlocks durch das Datenbanksystem erkannt und aufgelöst werden. Im verteilten Fall kann dies zwar nicht immer garantiert werden, dennoch können auch hier Konflikte frühzeitig erkannt werden.

Optimistische Verfahren prüfen erst nach der vollständigen Ausführung einer Transaktion, ob Konflikte mit anderen Transaktionen auftreten. So wird im zentralen Fall zunächst die Lesephase durchlaufen, in der alle Operationen einer Transaktion lokal ausgeführt werden. Erst nach dieser Phase wird geprüft, ob Konflikte mit anderen Transaktionen existieren. Ist dies der Fall, so wird die Transaktion abgebrochen. Andernfalls werden die Änderungen in der Schreibphase dauerhaft gemacht. Wird ein verteiltes Datenbanksystem verwendet, so ist hier die Validierungsphase von besonderem Interesse, da diese auf allen Knoten durchgeführt werden muss, auf denen Operationen einer Transaktion verarbeitet wurden. Es sind folgende Validierungsschemata möglich [TR90]:

- Zentrales Validierungsschema: Alle Transaktionen werden sequentiell auf einem zentralen System bearbeitet und dort validiert. Dies hat zur Folge, dass zusätzliche Nachrichten notwendig sind, um die Transaktionen im zentralen System abwickeln zu können. Weiterhin ist das zentrale System ein potentieller Flaschenhals.

- Verteiltes Validierungsschema: Die Transaktionen werden auf allen Knoten validiert, auf denen sie ausgeführt wurden. Ist von einer Transaktion nur ein Knoten betroffen, so sind keine Nachrichten bezüglich der Validierung auf anderen Knoten notwendig. Andernfalls muss die Validierung auf all diesen Knoten erfolgen, wobei die Validierung z. B. durch das 2-Phasen-Commit-Protokoll unterstützt werden kann.

Im Gegensatz zu pessimistischen Verfahren kann mit optimistischen Verfahren potentiell eine höhere Parallelität und ein höherer Durchsatz an Transaktionen erreicht werden [TR90]. Allerdings werden im Vergleich zu pessimistischen Verfahren häufiger Transaktionen zurückgesetzt. Des Weiteren werden die Transaktionen bei Konflikten erst nach der kompletten Ausführung abgebrochen, so dass gegebenenfalls sehr viel unnötige Arbeit durchgeführt wurde. Diese Gefahr besteht vor allem für lange Transaktionen und für Hot Spots [HR01].

Um den Unterschied zwischen synchronen und asynchronen Verfahren zu verdeutlichen, dient folgendes Ausgangsszenario: Eine Transaktion hat ihren Ursprung auf einem Knoten K. Die in der Transaktion enthaltenen Operationen müssen aber ebenfalls auf weiteren Knoten K_i ausgeführt werden.

Bei einem synchronen Verfahren werden die Operationen einer Transaktion auf allen notwendigen Knoten vor dem Commit der Transaktion ausgeführt. Somit sind im synchronen Fall die Datenstände auf allen Knoten (K_i und K) bei einem Commit aktuell. Synchrone Verfahren sind geeignet, um eine hohe Verfügbarkeit der Daten zu erzielen [LKPnMJP05], da durch diese Vorgehensweise alle Daten stets aktuell gehalten werden. Bei dieser Synchronisierung sind die Ausführungsfolgen der Transaktionen serialisierbar, wodurch keine Mehrbenutzeranomalien auftreten können[WPS+00]. Die Nachteile des synchronen

Ansatzes liegen in der Verzögerung der Transaktionsausführung und der verminderten Skalierbarkeit des Systems [LKPnMJP05]. Zusätzlich verschlechtert sich die Performanz der Änderungsoperationen [WPS+00].

Bei asynchronen Verfahren müssen alle Operationen einer Transaktion bei Commit auf dem Ursprungsknoten K ausgeführt sein. Auf allen anderen Knoten (K_i) können, aber müssen sie noch nicht durchgeführt sein. Aus diesem Grund resultieren unterschiedliche Datenstände auf unterschiedlichen Knoten. Asynchrone Verfahren sind für schnelle lokale Datenzugriffe geeignet. Zusätzlich ist bei diesem Ansatz die Skalierbarkeit des Systems kein Problem [LKPnMJP05]. Es sind mögliche Konsistenzprobleme zu beachten, sofern Seitenfehler auftreten, bevor Änderungsoperationen weitergeleitet werden konnten [LKPnMJP05]. Darüber hinaus gibt es keine automatischen Rücksetzmöglichkeiten von Daten, die auf einigen Replikaten bereits erfolgreich durch Commit dauerhaft gemacht wurden [WPS+00].

4.3 Architektur des verwendeten Gesamtsystems

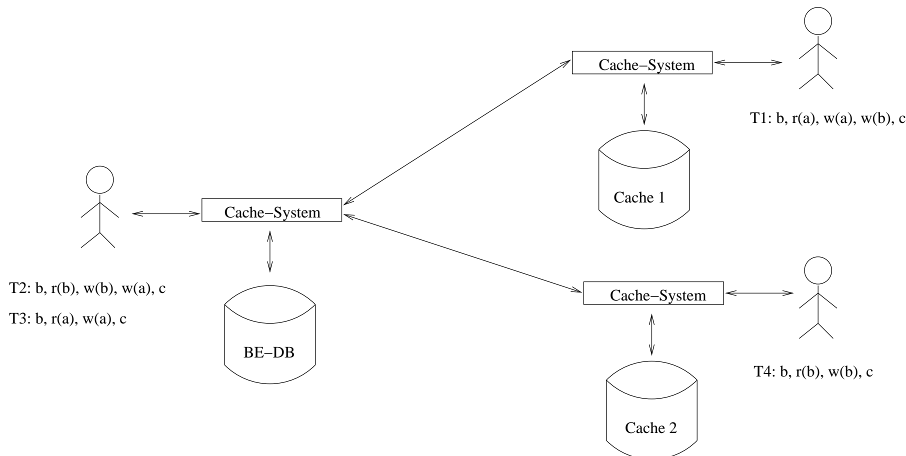


Abbildung 4.3: Architektur des Gesamtsystems

In Abbildung 4.3 wird die Architektur des Gesamtsystems veranschaulicht, wie sie im Constraint-basierten Datenbank-Caching verwendet wird. *Cache1* und *Cache2* stehen hier repräsentativ für alle vorhandenen Caches. Mit Hilfe dieser Architektur des Gesamtsystems wird im Folgenden auf Synchronisierungsverfahren hingewiesen, die für den Einsatz im Constraint-basierten Datenbank-Caching geeignet sind. Basierend auf dieser Analyse können dann in Kapitel 4.4 konkrete Ziele für ein Synchronisierungsverfahren des Constraint-basierten Datenbank-Caching vorgestellt werden.

Durch das Caching entstehen Replikate, welche im Zuge der Synchronisierung aktualisiert werden müssen. Aufgrund der Definition des Constraint-basierten Datenbank-Caching entstehen hierbei jedoch keine Replikate aller Datenelemente, sondern nur Teilmengen der Daten, die häufig im Cache angefragt werden. Lediglich das Backend enthält alle Datenelemente. Aufgrund

dieser Speicherung von Datenkopien können alle Verfahren für die Synchronisierung verworfen werden, die eine verteilte Datenspeicherung annehmen. Die verteilte Datenspeicherung beinhaltet, dass die Datenelemente verteilt auf unterschiedlichen Knoten und nicht repliziert gespeichert werden.

Aufgrund der zentralen Instanz, dem Backend, bietet sich die Umsetzung einer Primary-Copy-Architektur an. Die Vor- und Nachteile dieser Architektur müssen allerdings bezüglich des Einsatzes beim Constraint-basierten Datenbank-Caching neu überdacht werden. So ist beispielsweise die Skalierbarkeit des Gesamtsystems ein Problem des Primary-Copy-Ansatzes, sofern auf allen Knoten alle Datenelemente repliziert werden. Da im Constraint-basierten Datenbank-Caching jedoch nur Teilmengen der Datenelemente auf den Caches repliziert werden, müssen aufgrund einer Änderungsoperation nur ein paar und nicht alle Replikate aktualisiert werden. Dies gilt allerdings nur unter der Annahme, dass das Backend entweder über den eingeschränkten oder vollen Wissensstand verfügt. Es besteht die Möglichkeit, dass zwar sehr viele Caches im System existieren, jedoch nur wenige aktualisiert werden müssen. Aus diesem Grund stellt die Skalierbarkeit des Systems keinen Nachteil im Constraint-basierten Datenbank-Caching dar.

Bei der Realisierung eines Primary-Copy-Ansatzes ist zu beachten, dass Änderungsoperationen nur auf der Primärkopie erlaubt sind und die anderen Replikate danach aktualisiert werden. Damit die Vorteile des Caching genutzt werden können, muss diese Einschränkung aufgelockert werden. Aus diesem Grund werden in Kapitel 4.3.1 konkrete Verfahren vorgestellt, die eine Behandlung von Änderungsoperation auf festgelegten Knoten betrachten.

Zusätzlich zur Abbildung einer Primary-Copy-Architektur wird auf das verwendete Datenbanksystem eine weitere Schicht, das Cache-System, aufgesetzt. Hierdurch kann ein Synchronisierungsverfahren entwickelt werden, das unabhängig vom konkret verwendeten Datenbanksystem ist. Unabhängig bedeutet in diesem Fall, dass keine Änderungen bezüglich der Synchronisierung im Datenbanksystem selbst notwendig sind. Allerdings hängt die Realisierung eines Synchronisierungsverfahrens vom Isolationsgrad des verwendeten Datenbanksystems ab. Die möglichen Isolationsgrade werden in Kapitel 4.3.2 vorgestellt. Hierfür kann bereits vorweg genommen werden, dass für die Synchronisierung im Constraint-basierten Datenbank-Caching lediglich hohe Isolationsgrade betrachtet werden, da dadurch eine hohe Datenkonsistenz garantiert werden kann.

Bezüglich der Realisierung der Architektur des Gesamtsystems ist die Kommunikation zwischen den Knoten besonders von Interesse. Wie in Abbildung 4.3 dargestellt, wird eine Kommunikation zwischen dem Backend und den einzelnen Caches angenommen. Dies entspricht einer Primary-Copy-Architektur. Darüber hinaus interagieren die Benutzer lediglich über das Cache-System mit der Datenbank. Ein Benutzer führt hierbei eine Transaktion immer nur über einen Cache aus. Dies bedeutet, dass einzelne Operationen derselben Transaktion nicht über mehrere Caches verteilt, sondern nur über einen Cache ausgeführt werden. Es ist ebenfalls denkbar, dass die Caches im Cluster angeordnet sind und untereinander kommunizieren können. Der Fokus dieser Arbeit liegt jedoch zunächst auf einer Primary-Copy-Architektur, da hierfür die Voraussetzungen bereits erfüllt sind. Zusätzlich muss bei dieser Architektur bezüglich der Kommunikation beachtet werden, dass zwischen dem Backend und den Caches eine Datenleitung mit einer hohen Latenz existiert. Mögliche Synchronisierungsverfahren für das Constraint-basierte Datenbank-Caching müssen dies zwingend beachten. Somit

müssen Verfahren, die z. B. eine Gruppenkommunikation oder eine synchrone Kommunikation annehmen, an diese Situation angepasst werden.

4.3.1 Änderungsoperationen auf Replikaten

Für die verschiedenen Möglichkeiten auf welchen Replikaten welche Operationen einer Transaktion ausgeführt werden dürfen, existieren beispielsweise folgende Verfahren:

- Der **Update-everywhere-Ansatz** erlaubt das Lesen sowie das Schreiben von Datenobjekten auf allen Replikaten.

  Die Vorteile des Update-everywhere-Ansatzes ergeben sich dadurch, dass der Zugriff auf die Daten sehr schnell erfolgen kann und dass Änderungsoperationen nicht an spezielle oder mehrere Knoten weitergeleitet werden müssen. Allerdings wird hierdurch die Koordination der Transaktionen und somit die Konfliktkontrolle sehr komplex [WPS+00].

- Beim **Primary-Copy-Verfahren** wird ein Replikat als Primärkopie der Daten definiert. Leseoperationen dürfen auf allen Replikaten ausgeführt werden. Änderungsoperationen sind allerdings nur auf der Primärkopie erlaubt und müssen gegebenenfalls dorthin weitergeleitet werden.

  Die Synchronisierung der Transaktionen muss lediglich auf der Primärkopie erfolgen, so dass die Kontrolle der Replikate deutlich einfacher als z. B. beim Update-everywhere-Ansatz ist [WPS+00]. Allerdings stellt die Primärkopie auch einen Flaschenhals dar, da nur hier die Änderungsoperationen ausgeführt werden dürfen [WPS+00]. Dadurch resultieren gegebenenfalls Verzögerungen in der Operationsausführung [LKPnMJP05].

- Das **ROWA(Read One Write All)-Verfahren** bietet diverse Ausprägungen an, welche die Lese- und Änderungsoperationen (R bzw. W) betreffen. Folgende Ausprägungen stehen zur Verfügung: *One (O), Many (M)* und *All (A)*. Wird beispielsweise die Ausprägung *All* für die Änderungsoperationen gewählt, so muss die Datenänderung auf allen Replikaten durchgeführt werden, bevor die Transaktion beendet werden kann. Wird hingegen *read one* ausgewählt, so muss die Leseoperation lediglich auf dem angefragten Replikat ausgewertet werden.

  Das ROWA-Verfahren ist ein naheliegender Ansatz, bei dem eine festgelegte Anzahl an Replikaten stets aktuell gehalten wird. Dieses Verfahren stellt eine erhöhte Verfügbarkeit der Daten für Lesezugriffe bereit, so lange mindestens ein Replikat verfügbar ist [Rah07]. Die Änderungsoperationen müssen auf diesen Replikaten synchron erfolgen, so dass vor jeder Änderungsoperation auf allen notwendigen Datenelementen Schreibsperren angefordert werden müssen. Dadurch resultiert ein enormer Zusatzaufwand an Kommunikation [Rah07] und eine Verzögerung der Operationsausführung [LKPnMJP05].

Aus diesen möglichen Verfahren resultieren unterschiedliche Optionen, auf welcher Knotenmenge eine Operation oder Transaktion ausgeführt werden muss. Unter der Annahme, dass alle Knoten synchron das Commit einer Transaktion bearbeiten, muss jeder Knoten in die Bearbeitung des Commit miteinbezogen werden, der mindestens eine Operation der abzuschließenden Transaktion

durchgeführt hat. Folgende Ausprägungen können bezüglich der Operationsausführung unterschieden werden:

- Wird der Update-Everywhere-Ansatz angenommen, so wird eine Transaktion zunächst auf einem Knoten durchgeführt. Erst nach Abschluß einer Transaktion werden die anderen Knoten aktualisiert. Dieser Fall kann als *single node synchronization* bezeichnet werden.
- Bei der Verwendung des Primary-Copy-Ansatzes können zwar Leseoperationen auf allen Knoten ausgeführt werden, Änderungsoperationen jedoch nur auf der Primärkopie. In diesem Szenario bilden der Ursprungsknoten und die Primärkopie ein Team bezüglich der Durchführung und des Abschluß einer Transaktion, da beide Knoten Teiloperationen der Transaktion bearbeiten. Dieser Fall beschreibt eine *team synchronization.*
- Wird eine Ausprägung des ROWA-Verfahrens verwendet, in der alle erreichbaren Knoten aufgrund einer Änderungsoperation aktualisiert werden müssen, so muss die Datenänderung innerhalb der Transaktion auf mehreren, jedoch nicht auf allen Knoten bearbeitet und erfolgreich abgeschlossen werden. Dieses Szenario kann als *cluster synchronization* bezeichnet werden.
- Wird das ROWA-Verfahren angenommen, so müssen Änderungsoperation auf allen Knoten verarbeitet werden. Teiloperationen werden auf allen Knoten ausgeführt und die Transaktion muss auf allen Knoten erfolgreich beendet werden. Dieser Fall kann *all synchronization* genannt werden.

4.3.2 Isolationsgrade von Datenbanksystemen

Ein Synchronisierungsverfahren ist abhängig von dem Isolationsgrad, den das verwendete Datenbanksystem garantiert. Aus diesem Grund werden in diesem Abschnitt die verschiedenen Isolationsgrade vorgestellt. Laut [BBG+95] können folgende Grade unterschieden werden:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Snapshot Isolation
- Serializable

Die Unterscheidung dieser Grade erfolgt aufgrund der Phänomene *Dirty Write, Dirty Read, Lost Update, Non-repeatable Read, Phantom, Read Skew* und *Write Skew.* Eine Erklärung dieser Phänomene (auch Anomalien genannt) befindet sich in Anhang A. In Tabelle 4.4 [BBG+95] werden die Isolationsgrade mit den dort auftretenden Phänomenen veranschaulicht. Weiterhin werden in Abbildung 4.5 die Abhängigkeiten der unterschiedlichen Isolationsgrade graphisch dargestellt. Die Kanten sind mit den Anomalien beschriftet, die im höheren Isolationsgrad nicht auftreten können. Einige Datenbanken garantieren Serialisierbarkeit (z. B. Konfliktserialisierbarkeit), während andere lediglich Snapshot

Isolationsgrad	Dirty Write	Dirty Read	Lost Update	Non-repeatable Read	Phantom	Read Skew	Write Skew
Read Uncommitted	nicht möglich	möglich	möglich	möglich	möglich	möglich	möglich
Read Committed	nicht möglich	nicht möglich	möglich	möglich	möglich	möglich	möglich
Repeatable Read	nicht möglich	nicht möglich	nicht möglich	nicht möglich	möglich	nicht möglich	nicht möglich
Snapshot Isolation	nicht möglich	nicht möglich	nicht möglich	nicht möglich	manchmal möglich	nicht möglich	möglich
Serializable	nicht möglich	nicht möglich	nicht möglich	nicht möglich	nicht möglich	nicht möglich	nicht möglich

Abbildung 4.4: ANSI SQL Isolationsgrade und ihre Phänomene [BBG+95]

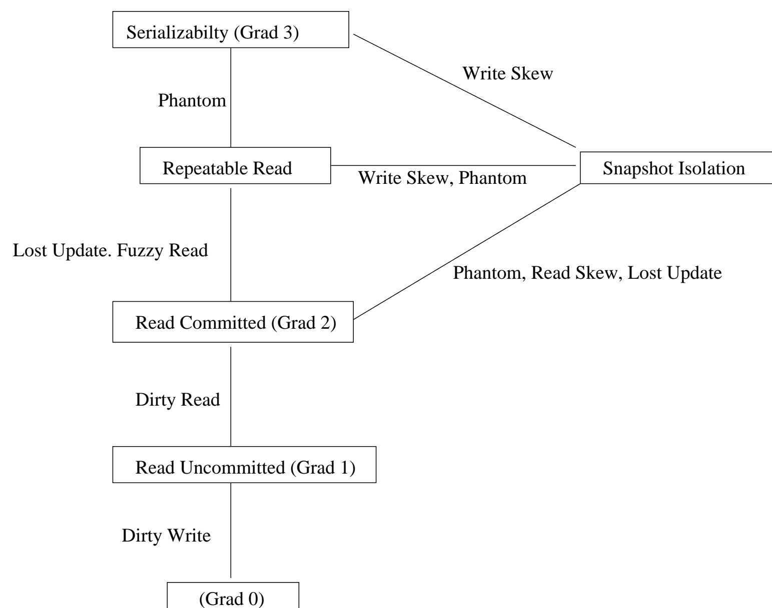


Abbildung 4.5: Abhängigkeiten der Isolationsgrade und der Anomalien

Isolation anbieten [FLO+05]. Bei der Snapshot Isolation greifen Transaktionen auf Datenobjekte zu, die auf einem Datenbank-Schnappschuss basieren.

Aufgrund der hohen Garantien sind für die Synchronisierung im Constraint-basierten Datenbank-Caching besonders die Isolationsgrade *Serialisierbarkeit* und *Snapshot Isolation* von Interesse.

4.4 Ziele für das CbDBC

Aufgrund des in Abbildung 4.3 dargestellten Grundaufbaus des Gesamtsystems lassen sich folgende Ziele bezüglich der Synchronisierung im Constraint-basierten Datenbank-Caching ableiten:

- Da in den Caches Teilmengen der Daten aus dem Backend repliziert werden, sind nur Lösungsansätze von Bedeutung, die eine Synchronisation von Replikaten unterstützen. Aufgrund dieser Einschränkung werden folglich alle Verfahren ausgeschlossen, die eine verteilte Speicherung der Datenelemente annehmen, wie z. B. in [MNKK06].

- Da die meisten Synchronisierungsverfahren syntaktische Verfahren realisieren [BD96], werden diese für den Einsatz im Constraint-basierten Datenbank-Caching favorisiert. Semantische Verfahren werden aufgrund ihres begrenzten Einsatzbereichs nicht betrachtet, obwohl sie möglicherweise Verbesserungsmöglichkeiten anbieten.

- Ein weiteres Ziel der Synchronisierungsverfahren für das Constraint-basierte Datenbank-Caching ist, dass das Verfahren möglichst unabhängig vom verwendeten Datenbanksystem sein soll. Aus diesem Grund liegt der Fokus auf Middleware-basierten Verfahren. Dadurch sind keine Änderungen im Datenbanksystem notwendig und es können viele Standard-Datenbanksysteme unterstützt werden. Es besteht zudem die Möglichkeit die Synchronisierung des Constraint-basierten Datenbank-Caching in heterogenen Systemen zu realisieren. Zusätzlich wird bereits das Füllen der Cache Groups mit Hilfe einer Middleware realisiert, so dass diese sowieso existiert und entsprechend erweitert werden kann.

- Primary-Copy-Verfahren sind sehr geeignet, da hier ein Replikat als Primärkopie der Daten definiert wird. Die Primärkopie wird im Fall des Constraint-basierten Datenbank-Caching im Backend gehalten und in den Caches werden lediglich Teilmengen dieser Daten gespeichert. Leseoperationen dürfen bei diesem Verfahren auf allen Replikaten ausgeführt werden, Änderungsoperationen jedoch nur auf der Primärkopie. Änderungsoperationen müssen gegebenenfalls erst dorthin weitergeleitet werden. Nach der Ausführung einer Änderungsoperation auf der Primärkopie werden die Daten auf den anderen Replikaten aktualisiert.

  Es ist allerdings sehr wünschenswert, Änderungsoperationen bereits im Cache ausführen zu können, sofern dort die benötigten Daten vorhanden sind. Sind die von einer Änderungsoperation betroffenen Daten auf dem Cache vorhanden, so kann sie nach dem Update-everywhere-Verfahren dort ausgeführt werden. Allerdings muss der Abschluss einer Transaktion synchron erfolgen, so dass ein konsistenter Datenzustand garantiert werden kann.

Aus der Kombination des Primary-Copy-Verfahrens mit dem Update-everywhere-Ansatz resultiert eine Variante des ROWA-Verfahrens.

- Damit alle Replikate immer den aktuellsten Stand der Datenobjekte erhalten, ist es notwendig, die Transaktionen vor dem Commit zu synchronisieren. Dadurch können serialisierbare Transaktionsabfolgen erzeugt und Mehrbenutzeranomalien verhindert werden.

  Darüber hinaus sind asynchrone Verfahren aufgrund ihrer Eigenschaften sehr interessant. So besteht die Möglichkeit, gegebenenfalls einige festgelegte Aktionen des Synchronisierungsprozess verzögert auszuführen. Dies hat jedoch oft zur Folge, dass der Konsistenzgrad der Daten beeinträchtigt wird.

- Zusätzlich sind die verwendeten Datenbanksysteme von Bedeutung, auf denen das Cache-System aufbaut. Datenbanksysteme bieten unterschiedliche Isolationsgrade an, die den Grad der Serialisierbarkeit von Transaktionen beschreiben. Aufgrund ihrer Eigenschaften soll mindestens Snapshot Isolation oder sogar Serialisierbarkeit garantiert werden.

- Ein weiteres Ziel der Synchronisierungsverfahren ist, dass Anomalien und Deadlocks verhindert werden.

Aufgrund dieser Analyse liegt das Hauptaugenmerk bezüglich möglicher Synchronisierungsverfahren kurz zusammengefasst auf folgenden Eigenschaften:

- Synchronisierung von Replikaten
- Middleware-basierte Ansätze
- Primary-Copy-Verfahren, Update-everywhere-Verfahren
- synchrone und syntaktische Verfahren, die Snapshot Isolation oder sogar 1-Kopien-Serialisierbarkeit garantieren

Im Folgenden wird ein Synchronisierungsverfahren vorgestellt, das nahezu alle Zieleigenschaften unterstützt und als Isolationsgrad Snapshot Isolation (SI) garantiert. Aus diesem Grund wird dieses Verfahren im Detail beschrieben und bezüglich seiner Einsatzmöglichkeit im Constraint-basierten Datenbank-Caching genau analysiert.

4.5 Synchronisierung, die SI garantiert

In [LKPnMJP05] wird ein Verfahren vorgestellt, das die Synchronisierung mit Hilfe einer Middleware durchführt. Es wird ein hybrider, Update-everywhere-Ansatz verfolgt, wobei die Middleware auf das Standard-Datenbank-Interface aufsetzt.

Das Verfahren ist für Anwendungen geeignet, die Fehlertoleranz und stetige Datenkonsistenz fordern. Dabei werden trotz hoher Änderungsraten schnelle Antwortzeiten und eine hoher Durchsatz erreicht. Grundlegend für diesen Middleware-basierten Ansatz ist die Garantie der *Snapshot Isolation*. Zur Bereitstellung von Snapshot Isolation muss ein Datenbanksystem mehrere Versionen der Datenobjekte bereitstellen.

Um die Einsatzmöglichkeit im Constraint-basierten Datenbank-Caching zu überprüfen, wird im folgenden dieser Synchronisierungsansatz im Detail analysiert. Hierzu wird zunächst der Serialisierbarkeitsgrad der Snapshot Isolation definiert. Danach wird ein zentralisierter Basisalgorithmus vorgestellt sowie dessen Probleme diskutiert. Ergänzt wird dieser Ansatz durch eine verteilte Version des Algorithmus. Abschließend wird die konkrete Einsatzmöglichkeit im Constraint-basierten Datenbank-Caching überprüft.

4.5.1 Snapshot Isolation (SI)

Unter der Garantie der Snapshot Isolation liest eine Transaktion T stets einen Schnappschuss der Datenbank, der alle Datenänderungen enthält, die bis zu ihrem *begin of transaction* (BOT) gültig waren. Folgende Definitionen sind im Bereich der Snapshot Isolation von Bedeutung:

Definition 1 (SI-Schedule)**.** *Jede Transaktion T_i wird durch ihr Read Set RS_i und ihr Write Set WS_i definiert. Sei nun T die Menge aller Transaktionen, die bereits durch Commit beendet wurden. Ein SI-Schedule S über T ist eine Operationssequenz $o \in \{b, c\}$. $(o_i <_H o_j) \in S$ beschreibt, dass o_i vor o_j in S auftritt. S hat folgende Eigenschaften [LKPnMJP05]:*

- *Für jedes $T_i \in T$ gilt: $(b_i <_H c_i) \in S$*
- *Wenn $(b_i <_H c_j <_H c_i) \in S$, dann gilt $WS_i \cap WS_j = \{\}$*

Mit Hilfe dieser Definition wird verdeutlicht, dass die Betrachtung und Behandlung von Write/Write-Konflikten notwendig ist, um Snapshot Isolation garantieren zu können. Existiert ein Write/Write-Konflikt zwischen zwei Transaktionen, so muss eine dieser beiden Transaktionen abgebrochen werden. Aufgrund lokaler Synchronisierungsverfahren in der Datenbank gewinnt die Transaktion, die zuerst die entsprechenden Datenelemente verändert hat.

Definition 2 (Strong SI)**.** *Eine Transaktionshistorie H ist „strong SI" genau dann, wenn die Historie ein abgeschlossener SI-Schedule ist und wenn jede Transaktion $T \in H$ garantiert den aktuellsten Datenbankzustand sieht. Dies bedeutet, wenn eine Transaktion T_1 vor dem Start einer zweiten Transaktion T_2 ihr Commit ausführt, kann Transaktion T_2 auf den Datenbankzustand zurückgreifen, der die Datenänderungen von T_1 beinhaltet. [DS06]*

Definition 3 (Strong Session SI)**.** *Eine Transaktionshistorie H erfüllt die Eigenschaft „strong session SI" genau dann, wenn*

- *H ist schwach SI („Schwach SI" bedeutet hier, dass eine Transaktion auf alle Schnappschüsse zugreifen kann, die vor deren Start gültig waren; „strong SI" hingegen, dass eine Transaktion nur den zuletzt gültigen Schnappschuss vor deren Start lesen kann. [DS06])*
- *Für jedes Transaktionspaar T_i und T_j aus H, das bereits durch Commit beendet wurde und für das $H(T_i) = H(T_j)$ gilt und c_i vor der ersten Operation von T_j ausgeführt wird, gilt $start(T_j) > commit(T_i)$.*

Es können folglich diverse Ausprägungen der Snapshot Isolation definiert werden. *Strong Session SI* beispielsweise garantiert den Isolationsgrad *Strong SI* bezogen auf die betrachteten Sessions. Im Folgenden wird der *Simple Replica Control Algorithm* beschrieben, der *strong SI* garantiert.

4.5.2 Basisalgorithmus SRCA

In [LKPnMJP05] wird der *Simple Replica Control Algorithm (SRCA)*[2] vorgestellt, der einen einfachen, zentralisierten Ansatz zur Synchronisierung verteilter Transaktionen umsetzt. Es handelt sich um einen Middleware-basierten Ansatz, der Snapshot Isolation garantiert.

Die Aufgaben der Middleware lassen sich folgendermaßen zusammenfassen: Die Middleware leitet alle Operationen einer Transaktion an ein Datenbank-Replikat weiter, auf dem die Operationen ausgeführt werden. Da Transaktionsoperationen zunächst nur auf einem festgelegten Replikat ausgeführt werden, wird dieses als „lokales Replikat“ bezeichnet. Alle anderen Replikate, auf denen die Transaktion noch nachgezogen werden muss, gelten als „Remote-Replikate“. Damit Transaktionsoperationen auf den Remote-Replikaten nachgezogen werden können, werden Write Sets der Operationen auf dem lokalen Replikat angefordert. In diesen Write Sets werden die geänderten Objekte, sowie deren Identifikatoren gespeichert. Die Write Sets werden zwischen den Replikaten mit Hilfe eines total geordneten Multicast ausgetauscht. Aufgrund dieser Write Sets können in der Middleware Write/Write-Konflikte zwischen Transaktionen erkannt und aufgelöst werden. Wie die Konflikterkennung und Konfliktbehandlung im Detail funktioniert, wird im folgenden Abschnitt beschrieben. Existieren keine Konflikte, so kann die Transaktion auf dem lokalen Replikat ihr Commit ausführen und auf allen Remote-Replikaten nachgezogen werden.

Vorgehensweise des SRCA

Zur Beschreibung der Vorgehensweise werden zunächst folgende Variablen definiert:

- *ws_list*: Liste aller bisher validierten Transaktionen
- *T.tid*: Identifikator einer Transaktion
- *tocommit_queue_k*: Liste, in der für jedes Replikat R_k die Write Sets gespeichert werden, die noch auf R_k nachgezogen und durch Commit beendet werden müssen
- *last_committed_tid_k*: *T.tid* der Transaktion, die zuletzt auf R_k ihr Commit durchgeführt hat

Bei Beginn einer Transaktion T_i wird zunächst festgelegt, auf welchem Replikat R_m diese Transaktion ausgeführt werden soll. Um sicherzustellen, dass bei Transaktionsbeginn keine andere Transaktion gerade eine Commit-Operation ausführt, wird für die *Beginn*-Operation der betrachteten Transaktion ein Mutex angefordert. Wird dieser gewährt, kann die Transaktion T_i gestartet werden. Gleichzeitig kann der genaue Zeitpunkt des Transaktionsstarts festgestellt werden. Lese- und Schreiboperationen werden dann direkt an das lokale Replikat R_m durch die Middleware weitergeleitet. Die Leseoperationen greifen hierbei auf den Datenbankzustand des Schnappschuss bei BOT von T_i zu, währenddessen Schreiboperationen neue Versionen der Datenobjekte erzeugen. Erreicht die *Commit*-Operation die Middleware, muss geprüft werden, ob Konflikte mit anderen Transaktionen existieren. Zunächst fragt die Middleware das Write Set der

[2] Der Pseudocode des SRCA kann in [LKPnMJP05] nachgelesen werden.

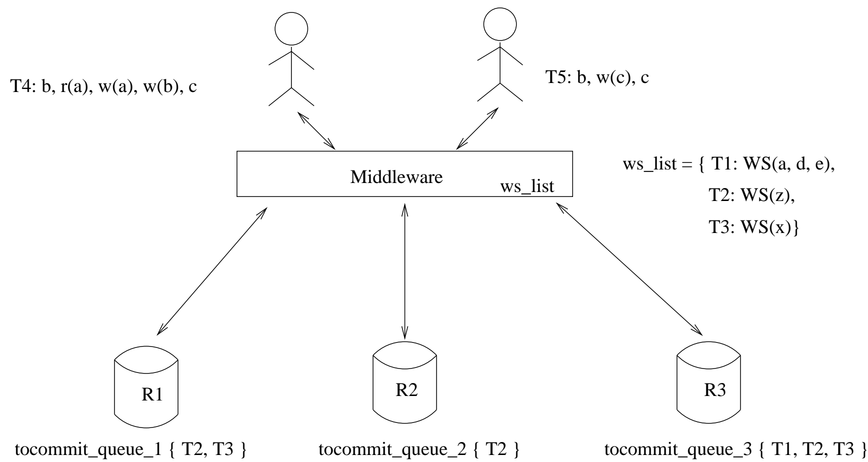


Abbildung 4.6: Beispielszenario zur Veranschaulichung des SRCA

Transaktion T_i beim lokalen Replikat R_m an. Ist dieses Write Set leer, so enthält die Transaktion nur Leseoperationen und kann lokal ihr Commit ausführen. Ist das Write Set nicht leer, so startet die Validierung. Hierbei ist zu beachten, dass immer nur eine Transaktion auf einmal validiert werden kann. Im Laufe der Validierung wird das Write Set mit allen Write Sets der bereits validierten Transaktionen (alle Transaktionen aus der Liste *ws_ list*) verglichen. Existiert mindestens ein Konflikt zwischen zwei Transaktionen, so wird die Transaktion T_i abgebrochen. Andernfalls erhält die Transaktion ihren Transaktionsidentifikator ($T.tid$). Danach wird das Write Set bei allen Remote-Replikaten der Menge *tocommit_ queue_ k* hinzugefügt. Auf dem lokalen Replikat R_m kann die Transaktion T_i erst dann ihr Commit durchführen, sobald alle Write Sets auf diesem Replikat nachgezogen wurden, die sich bei Commit der Transaktion T_i in der *tocommit_ queue_ m* befanden. Nachdem die Transaktion T_i erfolgreich auf einem Replikat ausgeführt wurde, wird das Write Set aus der *tocommit_ queue* des entsprechenden Replikats entfernt.

In Abbildung 4.6 wird ein Beispielszenario vorgestellt, mit dessen Hilfe die Vorgehensweise des SRCA verdeutlicht werden kann. Zunächst wird die Durchführung der Transaktion $T4$ betrachtet. Es wird festgelegt, dass $R1$ das lokale Replikat für diese Transaktion ist. Nachdem die einzelnen Operationen ausgeführt wurden, soll die Transaktion durch das Commit beendet werden. Hierzu überprüft die Middleware, ob Write/Write-Konflikte zwischen der Transaktion $T4$ und den bereits validierten Transaktionen $T1, T2, T3$ (Elemente der *ws_ list*) existieren. Dies geschieht, indem für jedes Element der *ws_ list* das Write Set mit dem Write Set der Transaktion $T4$ verglichen wird. Ist die Schnittmenge der betrachteten Write Sets leer, so existieren keine Write/Write-Konflikte zwischen diesen Transaktionen und die Transaktion $T4$ kann durch Commit abgeschlossen werden. Andernfalls muss die Transaktion $T4$ aufgrund dieser Konflikte abgebrochen werden. In diesem Beispielszenario existiert ein Write/Write-Konflikt mit der Transaktion $T1$, so dass $T4$ abgebrochen wird.

Wird hingegen Transaktion $T5$ betrachtet, so kann während der Validierung festgestellt werden, dass keine Write/Write-Konflikte existieren. Wurde diese

Transaktion z. B. auf dem lokalen Replikat $R2$ aufgeführt, so kann das Commit für $T5$ dort erst dann ausgeführt werden, wenn Transaktion $T3$ erfolgreich nachgezogen wurde. Dies resultiert daraus, dass $T5$ nach $T3$ der *tocommit_ queue_ 2* hinzugefügt wurde. Aufgrund der erfolgreichen Validierung von Transaktion $T5$ wird dessen Write Set in *tocommit_ queue_ 1* und *tocommit_ queue_ 3* hinzugefügt. Sobald die Transaktion auf den Replikaten $R1, R2$ oder $R3$ ausgeführt wurde, wird die Transaktion und deren Write Set aus der entsprechenden *tocommit_ queue* entfernt.

Bevor die Einsatzmöglichkeiten des SRCA im Constraint-basierten Datenbank-Caching betrachtet werden, werden zunächst die Vor- und Nachteile des Basisalgorithmus diskutiert. Darüber hinaus wird eine Modifizierung des SRCA vorgestellt, der basierend auf der Analyse der Nachteile in [LKPnMJP05] entwickelt wurde.

Analyse des SRCA

Dass nur Write/Write-Konflikte beachtet werden, hat den Vorteil, dass Leseoperationen zu keiner Zeit mit Änderungsoperationen im Konflikt stehen, da die Leseoperationen von einem Schnappschuss lesen [LKPnMJP05]. Ein weiterer Vorteil besteht darin, dass eine bessere Systemperformanz für Lesetransaktionen erreicht werden kann [DS06]. Zusätzlich werden aufgrund der Definition der Snapshot Isolation folgende Anomalien des Mehrbenutzerbetriebs verhindert: *Dirty Write, Dirty Read, Non-repeatable Read, Lost Update* [DS06]. Allerdings können *Write Skew* und das *Phantom-Problem* hier nicht immer verhindert werden. Zusätzlich besteht die Gefahr von Transaktions-Inversionen. Dies bedeutet, dass die Benutzer nicht zwingend den Datenbankzustand sehen können, der aufgrund ihrer Transaktionsoperationen erstellt wurde, da dieser bisher noch nicht auf dem Replikat vollständig umgesetzt werden konnte.

Das Hauptproblem des SRCA-Algorithmus ist, dass die lokalen Synchronisierungsverfahren der konkret eingesetzten Datenbanksysteme nicht betrachtet werden. Allerdings müssen ebenfalls die Auswirkungen der lokalen Synchronisierung analysiert werden. Zunächst resultiert aus dieser Tatsache eine Optimierungsmöglichkeit des Algorithmus. Da die Datenbanksysteme selbst ebenfalls eine Validierung durchführen, muss die Middleware keine Transaktion gegenüber anderen lokalen Transaktionen überprüfen. Allerdings hat die Verwendung von Sperrprotokollen diverse Probleme zur Folge. Zunächst ist es möglich, dass lokale Transaktionen Remote-Transaktionen blockieren. Dadurch besteht die Gefahr, dass Remote-Transaktionen abgebrochen werden. Da diese aber auf allen Replikaten ausgeführt und zwingend durch Commit abgeschlossen werden müssen, führt dies zu einem Widerspruch. Zusätzlich besteht die Gefahr von Deadlocks, die durch lokale und Remote-Transaktionen ausgelöst werden können. Das Auftreten eines Deadlock wird anhand des folgenden Szenario verdeutlicht. Es werden zwei nebenläufige lokale Transaktionen angenommen. T_i hält eine Sperre auf dem Datenobjekt x und T_j eine Datensperre auf dem Objekt y. Weiterhin wird eine bereits validierte Transaktion T_r auf diesem Replikat ausgeführt. Wenn nun T_r ebenfalls das Datenobjekt y ändern möchte, fordert es die entsprechende Sperre an. Allerdings wird T_r blockiert, da bereits T_j diese Sperre hält. Nun wird T_i beendet und kann erfolgreich validieren, da z. B. mit T_r kein Konflikt entsteht. Allerdings kann T_i nicht ihr Commit durchführen, solange T_r nicht

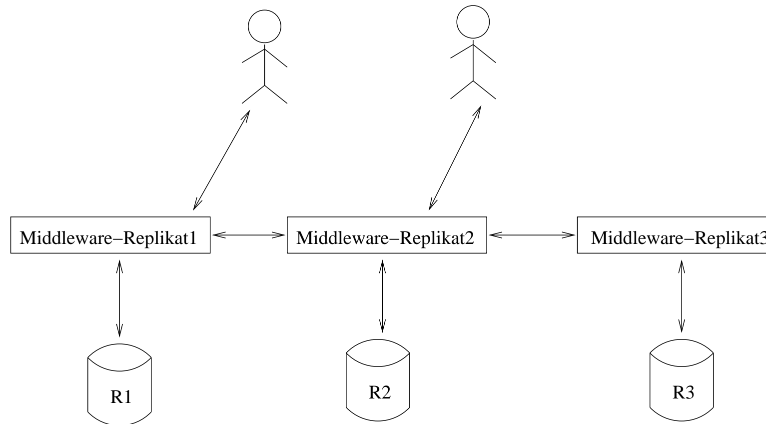


Abbildung 4.7: Architektur für die Realisierung des SRCA-Rep

fertig abgearbeitet wurde. T_j möchte nun auch das Datenobjekt x verändern und versucht die entsprechende Sperre anzufordern. Dies scheitert, da noch T_i diese Sperre hält. Aufgrund dieser Konstellation existiert im Datenbanksystem kein Deadlock, da lediglich folgende Wartebeziehungen vorhanden sind:

$$T_j \rightarrow T_i \text{ und } T_r \rightarrow T_j$$

Allerdings ist aus Sicht der Middleware ebenfalls die Wartebeziehung $T_i \rightarrow T_r$ bekannt, so dass ein sogenannter versteckter globaler Deadlock existiert.

Modifizierung des SRCA (SRCA-Rep)

Aufgrund der zuvor beschriebenen Probleme wurde der SRCA-Algorithmus in [LKPnMJP05] überarbeitet. Zunächst wurden in den Algorithmus lokale Synchronisierungsverfahren durch die verwendeten Datenbanksysteme miteinbezogen. Daraus ergibt sich beispielsweise, dass eine Transaktion auf dem lokalen Replikat nicht gegenüber anderen lokalen Transaktionen validiert werden muss, da dies bereits durch lokale Synchronisierungsverfahren im Datenbanksystem geschieht. Weiterhin wird im SRCA eine Transaktion T_i erst dann auf dem lokalen Replikat R_k durch Commit abgeschlossen, wenn alle notwendigen Write Sets aus der *tocommit_queue_k* erfolgreich nachgezogen wurden. Diese Verzögerung der Commit-Ausführung kann dann verhindert werden, wenn eine Transaktion T_i auf dem lokalen Replikat R_k direkt nach der erfolgreichen Validierung das Commit abschließen darf, obwohl noch weitere Write Sets davor nachgezogen werden müssen. Dies ist allerdings nur dann möglich, wenn keine Konflikt-Transaktionen vor dieser Transaktion T_i in der *tocommit_queue_k* angeordnet sind.

Aufgrund dieser Überlegungen wurde der SRCA-Rep-Algorithmus entwickelt, der neben der Datenreplikation auf der Verwendung einer replizierten Middleware basiert (siehe Abbildung 4.7).

Die Vorgehensweise des SRCA-Rep basiert auf der Grundlage des SRCA. Transaktionen werden zunächst auf einem lokalen Replikat R_k ausgeführt. Bei Transaktionsende erhält das Middleware-Replikat M_k das Write Set der Transaktion und leitet dies mit Hilfe eines total geordneten Multicast an alle anderen

Middleware-Replikate weiter. Dann beginnt auf allen Middleware-Replikaten die Validierung. Hierbei ist es notwendig, dass alle Middleware-Replikate die gleiche Entscheidung bezüglich der Validierung treffen müssen. Aufgrund des total geordneten Multicast ist sichergestellt, dass alle Middleware-Replikate die Transaktionen in der gleichen Reihenfolge validieren. Die Validierung erfolgt wie beim SRCA, nur dass auf dem lokalen Middleware-Replikat nicht gegenüber lokalen Transaktionen validiert werden muss, da dies durch das Datenbanksystem übernommen wird.

4.5.3 Einsatzmöglichkeit in CbDBC

Bezüglich des Einsatzes im Constraint-basierten Datenbank-Caching sind folgende Ansätze des SRCA / SRCA-Rep von Interesse:

- Zunächst ist festzustellen, dass es sich um einen Middleware-basierten Ansatz handelt und somit dessen Vorteile genutzt werden können. Diese sind beispielsweise, dass Middleware-basierte Lösungen unabhängig vom verwendeten Datenbanksystem sind und dass sie in heterogenen Umgebungen eingesetzt werden können [LKPnMJP05]. Zusätzlich müssen keine Leseoperationen in der Middleware beachtet werden, die nur schwer beobachtbar sind.

- Weiterhin kann der Ansatz genutzt werden, die Transaktionen zunächst auf lokalen Replikaten auszuführen, und nach erfolgreicher Validierung auch auf allen anderen Replikaten. Dadurch können alle Datenreplikate aktuell gehalten werden.

Allerdings bringt dieser Ansatz einige Bedingungen mit sich, die den Einsatzbereich im Constraint-basierten Datenbank-Caching eingrenzen. Folgende Probleme können identifiziert werden:

- Zunächst bietet dieser Ansatz keine Lösungen an, um beispielsweise die Anomalie *Write Skew* des Mehrbenutzerbetriebs zu umgehen.

- Weiterhin stellt die Gruppenkommunikation ein Problem im Constraint-basierten Datenbank-Caching dar. Da eine Kommunikation eines total geordneten Multicast zwischen allen Knoten angenommen wird, muss dies jedes Mal über den „langsamen" Datenaustausch zwischen Backend und Cache erfolgen. Ein Vorteil ist, dass nicht in allen Knoten die Änderungsoperationen der Transaktion nachgezogen werden, da in den Caches lediglich Teilmengen der Daten vorhanden sind. Somit müssen nicht auf allen Caches die Daten aktualisiert werden und die Kommunikation muss nicht über alle Knoten erfolgen, sondern nur unter den Beteiligten.

- Die Einschränkung dieses Ansatzes ergibt sich durch die Forderung, dass die eingesetzten Datenbanksysteme einen konsistenten Schnappschuss vorhalten müssen. Diese Voraussetzung ist sehr restriktiv und kann nicht durch alle Datenbanksysteme realisiert werden. Beispielsweise wird für den derzeitigen Prototypen *ACCache* das Datenbanksystem DB2 verwendet, das keine Synchronisierung auf Basis der Snapshot Isolation bereitstellt. Aus diesen Gründen wurde überprüft, ob dennoch künstlich mehrere Versionen eines Datenelements erzeugt werden könnten. Hierzu stehen folgende Varianten zur Verfügung:

a Die Middleware speichert mehrere Versionen der Datenobjekte.

b Das Datenbanksystem speichert mehrere Versionen der Datenobjekte, indem entsprechende Informationen in den Datensätze gespeichert werden.

Diese beiden Ansätze haben zur Folge, dass die Ausführung der Anfragen entsprechend angepasst werden muss, so dass auf die richtigen Datenversionen zugegriffen werden kann. Dies bedeutet, dass ein künstliches Mehrversionenverfahren nachimplementiert werden muss, was einen erheblichen und unnötigen Aufwand darstellt. Weiterhin muss ein Garbage Collector entwickelt werden, der nicht mehr verwendete Datenversionen löscht. Werden in den Datensätzen die Versionsinformationen gespeichert, so muss der Primärschlüssel angepasst werden. Allerdings können dann die Fremdschlüsselbeziehungen nicht mehr abgebildet werden, da diese nicht auf einer Teilmenge des Primärschlüssels definiert werden können.

c Phaseneinteilung zwischen der Ausführung von lokalen Transaktionen und Remote-Transaktionen

In diesem Ansatz müssen optimale Zeitspannen festgelegt werden, in denen die entsprechenden Transaktionen ausgeführt werden können, ohne sich gegenseitig zu lange zu blockieren. Allerdings entstehen hierdurch Verzögerungen in der Ausführung der Transaktionen. Des Weiteren besteht die Gefahr, dass dadurch die Parallelität der Transaktionsausführung stark reduziert wird.

Da aus diesen Optionen weitere Probleme resultieren, hat die Voraussetzung der Unterstützung mehrerer Datenversionen durch das Datenbanksystem weitere Einschränkungen zur Folge. Somit ist es zunächst sinnvoller, eine, vom konkret verwendetem Datenbanksystem, unabhängige Lösung zu finden.

Mit Hilfe dieser Analyse konnte festgestellt werden, dass die grundsätzlichen Ideen des SRCA für den Einsatz im Constraint-basierten Datenbank-Caching geeignet sind, so dass es sehr wünschenswert wäre, dieses Verfahren dort anzuwenden. Allerdings überwiegen die noch zu lösenden Problemstellungen bezüglich der Einsetzbarkeit im Constraint-basierten Datenbank-Caching. Das Hauptproblem stellt hierbei der „langsame“ Datenaustausch zwischen Backend und den Caches dar. Für den SRCA/SRCA-Rep wird eine Gruppenkommunikation angenommen, die durch den „langsamen“ Datenaustausch sehr beeinträchtigt wird.

Zusammenfassend konnte bisher festgestellt werden, dass der Ansatz der Snapshot Isolation den Einsatzbereich des Constraint-basierten Datenbank-Caching sehr einschränkt, da er eine Datenbank mit Mehrversionenverwaltung voraussetzt. Dieser Ansatz wird zwar nicht ganz verworfen, allerdings wird zunächst angenommen, dass die verwendeten Datenbanksysteme keine Mehrversionenverwaltung anbieten. Darüber hinaus kann der Austausch der Nachrichten verringert werden, wenn genau festgelegt wird, welche Nachrichten an welche Knoten weitergeleitet werden. Auf diese Weise können Nachrichten beispielsweise gegenüber einem Multicast, wie er beim SRCA verwendet wird, eingespart werden.

Zur Entwicklung eines Synchronisierungsverfahren für das Constraint-basierte Datenbank-Caching liegt im Folgenden der Fokus auf einem Primary-Copy-Ansatz, da hierfür bereits die meisten Vorgaben erfüllt sind, wie z. B. die Architektur des Gesamtsystems. Ein konkreter Primary-Copy-Ansatz für das Constraint-basierte Datenbank-Caching wird im folgenden Abschnitt sowie in Kapitel 5 beschrieben und analysiert.

4.6 Primary-Copy-Ansatz für das CbDBC

Im Folgenden wird ein konkreter Primary-Copy-Ansatz für den Einsatz im Constraint-basierten Datenbank-Caching vorgestellt, wobei der Erkennung von *Write/Write-Konflikten* besondere Aufmerksamkeit gewidmet wird.

Die grundsätzliche Vorgehensweise basiert auf der Annahme, dass auf das verwendete Datenbanksystem eine weitere Schicht, das Cache-System, aufgesetzt wird, so dass das Verfahren vom Datenbanksystem unabhängig ist. Zusätzlich wird angenommen, dass das verwendete Datenbanksystem als Isolationsgrad die Konfliktserialisierbarkeit garantiert.

Die Durchführung von Transaktionsoperationen erfolgt basierend auf einer Variation des ROWA-Verfahrens. Die ROWA-Ausprägung kann wie folgt beschrieben werden: Operationen einer Transaktion werden, sofern die benötigten Daten auf dem Knoten vorhanden sind, direkt lokal ausgeführt. Wie bereits in Kapitel 3 beschrieben, ist vor allem von Interesse, ob Änderungsoperationen ihren Ursprung im Backend oder im Cache haben. Hat die Änderungsoperation ihren Ursprung im Backend, so wird die Operation dort ausgeführt und von dort aus auf den Caches nachgezogen. Liegt der Ursprung einer Änderungsoperation im Cache, so kann zunächst mit Hilfe der Sondierung festgestellt werden, ob die Datenänderung dort ausgewertet werden kann. Sind laut Sondierung nicht alle notwendigen Daten im Cache vorhanden, muss die Operation an das Backend weitergeleitet und dort durchgeführt werden. Andernfalls bestehen folgende Alternativen: Die Änderungsoperation wird direkt im Cache ausgeführt oder die Datenänderung wird ohne vorherige Ausführung im Cache an das Backend weitergeleitet. In beiden Fällen muss die Datenänderung auf jeden Fall an das Backend propagiert werden. Dann kann das Backend eine Koordinatorfunktion für die Datenänderungen übernehmen und die Änderungen werden auf allen notwendigen Knoten nachgezogen.

Eine detaillierte Vorgehensweise wird im folgenden Kapitel beschrieben. Weiterhin werden die Problemstellungen der Synchronisierung im Constraint-basierten Datenbank-Caching betrachtet und konkrete Lösungsansätze vorgestellt.

Kapitel 5

Synchronisierung im CbDBC

In diesem Kapitel wird ein Synchronisierungsverfahren entwickelt, das auf dem Primary-Copy-Ansatz basiert und speziell an die Bedürfnisse des Constraint-basierten Datenbank-Caching angepasst wird. Bevor das Synchronisierungsverfahren im Detail vorgestellt wird, werden zunächst grundlegende Annahmen für das Verfahren festgelegt. Darüber hinaus wird zur Abschätzung der Komplexität des Verfahrens ein Kostenmodell beschrieben. Danach wird die Grundidee und der Basisalgorithmus des hier entwickelten Verfahrens dargestellt. Damit der Basisalgorithmus nicht zu überladen wird und die Idee des Ablaufs leichter zu verstehen ist, werden danach Teilaspekte und auch Problemstellungen des Basisalgorithmus detailliert analysiert, wie z. B. die Erkennung und Auflösung von Konflikten oder Deadlocks. Abschließend wird ein Fazit gezogen und ein Ausblick auf zukünftige Forschungsarbeiten gegeben.

5.1 Grundlegende Annahmen

Damit das Verfahren und dessen Lösungsansätze zu verschiedenen Problemstellungen einfacher beschrieben werden können, werden zunächst folgende Annahmen festgelegt:

- Die verwendeten Datenbanksysteme erkennen Konflikte zwischen Transaktionen und können diese auflösen. Somit muss das Datenbanksystem auch Rollback- und Recovery-Mechanismen anbieten. Zur Betrachtung von Konflikten wird das R/X-Sperrverfahren angenommen. Dieses Sperrverfahren ist als Basis geeignet, da dessen Mächtigkeit ausreicht, um alle Probleme der Synchronisierung aufzeigen zu können.

- Bei der Vorstellung des Synchronisierungsverfahrens werden Nachrichtenverluste nicht beachtet, da es sich hier mehr um technische als um konzeptionelle Aspekte handelt. Aus diesem Grund wird ein Nachrichtenaustausch mit Hilfe verlustfreier Kanäle angenommen. Diese Kanaleigenschaft wird darüber hinaus durch die FIFO[1]-Eigenschaft erweitert. Dadurch kann die Reihenfolge der Nachrichten und somit die Abfolge der Operationen einer Transaktion auf allen Knoten eingehalten werden.

[1] First-In First-Out

- Transaktionen werden mit dem 2-Phasen-Commit-Protokoll abgeschlossen.

5.2 Verwendetes Kostenmodell

In diesem Abschnitt wird ein einfaches Kostenmodell vorgestellt, mit dem die Komplexität des Synchronisierungsverfahren abgeschätzt werden kann. In [HR01] wird ein Selektivitätsfaktor definiert, mit dessen Hilfe die Anzahl der erwarteten Tupel, die ein Prädikat erfüllen, berechnet werden kann. Mit Hilfe der Selektivitätsausdrücke können beispielsweise für mögliche Ausführungspläne einer Anfrage die Anzahl der Seitenzugriffe abgeschätzt werden. Im hier vorgestellten Synchronisierungsverfahren kann mit dem Selektivitätsfaktor z. B. die Anzahl der Datensätze abgeschätzt werden oder aber die Menge der nachzuladenden Datensätze aufgrund einer Datenänderung. Die benötigten Formel werden in Kapitel 5.2.1 vorgestellt und die Berechnung anhand einiger Beispiele verdeutlicht. Darüber hinaus wird in Kapitel 5.2.2 eine Cache-Selektivität zur Abschätzung der Anzahl zu aktualisierender Caches definiert. Des Weiteren ist der Nachrichtenaufwand des Synchronisierungsverfahrens von Interesse. Diesbezüglich müssen zum einen die Änderungsoperationen zwischen dem Backend und dem Cache ausgetauscht werden und zum anderen wird eine Transaktion mit Hilfe des 2-Phasen-Commit-Protokolls abgeschlossen. Damit der Nachrichtenaufwand besser abgeschätzt werden kann, wird in Kapitel 5.2.3 das Nachrichtenaufkommen des 2-Phasen-Commit-Protokoll im Detail vorgestellt.

5.2.1 Selektivitätsfaktor

Damit der Wert des Selektivitätsfaktors [HR01] verwendet werden kann, werden folgende Annahmen getroffen:

- Es wird eine Gleichverteilung der Werte eines Attributs angenommen.
- Weiterhin wird eine stochastische Unabhängigkeit der Werte verschiedener Attribute unterstellt.

Diese Annahmen werden beispielsweise ebenfalls durch kostenbasierte Anfrageoptimierer, zur Berechnung der Kosten eines auszuführenden Operators oder des kompletten Ausführungsplan einer Anfrage, verwendet [PI97]. Allerdings ist eine stochastische Unabhängigkeit der Attributwerte in der Realität nicht immer gegeben. Die Genauigkeit der Berechnung des Selektivitätsfaktors kann verbessert werden, sofern die Attributverteilung nicht durch eine einzige Zahl, sondern durch Histogramme beschrieben wird [HR01]. Dieser Ansatz wird beispielsweise in [PI97] verfolgt, wodurch jedoch die Formel zur Abschätzung der betroffenen Tupel einer Anfrage deutlich komplexer als bei [HR01] wird. Die Art der Berechnung nach [HR01] reicht für die Abschätzung des hier vorgestellten Synchronisierungsverfahrens aus, kann aber jederzeit analog zu dem Ansatz in [PI97] angepasst werden.

Mit Hilfe der nachfolgenden Formeln kann der *Selektivitätsfaktor* (SF) nach [HR01] bestimmt werden. Mit Hilfe des Selektivitätsfaktors kann die Menge an Datensätzen ermittelt werden, der ein Prädikat P erfüllen. Hierfür wird die Menge aller Tupel einer Tabelle mit *Kardinalität einer Tabelle* ($card(Tabelle)$)

bezeichnet. Die Menge aller Datensätze, die ein Prädikat erfüllen, kann wie folgt berechnet werden:

$$card(\sigma_P(R)) = SF(P) \cdot card(R) \tag{5.1}$$

Weiterhin werden folgende Formeln definiert:

- $A_i = a_i$:

$$SF = \begin{cases} \frac{1}{j_i} & \text{wenn ein Index auf } A_i \text{ definiert ist} \\ \frac{1}{10} & \text{sonst} \end{cases} \tag{5.2}$$

- $A_i = A_k$:

$$SF = \begin{cases} \frac{1}{max(j_i, j_k)} & \text{wenn ein Index auf } A_i \text{ und } A_k \text{ definiert ist} \\ \frac{1}{j_i} & \text{wenn ein Index auf } A_i \text{ definiert ist} \\ \frac{1}{j_k} & \text{wenn ein Index auf } A_k \text{ definiert ist} \\ \frac{1}{10} & \text{sonst} \end{cases} \tag{5.3}$$

- $A_i \geq a_i$ **oder** $A_i > a_i$:

$$SF = \begin{cases} \frac{a_{max} - a_i}{a_{max} - a_{min}} & \text{wenn Index auf } A_i \text{ und der Wert interpolierbar ist} \\ \frac{1}{3} & \text{sonst} \end{cases} \tag{5.4}$$

- A_i **between** a_i **and** a_k:

$$SF = \begin{cases} \frac{a_k - a_i}{a_{max} - a_{min}} & \text{wenn Index auf } A_i \text{ und der Wert interpolierbar ist} \\ \frac{1}{4} & \text{sonst} \end{cases} \tag{5.5}$$

Aufgrund der stochastischen Unabhängigkeit der verschiedenen Attributwerte gelten folgende Gleichungen:

$$\begin{aligned} SF(P(A) \wedge P(B)) &= SF(P(A)) \cdot SF(P(B)) \\ SF(P(A) \vee P(B)) &= SF(P(A)) + SF(P(B)) - SF(P(A)) \cdot SF(P(B)) \\ SF(\neg P(A)) &= 1 - SF(P(A)) \end{aligned} \tag{5.6}$$

Zusätzlich zum Selektivitätsfaktor wird ein *Join-Selektivitätsfaktor* (JSF) definiert, der eine Abschätzung der erwarteten Datenmenge bei der Durchführung von Equi-Joins ermöglicht. Hierfür ist folgende Gleichung von Bedeutung:

$$card(R \bowtie S) = JSF \cdot card(R) \cdot card(S) \tag{5.7}$$

Bei einem verlustfreien (N:1)-Join gilt:

$$card(R \bowtie S) = Max(card(R), card(S)) \tag{5.8}$$

Mit Hilfe dieser Werte und Gleichungen kann für das in Kapitel 5.4 vorgestellte Synchronisierungsverfahren abgeschätzt werden, wie viele Datensätze von einer Änderung betroffen sind und wie viele Datensätze gegebenenfalls aktualisiert oder nachgeladen werden müssen. Die Berechnung des Selektivitätsfaktors wird anhand des folgenden Beispiels veranschaulicht.

Person	id	name	beruf	alter	card(Person) = 5000
		#W = 100	#W = 20		

Manager	id	name	abteilung	card(Manager) = 200
		#W = 50		

Abbildung 5.1: Beispieltabellen zur Berechnung des Selektivitätsfaktors

Einfache Beispiele zur Berechnung des Selektivitätsfaktors

In Abbildung 5.1 werden Tabellen und die Anzahl der unterschiedlichen Werte (#W) je Attribut dargestellt. Mit Hilfe der zuvor vorgestellten Gleichungen kann die Anzahl der Tupel (#T) des Resultats der folgenden Statements berechnet werden. Hierfür wird zunächst der Selektivitätsfaktor aufgrund der Where-Klausel (WK) bestimmt und im nächsten Schritt die Anzahl der Tupel mit Hilfe der Formel 5.1 berechnet.

select * from Person where name = 'Mueller'

Aufgrund der Formel 5.2 beträgt der Selektivitätsfaktor

$$SF_{WK} = \begin{cases} \frac{1}{100} & \text{wenn Index auf dem Attribut } name \text{ definiert ist} \\ \frac{1}{10} & \text{sonst} \end{cases}$$

$$\#T = SF_{WK} \cdot card(Person)$$

select * from Person where name = 'Mueller' and beruf = 'Programmierer'

Aufgrund der Formeln 5.2 und 5.6 gilt für diese *Select*-Anfrage

$$SF_{WK} = \begin{cases} \frac{1}{100} \cdot \frac{1}{20} & \text{wenn Indizes auf beiden Attributen definiert sind} \\ \frac{1}{100} \cdot \frac{1}{10} & \text{wenn Index nur auf } Person.name \text{ definiert ist} \\ \frac{1}{10} \cdot \frac{1}{20} & \text{wenn Index nur auf } Person.beruf \text{ definiert ist} \\ \frac{1}{10} \cdot \frac{1}{10} & \text{sonst} \end{cases}$$

$$\#T = SF_{WK} \cdot card(Person)$$

select Person.*, Manager.* from Person, Manager where Person.id = Manager.id and Person.name = 'Mueller'

Aufgrund der Formel 5.2 gilt für die Bedingung *Person.name = 'Mueller'*

$$SF_{WK} = \begin{cases} \frac{1}{100} & \text{wenn Index auf dem Attribut } name \text{ definiert ist} \\ \frac{1}{10} & \text{sonst} \end{cases}$$

Wird ein verlustfreier (N:1)-Join angenommen, so gilt laut Formel 5.8:

$$\#T = card(Person \bowtie Manager) \cdot SF_{WK}$$
$$= Max(card(Person), card(Manager)) \cdot SF_{WK}$$

Analog wird die Where-Klausel einer Änderungsoperation ausgewertet, um die Anzahl zu aktualisierender Datensätze abschätzen zu können.

5.2.2 Cache-Selektivität

In diesem Abschnitt wird die Cache-Selektivität definiert, als die Anzahl zu aktualisierender Caches. Diese wird zur Bestimmung der Anzahl der Knoten, die von der Änderungsoperation betroffen sind, genutzt.

Da im Backend alle Daten existieren, wird eine Änderungsoperation dort immer ausgeführt. In den Caches werden jedoch nur Teilmengen der Daten abgebildet, so dass nicht alle Änderungsoperationen an die Caches weitergeleitet werden müssen. Dies tritt beispielsweise dann ein, wenn die notwendigen Daten lediglich im Backend vorhanden sind. Muss die Änderungsoperation an keinen Cache propagiert werden, so ist die Cache-Selektivität Null. Aus diesem Grund ist nur der Fall von Interesse, dass mindestens ein Cache aktualisiert werden muss. Sei dieses Ereignis durch *Änderung* mit der Wahrscheinlichkeit W_{prop} gegeben. Weiterhin wird angenommen, dass unter der Bedingung *Änderung* 5% der restlichen Caches ebenfalls aktualisiert werden müssen. Somit ergibt sich für die Cache-Selektivität (CSF):

$$CSF = \begin{cases} W_{prop} \cdot \lceil 1 + 0,05 \cdot (Cacheanzahl - 1) \rceil & \text{unter Bedingung: } \textit{Änderung} \\ 0 & \text{sonst} \end{cases} \tag{5.9}$$

Hat eine Änderungsoperation ihren Ursprung im Backend, entspricht die Cache-Selektivität der Anzahl zu aktualisierender Knoten. Liegt der Ursprung im Cache, muss zusätzlich das Backend aktualisiert werden, womit sich die Anzahl zu aktualisierender Knoten zu Cache-Selektivität +1 ergibt.

Mit Hilfe dieser Formel kann im Folgenden beispielsweise der Nachrichtenaufwand aufgrund einer Änderungsoperation abgeschätzt werden. Diese Abschätzung kann gegebenenfalls in zukünftigen Forschungsarbeiten mit Hilfe von Messungen bestätigt oder genauer bestimmt werden.

5.2.3 Nachrichten des 2-Phasen-Commit-Protokolls

Da im Synchronisierungsverfahren für den Abschluß der Transaktionen das 2-Phasen-Commit-Protokoll (2PC-Protokoll) verwendet wird, wird in diesem Abschnitt dessen Nachrichtenaufwand betrachtet. Im 2PC-Protokoll werden die in Abbildung 5.2 dargestellten Nachrichten ausgetauscht. Unter der Annahme, dass das Backend die Koordinatorfunktion übernimmt, sind zwischen dem Backend und einem Cache (Agent) 4 Nachrichten notwendig. Da allerdings Änderungsoperationen gegebenenfalls auf mehreren Caches nachgezogen werden müssen, müssen diese 4 Nachrichten mit der Anzahl zu aktualisierender Caches multipliziert werden, also:

$$\text{Anzahl Nachrichten} = 4 \cdot \text{Anzahl zu aktualisierender Caches} \tag{5.10}$$

Bei dieser Nachrichtenkommunikation sind die synchronen Log-Ausgaben besonders von Interesse, da diese unter keinen Umständen verzögert durchgeführt werden können. Die Anzahl der synchronen Log-Ausgaben beträgt

$$\text{Anzahl synchroner Log-Ausgaben} = 2 + 2 \cdot \text{Cacheanzahl}$$

Mit Hilfe dieses Nachrichtenaufwands des 2PC-Protokolls kann der Gesamtnachrichtenaufwand des Synchronisierungsverfahrens genauer abgeschätzt werden.

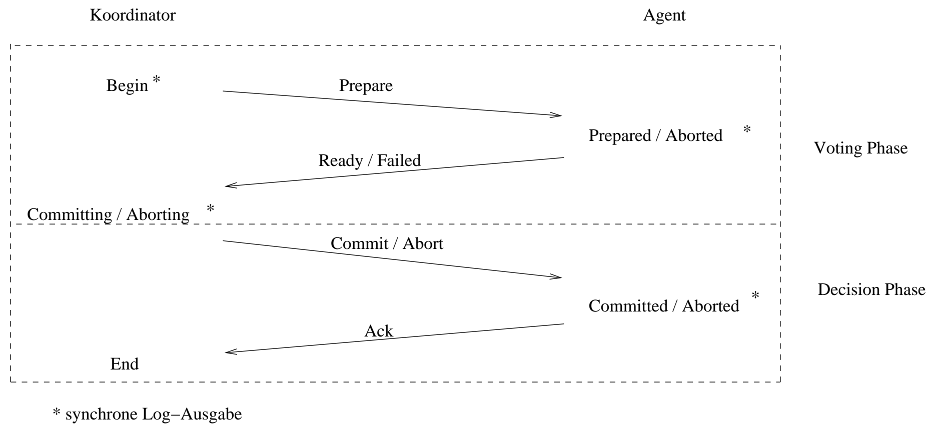


Abbildung 5.2: Nachrichten des 2PC-Protokolls [HR01]

5.2.4 Zusammenfassung des Kostenmodells

Folgende Abkürzungen werden definiert:

- Anzahl der Nachrichten einer Transaktion: N
- Anzahl der Nachrichten des 2-Phasen-Commit-Protokolls: N_{2PC}
- Anzahl der Nachrichten zum Nachladen von Datensätzen: N_L
- Anzahl der vorhandenen Caches: C
- Anzahl der zu aktualisierenden Caches: C_{akt}
- Anzahl der von einer Änderung betroffenen Datensätze: T
- Anzahl der Operationen innerhalb einer Transaktion: O
- Anzahl der Write-Operationen einer Transaktion: O_w
- Es werden unterschiedliche Tabellen betrachtet und mit $A, B, C, ...$ bezeichnet.
- Anzahl der Datensätze der Tabelle A: $card(A)$

Somit können die bisher festgelegten Formeln wie folgt konkretisiert werden:

- In Kapitel 5.2.2 wurde die Cache-Selektivität als die Anzahl zu aktualisierender Caches definiert. Somit gilt Cache-Selektivität = C_{akt} und die Formel 5.9 kann folgendermaßen konkretisiert werden:

$$C_{akt} = \begin{cases} W_{prop} \cdot \lceil 1 + 0,05 \cdot (C-1) \rceil & \text{unter der Bedingung } \textit{Änderung} \\ 0 & \text{sonst} \end{cases} \tag{5.11}$$

- Die Nachrichtenanzahl des 2-PC-Protokolls aus Formel 5.10 betragen

$$N_{2PC} = 4 \cdot C_{akt}$$

$$N_{2PC} = \begin{cases} 4 \cdot W_{prop} \cdot \lceil 1 + 0,05 \cdot (C-1) \rceil & \text{unter der Bedingung } \textit{Änderung} \\ 0 & \text{sonst} \end{cases} \quad (5.12)$$

5.3 Grundidee

Grundlegend für dieses Synchronisierungsverfahren ist, dass auf das verwendete Datenbanksystem eine weitere Schicht, das Cache-System, aufgesetzt wird. Die komplette Interaktion des Benutzers sowie die Kommunikation zwischen dem Backend und den Caches geschieht über das Cache-System.

Die Grundidee des Basisalgorithmus liegt darin, die Ideen folgender Verfahren zu kombinieren: Update-everywhere, Primary-Copy und das ROWA-Verfahren. Zur Umsetzung der Architektur des Primary-Copy-Verfahrens stellt das Backend die Primärkopie der Daten bereit. Die Caches repräsentieren die Backup-Replikate, wobei hier lediglich Teilmengen der Backend-Daten gespeichert werden. Wird ein reiner Primary-Copy-Ansatz verwendet, so dürfen Änderungsoperationen nur auf der Primärkopie erfolgen. Erst danach werden diese Datenänderungen auf den Caches nachgezogen. Diese Vorgehensweise soll dahingehend verbessert werden, dass bereits im Cache Änderungsoperationen ausgeführt werden dürfen. Aus diesem Grund werden folgende angepasste Eigenschaften des ROWA-Verfahrens als Voraussetzungen aufgenommen:

- Leseoperationen dürfen auf allen Knoten (Backend oder Cache) ausgeführt werden, welche die durch die Anfrage benötigten Daten enthalten. Wird jedoch eine Anfrage an einen Cache gestellt, der nicht alle notwendigen Datenelemente enthält, so wird die Leseoperation an das Backend weitergeleitet und dort ausgewertet.

- Änderungsoperationen sind ebenfalls auf all den Knoten erlaubt, die alle von der Änderungsoperation betroffenen Datenelemente vorhalten. Wird eine Änderungsoperation an den Cache gestellt, so muss dieser die Datenänderung zwingend an das Backend weiterleiten, egal ob die Änderungsoperation im Cache ausgeführt werden konnte oder nicht. Das Backend übernimmt dann die Verantwortung, dass die Datenänderung auf allen Caches nachgezogen wird, so dass die Datenelemente auf allen Knoten aktualisiert werden und somit alle den gleichen Datenzustand darstellen.

Diese Vorgehensweise könnte man als folgende Ausprägung des ROWA-Verfahrens bezeichnen: *read one if possible write all necessary (ROpWAn)*. *Read one if possible* bedeutet in diesem Zusammenhang, dass eine *Select*-Anfrage direkt durch den angefragten Knoten ausgeführt wird, sofern dies möglich ist. Andernfalls muss die Anfrage auf der Primärkopie erfolgen. *Write all necessary* beschreibt die Situation, dass nur solche Caches aktualisiert werden müssen, die von der Änderung betroffene Daten enthalten. Änderungsoperationen müssen vom Backend aus an all die Caches weitergeleitet werden, die von der Änderung betroffenen Daten enthalten. Auf diese Weise können die Daten auf allen Knoten aktuell gehalten werden, indem beispielsweise mit Hilfe von Write Sets die Datenänderung nachgezogen wird. In der folgenden Beschreibung des Basisalgorithmus werden im Fall der Weiterleitung einer Änderungsoperation allgemein alle Caches referenziert, ohne explizit darauf hinzuweisen, dass nur die Caches

von Interesse sind, die zu aktualisierende Datenelemente enthalten. Da je nach Wissensstand des Backend die Anzahl der zu aktualisierenden Caches variiert, wird dieser Aspekt in Kapitel 5.5.1 betrachtet. Dort wird im Detail beschrieben, wie all die Caches identifiziert werden können, an die eine Änderungsoperation weitergeleitet werden muss.

5.4 Basisalgorithmus

Basierend auf der zuvor beschriebenen Grundidee wird in diesem Abschnitt ein Basisalgorithmus entwickelt, der die Synchronisierung der Transaktionen übernimmt. Da sich eine Transaktion sich aus Lese- und Änderungsoperationen und aus der Commit- oder Abortoperation zusammensetzt, werden diese Operationen getrennt bezüglich ihrer Durchführung betrachtet. Aufgrund von Änderungsoperationen müssen gegebenenfalls Datensätze in den Cache nachgeladen werden. In den folgenden Erläuterungen wird der Mehraufwand, der durch das Nachladen von Datensätzen entsteht, zunächst vernachlässigt. In Kapitel 5.5.5 wird dies jedoch im Detail betrachtet.

5.4.1 Leseoperationen

Hat eine Leseoperation ihren Ursprung im Backend, so wird diese Operation zur Ausführung an die Datenbank weitergeleitet. Sobald die Leseoperation dort erfolgreich ausgewertet wurde, wird das Ergebnis über das Cache-System an den Benutzer zurückgegeben.

Liegt der Ursprung der Leseoperation im Cache, so muss zunächst mit Hilfe der Sondierung geprüft werden, ob die Anfrage im Cache ausgewertet werden kann. Sind alle notwendigen Daten im Cache vorhanden, so wird die Anfrage lokal im Cache bearbeitet. Andernfalls wird die Leseoperation an das Backend weitergeleitet und dort ausgeführt. Das Ergebnis wird dann vom Backend über den Cache an den Benutzer zurückgegeben.

5.4.2 Änderungsoperationen

Bei der Vorstellung der Grundidee wurde beschrieben, dass Änderungsoperationen nach dem Primary-Copy-Ansatz oder nach dem ROpWAn-Verfahren behandelt werden können. In diesem Abschnitt werden die Umsetzungen dieser beiden Alternativen im Basisalgorithmus vorgestellt.

Umsetzung des Primary-Copy-Ansatzes

Werden Änderungsoperationen nach dem Primary-Copy-Ansatz umgesetzt, so dürfen alle Datenänderungen nur auf der Primärkopie, also auf dem Backend, erfolgen. Hat eine Änderungsoperation ihren Ursprung im Cache, so wird die Operation direkt an das Backend propagiert, um sie dort auszuführen.

Eine Änderungsoperation wird folgendermaßen im Backend durchgeführt: Zunächst wird die Operation vom Cache-System an das Datenbanksystem weitergeleitet und dort bearbeitet. Durch die Verarbeitung im Datenbanksystem wird ein entsprechendes Write Set erzeugt, das all die geänderten Objekte enthält, und im Cache-System gespeichert. Muss die Änderungsoperation in den

Caches nachgezogen werden, so wird das Write Set der Datenänderung an die Caches propagiert. Mit Hilfe dieses Write Set können die von der Änderung betroffenen Daten im Cache aktualisiert werden. Darüber hinaus kann sichergestellt werden, dass auf jedem Knoten dieselben Datenelemente verändert werden.

Aufgrund der Verarbeitung aller Änderungsoperationen aller Transaktionen im Datenbanksystem des Backend können Konflikte zwischen diesen Operationen auftreten. Diese Konflikte können mit Hilfe der vergebenen Sperren direkt durch das Datenbanksystem erkannt und aufgelöst werden.

Umsetzung des ROpWAn-Verfahrens

Die Umsetzung des ROpWAn-Verfahrens unterscheidet sich insofern von der des Primary-Copy-Ansatzes, dass die Bearbeitung von Änderungsoperationen bereits im Cache erlaubt wird. Soll eine Datenänderung im Cache ausgeführt werden, so wird zunächst mit Hilfe der Sondierung, die bereits für die Auswertung von *Select*-Anfragen verwendet wird, überprüft, ob alle von der Änderung betroffenen Daten im Cache existieren. Sind nicht alle Daten im Cache vorhanden, so erfolgt der Ablauf analog zum Primary-Copy-Ansatz. Existieren hingegen die notwendigen Daten im Cache, so wird die Änderungsoperation im lokalen Datenbanksystem bearbeitet. Bei der Ausführung der Datenänderung wird das entsprechende Write Set erzeugt, das vom Cache an das Backend weitergeleitet wird. Von dort aus können die Daten aller anderen Knoten aktualisiert werden, indem diese Knoten die Änderungsoperation mit Hilfe des Write Set nachziehen.

Aufgrund dieses Ablaufs können Konflikte zwischen Transaktionen im Cache und im Backend auftreten. Analog zum Primary-Copy-Verfahren können Konflikte zwischen Transaktionsoperationen lokal durch das Datenbanksystem erkannt und aufgelöst werden. Zusätzlich können Konflikte zwischen Write Sets verschiedener Transaktionen im Cache-System erkannt werden. Sobald ein Write Set einen Knoten erreicht, überprüft das Cache-System, ob Konflikte zu bereits vorhandenen Write Sets auftreten. Existieren Konflikte zwischen Write Sets, so kann bereits vor der Verarbeitung des Write Set ein Write/Write-Konflikt durch das Cache-System erkannt werden und eine der Konflikt-Transaktionen abgebrochen werden. Eine detaillierte Analyse, welche dieser Konflikt-Transaktionen abgebrochen werden sollte, wird in Kapitel 5.5.2 diskutiert.

Allerdings besteht basierend auf dieser Realisierung des ROpWAn-Verfahrens die Gefahr, dass Änderungsoperationen im Cache durchgeführt werden, jedoch im Backend beispielsweise aufgrund einer Verletzung einer Integritätsbedingung zurückgewiesen werden. Dieses Problem kann beispielsweise für Fremdschlüsselbeziehungen folgendermaßen verhindert werden: Die Fremdschlüsselbeziehungen werden zwar nicht in der Cache-Datenbank angelegt, sind jedoch im Cache bekannt. So können zusätzlich zur Sondierung die Fremdschlüsselbeziehungen überprüft werden. Nur wenn die Sondierung und die Überprüfung der Fremdschlüsselbeziehungen ein positives Ergebnis zurückgeben, darf eine Änderungsoperation im Cache ausgeführt werden. Weitere mögliche Probleme, die bei einer direkten Ausführung einer Änderungsoperation im Cache auftreten können, werden in Kapitel 5.5.6 im Detail betrachtet.

Im Kommunikationsablauf zwischen dem Benutzer und den Knoten ist zusätzlich von Interesse, wann der Benutzer über die erfolgreiche oder erfolglose Durchführung einer Operation informiert wird. Kann eine Änderungsoperati-

on nicht im Cache ausgeführt werden, wird sie an das Backend weitergeleitet und die Benachrichtigung erfolgt erst nach der dortigen Durchführung. Wurde jedoch die Datenänderung bereits im Cache verarbeitet, so stehen folgende Alternativen zur Verfügung.

- Sobald der Cache die Änderungsoperation ausgeführt hat, wird das Write Set an das Backend weitergeleitet und der Benutzer erhält die Benachrichtigung. Somit muss der Benutzer nicht warten, bis die Änderung im Backend durchgeführt wurde, sondern kann früher mit weiteren Operationen innerhalb der Transaktion fortfahren. Allerdings besteht die Gefahr, dass ein Benutzer glaubt, dass die Änderungsoperation erfolgreich ausgeführt wurde, diese jedoch aufgrund z. B. der Verletzung von Integritätsbedingungen im Backend zurückgewiesen wird. Dies erfährt der Benutzer gegebenenfalls erst nachdem er weitere Operationen ausgeführt hat. Dieses Vorgehen darf unter keinen Umständen zugelassen werden, da hierdurch die Transparenz des Gesamtsystems verletzt wird. Darüber hinaus können auf diese Weise die ACID-Eigenschaften einer Transaktion nicht gewährleistet werden.

- Der Benutzer wird erst nach der Ausführung im Backend über die Durchführung informiert. Dadurch muss der Benutzer zwar länger warten, bis er weitere Operationen innerhalb der Transaktion erstellen kann, aber es kann sicher gestellt werden, dass die Datenänderung ausgeführt werden darf.

Somit wird in dieser Realisierung des ROpWAn-Verfahrens der Benutzer erst nach der Verarbeitung einer Änderungsoperation im Backend über deren erfolgreiche oder erfolglose Durchführung informiert.

Im Verlauf des Basisalgorithmus kann eine Reduzierung des Nachrichtenaufwands erreicht werden, sobald eine Änderungsoperation im Cache ausgeführt werden kann. Laut des ROpWAn-Ansatzes wird die Änderungsoperation im Cache durchgeführt, bevor die Datenänderung an das Backend weitergeleitet wird. Danach wird die Datenänderung im Backend nachgezogen und die Daten in den Caches auf den neuesten Stand gebracht. Diese Aktualisierung ist allerdings nicht für den Ursprungs-Cache notwendig, da dort die Änderungsoperation bereits verarbeitet wurde. Um dies im Backend erkennen zu können, kann der Cache z. B. in der Nachricht der Datenänderung bereits das Ergebnis der Sondierung mitschicken. So hat das Backend das Wissen, ob eine Änderungsoperation bereits im Cache ausgeführt wurde, oder nicht. Entsprechend muss gegebenenfalls ein Cache weniger aktualisiert werden.

5.4.3 Commit

Da die Daten einerseits im Backend gespeichert werden und andererseits Teilmengen dieser Daten ebenfalls in Caches abgebildet werden, müssen einzelne Operationen einer Transaktion gegebenenfalls nicht nur auf einem Knoten ausgeführt werden. Es können folgende Szenarien unterschieden werden, auf die bereits in Kapitel 4.3.1 hingewiesen wurde:

single node synchronization

Alle Operationen einer Transaktion und somit auch das Commit werden

lokal auf einem Knoten durchgeführt. Dieser Fall tritt beispielsweise dann auf, wenn eine Transaktion ihren Ursprung im Backend hat und lediglich Operationen enthält, die nicht an die Caches weitergeleitet werden müssen. Hat eine Transaktion hingegen ihren Ursprung im Cache, so ist dieser Fall lediglich bei Transaktionen möglich, die nur Leseoperationen enthalten, die alle im Cache ausgewertet werden können. In diesem Szenario wird die Entscheidung über das Commit der Transaktion nur lokal getroffen, da keine weiteren Knoten von der Transaktion betroffen sind.

team synchronization
Das Backend und ein Cache bilden ein Team bezüglich der Durchführung einer Transaktion. In einem solchen Fall hat eine Transaktion zum einen ihren Ursprung im Cache und zum anderen können nicht alle Operationen dieser Transaktion allein im Cache bearbeitet werden. Dies können z. B. Lese- oder Änderungsoperationen sein, für welche die benötigten Daten im Cache fehlen. Damit in diesem Szenario eine Transaktion abgeschlossen werden kann, muss die Entscheidung über das Commit mit Hilfe des 2PC-Protokolls über diese beiden Knoten (Backend und Cache) erfolgen und durchgeführt werden. Hier kann der Cache die Koordinator-Funktion im 2PC-Protokoll übernehmen.

cluster synchronization
Eine Transaktion wird im Backend und auf mehreren Caches ausgeführt. Dieser Fall tritt dann auf, wenn Änderungsoperationen einer Transaktion auf den Caches nachgezogen werden müssen. Die Entscheidung über das Commit dieser Transaktion muss mit Hilfe des 2PC-Protokolls über alle Knoten (Backend und Caches) erfolgen, die Änderungsoperationen dieser Transaktion ausgeführt oder nachgezogen haben. Das Backend übernimmt in diesem Fall die Koordinatorfunktion, da alle Nachrichten des 2PC-Protokolls über das Backend erfolgen müssen, weil die Caches nicht untereinander kommunizieren können. Die Commit-Anforderung muss hierfür gegebenenfalls zunächst vom Cache an das Backend weitergeleitet werden.

all synchronization
Eine Transaktion wird auf dem Backend und allen vorhandenen Caches ausgeführt. Dieser Fall wird prinzipiell aufgrund der in Kapitel 5.2.2 definierten Cache-Selektivität verhindert, da hier beschrieben wird, dass eine Änderung lediglich einige und nicht alle Caches betrifft. Jedoch besteht die Möglichkeit, dass eine Transaktion eine Menge an Änderungsoperationen enthält, so dass letztendlich alle Caches an der Transaktionsdurchführung beteiligt sind. In diesem Fall erfolgt das 2PC-Protokoll über alle Knoten, wobei das Backend die Koordinator-Funktion übernimmt. Dieser Fall wird allerdings als sehr unwahrscheinlich angenommen.

Mit Hilfe dieser Fälle kann genau bestimmt werden, welche Knoten in die Entscheidung über das Commit einer Transaktion im Laufe des 2PC-Protokolls einbezogen werden müssen.

Wird eine Operation auf einem Knoten ungleich ihres Ursprungsknoten bearbeitet, so wird diese Operation auf diesem Knoten als Nachzieheoperation bezeichnet. Die Menge aller Nachzieheoperationen einer Transaktion beschreibt eine Nachziehtransaktion auf dem Knoten. Allgemein muss somit die Entscheidung

über das Commit und dessen Ausführung über den Ursprungsknoten und all die Knoten erfolgen, die eine entsprechende Nachziehtransaktion bearbeitet haben. Somit werden im Backend und im Cache lokale Transaktionen sowie Nachziehtransaktionen ausgeführt.

Zur Veranschaulichung der notwendigen Aktionen zur Ausführung einer Transaktion im Backend und im Cache dient der Pseudocode in Anhang B.1 und B.2. Im Pseudocode wird das ROpWAn-Verfahren umgesetzt. Zusätzlich wird die Optimierung realisiert, dass das Sondierungsergebnis mit der Datenänderung an das Backend geschickt wird.

5.4.4 Kosten des Basisalgorithmus

Mit Hilfe des in Kapitel 5.2 vorgestellten Kostenmodells können folgende Formeln entwickelt werden, welche die Nachrichtenanzahl pro Transaktion abschätzen.

Transaktion wird auf dem Backend initiiert

Unter der Bedingung, dass die Änderungsoperationen an die Caches weitergeleitet werden müssen, lässt sich die Anzahl der Nachrichten einer Transaktion wie folgt berechnen: Zunächst müssen alle Write Sets der Änderungsoperationen an alle zu aktualisierenden Caches weitergeleitet werden ($O_w \cdot C_{akt}$), so dass dort die Daten aktualisiert werden können. Zusätzlich fallen die Nachrichten des 2PC-Protokolls an (N_{2PC}), um die Transaktion auf allen Knoten erfolgreich zu beenden oder abzubrechen. Weiterhin kann es notwendig sein, dass in den Caches aufgrund der Datenänderung Datensätze nachgeladen werden müssen ($N_L \cdot C_{akt}$). Hierzu sind gegebenenfalls weitere Nachrichten notwendig und es resultiert folgende Abschätzung:

$$N = O_w \cdot C_{akt} + N_{2PC} + N_L \cdot C_{akt} \tag{5.13}$$

Transaktion wird auf dem Cache initiiert

Primary-Copy-Ansatz Folgende Nachrichten sind bei diesem Ansatz notwendig: Zunächst müssen alle Änderungsoperationen an das Backend weitergeleitet werden, um dort die Datenänderung (O_w) auszuführen. Sobald eine Änderung erfolgreich durchgeführt wurde, wird der Cache darüber informiert, so dass der Benutzer gegebenenfalls weitere Operationen ausführen kann (O_w). Danach muss das Backend alle Caches aktualisieren ($O_w \cdot C_{akt}$), indem das Write Set an alle betroffenen Caches weitergeleitet wird. Zum Abschluss einer Transaktion wird die Commit-Nachricht an das Backend propagiert und das Backend übernimmt die Rolle des Koordinators im 2PC-Protokoll, so dass hier die entsprechenden Nachrichten anfallen (N_{2PC}). Aufgrund der Datenänderungen sind gegebenenfalls Nachrichten zum Nachladen von Datensätzen notwendig ($N_L \cdot C_{akt}$). Folgende Formel resultiert für den Primary-Copy-Ansatz:

$$N = 2 \cdot O_w + O_w \cdot C_{akt} + 1 + N_{2PC} + N_L \cdot C_{akt} \tag{5.14}$$

ROpWAn-Verfahren und das Sondierungsergebnis wird mitgeschickt Bezüglich der Formel 5.14 ergibt sich der Unterschied, dass vom Backend aus

nicht alle Caches aktualisiert werden müssen. Der Cache auf dem die Transaktion ihren Ursprung hat, hat die Änderungsoperation bereits ausgeführt und muss somit nicht über die Datenänderung informiert werden. Der Wert C'_{akt} spiegelt diese Situation wider. Somit resultiert folgende Formel, sofern eine erfolgreiche Ausführung aller Änderungsoperationen im Cache angenommen wird:

$$C'_{akt} = \begin{cases} W_{prop} \cdot \lceil 0,05 \cdot (C-1) \rceil & \text{unter der Bedingung } \textit{Änderung} \\ 0 & \text{sonst} \end{cases}$$

$$N = 2 \cdot O_w + O_w \cdot C'_{akt} + 1 + N_{2PC} + N_L \cdot C_{akt} \tag{5.15}$$

5.5 Lösungen der konkreten Problemstellungen

In der Vorstellung der Grundidee (vgl. Kapitel 5.3) und des Basisalgorithmus (vgl. Kapitel 5.4) wurden einige Aspekte nicht im Detail betrachtet, damit die Idee des Basisalgorithmus deutlicher wird. Aus diesem Grund werden folgende Problemstellungen in den nächsten Abschnitten im Detail analysiert:

- Wissensstand des Backend (eingeschränktes Wissen vs. volles Wissen)
- Konflikterkennung und Konfliktbehandlung
- Deadlocks
- Anomalien
- Nachladeaktionen

Zusätzlich wird erneut auf die Probleme eingegangen, die bezüglich der Ausführung von Änderungsoperationen im Cache auftreten können und bereits in Kapitel 3.4 kurz vorgestellt wurden.

5.5.1 Wissensstand des Backend

Wie bereits in Kapitel 3.3 beschrieben, können je nach Wissensstand des Backend die Aktionen beim Weiterleiten eines Write Set an den Cache unterschiedlich ausfallen. Folgende Szenarien wurden vorgestellt

1. Das Backend weiß lediglich von der Existenz der Caches. Das Backend hat sehr wenig Wissen.
2. Das Backend kennt die vorhandenen Caches, sowie deren *Cache-Group-Definition*. Das Backend hat eingeschränktes Wissen.
3. In diesem Szenario kennt das Backend alle vorhandenen Caches inklusive ihrer *Cache-Group-Definition* und deren *Cache-Group-Inhalt*. Das Backend hat volles Wissen.

In Kapitel 3.3 wurden bereits deren Vor- und Nachteile erörtert. Basierend auf dieser Analyse kann festgestellt werden, dass in Variante 1 die Nachteile überwiegen und somit dieser Ansatz nicht umgesetzt werden sollte. Für die beiden anderen Ansätze wird in diesem Abschnitt diskutiert, welche Informationen gespeichert werden müssen und wie die benötigten Daten ausgewertet werden können.

Backend hat eingeschränktes Wissen

Um feststellen zu können, welche Caches von einer Änderungsoperation betroffen sind, muss zunächst die Cache-Group-Definition der existierenden Cache Groups im Backend gespeichert werden. Ein Vorschlag für eine Datenstruktur zum Speichern dieser Daten wird in Abbildung 5.3 dargestellt.

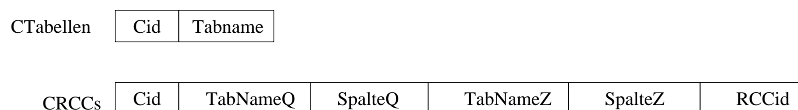


Abbildung 5.3: Datenstruktur zur Speicherung der Cache-Informationen (Eingeschränktes Wissen im Backend)

Aufgrund des eingeschränkten Wissensstands des Backend kann nur anhand der betroffenen Tabelle und nicht auf Datensatzebene entschieden werden, welche Caches aktualisiert werden müssen. Somit muss lediglich der Tabellenname (tabellenName) aus dem Write Set extrahiert werden und folgendes Statement kann die zu aktualisierenden Caches identifizieren:

```
SelectCaches:
select Cid from CTabellen where TabName = tabellenName;
```

Hierbei ist zu beachten, das mit diesem Statement lediglich die Caches identifiziert werden, die eine einzelne Änderungsoperation nachziehen müssen. Da das 2PC-Protokoll über all die Caches erfolgen muss, die mindestens eine der Änderungsoperationen einer Transaktion ausgeführt haben, ist es notwendig alle Caches bezüglich einer Transaktion zu identifizieren. Aus diesem Grund könnte z. B. je aktueller Transaktion eine Liste im Backend gespeichert wird, die alle Caches enthält, an die mindestens eine Änderungsoperation weitergeleitet wurde. Alternativ könnte vor dem Beginn des 2PC-Protokolls die Liste der Caches erstellt werden, indem erneut für alle Änderungsoperationen einer Transaktion das Statement *SelectCaches* ausgewertet wird.

Mit Hilfe dieser Informationsgewinnung kann die Methode *propagateWS()* im Pseudocode des Backend (siehe Anhang B.1) folgendermaßen realisiert werden: Zuerst wird mittels der Auswertung des Statement *SelectCaches* die Liste aller Caches erstellt, welche die betrachtete Datenänderung nachziehen müssen. Wurde die Änderungsoperation bereits in einem Cache ausgeführt, so wird dieser aus der Liste der Caches entfernt. Abschließend wird das Write Set der betrachteten Datenänderung an alle Caches dieser Liste gesendet.

Der Vorteil der soeben vorgestellten Realisierung liegt in der einfachen Identifizierung der Caches zur Weiterleitung der Write Sets. Das Nachladen von Datensätzen wird in dieser Realisierung nicht betrachtet, da das Backend nicht alle notwendigen Informationen besitzt, um die benötigten Datensätze für einen bestimmten Cache zu identifizieren. Zusätzlich ist der Zeitpunkt des Nachladens der Datensätze flexibel. Eine detaillierte Analyse möglicher Zeitpunkte zum Nachladen der Datensätze wird in Kapitel 5.5.5 betrachtet. Die Aufgabe zu überprüfen, ob ein zu ändernder Datensatz eine Daseinsberechtigung für den Cache erhält, kann in diesem Szenario nicht durch das Backend erfolgen.

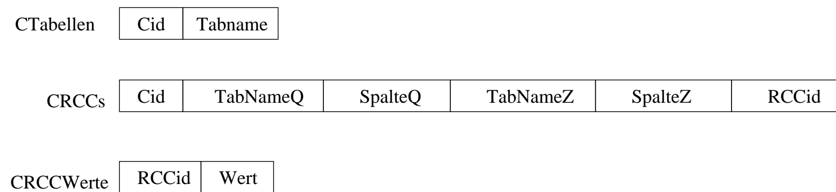


Abbildung 5.4: Datenstruktur zur Speicherung der Cache-Informationen (Volles Wissen im Backend)

Ist der Datensatz im Cache vorhanden, so muss der Cache entscheiden, ob die Datenänderung ausgeführt oder ob der Datensatz aus dem Cache entladen wird.

Backend hat volles Wissen

Werden im Backend die Informationen über die existierenden Cache Groups inklusive deren Inhalt gespeichert, so kann sehr detailliert festgestellt werden, welche Caches eine Datenänderung nachziehen müssen. Ein Vorschlag für eine Datenstruktur zur Speicherung dieser Daten wird in Abbildung 5.4 dargestellt. Diese Datenstruktur entspricht der Datenstruktur des eingeschränkten Wissensstands, wurde jedoch um die Speicherung der Werte der RCC-Quellspalten erweitert. Zusätzlich ist bei diesem Wissensstand der Aufbau eines Write Set von Interesse, da hier basierend auf den Werten die Caches sowie die Datensätze zum Nachladen in den Cache identifiziert werden können. In der folgenden Analyse wird davon ausgegangen, dass ein Write Set für ein *Update*-Statement folgenden Aufbau hat:

```
tabellenName, DS-Id_1 (Primärschlüssel),
    attributX oldValueX newValueX,
    attributY oldvalueY newValueY, ... ;
tabellenName, DS-Id_2 (Primärschlüssel),
    attributX oldValueX newValueX,
    attributY oldvalueY newValueY, ... ;
...
```

Ein Write Set für ein *Insert*- oder *Delete*-Statement stellen Spezialfälle des *Update*-Statement dar. So existieren bei dem Write Set des *Insert*-Statement lediglich die neuen Werte der Attribute, währenddessen beim *Delete*-Statement lediglich der Datensatz-Identifikator bekannt ist.

Die Identifizierung der von einer Änderungsoperation betroffenen Caches wird zunächst am Write Set des *Update*-Statement veranschaulicht, bevor auf die Spezialfälle *Insert* und *Delete* eingegangen wird. Aufgrund des Write Set kann für jeden einzelnen Eintrag geprüft werden, in welchem Cache die Änderung nachgezogen werden muss. So kann mit Hilfe der alten Attributwerte eines Eintrags folgendes Statement die Caches identifizieren, die diesen Datensatz enthalten:

```
select distinct(rcc.Cid) from CRCCs rcc, CRCCWerte values
where rcc.RCCid = values.RCCid
and (
```

```
    (TabNameQ = tabellenName and SpalteQ = attributX)
     or (TabNameZ = tabellenName and SpalteZ = attributX)
     and value.Wert = oldValueX
) and (
    (TabNameQ = tabellenName and SpalteQ = attributY)
     or (TabNameZ = tabellenName and SpalteZ = attributY)
     and value.Wert = oldValueY
);
```

Allerdings können durch dieses Statement nicht alle notwendigen Caches identifiziert werden, wenn lediglich Attributwerte geändert werden, die keinem RCC angehören. So könnte lediglich anhand des Tabellennamen, analog zum eingeschränkten Wissensstand, die Cache-Identifizierung korrekt erfolgen. Es besteht jedoch die Möglichkeit, in den Write Sets nicht nur die Informationen der geänderten Attributwerte zu speichern, sondern zusätzlich alle alten Attributwerte. So können mit Hilfe des zuvor vorgestellten Statement alle Caches identifiziert werden, welche die Datenänderung nachziehen müssen. Allerdings wird dadurch der Umfang des Write Set entsprechend größer. Werden nicht alle alten Attribute im Write Set gespeichert, kann beispielsweise für das Write Set eines *Delete*-Statements die Cache-Identifizierung nur anhand des Tabellennamens erfolgen. Werden hingegen alle Attribute im Write Set gespeichert, so kann basierend auf den Attributwerten die Identifizierung der Caches geschehen. Das Write Set eines *Insert*-Statement stellt bezüglich dieser Analyse einen Ausnahmefall dar, da hier nicht aufgrund der alten sondern mit den neuen Attributwerten die Identifizierung erfolgen muss. Hierbei kann das zuvor vorgestellte Statement verwendet werden, jedoch müssen die neuen, anstatt der alten Attributwerte eingefügt werden. In diesem Fall werden zudem nur diese Caches durch die Auswertung der Anfrage zurückgegeben, in denen die Daseinsberechtigung für den Cache gegeben ist. Für die Write Sets der *Update*-Statements muss diese Überprüfung separat im Cache erfolgen. Dort wird dann entsprechend entschieden, ob der Datensatz entladen wird oder die Datenänderung ausgeführt wird. Es gilt analog zum eingeschränkten Wissensgrad, dass die Caches mit Hilfe des vorgestellten Statement nur aufgrund einer einzelnen Änderungsoperation identifiziert werden. Sind alle Operationen einer Transaktion wichtig, beispielsweise zur Durchführung des 2PC-Protokolls, so müssen alle in der Transaktion enthaltenen Änderungsoperationen separat analysiert werden.

Zusätzlich zur Identifizierung der betroffenen Caches ist die Bestimmung der Datensätze von Interesse, die potentiell aufgrund der Änderungsoperation in den Cache nachgeladen werden müssen. Dies erfolgt in den folgenden zwei Schritten:

Schritt 1: Identifizierung der Tabellen, in die eventuell Datensätze nachgeladen werden müssen

Mit Hilfe des folgenden Statement können die Tabellen identifiziert werden, die durch ausgehende RCCs, der von der Änderungsoperation betroffenen Tabelle, referenziert werden.

```
select Cid, TabNameZ, SpalteZ from CRCCs
where TabNameQ = tabellenName and SpalteQ = attributX;
```

Dieses Statement gibt die Tabellen zurück, in die gegebenenfalls Daten aufgrund einer Wertänderung auf dem Attribut *attributX* nachgeladen werden müssen.

Schritt2: Datensätze zum Nachladen identifizieren

Um die Datensätze bestimmen zu können, ist auf der Tabelle eine entsprechende *Select*-Anfrage notwendig, z. B.:

```
select * from TabNameZ where SpalteZ = newValueX;
```

Hierbei werden allerdings noch nicht die Datensätze ausgeschlossen, die sich bereits im Cache befinden.

Diese Schritte wurden jedoch nur für lediglich eine Wertänderung auf einem Attribut vorgestellt. Diese Schritte müssen für alle Attribute wiederholt werden, deren Werte sich verändert haben. Zusätzlich müssen diese Schritte rekursiv auf allen ausgehenden RCCs ausgeführt werden, um letztendlich alle Datensätze zum Nachladen in den Cache bestimmen zu können.

In Anhang B.3.2 wird der Pseudocode für die Methode *propagateWS()* dargestellt. Dieser Pseudocode enthält die soeben vorgestellte Vorgehensweise zur Weiterleitung der Datenänderung sowie die Bestimmung der nachzuladenden Datensätze. Weiterhin wird die Optimierung umgesetzt, dass der Cache das Sondierungsergebnis mit der Änderungsoperation an das Backend schickt.

Der Vorteil des vollen Wissensgrads im Backend ist, dass bereits mit der Weiterleitung des Write Set die nachzuladenden Datensätze mitschickt werden und dadurch Nachrichten eingespart werden können. Dies hat allerdings ebenfalls zur Folge, dass im Fall eines Transaktionsabbruchs gegebenenfalls unnötig Datensätze in den Cache geladen wurden. Es besteht jedoch ebenfalls die Möglichkeit, die nachzuladenden Datensätze nicht mit den Write Sets zu verschicken und damit bezüglich des Nachladezeitpunktes flexibel zu bleiben. Diese Alternative wird in Kapitel 5.5.5 diskutiert.

Aufgrund der erneuten Analyse der Szenarien bezüglich des Wissensstands des Backend kann festgestellt werden, dass der *eingeschränkte Wissensgrad* und der *volle Wissensstand* jeweils eigene Vorteile haben. Bezüglich der Realisierung im Synchronisierungsverfahrens wird zunächst der eingeschränkte Wissensstand verwendet, so dass der Umfang der Informationen im Write Set frei wählbar bleibt. Dies kann jedoch jederzeit auf den vollen Wissensstand erweitert werden.

5.5.2 Konflikterkennung und Konfliktbehandlung

In diesem Abschnitt wird im Detail die Konflikterkennung und die Konfliktbehandlung diskutiert, die bereits im Basisalgorithmus angesprochen wurden. Damit auftretende Konflikte anhand konkreter Beispiele leichter verdeutlicht werden können, werden zunächst folgende Bezeichnungen definiert:

- $w_1(a)$ beschreibt eine Änderungsoperation auf dem Datenelement a der Transaktion T1, $r_1(a)$ analog eine Leseoperation.
- $WS_1(a)$ entspricht dem Write Set, das aufgrund der Änderung $w_1(a)$ erzeugt wird.

- $wL_1(a)$ repräsentiert das Halten einer Schreibsperre im Datenbanksystem auf dem Datenelement a durch die Transaktion $T1$. $rL_1(a)$ repräsentiert analog zur Schreibsperre eine gehaltene Lesesperre.
- $WS_1^*(a)$ bedeutet, dass das Write Set einer Änderungsoperation auf dem Datenelement a der Transaktion T1 bereits in der Datenbank erfolgreich nachgezogen wurde.
- $WS_1^+(a)$: $WS_1(a)$ ist aufgrund einer vergebenen Sperre blockiert.

Da auf einem Knoten (Backend oder Cache) Operationen lokaler Transaktionen und Nachziehtransaktionen ausgeführt werden, können folgende Konfliktarten auftreten:

- Konflikte zwischen Transaktionsoperationen im Datenbanksystem
- Konflikte zwischen Write Sets verschiedener Transaktionen im Cache-System

Anhand der folgenden Beispiele wird verdeutlicht, wie diese Konflikte erkannt und aufgelöst werden können.

Konflikte zwischen Transaktionsoperationen

Im Backend wird die Transaktion

$$\text{T1: } b_1, w_1(a), c_1$$

und im Cache die Transaktion

$$\text{T2: } b_2, w_2(a), c_2$$

ausgeführt. Allerdings kann die Änderungsoperation der Transaktion T2 nicht im Cache bearbeitet werden, so dass diese an das Backend weitergeleitet wird. In diesem Fall wird $w_1(a)$ im Backend ausgeführt sowie $w_2(a)$. Diese Datenänderungen werden vom Cache-System an das Datenbanksystem weitergeleitet und dort verarbeitet. In diesem Fall kann nur eine Transaktion ihre Datenänderung erfolgreich ausführen und die andere Transaktion wird aufgrund vergebener Datensperren blockiert. So wird im Datenbanksystem der Write/Write-Konflikt erkannt und die wartende Transaktion wird abgebrochen. Analog können auch Read/Write-Konflikte im Datenbanksystem erkannt und aufgelöst werden.

Konflikte zwischen Write Sets

Im Backend wird die Transaktion

$$\text{T1: } b_1, w_1(a), c_1$$

und im Cache die Transaktion

$$\text{T2: } b_2, w_2(a), c_2$$

ausgeführt. Für dieses Beispiel werden die beiden Schreiboperationen auf dem Datenelement a gleichzeitig im Cache und im Backend durchgeführt und versendet. Dies hat zur Folge, dass $w_1(a)$ bereits im Backend ausgeführt wurde,

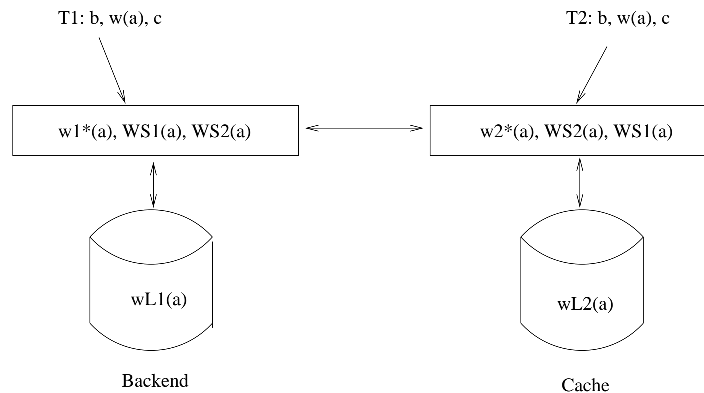


Abbildung 5.5: Beispiel zur Veranschaulichung der Konflikterkennung

bevor das Write Set $WS_2(a)$ der Transaktion 2 im Backend eintrifft. Entsprechend wurde im Cache bereits $w_2(a)$ ausgeführt, bevor das Write Set $WS_1(a)$ der Transaktion 1 den Cache erreicht. Diese Situation wird in Abbildung 5.5 veranschaulicht und es kann nun wie folgt reagiert werden:

- Es wird zunächst angenommen, dass das Cache-System keine Write/Write-Konflikte erkennt.

  Im Backend wird versucht, das Write Set $WS_2(a)$ in der Datenbank nachzuziehen. Allerdings hält Transaktion T1 bereits eine Sperre für das Datenelement a, so dass hier ein Write/Write-Konflikt im Datenbanksystem vorliegt. Analog erfolgt die Ausführung des Write Set $WS_1(a)$ im Cache und es existiert dort ebenfalls ein Write/Write-Konflikt im Datenbanksystem. Bezüglich des weiteren Ablaufs dieser beiden Transaktionen liegt folgende Situation vor: Da der Benutzer der Cache-Transaktion T2 erst nach der Ausführung der Änderungsoperation im Backend informiert wird, muss T2 warten, bis die Datenänderung im Backend bearbeitet wurde. Allerdings kann T1 nicht erfolgreich beendet werden, solange das Write Set $WS_1(a)$ nicht im Cache nachgezogen wurde. Allerdings ist im Cache für dieses Datenelement bereits eine Datensperre vergeben. Transaktion T2 kann ebenfalls nicht weiter ausgeführt werden, da im Backend die Datensperre für das betroffene Datenobjekt bereits vergeben ist. Somit liegen, lokal gesehen, folgende Abhängigkeiten vor:

  $$\text{Backend: } T1 \leftarrow T2 \text{ Cache: } T1 \rightarrow T2$$

  Allerdings führt dies global gesehen zu einem Deadlock zwischen diesen beiden Transaktionen.

- Wird hingegen bereits im Cache-System festgestellt, ob Write/Write-Konflikte zwischen Write Sets vorliegen, resultiert folgendes Vorgehen in dieser Situation:

  Aufgrund der Voraussetzung existieren im Backend die Write Sets $WS_1^*(a)$, $WS_2(a)$ und im Cache $WS_2^*(a), WS_1(a)$. Somit kann das Cache-System

bereits vor weiteren Operationen einer Transaktionen und vor dem Nachziehen der Änderungsoperation in der Datenbank den Write/Write-Konflikt erkennen. In diesem Fall kann das Cache-System entscheiden, welche der beiden Transaktionen abgebrochen werden soll. Aufgrund festgelegter Regeln bezüglich der Auflösung solcher Konflikte, die im nächsten Abschnitt beschrieben werden, kann in allen Knoten die gleiche Entscheidung getroffen werden und die Wahrscheinlichkeit für das Auftreten von Deadlocks verringert werden.

Darüber hinaus ist das Erkennen von Write/Write-Konflikten durch das Cache-System dann relevant, wenn das Datenbanksystem z. B. nur Snapshot Isolation garantieren kann. In diesem Fall kann nicht anhand von Datensperren erkannt werden, ob Write/Write-Konflikte auftreten, da bei jeder Datenänderung neue Objektversionen erzeugt werden.

Behandlung von Konflikten zwischen Write Sets

Nachdem die Vorteile der frühen Konflikterkennung durch das Cache-System vorgestellt wurden, werden im Folgenden Handlungsalternativen bezüglich der Konfliktauflösung analysiert, die aufgrund dieser Konflikterkennung resultieren können.

Sobald im Cache-System Write/Write-Konflikte entdeckt werden, kann dort die Entscheidung getroffen werden, welche Konflikt-Transaktion abgebrochen werden soll. Diese Entscheidung ist allerdings nicht sehr einfach, da hier die folgenden Aspekte betrachtet werden müssen. Es spielt beispielsweise die Reihenfolge, in der Änderungsoperationen auf den Knoten ausgeführt werden, eine wichtige Rolle. Weiterhin kann von Interesse sein, auf wie vielen Knoten eine Änderungsoperation bereits ausgeführt wurde. Diese Anzahl kann abgeschätzt werden, je nach dem wo der Ursprung der Änderungsoperation lag und ob sie bereits im Backend ausgeführt wurde. Wurde die Änderungsoperation bereits im Backend durchgeführt, so ist bereits bekannt, dass dort bisher keine Konflikte zu anderen Transaktionen auftraten. Wird die Änderungsoperation auf einem Cache nachgezogen und existieren dort Konflikte zwischen dieser und einer anderen Operation, so können die folgenden Schlußfolgerungen gezogen werden:

- Es existiert ein Read/Write-Konflikt und die Änderungsoperation kann erst dann nachgezogen werden, wenn die lokale Transaktion vollständig durchgeführt wurde.

- Existiert hingegen ein Write/Write-Konflikt, so gibt es eine lokale Transaktion, die ebenfalls die gleichen Datenelemente ändert. Diese lokale Änderungsoperation wurde allerdings noch nicht im Backend ausgeführt, da sonst der Write/Write-Konflikt bereits dort erkannt worden wäre.

  Es besteht jedoch die Möglichkeit, dass der gleiche Write/Write-Konflikt zur gleichen Zeit im Backend erkannt wird. Mit Hilfe festgelegter Regeln kann sichergestellt werden, dass in diesem Fall in beiden Knoten die gleiche Entscheidung bezüglich des Abbruchs der Konflikt-Transaktionen getroffen wird.

Eine Regel bezüglich des Abbruchs der Konflikt-Transaktionen resultiert aus dem folgenden Szenario, in dem eine Änderungsoperation ihren Ursprung im

Cache hat. Wurde eine Änderungsoperation bereits im Cache erfolgreich ausgeführt und im Backend nachgezogen, so wird die Änderungsoperation an die noch zu aktualisierenden Caches propagiert. Tritt jetzt allerdings auf einem zu aktualisierenden Cache ein Konflikt mit einer lokalen Transaktion auf, so ist es sinnvoller die lokale Transaktion abzubrechen, da die Nachziehtransaktion potentiell schon auf mehr Knoten ausgeführt wurde als die lokale Transaktion. Aus diesem Grund kann man beispielsweise folgende Regel zur Konfliktauflösung definieren:

Sei T1 eine lokale Transaktion und sei T2 eine Nachziehtransaktion auf einem Cache und zwischen diesen beiden Transaktionen existiert ein Write/Write-Konflikt. Unabhängig von der Ausführungsreihenfolge der Änderungsoperationen wird immer die lokale Transaktion und nicht die Nachziehtransaktion abgebrochen.

Aufgrund der Regel, dass Nachziehtransaktionen den lokalen Transaktionen vorgezogen werden, könnte man versuchen für das ROpWAn-Verfahren folgende Art der Verarbeitung vorzunehmen: Sobald eine Änderungsoperation im Ursprungs-Cache und im Backend erfolgreich ausgeführt wurde, kann die Transaktion das Commit ausführen, ohne dass die Datenänderung bereits auf den anderen Caches nachgezogen wurde. Es muss dann allerdings garantiert werden, dass die Datenänderung ebenfalls auf allen restlichen Caches erfolgreich nachgezogen wird, da sie bereits erfolgreich abgeschlossen wurde. Durch die Gewährung dieser Garantie entsteht jedoch folgendes Problem: Die Datenelemente spiegeln nicht auf allen Knoten den gleichen Datenzustand wider und somit besteht die Möglichkeit, dass Transaktionen einen inkonsistenten Schnappschuss lesen. Dieses Szenario wird in Abbildung 5.6 veranschaulicht. Im Cache-System werden alle bisher ausgeführten sowie die wartenden Operationen dargestellt. Transaktion T1 wurde auf dem Backend und im Cache 1 bereits erfolgreich beendet, bevor alle restlichen Caches die Nachziehtransaktion ausgeführt haben und somit ist in Cache 2 das Datenelement a noch veraltet. Allerdings läuft in diesem Szenario die Transaktion T2, noch vor der Änderungsoperation $WS_1(a)$ ab. Im Cache 2 werden in dieser Situation die Datenelemente a und c gelesen. Allerdings existiert das Datenelement b nicht im Cache, so dass diese Leseoperation im Backend ausgeführt wird und T2 dort b' liest. Transaktion T2 liest eine inkonsistenten Schnappschuss, da sie den Datenzustand a, b', c sieht. Ein konsistenter Schnappschuss wird entweder durch die Datenzustände a, b, c oder a', b', c repräsentiert. Um das Lesen inkonsistenter Schnappschüsse zu verhindern, muss sichergestellt werden, dass auf alte Datenversionen zugegriffen werden kann. In einem solchen Fall muss das verwendete Datenbanksystem mehrere Versionen der Datenelemente bereitstellen. Da dieser Ansatz allerdings, wie auch bereits der in Kapitel 4.5.2 vorgestellte SRCA-Algorithmus [LKPnMJP05], den Einsatzbereich des ROpWAn-Verfahrens eingrenzt, wird das synchrone Abschließen einer Transaktion auf allen Knoten im ROpWAn-Verfahren eingesetzt.

Eine weitere nahe liegende Regel kann bezüglich der Konfliktauflösung im Backend aufgestellt werden.

Wird im Backend ein Write/Write-Konflikt aufgrund von Write Sets erkannt, so wird immer die Transaktion abgebrochen, deren Write Sets noch nicht in der Datenbank ausgeführt wurde.

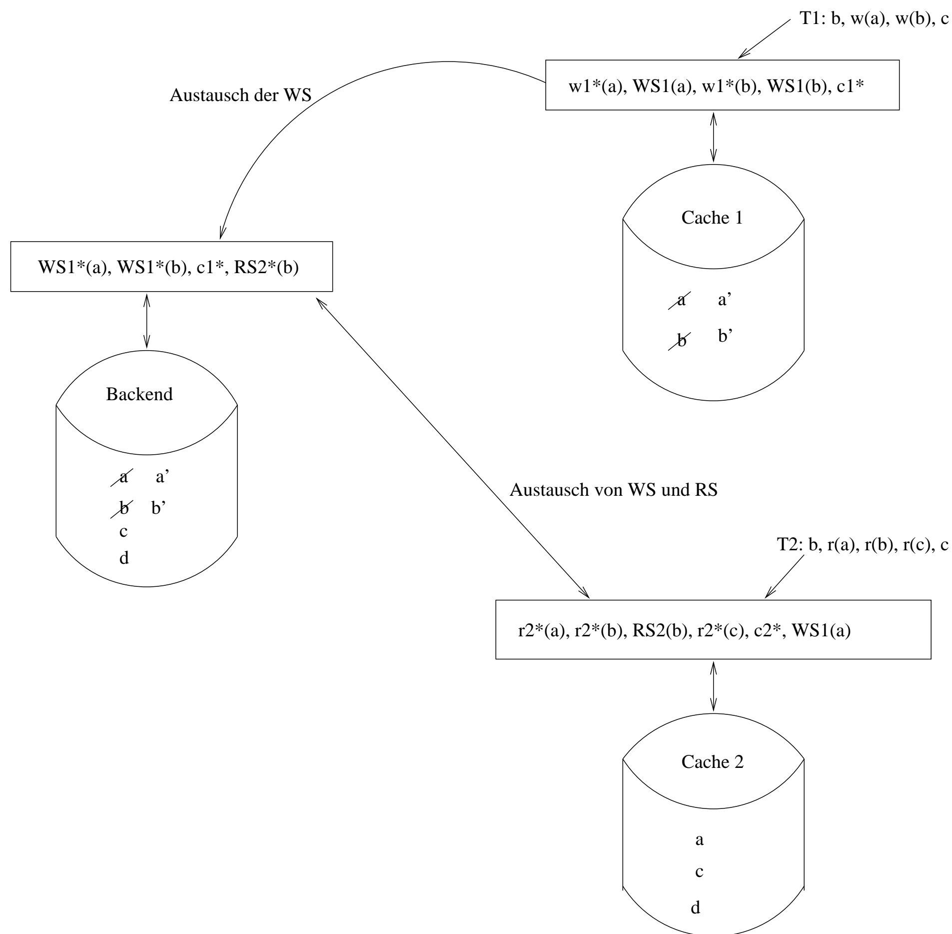


Abbildung 5.6: Beispielszenario für das Lesen eines inkonsistenten Datenbank-Schnappschuss

Dies bedeutet, das hier das FIFO-Prinzip zum Tragen kommt und immer die bereits durchgeführte Transaktionsoperation gewinnt.

5.5.3 Deadlocks

In Kapitel 5.5.2, in dem die Konflikte zwischen Write Sets im Mittelpunkt standen, wurde bereits auf das Auftreten von Deadlocks hingewiesen. Die Auftrittswahrscheinlichkeit kann zwar durch die Erkennung von Write/Write-Konflikten durch das Cache-System verringert werden (vgl. Kapitel 5.5.2), dennoch sind Deadlocks möglich. In diesem Abschnitt werden diverse Szenarien vorgestellt, in denen Deadlocks zwischen Transaktionen auftreten. Nach der Beschreibung der Deadlock-Situation werden Möglichkeiten zur Erkennung und Auflösung des Deadlock beschrieben, sofern diese existieren.

Abbildung 5.7: Deadlock-Szenario 1

Deadlock-Szenario 1

Abbildung 5.7 veranschaulicht ein Szenario, in dem ein Deadlock zwischen den folgenden zwei Transaktionen auftritt:

$$\text{Backend: T1 } b_1, w_1(a), w_1(b), c_1$$

$$\text{Cache: T2 } b_2, w_2(b), w_2(a), c_2$$

Wurde eine Änderungsoperation erfolgreich in der Datenbank ausgeführt, so wird ihr Write Set erstellt und an die anderen Knoten weitergeleitet. Die folgenden Operationsabfolgen sind auf dem Cache-System beobachtbar:

$$\text{Backend: } w_1^*(a), WS_1(a), WS_2^*(b), w_1^+(b)$$

$$\text{Cache: } w_2^*(b), WS_2(b), WS_1^*(a), w_2^+(a)$$

Anhand dieser Abfolgen können folgende lokale Abhängigkeiten zwischen den beiden Transaktionen T1 und T2 festgestellt werden:

$$\text{Backend: } T1 \rightarrow T2 \text{ und im Cache: } T1 \leftarrow T2$$

Diese Abhängigkeiten erzeugen lokal noch keinen Deadlock, allerdings existiert global gesehen ein Deadlock zwischen diesen beiden Transaktionen. Das Hauptproblem bei der Erkennung dieses Deadlocks und dessen Auflösung ist, dass er weder im Backend noch im Cache erkannt werden kann. Das Cache-System kann die gegenseitigen Abhängigkeiten ebenfalls nicht erkennen, da hier lediglich Konflikte zwischen einzelner Write Sets erkannt werden können.

Eine Idee zur Lösung dieses Problems liegt darin, den Deadlock im Backend sichtbar zu machen. Das Backend ist hierfür vor allem deshalb geeignet, da dort alle Änderungsoperationen ausgeführt werden müssen. Darf eine Änderungsoperation im Cache durchgeführt werden, so besteht die Möglichkeit vor deren Ausführung die Datenänderung an das Backend weiterzuleiten, so dass der

Deadlock im Backend erkannt werden kann. Somit resultieren für das Szenario aus Abbildung 5.7 folgende Operationsabläufe:

$$\text{Backend: } w_1^*(a), WS_1(a), w_2^*(b), w_1^+(b), w_2^+(a)$$

$$\text{Cache: } w_2^*(b), WS_2(b), WS_1^*(a), w_2^+(a)$$

Aufgrund dieser Ausführungsreihenfolge im Backend kann dort der Deadlock durch das Datenbanksystem erkannt und aufgelöst werden.

Deadlock-Szenario 2

Analog zum Deadlock-Szenario 1 kann ein weiteres Szenario erzeugt werden. Folgende Transaktionen spielen hierbei eine Rolle:

$$\text{Backend: T1 } b_1, r_1(a), w_1(b), c_1$$

$$\text{Cache: T2 } b_2, r_2(b), w_2(a), c_2$$

Wird eine Vorgehensweise analog zu der in Abbildung 5.7 angenommen, so existieren folgende Operationsabfolgen im Cache-System:

$$\text{Backend: } r_1^*(a), w_1^*(b), WS_1(b), WS_2^+(a)$$

$$\text{Cache: } r_2^*(b), w_2^*(a), WS_2(a), WS_1^+(b)$$

Aufgrund dieser Abfolgen existieren erneut folgende lokale Abhängigkeiten

$$\text{Backend: } T1 \leftarrow T2 \text{ und im Cache: } T1 \rightarrow T2$$

und diese führen zu einem globalen Deadlock. Können Leseoperationen auf einem Knoten ausgewertet werden, so müssen diese nicht zwischen den Knoten ausgetauscht werden. Die Realisierung dieser Eigenschaft verhindert in diesem Szenario, dass der Deadlock im Cache-System oder im Datenbanksystem erkannt werden kann.

Dieser Deadlock könnte erkannt werden, wenn im lokalen Cache-System Read Sets der Leseoperationen gespeichert werden. So könnte das Cache-System bereits vor der Ausführung im Datenbanksystem Read/Write-Konflikte zwischen Read Sets und Write Sets erkennen. Das Problem dieser Lösung stellt die Erzeugung des Read Set dar, da dies nicht analog zum Write Set erstellt werden kann. Würde man das Anfrageergebnis der Leseoperation als Read Set speichern, so hilft dieses leider bezüglich der Deadlock-Auflösung nicht weiter, da vom Anfrageergebnis nicht zwingend auf die einzelnen Datenelemente geschlossen werden kann. Dies tritt beispielsweise bei einem Join auf. Da kein Read Set aufgrund einer Leseoperation erzeugt werden kann, kann der Deadlock aus diesem Szenario nicht erkannt werden und die beiden Transaktionen werden beispielsweise aufgrund eines Timeout abgebrochen.

Deadlock-Szenario 3

Werden die Transaktionsoperationen des Deadlock-Szenario 2 in einer anderen Reihenfolge angeordnet, so entsteht eine neue Deadlock-Situation, die nicht mit

den bereits vorgestellten Möglichkeiten aufgelöst werden kann. Folgende Transaktionsabfolgen liegen vor:

$$\text{Backend: T1 } b_1, w_1(a), r_1(b), c_1$$

$$\text{Cache: T2 } b_2, w_2(b), r_2(a), c_2$$

Daraus resultieren folgende Operationsabfolgen im Cache-System:

$$\text{Backend: } w_1^*(a), WS_1(a), WS_2^*(b), r_1^+(b)$$

$$\text{Cache: } w_2^*(b), WS_2(b), WS_1^*(a), r_2^+(a)$$

Aus diesen Ausführungsreihenfolgen resultieren folgende lokale Abhängigkeiten zwischen den beiden Transaktionen:

$$\text{Backend: } T1 \rightarrow T2 \text{ und im Cache: } T1 \leftarrow T2$$

Diese lokalen Abhängigkeiten führen zu einem globalen Deadlock, der in keinem der beiden Knoten erkannt werden kann, da bei diesen Leseoperationen noch nicht bekannt ist, welche Datenelemente ausgewertet werden sollen. Darum könnte dieser Deadlock auch nicht mit Hilfe von Read Sets erkannt werden, sofern die Erzeugung der Read Sets möglich wäre. Somit kann der hier vorgestellte Deadlock nicht erkannt werden. Erst beispielsweise nach einem *Timeout* wird dieser Deadlock aufgelöst, indem beide Transaktionen abgebrochen werden.

In diesem Abschnitt wurden drei Szenarien vorgestellt, in denen jeweils ein Deadlock zwischen Transaktionen auftritt. Leider konnte nur der Deadlock aus dem ersten Szenario folgendermaßen aufgelöst werden: Änderungsoperationen, die im Cache ausgeführt werden können, werden noch vor deren Abarbeitung im Cache an das Backend weitergeleitet. So können Deadlock-Situationen zwischen Änderungsoperationen im Backend sichtbar gemacht werden. Leider können nicht alle Deadlock-Szenarien verhindert werden können. Allerdings kann zumindest die Wahrscheinlichkeit für das Auftreten von Deadlocks verringert werden.

In Kapitel 4.3.2 wurde bereits darauf hingewiesen, dass ebenfalls der Isolationsgrad Snapshot Isolation für das Synchronisierungsverfahren interessant ist. Da dieser Ansatz zwar zurückgestellt, aber nicht verworfen wurde, werden die Deadlock-Szenarien bezüglich der Verwendung mehrerer Datenversionen analysiert. Es können folgende Schlußfolgerungen gezogen werden:

- Deadlock-Szenario 1 kann nicht auftreten, da jeweils lokal neue Objektversionen durch die Transaktionen erzeugt werden. Daraus resultiert jedoch das Problem, dass im Backend und im Cache jeweils unterschiedliche Reihenfolgen der Objektversionen erzeugt werden. Hierbei muss beispielsweise bei Commit sichergestellt werden, dass eine global eindeutige Reihenfolge der Objektversionen festgelegt wird.

- Deadlock-Szenario 2 und 3 können nicht auftreten, da die Leseoperationen auf einen Datenbank-Schnappschuss zugreifen und somit nicht blockiert werden.

5.5.4 Anomalien

In diesem Abschnitt wird im Detail analysiert, ob unter der Verwendung des Basisalgorithmus Anomalien auftreten können. Hierzu werden die in Anhang A definierten Anomalien nacheinander betrachtet. Hierbei wird geprüft, ob die Anomalie unter der Verwendung von R/X-Sperrverfahren oder unter der Garantie der Snapshot Isolation auftreten kann.

Dirty Read

Damit die Anomalie *Dirty Read* zwischen zwei Transaktionen auftreten kann, ist beispielsweise folgende Historie notwendig:

$$H = b_1, b_2, w_1(a), r_2(a), a_1, c_2$$

Wird diese Historie auf einem Knoten ausgeführt, so verhindern lokale Synchronisierungsverfahren (z. B. R/X-Sperrprotokolle oder Mehrversionenverfahren) das Auftreten eines Dirty Read. Somit muss im Folgenden überprüft werden, ob ein Dirty Read bei einer verteilter Ausführung auftreten kann. Wird beispielsweise Transaktion T1 im Backend und Transaktion T2 im Cache durchgeführt, so wird die Änderungsoperation von Transaktion T1 an den Cache weitergeleitet, um dort die Datenänderung nachzuziehen. Dies ist notwendig, da das Datenelement a aufgrund der auswertbaren Leseoperation im Cache vorhanden ist. Ein Dirty Read könnte dann auftreten, wenn T2 den Datensatz liest, nachdem die Änderungsoperation von T1 im Cache durchgeführt wurde, jedoch noch nicht durch Commit beendet wurde. Unter der Annahme des R/X-Sperrverfahrens verhindern die Sperren auf den Datenelementen das Auftreten des Dirty Read. Ebenfalls unter der Garantie der Snapshot Isolation kann kein Dirty Read auftreten, da die Leseoperation auf Daten eines konsistenten Datenbank-Schnappschusses zugreift.

Dirty Write

Die Anomalie *Dirty Write* tritt auf, sofern zwei Transaktionen das gleiche Datenelement ändern. Folgende Historie beschreibt diese Situation.

$$H = b_1, b_2, w_1(a), w_2(a), c_1, c_2$$

Die Durchführung dieser Historie auf einem Knoten wird durch das Halten von Sperren bei R/X-Sperrverfahren verhindert. Für die Betrachtung im verteilten Fall wird angenommen, dass Transaktion T1 im Backend und Transaktion T2 im Cache ihren Ursprung hat. Basierend auf diesem Szenario können folgende Fälle unterschieden:

- $w_1(a)$ kann aufgrund fehlender Daten nicht im Cache verarbeitet werden: In diesem Fall wird die Änderungsoperation an das Backend weitergeleitet und dort wird der Konflikt dieser beiden Schreiboperationen erkannt. Somit kann dieser Fall analog zu der Ausführung auf einem Knoten betrachtet werden und das Dirty Write kann nicht auftreten.

- $w_1(a)$ kann im Cache ausgeführt werden: In diesem Fall werden beispielsweise beide Änderungsoperationen lokal verarbeitet und das Write Set untereinander ausgetauscht. Dieses Szenario wurde bereits in Kapitel 5.5.2

betrachtet. Dieser Konflikt kann bereits im Cache-System erkannt werden und behandelt werden. Auf dieser Grundlage kann kein Dirty Write auftreten.

Wird hingegen die Garantie der Snapshot Isolation unterstellt, so muss durch das Cache-System sichergestellt werden, dass Write/Write-Konflikte durch das Cache-System erkannt werden.

Non-repeatable Read

Ein *Non-repeatable Read* könnte auftreten, sofern z. B. folgende Transaktionsabfolge auf einem Knoten ausgeführt wird:

$$H = b_1, b_2, r_1(a), w_2(a), c_2, r_1(a), c_1$$

Ein Non-repeatable Read ist allerdings unter der Verwendung von R/X-Sperrverfahren nicht möglich, da $w_2(a)$ aufgrund einer Lesesperre auf dem Datenelement a durch Transaktion T1 solange blockiert wird, bis T1 entweder erfolgreich beendet oder abgebrochen wird. Dies gilt für die Fälle, dass beide Transaktionen auf einem oder auf mehreren Knoten ausgeführt werden. Wird hingegen die Garantie der Snapshot Isolation unterstellt, so liest Transaktion T1 in beiden Leseoperationen die gleichen Daten des Schnappschusses, der bei Beginn der Transaktion T1 gültig war und ein Non-repeatable Read kann nicht auftreten.

Lost Update

Zur Erzeugung eines *Lost Update* könnte folgende Historie führen:

$$H = b_1, b_2, r_1(a), w_2(a), w_1(a), c_1, c_2$$

Unter der Annahme der Verwendung des R/X-Sperrverfahrens wird das Lost Update, genauso wie das Dirty Wirte, einerseits aufgrund der Datensperren verhindert und andererseits durch die Erkennung von Write/Write-Konflikten im Cache-System. Wird die Garantie der Snapshot Isolation unterstellt, so muss, analog zum Dirty Write, durch das Cache-System garantiert werden, dass Write/Write-Konflikte erkannt werden.

Phantom

Damit das Phantom-Problem auftreten kann, müsste folgende Historie auf einem Knoten ausgeführt werden:

$$H = b_1, b_2, r_1(S), w_2(S_i), c_2, r_1(S), c_1$$

In dieser Abfolge repräsentiert S eine Menge von Datensätzen, die ein festgelegtes Suchkriterium erfüllen. S_i beschreibt einen einzelnen Datensatz, der ebenfalls dem festgelegten Suchkriterium genügt. Unter der Annahme der Verwendung von R/X-Sperrverfahren kann das Phantom-Problem auftreten, sofern durch $w_2(S_i)$ ein neuer Datensatz erzeugt wird. In diesem Fall wird der neue Datensatz durch die zweite Leseoperation von T1 gelesen, da hier keine gehaltene Sperre dies verhindert. Soll hingegen durch $w_2(S_i)$ ein Datensatz gelöscht

werden, so kann dies erst dann ausgeführt werden, sobald Transaktion T1 fertig ist, da hier eine Lesesperre durch Transaktion T1 gehalten wird. Unter der Verwendung von R/X-Sperrverfahren tritt somit das Phantom-Problem auf. Dies kann verhindert werden, wenn hierarchische Sperren verwendet werden, so dass beispielsweise von einer Transaktionsoperation betroffene Tabellen gesperrt werden. Unter der Garantie der Snapshot Isolation besteht laut [BBG+95] die Möglichkeit, dass das Phantom-Problem auftritt, jedoch ist dies nicht immer der Fall.

Read Skew

Folgende Historie beschreibt die Anomalie Read Skew, sofern zwischen den Datenelementen a und b eine Bedingung definiert ist.

$$H = b_1, b_2, r_1(a), w_2(a), w_2(b), c_2, r_1(b), c_1$$

Wird diese Historie ausgeführt, so liest Transaktion T1 einen inkonsistenten Datenbankzustand. Dieser resultiert daraus, dass Transaktion T2 erfolgreich beendet wurde und damit Transaktion T1 einen Datenzustand liest, bei dem die Bedingung zwischen a und b nicht mehr erfüllt ist. Soll diese Historie auf einem Knoten unter der Verwendung von R/X-Sperrverfahren ausgeführt werden, so verhindern die Sperren auf den Datenelementen das Auftreten der Anomalie Read Skew. Im verteilten Fall wird beispielsweise Transaktion T1 im Backend und Transaktion T2 im Cache ausgeführt. In diesem Fall werden die Änderungsoperationen von T2 an das Backend weitergeleitet werden, so die Anomalie Read Skew aufgrund der Datensperren dort verhindert werden kann. Unter der Annahme der Snapshot Isolation kann die Anomalie Read Skew ebenfalls nicht auftreten, da die Leseoperationen der Transaktion T1 auf einen Schnappschuss zugreifen, der einen konsistenten Datenbank-Zustand widerspiegelt.

Write Skew

Die Anomalie Write Skew kann beispielsweise mit Hilfe der folgenden Historie erzeugt werden:

$$H = b_1, b_2, r_1(a), r_1(b), r_2(a), r_2(b), w_1(a), w_2(b), c_1, c_1$$

Weiterhin ist notwendig, dass eine Bedingung zwischen den Datenelementen a und b definiert ist, z. B. $a + b \leq 200$ unter der Annahme, dass a, b positive Zahlen repräsentieren. Unter der Verwendung des R/X-Sperrverfahrens kann die Anomalie des Write Skew nicht auftreten, sofern beide Transaktionen lokal auf einem Knoten ausgeführt wurden, da dies durch das Halten der entsprechenden Sperren auf den Datenelementen verhindert wird. Werden die Transaktionen jeweils auf unterschiedlichen Knoten ausgeführt, so resultiert folgendes Szenario: Auf dem Knoten $K1$ wird die Transaktion $T1$ durchgeführt und Transaktion $T2$ auf Knoten $K2$. Es wird angenommen, dass die Leseoperationen jeweils lokal ausgeführt werden und damit ebenfalls die Schreiboperation der entsprechenden Transaktion. Im nächsten Schritt, muss die Schreiboperation an alle notwendigen Replikate weitergeleitet werden, so dass das Write Set für die Operation $w_1(a)$ an den Knoten $K2$ und das Write Set von $w_2(b)$ an Knoten $K1$ propagiert wird. Aufgrund der bereits vergebenen Lesesperren wird das Nachziehen

der Write Sets blockiert und somit ebenfalls das später zu realisierende Commit der Transaktionen. In diesem Szenario existiert, anstatt eines Write Skew, ein Deadlock zwischen diesen beiden Transaktionen.

Wird hingegen Snapshot Isolation garantiert, so kann ein Write Skew auftreten, da das Cache-System keinen Write/Write-Konflikt zwischen diesen beiden Transaktionen erkennen kann. So können beide Transaktionen ihr Commit realisieren, da transaktionsweise keine Verletzung der Bedingung bezüglich der beiden Datenelemente a und b vorliegt. Allerdings kann in Folge der Realisierung der beiden Datenänderungen dennoch die Bedingung nicht erfüllt sein. Dies wird anhand des folgenden Beispiels gezeigt. Die Bedingung zwischen a und b lautet $a+b \leq 200$ und die Ausgangswerte sind $a = 100$ und $b = 50$. Transaktion T1 ändert das Datenelement a auf den Wert 120, während Transaktion T2 das Datenelement b auf den Wert 100 erhöht. Diese beiden Transaktionen können unter der Garantie Snapshot Isolation erfolgreich beendet werden, da keine Verletzung der Bedingung zwischen a und b vorliegt. Dies gilt, da bei dem Commit der Transaktion T1 $a + b = 120 + 50 = 170 \leq 200$ gilt und für Transaktion T2: $a + b = 100 + 100 = 200 \leq 200$. Allerdings ist nach dem Commit dieser beiden Transaktionen die Bedingung: $a+b = 120+100 = 220 \leq 200$ nicht mehr erfüllt. Damit das Problem des Write Skew verhindert werden kann, werden in [FLO+05] folgende Lösungsansätze vorgeschlagen:

- Es wird ein *Select-For-Update-Statement* statt der Leseoperation ausgeführt. Dadurch erscheint es, als ob beide Datenelemente verändert werden, obwohl die Datenelemente nur gelesen werden. Allerdings kann auf diese Weise der Konflikt erkannt werden.

- Jede Bedingung zwischen Datenelementen wird in einem neuen Datenelement Z materialisiert. Dann muss jede Datenänderung auf a oder b ebenfalls das Datenelement Z aktuell halten, so dass Konflikte zwischen diesen Transaktionen erkannt werden können.

Bei diesen Ansätzen sind allerdings Anpassungen in der Operationsausführung notwendig.

Aufgrund dieser Analyse kann zusammengefasst werden, dass unter der Annahme, dass zumindest eine lokale Synchronisierung durch R/X-Sperren realisiert wird, lediglich das Phantom-Problem auftreten kann. Dies kann jedoch verhindert werden, indem beispielsweise hierarchische Sperrprotokolle verwendet werden.

5.5.5 Nachladen von Datensätzen

In den bisherigen Erläuterungen wurde noch nicht betrachtet, welcher Mehraufwand durch das Nachladen von Datensätzen entsteht. Hierbei ist der Zeitpunkt des Nachladeprozess von Interesse, sowie die Menge der nachzuladenden Datensätze.

Zeitpunkte für das Nachladen von Datensätzen

Zur Realisierung des Nachladens stehen folgende Optionen zur Verfügung:

- Die Datensätze werden direkt mit dem Write Set in den Cache nachgeladen.

  In Kapitel 5.5.1 wurde dieser Ansatz lediglich für den vollen Wissensstand vorgestellt, da nur hier alle notwendigen Daten zur Auswertung der Informationen im Backend vorhanden sind. Mit Hilfe der direkten Weiterleitung der nachzuladenden Datensätze kann garantiert werden, dass die RCCs im Cache auch während einer Datenänderung gültig sind. Allerdings besteht beim Abbruch einer Transaktion die Gefahr, dass Datensätze unnötig in den Cache geladen wurden.

- Die Datensätze werden synchron in den Cache nachgeladen. Dies bedeutet, dass entweder während der Transaktionsausführung oder spätestens zum Zeitpunkt des Commit die Datensätze in den Cache geladen wurden.

  Bei diesem Szenario kann die Gültigkeit der RCCs im Cache nach einer Änderungsoperationen erhalten werden, da bei *Commit* einer Transaktion alle notwendigen Datensätze im Cache nachgeladen wurden. So gelten die RCCs in den Cache Groups jederzeit und es sind keine Anpassungen beispielsweise bezüglich der Operationsausführung im Cache notwendig. Allerdings wäre es sehr wünschenswert, die Daten verzögert nachzuladen.

- Die Datensätze werden asynchron in den Cache geladen.

  Dieses Szenario hat den Vorteil, dass bei einem Transaktionsabbruch keine Datensätze unnötig in den Cache geladen werden. Zusätzlich kann das Nachladen der benötigten Datensätze unabhängig von der Änderungsoperation und deren Transaktion erfolgen. Dadurch kann die Transaktion schneller abgeschlossen werden und muss nicht aufgrund des Nachladeprozess verzögert werden. Allerdings sind nach dem *Commit* einer Transaktion die RCCs im Cache gegebenenfalls nicht mehr gültig. Um dennoch Operationen im Cache ausführen zu können, muss die Ausführung der Anfragen an diese Gegebenheiten angepasst werden.

Aufgrund der Vorteile ist die Umsetzung des asynchronen Nachladens sehr wünschenswert. Aus diesem Grund wird diese Art des Nachladens im weiteren Verlauf näher betrachtet.

Asynchrones Nachladen von Datensätzen

In Kapitel 3.2.2 wurden bereits einige Ideen vorgestellt, die verhindern können, dass ungültige RCCs im Cache zur Anfrageausführung verwendet werden. Diese Ansätze sind:

1. Invalidierung kompletter ausgehender RCCs von Tabellen, deren Werte der RCC-Quellspalten sich aufgrund einer Änderungsoperation verändert haben.

2. Invalidierung ausgehender RCCs auf Basis von Werten, die aufgrund einer Änderungsoperation nicht wertvollständig in der Zieltabelle der RCCs vorhanden sind.

3. Speicherung zusätzlicher Informationen bezüglich der Gültigkeit von Werten ausgehender RCCs in den Datensätzen selbst.

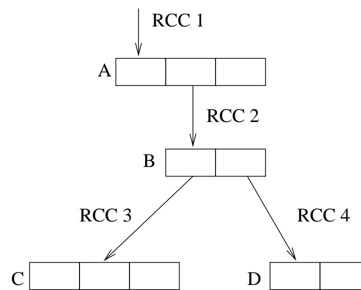


Abbildung 5.8: Beispiel einer Cache Group

In all diesen Optionen ist eine Anpassung der Anfrageverarbeitung notwendig, so dass inkonsistente Datenzustände verhindert werden. Jedoch wird die Variante 2 favorisiert, da diese bezüglich der Option 1 den Vorteil hat, dass hier nicht die kompletten, sondern nur Wertebereiche des RCCs invalidiert werden müssen. Bezüglich der Option 3 liegt der Vorteil der Variante 2, dass keine zusätzlichen Informationen in den Datensätzen und somit in den Tabellen gespeichert werden müssen.

Zum Nachladen der Datensätze in den Cache stehen, wie beim Füllen des Cache, prinzipiell die Alternativen bottom-up oder top-down zur Verfügung. Zur einfacheren Beschreibung, welche RCCs zu welchen Zeitpunkten invalidiert und nicht nutzbar sind, dient Abbildung 5.8. Erfolgt das Nachladen der Datensätze bottom-up, so ist der RCC 2 solange nicht verfügbar, bis alle notwendigen Datensätze in die Tabellen D, C und B nachgeladen wurden. Es müssen jedoch keine weiteren RCCs invalidiert werden, da während des Nachladens keine weiteren RCCs verletzt werden. Allerdings muss das Nachladen in diesem Fall entweder synchron erfolgen oder der RCC 2 bleibt sehr lange Zeit invalidiert. Würde man hingegen die Datensätze top-down in den Cache nachladen, so muss der RCC 2 solange invalidiert werden, bis alle notwendigen Datensätze in die Tabelle B geladen wurden. Nachdem alle Datensätze in die Tabelle B nachgeladen wurden, kann der RCC 2 wieder verwendet werden, jedoch RCC 3 und RCC 4 nicht, bis alle notwendigen Datensätze in die Tabellen C und D geladen wurden. Der Hauptvorteil und die Idee ist eine kurze inkrementelle Invalidierung der RCCs, um ein asynchrones Nachladen zu ermöglichen. In beiden Ansätzen ist es notwendig, dass die Datensätze einer Tabelle oder einer atomaren Zone innerhalb einer Transaktion nachgeladen werden. Dadurch kann verhindert werden, dass bereits während einzelner Nachladeschritte auf die nachgeladenen Datensätze zugegriffen wird.

Im ROpWAn-Verfahren wird vorerst ein synchrones Nachladen der Datensätze angenommen, jedoch ein asynchrones Nachladen angestrebt. Aus diesem Grund wurde überprüft, ob gegebenenfalls eine Kombination beider Fälle möglich wäre. In [GLR05] werden beispielsweise unterschiedliche Konsistenzgrade der Daten vorgestellt. In der Definition von Select-Anfragen wird dann definiert, welchen Konsistenzgrad die angefragten Daten erfüllen müssen. Es besteht die Möglichkeit, dass die angefragten Daten einen konsistenten Zustand darstellen müssen, oder dass sie z. B. maximal 10 Minuten alt sein dürfen. Diese Idee könnte folgendermaßen auf die RCCs im Cache angewandt werden: Wird ein

RCC sehr oft verwendet oder müssen die Daten in der RCC-Zieltabelle immer auf dem aktuellsten Stand sein, so sollten die Daten in der RCC-Zieltabelle synchron nachgeladen werden. Können die Daten in einer RCC-Zielspalte jedoch in gewissen Maßen vom aktuellsten Datenstand abweichen, so könnte für die Verwendung des RCC eine Eigenschaft definiert werden, dass gegebenenfalls veraltete Daten bei der Anfrageverarbeitung miteinbezogen werden dürfen. Somit besteht die Möglichkeit, Datensätze asynchron in den Cache nachzuladen, ohne RCCs komplett oder teilweise invalidieren zu müssen. Diese Vorgehensweise wird anhand von Abbildung 5.8 verdeutlicht. Es wird angenommen, dass RCC2 konsistent sein muss, RCC3 und RCC4 jedoch veraltete Daten mit einbeziehen dürfen. Wird eine Änderung auf der Tabelle A ausgeführt und ein Nachladen von Datensätzen ausgelöst, so müssen die Datensätze in Tabelle B synchron nachgeladen werden. Die Datensätze der Tabellen C und D können hingegen verzögert nachgeladen werden. Auf diese Weise können die RCCs immer verwendet werden und es müssen keine Anpassungen bezüglich der Anfrageverarbeitung umgesetzt werden. Allerdings muss sichergestellt werden, dass die RCCs jederzeit ihre Eigenschaft, z. B. maximal 10 Minuten alte Daten in der Zieltabelle, erfüllen. Hierzu müsste beispielsweise zusätzlich der Zeitstempel eines Datensatzes gespeichert werden und es muss regelmäßig überprüft werden, ob die RCC-Eigenschaften erfüllt sind. Darüber hinaus kann diese Variante des Nachladens leider nicht in dieser Form umgesetzt werden, da die Entscheidung über die Konsistenz der Daten nur durch den Benutzer und nicht durch den Datenbank-Administrator erfolgen kann.

Berechnung der Anzahl nachzuladender Datensätze

Damit der Mehraufwand des Nachladens der Datensätze genauer abgeschätzt werden kann, wird mit Hilfe des in Kapitel 5.2 vorgestellten Kostenmodells für eine Cache Group abgeschätzt, wie viele Datensätze aufgrund einer Änderungsoperation nachgeladen werden müssen. Anhand der folgenden Beispiele wird beschrieben, wie die Anzahl der nachzuladenden Datensätze abgeschätzt werden kann.

Beispiel 1

Wird in Abbildung 5.9 eine Datenänderung auf dem Attribut $A.a$ ausgeführt, so beschreibt T_{akt} die Anzahl der Datensätze, die von dieser Änderungsoperation betroffen sind. Diese Anzahl kann mit Hilfe des Selektivitätsfaktors, der in Kapitel 5.2 definiert wurde, abgeschätzt werden. Wird der schlechteste Fall angenommen, so existieren nach der Ausführung der Datenänderung T_{akt} neue Werte in $A.c$. Diese Werte müssen aufgrund des RCC in Tabelle B wertvollständig gemacht werden. In Tabelle B enthalten $SF_{B.e} \cdot card(B)$ Datensätzen denselben Attributwert. Diese Anzahl resultiert aus der Formel 5.1 und dem Selektivitätsfaktor $SF_{B.e}$ des Attributs $B.e$. Zusätzlich kann abgeschätzt werden, dass etwa $T_{akt} \cdot SF_{A.c}$ neue Werte in der Spalte $A.c$ existieren. Somit müssen im schlechtesten Fall

$$T_L(B) = T_{akt} \cdot SF_{A.c} \cdot SF_{B.e} \cdot card(B)$$

Datensätze in der Tabelle B nachgeladen werden. Weiterhin muss überprüft werden, wie viele Datensätze in den Tabellen C und D nachgeladen werden

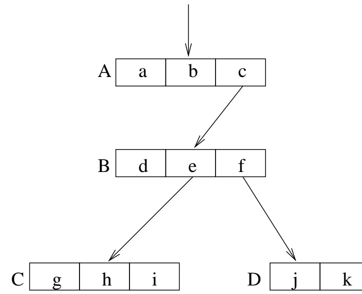


Abbildung 5.9: Beispiel 1: Hierarchische Abhängigkeiten

müssen. Da im schlechtesten Fall $T_{akt} \cdot SF_{A.c}$ neue Werte in die Spalte $B.e$ geladen wurden, müssen diese in $C.h$ wertvollständig gemacht werden. Folgende Anzahl nachzuladender Datensätze ist für Tabelle C notwendig:

$$T_L(C) = T_{akt} \cdot SF_{A.c} \cdot SF_{C.h} \cdot card(C)$$

In Spalte $B.f$ werden, analog zu Spalte $A.c$, im schlechtesten Fall $T_L(B) \cdot SF_{B.f}$ neue Werte nachgeladen. Aus diesem Grund werden in die Tabelle D

$$T_L(D) = T_L(B) \cdot SF_{B.f} \cdot SF_{D.j} \cdot card(D)$$

Datensätze nachgeladen. Um die Anzahl aller, aufgrund einer Änderungsoperation, nachzuladenden Datensätze bestimmen zu können, müssen die soeben beschriebenen Werte aufsummiert werden, also:

$$T_L = T_L(B) + T_L(C) + T_L(D)$$

Beispiel 2

Da in einem homogenen Zyklus (siehe Abbildung 5.10) benötigte Datensätze nur einmal pro Tabelle in den Cache geladen werden, kann der Zyklus in eine Hierarchie aufgespalten werden, um die Anzahl der nachzuladenden Datensätze abschätzen zu können. So muss der RCC $B.f \rightarrow A.b$ für die Analyse nicht betrachtet werden und somit gilt analog zur Berechnung von $T_L(B)$ aus Beispiel 1:

$$T_L(B) = T_{akt} \cdot SF_{A.b} \cdot SF_{B.e} \cdot card(B)$$

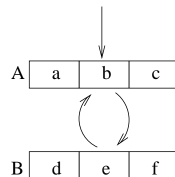


Abbildung 5.10: Beispiel 2: Homogener Zyklus

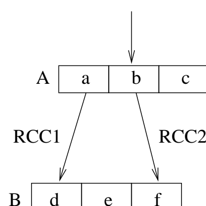


Abbildung 5.11: Beispiel 3: Beispiel einer Cache Group

Beispiel 3

Das Beispiel der Abbildung 5.11 verdeutlicht, wie sich die Anzahl der nachzuladenden Datensätze verändert, sofern mehrere ausgehende RCCs auf dieselbe Tabelle weisen. Hier gilt ebenfalls, dass $T_{akt} \cdot SF_{A.a}$ bzw. $T_{akt} \cdot SF_{A.b}$ neue Werte in der Tabelle A existieren. Wird der schlechteste Fall angenommen, überschneiden sich die Mengen der nachzuladenden Datensätzen aufgrund von $RCC1$ und $RCC2$ nicht. Sind die Datensatzmengen von RCC1 und RCC2 disjunkt, so entspricht dies dem Fall aus Beispiel 1, sofern nur die Tabellen B, C und D betrachtet werden und es gilt für die Abbildung 5.11:

$$T_L = T_{akt} \cdot SF_{A.a} \cdot SF_{B.d} \cdot card(B) + T_{akt} \cdot SF_{A.b} \cdot SF_{B.f} \cdot card(B)$$

Im besten Fall überschneiden sich die Mengen der nachzuladenden Datensätze so, dass sie ineinander enthalten sind, z. B. Menge(RCC1) $\subseteq$ Menge(RCC2) und es gilt:

$$T_L = \cdot max(T_{akt} \cdot SF_{A.a} \cdot SF_{B.d} \cdot card(B),\ T_{akt} \cdot SF_{A.b} \cdot SF_{B.f} \cdot card(B))$$

In Anhang B.3.3 wird ein Pseudocode zur Berechnung der Anzahl nachzuladender Datensätze dargestellt. In der Methode *computeLoad()* wird ausgehend von der Tabelle, auf der Datensätze verändert wurden, die Berechnung gestartet. Hierbei ist zu beachten, dass dies nur für eine Wertänderung auf einem Attribut des Datensatzes vorgestellt wird. Werden mehrere Attribute verändert, so muss die Methode entsprechend erweitert werden. Ausgehend von der veränderten Tabelle werden alle ausgehenden RCCs der Spalte verfolgt, in der aufgrund der Änderung neue Werte existieren. Basierend auf diesen RCCs wird die entsprechende Anzahl nachzuladender Datensätze pro RCC-Zieltabelle berechnet. Im nächsten Schritt werden rekursiv für alle ausgehenden RCCs der RCC-Zieltabelle die Anzahl der Datensätze berechnet (*computeRekLoad()*) und schließlich aufsummiert. Analog zur Vorgehensweise im Beispiel 1 wird unterschieden, ob in der RCC-Quellspalte neue Werte aufgrund der Änderung selbst (Beispiel 1, Fall: B.e $\rightarrow$ C.h) oder aufgrund neu geladener Werte (Beispiel 1, Fall: B.f $\rightarrow$ D.j) existieren. Je nachdem wird der aktuelle Wert T_{akt} oder aber der Wert $T_L(RCC - Quelltabelle)$ verwendet.

5.5.6 Probleme bei der Ausführung von Änderungsoperationen im Cache

Aufgrund der Tatsache, dass im Cache nur Teilmengen der Daten aus dem Backend abgebildet werden, können Fehler bei der Ausführung von Änderungs-

operationen durch den Cache auftreten. In Kapitel 3.4 wurden bereits auf diese Problemstellungen hingewiesen. Diese Probleme werden in diesem Abschnitt wieder aufgegriffen, um hierfür Lösungsansätze vorzustellen.

Es besteht die Möglichkeit, dass bei der Bearbeitung einer Änderungsoperation im Backend eine Verletzung einer Integritätsbedingung auftritt und daraufhin die Änderungsoperation zurückgewiesen wird. Dies kann allerdings nicht im Cache erkannt werden, da dort keine Fremdschlüsselbeziehungen abgebildet werden. Es besteht allerdings dennoch eine Möglichkeit z. B. Fremdschlüsselbeziehungen bereits im Cache zu überprüfen, sofern diese Beziehungen im Cache gespeichert werden. Hierbei ist zu beachten, dass diese Beziehungen nicht in der Cache-Datenbank angelegt werden. Werden die Beziehungen im Cache gespeichert, so können zusätzlich zur Sondierung ebenfalls Integritätsbedingungen geprüft werden. Erst wenn beide Überprüfungen erfolgreich beendet wurden, darf die Änderungsoperation im Cache ausgeführt werden, andernfalls muss die Datenänderung an das Backend weitergeleitet werden. Da die Fremdschlüsselbeziehungen bekannt sind, können ebenfalls kaskadierende Änderungsoperationen korrekt ausgeführt werden, z. B. wenn ein *on delete set null* auf einer Fremdschlüsselbeziehung definiert ist.

Es besteht zusätzlich die Möglichkeit, dass im Cache die Where-Klausel der Änderungsoperation nicht ausgewertet kann, weil die referenzierten Tabellen nicht im Cache abgebildet sind. In diesem Fall gibt die Sondierung ein negatives Ergebnis zurück und die Änderung wird zur Verarbeitung an das Backend weitergeleitet.

Darüber hinaus kann es passieren, dass ein Datensatz nicht im Cache existiert, jedoch nach der Änderungsoperation in den Cache geladen werden muss, damit die Cache Constraints gültig bleiben. In diesem Fall wird die Änderungsoperation an das Backend weitergeleitet und dort ausgeführt. Danach wird das entsprechende Write Set an den Cache propagiert, so dass die Änderung im Cache nachgezogen werden kann. Durch das Nachziehen der Änderung im Cache wird der Datensatz in den Cache geladen.

Des Weiteren kann das Problem der Transaktions-Inversionen auftreten. Transaktions-Inversionen beschreiben die Situation, dass ein Benutzer innerhalb einer Transaktion nicht die Datenänderungen sehen kann, die aufgrund seiner Datenänderungen sichtbar sein sollten. Konnte eine Änderungsoperation bereits im Cache erfolgreich ausgeführt werden, enthält der Cache die aktuellsten Daten und der Benutzer kann auf diese zugreifen. Somit können nur dann Transaktions-Inversionen auftreten, wenn die Änderung nicht direkt im Cache bearbeitet werden kann. In diesem Fall werden mögliche Reaktionen anhand des folgenden Beispiels verdeutlicht. Es existieren die in Abbildung 5.12 dargestell-

Person:

id	name	alter
1	a	10
2	b	10
3	c	10
4	d	40
5	e	10

C_Person:

id	name	alter
1	a	10
2	b	10
3	c	10

Abbildung 5.12: Beispielszenario zur Diskussion von Transaktions-Inversionen

ten Datensätze und im Cache wird die Transaktion $T1 : b, A1, R1, c$ mit

```
A1: update Person set alter = 11 where alter = 10;
R1: select * from Person where id = 3;
```

ausgeführt. Da nicht alle von der Änderungsoperation A1 betroffenen Datensätze im Cache existieren, muss die Datenänderung an das Backend weitergeleitet werden. Sobald A1 im Backend erfolgreich durchgeführt wurde, wird das entsprechende Write Set erstellt. Danach wird dieses Write Set sowie die Benachrichtigung über die Operationsausführung an den Cache gesendet. Basierend auf dieser Voraussetzung sind folgende alternative Aktionen im Cache möglich:

- Die Benachrichtigung über die erfolgreiche Ausführung der Änderungsoperation wird an den Benutzer weitergeleitet, bevor das Write Set im Cache nachgezogen wurde.

  In diesem Fall kann der Benutzer die Leseoperation R1 initiieren, bevor das Write Set vollständig nachgezogen wurde. Es wurde beispielsweise nur der Datensatz mit $id = 1$ bereits im Cache aktualisiert. In diesem Fall liest R1 den alten Datenzustand des Datensatzes mit der $id = 3$, obwohl diese Transaktion eigentlich den neuen Stand lesen müsste. Diese Situation beschreibt eine Transaktions-Inversion.

- Im Cache wird zunächst das Write Set der Transaktion T1 nachgezogen, bevor der Benutzer über die erfolgreiche Ausführung der Datenänderung A1 informiert wird.

  Da das Write Set bereits erfolgreich nachgezogen wurde, kann der Benutzer mit der Leseoperation R1 den aktuellsten Datenbankzustand lesen. Somit tritt in diesem Szenario keine Transaktions-Inversion auf.

Anhand dieser zweiten Alternative wurde bereits beschrieben, wie Transaktions-Inversionen verhindert werden können. So darf der Benutzer erst dann über die erfolgreiche Durchführung einer Änderungsoperation informiert werden, sofern das Write Set im Cache vollständig nachgezogen wurde.

5.6 Fazit

In diesem Kapitel wurde ein Synchronisierungsverfahren vorgestellt, das auf einem Primary-Copy-Ansatz basiert und um Eigenschaften des ROWA-Verfahrens erweitert wurde. Leseoperationen werden wenn möglich lokal ausgewertet und nur dann an das Backend weitergeleitet werden, wenn die notwendigen Datenelemente nicht auf dem Knoten vorhanden sind. Änderungsoperationen hingegen werden in jedem Fall an das Backend propagiert und dort ausgeführt. Eine Optimierung kann durch eine lokale Ausführung der Datenänderung erfolgen, sofern die von der Änderung betroffenen Daten vorhanden sind. Im Basisalgorithmus wurde vorgestellt, dass in diesem Fall statt der Änderungsoperation das Write Set an das Backend weitergeleitet wird. Die Analyse der Deadlocks hat allerdings ergeben, dass die Änderungsoperation vor der Ausführung im Cache an das Backend weitergeleitet werden muss, um einige Deadlocks verhindern zu können. Sobald eine Datenänderung im Backend verarbeitet wurde, wird diese in Form eines Write Set an all die Caches weitergeleitet, die von der Änderung

betroffene Daten enthalten. Damit das Synchronisierungsverfahren weitestgehend unabhängig vom verwendeten Datenbanksystem agieren kann, wird auf das verwendete Datenbanksystem eine weitere Schicht, das Cache-System, aufgesetzt. Somit können Konflikte zwischen Transaktionen einerseits in den verwendeten Datenbanksystemen erkannt und aufgelöst werden und andererseits durch das Cache-System. Das Cache-System kann Write/Write-Konflikte zwischen Transaktionen anhand ihrer Write Sets erkennen. Die hier beschriebene Vorgehensweise wurde bezüglich möglicher Problemstellungen untersucht, wie z. B. Konflikte zwischen Operationen, Deadlocks, Anomalien oder Probleme bei der Ausführung von Änderungsoperationen im Cache. Während dieser Analyse konnten bereits diverse Lösungsansätze für die unterschiedlichen Problemstellungen vorgestellt werden. Allerdings konnte nicht für jede Problemstellung eine Lösung entwickelt werden. Die wichtigsten offenen Problemstellungen sind:

- In Kapitel 5.5.3 wurden diverse Szenarien vorgestellt, in denen Deadlocks auftreten können. Für viele dieser Situationen konnten Vorschläge zur Verhinderung des Deadlock gemacht werden, jedoch nicht für das *Deadlock-Szenario 2* und *Deadlock-Szenario 3*. Hier wird der Deadlock beispielsweise aufgrund eines Timeout aufgelöst, indem beide Transaktionen abgebrochen werden. Jedoch könnte weiter untersucht werden, ob noch andere Lösungsansätze diesen Deadlock verhindern können.

- Aufgrund der Analyse konnte festgestellt werden, dass unter der Garantie der Snapshot Isolation die Anomalien *Phantom-Problem* auftreten kann.

- Weiterhin könnten weitere Regeln bezüglich der Auflösung von Konflikten definiert werden. Somit können gegebenenfalls noch gezielter Transaktionen abgebrochen werden.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurden grundlegende Probleme bei der Durchführung von Änderungsoperationen im Constraint-basierten Datenbank-Caching untersucht. Basierend auf dieser Analyse wurde ein Konzept eines Synchronisierungsverfahrens entwickelt, das speziell die Bedürfnisse des Constraint-basierten Datenbank-Caching betrachtet.

Da das Constraint-basierte Datenbank-Caching die Grundlage dieser Arbeit darstellt, wurde dieser Caching-Ansatz zunächst in Kapitel 2 anhand dessen Eigenschaften und Funktionen beschrieben. Ergänzt wurden diese Erläuterungen durch die Vorstellung des Prototypen *ACCache*. So konnte beispielsweise das Laden von Datensätzen in den Cache anhand der konkreten Realisierung im *ACCache* veranschaulicht werden.

Auf dieser Grundlage wurden in Kapitel 3 Problemstellungen diskutiert, die bei der Durchführung von Änderungsoperationen im Cache auftreten können. Diesbezüglich waren beispielsweise folgende Aufgabenstellungen von Bedeutung:

- Auswirkungen der Durchführung einer Änderungsoperation im Cache:
  - Überprüfung der Daseinsberechtigung eines geänderten Datensatzes
  - Identifikation der Datensätze, die aufgrund einer Datenänderung in den Cache nachgeladen werden müssen, so dass die Cache Constraints jederzeit gültig sind.
  - Identifikation möglicher Fehler bei der Ausführung von Änderungsoperationen im Cache, die lediglich im Backend erkannt werden können, wie z. B. die Verletzung von Integritätsbedingungen.
- Austausch von Änderungsoperationen zwischen Backend und Cache. Hier stand der Wissensstand des Backend über die vorhandenen Caches im Vordergrund, da hierdurch die Anzahl der Caches variieren kann, an die eine Änderungsoperation vom Backend aus weitergeleitet werden muss.

Diese Aufgaben wurden in Kapitel 4 mit den Problemstellungen der Synchronisierung von Replikaten ergänzt, wie z. B.:

- Auftreten von Anomalien

- Existenz von Deadlocks
- Welche Informationen werden zwischen Backend und Cache ausgetauscht? Es kann beispielsweise sinnvoll sein, lediglich die Änderungsoperationen zu propagieren.
- Zeitpunkt der Synchronisierung von Transaktionen auf allen Replikaten

Darüber hinaus wurden Ziele eines zu realisierenden Synchronisierungsverfahrens im Constraint-basierten Datenbank-Caching beschrieben. Aufgrund dieser Zielvorstellungen wurde in Kapitel 5 ein Konzept eines Synchronisierungsverfahrens für das Constraint-basierte Datenbank-Caching erstellt.

Das Verfahren basiert auf einem Primary-Copy-Ansatz, der um Eigenschaften des ROWA-Verfahrens erweitert wurde. Hierbei werden Leseoperationen wenn möglich lokal ausgewertet und nur dann an das Backend weitergeleitet, wenn die notwendigen Datenelemente nicht auf dem Knoten vorhanden sind. Änderungsoperationen hingegen werden in jedem Fall an das Backend propagiert und dort ausgeführt. Optimiert wird dies, indem die Datenänderung zunächst lokal ausgeführt wird, sofern die von der Änderung betroffenen Daten vorhanden sind. Dieses Verfahren wurde bezüglich der in Kapitel 3 und Kapitel 4 identifizierten Problemstellungen analysiert und so konnten folgende Lösungsansätze entwickelt werden:

- Durch die Verwendung des Cache-Systems ist dieses Synchronisierungsverfahren weitestgehend unabhängig von dem darunter liegenden Datenbanksystem. Es muss lediglich der vom Datenbanksystem garantierte Isolationsgrad beachtet werden, damit die Konflikte der Transaktionsoperationen korrekt behandelt werden können.
- Zwischen den Knoten werden lediglich Änderungsoperationen oder Write Sets ausgetauscht. Leseoperationen werden nur zwischen Backend und Cache ausgetauscht, sofern die angefragten Daten nicht im Cache vorhanden sind. Somit ist ein Austausch von Leseoperationen zwischen Backend und Cache in den meisten Fällen nicht notwendig, wodurch Nachrichten eingespart werden können.
- Es wurden Datenstrukturen zur Speicherung der Cache-Struktur (Cache-Group-Definition und Cache-Group-Inhalt) im Backend vorgestellt. Mit Hilfe dieser Informationen kann die Menge der Caches, an die Änderungsoperationen zur Aktualisierung weitergeleitet werden müssen, eingegrenzt werden.
- Aufgrund der für dieses Verfahren beschriebenen Nachrichtenkommunikation kann sichergestellt werden, dass eine Änderungsoperation auf jedem Knoten auf der gleichen Datenmenge ausgeführt wird. Dies wird einerseits aufgrund der Write Sets erreicht und andererseits durch die im Datenbanksystem verwendeten Datensperren.
- Konflikte können einerseits durch die verwendeten Datenbanksysteme erkannt und aufgelöst werden und andererseits durch das Cache-System. So können beispielsweise Write/Write-Konflikte bereits durch das Cache-System mit Hilfe von Write Sets erkannt werden.

- Die Wahrscheinlichkeit für das Auftreten von Deadlocks kann verringert werden, da mit Hilfe von Write Sets frühzeitig Write/Write-Konflikte erkannt werden können.
- Bei der Analyse der möglichen Anomalien konnte festgestellt werden, dass unter der Annahme von R/X-Sperrverfahren lediglich das Phantom-Problem auftreten kann. Dies kann jedoch unter der Verwendung von hierarchischen Sperrverfahren verhindert werden.

Leider konnten im Rahmen dieser Arbeit nicht für alle Problemstellungen Lösungsansätze zu deren Verhinderung entwickelt werden. Aus diesem Grund sind beispielsweise in zukünftiger Forschung noch folgende Fragen zu klären:

- In Kapitel 5.5.3 wurden diverse Szenarien vorgestellt, in denen Deadlocks auftreten können. Für das *Deadlock-Szenario 1* konnte ein Vorschlag zur Verhinderung des Deadlock gemacht werden, jedoch nicht für das *Deadlock-Szenario 2* und *Deadlock-Szenario 3*.
- Aufgrund der Analyse konnte festgestellt werden, dass unter der Garantie der Snapshot Isolation die Anomalie *Phantom-Problem* auftreten kann. Es wurden bisher noch keine Lösungsansätze entwickelt, um das Auftreten dieser Anomalie zu verhindern, da der Fokus der Analyse des Basisalgorithmus nicht unter der Garantie der Snapshot Isolation erfolgte.
- Es könnten noch weitere Regeln bezüglich der Auflösung von Konflikten zwischen Transaktionen definiert werden, die noch nicht in Kapitel 5.5.2 beschrieben wurden.

Zusätzlich zu diesen offenen Problemstellungen ist die Aufgabe zukünftiger Forschung dieses Konzept zu realisieren und im Prototyp *ACCache* zu implementieren. Darüber hinaus wurde anhand des in Kapitel 5.2 vorgestellten Kostenmodells angedeutet, welcher Nachrichtenaufwand im Basisalgorithmus anfällt. Zusätzlich wurde beschrieben, wie die Anzahl der nachzuladenden Datensätze abgeschätzt werden kann. Diese Abschätzungen können nach der Implementierung des Verfahrens mit Messungen konkretisiert werden.

Anhang A

Anomalien des Mehrbenutzerbetriebs

In diesem Kapitel werden die Anomalien [BBG+95, HR01] vorgestellt, die aufgrund des Mehrbenutzerbetriebs auftreten können. Diese Anomalien können nicht auftreten, wenn das System Serialisierbarkeit garantiert.

Dirty Read

Ein Dirty Read tritt beispielsweise in folgender Historie auf:

$$H = b_1, b_2, w_1(a), r_2(a), a_1, c_2$$

Transaktion T_1 modifiziert das Datenobjekt a. Danach liest Transaktion T_2 dieses geänderte Datenobjekt a, bevor T_1 erfolgreich beendet oder abgebrochen wurde. Wird Transaktion T_1 abgebrochen, so hat T_2 einen Datenzustand gelesen, der nicht durch Commit bestätigt wurde oder wird, woraus Inkonsistenzen resultieren.

Dirty Write

Folgende Historie dient zur Darstellung des Dirty Write:

$$H = b_1, b_2, w_1(a), w_2(a), c_1, c_2$$

Transaktion T_1 modifiziert Datenobjekt a und Transaktion T_2 modifiziert ebenfalls dieses Datenobjekt, bevor T_1 erfolgreich beendet oder abgebrochen wird. Das Problem Dirty Write entsteht, sobald T_1 oder T_2 abgebrochen wird, da dann nicht klar ist, welcher der korrekte Wert des Datenobjekts a sein sollte.

Non-repeatable Read / Inconsistent Read

Folgende Historie dient zur Veranschaulichung:

$$H = b_1, b_2, r_1(a), w_2(a), c_2, r_1(a), c_1$$

Transaktion T_1 liest ein Datenobjekt a und Transaktion T_2 modifiziert oder löscht dieses Datenobjekt und das Commit wird ausgeführt. Dann versucht T_1

erneut Datenobjekt a zu lesen. In diesem Fall sieht T_1 entweder das modifizierte Datenobjekt oder entdeckt, dass Werte gelöscht wurden.

Phantom

Das Phantom-Problem entsteht beispielsweise in der folgenden Historie:

$$H = b_1, b_2, r_1(S), w_2(S), c_2, r_1(S), c_1$$

Transaktion T_1 liest eine Menge von Datenobjekten, die einem Suchkriterium S genügen. Transaktion T_2 erzeugt oder löscht ein Datenobjekt, das ebenfalls das Suchkriterium S erfüllt und T_2 wird durch Commit abgeschlossen. Wenn T_1 erneut die Menge der Datenobjekte liest, die S erfüllen, so ist diese Menge nun unterschiedlich zu der zuvor gesehenen Datenmenge.

Lost Update

Folgende Historie veranschaulicht das Lost-Update-Problem:

$$H = b_1, b_2, r_1(a), w_2(a), w_1(a), c_1, c_2$$

Transaktion T_1 liest Datenobjekt a und Transaktion T_2 ändert dieses Datenobjekt. T_1 modifiziert ebenfalls, basierend auf dem vorherigen Lesen, das Datenobjekt a und wird erfolgreich beendet. Ein Problem resultiert, wenn T_2 ebenfalls das Commit ausführt, weil dieses Update von T_2 verloren geht.

Read Skew

Folgende Historie stellt das Phänomen des Read Skew dar:

$$H = b_1, b_2, r_1(a), w_2(a), w_2(b), c_2, r_1(b), c_1$$

Transaktion T_1 liest das Datenobjekt a. Dann ändert Transaktion T_2 die Datenobjekte a und b und diese neuen Werte werden erfolgreich verändert. Danach liest T_1 das Datenobjekt b. Wenn zwischen den Datenelementen a und b eine Bedingung definiert ist, liest Transaktion T_1 aufgrund der bereits erfolgreich beendeten Transaktion T_2 einen inkonsistenten Datenbankzustand.

Write Skew

Folgende Historie dient zur Veranschaulichung:

$$H = b_1, b_2, r_1(a), r_1(b), r_2(a), r_2(b), w_1(a), w_2(b), c_1, c_1$$

Transaktion T_1 liest die Datenobjekte a und b und verändert a. Dann liest Transaktion T_2 ebenfalls diese beiden Datenobjekte, modifiziert Objekt b und wird erfolgreich beendet. Ein Problem entsteht, wenn Bedingungen zwischen diesen beiden Objekten definiert sind, da diese verletzt werden könnten, obwohl beide Transaktionen die Bedingungen erfüllen.

Anhang B

Pseudocode für ROpWAn

B.1 Code für das Backend

```
receiveMsg();
/* MsgInfos:
 * taid, operation, statement -> TA lokal im Backend
 * taid, operation, statement, writeSet, info, cid -> TA aus Cache
 */
if (begin == operation){
    taid = getConnection();
} else if (read == operation){
    // statement in Datenbank ausführen
    Result Set rs = forwardRead(statement);
    if (cid != null){
        // Result Set an Cache zurückschicken
        sendMsg(cid, rs);
    } else {
        // Result Set an Client schicken
        forwardRS(rs);
    }
} else if (write == operation){
    boolean conflicts = false;
    if (writeSet != null){
        conflicts = checkForConflicts(writeSet);
    }
    if (!conflicts){
        // statement in Datenbank ausführen
        forwardWrite(statement);
        // Write Set erzeugen und an Caches weiterleiten
        propagateWS( createWS(statement) );
    } else {
        // eine der konfliktären TAs abbrechen, BE-TA hat Vorrang
        abort(conflictingTA);
    }
} else if (commit == operation){
    // auslösen des Abschluss des 2PC
    initiate2PC(taid);
}
```

B.2 Code für den Cache

```
receiveMsg();
/* MsgInfos:
 * taid, operation, statement -> lokale TA auf Cache
 * taid, writeSet -> Nachziehtransaktion vom BE
 */
if (begin == operation){
    taid = getConnection();
} else if (read == operation){
    // Überprüfen, ob Leseoperation im Cache ausgeführt werden kann
    if (sondieren(statement)){
        // statement in Datenbank ausführen
        Result Set rs = forwardRead(statement);
        // Result Set an Client schicken
        forwardRS(rs);
    } else {
        // Operation kann nicht im Cache ausgeführt werden
        // und muss folglich ans BE weitergeleitet werden
        propagateToBE(taid, operation, statement, cid);
    }
} else if (write == operation){
    boolean info = sondieren(statement);
    if (info){
        // statement in Datenbank ausführen
        forwardWrite(statement);
        // Write Set erzeugen
        WriteSet ws = createWS(stmt);
    }
    // Änderungsoperation ans BE weiterleiten
    propagateToBE(taid, operation, statement, ws, info, cid);
} else if (commit == operation){
    // Commit-Anforderung an Backend weiterleiten
    propagateToBE(taid, operation);
}

if (writeSet != null){
    if (! checkForConflicts(writeSet){
       // Änderungsoperation muss im Cache nachgezogen werden
       forwardWS(writeSet);
    } else {
       // eine der konfliktären TAs abbrechen, BE-TA hat Vorrang
       abort(conflictingTA);
    }
}
```

B.3 Weiterleitung von Write Sets

B.3.1 Backend hat eingeschränkte Wissensbasis

```
propagateWS( statement, writeSet, info, taId, cacheId ){
    String tablename = extractTableName(statement);
    List caches = {};
    // Statement zur Identifizierung der Caches erstellen
    String stmt = " select Cid from CTabellen
                  where Tabname = " + tablename ;

    // überprüfen, ob Änderungsoperation bereits im Cache ausgeführt wurde
    if (info != null && info){
        // alle Caches ungleich des Ursprungs-Cache identifizieren
        caches += " and Cid != " + cacheId;
    }
    // Liste der Caches, an die das Write Set weitergeleitet werden muss
    caches = executeDBstmt( stmt );

    Für jedes Element von caches {
        // Write Set an Caches weiterleiten
        sendWS(taid, writeSet);
    }
}
```

B.3.2 Backend hat vollen Wissensstand

```
propagateWS( statement, writeSet, info, taId, cacheId ){
    /* Folgende Informationen werden aus dem Statement extrahiert:
     * tabellenName (betroffene Tabelle)
     * attribut_i (Attribute der Where-Klausel)
     * wert_i (Werte der Where-Klausel)
     */
    List caches = {};
    // Statement zur Identifizierung der Caches erstellen
    String stmt = " select distinct(rcc.Cid)
                        from CRCCs rcc, CRCCWerte values
                        where rcc.RCCid = values.RCCid " ;

    Für jedes i {
        /* Für jedes Attribut die Bedingung erstellen
         * update/delete bearbeiten: wert_i = oldValue des Attributs
         * insert bearbeiten: wert_i = newValue des Attributs
         */
        stmt += " AND ( (TabNameQ = tabellenName and SpalteQ = attribut_i)
                     or (TabNameZ = tabellenName and SpalteZ = attribut_i)
                      ) and values.Wert = wert_i ";
    }

    // überprüfen, ob Änderungsoperation bereits im Cache ausgeführt wurde
    if (info != null && info){
        // alle Caches ungleich des Ursprungs-Cache identifizieren
        stmt += " and rcc.Cid = " + cacheId ;
    }
```

```
    // Liste der Caches, an die das Write Set weitergeleitet werden muss
    caches = executeDBstmt(stmt);

    Für jedes Element von caches {
        // Datensätze identifizieren, die in den Cache nachgeladen
        // werden müssen
        List dataSetsLoading = getDsToLoad(statement, cacheId);
        // Datenänderung mit den nachzuladenen Datensätzen an Cache senden
        sendWrite(taid, writeSet, dataSetsLoading);
    }
}

/* Menge der nachzuladenden Datensätze identifizieren */
getDsToLoad(statement, cacheId){
    /* Folgende Informationen werden aus dem Statement extrahiert:
     * table (betroffene Tabelle)
     * a_i (Menge aller Attribute, deren Werte geändert werden)
     * w_i (neue Werte der zu ändernden Attribute)
     */
    List dataSet = {};

    Für jedes i {
       dataSet += getDSrek(table, a_i, w_i, cacheId);
    }
    return dataSet;
}

getDSrek(tabelle, spalte, wert, cacheId){
    List dataSet = {};
    // Liste aller ausgehenden RCCs von tabelle
    List ausRCCs = executeDBstmt( " select * from CRCCs
                                    where TabNameQ = " + tabelle
                                    + " and SpalteQ =  " + spalte
                                    + " und Cid = " + cacheId; " );

    Für jedes Element aus ausRCCs {
       // jedes Element enthält: TabNameZ, SpalteZ
       // die nachzulandenden Datensätze für TabNameZ bestimmen
       dataSet += executeDBstmt( " select * from TabNameZ
                                  where SpalteZ = wert; " );
       // rekursiv alle ausgehenden RCCs verfolgen
       dataSet += getDSrek(TabNameZ, SpalteZ, wert, cacheId);
    }
    return dataSet;
}
```

B.3.3 Berechnung der Anzahl nachzuladender Datensätze

```
computeLoad(tabelle, spalte, start){
    /* tabelle: Tabelle, in der Datensätze geändert wurden, aufgrund
     *          dieser Änderungen wird das Nachladen ausgelöst
     * spalte: Attribut, wo neue Werte aufgrund der Datenänderung existieren
     * start: Anzahl neuer Werte aufgrund der Datenänderung
     */
    tLoad = 0; // Anzahl nachzuladender Tupel
    // Liste aller ausgehenden RCCs
    List ausRCCs = executeDBstmt( " select * from CRCCs
                                    where TabNameQ = tabelle and
                                    SpalteQ = spalte und Cid = cacheId; " );

    Für jedes Element aus ausRCCs {
       // jedes Element enthält: TabNameZ, SpalteZ, SpalteQ
       if (SpalteQ == spalte){
           // neue Werte der Quellspalte existieren aufgrund der Änderung
           sf = start;
       } else {
           // neue Werte der Quellspalte, aufgrund des Ladens der Datensätze
           sf = start * SF(SpalteQ);
       }
       // nachzuladende Tupelanzahl für diesen RCC berechnen
       tLoad_RCC = sf * SF(SpalteZ) * card(TabNameZ);
       tLoad += tLoad_RCC;
       // rekursiv alle ausgehenden RCCs von TabNameZ verfolgen
       tLoad += computeRekLoad(TabNameZ, SpalteZ, start, tLoad_RCC);
    }
    return tLoad;
}


computeRekLoad(tabelle, spalte, start, tLoad){
    /* tabelle: Tabelle, in die Datensätze nachgeladen werden müssen
     * spalte: Attribut, wo neue Werte aufgrund der Datenänderung existieren
     * start: Anzahl neuer Werte aufgrund der Datenänderung selbst
     * tLoad: Anzahl neuer Werte einer Spalte, aufgrund des Ladens
     *        von Datensätzen
     */
    tLoad = 0; // Anzahl nachzuladender Tupel
    // Liste aller ausgehenden RCCs
    List ausRCCs = executeDBstmt( " select * from CRCCs
                                    where TabNameQ = tabelle and
                                    SpalteQ = spalte und Cid = cacheId; " );

    Für jedes Element aus ausRCCs {
       // jedes Element enthält: TabNameZ, SpalteZ, SpalteQ
       if (SpalteQ == spalte){
           // neue Werte der Quellspalte existieren aufgrund der Änderung
           sf = start;
       } else {
           // neue Werte der Quellspalte, aufgrund des Ladens der Datensätze
           sf = tLoad;
       }
```

```
        // nachzuladende Tupelanzahl für diesen RCC berechnen
        tLoad_RCC = sf * SF(SpalteZ) * card(TabNameZ);
        tLoad += tLoad_RCC;
        // rekursiv alle ausgehenden RCCs von TabNameZ verfolgen
        tLoad += computeRekLoad(TabNameZ, SpalteZ, start, tLoad_RCC);
    }
    return tLoad;
}
```

Literaturverzeichnis

[ABK+03] M Altinel, C Bornhövd, S Krishnamurthy, C Mohan, H Pirahesh, and B Reinwald. Cache tables: Paving the way for an adaptive database cache. In *VLDB Conference*, pages 718–729, 2003. `http://citeseer.ist.psu.edu/638059.html`[last access: 9. September 2007].

[APTP03] K Amiri, S Park, R Tewari, and S Padmanabhan. DBProxy: A dynamic data cache for web applications. In *ICDE Conference*, pages 821–831, 2003. `http://citeseer.ist.psu.edu/amiri03dbproxy.html`[last access: 9. September 2007].

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabetz O'Neil, and Patrick O'Neil. A Crititque of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995. `http://citeseer.ist.psu.edu/berenson95critique.html`[last access: 9. September 2007].

[BD96] T. Beuter and P. Dadam. Prinzipien der Replikationskontrolle in verteilten Datenbanksystemen. In *GI Informatik Forschung und Entwicklung (Nov. 1996)*, pages 203–212, 1996.

[BDD+98] RG Bello, K Dias, A Downing, JJ Jr. Feenan, JL Finnerty, WD Norcott, H Sun, A Witkowski, and M Ziauddin. Materialized views in Oracle. In *VLDB Confrence*, pages 659–664, 1998.

[Bü06] Andreas Bühmann. Ein Schritt zurück ist kein Rückschritt. In *Informatik - Forschung und Entwicklung*, pages 184–195, 2006. `http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/pubs/papers/Bue05.IFE.html` [last access: 9. September 2007].

[Cor04] Oracle Corporation. Internet application server documentation library, 2004. `http://otn.oracle.com/documentation/appserver10g.html`.

[DS06] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB Conference*, pages 715–726, 2006.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. In *ACM Transactions on Database Systems*, pages 492–528, 2005.

[GL01] J Goldstein and P Larson. Using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[GLR05] Hongfei Guo, Per-Âke Larson, and Raghu Rmakrishnan. Caching with „Good Enough“ Currency, Consistency, and Completeness. In *VLDB Conference*, pages 457–468, 2005.

[HB07] Theo Härder and Andreas Bühmann. Value Complete, Column Complete, Predicate Complete - Magic Words Driving the Design of Cache Groups. In *VLDB Journal*, 2007. `http://wwwdvs.informatik.uni-kl.de/pubs/papers/HB06.VLDB.html` [last access: 9. September 2007].

[HR01] Theo Härder and Erhard Rahm. Datenbanksysteme - Konzepte und Techniken der Implementierung, 2001.

[Kle06] Joachim Klein. Entwicklung einer automatisierten Messumgebung für das Constraint-basierte Datenbank-Caching, 2006. `http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Kle06.DA.pdf`[last access: 9. September 2007].

[LKPnMJP05] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation, 2005.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering Queries Using Views. In *Proceedings of the 14th {ACM} {SIGACT}-{SIGMOD}-{SIGART} Symposium on Principles of Database Systems*, pages 95–104, 1995. `http://citeseer.ist.psu.edu/levy95answering.html`[last access: 9. September 2007].

[Mer05] Christian Merker. Konzeption und Realisierung eines Constraint-basierten Datenbank-CAche, 2005. `http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Mer05.DA.pdf`[last access: 9. September 2007].

[MNKK06] Aekyung Moon, Han Namgoong, Hyoungsun Kim, and Hyun Kim. A Data Replication Scheme based on Primary Copy for Ensuring Data Consistency in Mobile Ad hoc Networks, 2006. `http://ww1.ucmss.com/books/LFS/CSREA2006/CIC5182.pdf`[last access: 9. September 2007].

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence

Assumption. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 486–495, 1997. http://www.informatik.uni-trier.de/~ley/db/conf/vldb/PoosalaI97.html.

[Rah07] Erhard Rahm. Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung, 2007. http://dbs.uni-leipzig.de/buecher/mrdbs/[last access: 9. September 2007].

[Tea02] The TimesTen Team. Mid-tier caching: The TimesTen approach. In *SIGMOD Conference*, pages 588–593, 2002.

[TR90] Alexander Thomasian and Erhard Rahm. A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking. In *ICDCS*, pages 294–301, 1990.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274. IEEE Computer Society Technical Commitee on Distributed Processing, 2000. http://citeseer.ist.psu.edu/286185.html[last access: 9. September 2007].