Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3401

# Implementierung einer Online-Reorganisation von Tenantspaces

Henning Mälzer

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. Bernhard Mitschang |
| **Betreuer:** | Dipl.-Inf. Oliver Schiller |
| **begonnen am:** | 2. August 2012 |
| **beendet am:** | 1. Februar 2013 |
| **CR-Klassifikation:** | H.2.4, H.3.2, H.3.3 |

## Kurzfassung

Ein inzwischen weit verbreitetes Mittel zur Bereitstellung von Diensten ist Software as a Service, ein Teilbereich des Cloud Computings. Dabei stellt ein Anbieter eine Funktionalität über das Internet zur Verfügung, wobei zumeist Datenbanken zum Einsatz kommen. Ein Kunde, der einen solchen Service bucht wird Mandant oder englisch Tenant genannt. In diesem Kontext ist es von Vorteil, die Datenbankanwendung so auszulegen, dass viele Mandanten gemeinsam eine Datenbank nutzen. In vorhergehenden Arbeiten wurden sogenannte Tenantspaces definiert, die ein physisches Speichergranulat darstellen und als Ganzes verwaltet werden können. Diese Arbeit hat zur Aufgabe, ein Verfahren zu entwickeln, das es ermöglicht, solche Tenantspaces bei Bedarf effizient zu reorganisieren, ohne dabei den Betrieb einstellen zu müssen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# Abkürzungsverzeichnis

**ACID** Atomarität (Atomicity) - Konsistenz (Consistency) - Isolation - Dauerhaftigkeit (Durability)

**CTID** Current Tuple Identifier

**DaMT** Data and Multi-Tenancy

**DB** Datenbank

**DBMS** Datenbankmanagementsystem

**HOT** Heap Only Tuple

**MVCC** Multi Version Concurrency Control

**SaaS** Software as a Service

**SLA** Service-Level-Agreement

**TID** Tuple Identifier

**URI** Uniform Resource Identifier

**WAL** Write-Ahead-Log

# Ausschnittverzeichnis

# 1. Einleitung

Ein inzwischen weit verbreitetes Mittel zur Bereitstellung von Diensten ist Software as a Service (SaaS), ein Teilbereich des Cloud Computings. Dabei stellt ein Anbieter eine Funktionalität über das Internet zur Verfügung. Der Kunde spart sich dadurch die Anschaffung der Infrastruktur und zahlt nur eine vergleichsweise geringe Gebühr pro Endanwender und Nutzungszeit. Der Dienst wird dabei meist als Datenbankanwendung realisiert und die Komplexität vor dem Kunden verborgen. In diesem Kontext ist großes Einsparpotential vorhanden. Die nötigen Betriebsmittel wie Webserver, Anwendungsserver, Festspeicher und Datenbank so wie Backup- und Disaster Recovery-Lösungen werden vom Dienstanbieter bereitgestellt und betrieben und vor dem Kunden verborgen. Auf technischer Ebene nutzen viele Kunden die gleiche Instanz, wodurch die Anwendung für jeden einzelnen Kunden billiger wird.

Als Beispiel sei ein Onlineshop genannt. Da die Anforderungen der einzelnen Kunden sehr ähnlich sind, kann eine einzelne Datenbankinstanz so ausgelegt werden, dass sie viele Kunden gleichzeitig bedient. Der Dienstanbieter stellt die Funktionalität "Onlineshop" zur Verfügung und kümmert sich um Betrieb und Absicherung aller nötigen Mittel. Der Kunde, im weiteren Mandant oder im englischen Tenant genannt, muss nur seine Instanz des Onlineshops buchen und konfigurieren, indem er Layout und Service-Level-Agreement (SLA) festlegt und firmenbezogene Daten und seine Produkte einstellt. Die Instanz des Mandanten wird über einen eigenen Uniform Resource Identifier (URI) im Internet verfügbar gemacht. Der Endkunde besucht die realisierte Webseite und kauft darüber die Produkte des Mandanten.

Im beschriebenen Beispiel ist von entscheidender Bedeutung, dass die Daten der einzelnen Mandanten isoliert voneinander gespeichert werden und kein Mandant Zugriff auf die Daten eines anderen Mandanten erhält. Diese Isolierung der Mandanten wird in bisherigen Implementierungen durch eine weitere Schicht oberhalb der Datenbank realisiert. Dabei kann jedoch nicht das volle Optimierungspotential ausgeschöpft werden, da zum Beispiel für alle Mandanten die gleiche Konfiguration des Datenbankmanagementsystem (DBMS) verwendet werden muss. Idealerweise sollte der Dienstanbieter in der Lage sein, verschiedenen Mandanten verschiedene Service Levels zu verschiedenen Preisen anzubieten.

Deshalb wird im Forschungsprojekt Data and Multi-Tenancy (DaMT) an der Universität Stuttgart daran gearbeitet, die Trennung der Mandanten bis auf das physische Speicherlayout der Datenbank herunterzubrechen und effizient zu verwalten.

## 1.1. Motivation

In den vorhergehenden Arbeiten [Sch10], [Sch11] und [Fic11] wurde das Konzept der Tenantspaces entwickelt und exemplarisch auf der Basis von PostgreSQL Version 8.4 implementiert. Darin wird in der einfachen Version pro Mandant für jede von der Anwendung benötigte Tabelle eine eigene Tabelleninstanz angelegt. Auf der physischen Ebene wird eine eigene Datei erstellt. Jeder Mandant erhält seinen eigenen Tenantspace. Der Tenantspace ist dabei als zusätzliches Speichergranulat der Datenbank definiert, ähnlich einem Tablespace, und kann mit entsprechenden Kommandos als Ganzes verwaltet werden. In einer weiteren Version können die Tenantspaces verschiedener Mandanten zu einem einzigen vereinigt werden. Die Isolation ist dann weiterhin gegeben, da jedes Tupel eine Markierung des zugehörigen Mandanten erhält, aber die Daten verschiedener Mandanten werden in einer Datei abgelegt. Auf dieser Basis können Mandanten mit gleichen Anforderungen gruppiert und gemeinsam verwaltet werden. Ein Mandant mit einem hohen Datenvolumen und einer hohen Frequenz der Endkunden-Zugriffe kann einen eigenen Tenantspace erhalten, während viele Mandanten mit geringem Datenvolumen in einem einzelnen Tenantspace zusammengefasst werden können.

Um die Gruppierung der Mandanten an veränderte Anforderungen anpassen zu können, stehen dem Administrator die Befehle SPLIT TENANTSPACE und MERGE TENANTSPACE zur Verfügung, die entsprechend einen Tenantspace aufspalten oder mehrere Tenantspaces vereinigen, ohne dass währenddessen der Dienst eingestellt werden muss.

Ein bekanntes Prinzip in Datenbanken ist der Write-Ahead-Log (WAL). Es besagt, dass alle Modifikationen der Daten zuerst protokolliert werden müssen, bevor sie in die eigentliche Datenbasis eingebracht werden können. Die angesprochenen Operatoren SPLIT und MERGE nutzen im bisherigen Algorithmus diesen WAL der Datenbank, um alle nebenläufigen Änderungen auf den Quellrelationen zu erfassen. Nach Abschluss der Transformation einer Tabelle wird der mitgeschriebene WAL auf den Zielrelationen angewendet. Während der WAL auf die Zielrelationen angewendet wird, wird ein weiterer WAL erzeugt, der wiederum danach angewendet wird. Der Umfang der anzuwendenden WALs nimmt dabei entsprechend ab. Erst dann, wenn kein neuer WAL mehr angewendet werden muss, wird die Quellrelation gelöscht und alle nebenläufigen Transaktionen können auf die neuen reorganisierten Relationen zugreifen.

Bei diesem Verfahren wird jede nebenläufig ändernde Transaktion doppelt ausgeführt, nämlich einmal auf der Quellrelation und einmal bei der Anwendung des WAL auf der Zielrelation. Deshalb entsteht beim Einspielen jedes neuen WAL ein erheblicher Einbruch der Antwortzeiten bei den nebenläufigen Transaktionen.

## 1.2. Problemstellung

Mit dieser Arbeit soll ein neuer Ansatz ausgearbeitet werden, der ohne den WAL auskommt und anstatt dessen die reorganisierten Teile direkt auf Seitenbasis verfügbar macht. De-

mentsprechend soll aus Sicht der Transaktionen ein fließender, kontinuierlicher Übergang von nicht reorganisierten Tabellen zu reorganisierten Tabellen entstehen. Jede nebenläufige Modifikation der Daten muss dann nur noch einmal eingebracht werden.

Ein weiterer erhoffter Effekt liegt darin, dass eine Einflussmöglichkeit auf die zu erwartenden Antwortzeiten der nebenläufigen Transaktionen entsteht. Idealerweise sollte es möglich sein, bei hohem Lastaufkommen die Reorganisation der Daten auszubremsen, um damit die nebenläufigen Transaktionen zu beschleunigen. Es soll untersucht werden, ob es möglich ist, diesen Einfluss mit einem einstellbaren Faktor sinnvoll zu skalieren.

Dabei muss das Transaktionen zugrundeliegende Prinzip Atomarität (Atomicity) - Konsistenz (Consistency) - Isolation - Dauerhaftigkeit (Durability) (ACID) eingehalten werden. Jede ändernde Transaktion muss atomar sein, also entweder ganz oder gar nicht eingebracht werden. Sie muss eine konsistente Datenbasis hinterlassen, sofern auch die vorherige konsistent war. Nebenläufige Transaktionen dürfen sich gegenseitig inhaltlich nicht beeinflussen und jede Änderung muss vor Abschluss der Transaktion dauerhaft gespeichert sein. [HR01, vgl. S. 393f]

Insbesondere müssen zu jedem Zeitpunkt die jeweils neuesten Daten - ob auf Quell- oder Zielrelation für alle nebenläufigen Transaktionen erreichbar sein.

Um den Aufwand auf ein Maß einzuschränken, das innerhalb der Bearbeitungszeit einer Diplomarbeit machbar ist, sollen nur die nebenläufigen Operatoren SELECT, INSERT, DELETE und UPDATE unterstützt werden.

## 1.3. Lösungsansatz

Der Ansatz zur Lösung des Problems besteht darin, während des Reorganisationsvorgangs eine weitere Indirektion von der Quell- zur Zielrelation auf Tupelebene einzuführen und diese auch für nebenläufige Operationen verfügbar zu machen. Nebenläufige Transaktionen öffnen beide Relationen und lesen von jedem Tupel die jeweils neueste Version abhängig vom Fortschritt der Reorganisation entweder auf der ersteren oder auf der letzteren. Es wird jeweils nur eine Relation gleichzeitig transformiert und nur auf dieser muss das entsprechende Verfahren angewendet werden.

Das Grundprinzip für nebenläufige Änderungen soll darin bestehen, alle logischen Änderungen der Daten sofern möglich in die Zielrelation zu schreiben und die entsprechenden Operationen so mit dem Reorganisationsvorgang zu verzahnen, dass die Konsistenz des Datenbestandes gewahrt bleibt.

Die Schwierigkeit liegt in der effizienten Synchronisation des Prozesses mit den nebenläufigen Transaktionen. Da der Reorganisationsvorgang die Daten nicht auf logischer Ebene verändert, sondern nur die physische Repräsentation verschiebt, nimmt er nur minimale Transaktionslevel-Sperren. Die physische Integrität der Daten wird durch Seitenlevel-Sperren garantiert.

## 1.4. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

In Kapitel **2. Ausgangsbasis** werden die in dieser Arbeit verwendeten Konzepte und Strukturen von PostgreSQL sowie das Konzept der Tenantspaces vorgestellt.

Danach wird in Kapitel **3. Konzept** ein neuer Algorithmus zur Reorganisation entwickelt und seine Korrektheit gezeigt.

In Kapitel **4. Implementierung** wird die konkrete Implementierung des neuen Algorithmus vorgestellt.

Kapitel **5. Evaluation** bewertet die neue Implementierung anhand von Funktionstests.

Schließlich fasst Kapitel **6. Zusammenfassung und Ausblick** die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte für weitere Arbeiten vor.

# 2. Ausgangsbasis

In diesem Kapitel werden die der Arbeit zugrundeliegenden Prinzipien und Strukturen erklärt. Unterkapitel 2.1 stellt die Architektur und Mechanismen der Datenbank, speziell die in PostgreSQL verwendeten, vor. Danach wird in Unterkapitel 2.2 näher auf das Prinzip der Tenantspaces eingegangen. Unterkapitel 2.3 stellt das Beispiel vor, an dem in der Folge die Strukturen des neu entwickelten Algorithmus erklärt werden.

## 2.1. Datenbanken

Die Kommunikation zwischen Anwendung oder allgemein Benutzer auf der einen Seite und Datenbank auf der anderen erfolgt in PostgreSQL auf Basis eines Client-Server-Modells. Wie in Abbildung 2.1 dargestellt, wird für jede Verbindung ein Backend genannter Prozess gestartet, der die Anfragen vom Client sequenziell abarbeitet. Die für den Mehrbenutzerbetrieb notwendige Synchronisation und Kommunikation zwischen den Prozessen wird über einen gemeinsamen Speicherbereich, den Shared Memory, abgewickelt. Der Shared Memory enthält außerdem den Puffer der Datenbank (DB), auch Bufferpool genannt. (vgl. [Sch11])

Abbildung 2.2 zeigt das Flussdiagramm des Backends. Für jede Anfrage wird der folgende Prozess durchlaufen. Der Parser liest die Anfrage. Der Traffic Cop entscheidet für komplexe

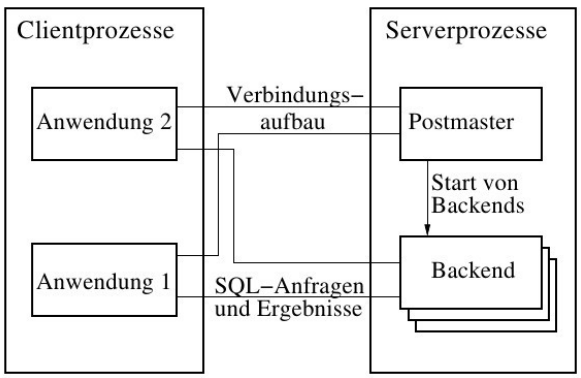


**Abbildung 2.1.:** Serveraufbau zur gleichzeitigen Abarbeitung mehrerer Anfragen, aus [Sch11]

Anfragen, diese zu optimieren und eine Ausführungsstrategie anzulegen. Komplexe Anfragen werden in einfachere Teilanfragen aufgeteilt und jede Teilanfrage rekursiv neu bearbeitet. Einfache Anfragen werden direkt an das Utility Modul weitergegeben und ausgeführt.

### 2.1.1. Tabellen und Tupel

Ein wesentliches Grundprinzip von Datenbanken besteht darin, die Daten strukturiert abzuspeichern. Jede Tabelle erhält eine feste Definition, wie ihre Einträge, auch Tupel genannt, auszusehen haben. Einzelne Spalten einer Tabelle können variable Typen haben, was bedeutet, dass die konkreten Einträge unterschiedliche Längen haben können. Auf der Ebene der Heapseiten, die im folgenden Unterkapitel 2.1.3 erklärt werden und auf der sich der Goßteil dieser Arbeit bewegt, haben jedoch alle Tupel eine feste Länge. Etwaige längere Tupel werden mit Hilfe anderer Mechanismen in kleinere Tupel zerlegt, bis die geforderte Eigenschaft der festen Länge erreicht ist.

### 2.1.2. Indizes

Um in großen Datenbeständen Tupel mit bestimmten Eigenschaften nicht aufwendig auf der ganzen Tabelle suchen zu müssen, können auf einer Tabelle verschiedene Indizes angelegt werden. Dabei ist jeder Index eine Liste aller Tupel, die die konkrete Eigenschaft erfüllen. Die aufwendige Suche ist somit nur einmal nötig, nämlich bei der Erstellung des Index. Jedes später auf der Tabelle einzufügende oder zu ändernde Tupel wird, sofern es die geforderte Eigenschaft erfüllt zugleich in den Index eingetragen.

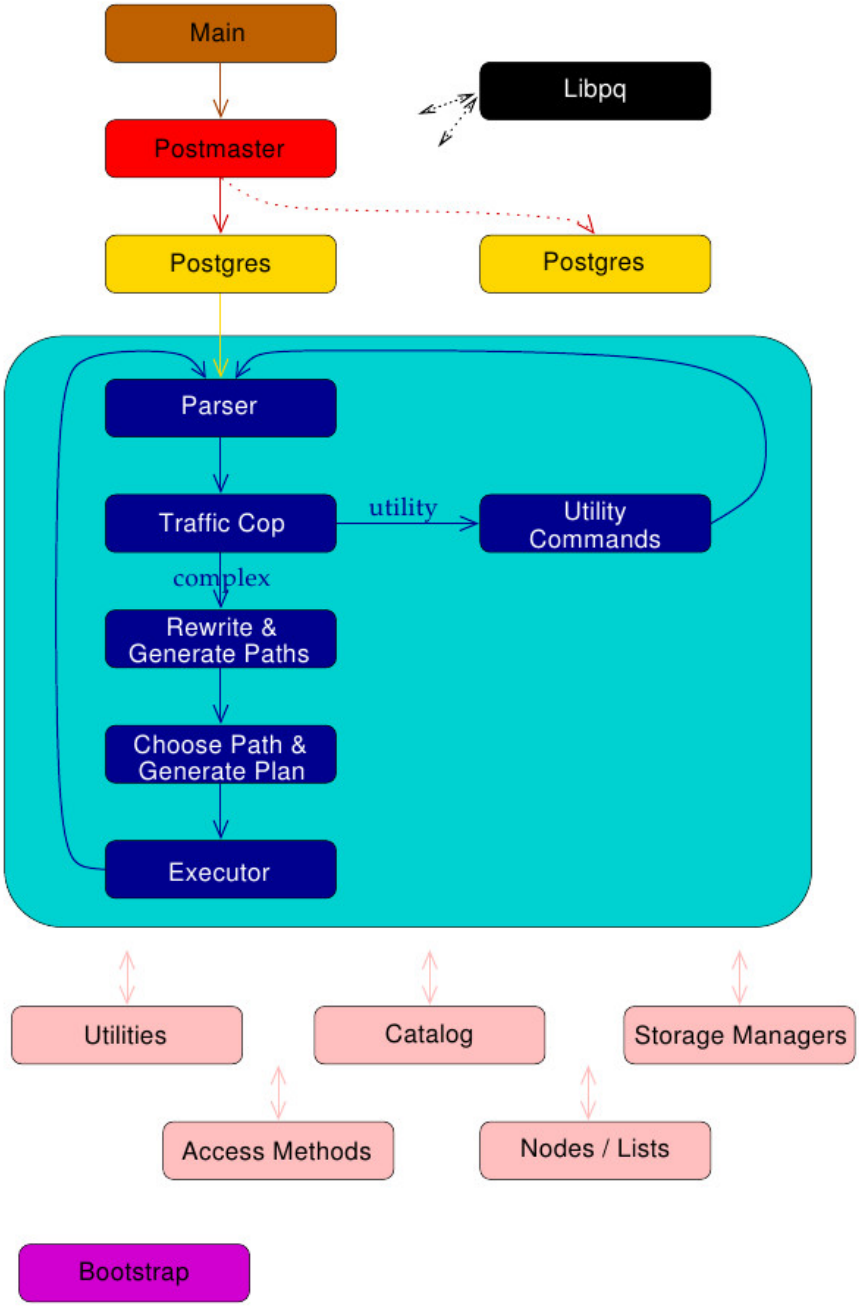


**Abbildung 2.2.:** Backend Flussdiagramm, angepasst nach [Momo1]

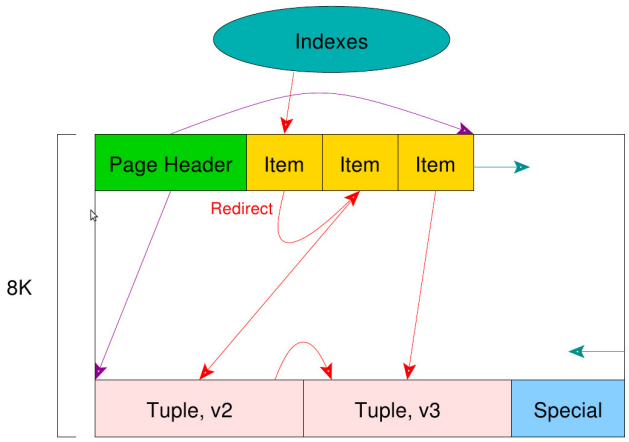


**Abbildung 2.3.:** Heapseitenbeispiel mit einer HOT Tupelkette, einem Redirect und einem Indexeintrag, angepasst nach [Mom12]

### 2.1.3. Segmente

Jeder Index und jede Tabelle wird in PostgreSQL als Array von Segmenten fester Größe abgespeichert. Die Größe der Segmente wird zur Übersetzungszeit festgelegt und beträgt üblicherweise 8 Kilobyte. [Gro09, vgl. K. 53.5, S. 1520 f.] Die einzelnen Segmente sind als Heapseiten organisiert, wie es in Abbildung 2.3 zu sehen ist. Das bedeutet, die Seiten werden strukturiert in zwei Richtungen befüllt.

Vom vorderen Ende nach hinten werden Metainformationen geschrieben. Zuallererst der Seitenheader, der Informationen über den Zustand der Seite enthält: Wann die Seite von welcher Transaktion zuletzt geändert wurde und wieviel freier Platz in der Seite noch enthalten ist. Danach folgt für jedes enthaltene Tupel ein Tupelheader mit den Informationen, von welcher Transaktion es erzeugt (xmin) und von welcher es gelöscht oder geändert wurde (xmax). Des weiteren der Typ der zuletzt ändernden Transaktion, bei einem geänderten Tupel die Referenz auf das nächste gültige Tupel und verschiedene Bitmasken, die weitere Informationen über den Zustand des Tupels enthalten.

Vom hinteren Ende nach vorn wird zuerst ein spezieller Teil geschrieben, der für unterschiedliche Typen von Heapseiten jeweils passend konkretisiert werden kann. Schließlich folgen die konkreten Tupeldaten. In der Mitte der Seite bleibt der freie Platz, der folglich immer kleiner wird. Wenn die Obergrenze der Metainformationen und die Untergrenze der Tupeldaten aufeinanderstoßen, muss eine neue Heapseite erzeugt und dort geschrieben werden.

Tupel werden in PostgreSQL indirekt mithilfe eines TID referenziert. Jeder TID ist eindeutig auf einer Tabelle. Er enthält zum einen die Seitennummer der Heapseite und zum anderen einen Offset, wo genau in der Heapseite der Tupelheader zu finden ist.

### 2.1.4. Multi Version Concurrency Control

Transaktionen sind allgemein Gruppen von Operationen auf der Datenbank, die atomar als Ganzes entweder committet, also für gültig und abgeschlossen erklärt oder aber als Ganzes rückgängig gemacht werden müssen. Jede Transaktion erhält eine Nummer, die TransaktionsID, die Datenbankweit eindeutig ist und fortwährend hochgezählt wird. Eine weitere notwendige Eigenschaft einer Transaktion ist die Isolation. Das heißt, nebenläufig - oder englisch concurrent - ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen. Zur Umsetzung dieses Prinzips gibt es verschiedene Möglichkeiten. Die einfachste Variante besteht darin, zu ändernde Tupel mit einer Schreibsperre zu versehen, die jedes weitere Lesen oder Schreiben verhindert, bis die Transaktion committet oder abgebrochen hat. Mit dieser Schreibsperre kann das Tupel in-place, also direkt überschrieben werden.

PostgreSQL verwendet an dieser Stelle ein anderes Prinzip namens Multi Version Concurrency Control (MVCC), um möglichst viele gegenseitige Blockierungen von Transaktionen zu vermeiden. Dabei wird das Tupel nicht direkt überschrieben, sondern die neue Version des Tupels wird an einer freien Stelle in der gleichen oder einer anderen Heapseite eingefügt. Im alten Tupel wird in-place eine Referenz auf die neue Version, genannt Current Tuple Identifier (CTID) gespeichert. Jede Tupelversion enthält außerdem die TransaktionsID der erzeugenden Transaktion sowie die der verändernden Transaktion. Das Löschen eines Tupels ist entsprechend dadurch realisiert, dass sie TransaktionsID der löschenden Transaktion im Tupel gespeichert wird.

Leseoperationen hangeln sich an der so entstandenen Kette von Tupeln entlang und überprüfen bei jeder Version den Status der erzeugenden und den der löschenden Transaktion. Hat die Transaktion, die eine Tupelversion erzeugt hat, committet, so ist das Tupel gültig. Sofern keine löschende Transaktion eingetragen ist, ist die neueste Version einer Tupelkette gefunden. Enthält das Tupel die TransaktionsID einer committeten ändernden Transaktion, so wird die eingetragene Referenz auf die nächste Tupelversion verfolgt und dieses entsprechend überprüft. Dieses Verfahren macht es unnötig, dass Leseoperationen von Schreiboperationen blockiert werden, denn auch nach der Änderung eines Tupels kann eine nebenläufige Leseoperation die aus ihrer Sicht gültige Version eines Tupels finden. Abbildung 2.4 zeigt eine entsprechende Kette von Tupeln.

In früheren Versionen von PostgreSQL wurde nun jede Tupelversion auf dem Index eingetragen. Leseoperationen, die einen Index einbeziehen, sogenannte Indexscans verfolgen dann keine Ketten, sondern schlagen mit jedem Indexeintrag genau eine Heapseiteneintrag nach. Das funktioniert deshalb, weil jede lesende Transaktion genau eine TransaktionsID hat und diese genau eine Tupelversion jeder Kette als die gültige identifiziert.

Eine weitere Optimierung besteht nun aber darin, nicht mehr jede Version eines Tupels auf dem Index einzutragen. Solange die vom Index geforderte Eigenschaft vor der Änderung wie nachher erfüllt ist, reicht ein Indexeintrag pro Tupelkette aus. Indexscans verfolgen dann die Ketten bis zur gültigen neuesten Version. Dieses Verfahren ist sinnvoll, solange eine neuere Tupelversion auf der gleichen Heapseite liegt. Denn würde der Indexscan Tupelketten über verschiedene Heapseiten hinweg verfolgen, so müssten dabei häufig zufällige Heapseiten

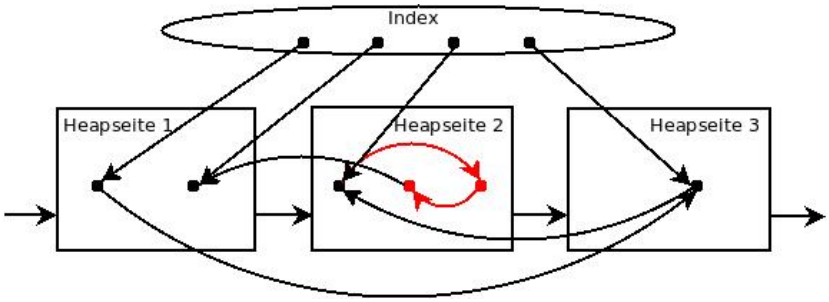


**Abbildung 2.4.:** Beispiel einer Tupelkette, die sich über mehrere Heapseiten erstreckt. Ein Teil davon ist HOT geupdatet und rot gekennzeichnet.

von der Festplatte gelesen werden, was sehr teuer ist. Deshalb verfolgt der Indexscan der verwendeten Version von PostgreSQL Tupelketten, solange sie auf der gleichen Heapseite liegen. Für die Versionen auf einer anderen Heapseite gibt es einen weiteren Indexeintrag. Dieser ist also genau für das jeweils erste Tupel einer Teilkette auf einer Heapseite nötig. Der TID dieses ersten Tupels wird auch ROOTTID genannt. Ab der zweiten Tupelversion einer solchen Teilkette werden die Tupel als Heap Only Tuple (HOT) markiert, da sie nur auf dem Heap auftauchen, aber nicht selbst im Index eingetragen sind. Bei einer Update - Operation wird immer zuerst versucht, das Tupel HOT, also innerhalb derselben Heapseite zu ändern. Nur dann, wenn dies nicht möglich ist, wird das Folgetupel in eine andere Heapseite geschrieben.

### 2.1.5. VACUUM

Tatsächlich physisch gelöscht wird eine ungültige Tupelversion erst dann, wenn sichergestellt ist, dass kein nebenläufiges Backend sie als gültige neueste Version einer Tupelkette ansehen würde. Zu diesem Zweck gibt es VACUUM, die PostgreSQL eigene automatische Speicherbereinigung. Sie blendet die vorderen ungültigen Tupelversionen von Teilketten innerhalb einer jeden Heapseite aus, indem sie im Seitenheader einen Redirect vom ersten TID zur ersten Tupelversion einfügt, die von einem beliebigen nebenläufigen Backend als neueste gesehen wird. Abbildung 2.3 enthält einen solchen Redirect. Eventuell werden von VACUUM auch ganze Heapseiten neu geschrieben.

### 2.1.6. Shared Memory

Der Sinn einer Datenbank besteht darin, es zu ermöglichen, dass viele Nutzer und Anwendungen gleichzeitig auf derselben Datenbasis arbeiten. Dabei bestehen innerhalb der Datenbank viele Konflikte, die geeignet gehandhabt werden müssen. Gleichermaßen ergeben sich aber auch viele Synergieeffekte durch die gemeinsame Nutzung derselben Strukturen und konkret derselben Instanzen dieser Strukturen von mehreren Prozessen. Sie liegen im gemeinsamen Speicherbereich, auch Shared Memory genannt.

| | LW_SHARED | LW_EXCLUSIVE |
|---|---|---|
| LW_SHARED | + | – |
| LW_EXCLUSIVE | – | – |

**Abbildung 2.5.:** Kompatibilitätsmatrix der LWLocks

### 2.1.7. Hashmap

Eine mögliche Struktur, um Daten geordnet zu speichern ist die Hashmap. Eine Hashmap ist ein Array von Zeigern variabler oder fester Größe. Für eine konkrete Hashmap-Instanz wird eine geeignete Hash-Funktion zur Verfügung gestellt, die für einen gegebenen Datensatz einen kurzen, eindeutigen Hash-Wert berechnet. Der Hash-Wert bestimmt die Position, an der der Datensatz im Array eingefügt wird. Da alle zugreifenden Prozesse die konkrete Hash-Funktion kennen und diese eindeutig ist, kann jeder so gespeicherte Datensatz wiedergefunden werden. PostgreSQL verwendet an vielen Stellen Hashmaps. Wird eine Hasmap im Shared Memory angelegt, so sollte sie eine feste Größe haben.

### 2.1.8. Sperren

PostgreSQL stellt Sperren auf verschiedenen Ebenen zur Verfügung, die hier besprochen werden sollen.

#### Leichtgewichtige Sperren

Als einfachste Sperre stellt PostgreSQL leichtgewichtige Sperren, sogenannte LWLocks zur Verfügung. Sie realisieren eine simple Lese-Schreib-Sperre, wie die Kompatibilitätsmatrix in Abbildung 2.5 zeigt. Eine Lesesperre, konkret LW_SHARED genannt, erlaubt anderen nebenläufigen Prozessen, dieselbe Lesesperre zu erhalten. Eine Schreibsperre, konkret LW_EXCLUSIVE genannt, schließt es aus, dass andere nebenläufige Prozesse dieselbe Sperre im Lese- oder Schreibmodus erhalten. Kann eine Sperre nicht gewährt werden, so wird gewartet, bis sie verfügbar ist. Erst dann wird sie erteilt.

Zur Identifizierung der einzelnen Sperren wird eine Nummer an die Funktionen LWLockAcquire und LWLockRelease übergeben. Dabei sind die ersten Nummern 1 bis 32 fest vergeben. An dieser Stelle kann man auch weitere einzelne Sperrnummern fest vergeben, was in dieser Arbeit verwendet wird. Darüber folgen die Nummern der Puffersperren, die benötigt werden, um die einzelnen Heapseiten im Bufferpool zu sperren.

| Sperrtyp | vom System angefordert für | steht in Konflikt mit |
|---|---|---|
| 1 AccessShareLock | SELECT | 8 |
| 2 RowShareLock | SELECT FOR UPDATE | 7,8 |
| 3 RowExclusiveLock | UPDATE, INSERT, DELETE | 5,6,7,8 |
| 4 ShareUpdateExclusiveLock | | 4,5,6,7,8 |
| 5 ShareLock | CREATE INDEX | 3,4,6,7,8 |
| 6 ShareRowExclusiveLock | | 3,4,5,6,7,8 |
| 7 ExclusiveLock | | 2,3,4,5,6,7,8 |
| 8 AccessExclusiveLock | DROP TABLE, ALTER TABLE, VACUUM | all |

**Tabelle 2.1.:** logische Sperrtypen in Postgresql frei nach [Lanoo, S. 18], aktualisiert nach [Gro12]

#### Puffersperren

Der Puffer liegt im Shared Memory der Datenbank. Jede Heapseite wird nur einmal gleichzeitig in den Puffer geladen. Alle nebenläufigen Backends, die dieselbe Heapseite lesen wollen, erhalten Zugriff auf ebendenselben Speicherbereich. Somit spart die Datenbank langsame Zugriffe auf die Festplatte ein. Jedoch wird es notwendig, diesen Speicherbereich vor unkoordinierten gleichzeitigen Zugriffen zu schützen. Puffersperren schützen daher die physische Integrität einer Heapseite im Puffer. Sie werden in der vorgelegten Arbeit immer dann verwendet, wenn davon die Rede ist, Heapseiten zu sperren.

Puffersperren werden mithilfe der Funktion LockBuffer angefordert und freigegeben, wobei die Sperrmodi BUFFER_LOCK_SHARE und BUFFER_LOCK_EXCLUSIVE zur Verfügung stehen. Sie sind auf Basis der LWLocks realisiert. Dabei wird der Sperrmodus BUFFER_LOCK_SHARE auf LW_SHARED abgebildet. Ein BUFFER_LOCK_EXCLUSIVE wird mithilfe eines LW_EXCLUSIVE gesetzt.

#### Logische Sperren

Tabelle 2.1 zeigt die logischen Sperrtypen von PostgreSQL und ihre Verwendung. Abbildung A.1 zeigt die konkrete Kompatibilitätsmatrix ausführlich. Sie werden verwendet, um die logische Integrität der Daten zu schützen und daher jeweils erst zum Ende einer Transaktion wieder freigegeben, also entweder bei einem Transaktionsabbruch oder einem Commit.

Der Reorganisationsprozess nutzt nur minimale logische Sperren, da er keine Daten logisch verändert. Er öffnet die zu transformierenden Tabellen mit einem AccessShareLock, um ein nebenläufiges Löschen der Tabelle, Ändern der Schema-Information oder Vacuum zu verhindern.

**Ausschnitt 2.1** Befehlssyntax im Umfeld von Tenants und Tenantspaces

```
CREATE TENANTSPACE <tenantspacename> ;
DROP TENANTSPACE [IF EXISTS] <tenantspacename> ;

SPLIT TENANTSPACE <tenantspacename> INTO <tenantspacename>, <tenantspacename> ;
MERGE TENANTSPACE <tenantspacename>, <tenantspacename> INTO <tenantspacename> ;

CREATE TENANT <name> IN TENANTSPACE <name> ;
DROP TENANT <name> ;

SET TENANT <name> ;
UNSET TENANT ;
SHOW TENANT ;

CREATE TABLE <name> <schema-description> SEGREGATED BETWEEN TENANTS ;
```

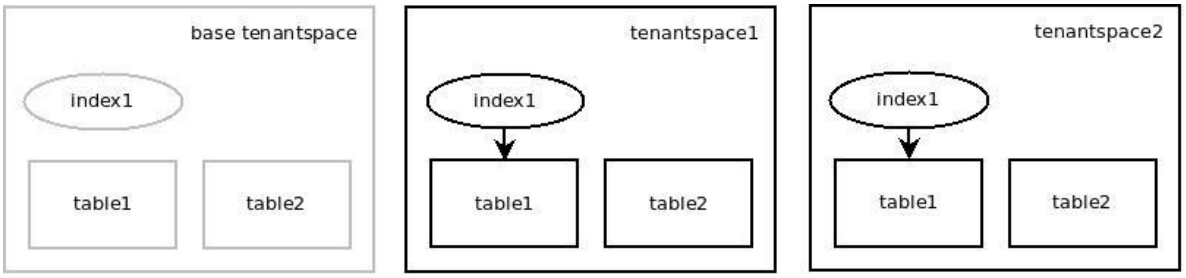


**Abbildung 2.6.:** Tenantspaces Beispiel, frei nach [Sch11]

## 2.2. Tenants und Tenantspaces

Wie in Kapitel 1.1 schon beschrieben, wird mit dem bereits umgesetzten Konzept der Tenantspaces pro Tenantspace eine Instanz jeder Tabelle erzeugt. Außerdem wird eine Instanz der Tabelle im Base Tenantspace erzeugt, die immer dann verwendet wird, wenn eine Anwendung keinen Mandanten benannt hat. Abbildung 2.6 stellt dies dar. Des weiteren erhält der Tupelheader die Information, welchem Mandanten ein konkretes Tupel zuzuordnen ist, in Form einer TenantID. Diese ist Datenbankweit eindeutig.

Der Sinn der Tenantspaces besteht darin, die Daten eines Mandanten oder einer Mandantengruppe als Einheit mit wenigen Befehlen bearbeiten zu können. Ausschnitt 2.1 zeigt die verfügbaren Kommandos zur Verwaltung von Mandanten, eine Erweiterung der SQL-Syntax. Sie wurden in [Sch11] ausführlich erläutert, sprechen für sich und bedürfen daher keiner weiteren Erläuterung. Die vorgelegte Arbeit behält sie bei und verändert lediglich die Implementierung der Algorithmen hinter den Operatoren SPLIT und MERGE.

| **Quellrelation** $X$ |
|---|
| $\mid A_1$ -> $A_2$ -> $A_3 \mid B_1$ -> $B_2$ -> $B_3$ -> $B_4 \mid C_1$ -> $D_2^d \mid (D_1^i) \mid$ |
| **Zielrelation** $Y$ |
| $\mid A'_1$ -> $A'_2$ -> $A'_3 \mid D'^i_1$ -> $D'^{ud}_2 \mid$ |
| **Zielrelation** $Z$ |
| $\mid B'_1$ -> $B'_2$ -> $B'_3$ -> $B'_4$-> $B'^u_5$ -> $B'^u_6 \mid$ |

**Tabelle 2.2.:** Bezeichner der Versionen einer Kette von Tupeln während der Transformation auf der Quell- und Zielrelation am Beispiel eines SPLIT-Vorgangs

## 2.3. Beispiel

In der weiteren Ausarbeitung werden die Prinzipien und Strukturen an diesem Beispiel erläutert. Tabelle 2.2 stellt vereinfacht eine SPLIT-Operation von Relation $X$ zu den Relationen $Y$ und $Z$ dar. Die MERGE-Operation vereinigt entsprechend umgekehrt die Relationen $X$ und $Y$ zu Relation $Z$. Zu beachten ist, dass die Tupelversionen in der Realität natürlich nicht sortiert auf den Heapseiten abgelegt werden, weshalb dieses Bild eine starke Vereinfachung ist. In der Datenbank können Folgetupel auf derselben oder einer anderen Heapseite liegen und wenn sie in der gleichen Heapseite liegen, so entweder vor oder hinter dem Tupel.

Die Legende sieht aus wie folgt:

- Die Großbuchstaben $A - M$ identifizieren eine Tupelkette.
- Die Großbuchstaben $N - Z$ identifizieren Relationen, wobei verschiedene Tenantspace - Instanzen derselben Tabelle verschiedene Buchstaben erhalten.
- Ein Apostroph ’ identifiziert eine transformierte Tupelversion auf der neuen Relation.
- Eine Zahl identifiziert eine Tupelversion.
- Die Kleinbuchstaben $u$,$i$ und $d$ identifizieren Tupelversionen, die durch ein zur Transformation nebenläufiges Update oder Insert entstanden sind oder von einem nebenläufigen Delete als gelöscht markiert wurden. Tupelversionen ohne Kleinbuchstaben wurden während der Transformationsoperation nicht von anderen Transaktionen geändert.
- Der Kleinbuchstabe $n$ steht allgemein für Versionsnummern von Tupeln.
- In Klammern geschriebene Tupel sind physisch nicht auf der Relation vorhanden. Sie dienen dem Algorithmus als Platzhalter auf der Quellrelation, sobald das physische Tupel auf die Zielrelation geschrieben ist.

# 3. Konzept

## 3.1. Überblick

Die Aufgabe besteht nun darin, Einheiten von Tenantspaces aufzuspalten oder zu vereinigen. Die Vereinigung ist eindeutig, denn es werden mehrere Mengen von Tabellen und Indizes zu einer vereinigt, wobei für jede allgemein definierte Tabelle die konkreten Instanzen aus jedem zur Vereinigung benannten Tenantspace zu einer Tabelle im neuen Tenantspace zusammengelegt werden. Desgleichen wird mit jedem Index verfahren. Um die Algorithmen überschaubar zu halten, wird die Anzahl der mit einer Operation vereinigbaren Mengen auf zwei eingeschränkt. Der entsprechende Operator wird im folgenden MERGE genannt. Alle Vereinigungen von mehr als zwei Tenantspaces sind durch mehrere aufeinanderfolgende MERGE-Operationen ausdrückbar.

Die entgegengesetzte Operation ist die Aufspaltung, auch SPLIT genannt. Dabei wird ein Tenantspace zu mehreren aufgespalten. Es stellt sich die Frage, nach welchen Kriterien gespalten werden soll, sprich wie die Verteilung der Mandanten auf die erzeugten Tenantspaces konkret definiert wird. Das ist in vielen Konstellationen denkbar, welche jedoch nicht Thema dieser Arbeit sein sollen, da die Mechanismen beim Vorgang Aufspaltung unabhängig von der Anzahl oder Größe der zu erzeugenden Tenantspaces die gleichen sein werden. Deshalb wird eine sehr einfache Splitstrategie verwendet, die die Tenants ohne Gewichtung auf den Zieltenantspaces gleichverteilt. Entsprechend umgekehrt zum MERGE wird jede Tabelle und jeder Index vom Quelltenantspace auf den Zieltenantspaces mit der gleichen Tabellen- oder Indexdefinition erzeugt. Jede Tabelle wird sequentiell eingelesen und Tupel für Tupel wird entsprechend der gespeicherten Mandantenverteilung auf der zugehörigen Tabelle im Zieltenantspace einsortiert. Desgleichen wird mit den Indexeinträgen verfahren. Wie beim MERGE wird die Anzahl der durch den SPLIT zu erzeugenden Tenantspaces auf zwei beschränkt, um die Algorithmen überschaubar zu halten.

Es ergibt sich also verallgemeinert, dass sowohl bei der SPLIT- als auch bei der MERGE-Operation alle Tabellen und Indizes von den Quelltenantspaces eingelesen werden und der jeweiligen Operation entsprechend auf den Zieltenantspaces neu geschrieben werden müssen. Im weiteren wird deshalb zumeist allgemein von Quelltabelle oder auch Quellrelation und Zieltabelle, beziehungsweise Zielrelation gesprochen.

Abbildung 3.1 zeigt den Vorgang für SPLIT und MERGE auf der Ebene der Reorganisation einer Tabelle und der zugehörigen Indizes. Es wird nötig sein, die TID der einzelnen Tupel auf einer Quellrelation in effizienter Art und Weise auf TID in der entsprechenden Zielrelation abzubilden. Diese Abbildung muss so lange verfügbar sein, wie die Reorganisation der Tabelle und der zugehörigen Indizes andauert. Insbesondere muss diese Abbildung

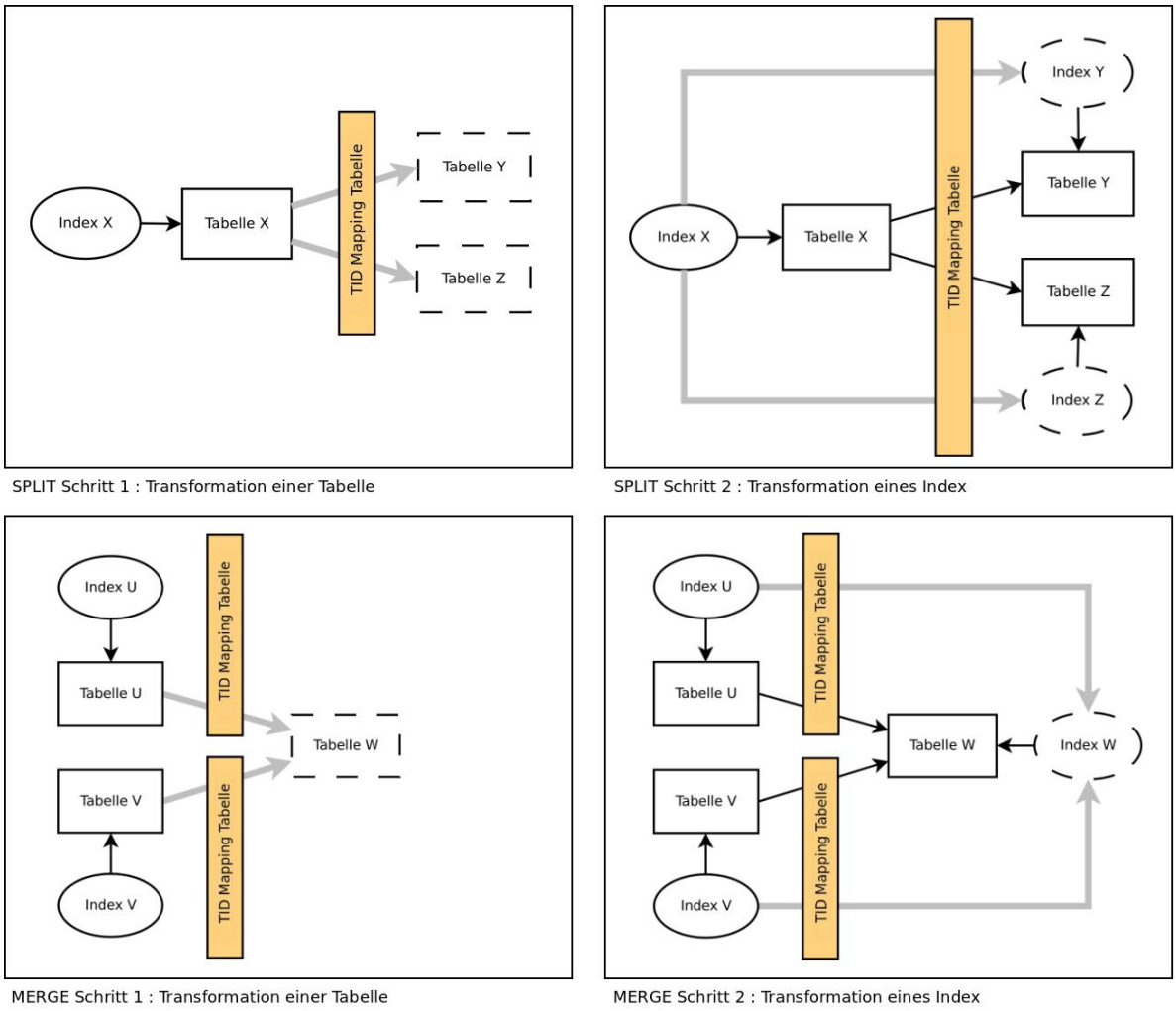


**Abbildung 3.1.:** Überblick zur Reorganisation: SPLIT und MERGE von Index und Tabelle

auch für nebenläufige Backends verfügbar sein, die als Bestandteil der Aufgabe reorganisierte Teile auf Seitenbasis verwenden können sollen. Außerdem muss diese Abbildung dauerhaft gespeichert werden, um nach einem eventuellen Absturz der Datenbank die Reorganisation fortführen zu können. Prinzipiell wäre es möglich, die TID der Reorganisierten Tupel im Header der Tupel auf der Quellrelation zu speichern, jedoch müsste man dazu diesen umdefinieren, was den Speicherverbrauch der gesamten Datenbank auch jenseits der Reorganisation beeinflussen würde. Um dies zu vermeiden, wird die Abbildung der TID als eigene Tabelle realisiert, die nach der Reorganisation einer jeden Quellrelation wieder gelöscht werden kann. Sie wird in der weiteren Ausführung Tidmap genannt und muss pro Quellrelation einmal angelegt werden, da jede TID wie schon beschrieben, nur auf einer Relation eindeutig ist.

In einer Datenbank, die Updates in-place realisiert, könnte man nun einfach die Tabellen sequentiell durchlaufen und Tupel für Tupel auf die Zielrelation kopieren. PostgreSQL verwendet jedoch, wie in Unterkapitel 2.1.4 beschrieben, das Prinzip der MVCC, wobei für jedes Tupel chronologisch eine Kette der vorher gültigen Tupelversionen bis zur derzeit gültigen Tupelversion gespeichert wird. Dabei können aus der Sicht von verschiedenen Transaktionen zum gleichen Zeitpunkt jeweils unterschiedliche Versionen eines Tupels gültig sein. Dieses Prinzip der Kette von Tupelversionen darf auch während des gesamten Reorganisationsvorgangs zu keinem Zeitpunkt gebrochen werden. Das heisst, zu jedem Zeitpunkt der Reorganisation muss jede Version eines Tupels für jedes nebenläufige Backend entweder auf der Quellrelation oder auf der Zielrelation sichtbar sein.

Man könnte auch für jeden Indexeintrag die entsprechende HOT Tupelkette lesen und als Ganzes in die Zielrelation schreiben. Der Scan auf den Quellrelationen soll jedoch möglichst effizient erfolgen und es müssen für die Reorganisation sowieso alle Tupel gelesen werden. Deshalb werden die Heapseiten sequentiell in aufsteigender Folge eingelesen. Auch innerhalb einer Heapseite werden die einzelnen Tupel nicht geordnet nach Tupelversionen von Ketten, sondern geordnet nach aufsteigenden Offsetnummern gelesen, sprich jede Heapseite wird so wie sie ist von vornnach hinten gelesen. Teilketten, die vor der Reorganisation HOT waren, sollen diesen Status auch nach der Reorganisation beibehalten. Um sicherzustellen, dass eventuelle HOT Teilketten vor dem Einfügen auf der Zielrelation komplett gelesen sind, werden Heapseiten als Ganzes eingelesen, bevor versucht wird, so viele Tupel wie möglich auf der Zielrelation einzufügen.

Das Einfügen der Tupel auf der Zielrelation soll ebenfalls möglichst effizient erfolgen. Außerdem sollen nebenläufige Transaktionen auf den momentan reorganisierten Daten effizient möglich sein. Würde man die Tupelketten nun von vorne nach hinten auf der Zielrelation einfügen - eben in der Reihenfolge, in der auch die Tupel auf der Quellrelation geschrieben wurden, so müssten nebenläufige Transaktionen jedes gefundene Tupel auf der Zielrelation zusätzlich auf der Quellrelation weiterverfolgen, da anders nicht entscheidbar wäre, ob eine Tupelkette schon fertig traversiert ist oder nicht. Deshalb wird die Regel eingeführt, dass für jedes schon reorganisierte Tupel alle folgenden Tupelversionen ebenfalls auf der Zielrelation zu finden sein müssen. Denn mit Einführung dieser Regel ergibt sich der nützliche Umstand, dass nebenläufige Leseoperationen nur jede Tupelversion von der Quellrelation in der Tidmap nachschlagen müssen. Sobald für eine Tupelversion ein Tidmapeintrag gefunden wird, kann das Lesen der zugehörigen Kette auf der Zielrelation fortgesetzt werden und ab diesem Zeitpunkt unterscheidet sich die Komplexität einer Leseoperation auf gerade reorganisierten Tabellen nicht von der auf normalen Tabellen.

Aus den genannten Gründen werden Tupelketten auf der Zielrelation in umgekehrter Reihenfolge geschrieben, also immer die jeweils neueste Tupelversion zuerst und die älteste noch verfügbare zuletzt. Jedes zu transformierende Tupel muss daher beim Schreiben in die Zielrelation nur einmal bearbeitet werden, da die TID des Folgetupels zu diesem Zeitpunkt schon bekannt ist.

Da bei der Erzeugung eines neuen Index die TID der neu geschriebenen Tupel schon bekannt sein müssen, werden jeweils zuerst die Tabelle und dann die eventuell vorhandenen Indizes transformiert.

| **oldtid** | TID der Tupelversion in der Quellrelation, zugleich Schlüsselelement der Tabelle |
|---|---|
| **newtid** | TID der Tupelversion in der Zielrelation |
| **oldctid** | CTID der Tupelversion in der Quellrelation, respektive TID der Folgeversion |
| **newctid** | CTID der nächsten Tupelversion in der Zielrelation, respektive TID der Folgeversion |
| **isTransformed** | Information, ob die Tupelversion schon transformiert ist |

**Tabelle 3.1.:** die Spalten der Tidmap

Um nebenläufige Transaktionen zu ermöglichen, werden die Operatoren SELECT, INSERT, DELETE und UPDATE so angepasst, dass sie erkennen, ob eine Tabelle gerade reorganisiert wird und, dass sie abhängig vom Fortschritt der Reorganisation mit den reorganisierten Daten arbeiten.

## 3.2. Tidmap

Die Tidmap realisiert, wie eingangs erwähnt, die Abbildung von auf der Quellrelation gültigen TID in TID auf der Zielrelation. Sie wird persistent gespeichert und ist auch für andere Backends erreichbar. Tabelle 3.1 beschreibt ihre Komponenten. Das Schlüsselelement ist oldtid. Für jedes vom Reorganisationsvorgang bereits eingelesene Tupel wird der auf der Quellrelation gültige TID gespeichert, der zugleich das Schlüsselelement der Tabelle darstellt. Außerdem wird der TID des eventuellen Folgetupels, genannt CTID gespeichert. Nicht bekannte Tupel Identifier werden durch einen invaliden TID repräsentiert. Sobald verfügbar, wird die gleiche Information bezüglich der Zielrelation gespeichert. Des weiteren wird als boolean gespeichert, ob das Tupel bereits transformiert ist oder nicht.

## 3.3. Tupelcache

Die neueste Version einer Tupelkette kann bei der Transformation jeweils direkt geschrieben werden. Für alle anderen Tupel einer Kette muss in der Tidmap nachgeschlagen werden, ob die transformierte TID des Folgetupels schon bekannt ist. Wenn ja, so kann es geschrieben werden. Wenn nicht, so muss es zwischengespeichert und später bearbeitet werden, wenn das Folgetupel transformiert wurde. Zu diesem Zweck wird ein spezieller Speicher, genannt Tupelcache, angelegt. Da diese Tupel noch nicht als transformiert gelten, muss auch kein anderes Backend darauf zugreifen können. Ein Tupel muss im Tupelcache nur solange verbleiben, bis es in die Zielrelation geschrieben wurde und es werden jeweils nach dem Lesen einer Heapseite so viele Tupel wie möglich in die Zielrelation übertragen. Der Tupelcache wird also nicht viel größer als eine Heapseite werden. Deshalb kann der Tupelcache

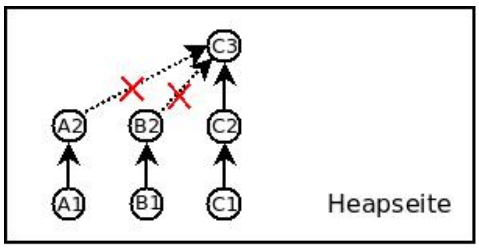


**Abbildung 3.2.:** korrupte Tupelketten

komplett im Hauptspeicher liegen und muss nicht vor nebenläufigen Zugriffen geschützt werden.

In speziellen Fällen, nämlich dann, wenn sich eine Tupelkette über mehrere Heapseiten erstreckt und eine neuere Tupelversion vor einer älteren vom Reorganisationsprozess gelesen wird, kann es sein, dass einzelne Tupel über die Reorganisation einer Heapseite hinaus im Tupelcache verbleiben. Sie werden dann bei der Reorganisation der Heapseite mitgenommen, die die neuere Tupelversion enthält.

Da der Tupelcache dazu dienen soll, Ketten rückwärts zu verfolgen, ist das Schlüsselelement nicht die TID eines Tupels, sondern die CTID. An dieser Stelle ist es wichtig, zu garantieren, dass keine Tupelketten vermischt werden. Denn prinzipiell ist es möglich, dass ein Tupel einen ungültigen CTID enthält, wie es in Abbildung 3.2 dargestellt ist. Im dargestellten Fall sollten nacheinander $A_2$, $B_2$ und $C_2$ geändert werden. Die Transaktionen, die $A_2$ und $B_2$ geändert haben, wurden abgebrochen. Der geänderte CTID verbleibt dabei im jeweiligen Tupel. Der verwendete Speicherplatz in der Heapseite kann jedoch von folgenden Transaktionen wiederverwendet werden. Die Transaktionen, die $C_2$ ändert, beschreibt nun den entsprechenden Platz in der Heapseite und comittet. Die CTID in $A_2$ und $B_2$ werden in der Folge dadurch als ungültig erkannt, dass ihre ändernde TransaktionsID als abgebrochen identifiziert wird. Im beschriebenen Fall dürfen die Tupelversionen $A_2$ und $B_2$ nicht im Tupelcache gespeichert werden, denn sie gehören nicht zur Versionskette von Tupel $C$.

## 3.4. Transformation - SPLIT und MERGE

Auf der Ebene der Tenantspaces ist die Transformation klar. Im dem Systemkatalog der Datenbank werden die Tabellen gesucht, die in dem gegebenen Tenantspace enthalten sind. Für jede enthaltene Tabelle wird zuerst die Tabelle transformiert und dann die zugehörigen Indizes. Auf der Ebene der Tabellen werden wie schon beschrieben die Heapseiten in aufsteigender Reihenfolge eingelesen. Jeweils nachdem eine Heapseite fertig eingelesen ist, werden so viele Tupel wie möglich in die Zielrelation geschrieben.

Beim Einlesen der Heapseite werden zum einen alle neuesten Tupelversionen in einer Liste namens ChainHeads gespeichert. Zum anderen werden in dieser Liste alle Tupel gespeichert,

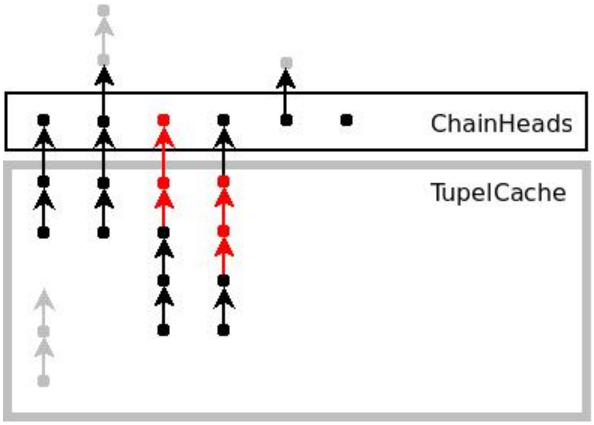


**Abbildung 3.3.:** Tupelketten aus der Sicht von FlushTupelCache

deren Folgetupel schon transformiert sind. Alle weiteren Tupel werden im Tupelcache zwischengespeichert.

### 3.4.1. FlushTupelCache

Wenn eine Heapseite fertig eingelesen ist, wird für jedes Tupel in ChainHeads der schon eingelesene Teil der Tupelkette im Tupelcache zurückverfolgt, die Tupelkette möglichst am Stück in die Zielrelation geschrieben und jedes transformierte Tupel entsprechend dem unteren Teil von Tabelle 3.2 in der Tidmap verzeichnet. Zu diesem Zweck wird die Funktion FlushTupelCache eingeführt. Abbildung 3.3 zeigt die Tupelketten aus der Sicht von FlushTupelCache. Teile von Ketten können HOT sein. Sie sind rot markiert. Wenn sich eine Kette über mehrere Heapseiten erstreckt, können Teile davon schon transformiert sein. Außerdem können Teile einer Kette noch nicht gelesen worden sein, nämlich alle die von Heapseiten mit höheren Nummern. Alle Tupel in ChainHeads hingegen stammen von der gerade gelesenen Heapseite. FlushTupelCache schreibt jeweils alle rot oder schwarz markierten Tupel in die Zielrelation.

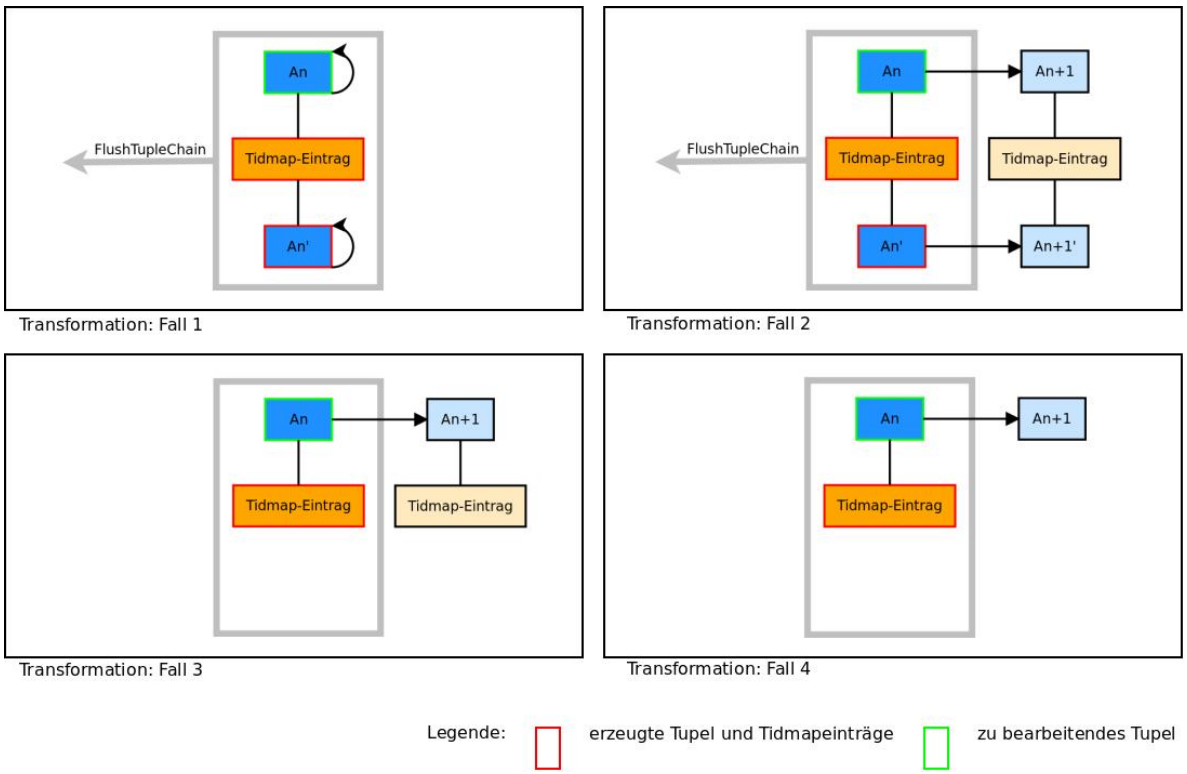


**Abbildung 3.4.:** Fallunterscheidung bei der Transformation von Tupelketten

### 3.4.2. Fallunterscheidung für jedes Tupel

Beim Einlesen einer Heapseite wird nun für jedes Tupel $A_n$ die in Abbildung 3.4 dargestellte Fallunterscheidung angewendet. Für jedes gelesene Tupel wird der entsprechende Tidmapeintrag aus Tabelle 3.2 erstellt.

**Tidmapeinträge beim Einlesen**

| | oldtid | newtid | oldctid | newctid | isTransformed |
|---|---|---|---|---|---|
| **Fall 1**<br>$A_n$.CTID $= A_n$ | $A_n$ | - | $A_n$ | - | false |
| **Fall 2**<br>$A_n$.CTID $\neq A_n$<br>$A_{n+1}$ ist bekannt<br>und transformiert | $A_n$ | - | $A_{n+1}$ | $A'_{n+1}$ | false |
| **Fall 3**<br>$A_n$.CTID $\neq A_n$<br>$A_{n+1}$ ist bekannt,<br>aber nicht transformiert | $A_n$ | - | $A_{n+1}$ | - | false |
| **Fall 4**<br>$A_n$.CTID $\neq A_n$<br>$A_{n+1}$ ist unbekannt | $A_n$ | - | $A_{n+1}$ | - | false |

**Tidmapeinträge beim Ausschreiben**

| | oldtid | newtid | oldctid | newctid | isTransformed |
|---|---|---|---|---|---|
| **Fall 5**<br>$A_n$.CTID $= A_n$<br>$A'_n$ ist erzeugt | $A_n$ | $A'_n$ | $A_n$ | $A'_n$ | true |
| **Fall 6**<br>$A_n$.CTID $\neq A_n$<br>$A_{n+1}$ ist bekannt<br>und transformiert<br>$A'_n$ ist erzeugt | $A_n$ | $A'_n$ | $A_{n+1}$ | $A'_{n+1}$ | true |

**Tabelle 3.2.:** Tidmapeinträge, die bei der Tansformation einer Tabelle erstellt werden

#### Fall 1: $A_n$ ist die neueste Tupelversion

Kriterien:

- $A_n$.CTID $= A_n$
- oder ( ( $A_n$.CTID $\neq A_n$ ) und ( $A_n$.xmax hat abgebrochen ) )

Wenn der CTID eines Tupels auf das eigene Tupel zeigt, oder die ändernde Transaktion abgebrochen hat, so ist es die neueste Version einer Kette. Das Tupel wird in zu ChainHeads hinzugefügt. In der Tidmap erfolgt vorerst der Eintrag Fall 1 aus Tabelle 3.2. FlushTupelCache wird das Tupel $A_n$ in die Zielrelation schreiben, wobei es eine dort gültige TID erhält. Damit wird es Fall 5 aus Tabelle 3.2 erfüllen und der entsprechende Tidmapeintrag kann erzeugt werden. Fortan wird FlushTupelCache die Versionskette von $A_n$ im Tupelcache zurückverfolgen und alle verfügbaren Vorgänger transformieren.

#### Fall 2: $A_n$ ist nicht die neueste Tupelversion und $A_{n+1}$ ist bekannt und transformiert

Kriterien:

- $A_n$.CTID $= A_{n+1}$
- Tidmap($A_{n+1}$) $\neq$ null
- Tidmap($A_{n+1}$).isTransformed $=$ true

Wenn die CTID eines Tupels $A_n$ auf ein Tupel $A_{n+1}$ zeigt, das schon bekannt, sprich in der Tidmap verzeichnet, und außerdem schon transformiert ist, so wird das Tupel $A_n$ ebenso zu ChainHeads hinzugefügt. In der Tidmap erfolgt vorerst der Eintrag Fall 2 aus Tabelle 3.2. FlushTupelCache wird das Tupel $A_n$ in die Zielrelation schreiben, wobei es eine dort gültige TID erhält. Damit wird es Fall 6 aus Tabelle 3.2 erfüllen und der entsprechende Tidmapeintrag kann erzeugt werden. Außerdem wird FlushTupelCache die Versionskette von $A_n$ im Tupelcache zurückverfolgen und alle verfügbaren Vorgänger transformieren.

#### Fall 3: $A_n$ ist nicht die neueste Tupelversion und $A_{n+1}$ ist bekannt aber nicht transformiert

Kriterien:

- $A_n$.CTID $= A_{n+1}$
- Tidmap($A_{n+1}$) $\neq$ null
- Tidmap($A_{n+1}$).isTransformed $=$ false

Wenn die CTID eines Tupels $A_n$ auf ein Tupel $A_{n+1}$ zeigt, dass schon gelesen wurde, sprich das in der Tidmap verzeichnet ist, das aber noch nicht transformiert ist, so wird es im Tupelcache zwischengespeichert. In der Tidmap erfolgt der Eintrag Fall 3 aus Tabelle 3.2.

FlushTupelCache wird das Tupel finden, wenn es die Versionskette von $A_{n+1}$ zurückverfolgt. Es kann nicht gesagt werden, wann genau dieser Fall eintritt. Er wird jedoch sicher eintreten, da per Definition alle Heapseiten gelesen werden.

Sobald also FlushTupelCache das Tupel $A_n$ findet, wird es transformiert. Damit wird es Fall 6 aus Tabelle 3.2 erfüllen und der entsprechende Tidmapeintrag kann erzeugt werden. Die Versionskette von $A_n$ wird von FlushTupelCache weiter zurückverfolgt und alle verfügbaren Vorgänger werden transformiert.

#### Fall 4: $A_n$ ist nicht die neueste Tupelversion und $A_{n+1}$ ist unbekannt

Kriterien:

- $A_n$.CTID $= A_{n+1}$
- Tidmap($A_{n+1}$) $=$ null

Wenn die CTID eines Tupels $A_n$ auf ein Tupel $A_{n+1}$ zeigt, dass wir noch nicht gesehen haben, so wird $A_n$ im Tupelcache zwischengespeichert. In der Tidmap erfolgt der Eintrag Fall 4 aus Tabelle 3.2.

FlushTupelCache wird das Tupel finden, wenn es die Versionskette von $A_{n+1}$ zurückverfolgt. Sofern $A_{n+1}$ auf einer anderen Heapseite liegt, wird dieser Fall auch erst beim entsprechenden späteren Durchlauf von FlushTupelCache eintreten. Er wird jedoch sicher eintreten, da per Definition alle Heapseiten gelesen werden. Die Tatsache, dass für $A_{n+1}$ noch kein Tidmapeintrag existiert, bedeutet, dass $A_{n+1}$ entweder in der gleichen Heapseite liegt und eine höhere Offsetnummer hat oder aber es liegt auf einer höheren Heapseite. Würde $A_{n+1}$ auf einer Heapseite vor der aktuell gelesenen stehen, so wäre an dieser Stelle Fall 2 oder Fall 3 eingetreten.

Sobald also FlushTupelCache das Tupel $A_n$ findet, wird es transformiert. Damit wird es Fall 6 aus Tabelle 3.2 erfüllen und der entsprechende Tidmapeintrag kann erzeugt werden. Die Versionskette von $A_n$ wird von FlushTupelCache weiter zurückverfolgt und alle verfügbaren Vorgänger transformiert.

### 3.4.3. Warten

Der Transformationsprozess muss jeweils auf der zu lesenden Heapseite in der Quellrelation eine Lesesperre und auf der aktuell zu schreibenden Heapseite in der Zielrelation eine Schreibsperre halten. Dabei werden auf beiden Relationen nebenläufige Schreiboperationen blockiert. Da die Heapseiten der Quellrelation in aufsteigender Reihenfolge gelesen werden und nebenläufige Leseoperationen sie ebenfalls aufsteigend lesen, könnte es sich ergeben, dass keine nebenläufige Transaktion das Lesen der Quellrelation abschließen kann, bevor der Transformationsprozess mit dem Einlesen der letzten Heapseite fertig ist. In diesem Fall wären nebenläufige Transaktionen sehr langsam. Um die nebenläufigen Transaktionen zu beschleunigen, wird auf Seite des Transformationsprozesses eine einstellbare Wartezeit eingeführt. Diese Wartezeit muss nach jedem Aufruf von FlushTupelCache verstreichen, um nebenläufige Backends überholen zu lassen. Zu diesem Zeitpunkt dürfen keine Sperren auf den Heapseiten, ob geteilt oder exklusiv, auf Quell- oder Zielrelation gehalten werden.

### 3.4.4. Ignorieren von Tupeln

Die Reorganisation kann zugleich einen reinigenden Effekt ähnlich wie VACUUM haben. Tote Tupel, also solche, die von keiner gegenwärtig laufenden Transaktion als gültig angesehen

würden, können weggelassen werden. Das sind zum einen Tupel, die mit den folgenden Eigenschaften als gelöscht gelten:

- Sie wurden als gelöscht markiert.
- Ihre xmax ist als committet bekannt.
- Ihre xmax ist kleiner als die kleinste aktuell laufende TransaktionsID.

Zum anderen sind es Tupel, die nie als gültig galten oder gelten werden, da ihre xmin, die TransaktionsID der erzeugenden Transaktion, abgebrochen hat. Der letztere Fall ist unkompliziert. Der erste Fall - gelöschte Tupel - trifft auch auf solche Tupel zu, für die mit einem UPDATE eine Folgeversion erstellt wurde.

### 3.4.5. Doppelte Tidmapeinträge

Wenn man solche geänderten Tupel weglässt, so muss darauf geachtet werden, dass sie Tupelkette auffindbar bleibt. Es muss also ein Tidmapeintrag erstellt werden, der den entsprechenden Indexeintrag auf das erste transformierte Folgetupel derselben Tupelkette abbildet. Dieser Tidmapeintrag muss nur dann erstellt werden, wenn das transformierte Folgetupel ursprünglich auf derselben Heapseite lag, denn für Tupelkettenteile auf anderen Heapseiten gibt es weitere Indexeinträge.

Es wird für also für Teiltupelketten einer Heapseite, in denen der vordere Teil weggelassen wird, ein Tupel geben, nämlich das erste transformierte, für das es zwei Tidmapeinträge gibt, die darauf zeigen: einmal der Tidmapeintrag, der bei der Transformation des Tupels selbst erzeugt wird und einmal der Tidmapeintrag für den ROOTTID.

Desgleichen muss mit Redirects verfahren werden, wie sie in Unterkapitel 2.1.5 erklärt wurden.

## 3.5. Indextransformation

Wenn eine Tabelle fertig gesplittet ist, werden auf den Zielrelationen die Indizes im gleichen Format wie auf den Quellrelationen angelegt und transformiert. Die Transformation der Indizes erfolgt nach dem gleichen Prinzip wie die der Tabellen. Die nötige Logik ist jedoch bedeutend einfacher, da hier keine Tupelketten verfolgt werden müssen. Jeder Indexeintrag wird in der Tidmap nachgeschlagen und der entsprechende Eintrag auf dem Zielindex erzeugt.

## 3.6. Nebenläufige Transaktionen

Um nebenläufige Transaktionen zu ermöglichen, die die in der Reorganisation befindliche Speicherstruktur berücksichtigen, werden die Operatoren SELECT, INSERT, DELETE und UPDATE so angepasst, dass sie erkennen, ob eine Tabelle gerade reorganisiert wird. Ist dies der Fall, so müssen sie entsprechend dem Fortschritt der Reorganisation die einzelnen Tupelketten über die Tidmap auflösen. Auf diese Weise sehen sie jeweils das neueste Tupel einer Kette. Um die Logik sinnvoll zu vereinfachen, ergeht die Regel, dass neue Tupelversionen ausschließlich auf der Zielrelation eingetragen werden. Das hat den Vorteil, dass schon am Anfang der Reorganisation das komplette zu reorganisierende Datenvolumen bekannt ist.

Die einzigen Änderungen, die an Tupeln auf der Quellrelation neben der Transformation noch erlaubt sein werden, sind das Löschen eines Tupels oder das Eintragen eines CTID, um das Auffinden eines Folgetupels zu ermöglichen. Diese Änderungen können in-place erfolgen.

### 3.6.1. Dummy-TID

Wenn neben der Reorganisation einer Tabelle eine logisch neue Tupelversion eingefügt werden soll, sprich bei nebenläufigen INSERT- oder UPDATE- Operationen, so werden diese auf der Zielrelation geschrieben. Da sie auf der Quellrelation niemals vorhanden waren oder sein werden, besteht das Problem, sie für weitere nebenläufige Transaktionen sichtbar zu machen. Zu diesem Zweck werden implizite sogenannte Dummy-TID eingeführt. Sie müssen auf der Quellrelation eindeutig sein, dürfen sich also nicht mit den TID von schon vorhandenen Tupeln überschneiden. Dies kann erreicht werden, indem zu Beginn der Transformation einer Tabelle die höchste auf ihr definierte TID gespeichert wird. Ein Zähler wird angelegt, der mit dieser höchsten TID initialisiert und bei jeder nebenläufigen Einfüge-Operation um eins erhöht wird. Mithilfe dieser Dummy-TID wird dann ein Eintrag sowohl in allen auf der Tabelle definierten Indizes als auch in der Tidmap erzeugt und somit bietet sich für nebenläufige Transaktionen fast das gleiche Bild wie bei fertig transformierten Tupelketten. Der einzige Unterschied besteht darin, dass das Tupel auf der Quellrelation tatsächlich nicht vorhanden ist. Die nebenläufigen Leseoperationen müssen deshalb so angepasst werden, dass das entsprechende Tupel auf der Quellrelation auch nie gelesen wird. Stattdessen muss jeder Dummy-TID immer gleich in der Tidmap aufgelöst werden. Dabei entsteht zugleich eine Optimierung des Leseoperators im Allgemeinen, denn es werden das Lesen einer Tupelversion und entsprechend dem Fortschritt der Reorganisation sogar Lesevorgänge von ganzen Heapseiten eingespart.

### 3.6.2. SELECT

Wenn man Lesevorgänge erlauben würde, die den Index nicht mit einbeziehen, wäre die Schwierigkeit, Tupel zu identifizieren, die nebenläufig eingefügt wurden. Würde man sie

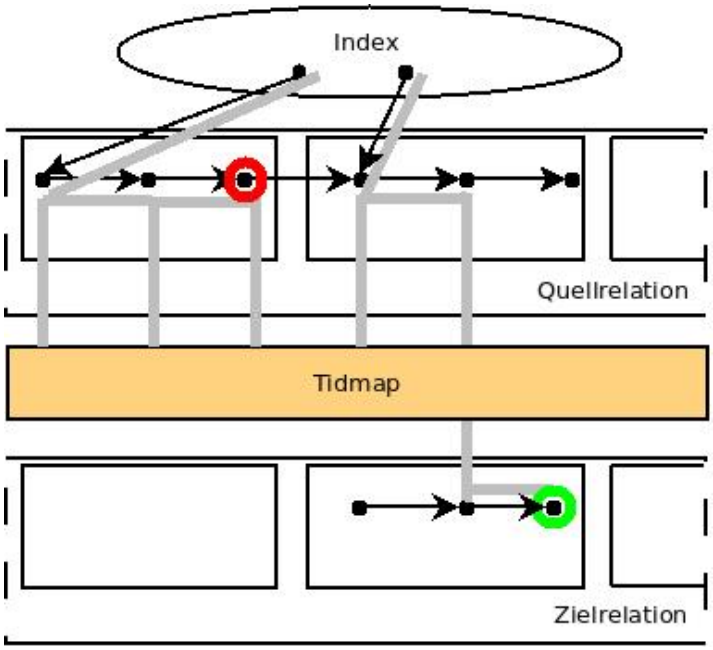


**Abbildung 3.5.:** Modifizierter Indexscan

einfach auf der Zielrelation suchen, so müsste man für jeden gefundenen Eintrag die Abbildung der Tidmap rückwärts verfolgen, um zu verifizieren, dass die gefundene Tupelkette eine nebenläufig eingefügte und keine transformierte ist. Die Tidmap ist aber gegenwärtig nur zur Suche in die eine Richtung von der Quellrelation zur Zielrelation ausgelegt. Eine andere Möglichkeit wäre die, auf der Tidmap alle Einträge von Dummy-TID zu suchen und die entsprechenden Tupel aus der Zielrelation zu lesen. Dann würde die Tidmap zum Teil als normaler Index fungieren. Diese Option wurde nicht implementiert.

Aus den genannten Gründen werden während der Transformation nur solche Lesevorgänge erlaubt, die den Index mit einbeziehen. Der Indexscan wird entsprechend angepasst. Abbildung 3.5 zeigt einen solchen modifizierten Indexscan. Jeder TID innerhalb einer Tupelkette auf der Quellrelation wird in der Tidmap gesucht. Sobald der aktuelle TID aufgelöst werden kann, wird das entsprechende Tupel nicht in der Quellrelation gelesen, sondern die Kette in der Zielrelation weiterverfolgt. In der Abbildung wird für den ersten Indexeintrag beim rot eingekreisten Tupel das Verfolgen der Kette abgebrochen, da das Tupel offensichtlich keine neueste Tupelversion ist und auch in der Tidmap nicht aufgelöst werden kann. Beim zweiten Indexeintrag kann ein TID aufgelöst werden und die entsprechende neueste Tupelversion wird von der Zielrelation gelesen.

#### Keine explizite Transformationsfront

In den Vorbereitungen zur Arbeit war überlegt worden, bei der Transformation eine Art Transformationsfront mitzuführen und zur Optimierung des Indexscans einzusetzen. Diese wäre einfach zu bestimmen, wenn man nach jedem Lesen einer Heapseite und Aufrufen von FlushTupleCache sagen könnte, diese Heapseite wäre fertig transformiert. Dies ist jedoch nicht der Fall, denn Tupel, deren Folgeversion auf einer folgenden Heapseite liegen, verbleiben im Tupelcache. Deshalb müsste man nach jedem Aufruf von FlushTupleCache die verbleibenden Tupel durchsuchen. Von dieser Menge von Tupeln, vereinigt mit der größten TID der letzten Heapseite müsste man die kleinste TID bestimmen. Diese wäre die TID, von der man sagen könnte, alle kleineren Tupel sind schon transformiert. Dann müsste man die entsprechenden kleineren TID gar nicht auf der Quellrelation nachschlagen. Der gleiche Effekt wird in der vorgelegten Implementierung dadurch erreicht, dass Tupel, die in der Tidmap verzeichnet sind, nicht von der Quellrelation gelesen werden. Deshalb wird eine explizite Front während der Transformation nicht bestimmt.

### 3.6.3. DELETE

Wie beim SELECT sind nur Indexscans erlaubt, um die Tabelle zu lesen. Abhängig vom Fortschritt der Transformation wird die neueste Tupelversion entweder auf der Quellrelation oder auf der Zielrelation gefunden. Dort, wo sie gefunden wird, wird sie auch gelöscht. Zu beachten ist dabei, dass Tupel in PostgreSQL, wie schon beschrieben, nicht direkt physisch gelöscht werden. Vielmehr werden sie nur als gelöscht markiert, indem die TransaktionsID der löschenden Transaktion im Tupel in-place eingetragen wird.

Es muss jedoch sichergestellt werden, dass der Transformationsprozess einen eventuellen nebenläufigen Löschvorgang nicht verpasst. Das wird erreicht, indem der DELETE eine exklusive Sperre auf dem Buffer, sprich auf der entsprechenden Heapseite hält, während er seine TransaktionsID in das Tupel einträgt. Somit kann es nicht passieren, dass der Transformationsprozess die selbe Seite zur gleichen Zeit liest. Auf die Sperren wird in Kapitel 3.7 noch näher eingegangen.

### 3.6.4. INSERT

Der zur Reorganisation nebenläufige INSERT wird immer auf der Zielrelation ausgeführt. Wie dies in Unterkapitel 3.6.1 erklärt wurde, wird ein auf der Quellrelation implizit gültiger Dummy-TID erzeugt. In der Tidmap wird mithilfe des Dummy-TID ein Eintrag entsprechend Tabelle 3.3 eingefügt. Außerdem muss ein Eintrag in allen auf der Quellrelation definierten Indizes mit der Dummy-TID gemacht werden.

**Tidmapeinträge nebenläufiger Transaktionen**

| | oldtid | newtid | oldctid | newctid | isTransformed |
|---|---|---|---|---|---|
| **INSERT** | $(A^i_n)$ | $A'^i_n$ | - | $A'^i_n$ | true |
| **UPDATE** | $(A^u_{n+1})$ | $A'^u_{n+1}$ | $(A^u_{n+1})$ | $A'^u_{n+1}$ | true |

**Tabelle 3.3.:** Tidmapeinträge, die bei nebenläufigen Transaktionen erstellt werden

### 3.6.5. UPDATE

Bei einem UPDATE wird wie beim Select die neueste Version jedes Tupels $A_n$ auf der Quellrelation oder $A'_n$ auf der Zielrelation gefunden.

Wenn die neueste Tupelversion auf der Zielrelation liegt, so wird sie dort HOT geupdatet. Es wird kein Indexeintrag und kein Eintrag in der Tidmap erzeugt. Falls das HOT UPDATE fehlschlägt, weil die Seite voll ist, wird die neueste Tupelversion $A'_n$ als gelöscht markiert und analog zum INSERT ein neues Tupel auf er Zielrelation eingefügt. Im Tupel $A'_n$ wird der TID des Folgetupels $A'^u_{n+1}$ gespeichert. Es wird ein Dummy-TID $(A^u_{n+1})$ auf der Quellrelation erzeugt und ein Indexeintrag sowie ein Eintrag in die Tidmap mit dieser Dummy-TID geschrieben.

Wenn die neueste Tupelversion $A_n$ auf der Quellrelation liegt, so wird diese als gelöscht markiert. Wie beim INSERT wird ein neues Tupel $A'^u_{n+1}$ erzeugt und in die Zielrelation geschrieben. Um das neue Tupel für nebenläufige Transaktionen sichtbar zu machen, wird eine Dummy-TID $(A^u_{n+1})$ auf der Quellrelation erzeugt, mit deren Hilfe ein Tidmapeintrag (siehe Tabelle 3.3) und ein Indexeintrag erstellt werden kann. Im Tupel $A_n$ auf der Quellrelation wird als CTID die Dummy-TID gespeichert.

## 3.7. Sperren

Da die Reorganisation keine Tupel auf logischer Ebene verändert, sondern nur deren physische Repräsentation, nimmt sie nur eine minimale Transaktionslevelsperre, nämlich den AccessShareLock. Dieser blockiert ein nebenläufiges VACUUM oder Löschen der Tabelle oder verändern der Tabellendefinition.

Zur Synchronisation werden leichtgewichtige Sperren und Puffersperren verwendet. Die Abbildungen 3.6, 3.7 und 3.8 zeigen die verwendeten Puffersperren. Ein **r** steht für eine Lesesperre, konkret einen SHARE Lock. Ein **w** steht für eine Schreibsperre, konkret einen EXCLUSIVE Lock.

Der Tupelcache wird nur vom Transformationsprozess geschrieben oder gelesen. Er muss daher nicht gelockt werden.

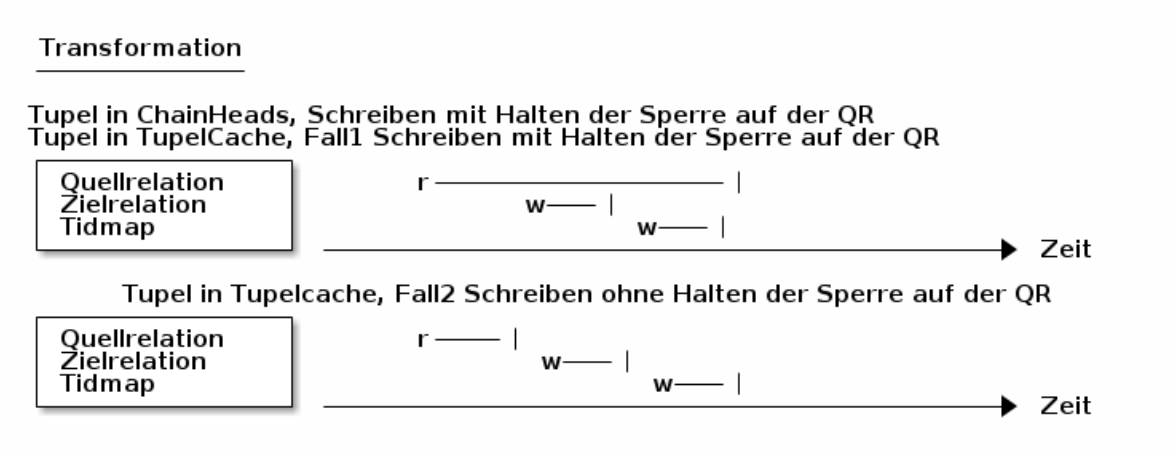


**Abbildung 3.6.:** Puffersperren der Transformation

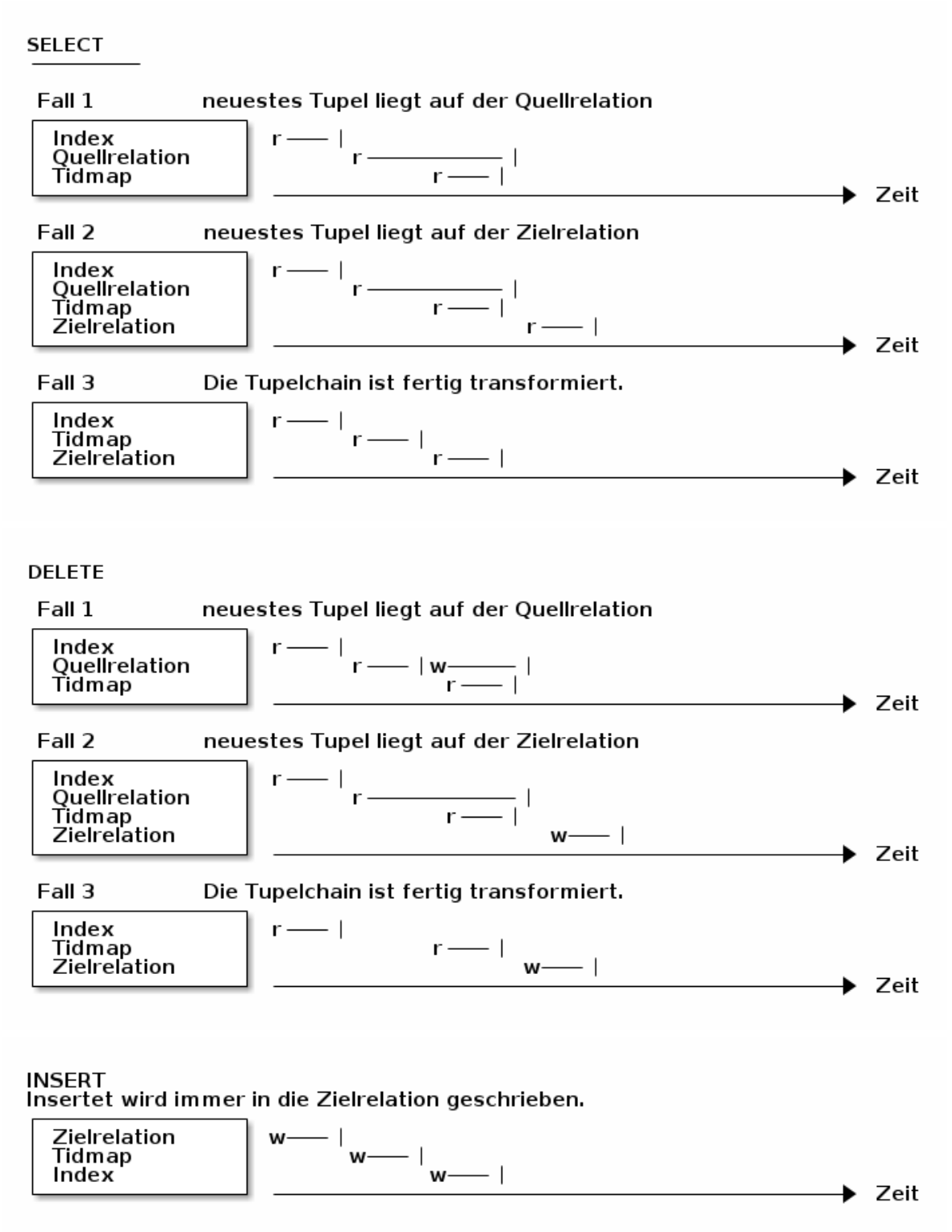


**Abbildung 3.7.:** Puffersperren des nebenläufigen SELECT, DELETE und INSERT

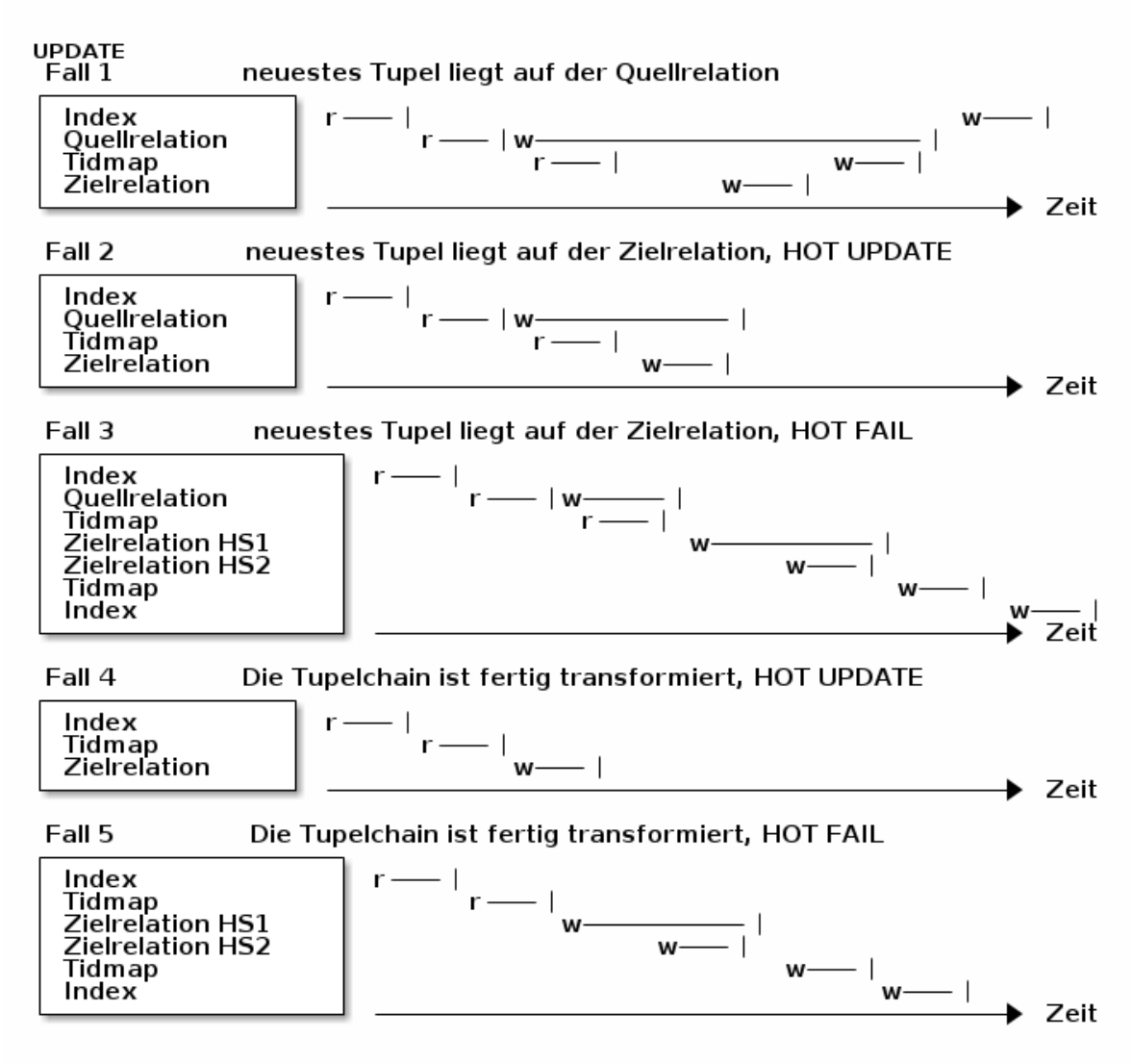


**Abbildung 3.8.:** Puffersperren des nebenläufigen UPDATE

## 3.8. Korrektheit des Verfahrens

### 3.8.1. Vorraussetzungen

#### Vorraussetzung 1

Für jede Tupelversion $A_n$ auf einer beliebigen Tabelle gilt

- Entweder $A_n$.CTID = $A_n$.TID , dann ist $A_n$ das **neueste Tupel** einer Tupelkette.
- Oder $A_n$.CTID $\neq$ $A_n$.TID und $A_n$.xmax hat abgebrochen, dann ist $A_n$ das **neueste Tupel** einer Tupelkette.
- Oder $A_n$.CTID $\neq$ $A_n$.TID und $A_n$.xmax hat committet, dann existiert ein $A_{n+1}$, das eine neuere Version des Tupels darstellt.

Zusammenfassend gilt für jedes Tupel, dass es entweder die **neueste Version** einer Tupelkette ist oder eine ältere Version, für die ein **Folgetupel existiert**.

#### Vorraussetzung 2

Für jede Tupelversion $A_n$ auf einer beliebigen Tabelle gilt

- Wenn $A_n$.CTID=$A_{n+1}$.TID, dann ist $A_n$.xmax=$A_{n+1}$.xmin

#### Vorraussetzung 3

Für alle Tupelversion auf einer beliebigen Tabelle gilt

- Für die erste Tupelversion jeder Tupelkette auf einer Heapseite existiert ein Indexeintrag.
- Für jede weitere Tupelversion einer Tupelkette, die nicht als HOT markiert ist, existiert ein Indexeintrag.

### 3.8.2. Verfahren

Die konkreten Tenantspace-Instanzen einer gegebenen Tabelle auf den Quelltenantspaces werden nach dem folgenden Verfahren zu Instanzen im Zieltenantspace transformiert.

#### Verfahrensregel 1

- Nacheinander wird auf die Tenantspace-Instanz der gegebenen Tabelle jedes zu transformierenden Tenantspaces Verfahrensregel 4 angewandt.
- Danach wird nacheinander jede Tenantspace-Instanz jedes Index der gegebenen Tabelle jedes zu transformierenden Tenantspaces transformiert.
- Die gegebene Tabelle gilt erst dann als transformiert, wenn alle ihre konkret zu transformierenden Tenantspace-Instanzen und deren Indizes transformiert sind.

#### Verfahrensregel 2

- Nach der Transformation aller Indizes, die auf einer gegebenen Tabelle in allen Tenantspace-Instanzen definiert sind und vor Abschluss der Transformation einer Tabelle werden die transformierten Tabellen-Instanzen und Index-Instanzen der Zieltenantspaces nebenläufigen Backends als die gültigen Instanzen verfügbar gemacht.
- Während diesem Vorgang darf keine nebenläufige Transaktion auf die Quell- oder Zielinstanzen der Tabellen oder Indizes Zugriff haben.

#### Verfahrensregel 3

Ein Transformationszyklus wird folgendermassen definiert:

- Eine gegebene Heapseite wird Tupel für Tupel in nach Offsetnummer aufsteigender Reihenfolge eingelesen.
- Nach dem Lesen des ersten Tupels wird eine Lesesperre auf der gegebenen Heapseite angefordert.
- Danach wird das erste Tupel neu von der Heapseite eingelesen.
- Sobald ein Tupel gelesen wird, das eine andere Heapseitennummer hat, gilt die gegebene Heapseite als fertig eingelesen.
- Alle Tupel aus ChainHaeds werden transformiert.
- Danach wird Lesesperre auf der gegebenen Heapseite freigegeben.

#### Verfahrensregel 4

- Auf alle Heapseiten einer gegebenen Relation wird in aufsteigender Reihenfolge der Transformationszyklus nach Verfahrensregel 3 angewendet.

#### Verfahrensregel 5

- In der Rangfolge der Information gilt für jedes Tupel :
  `Quellrelation < TidMap < Zielrelation`
- Bezüglich dieser Rangfolge werden die Informationen für jedes Tupel vom transformierenden Prozess **von hinten nach vorne** geschrieben.
- Nebenläufige Backends lesen die Informationen jedes Tupels bezüglich dieser Rangfolge **von vorne nach hinten**.

#### Verfahrensregel 6

- Eine Tupelversion wird nur dann in die Zielrelation geschrieben, wenn es entweder die nach Vorraussetzung 1 neueste Version einer Tupelkette ist oder aber das Folgetupel schon transformiert ist.

#### Verfahrensregel 7

- Jedes Tupel, das vom Transformationsprozess in die Zielrelation geschrieben wird behält seine vorher gültige erzeugende TransaktionsID xmin und löschende TransaktionsID xmax.

#### Verfahrensregel 8

- Jede neueste Version einer Tupelkette nach Vorraussetzung 1 wird während dem Einlesen der Heapseite von der Quellrelation zur Liste ChainHaeds hinzugefügt.

#### Verfahrensregel 9

- Jede Tupelversion, die nicht die neueste nach Vorraussetzung 1 ist wird, sofern das Folgetupel während dem Einlesen der Heapseite transformiert ist, ebenfalls während dem Einlesen der Heapseite von der Quellrelation zur Liste ChainHaeds hinzugefügt.

#### Verfahrensregel 10

- Jede Tupelversion, mit der nicht nach Verfahrensregel 8 oder Verfahrensregel 9 verfahren werden kann, wird während dem Einlesen der Heapseite in den Tupelcache eingefügt.

#### Verfahrensregel 11

- Jede Tupelversion, die während dem Einlesen der Heapseite zur Liste ChainHaeds hinzugefügt wurde, wird im gleichen Transformationszyklus, in dem sie von der Quellrelation gelesen wurde, in die Zielrelation geschrieben. Wenn die xmax der Tupelversion als committet bekannt ist, wird die CTID in der Tidmap aufgelöst und durch die transformierte CTID ersetzt. Ansonsten wird die CTID invalid gesetzt. Während dieses Schreibvorgangs wird eine Lesesperre auf der Heapseite in der Quellrelation gehalten. Direkt im Anschluss wird Verfahrensregel 12 angewendet.

#### Verfahrensregel 12

- Für jede Tupelversion, die in die Zielrelation geschrieben wird, wird im Tupelcache nach einem Vorgängertupel gesucht. Wenn es vorhanden ist, so wird es ebenfalls in die Zielrelation geschrieben und im Tupelcache gelöscht. Dabei wird die CTID durch die transformierte CTID ersetzt. Rekursiv wird Verfahrensregel 12 angewendet.

#### Verfahrensregel 13

- Der Transformationsprozess darf erst dann Tupel in die Zielrelation schreiben, wenn sichergestellt ist, dass **alle** nebenläufigen Backends, die zur gleichen Zeit Lese- oder Schreiboperationen auf der transformierten Tabelle ausführen, über die Reorganisation der Tabelle Bescheid wissen.
- Der Transformationsprozess darf nach Abschluss der Transformation erst dann die Quellrelationen und Tidmaps löschen, wenn sichergestellt ist, dass **alle** nebenläufigen Backends vom Abschluss der Transformation der Tabelle wissen und auf die konkreten Instanzen in ihrer transformierten Form zugreifen.

#### Verfahrensregel 14

- Nach dem Start der Transformation einer Tabelle werden auf der Quellrelation keine neuen Tupel mehr geschrieben.

- Hingegen werden bestehende Tupel auf der Quellrelation als gelöscht markiert, indem im Tupel xmax gesetzt wird, sofern sie die jeweils neueste Tupelversion nach Vorraussetzung 1 ihrer Kette sind.
- Alle neuen Tupelversionen, die nach dem Start der Transformation einer Tenantspace-Instanz einer Tabelle hinzugefügt werden sollen, werden in die Zielrelation geschrieben. Es wird eine auf der Quellrelation gültige Dummy-TID erzeugt und mit deren Hilfe ein Tidmapeintrag geschrieben. Bei einem Update eines Tupels von der Quellrelation wird die Dummy-TID in diesem Tupel auf der Quellrelation vermerkt. Ein nötiger Indexeintrag wird ebenso mithilfe der Dummy-TID erzeugt.

#### Verfahrensregel 15

- Eine neueste Tupelversion, die während der Reorganisation von nebenläufigen Backends auf der Quellrelation gefunden wird, wird nur dann als neueste Tupelversion akzeptiert, wenn sie nach Erhalt der letzten Lese- oder Schreibsperre und vor der Ausgabe nicht in der Tidmap aufgelöst werden kann.
- Andernfalls muss sie entsprechend dem gefundenen Tidmapeintrag in der Zielrelation gelesen und weiterverfolgt werden.

#### Verfahrensregel 16

- Alle auf der Quellrelation bestehenden Tupel bleiben für die Dauer der Transformation physisch erhalten.

#### Verfahrensregel 17

- Nebenläufige Transaktionen nutzen die Indizes der Quelltenantspaces und lösen die gefundenen Einträge in der Tidmap auf.
- Erst nach Abschluss der Transformation nutzen nebenläufige Transaktionen die Indizes der Zieltenantspaces.

#### Verfahrensregel 18

- Jede nebenläufige Änderung von Daten, die sich auf den Zielrelationen ergeben, erzeugen einen Indexeintrag in den Indizes der Zieltenantspaces, sofern die jeweilige Indexdefinition dies gebietet.

### 3.8.3. Zu zeigende Eigenschaften

#### Eigenschaft 1

- Für jede Tupelversion $A_n$ auf der Quellrelation gilt: Wenn ein Tidmap-Eintrag für $A_n$ existiert und im Tidmapeintrag($A_n$).isTransformed=true, so existiert auch ein Tupel $A'_n$ auf der Zielrelation.

#### Eigenschaft 2

- Wenn eine Tupelversion transformiert ist, so sind auch alle neueren Tupelversionen dieser Kette transformiert.

#### Eigenschaft 3

- Während der Transformation sind zu jeder Zeit alle neuesten Tupelversionen sichtbar.

#### Eigenschaft 4

- Letztendlich wird für jede gültige Tupelversion von der Quellrelation eine Kopie auf der Zielrelation erzeugt.

#### Eigenschaft 5

- Während der Transformation gehen keine Änderungen verloren.

#### Eigenschaft 6

- Die Transformation vermischt keine Tupelketten.

### 3.8.4. Nachweis

Beim Nachweis einer jeden Eigenschaft wurde darauf geachtet, jeweils keine Eigenschaft zu verwenden, die nicht vorher schon gezeigt wurde.

Es wird empfohlen, den folgenden Nachweis in der Pdf-Datei nachzuverfolgen. Alle Voraussetzungen, Verfahrensregeln und Eigenschaften sind verlinkt.

#### Eigenschaft 1

Eigenschaft 1 wird durch Verfahrensregel 5 sichergestellt. Jeder beteiligte Prozess, sowohl der transformierende als auch solche, die nebenläufige Transaktionen ausführen, schreiben die Informationen der gegebenen Rangfolge entsprechend von hinten nach vorne. Zuerst wird ein Tupel in der Zielrelation eingefügt. Erst dann wird ein Tidmapeintrag erstellt. Erst nach dem Tidmapeintrag wird im eventuell zu updatenden Tupel der CTID für das Folgetupel gesetzt.

Somit gilt Eigenschaft 1, dass **wenn ein Tidmapeintrag existiert**, der anzeigt, dass ein Tupel transformiert ist, auch **das Tupel auf der Zielrelation existiert**.

#### Eigenschaft 2

Eigenschaft 2 ist ebenso ersichtlich, denn für jedes Tupel gilt einer der drei Fälle aus Vorraussetzung 1. In den beiden Fällen, in denen ein Tupel eine neueste Version ist, gibt es kein Folgetupel, das transformiert sein müsste. Im weiteren Fall, dass ein Tupel eine Folgeversion hat, so gilt einer der folgenden Fälle:

- Es war entweder eines der Tupel, die nach Verfahrensregel 10 im Tupelcache zwischengespeichert wurden. In diesem Fall wurde es nach Verfahrensregel 12 transformiert, was bedeutet, es muss ein Folgetupel gegeben haben, für das die rekursive Rückwärtstraversierung und Transformation der Tupelkette gestartet wurde. Und dieses Folgetupel muss nach Verfahrensregel 6 schon vor dem Tupel selbst ausgeschrieben worden sein.
- Oder aber es war eines der Tupel, die nach Verfahrensregel 9 zur Liste ChainHaeds hinzugefügt und transformiert worden sind. Auch in diesem Fall muss es ein Folgetupel gegeben haben, denn Verfahrensregel 9 wird nur dann angewendet, wenn ein Tidmapeintrag für das Folgetupel gefunden wurde. Da der Tidmapeintrag gefunden wurde, existiert mit Eigenschaft 1 auch das transformierte Folgetupel auf der Zielrelation.
- Die dritte Möglichkeit liegt darin, dass das Tupel selbst nach Verfahrensregel 14 auf der Zielrelation erzeugt wurde. Dann liegen auch alle Folgeversionen wiederum wegen Verfahrensregel 14 auf der Zielrelation und sind transformiert.

In jedem Fall gilt also Eigenschaft 2, dass **für jedes transformierte Tupel alle Folgetupel ebenso transformiert sind**.

#### Eigenschaft 3

Während der Transformation wird bei der Leseoperation von nebenläufigen Backends jeder TID, die bei der Traversierung einer Tupelkette auftaucht, zuerst in der Tidmap nachgeschlagen.

- Wenn sie nicht aufgelöst werden kann, so ist das entsprechende Tupel auch nicht transformiert, denn bei jedem Schreibvorgang, ob nach Verfahrensregel 11 (Schreiben aus ChainHeads), Verfahrensregel 12 (rekursive Rückwärtstraversierung und Transformation) oder Verfahrensregel 14 (nebenläufiges Einfügen oder Ändern von Tupeln) ist ein Eintrag in die Tidmap enthalten. Somit ist die Tupelversion auf der Quellrelation die gültige und die Traversierung der Kette kann ebendort fortgeführt werden.
- Wenn ein TID in der Tidmap aufgelöst werden kann, so ist nach Eigenschaft 1 das transformierte Tupel auf der Zielrelation sichtbar. Mit Eigenschaft 2 liegen auch alle eventuell vorhandenen weiteren Tupel auf der Zielrelation. Die Tupelkette wird deshalb auf der Zielrelation weiterverfolgt und die dortige neueste Tupelversion ausgegeben.

Kann keiner der TID einer Tupelkette von der Quellrelation in der Tidmap aufgelöst werden und Verfahrensregel 15 wurde befolgt, so ist die auf der Quellrelation gefundene neueste Version die gültige, denn neuere Versionen hätten nach Verfahrensregel 14 auf der Zielrelation eingefügt werden müssen und das Tupel müsste als CTID einen TID enthalten, der in der Tidmap aufgelöst werden könnte. Das Tupel von der Quellrelation kann also ausgegeben werden.

Somit sind jeweils **immer alle neuesten Tupelversionen sichtbar** und Eigenschaft 3 gültig.

#### Eigenschaft 4

Der Transformationsprozess wendet Verfahrensregel 4 an. Für jedes Tupel $A_n$ gilt mit Voraussetzung 1, dass es entweder die neueste Version einer Tupelkette ist oder ein Folgetupel $A_{n+1}$ existiert.

- Im Fall, dass $A_n$ die neueste Version ist, wird es direkt nach dem Lesen der Heapseite mit Verfahrensregel 8 und Verfahrensregel 11 transformiert.
- Im Fall, dass $A_n$ nicht die neueste Version ist und $A_{n+1}$ transformiert ist, wird $A_n$ direkt nach dem Lesen der Heapseite mit Verfahrensregel 9 und Verfahrensregel 11 transformiert.
- Im Fall, dass $A_n$ nicht die neueste Version ist und $A_{n+1}$ nicht transformiert ist, wird $A_n$ mit 3.8.2 in den Tupelcache eingefügt.

Bei der Rückwärtstraversierung und Transformation eines Tupels $A_n$ nach Verfahrensregel 12 gilt das folgende:

- Solange ein Vorgängertupel $A_{n-1}$ zu $A_n$ gefunden wird, wird Verfahrensregel 12 angewendet.
- Sobald für ein Tupel $A_{n-x}$ kein weiteres Vorgängertupel $A_{n-x-1}$ mehr im Tupelcache gefunden wird, gilt das folgende:
  - Entweder ist $A_{n-x}$ das erste zu transformierende Tupel einer Kette. In diesem Fall ist die Tupelkette fertig transformiert.
  - Oder $A_{n-x-1}$ liegt auf einer folgenden Heapseite. In diesem Fall wird bei der Transformation ebendieser folgenden Heapseite Verfahrensregel 9 auf das Tupel $A_{n-x-1}$ angewendet werden, denn $A_{n-x}$ ist eben transformiert worden.
- Der Fall, dass $A_{n-x-1}$ auf einer vorhergehenden Heapseite liegt, kann nicht eintreten, denn dann
  - würde $A_{n-x-1}$ entweder im Tupelcache liegen und könnte mit Verfahrensregel 12 angewendet auf $A_{n-x}$ transformiert werden.
  - oder aber $A_{n-x-1}$ wäre schon transformiert und in diesem Fall hätten nach Eigenschaft 2 auch $A_{n-x}$ und $A_n$ schon transformiert sein müssen.

Somit gilt Eigenschaft 4, dass **letztendlich jede gültige Tupelversion von der Quellrelation transformiert wird**.

#### Eigenschaft 5

Aufgrund von Verfahrensregel 13 und Verfahrensregel 14 werden alle von nebenläufigen Änderungen erzeugten Tupelversionen in die Zielrelation geschrieben.

- Bei einer INSERT-Operation wird das erzeugte Tupel entsprechend Verfahrensregel 14 verfügbar gemacht.
- Eine DELETE-Operation findet nach Eigenschaft 3 jedes gültige neueste Tupel einer Versionskette. Dieses wird als gelöscht markiert, indem xmax auf die aktuelle TransaktionsID gesetzt wird.
  - Wenn die neueste Tupelversion auf der Zielrelation liegt, so gilt eindeutig Eigenschaft 5.
  - Wenn die neueste Tupelversion auf der Quellrelation liegt, so wird der Transformationsprozess die gesetzte xmax lesen, denn er hält nach Verfahrensregel 3 eine Lesesperre auf der Heapseite, während er das Tupel liest und transformiert es nach Verfahrensregel 8 und Verfahrensregel 11 im gleichen Transformationszyklus. Die Lesesperre blockiert die Schreibsperre, die zum setzen der xmax nötig ist. Wurde Verfahrensregel 15 eingehalten, so muss dies die neueste Version sein und der Transformationsprozess wird das Tupel erst später lesen und transformieren.

- Eine UPDATE-Operation findet nach Eigenschaft 3 jedes gültige neueste Tupel einer Versionskette.
  - Wenn die neueste Tupelversion auf der Zielrelation liegt, so wird sie dort geändert und es gilt Eigenschaft 5.
  - Wenn die neueste Tupelversion auf der Quellrelation liegt, so wird das erzeugte Tupel entsprechend Verfahrensregel 14 verfügbar gemacht. Wie beim DELETE wird dabei Verfahrensregel 15 eingehalten.

Insbesondere können auf der Quellrelation während der Transformation keine Teilketten länger als ein Tupel entstehen, deren xmax die TransaktionsID einer nicht abgeschlossenen Transaktion enthält. Somit gilt auf der Quellrelation während der Transformation, dass nur maximal eine Tupelversion einer Kette, und zwar die letzte (die neueste) einen nicht abgeschlossenen Transaktionsstatus haben kann. Dieses Tupel mit nicht abgeschlossenem Transaktionsstatus wird vom Transformationsprozess auf jeden Fall nach Verfahrensregel 8 und Verfahrensregel 11 mit einer Lesesperre auf der Quellheapseite transformiert. Somit wird beim **Transformieren jedes Tupels, das einen nicht abgeschlossenen Transaktionsstatus** hat, eine **Lesesperre auf der Quellheapseite** gehalten.

Somit gilt Eigenschaft 5, dass **während der Transformation keine nebenläufigen Änderungen verloren** gehen.

#### Eigenschaft 6

Die Rückwärtstraversierung der Tupelketten erfolgt an genau zwei Stellen, nämlich einmal nach Verfahrensregel 9 und einmal nach Verfahrensregel 12.

- Bei der Anwendung von Verfahrensregel 12 auf ein Tupel $A_n$ kann es nicht passieren, dass im Tupelcache ein Tupel $B_n$ einer anderen Tupelkette mit $B_n$.CTID=$A_n$ gefunden wird, denn dieses Tupel $B_n$ hätte die TransaktionsID einer abgebrochenen Transaktion als xmax gespeichert haben müssen und wäre infolgedessen nach Vorraussetzung 1 als neueste Tupelversion der Kette $B$ erkannt worden. Deshalb wäre es nach Verfahrensregel 8 nicht in den Tupelcache sondern in ChainHeads eingefügt worden.
- Bei der Anwendung von Verfahrensregel 9 auf ein Tupel $A_n$ - Hinzufügen eines Tupels zur Liste ChainHeads, weil das Folgetupel transformiert ist - kann es nicht passieren, dass eine TID aus einer fremden Tupelkette $B'_n$ im transformierten Tupel $A'_n$ gesetzt wird, weil nach Verfahrensregel 11 $A_n$.xmax als abgebrochen hätte identifiziert werden müssen. Somit hätte es kein Folgetupel geben können.

Somit gilt auch Eigenschaft 6, dass **bei der Transformation keine Tupelketten vermischt werden**.

### 3.8.5. Bemerkung zum Nachweis

Man führe sich vor Augen, dass man dem Algorithmus noch eine weitere Optimierung hinzufügen kann, nämlich die folgende: Tupel, deren erzeugende TransaktionsID (xmin) als abgebrochen bekannt ist, werden vom gegebenen Verfahren als neueste Tupelversion erkannt. Sie würden zur Liste ChainHeads hinzugefügt und transformiert werden. Solche Tupel kann man ignorieren, ohne dabei den gegebenen Nachweis zu brechen. Ein eventuelles Vorgängertupel würde nie in den Tupelcache aufgenommen werden, da die abgebrochene TransaktionsID ebenso im Vorgängertupel als xmax gespeichert wäre und somit das Vorgängertupel selbst als neueste Tupelversion erkannt würde. Ob man die genannte Optimierung einfügt oder nicht, ändert also nichts am Korrektheitsnachweis.

# 4. Implementierung

Die Implementierung erfolgte auf Basis des Prototyps der vorhergehenden Arbeit [Sch11] mit verschiedenen Korrekturen. Dieser basierte nach [Sch11, S. 63] auf PostgreSQL Version 8.4.

Das folgende Kapitel erklärt die konkrete Implementierung. Dabei wurden die Verfahrensregeln aus Kapitel 3.8.2 so gut wie möglich umgesetzt.

## 4.1. Datenmodell

### 4.1.1. Namensgebung

Alle bei in dieser Arbeit erstellten Tabellen, Funktionen und globalen Variablen erhalten das Prefix "mtor" (Multi Tenancy Online Reorganisation), um sie als zugehörig zum beschriebenen Algorithmus zu kennzeichnen.

### 4.1.2. Globale Variablen

**mtorFrozenMaxTid**
Die mtorFrozenMaxTid bezeichnet die TID, die zum Zeitpunkt des Starts der Transformation die größte vergebene TID auf der Quellrelation ist. Da die Quellrelation während des Splits nicht mehr wächst, sind alle größeren TIDs, die während der Transformation auftauchen, Dummy-TID, die von INSERT oder UPDATE Operationen erzeugt wurden. Die mtorFrozenMaxTid wird für jede Quellrelation gespeichert.

**mtorActualMaxTid**
Die mtorActualMaxTid bezeichnet die höchste vergebene TID auf der Quellrelation. Sie wird mit der mtorFrozenMaxTid initialisiert. Eine Dummy-TID wird erzeugt, indem mtorActualMaxTid um eins hochgezählt wird. Sie wird für jede Quellrelation gespeichert.

#### mtorMode

Der mtorMode ist eine Variable vom Typ Boolean und innerhalb eines Backends global verfügbar. Er kann die Werte MtorIdle oder MtorRunning annehmen. Er wird verwendet, um in jeglichen angepassten Funktionen die entsprechenden Programmteile auszuklammern, sofern gerade keine Reorganisation von Tenantspaces läuft. Es wird eine Get- und eine Set-Funktion zur Verfügung gestellt. Der mtorMode wird zu Beginn einer jeden nebenläufigen Transaktion auf MtorRunning gesetzt, wenn im Shared Memory der Umstand verzeichnet ist, dass gerade eine Reorganisation läuft und die zu transformierende RelationID nicht invalid ist.

Außerdem wird der mtorMode verwendet, um die Programmteile, die für nebenläufige Backends bestimmt sind von denen zu trennen, die für den Reorganisationsprozess selbst bestimmt sind. Das gilt speziell an den Stellen, wo eine Relation geöffnet wird. Das Backend, das die Transformation ausführt, öffnet Quell- und Zielrelation in Form eines Arrays von Relationen. Hingegen erhält bei nebenläufigen Transaktionen der Relationendeskriptor der Quellrelation einen Zeiger auf die Zielrelation. Deshalb bleibt der mtorMode im transformierenden Backend jederzeit auf MtorIdle.

### 4.1.3. Präprozessordirektiven

#### USE_MTOR

Durch das Setzen der Präprozessordirektive USE_MTOR wird der implementierte Algorithmus zur Übersetzungszeit aktiviert. Ist diese Direktive nicht gesetzt, so verwendet die Datenbank zur Umsetzung der Befehle SPLIT und MERGE den Algorithmus der vorhergehenden Arbeiten.

#### MTOR_WAITTIME_SPLITMERGE_INIT_USEC

Ist die Präprozessordirektive MTOR_WAITTIME_SPLITMERGE_INIT_USEC größer als 0, so wird im transformierenden Backend die entsprechende Zeit gewartet, nachdem die Informationen zur Reorganisation nebenläufigen Backends verfügbar gemacht wurden und sowohl Zielrelation als auch Tidmap erzeugt sind, aber bevor die eigentliche Reorganisation der konkreten Tupel anläuft. Im Logfile der Datenbank erscheinen Info-Statements, die die momentan zu reorganisierende Tabelle identifizieren. Dies wird hauptsächlich zur Fehlersuche verwendet und kann später auf 0 gesetzt werden.

#### MTOR_WAITTIME_SPLITMERGE_BETWEEN_PAGES_USEC

Nach jedem Aufruf von mtorFlushTupelCache wird
MTOR_WAITTIME_SPLITMERGE_INIT_USEC Mikrosekunden gewartet, um nebenläufige Backends überholen zu lassen. Zu diesem Zeitpunkt dürfen keine Bufferlocks, ob geteilt oder exklusiv, auf Quell- oder Zielrelation gehalten werden.

#### MTOR_WAITTIME_INDEXSCAN_USEC

Die Präprozessordirektive MTOR_WAITTIME_INDEXSCAN_USEC kann zur Fehlersuche größer als 0 gesetzt werden. In diesem Fall wird bei jedem Indexeintrag entsprechend gewartet, wodurch es einfacher wird, die Korrektheit der einzelnen Schritte im Logfile der Datenbank nachzuverfolgen.

#### MTOR_WAIT_BASE

Mithilfe der MTOR_WAIT_BASE können alle Wartezeitdirektiven auf 0 gesetzt werden.

### 4.1.4. Tabellen

Die folgenden Indexe und Variablen werden pro Quellrelation gespeichert:

#### mtorTidMap

In der mtorTidMap werden Mapping-Einträge gespeichert, die einzelne Tupelversionen der Quellrelation auf Tupelversionen in der Zielrelation abbilden. Sie wurde in Kapitel 3.2 besprochen. Prinzipiell könnte man sie als Hashmap implementieren, jedoch ist dann eine persistente Speicherung in der verwendeten Version von PostgreSQL nicht verfügbar. Deshalb wird die mtorTidMap als Heap mit einem Index in Form eines B-Baum realisiert. Sie kann somit persistent gespeichert werden. Das Schlüsselelement ist die TID des Tupels auf der Quellrelation. Es wird eine mtorTidMap pro Quellrelation gespeichert. Sie wird als Klasse definiert, erhält eine AddEntry-, eine GetEntry-, eine ReplaceEntry- aber keine DeleteEntry-Funktion. Der Anwendungsfall für DeleteEntry taucht nicht auf, da während der Transformation kein VACUUM auf der Tabelle laufen darf und nach der Transformation sowieso die ganze mtorTidMap gelöscht wird.

Die Datenstruktur der Einträge bleibt nach außen verborgen, um korrekte Zugriffe zu garantieren. Deshalb wird auch für jeden Teil eines Eintrags eine EntryGet-Funktion implementiert.

Ausserdem wird ein Wrapper mtorTidMapGetTransformedValue für nebenläufige Backends zur Verfügung gestellt, der nur die oldtid nimmt und die newtid zurückgibt.

Da die mtorTidMap als Heap mit Index realisiert ist, wird sie durch logische Locks geschützt. Bei Add und Replace wird ein RowExclusiveLock angefordert. Bei Get wird ein AccessShare-Lock geholt.

Der Heap und Index einer mtorTidMap wird im Base Tenantspace erzeugt, also mit invalider TenantID, um auszuschließen, dass die mtorTidMap wiederum transformiert werden könnte.

| oldtid | newtid | oldctid | newctid | isTransformed |
|---|---|---|---|---|

#### mtorTupelCache

Im mtorTupelCache werden während der Transformation die Tupel zwischengespeichert, die nicht die neuesten Versionen einer Tupelkette sind und deren CTID noch nicht in der mtorTidMap aufgelöst werden können. Sie werden bei der Rückwärtstraversierung der Tupelkette wieder aus dem mtorTupelCache entfernt. Es wird eine mtorTupelCache pro Zielrelation gespeichert.

Der mtorTupelCache ist per Definition ein Speicher, der oft schnell befüllt wird. Fast genauso schnell wird er wieder geleert. In der gleichen Häufigkeit, wie Einträge gelöscht werden, werden sie auch gesucht. Auf diesem Speicher werden also jeweils gleich viele INSERT- wie DELETE- und Suchoperationen ausgeführt. Die sinnvollste Art ist es daher, ihn als Hashmap anzulegen, denn dann beträgt die Komplexität jedes dieser Operatoren $O(1)$. [Momo1, vgl. S. 69] Da er zur Rückwärtsverfolgung von Tupelketten verwendet wird, ist das Schlüsselelement die CTID des einzutragenden Tupels. Eine persistente Hashmap ist in PostgeSQL nicht verfügbar, was jedoch unbedenklich ist, da die hier gespeicherten Tupel nicht als transformiert gelten und von nebenläufigen Backends nicht gesehen werden müssen. Nach einem Absturz der Datenbank können sie neu von der Quellrelation eingelesen und als noch nicht transformiert identifiziert werden, da die mtorTidMap persistent gespeichert ist.

Wie in Kapitel 3.3 diskutiert, ist darauf zu achten, dass keine Tupel in den mtorTupelCache eingefügt werden, deren xmax als abgebrochen bekannt ist.

Diese Tabelle wird nur lokal vom Transformationsprozess verwendet und muss daher nicht geschützt werden.

| oldctid | oldtid | tuple | roottid | isDead | isHot |
|---|---|---|---|---|---|

#### mtorChainHeads

Sinnvollerweise haben Einträge in mtorChainHeads den gleichen Datentyp wie Einträge im mtorTupelCache. Der Unterschied besteht lediglich darin, dass mtorChainHeads als Liste verwaltet wird. Diese Liste wird nur lokal vom Transformationsprozess verwendet und muss daher nicht geschützt werden.

#### tstotsmap

Die tstotsmap wurde aus der vorherigen Arbeit [Sch11, S. 89] übernommen. Sie wird verwendet, um innerhalb des transformierenden Backends die jeweiligen Relationendeskriptoren von Quelltenantspace und Zieltenantspace in Arrays zu verwalten und den ausführenden Funktionen zu überreichen. Dabei wird ein Array mit Einträgen vom Typ tstotsmap angelegt. Jeder Eintrag hat den Typ tstotsmap, welcher als C-Struct mit den gezeigten vier Variablen definiert ist. Er enthält die TenantspaceIDs von Quelltenantspace und Zieltenantspace, hier historisch gewachsen suffix genannt, da sie zugleich das Suffix der konkreten Datendatei darstellen.

Außerdem enthält die tstotsmap Arraypositionen in Form eines Integers für eine jede Kombination aus Quell- und Zielrelation, die in der konkreten Transformation nötig ist. Währen der

Transformation werden von den entsprechenden Funktionen zur Verwaltung der verschiedenen Tabellen Arrays von RelationenIDs, Relationendeskriptoren, Tidmaps und weiterer Strukturen angelegt, die jeweils pro Relation vorzuhalten sind. In den entsprechenden Arrays werden die hier definierten Arraypositionen benutzt.

Man führe sich dabei vor Augen, dass ein SPLIT hier mehrere Einträge mit gleichem src_suffix erzeugt, während ein MERGE mehrere Einträge mit gleichem dest_suffix hervorruft. Mithilfe dieser Struktur wird es möglich, zur Ausführung von SPLIT und MERGE die gleichen Funktionen zu benutzen. Die ausführenden Funktionen laufen an den entscheidenden Stellen über das Array von tstotsmap-Einträgen und erkennen einen SPLIT an den gleichen gleichen src_suffix und MERGE an den gleichen dest_suffix. Diese Tabelle wird nur lokal vom Transformationsprozess verwendet und muss daher nicht geschützt werden.

| src_suffix | dest_suffix | srcrel_array_pos | destrel_array_pos |
|---|---|---|---|

#### mtorTenantToTsMap

In der bisherigen Implementierung bemerken nebenläufige Backends die Reorganisation von Tenantspaces nicht, da sie alle Arbeiten auf den Quelltenantspaces verrichten. Erst nach Abschluss der Reorganisation greifen sie auf die Zieltenantspaces zu, wobei sie die Quelltenantspaces nicht mehr kennen. In der hier entwickelten Implementierung hingegen sollen nebenläufige Backends die Relationen von beiden Tenantspaces zugleich öffnen und dem Fortschritt der Reorganisation entsprechend in den verzahnten Daten die neuesten Tupelversionen suchen. Deshalb wird eine neue Struktur benötigt, die nebenläufigen Backends anzeigt, welche Tenantspaces gerade zu welchen transformiert werden. Zu diesem Zweck wurde die mtorTenantToTsMap entworfen.

Da nebenläufige Backends sich jeweils nur für die Daten eines konkreten Mandanten interessieren, wird pro Mandant eine Zeile in diese Tabelle eingefügt, die den Quelltenantspace und den Zieltenantspace angezeigt. Die Tabelle wurde als Hashmap fester Größe im Hauptspeicher implementiert. Ihre Größe ist durch die maximale Anzahl von Mandanten limitiert, die innerhalb von zwei Tenantspaces erlaubt sind. Da diese Anzahl in der gegebenen Implementierung nicht definiert ist, wird stattdessen als Obergrenze die Anzahl der Einträge im Systemkatalog pg_authid verwendet, wo die Mandanten gespeichert werden. Diese Tabelle wird beim Schreiben durch eine Schreibsperre geschützt und entsprechend beim Lesen durch eine Lesesperre. Diese Sperren sind als LWLock realisiert und gelten jeweils auf der ganzen Tabelle.

| TenantId | srcTenantspaceId | destTenantspaceId |
|---|---|---|

#### mtorRunningTransformation

Die Tabelle mtorRunningTransformation wird verwendet, um nebenläufigen Backends anzuzeigen, welche Tabelle, auch Relation genannt, gerade transformiert wird. Sie wurde als Hashmap fester Größe im Shared Memory implementiert. Sie hat maximal zwei Einträge, nämlich einen während eines SPLIT und zwei während einem MERGE. Sie speichert die RelationId des zur mtorTidMap gehörigen Heap und Index und außerdem die mtorFrozenMaxTid

und mtorActualMaxTid. Die mtorActualMaxTid wird verwendet, um Dummy-TID zu erzeugen. Diese Tabelle wird beim Schreiben durch eine Schreibsperre geschützt und entsprechend beim Lesen durch eine Lesesperre. Diese Sperren sind als LWLock realisiert und gelten jeweils auf der ganzen Tabelle.

| relid | srcTenantspaceId | tidmapHepOid | tidmapIdxOid | frzMaxTid | actMaxTid |
|---|---|---|---|---|---|

## 4.2. Transformation - SPLIT und MERGE

Zum Startzeitpunkt einer Transformation wird entsprechend der Eingaben die tstotsmap erzeugt. SPLIT und MERGE erhalten jeweils als Parameter die zu transformierenden und zu erzeugenden Tenantspaces. Siehe dazu Ausschnitt 2.1. Die Verteilung der Mandanten auf die Zieltenantspaces wird aus der vorherigen Arbeit [Sch11, S. 53] übernommen. In späteren Arbeiten kann sie ein Ansatzpunkt für automatische Optimierungen sein. Für diese Arbeit reicht es aus, eine simple Gleichverteilung zu erzeugen. Bei der Erzeugung der Mandantenverteilung wird zugleich für jeden Mandanten ein Eintrag in der mtorTenantToTsMap gemacht. Im Systemkatalog pg_relrel werden alle Relationen gesucht, die im zu transformierenden Tenantspace enthalten sind. Für jede Tabelle wird zuerst mtorBeginSplitMergeRel gestartet, welches die nötigen Strukturen anlegt. Danach wird committet, damit nebenläufige Backends Zielrelationen und Tidmaps lesen können. Ist dies erledigt, so kann mtorSplitMergeRel gestartet werden.

### 4.2.1. Funktionen

#### mtorBeginSplitMergeRel

In den jeweiligen Zieltenantspaces werden die Zielrelationen erzeugt. An diesem Punkt unterscheidet sich der Algorithmus vom vorherigen in [Sch11], denn dort wurden die Zielrelationen zuerst mit einer anderen RelationID erzeugt und nach der Transformation auf Dateiebene umbenannt. Das macht Sinn, wenn kein nebenläufiges Backend zugleich darauf zugreifen soll. Ebendiese Anforderung hat sich aber mit der Problemstellung 1.2 geändert. In der vorgelegten Implementierung wird es unnötig, die Relation bei Abschluss der Transformation auf Dateiebene zu verschieben. Dabei ergaben sich jedoch Probleme beim korrekten öffnen einer Relation, was im folgenden noch erörtert wird.

Im Base Tenantspace werden die mtorTidMap-Instanzen erzeugt. Mit den entsprechenden Informationen wird ein Eintrag in mtorRunningTransformation geschrieben, der nebenläufigen Backends anzeigt, welche Relation gerade transformiert wird. Danach wird gewartet, bis alle aktuell laufenden Transaktionen ausgelaufen sind. Damit wird sichergestellt, dass alle nebenläufigen Transaktionen auf dieser Tabelle die Reorganisation bemerkt haben und Quell-und Zielrelation öffnen.

#### mtorSplitMergeRel

mtorSplitMergeRel wurde in seiner Struktur und Funktionsweise aus der vorherigen Arbeit [Sch11, S. 89] übernommen und an die neuen Bedingungen angepasst. Teile wurden ausgelagert. mtorSplitMergeRel öffnet alle nötigen Quell- und Zielrelationen sowie Tidmaps und startet mtorSplitMergeRel, welches die konkreten Tupel in die Zielrelationen einsortiert. Nach Abschluss der Transformation der Tupel kann mtorSwitchIndex gestartet werden, welches die Indizes transformiert. Danach können Zielrelationen und Zielindizes als gültige Instanzen verfügbar gemacht und Tidmaps und Quellrelationen gelöscht werden.

#### mtorRewriteRels

Auch mtorRewriteRels wurde in seiner Struktur und Funktionsweise aus der vorherigen Arbeit [Sch11, S. 89] übernommen und an die neuen Bedingungen angepasst. mtorRewriteRels liest eine Quellrelation nach der anderen ein. Innerhalb einer jeden Quellrelation liest es die Heapseiten in aufsteigender Reihenfolge mithilfe eines Heapscans ein. Dabei wird die entsprechende Funktion explizit nicht vom modifizierten Indexscan gestört. Innerhalb jeder Heapseite wird für jedes Tupel die Fallunterscheidung aus Unterkapitel 3.4.2 angewendet. Die Eigenschaften HOT und tot werden hier extrahiert und als Boolean im Tupelcacheentry gespeichert. Sobald der Heapscan ein Tupel liefert, das eine andere Heapseitennummer wie das letzte enthält, ist eine Heapseite fertig eingelesen und mtorFlushTupelCache kann gestartet werden. Erst danach wird die Lesesperre auf der Heapseite der Quellrelation freigegeben und es wird gewartet, um nebenläufige Transaktionen durchzulassen. Danach wird aus dem eben gelesenen Tupel die Heapseitennummer gespeichert. Die neue Heapseite wird mit einem BUFFER_LOCK_SHARE gesperrt und das Tupel neu gelesen. Zu Beginn einer jeden neuen Heapseite wird mit der PostgreSQL-Funktion heap_get_root_tuples die Roottid jedes auf der Seite enthaltenen Tuples bestimmt. Diese wird ebenfalls in jedem Tupelcacheentry gespeichert.

#### mtorFlushTupelCache

Für jedes Tupel in ChainHeads wird der schon eingelesene Teil der Tupelkette im Tupelcache zurückverfolgt und auf einem Stack gespeichert. Entweder für jedes einzelne Tupel oder aber, falls Teile einer Tupelkette HOT sind für jeden HOT Tupelkettenteil, wird mit dem Stack die Funktion mtorWriteTupleStack gestartet. Vorher wird jeweils mtorDestHeapInsertNeededSpace mit der Summe der Tupelgrößen im aktuellen Stack aufgerufen, um zu garantieren, dass die aktuell verwendete Heapseite auf der Zielrelation noch genügend freien Speicherplatz zur Verfügung hat. Somit wird jeder Stack garantiert in eine einzige Heapseite auf der Zielrelation geschrieben. Insbesondere werden Teilketten, die vor der Transformation HOT waren, auch nach der Transformation wieder HOT sein.

Nach jedem Aufruf von mtorWriteTupleStack wird mtorWriteOptionalRoottid gestartet, falls auf dem letzten Stack Tupel waren und mtorWriteTupleStack anzeigt, dass die roottid nicht geschrieben wurde.

#### mtorWriteTupleStack

Der eingegebene Stack ist eine Liste von mtorTupelCache - Einträgen. mtorFlushTupelCache ist so geschrieben, dass dies alles Tupel der gleichen Tupelkette in ihrer gültigen Folge rückwärts sind. Ausserdem stammen die Tupel alle von der gleichen Heapseite. Deshalb wird die Roottid des ersten Tupels vom Stack - der neuesten Tupelversion - mit der TID jedes weiteren Tupels verglichen. Jedes Tupel wird durch Aufrufen von mtorDestHeapInsert in die Zielrelation geschrieben. Dabei erhält es eine auf der Zielrelation gültige TID, mit deren Hilfe dann der Tidmapeintrag angepasst werden kann.

#### mtorDestHeapInsertNeededSpace

In der Datenbank ist ein sogenannter FillFactor für Relationen einstellbar. Wenn der benötigte freie Speicherplatz in Summe mit dem vom FillFactor abhängigen geschützten Freibereich größer ist als der tatsächlich in der Heapseite noch vorhandene Speicherplatz, so wird eine neue Heapseite auf der Zielrelation angefordert, auf der genügend Platz vorhanden ist.

#### mtorDestHeapInsert

mtorDestHeapInsert schreibt ein Tupel in die Zielrelation. Konkret wurde eine modifizierte Version des von PostgreSQL gegebenen heap_insert verwendet.

heap_insert lädt selbstständig eine neue Heapseite in den Buffer, wenn in der aktuellen kein Platz mehr vorhanden ist. Dieser Teil wurde auskommentiert, da diese Aufgabe von mtorDestHeapInsertNeededSpace übernommen wird und außerdem garantiert werden soll, dass HOT Tupelketten nach der Transformation wieder HOT sind.

Außerdem setzt heap_insert im Tupel die schreibende TransaktionsID (xmin) auf die aktuell vorhandene und die löschende oder ändernde TransaktionsID (xmax) auf eine invalide TransaktionsID. Da die Reorganisation ein Tupel nicht logisch verändern darf, wurden auch diese Teile auskommentiert.

#### mtorWriteOptionalRoottid

mtorWriteOptionalRoottid schreibt nur dann den in Unterkapitel 3.4.5 beschriebenen zweiten Tidmapeintrag, wenn die erste transformierte TID des von mtorWriteTupleStack transformierten Stacks ungleich der Roottid des neuesten Tupels auf diesem Stack ist.

**Ausschnitt 4.1** mtorsetenv - Setzen von Umgebungsvariablen bei nebenläufigen Transaktionen

```
CREATE FUNCTION mtorsetenv() RETURNS VOID AS $$
DECLARE
    -- declarations
BEGIN
  SET enable_indexscan=on;
  SET enable_bitmapscan=off;
  SET enable_seqscan=off;
  SET enable_tidscan=off;
END;
$$ LANGUAGE 'plpgsql';
```

#### mtorSwitchIndex

Die Struktur und Funktionsweise von mtorSwitchIndex wurde aus der Implementierung zur vorherigen Arbeit [Sch11, S. 89] übernommen und noch nicht an die neuen Bedingungen angepasst. Die zugehörige Mapping-Tabelle für TID wird während der Transformation vom transformierenden Prozess gepflegt, jedoch nicht von den nebenläufigen Backends. Somit sind nebenläufige INSERTS und solche UPDATES, die nicht HOT ausgeführt werden, **nach Abschluss der Reorganisation** verloren. Während der Reorganisation sind sie verfügbar.

## 4.3. Nebenläufige Transaktionen

### 4.3.1. Umgebungsvariablen

#### mtorsetenv

Während eine Transformation auf einer Tabelle läuft, sind auf dieser Tabelle nur Index Scans, aber keine Sequential Scans, Tid Scans oder Bitmap Scans erlaubt. Denn in der gegenwärtigen Implementierung kann nur der Index Scan mit einer nebenläufigen Reorganisation umgehen. Andere Scans würden bei lesenden Operationen nicht abstürzen, aber eventuell veraltete Daten liefern, da die Zielrelation nicht miteinbezogen würde. Ändernde Operationen würden die Änderungen auf der Quellrelation ausführen, wobei im Reorganisationsprozess die Annahme gebrochen würde, dass eine gerade reorganisierte Tabelle nicht mehr wächst.

Zu diesem Zweck wird die Funktion mtorsetenv aus Ausschnitt 4.1 zur Verfügung gestellt. Sie muss als SQL-Funktion in der Datenbankumgebung innerhalb einer Session vor dem Aufruf eines SPLIT oder MERGE Operators erzeugt werden. Alle nebenläufigen Backends müssen sie aufrufen, bevor sie auf gerade reorganisierte Tabellen zugreifen. Idealerweise sollten die entsprechenden Schalter zukünftig automatisch gesetzt werden, sobald eine nebenläufige Reorganisation erkannt wird, also zusammen mit dem mtorMode.

**Ausschnitt 4.2** Anpassungen im Relationendeskriptor

```
TenantspaceId rd_mtor_dest_suffix            ;
Oid           rd_mtor_dest_tmprelid          ;
Relation      rd_mtorDestRelation            ; /* Destination Relation      */
mtorTidMap    rd_mtortidmap                  ; /* TID Mapping               */
bool          rd_mtor_tuple_from_destrel     ; /* used for DELETE, UPDATE   */
bool          rd_mtor_isdestrel              ;
bool          rd_mtorTransformationInProgress ; /* true, if this relation
                                                   is currently transformed. */
```

### 4.3.2. Relationendeskriptor

Ausschnitt 4.2 zeigt die Änderungen am Relationendeskriptor. Der Boolean-Wert rd_mtor_tuple_from_destrel wird beim nebenläufigen Indexscan verwendet, um den UPDATE oder DELETE Operatoren anzuzeigen, dass ein zurückgegebenes Tupel von der Zielrelation stammt. Sie können dann gegebenenfalls entsprechend reagieren.

### 4.3.3. mtorDestRelationOpen

Die Funktion relation_open wird in PostgreSQL verwendet, um jegliche Relationen zu öffnen, sowohl Heapdateien als auch Indizes. Sie ruft in der gegenwärtigen Implementierung relation_mtopen auf, welches für den aktuell gesetzten Mandanten den zugehörigen Tenantspace ermittelt und die entsprechende Tenantspace-Instanz einer Tabelle öffnet. Alternativ kann relation_mtopen mit einer erzwungenen TenantID oder TenantspaceID aufgerufen werden. relation_mtopen wurde nun so angepasst, dass es mit einer nebenläufigen Transformation einer Tabelle umgehen kann.

Zu diesem Zweck wird am Ende von relation_mtopen, wenn die Quellrelation geöffnet ist, mtorDestRelationOpen aufgerufen. mtorDestRelationOpen erhält als Eingabe den Zeiger auf den Relationendeskriptor der Quellrelation und wird genau diesen Zeiger auch in jedem Fall wieder zurückgeben, der dann von relation_mtopen zurückgegeben wird.

mtorDestRelationOpen gibt diesen Zeiger auf den Relationendeskriptor ohne Änderungen zurück, wenn die Relation, die mit relation_mtopen geöffnet wurde, keine Quellrelation ist, die gerade transformiert wird.

Ist sie hingegen eine gerade transformierte Quellrelation, so öffnet mtorDestRelationOpen die entsprechende mtorTidMap und die zugehörige Zielrelation durch Aufrufen von relation_mtopen und speichert die entsprechenden Zeiger im Relationendeskriptor der Quellrelation. Ebendort wird auch rd_mtorTransformationInProgress=true gesetzt.

Im Relationendeskriptor der Zielrelation wird rd_mtor_isdestrel=true gesetzt.

Eine große Schwierigkeit lag im korrekten öffnen der Zielrelation ohne Beeinträchtigung der normalen Funktionsweise von Tenantspaces. Normalerweise soll jeder Mandant genau eine Instanz einer Tabelle öffnen können. Beim ersten Öffnen einer konkreten Tabellen-Instanz

wird der Relationendeskriptor erzeugt und im RelationCache gespeichert. Alle folgenden Aufrufe desselben Backends von relation_mtopen suchen den Relationendeskriptor dort und geben einen Zeiger auf eben diese Instanz des Relationendeskriptors zurück.

Beim ersten Öffnen der Zielrelation wurde das Mapping der Tenantspace-Instanz auf ein konkretes Suffix der die Daten enthaltenden Datei entsprechend dem Quelltenantspace aus dem Systemkatalog geladen. Auch mit einer erzwungenen TenantspaceID wurde der Relationendeskriptor nicht korrekt im Zieltenantspace erzeugt. Die entsprechenden Stellen im Code konnten aber auch nicht direkt angepasst werden, den die Quellrelationen sollten ja weiterhin zu öffnen bleiben.

Das Problem ergibt sich mit dem Gedanken, dass es bisher nicht vorgesehen war, im selben Backend zur selben Zeit gleiche Tabellen-Instanzen aus unterschiedlichen Tenantspaces zu öffnen und geöffnet zu halten.

Es wurde dadurch umgangen, dass beim Öffnen der Zielrelation an allen Stellen in der vorgelegten Implementierung die TenantspaceID des Zieltenantspaces kurzfristig in einer im konkreten Backend globalen Variable gespeichert wird. Diese wird abgefragt, wenn ein Relationendeskriptor erzeugt wird.

### 4.3.4. mtorIncActualMaxTid

Das atomare Hochzählen der Dummy-TID wird durch lesen, inkrementieren und schreiben des entsprechenden Feldes mtorActualMaxTid in mtorRunningTransformation erreicht. Über die ganze Dauer von Lese- und Schreiboperation wird eine Schreibsperre auf dieser Tabelle angefordert. Sie schließt sich nicht mit der Schreibsprerre von

### 4.3.5. SELECT

In nebenläufigen Backends sind während der Transformation einer Tabelle nur Indexscans erlaubt. Deshalb wurde die Funktion index_getnext entsprechend angepasst. Sie findet nach den Verfahrensregeln aus Kapitel 3.8.2 und das jeweils neueste Tupel. Diese Funktionalität wird für den DELETE und UPDATE verwendet.

### 4.3.6. INSERT

Der INSERT wurde umgesetzt, indem während der Transformation einer Tabelle von nebenläufigen Backends heap_insert nicht auf der Quell-, sondern auf der Zielrelation ausgeführt wird. Mit mtorIncActualMaxTid wird eine Dummy-TID erzeugt, mit deren Hilfe ein Indexeintrag erstellt wird.

### 4.3.7. DELETE

Zur Umsetzung von DELETE wurde ExecDelete so angepasst, dass heap_delete abhängig davon, wo index_getnext das neueste Tupel gefunden hat - auf Quell- oder Zielrelation - es ebendort als gelöscht markiert wird.

### 4.3.8. UPDATE

Für die Funktionalität von UPDATE wurde ExecUpdate angepasst, so dass mtor_heap_update ausführt anstatt von heap_update, wenn eine Transformation erkannt wird. mtor_heap_update ist dabei eine angepasste Version von heap_update. Die Schwierigkeit besteht hier vor allem dann, wenn ein zu updatendes Tupel auf der Quellrelation liegt.

### 4.3.9. Schutz der Zugriffe nebenläufiger Backends

Verfahrensregel 13 und Verfahrensregel 2 wurde dadurch sichergestellt, dass während einer jeden gesamten nebenläufigen Transaktion eine Lesesperre auf Basis eines LWLocks gehalten werden muss. Dabei wird eine einzige fest codierte LockId verwendet, die MTORRunning-TransformationLock. Während die entsprechenden Schalter vom Transformationsprozess gesetzt oder annulliert werden, die nebenläufigen Backends anzeigen, welche Tabelle gerade transformiert wird, wird mit MTORRunningTransformationLock eine Schreibsperre gehalten. Die entsprechende Funktion zur Anforderung der Sperre wartet also solange, bis keine nebenläufigen Backends mehr Transaktionen ausführen. Dann setzt der transformierende Prozess die Daten zum Zugriff auf die Zielrelation und gibt die Schreibsperre wieder frei.

# 5. Evaluation

## 5.1. Stand der Implementierung

Zum Zeitpunkt des Drucks dieser Ausarbeitung gehen alle nebenläufigen INSERTS und solche UPDATES, die nicht HOT ausgeführt werden, **nach Abschluss der Reorganisation** verloren. Während der Reorganisation sind sie verfügbar.

# 6. Zusammenfassung und Ausblick

Eingangs wurde in Kapitel 1 das Thema Mandanten und Tenantspaces eingeführt und die Aufgabe der Reorganisation gestellt. Danach wurde in Kapitel 2 das Umfeld der Arbeit näher beleuchtet. In Kapitel 3 folgte die Entwicklung einer konkreten Verfahrensweise und mündete mit Unterkapitel 3.8 in der Erstellung von 17 konkreten Verfahrensregeln. Sechs nicht direkt ersichtliche Eigenschaften des Verfahrens wurden als gültig nachvollzogen. In Unterkapitel 4 folgte die Beschreibung des konkret implementierten Prototyps.

Die größten Probleme traten auf bei der korrekt synchronisierten Verfügbarmachung der Instanzen von Tabellen innerhalb der Quell- und Zieltenantspaces, insbesondere deshalb, weil Strukturen gebaut werden mussten, die es ermöglichen, dass alle Backends gleichzeitig Relationen in verschiedenen Tenantspaces öffnen können müssen. Dies steht entgegen dem Prinzip der Tenantspaces.

Es konnten keine Messungen der Performanz gemacht werden, da die Transformation der Indizes noch die Tidmap der alten Implementierung [Sch11] verwendet und diese bei nebenläufigen UPDATE und INSERT-Operationen nicht korrekt geführt wird. Somit ist die konkrete Implementierung unvollständig und Messungen sinnlos. Dennoch ist die Implementierung möglich, was in Kapitel 3.8 gezeigt wurde.

## 6.1. Ausblick

Zum Zeitpunkt des Drucks dieser Ausarbeitung steht die Aufgabe noch aus, bei der Transformation der Indizes die mtorTidMap zu verwenden. Erst danach kann die Performanz der konkreten Implementierung mit geeigneten Tests ermittelt werden.

Mit der Präprozessordirektive MTOR_WAITTIME_SPLITMERGE_INIT_USEC ist ein einstellbarer Parameter vorhanden, der den Reorganisationsprozess wie in der Problemstellung (1.2) genannt, ausbremst. Ein eventueller vorteilhafter Effekt auf nebenläufige Prozesse muss noch gezeigt werden. Wenn der vorteilhafte Effekt gegeben ist, so kann dieser Parameter als automatisch anpassbarer Schieberegler ausgebaut werden, der die Geschwindigkeit der Reorganisation nach noch zu definierenden Kriterien an das jeweilige momentane Lastaufkommen anpasst.

## 6.2. Fazit

Die Synchronisation von nebenläufigen Prozessen ist sehr aufwendig.

# A. Anhang

| Requested Lock Mode | Current Lock Mode | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
| ACCESS SHARE | | | | | | | | X |
| ROW SHARE | | | | | | | X | X |
| ROW EXCLUSIVE | | | | | X | X | X | X |
| SHARE UPDATE EXCLUSIVE | | | | X | X | X | X | X |
| SHARE | | | X | X | | X | X | X |
| SHARE ROW EXCLUSIVE | | | X | X | X | X | X | X |
| EXCLUSIVE | | X | X | X | X | X | X | X |
| ACCESS EXCLUSIVE | X | X | X | X | X | X | X | X |

**Abbildung A.1.:** Kompatibilitätsmatrix der Logischen Sperren von PostgreSQL [Gro12]

# Literaturverzeichnis


[Fic11] U. Ficht. *Effiziente, dynamische Pufferskalierung für Mandanten*. Diplomarbeit, Universität Stuttgart, 2011. (Zitiert auf Seite 10)

[Gro09] P. G. D. Group. *PostgreSQL 8.4.15 Documentation*. PostgreSQL Global Development Group, 2009. URL http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4-A4.pdf. (Zitiert auf Seite 16)

[Gro12] P. G. D. Group. PostgreSQL 8.4.15 Documentation, 2012. URL http://www.postgresql.org/docs/8.4/static/explicit-locking.html. (Zitiert auf den Seiten 20 und 70)

[HR01] T. Härder, E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer DE, 2 Auflage, 2001. (Zitiert auf Seite 11)

[Lan00] T. Lane. Transaction Processing in PostgreSQL, 2000. URL http://www.postgresql.org/files/developer/transactions.pdf. (Zitiert auf Seite 20)

[Mom01] B. Momjan. PostgreSQL Internals Through Pictures, 2001. URL http://www.postgresql.org/files/developer/internalpics.pdf. (Zitiert auf den Seiten 15 und 56)

[Mom12] B. Momjan. Mvcc Unmasked, 2012. URL http://momjian.us/main/writings/pgsql/mvcc.pdf. (Zitiert auf Seite 16)

[Sch10] B. Schiller. *Mandantenpartitionierung in einem RDBMS*. Diplomarbeit, Universität Stuttgart, 2010. (Zitiert auf Seite 10)

[Sch11] B. Schiller. *Tenantspace - Ein Raum für Mandanten*. Diplomarbeit, Universität Stuttgart, 2011. (Zitiert auf den Seiten 10, 13, 21, 53, 56, 58, 59, 61 und 67)


Alle URLs wurden zuletzt am 30. 01. 2013 geprüft.

**Danksagung**

Ich danke Herrn Oliver Schiller, meinem Betreuer, für die geduldigen langen Gespräche über das Thema.

Weiterhin danke ich meinem Vater Thomas Mälzer für seine Geduld und die Unterstützung im Studium.

**Erklärung**

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

---

Ort, Datum, Unterschift