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

Diplomarbeit Nr. 3217

Effiziente, dynamische Pufferskalierung für Mandanten

Urban Ficht

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Oliver Schiller
begonnen am:	24. Mai 2011
beendet am:	23. November 2011
CR-Klassifikation:	H.2.4, H.2.7, H.3.4

Kurzbeschreibung

Software as a Service dient als Vertriebsmodell zur Bereitstellung von Datenbankdiensten. Der Dienstanbieter nutzt seine Position zur Spezialisierung und Konsolidierung, wodurch die Kosten pro Kunde sinken. Um dies zu erreichen, setzt der Dienstanbieter mandantenfähige Datenbanksysteme in einem Rechencluster ein. Damit das Datenbanksystem die Leistungsanforderungen der Kunden erfüllt, muss es horizontal skalierbar sein. Die horizontale Skalierung kann durch einen Migrationsansatz umgesetzt werden. In dieser Arbeit wird dazu ein Entwurf für eine Migration, welche im laufenden Datenbankbetrieb durchgeführt wird, erarbeitet. Mit Hilfe dieses Migrationsansatzes kann das mandantenfähige Datenbanksystem dynamisch und effizient auf Änderungen in der Transaktionslast reagieren. Des Weiteren wird in dieser Arbeit ein auf diesem Ansatz basierender Prototyp entwickelt und vorgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Ausschnitte

1. Einleitung

Viele Firmen stellen Software als Dienstleistung für ihr Kunden bereit. Insbesondere durch den Ausbau und die Integration des Internets in die Geschäftsprozesse von Unternehmen haben sich Software-as-a-Service-Geschäftsmodelle (SaaS) etabliert. Dabei treten Firmen als Dienstleister auf, die Software im Rahmen eines Dienstes über das Internet zur Verfügung stellen. Die Kunden dieser Firmen nutzen diese Software entweder direkt über den Webbrowser oder über proprietäre Applikationen. Wichtig ist dabei die Fähigkeit zur horizontalen Skalierung der als Dienst angebotenen Software. Diese findet, wie im Cloud Computing üblich, für den Nutzer transparent statt. Als allgemeine Regel gilt, dass Software meistens nur dann als Dienst bereitgestellt wird, wenn es für den Anbieter viele potentielle Kunden mit ähnlichen Anforderungen gibt.
Onlineshops bilden ein Geschäftsmodell, welches auf SaaS-Anwendungen basieren kann. Im folgenden wird dieses Beispiel nun näher erläutert. Es sind dabei drei Gruppen beteiligt. Zum einen der SaaS-Anbieter, welcher die von dem Onlineshop genutzte Software, wie zum Beispiel eine Datenbank, zur Verfügung stellt. Als zweites der Onlineshopbetreiber, welcher Kunde des SaaS-Anbieters ist. Dieser wird aus Sicht des Dienstleisters als *Mandant* bezeichnet. Als drittes die Kunden des Onlineshops, welche Einkäufe tätigen wollen. Die Anfragen dieser Kunden werden über den Onlineshop des Mandanten an die als Dienst zur Verfügung gestellte Datenbank weitergeleitet und dort bearbeitet. Diese Anfragen umfassen beispielsweise die Auflistung des Sortiments oder die Abgabe einer Bestellung.
Die von dem Softwaredienstleister zur Verfügung gestellte Datenbank muss dafür einige Anforderungen erfüllen, welche als Mandantenfähigkeit bezeichnet werden. Die Erfüllung dieser Anforderungen bildet die Grundlage für das Software-as-a-Service-Geschäftsmodell des Dienstanbieters.
Die Anforderungen zeigen sich im Vergleich zu konventionell beim Kunden vor Ort betriebener Software. Der Dienstanbieter hat aufgrund seiner Spezialisierung in diesem Bereich das nötige Wissen und Kompetenz für den Betrieb und die Wartung der Software. Deswegen ist er in der Lage, die Erfüllung der Anforderungen zu garantieren. Außerdem kann der Dienstanbieter mehrere Mandanten auf seiner Hardware konsolidieren. Die dadurch effizientere Hardwarenutzung führt zu geringeren Kosten pro Mandant. Dazu kommt, dass der Mandant keine Personalkosten oder Investitionskosten für die Hardware in diesem Bereich tragen muss. Zusammenfassend ist diese Ausnutzung der *Economics of Scale* durch den Dienstanbieter der entscheidende Grund für die Nutzung von Software-as-a-Service. Darüber hinaus erfüllt der Dienstanbieter weitere für den Kunden wichtige Anforderungen wie Verfügbarkeit und Skalierbarkeit.
Wichtig ist neben der Unterstützung dieser Anforderung auch, dass der Dienstanbieter die Datensicherheit gewährleisten kann. Dazu gehören neben den allgemeinen Maßnahmen zur Zugriffskontrolle insbesondere die Isolation der Daten einzelner Mandanten. Die Isolation zu

gewährleisten ist deshalb notwendig, da im Rahmen der Konsolidierung die Daten mehrerer Mandanten auf dem selben System vorgehalten werden. In dem Onlineshopbeispiel bedeutet dies, dass verhindert werden muss, dass ein Onlineshop beispielsweise die Aufträge eines Konkurrenten auslesen kann. Die Gewährleistung dieser Isolation ist Aufgabe einer mandantenfähige Datenbank, da sie solche sensiblen Daten mehrerer Mandanten umfasst.
Diese Arbeit befasst sich damit, die horizontale Skalierung für eine mandantenfähige Software-as-a-Service-Anwendung in diesem Umfeld umzusetzen. Dem Dienstanbieter steht dabei die für die Skalierung nötige Infrastruktur zur Verfügung. Die horizontale Skalierung soll dabei als kurzfristige Maßnahme zur Abfederung von Lastspitzen und zur Lastbalancierung des Anwendungssystems dienen. In dem eben eingeführten Onlineshopbeispiel können solche Lastspitzen etwa durch Sonderangebote oder durch saisonale Bedingungen, wie die Vorweihnachtszeit, entstehen. Damit trotz dieser Lastzunahme die vereinbarten Servicelevels von dem SaaS-Dienstleister eingehalten werden können, muss er die Anwendung und deren Daten auf seiner Infrastruktur skalieren. Dies kann zum Beispiel durch die Nutzung weiterer Server geschehen.
Als Ansatz für die horizontale Skalierung wird in dieser Arbeit die *Live Migration* auf Datenbankebene vorgestellt. Der Schwerpunkt liegt hierbei darauf, dass die Migration im laufenden Betrieb, also ohne Unterbrechung der Transaktionsabwicklung, durchgeführt wird. Eine Teilmaßnahme der Migration ist dabei die Pufferskalierung, welche durch kurze Reaktionszeit zur Bekämpfung von Lastspitzen eingesetzt werden kann. Bei der Live Migration einer mandantenfähigen Datenbank kann ausgenutzt werden, dass eine Partitionierung der Datenbank durch sogenannte *Tenantspaces* bereits vorliegt. Ein Tenantspace bündelt die physische Repräsentation der Daten mehrerer Mandanten und dient somit als Einheit für die Migration.
Die Arbeit gliedert sich wie folgt: Der Rest dieses Kapitels konkretisiert die Problemstellung und skizziert den erarbeiteten Lösungsansatz. Im Anschluss werden in Kapitel *Stand der Technik* Hintergrundinformationen zu mandantenfähigen Datenbanken und verwandte Ansätze vorgestellt. Außerdem enthält dieses Kapitel eine kurze Vorstellung der PostgreSQL-Datenbank, welche als Grundlage für den Prototyp dient. Darauf folgen im Kapitel *Mandantenfähige Datenbanken im Rechencluster* Informationen über Systemarchitekturen für einen mandantenfähigen Datenbankcluster, sowie die Auswahl eines Ansatzes. Anschließend folgt in Kapitel *Entwurf* die detaillierte Vorstellung des Live Migration-Ansatzes. In Kapitel *Implementierung* folgen nun die Details der prototypischen Umsetzung des Entwurfs. Die Auswertung des implementierten Prototyps und die Vorstellung von Anknüpfungspunkten wird in Kapitel *Abschließende Betrachtung* durchgeführt. Zum Abschluss bietet das Kapitel *Zusammenfassung* eine Übersicht über die Ergebnisse der Arbeit.

1.1. Motivation

In diesem Abschnitt wird zunächst eine Übersicht der allgemeinen Anforderungen an mandantenfähige Datenbanksysteme gegeben. Anschließend wird daraus die Problemstellung abgeleitet. Darauf folgt eine kurze Vorstellung des Lösungsansatzes.
Neben Anforderungen, die auch an konventionelle Datenbanksysteme gestellt werden,

müssen mandantenfähige Datenbanksysteme noch weitere Anforderungen erfüllen. Diese resultieren zumeist aus der Tatsache, dass mehrere Mandanten in einer Datenbankinstanz zusammengefasst werden. Diese Konsolidierung hat gegenüber dem konventionellen Einsatz den Vorteil, dass die zur Verfügung stehenden Ressourcen besser ausgenutzt werden können.
Allgemeine Anforderungen an mandantenfähige Datenbanksysteme wurden bereits in [AJPK09] untersucht. Diese und weitere Anforderungen werden nun im folgenden kurz vorgestellt und am Beispiel eines Dienstes für Onlineshops konkretisiert.

Mandantenkonsolidierung wird genutzt, um mehrere Mandanten auf einer Datenbankinstanz zusammenzufassen. Dies ermöglicht eine optimierte Ressourcenauslastung und verringert so die Kosten des Dienstanbieters. Die Mandantenkonsolidierung stellt eine Grundlage für das wirtschaftliche Anbieten dieses Dienstes dar. Die Konsolidierung soll dabei uneingeschränkt von der Größe oder Anzahl an Mandanten durchgeführt werden können.

Isolation der Mandanten muss gewährleistet werden, damit die Sicherheit der Kundendaten vor unbefugten Zugriff gegeben ist. Isolationsmechanismen können der mandantenfähigen Datenbank vorgeschaltet oder in sie integriert sein. Die Integration bietet ein größeres Maß an Sicherheit und eine effizientere Nutzung der internen Mechanismen der Datenbank. Ein Onlineshop, der Kunde des Dienstleisters ist, muss sich darauf verlassen können, dass seine Daten von denen anderen Mandanten isoliert sind. Dies ist zum einen wichtig für den fehlerfreien Ablauf seines Geschäftsprozesses und zum anderen kann die Datenbank sensible Daten, wie beispielsweise Kreditkarteninformationen, beinhalten, die nicht nach außen gelangen dürfen.

Ressourcenverwaltung und -überwachung, sowie **Lastverteilung** werden für eine optimale Auslastung der Ressourcen benötigt. Bei einem mandantenfähigen System müssen diese Mechanismen auch auf Ebene der Mandanten durchgeführt werden können. Die Ressourcenzuteilung muss zum Beispiel darauf eingehen, wenn einzelne Mandanten spezielle Serviceleistungen, wie besonders schnelle Verarbeitung der Anfragen, buchen. Da das mandantenfähige Datenbanksystem beim Dienstleister in einem Clusterumfeld betrieben wird, spielt hierfür auch die Lastverteilung eine wichtige Rolle. Um die optimale Leistung des Clusters ausnutzen zu können, ist es notwendig im laufenden Betrieb kurzfristig auf Laständerungen reagieren zu können. Dies ist wichtig, damit einzelne Server des Clusters vor Überlastung und der damit einhergehenden Verringerung der Leistung geschützt werden. Damit die Lastverteilung effektiv umgesetzt werden kann, müssen Migrationsmechanismen vorhanden sein. Diese sind unter anderem dafür zuständig, die Daten und deren Bearbeitung von einem Server auf den anderen zu übertragen.
Die Ressourcenverwaltung und -überwachung als auch die Lastverteilung können manuell oder automatisiert durchgeführt werden. Für eine automatisierte Lösung sprechen der geringere Aufwand und die Verfügbarkeit. Dazu nutzt der Cluster ein Koordinatorsystem, welches diese Aufgaben übernimmt. Es erfasst die Daten aus der Ressourcenüberwachung und initiiert auf diesen basierend die nötigen Schritte zur

Lastverteilung im Cluster.
Diese Mechanismen dienen dazu, dass der Kunde seine Leistung dynamisch beim Dienstleister buchen kann. Ein Kunde, wie ein Onlineshop, könnte zum Beispiel für Buchungsabschlüsse zusätzliche Leistung benötigen. Aber auch für unvorhergesehene Ereignisse, wie ein überdurchschnittlich großes Kundenaufkommen, wird die nötige Leistung automatisiert und kurzfristig durch das mandantenfähige System des Dienstleisters zur Verfügung gestellt.

Skalierbarkeit folgt aus diesen eben genannten Punkten der Lastverteilung und dynamischer Leistungsbuchung. Das mandantenfähige Datenbanksystem muss die Möglichkeit besitzen im laufenden Betrieb neue Ressourcen einzubinden und zu nutzen. Für mandantenfähige Datenbanken spielt in dieser Hinsicht vor allem die horizontale Skalierbarkeit eine wichtige Rolle. Die horizontale Skalierbarkeit des Systems stellt sicher, dass automatisch auf Änderungen an der Anzahl der Mandanten, dem Volumen der Daten oder der Anzahl an Zugriffen reagiert werden kann. Im Beispiel des Dienstleisters für Onlineshops wird die Fähigkeit zur horizontalen Skalierbarkeit genutzt, wenn sich neue Onlineshops dazu entschließen, die Datenbank des Dienstleisters zu nutzen. In der mandantenfähige Datenbank des Dienstanbieters werden nun automatisch die neuen Mandanten und ihre Daten eingepflegt, sodass die Anfragen der Endkunden bearbeitet werden können. Um diese horizontale Skalierbarkeit zu gewährleisten, muss die Möglichkeit bestehen, dass das mandantenfähige Datenbanksystem neue Instanzen auf neuen Servern hinzunimmt oder sich bei Lastabbau auf wenige Server konsolidiert.

Verfügbarkeit ist eine grundlegende Anforderung an Software-as-a-Service-Dienstleister. Die Kunden des Dienstleisters brauchen für ihre Geschäftsprozesse uneingeschränkten Zugriff auf ihre Daten. Im Beispiel des Onlineshops können Endkunden nur Einkäufe tätigen, wenn die Datenbank beim Dienstleister auch verfügbar ist. Kunden möchten über das Internet jederzeit Zugriff auf den Onlineshop haben. Wenn dieser nicht verfügbar wäre, ginge ihm Umsatz verloren.

Wartbarkeit ist ein weiterer wichtiger Punkt. Zum einen muss die Software regelmäßig aktualisiert und an neue Anforderungen angepasst werden und zum anderen darf dies nicht im Widerspruch zur Verfügbarkeit des Systems stehen. Das mandantenfähige System muss also Mechanismen unterstützen, die dies ermöglichen. Zu diesen Mechanismen kann eine Update-Methode, die im laufenden Betrieb angewendet werden kann, zählen. Eine Alternative für die Wartung in einem Cluster ist, dass einzelne Server nacheinander aus dem Cluster genommen und gewartet werden.

Datensicherheit spielt, wie für konventionelle Datenbanksysteme, eine zentrale Rolle. Die mandantenfähige Datenbank muss mittels Backups und Recovery sicherstellen, dass die Daten und Transaktionen dauerhaft bestand haben. Mandantenfähige Datenbanksysteme sollte darüber hinaus die Möglichkeit haben, die Daten eines einzelnen Mandanten zu recovern oder zu extrahieren. Dies könnte zum Beispiel notwendig sein, wenn die Applikationssoftware des Onlineshops, welche auf die Datenbank zugreift, einen logischen Fehler verursacht hat. In diesem Fall kann es nun wünschenswert sein, dass mit Hilfe der Recovery ein bestimmter Zustand der Datenbank aus Sicht des Mandaten wieder hergestellt wird.

1.1.1. Problemstellung

Aus den in 1.1 vorgestellten Anforderungen an ein mandantenfähiges Datenbanksystem lässt sich die nun im folgenden beschriebene Problemstellung ableiten. Der Lösungsansatz dieser Arbeit wird dann im darauf folgenden Abschnitt vorgestellt.
Konzepte für die Konsolidierung und Isolation von Mandanten wurden in den vorherigen Arbeiten [SSBM11], [Sch10] sowie [Sch11] vorgestellt. Dabei wurde auch das Konzept der *Tenantspaces* eingeführt, welches eine physische Partitionierung mehrerer Mandanten ermöglicht.
Für die horizontale Skalierbarkeit mandantenfähiger Datenbanksysteme gibt es noch keinen Ansatz. Diese spielt aber für den Einsatz eines mandantenfähigen Datenbanksystems eine wichtige Rolle, um die Anforderungen an dynamische Lastbalancierung zu erfüllen. Die horizontale Skalierung kann dadurch umgesetzt werden, dass neue Server mit Datenbankinstanzen in den mandantenfähige Datenbankcluster integriert und auch wieder entfernt werden können. Damit auch auf Lastspitzen kurzfristig reagiert werden kann, ist eine kurze Reaktionszeit dieses Mechanismus notwendig. Außerdem soll die Skalierung im laufenden Betrieb vollzogen werden, damit die Verfügbarkeit der Datenbank nicht eingeschränkt wird.

1.1.2. Lösungsansatz

Ein mandantenfähiges Datenbanksystem muss diese in 1.1 vorgestellten Anforderungen erfüllen. Als Grundlage für deren Umsetzung ist zunächst die Systemarchitektur festzulegen. Die Systemarchitektur bestimmt die beim Dienstbetreiber notwendige Infrastruktur und Ressourcen. Da die Anforderungen nicht von einem einzelnen Server erfüllt werden können, muss der Anbieter die Server in einem Rechnerverbund vorhalten. Die einzelnen Rechner dieses sogenannten Clusters sind untereinander mit Netzwerkinfrastruktur verbunden und verfügen über eine Anbindung an das Internet für die Kommunikation nach außen.
Der Lösungsansatz sieht eine *shared-nothing* Architektur für den Betrieb des mandantenfähiges Datenbanksystems vor. Durch die Wahl dieser Architektur wird keine spezielle Hardware benötigt. Außer einem dem Datenbankcluster vorstehenden Koordinator wird keine zusätzliche Schicht an Software für den Betrieb benötigt. Dieser Koordinator übernimmt die Überwachung des mandantenfähigen Datenbanksystems und leitet entsprechende Maßnahmen wie eine Migration ein. Außerdem routet er die Zugriffe durch die Mandanten auf die entsprechenden Server des Clusters.
Der Schwerpunkt dieser Arbeit liegt auf der Umsetzung der horizontalen Skalierung. Dazu wird ein Live Migration-Mechanismus entwickelt, welcher es ermöglicht die Daten von einem Server auf einen anderen zu übertragen. Dieser Migrations-Mechanismus ermöglicht die Anpassung der Ressourcenauslastung im Cluster an den aktuellen Bedarf. Eine Voraussetzung für die Migration bilden die in [Sch11] eingeführten Tenantspaces. Diese erzeugen eine Partition der Datenbank und dienen als Einheit für die Migration . Für die Datenbankleistung unter Last spielt die Puffergröße und die Zugriffsgeschwindigkeit auf die Festplatten eine entscheidende Rolle. Da diese Ressourcen unter Lastspitzen an ihre Grenzen kommen, nutzt der Live Migration-Ansatz als kurzfristige Maßnahme mit schneller

Reaktionszeit die Pufferskalierung. Die Pufferskalierung ermöglicht die Vergrößerung des Datenbankpuffers indem die Ressourcen eines weiteren Server hinzugenommen werden. Dieser Server, welcher im ersten Schritt zur Pufferskalierung genutzt wird, bildet im zweiten Schritt das Ziel der Migration. Außerdem verringert die Pufferskalierung direkt die Last auf dem Ursprungsserver, da mit dem Puffer auch die Bearbeitung von Anfragen an das Zielsystem ausgelagert werden. Die Migration der Mandanten entbindet mittelfristig den Ursprungsserver vollständig von der durch diese Mandanten verursachten Last.
Ein wichtiger Punkt bei der Migration ist, dass diese im laufenden Betrieb durchgeführt werden kann. Dies ist zum einen aufgrund der Verfügbarkeit notwendig und zum anderen wird die Migration im Rahmen der horizontalen Skalierung unter momentan hoher Last ausgeführt. Eine längere Unterbrechung zur Durchführung der Migration würde dieses Problem noch weiter verschärfen. Dies bedingt allerdings auch, dass die beteiligten Server parallel in Betrieb sind. Deswegen umfasst der Lösungsansatz hier Methoden zur Synchronisation der beteiligten Systeme.
Des Weiteren kann durch die Migration die Wartbarkeit des Systems verbessert werden, weil Durch die Migration sämtlicher Daten eines Servers dieser aus dem Cluster entfernt werden kann. Anschließend kann dieser software- und hardwaretechnisch gewartet und danach wieder in den Cluster integriert werden.

2. Stand der Technik

Dieses Kapitel befasst sich mit dem für diese Arbeit notwendigen Hintergrundwissen. Es werden zunächst verschiedene Konzepte der Mandantenfähigkeit erläutert. Dabei bildet das darin erläuterte *Shared Table*-Modell die Grundlage für diese Arbeit.
Anschließend folgt die Erläuterung bisheriger Ansätze, welche zu dieser Arbeit verwandte Gebiete behandeln. Sie befassen sich mit Themen im Umfeld der Migration, welche eine zentrale Rolle in dieser Arbeit spielt.
Zum Abschluss dieses Kapitels wird eine Einführung in die PostgreSQL-Datenbank gegeben. Dabei werden vor allem die Module behandelt, welche für die Realisierung des Prototypen von Bedeutung sind.

2.1. Mandantenfähigkeit

Die Eigenschaft der Mandantenfähigkeit bezeichnet die Fähigkeit von Systemen Mandanten als logische Objekt zu verwalten. Mandantenfähigkeit zeichnet sich dabei dadurch aus, dass mehrere dieser Mandanten auf einer Instanz des Systems zusammengefasst werden. Damit wird eine Konsolidierung der Ressourcennutzung angestrebt.
Den Unterschied zwischen Mandantenfähigkeit und Mehrnutzerbetrieb werden in [BZ10] aufgezeigt. Bei beiden Ansätzen teilen sich die Mandanten beziehungsweise die Nutzer eine Applikation. Man spricht allerdings dann von Mandantenfähigkeit, wenn erweiterte Konfigurationsmöglichkeiten bestehen. Die Applikation kann also im Gegensatz zum Mehrnutzerbetrieb für den einzelnen Mandanten individualisiert werden. Diese Anpassungen müssen dabei auch auf der Datenebene unterstützt werden. Dafür gibt es drei Methoden, die nun im folgenden vorgestellt werden.

2.1.1. Umsetzung auf Datenebene

Für die Realisation der Mandantenfähigkeit ist die Umsetzung auf der Datenebene, welche das Fundament der gemeinsamen Applikationsschicht bildet, entscheidend. Die einzelnen Ansätze wurden in [JA07] vorgestellt und unterscheiden sich in ihrer Konsolidierungseffizienz und der Isolation zwischen den Mandanten.

Shared Machine: Bei diesem Ansatz teilen sich die Mandanten eine physische Maschine, besitzen aber jeweils eine eigene Datenbankinstanz.

Shared Process: Die Mandanten teilen sich hier eine Datenbankinstanz, operieren aber jeweils auf einem eigenen Schema.

Shared Table: Hier werden die Daten mehrerer Mandanten in der selben Relation gespeichert.

Der Shared Machine-Ansatz benötigt keine Änderung der Datenbankimplementierung, da die Mandanten durch übergeordnete Systeme verwaltet werden müssen. Des Weiteren ermöglicht dieser Ansatz die stärkste Isolation der Mandanten auf der Datenebene. Wenn jede Datenbankinstanz in einer eigenen virtuellen Machine läuft, dann sind die Daten der einzelnen Mandanten automatisch voneinander isoliert.
Dem gegenüber stehen Nachteile in der Ressourcennutzung. Die einzelnen Datenbankinstanzen teilen sich den zur Verfügung stehenden Arbeitsspeicher auf. Zum einen verursacht dabei jede Instanz Overhead für die interne Verwaltung der Datenbank. Zum anderen wird auch der Datenbankpuffer nicht geteilt, sodass auch allgemeine Tabellen, die gemeinsam genutzt werden könnten, von jeder Instanz selbst im Puffer verwaltet werden müssen. Durch die damit einhergehende geringere Konsolidierungseffizienz wird auch die Skalierbarkeit sehr eingeschränkt.
Bei dem Shared Process-Ansatz, der in Abbildung 2.1 skizziert ist, besitzt jeder Mandant private Relationen. Durch das Anlegen privater Tablespaces werden die Daten der Mandanten auch physisch getrennt. Durch die gemeinsame Datenbankinstanz ist der Overhead geringer als beim Shared Machine-Ansatz. Allerdings muss die Isolation und Datensicherheit entweder auf Datenbank- oder Applikationsebene umgesetzt werden. Dies steigert die Komplexität dieses Ansatzes.
Der Shared Table-Ansatz besitzt die höchste Konsolidierungseffizienz. Da die Daten mehrerer Mandanten in einer Tabelle zusammengefasst sind, kann auch der zur Verfügung stehende Datenbankpuffer besser genutzt werden. Dies liegt daran, dass sich Anfragen verschiedener Mandanten auf die gleichen Datenbereiche beziehen können. Diese befinden sich beispielsweise im Rahmen einer Datenbankseite teilweise bereits im Puffer. Darüber

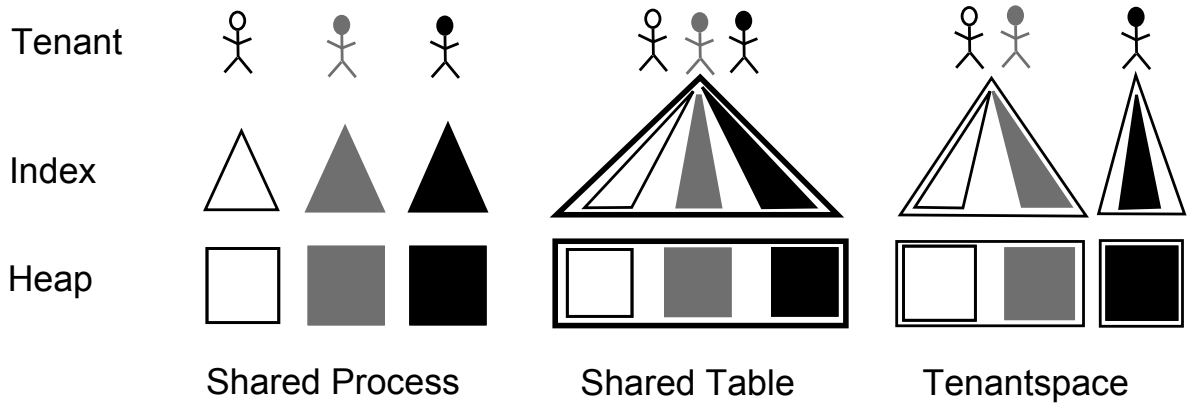

Abbildung 2.1.: Datenorganisation für Mandantenfähigkeit (in Anlehnung an [SB11])

hinaus ist die Skalierbarkeit zunächst nur direkt über die auf dem System maximal mögliche Tabellengröße begrenzt.
Dem gegenüber stehen aber einige Nachteile, welche über die Datenbanklogik ausgeglichen werden müssen. Zum einen müssen für die Isolation aufwändigere Maßnahmen als in den anderen Ansätzen getroffen werden, da sich die Daten einzelner Mandanten nicht direkt trennen lassen. Darüber hinaus muss die Erweiterbarkeit des Schemas für einzelne Mandanten und damit das Ermöglichen der Individualisierung auf die Datenbank abgebildet werden. Als letzter Punkt ist die Migration einzelner Mandanten zu lösen. Diese Migration ist beispielsweise aus Gesichtspunkten der Skalierbarkeit notwendig. Da die Daten mehrerer Mandanten in den Relationen zusammen gespeichert sind, ist für die Migration eine Partitionierung dieser Daten notwendig.
Da der Shared Table-Ansatz hinsichtlich der Konsolidierungseffizienz am vielversprechendsten ist, bildet er die Grundlage dieser Arbeit. Die Datenorganisation der Relationen im Rahmen dieses Ansatzes wird im folgenden Abschnitt erläutert.

2.1.2. Datenorganisation bei Shared Table

Für die physische Organisation der Daten im Shared Table-Ansatz werden in [SB11] und in [Sch11] Methoden vorgestellt. Diese Ansätze sind in Abbildung 2.1 auf Seite 14 schematisch dargestellt. Auf zwei für diese Arbeit wichtige Shared Table-Ansätze wird nun im folgenden eingegangen.

Shared Index: Die Mandanten teilen sich nicht nur die Tabellen für die *Heap*-Daten sondern auch die Zugriffsstrukturen, welche diese Daten indizieren.

Tenantspace: Dieser Ansatz baut auf den Shared Index-Ansatz auf. Er erweitert diesen durch eine Trennung der Heap- und Indexdaten nach Mandantengruppen und nicht nur nach einzelnen Mandanten.

Der Shared Index-Ansatz zeichnet sich durch eine hohe Trefferrate im Puffer aus. Dies zeigt sich beispielsweise im Vergleich zu einem Ansatz, der für jeden Mandanten einen getrennten Index vorhalten würde. Für einen effizienten Zugriff solten die Indexseiten für die Bearbeitung der Anfrage bereits im Puffer sein. Bei sehr vielen Mandanten sinkt allerdings die Wahrscheinlichkeit, dass die entsprechende Indexseite bereits im Puffer ist. Dies liegt daran, dass Seiten, die relativ lange nicht mehr genutzt wurden, den Puffer wieder verlassen müssen. Bei einem Shared Index hingegen nutzen viele Mandanten die gleichen Indexseiten, sodass diese seltener aus dem Puffer geweorfen werden.
Tenantspaces bieten nun den Vorteil, dass sie eine Möglichkeit zur Partitionierung der Mandanten in Gruppen darstellen. Dadurch wird der Nachteil des Shared Table-Ansatzes, dass die Migration einzelner Mandanten und derer physischer Daten sehr aufwändig ist, ausgeglichen.

2.2. Bisherige Ansätze

Für die Durchführung einer Migration gibt es verschiedene Ansätze aus unterschiedlichen Gebieten. Der Zweck der Migration ist hierbei nicht auf die Skalierung eines Systems eingeschränkt, sondern kann auch aus Verwaltungs- oder Leistungsgründen durchgeführt werden. Im folgenden werden nun Arbeiten vorgestellt, die für unterschiedliche Anforderungen ähnliche Ansätze beinhalten. Zunächst werden allgemeine Mechanismen zur Online Reorganisation von Datenbanken vorgestellt. Anschließend folgt ein Ansatz zur Live Migration von virtuellen Maschinen und als letztes werden zwei Ansätze auf mandantenfähigen Datenbanken erläutert.

Online Reorganisation von Datenbanken In [SI09] wird eine Übersicht über Ansätze zur Online Reorganisation rationaler Datenbanken gegeben. Diese Maßnahmen werden bei Datenbanken zu Wartungszwecken oder für Änderungen am physischen oder logischen Datenbankschema eingesetzt. Dabei soll der Einfluss auf den laufenden Betrieb möglichst gering ausfallen. Reorganisationsansätze, welche eine Kopie der Daten einer Partition an einen neuen Speicherbereich beinhalten, ähneln der Migration.
Eine Strategie, die auch auf die Live Migration von mandantenfähigen Datenbanken übertragen werden könnte, ist die *Fuzzy Reorganisation*. Dabei wird eine Kopie der Daten mit dem Ziel der Reorganisation angelegt. Während dessen ist das System unter Last und es werden somit Lese- und Schreiboperationen ausgeführt.
In dieser Strategie wird zunächst ein Kopie der Partition erstellt, welche reorganisiert werden soll. Dies kann mit Hilfe von Backupmechanismen derart geschehen, dass der Betrieb nicht unterbrochen wird. Die Kopie wird nun reorganisiert, was relativ einfach möglich ist, da sie selbst nicht unter Last ist. Nachdem die reorganisierte Kopie zur Verfügung steht, werden auf ihr die in der Zwischenzeit durch Schreiboperationen angefallenen Logeinträge aufgespielt. Dadurch wird erreicht, dass die Kopie sich dem aktuellen Zustand des Originals annähert. Anschließend werden die Schreiboperationen auf dem Original angehalten und bis zum Ende der Reorganisation aufgeschoben. Es entstehen jetzt keine neuen Logeinträge mehr. Die restlichen Logeinträge werden jetzt in die Kopie eingebracht, sodass Kopie und Original in den gleichen Zustand kommen. Anschließend werden auch die Leseoperationen auf dem Original unterbrochen, damit intern das Mapping der Zugriffe auf die Kopie umgestellt werden kann. Zum Abschluss können die angehaltenen Operationen sowie sämtliche neue Operationen auf der Kopie durchgeführt werden. Damit ist die Reorganisation abgeschlossen.
Wenn man für die Reorganisation nicht nur eine Kopie auf einen neuen Speicherbereich vornimmt sondern eine Kopie auf ein anderes System, dann ähnelt der Mechanismus einer Live Migration. Ein Nachteil dieses Ansatzes ist, dass zur Erstellung der Kopie das Quellsystem belastet wird.

Live Migration von virtuellen Maschinen Der in [KCF+05] vorgestellte Ansatz zur Live Migration von virtuellen Maschinen bietet einige Methoden, die auch auf die Migration mandantenfähiger Datenbanksysteme übertragen werden können.

Die Live Migration bedingt, dass neben den persistenten Daten auch der aktuelle Hauptspeicherinhalt transferiert wird. Dieser Ansatz zur Live Migration von virtuellen Maschinen gliedert sich in mehrere Stufen, von denen die *Iterative Pre-Copy*- und die *Stop-and-Copy*-Phasen die für die Migration entscheidenden sind.
In der Iterative Pre-Copy-Phase wird ein Hauptspeicherabbild der Quellmaschine erstellt und auf den Zielserver übertragen. Die Hauptspeicherseiten, welche sich während einer Iteration ändern, werden anschließend in der nächsten Iteration übertragen. Um die geänderten Seiten mitzuloggen, wird eine Bitmap-Datenstruktur vorgehalten, die die geänderten Seiten markiert.
Darauf folgt die Stop-and-Copy-Phase. In dieser Phase wird die laufende Betriebssysteminstanz auf der Quelle unterbrochen, damit ein konsistenter Zustand erreicht wird. In dieser Phase werden die in der letzten Phase übrig gebliebenen Seiten und der aktuelle Stand der CPU übertragen. Damit ist nun das Ziel in der Lage, den Betrieb genau an der Stelle, an der das Quellsystem unterbrochen wurde, wieder aufzunehmen.
Im nun laufenden Betrieb auf dem Ziel kann dieses bei Bedarf noch einzelne Seiten anfordern, wenn diese bisher noch nicht von der Quelle übertragen wurden. Dies kann für persistente Daten, die sich auf dem Externspeicher befinden, der Fall sein.
Wie sich Teile dieses Migrationsansatzes auf Datenbanken übertragen lassen, soll nun kurz skizziert werden:

- Das Konzept zur Übertragung der Hauptspeicherseiten, welche im Rahmen der Pre-Copy-Phase kopiert werden, lässt sich für Datenbanken auf Pufferseiten abbilden.
- Die Unterbrechung zur Übertragung des CPU-Zustandes während der Stop-and-Copy-Phase würde bei Datenbanken dazu genutzt werden, die Informationen der aktuell aktiven Transaktionen zu übertragen.
- Die Seitenanforderung zur Übertragung persistenter Daten durch das Zielsystem kann auch auf Datenbanken angewandt werden.

Iterative Migration mandantenfähiger Shared Process-Datenbanken In [DNAA10] wird eine Technik namens *Iterative Copy* zur Online Database Migration vorgestellt. Als Grundlage wird hier von einem *Shared Process*-Modell zur Unterstützung der Mandantenfähigkeit ausgegangen. Dies bedeutet, dass die Datenbank bereits nach Mandanten partitioniert ist.
Die Migration findet in einem Rechencluster statt, in dem die einzelnen Server mittels eines Shared-Disk-Konzepts verbunden sind. Deshalb stehen die persistenten Daten grundsätzlich auf allen Knoten des Rechnerverbundes zur Verfügung. Bei der Migration müssen also lediglich die aktuellen Daten, wie sie im Datenbankpuffer vorliegen, und die Besitzrechte übertragen werden.
Der Iterative Copy-Mechanismus gliedert sich dabei in mehrere Phasen. Da die persistenten Daten durch die Shared-Disk-Umgebung nicht übertragen werden müssen, begrenzt sich die Migration auf die Daten im Hauptspeicher.
Zunächst wird ein Snapshot von der Quelle angefertigt und auf das Ziel übertragen. Die Quelle bearbeitet währenddessen weiterhin die Anfragen. Da die Quelle dadurch nun einen anderen Zustand als das Ziel hat, müssen die Änderungen seit dem letzten Snapshot an das

Ziel übertragen werden. Dies geschieht iterativ solange, bis nur noch wenige Änderungen seit dem letzten Snapshot vorgefallen sind.
Die nächste Phase wird als *Atomic Handover* bezeichnet. Dabei wird die Bearbeitung auf der Quelle kurzzeitig gestoppt und die fehlenden Änderungen werden an das Ziel übertragen. Da zu diesem Zeitpunkt noch Transaktionen aktiv sein können, müssen diese abgebrochen werden. Mit dieser Phase ist die Migration abgeschlossen und der Zielserver übernimmt nun die Bearbeitung von Anfragen.
Der Nachteil dieses Verfahren ist es, dass der Datenbankservice trotzdem für kurze Zeit unterbrochen werden muss. Zusätzlich kommt es zum Abbruch einiger Transaktionen. Ansonsten ähnelt der Ansatz dem der Live Migration von virtuellen Maschinen.

Live Migration mandantenfähiger Shared Process-Datenbanken In [EDAA11] wird mit **Zephyr** ein Verfahren zur Online Migration vorgestellt, dass die Nachteile der eben vorgestellten Iterativen Migration verringert. Es gibt bei diesem Ansatz für die Migration keine Unterbrechung, während der neue Transaktionen zurückgewiesen oder aufgeschoben werden. Allerdings werden trotzdem in Spezialfällen Transaktionen abgebrochen. Zephyr setzt ebenfalls auf eine Mandantenfähigkeit, die auf Shared Process beruht. Zu Grunde liegt Zephyr allerdings eine Shared-Nothing-Architektur, sodass die Daten komplett von einem Server auf den anderen übertragen werden müssen.
Auch Zephyr gliedert sich in mehrere Phasen. Sobald die Migration gestartet wird, beginnt die Phase des dualen Arbeitens von Quell- und Zielserver. Neue Anfragen werden automatisch an den Zielserver weitergeleitet und von diesem bearbeitet, während der Quellserver noch die Transaktionen abarbeitet, die bereits vor dieser Phase gestartet wurden. Da der Zielserver zu Beginn auf einer leeren Datenbank arbeitet, die nur die Metadaten enthält, holt er sich die von den aktuellen Transaktionen benötigten Seiten von der Quelle. In dieser Phase werden also gleichzeitig Transaktionen auf dem Quell- und Zielserver ausgeführt. Sind alle Transaktionen auf der Quelle beendet, ist diese Phase abgeschlossen. Jetzt werden noch die restlichen Seiten, welche vom Ziel bisher noch nicht angefragt wurden, übertragen. Wenn auch diese Phase abgeschlossen ist, wurde die gesamte Datenbankpartition übertragen und die Migration kann beendet werden.
Das System hat somit während der gesamten Migration keinen Ausfall und bleibt verfügbar. Lediglich in dem Fall, dass Transaktionen auf dem Ziel auf Seiten zugreifen, die von der Quelle noch in Bearbeitung sind, tritt ein Konflikt auf. Dieser wird durch das Abbrechen der entsprechenden Transaktion auf der Quelle gelöst, sodass anschließend die Seite auf das Ziel übertragen werden kann.
Die Online Migration mittels Zephyr führt somit nur zu wenigen abgebrochen Transaktionen. Allerdings verschlechtert sich die Antwortzeit von Transaktionen, die auf dem Ziel ausgeführt werden, da am Anfang der Puffer des Ziels leer ist. Es werden also zunächst alle Seiten bei Bedarf extra von der Quelle angefordert.

2.3. PostgreSQL

Das PostgreSQL Datenbanksystem bildet die Grundlage für den Prototypen. Es wurde bereits für die vorhergehenden Arbeiten verwendet, da es gut dokumentiert ist und als Open Source-Datenbanksystem zur Verfügung steht. In [Sch10] und [Sch11] wurde PostgreSQL in der Version 8.4 um Mandantenfähigkeit erweitert. Die Datenbank und ihre Module sind in der Programmiersprache C geschrieben.
Allgemein zeichnet sich PostgreSQL als eine relationale Datenbank aus, welche sich durch die in der Übersicht [Pos11] aufgelisteten Punkte hervorhebt. Die wichtigsten dieser Eigenschaften sind die Multi-Version Concurrency Control, Point in Time Recovery, Online Backups und Write-Ahead-Logging. Die für diese Arbeit wichtigen Komponenten werden in den folgenden Abschnitten näher erläutert.

2.3.1. Prozessarchitektur

Das PostgreSQL-Datenbanksystem gliedert sich in mehrere Prozesse, die unterschiedliche Funktionen warnehmen. Diese teilen sich in zwei Gruppen: Zum einen gibt es Prozesse, die interne Funktionalität der Datenbank übernehmen, und zum anderen gibt es Backends, die für die Kommunikation mit den Clients und für die Ausführung von SQL-Anfragen zuständig sind.
Der Postmasterprozess, der die generelle Verwaltung und das Starten der anderen Prozesse übernimmt, bildet das Zentrum der internen Funktionalität. Der Background-Writer-Prozess übernimmt des Weiteren die Pufferverwaltung und sorgt so dafür, dass Pufferseiten ausgeschrieben und so Platz für neue Seiten geschaffen wird. Der WAL-Writer-Prozess schreibt die bei Schreiboperationen entstehende Write-Ahead-Log-Einträge regelmäßig in die entsprechende Logdatei auf die Festplatte. Der Autovacuum-Launcher-Prozess überprüft regelmäßig, ob ein *Vacuum* von Relationen ausgeführt werden soll. Vacuum dient dazu, ungenutzten Platz innerhalb der Relationen freizugeben
Dem gegenüber stehen die Backendprozesse. Jedes Backend wird dabei von einem Client angesprochen und hält zu diesem Client eine Verbindung. Ein Client kann eine Verbindung beispielsweise mittels des *psql*-Frontends aufbauen. Wenn das Backend eine SQL-Anfrage bekommt, führt es die folgenden Schritte aus, um die Anfrage zu beantworten.
Die SQL-Anfrage wird innerhalb des Backends durch den Parser in einen Syntaxbaum umgewandelt. Dieser Syntaxbaum wird über Zwischenschritte durch den Optimizer und Planer in einen konkreten Ausführungsplan umgewandelt. Dieser enthält zum Beispiel auch die Indizes, welche für diese Anfrage verwendet werden sollen. Abschließend führt der Executor diesen Plan aus. Er führt Änderungen auf Tupelebene aus und sammelt die Ergebnistupel. Das Ergebnis, welches dem Clienten angezeigt werden soll, wird zum Abschluss an das Frontend zurückgeschickt.

2.3.2. Multi-Version Concurrency Control

PostgreSQL nutzt für die Serialisierbarkeit einen Multi-Version Concurrency Control-Ansatz, welcher hohe Parallelität bei den Zugriffen ermöglicht. Dies liegt daran, dass Schreib- und Leseanfragen nicht im Konflikt stehen und sich daher nicht blockieren. Konkret bedeutet dies, dass es von jedem Tupel mehrere Versionen geben kann. Für jede Transaktion ist dabei genau eine Version gültig.
Damit die für die aktuelle Transaktion gültige Tupelversion ermittelt werden kann, werden im Header des Tupels die ID der Transaktion, welche es erzeugt hat, und gegebenenfalls die ID der Transaktion, welche es gelöscht hat, gespeichert. Ein Spezialfall bildet das Update eines Tupels. Dabei wird das bisherige Tupel, anstatt es zu überschreiben, als gelöscht markiert und ein neues Tupel mit den aktualisierten Werten wird in die Relation eingefügt.
Damit nun Transaktionen ermitteln können, welches Tupel im Rahmen der Serialisierbarkeit für sie sichtbar ist, erzeugen sie zu Transaktionsbeginn einen *Snapshot*. Dieser Snapshot enthält unter anderem die ID der als letztes erfolgreich mit Commit beendeten Transaktion. Die neue Transaktion kann also alle Tupel lesen, die von dieser Transaktion oder davor erzeugt wurden. Es sei denn sie wurden inzwischen wieder gelöscht.
Außerdem enthält der Snapshot eine Liste aller zur Zeit aktiven Transaktionen. Änderungen dieser Transaktionen darf die neue Transaktion nicht sehen, da noch nicht sicher ist, ob nicht die Änderungen dieser Transaktionen durch ein Abort ungültig werden. Mit Hilfe dieses Snapshots überprüft die aktuelle Transaktion für jedes Tupel, ob es momentan sichtbar ist. Durch den Snapshot ist sichergestellt, dass die Serialisierbarkeit nicht verletzt wird.
Zusätzlich muss mit Hilfe des Commit-Logs überprüft werden, ob die Transaktion, die das Tupel erstellt oder als gelöscht markiert hat, auch tatsächlich durch ein Commit beendet wurde. Wenn dies nicht der Fall ist, dann wurden die Änderungen dieser Transaktion aufgrund eines Aborts ungültig. Dies hat natürlich entsprechende Folgen für die Tupelsichtbarkeit. Wurde das Tupel nun als sichtbar erkannt, dann kann die Transaktion dieses Tupel verarbeiten, andernfalls wird es übersprungen.

2.3.3. Write-Ahead-Logging

Ein Datenbanksystem muss sicherstellen, dass nach Abschluss einer Transaktion die Änderungen dauerhaft Bestand haben. Da aber bei einem Ausfall der Hauptspeicherinhalt verloren geht, muss für diesen Fall vorgesorgt werden. Eine einfache aber die Leistung stark verringernde Maßnahme ist das Ausschreiben aller Änderungen auf den Externspeicher, bevor die Transaktion mit Commit beenden darf. Da dieses permanente Ausschreiben oft unnötig das System belastet, wird stattdessen die Write-Ahead-Log-Technik genutzt.
In diesem Write-Ahead-Log werden die Informationen gespeichert, die nötig sind, um die Änderungen nach einem Aufall im Rahmen der Redo-Recovery wieder einzufügen. Die Write-Ahead-Log-Einträge müssen deswegen vor dem eigentlichen Commit der Transaktion auf den Externspeicher ausgeschrieben werden, damit sie nach einem Systemausfall noch zur Verfügung stehen. Damit nicht für jede Transaktion die Write-Ahead-Log-Einträge einzeln ausgeschrieben werden müssen, werden die Einträge mehrerer Transaktionen gebündelt und beim Group-Commit dieser Transaktionen zusammen ausgeschrieben.

Die Write-Ahead-Log-Einträge enthalten im Allgemeinen den physischen Inhalt des geänderten Tupels, sodass in der Recovery nur diese Änderungen wieder in die Seite kopiert werden müssen. Nur beim ersten Write-Ahead-Log-Eintrag jeder Seite wird diese komplett in das Log geschrieben. Dies ist notwendig für den Fall, dass das System beim Ausschreiben dieser Seite abstürzt und diese deshalb nur unvollständig und möglicherweise unkorrekt auf den Externspeicher geschrieben wurde. Während der Recovery werden dann alle Write-Ahead-Log-Einträge nacheinander gelesen und wieder in die Seiten eingebracht. Die Seiten, auf welche die Einträge angewandt werden, werden entweder vom Externspeicher gelesen oder sie sind als kompletter Seiteneintrag im Write-Ahead-Log vorhanden.
Damit bei der Recovery nicht grundsätzlich vom ersten Erzeugen einer Seite an alle Änderungen nacheinander wieder eingebracht werden müssen, werden *Checkpoints* eingeführt. Bei einem Checkpoint werden alle Seiten im Hauptspeicher auf den Externspeicher ausgeschrieben. Dadurch sind alle Änderungen bis zu diesem Zeitpunkt nun dauerhaft zugänglich. Es werden jetzt also die alten Write-Ahead-Log-Einträge nicht mehr gebraucht und können gelöscht werden. Die seit Beginn des Checkpoints durch Änderungen anfallenden Write-Ahead-Log-Einträge werden in einem neuen Log-Abschnitt weitergeführt. Dadurch reicht es aus, bei der Recovery nur die Write-Ahead-Log-Einträge seit diesem Checkpoint wieder einzuspielen.

2.3.4. Verwaltung von Datenbankobjekten

PostgreSQL ist eine objektorientierte Datenbank. Die Objekte besitzen eine interne Struktur. Für dauerhafte Objekte wird eine physische Repräsentation dieser vorgehalten. Im Allgemeinen sind die Objekte physisch über Seiten strukturiert. Diese Seiten gliedern sich jeweils in einen Bereich für den *Page Header* und in einen Bereich für den Inhalt.
Der Page Header enthält neben Informationen über die interne Organisation der Seite auch die Log-Sequence-Number (LSN). Diese LSN gibt an, welcher Eintrag des Write-Ahead-Logs die letzte Änderung auf dieser Seite beinhaltet. Dadurch kann während der Recovery erkannt werden, ob die Seite bereits die Änderungen eines WAL-Eintrags beinhaltet oder ob dieser nun eingespielt werden muss.
In PostgreSQL gibt es zwei Typen an Relationen: *Heap* und *Index*. Die Heap-Relationen enthalten die Daten der Datenbank, während in den Index-Relationen die zugehörigen Zugriffsstrukturen gespeichert sind. Der Inhaltsbereich dieser Seiten wird durch die Tupel, welche einen einzelnen Eintrag darstellen, gefüllt. Ein Tupel des Heaps bildet also eine Reihe der Relation, während ein Tupel eines Indexes beispielsweise einen Verweis auf das entsprechende Heap-Tupel enthält.
Auch Tupel besitzen einen Header. Dieser *Tuple Header* dient zur Verwaltung und Organisation des Tupels und wird darüber hinaus dazu genutzt, die Multi-Version Concurrency Control-Sichtbarkeit zu regeln. Dazu werden im Tupel Header wie in 2.3.2 beschrieben die TransaktionsID der Transaktionen, welche dieses Tupel erzeugt oder gelöschten haben, gespeichert.
Intern werden diese physischen Datenbankobjekte mit Metadaten angereichert. Zu deren Aufbau werden die *Systemkataloge* genutzt. Die Systemkataloge selbst werden als Heap-Relationen gespeichert. Mit Hilfe der Systemkataloge kann zum Beispiel das logische Schema

einer Relation oder die zugehörigen Indizes ermittelt werden. Für eine Relation beinhalten die Metadaten unter anderem den *RelFileNode*, das zugehörige Schema und die passenden Indizes. Der RelFileNode dient dabei dazu, die physischen Dateien der Relation zu identifizieren.
Für einzelne Tupel und Pufferseiten werden Deskriptoren erzeugt. Diese werden dazu genutzt, den Inhalt des zugehörigen physischen Objekts zu interpretieren. Für Tupel enthält der Deskriptor beispielsweise die Anzahl und den Datentyp der einzelnen Attribute, sodass auf die entsprechenden Werte zugegriffen werden kann. Der Deskriptor einer Pufferseite enthält neben Daten zur Identifikation der Seite auch Mechanismen, um Sperren auf der Pufferseite zu erwerben. Deswegen befindet sich der Pufferdeskriptor auch in dem im nächsten Abschnitt vorgestellten *Shared Memory*, sodass auf ihn von verschiedenen Prozessen zugegriffen werden kann. Auch Metadaten, die sich auf Relationen beziehen, werden im Shared Memory vorgehalten. Dadurch kann zum einen Platz im Hauptspeicher gespart werden und zum andern muss nicht jeder Prozess selbst diese Datenstrukturen aufbauen und aktualisieren.

2.3.5. Interne Strukturen

In diesem Abschnitt wird der Shared Memory von PostgreSQL und das Tenantspace-Konzept vorgestellt. Diese internen Strukturen übernehmen Aufgaben, welche auch später im Rahmen der Implementierung dieser Arbeit genutzt werden.

2.3.5.1. Shared Memory

Der *Shared Memory* ist ein Bereich im Hauptspeicher, auf welchen von allen Prozessen zugegriffen werden kann. Der Platz für Strukturen im Shared Memory wird zu Beginn durch den Postmasterprozess allokiert. Die folgenden Prozesse erben von dem Postmasterprozess Zeiger auf diese Strukturen und können darüber auf diese zugreifen. Da die Strukturen im Shared Memory nun von allen Prozessen gelesen und beschrieben werden können, werden sie zur Interprozesskommunikation genutzt.

Lightweight Locks dienen nun dazu den Zugriff auf die Shared Memory Strukturen zu kontrollieren. Die Lightweight Locks werden von einem Prozess erworben, um exklusiven Zugriff auf eine Struktur zu erhalten. Dadurch wird eine Verfälschung der Werte durch unkontrollierten parallelen Zugriff verhindert. Der Lightweight Lockmanager übernimmt dabei die Verwaltung und Zuteilung dieser Sperren.

ShmemVariableCache ist eine Shared Memory Struktur, welche wie in 2.3.2 beschrieben für die Multi-Version Concurrence Control eine wichtige Rolle spielt. Im ShmemVariableCache sind globale Daten über Transaktionen gespeichert. Dazu gehören unter anderem die nächste zu vergebende TransaktionsID und die TransaktionsID der Transaktion, welche als letztes beendet wurde.

2.3.5.2. Tenantspace

Tenantspaces für PostgreSQL wurden in [Sch11] eingeführt. Sie dienen zur Paritionierung der Mandanten in Mandantengruppen. Relationen, die in Tenantspaces getrennt sind, werden auch physisch in getrennten Dateien gespeichert. Dies bedeutet, für jeden Tenantspace und für jede Relation gibt es eine eigene Datei. Dabei hat die Relation für alle Tenantspaces die gleiche ID zur internen Verwaltung. Damit aber die einzelnen Tenantspaces auseinandergehalten werden können, wurde das `suffix`-Attribut in den zur Identifikation der physischen Repräsentation genutzten `RelFileNode` eingefügt.
Durch *Split-* und *Merge*operationen kann angepasst werden, welche Mandanten in einem Tenantspace enthaltenen sind. Dadurch wird die Partitionierung der Mandanten verändert. Im Rahmen dieser Operation müssen die Daten der Mandanten auch physisch umgruppiert werden.
In dieser Arbeit bilden die Tenantspaces die Grundlage für die Migration. Die Aufteilung mittels Split und Merge kann nun dazu genutzt werden, Mandantengruppen für die Migration zu bilden. Dadurch kann auf den aktuellen Ressourcenbedarf einzelner Mandanten eingegangen werden. Dies hat zur Folge, dass eine Lastbalancierung auf Mandantenebene ermöglicht wird.

3. Mandantenfähige Datenbanken im Rechencluster

In der Praxis wird ein Datenbanksystem in einem Rechencluster betrieben, damit die an das System gerichteten Anforderungen erfüllt werden können. Der Cluster besteht aus einem System von Servern, welche untereinander verbunden sind und miteinander kommunizieren. Ziel des Clusterbetriebs ist es, die Anforderungen an ein Datenbankmanagementsystem, wie beispielsweise Verfügbarkeit und Skalierbarkeit, zu erfüllen. Dies gilt ebenso für mandantenfähige Datenbanksysteme.
Für den Aufbau des Clusters und die einzusetzende Software spielt die Systemarchitektur eine wichtige Rolle. Für die Systemarchitektur kann ein *Shared-Nothing*- oder ein *Shared-Disk*-Ansatz genutzt werden. Beim Shared-Nothing-Ansatz ist jeder Knoten im Cluster ein vollwertiger Server, während sich bei Shared-Disk mehrere Serverknoten den Externspeicher teilen.
Ein Datenbankcluster benötigt unabhängig von der zugrunde liegenden Systemarchitektur eine gesonderte Komponente, die clusterweite Verwaltungsaufgaben übernimmt. Zum einen ist diese verantwortlich für das Routing von Anfragen an den entsprechenden Server. Zum anderen übernimmt diese Komponente das Metadatenmanagement des Datenbanksystems. Eine zentrale Stelle für diese Aufgaben ist relativ leicht zu warten und zu implementieren. Sie stellt allerdings auch einen Single-Point-of-Failure und einen potentiellen Flaschenhals dar. Demgegenüber steht die Möglichkeit ein verteiltes Metadatenmanagement einzusetzen. Dies benötigt einen erhöhten Kommunikationsaufwand, zum Beispiel um Sperren auszuhandeln. Diese sind nötig um vor ungewolltem, parallelem Zugriff auf die gemeinsamen Metadaten zu schützen.
In einem Cluster werden dabei Replikations- und Migrationsmechanismen eingesetzt. Diese dienen dazu, die Anforderungen an Verfügbarkeit, Datensicherheit und Skalierbarkeit eines mandantenfähiges Datenbanksystems zu erfüllen. Dabei trägt die Replikation von Daten und Servern zu einer hohen Verfügbarkeit und somit auch zur Datensicherheit bei. Zusätzlich wird die Migration von Daten und Prozessen dazu genutzt, die Ressourcen dynamisch zu skalieren. Damit wird eine Lastbalancierung im Cluster ermöglicht. Diese hat zur Folge, dass Engpässe im Cluster verhindert werden und somit die Gesamtleistung steigt.
Sowohl die Replikation als auch die Migration haben gemeinsam, dass Daten und deren Zugriffsberechtigung von einem Server auf einen anderen übergeben werden. Diese Gemeinsamkeit kann genutzt werden, um aus bestehenden Replikationsansätzen Schlüsse für die Migration zu ziehen.
Für die Systemarchitektur, die dieser Arbeit zugrunde liegt, wurde ein *Shared-Nothing*-Ansatz gewählt. Mit diesem lassen sich die Anforderungen mandantenfähiger Datenbanksysteme an Isolation und Skalierbarkeit unter anderem durch die Partition der Datenbank erfüllen.

In diesem Kapitel werden nun zunächst die Grundlagen, die zur Auswahl des Shared-Nothing-Ansatzes beigetragen haben, erläutert. Dazu wird die Replikation im Rahmen des Datenmanagement in einem Cluster vorgestellt. Darauf folgen die sich aus der Replikation ergebenden Möglichkeiten für die Migration. Auf dieser Grundlage werden im Anschluss verschiedene Systemarchitekturen unter dem Aspekt der Replikation und Migration behandelt. Im letzten Kapitel wird dann die Entscheidung für ein Shared-Nothing-System begründet und die Vorteile der Partitionierung in diesem Zusammenhang erläutert.

3.1. Replikation

Replikationsmechanismen werden dazu genutzt, eine hohe Verfügbarkeit der Daten zu ermöglichen. Gleichzeitig schützt die Replikation auch vor Datenverlust und bildet somit eine Komponente der Datensicherheit. Replikationsmechanismen fertigen im Allgemeinen eine physische Kopie der Daten einer Datenbank an. Im Rahmen des sogenannten *Failovers* wird beim Ausfall des Originals automatisch auf die Kopie zugegriffen. Damit die Kopie auch zur Sicherstellung der Verfügbarkeit der Daten genutzt werden kann, muss die Kopie deswegen regelmäßig aktualisiert werden.
Darüber hinaus gibt es noch weitere Bereiche, in denen die Replikation genutzt wird. Für die Replikation können Server eingesetzt werden, die in unterschiedlichen Lokationen stehen. Dadurch kann das System die Verfügbarkeit nicht nur für lokale Ausfälle eines Servers, sondern auch bei Ausfall eines ganzen Rechenzentrums ermöglichen. Ein anderer Ansatz wird im Rahmen des Data Warehousing genutzt. Dort wird die Replikation der Daten eingesetzt, besipielsweise um die beim Online Analytical Processing anfallenden intensiven Leseoperationen auf das Replikationssystem auszulagern. Dadurch haben diese Operationen keinen negativen Einfluss auf die Last, die durch die normale Transaktionsverarbeitung entsteht.
Die Replikation kann dabei auf unterschiedlichen Ebenen erfolgen, welche nun vorgestellt werden.

3.1.1. Datenbankebene

Auf Datenbankebene wird die Replikation durch sogenannte Master/Slave- oder Multi-Master-Cluster umgesetzt. Die einzelnen Server des Clusters übernehmen die Master- oder Slaverolle. Der Master führt dabei aktiv Transaktionen durch, während der Slave von einem Master abhängt und diesen repliziert.
Man spricht hier von einer Shared-Nothing-Architektur, da bei diesem Ansatz die Rechner des Clusters über komplett eigene Hardware, wie Prozessor, RAM und Speicher, verfügen. Für diese Art von Replikation laufen auf allen Rechnern des Cluster Instanzen des Datenbanksystems. Ein Rechner übernimmt dabei die Aufgabe des Masters und führt die Transaktionen aus. Anschließend werden die dadurch entstandenen Änderungen durch den Slave nachvollzogen. Diese Redundanz trägt zur Datensicherheit des Systems bei. Hochverfügbarkeit erreicht man durch ein schnelles Umschalten des Masters auf den Slaveserver im

Fehlerfall.
In einem Multi-Mastercluster kann durch die Partitionierung und Verteilung von Aufgaben auf verschiedene Masterserver zudem horizontale Skalierbarkeit ermöglicht werden.

3.1.2. Dateiebene

Die Replikation auf Dateiebene kann durch verteilte Dateisysteme oder durch Externspeicherlösungen, wie Network Attached Storage (NAS), erreicht werden.
Verteilte Dateisysteme zeichnen sich hierbei durch einen über den Cluster verteilten Zugriff auf Ebene der Dateien aus. Die einzelnen Dateien einer Datenbank werden auf verschiedenen Rechnern im Netzwerk gespeichert. Dadurch werden die Ressourcen zwischen den Rechnern geteilt und können von allen genutzt werden. Durch eine zusätzliche Replikation der Dateien wird Redundanz und somit Fehlertoleranz ermöglicht.
Der Zugriff auf Dateien braucht neben den üblichen Prozessor- und Plattensuchzeiten auch Zeit für die Übermittlung durch das Netzwerk. Dieser Overhead entsteht, da die Dateien nicht immer lokal vorliegen und über das Netzwerk übertragen werden müssen. Zusätzlich müssen Mechanismen zur Kontrolle des gleichzeitigen Zugriffs umgesetzt werden. Diese müssen durch das verteilte Dateisystem erfüllt werden und führen wiederum zu erhöhtem Aufwand durch zusätzliche Kommunikation.
Ähnlich wie bei verteilten Dateisystemen findet auch bei Externspeicherlösungen über NAS der Zugriff auf Dateien über das Netzwerk statt. Der NAS besteht allerdings nicht aus den lokalen Festplatten der Rechner des Clusters, sondern bildet eine gesonderte Komponente. Diese Komponenten übernimmt die Datei- und Speicherverwaltung für den über das Netzwerk angeschlossene Cluster.
Intern kann die NAS-Komponente die Datensicherheit und Verfügbarkeit sicherstellen, indem sie redundante Hardware und interne Replikation der Daten nutzt. Da der Zugriff über die Dateischnittstelle erfolgt, treten auch hier die für verteile Dateisysteme genannten Nachteile durch den Netzwerkoverhead auf.

3.1.3. Blockebene

Eine dritte Form der Replikation findet direkt auf der Speicherebene statt. Der Zugriff auf die Daten erfolgt hierbei über die einzelnen Blöcke, aus denen sich die Dateien aufbauen. Die Replikation auf dieser Ebene kann entweder innerhalb eines Clustered File Systems oder durch RAID-Technologien erfolgen.
Wenn dies im Rahmen eines Clustered File System geschieht, wird ein globales Dateisystem auf allen Servern des Clusters gemountet. Dies hat zur Folge, dass aus Sicht des Servers alle Dateien lokal vorliegen. Dadurch werden die eigentlichen Netzwerkzugriffe verdeckt.
Der Zugriff auf das Dateisystem selbst erfolgt dann zum Beispiel über ein Storage Area Network (SAN). In diesem wird der Externspeicher verwaltet. Über das Clustered File System wird nun der Externspeicher zwischen den einzelnen Rechnern des Clusters geteilt, man spricht also von einer Shared-Disk-Umgebung. Das SAN kann dabei Replikationen der Daten auf Blockebene vorhalten, sodass die Datensicherheit und Verfügbarkeit gewährleistet

sind. Diese Mechanismen sind transparent für den einzelnen Server und somit auch für die auf ihm laufende Software.
Ein anderer Ansatz zur Replikation auf Blockebene wird durch den Einsatz eines RAID-Systems (Redundant Array of Independent Disks) umgesetzt. Dabei ermöglicht die Konfiguration mehrerer Festplatten in einem RAID-System Fehlertoleranz gegenüber Ausfällen einzelner Festplatten. Unter bestimmten Voraussetzungen ist dadurch auch eine Leistungssteigerung möglich, da in einem RAID-System eine Datei auf mehrere Platten verteilt werden kann und so mehrere Blöcke dieser Datei gleichzeitig gelesen werden können. RAID-Systeme eignen sich allerdings nicht um vor einem Ausfall eines Servers zu schützen.

3.2. Von der Replikation zur Migration

Die Migration unterscheidet sich von der Replikation in der Hinsicht, dass nicht nur Daten kopiert, sondern anschließend auch der Zugriff übertragen wird. Konkret äußert sich der Unterschied darin, dass die Replikation zwar dauerhaft stattfindet, aber erst im Fehlerfall genutzt wird. Demgegenüber ist die Migration zeitlich begrenzt und sollte möglichst schnell beendet werden, um den Einfluss durch den Migrationsoverhead auf das System gering zu halten. Hinzukommt, dass bei der Replikation in der Regel der gesamte Datenbestand betroffen ist, um die Anforderung an Hochverfügbarkeit zu erfüllen. Im Gegensatz dazu, kann die Migration auf eine Partition der Daten begrenzt werden.
Wenn man die Mechanismen der Replikation betrachtet, kann man erkennen, dass sich einige für die Migration nutzen lassen. Dies liegt daran, dass auch bei der Replikation im Fehlerfall durch den *Failover* der Zugriff und die Bearbeitung auf das Zielsystem übergeht. Diese Übergabe ist bei die Migration neben der Übertragung der Daten die zentrale Aufgabe. Bei der Migration liegt der Schwerpunkt auf der Skalierung. Deshalb reicht es aus, nur die Partitionen der Daten zu behandeln, welche ein verändertes Lastaufkommen erfahren. Der grundsätzliche Mechanismus zur Anfertigung einer Kopie und deren Aktualisierung kann dabei von Replikations- auf Migrationsansätze übertragen werden.
Die Migrationstechniken sind für die Skalierbarkeit von besonderer Bedeutung, da sie ermöglichen die zur Verfügung stehenden Ressourcen dynamisch an sich ändernde Lastanforderungen anzupassen. Die Migration überträgt dazu sowohl Daten als auch die Zugriffsrechte vom Quellserver auf einen Zielserver. Dabei bilden die zu migrierenden Daten eine abgeschlossene Partition der Datenbank. Dies hat den Vorteil, dass der Zielserver bei Abschluss der Migration unabhängig von der Quelle auf dieser Partition operieren kann.
Der einfachste Migrationsansatz wird als **Stop and Copy** bezeichnet. Dabei wird der Zugriff auf die zu migrierenden Daten auf der Quelle komplett angehalten. Anschließend werden diese Daten auf das Ziel kopiert. Wenn der Kopiervorgang vollständig abgeschlossen ist, werden die logischen Zugriffsrechte auf den Zielserver übertragen. Die Datenbank kann nun auf dem Zielserver mit den migrierten Daten arbeiten und wieder Anfragen bearbeiten. Durch die komplette Unterbrechung der Anfragen ist diese Technik nicht mit den Anforderung an die Verfügbarkeit vereinbar.
Bei Online Migration-Verfahren besteht nicht der Nachteil von Stop and Copy, dass die Bearbeitung von Anfragen für eine längere Zeit unterbrochen werden muss. Bei der Online

Migration sollen während der gesamten Migrationsphase Anfragen an die Datenbank beantwortet werden können. Ziele sind hierbei die Isolation und Konsistenz von Transaktionen einzuhalten, den Einfluss durch die Migration auf die Bearbeitungszeit möglichst gering zu halten, sowie möglichst wenige Anfragen abbrechen oder zurückweisen zu müssen.

3.3. Systemarchitektur

Die Systemarchitektur in einem Rechencluster für ein mandantenfähiges Datenbanksystem kann durch den Shared-Nothing- oder den Shared-Disk-Ansatz erfolgen. Diese Ansätze bestimmen die für den Betrieb des Clusters nötige Hardware und die auf den Servern des Clusters laufende Software. Zu dieser Software zählen Betriebs- und Dateisysteme, welche die Grundlage für das im Cluster betriebene Datenbanksystem bilden.
Die Shared-Nothing-Architektur setzt beim Aufbau des Clusters auf dedizierte Server, welche auch unabhängig von den anderen Rechnern betrieben werden könnten. Diese werden im Cluster zusammengeschaltet und kommunizieren unter einander über Netzwerkschnittstellen. Das auf den Servern installierte Betriebssystem ist ebenfalls clusterunabhängig. Die Clusterfunktionalität wird dann auf diese Ausgangsbasis aufgesetzt. Dies wird erreicht, indem die Applikationen ein Bewusstsein über die Clusterumgebung besitzen und so auch Verwaltungsaufgaben übernehmen können. Solch ein Programm kann entweder direkt das Datenbankmanagementsystem sein oder ein externes Programm, welches extra zur Übernahme dieser Clusteraufgaben eingesetzt wird.
Demgegenüber steht die Shared-Disk-Architektur, welche eine Trennung zwischen Externspeicher und den einzelnen Servern vornimmt. Diese Trennung kann entweder auf Hardwareebene, beispielsweise wie in 3.1.2 beschrieben durch das Nutzen eines NAS, oder auf Softwareebene erfolgen. Eine softwareseitige Shared-Disk-Architektur wird beispielsweise über ein Clustered File System erzeugt. Dieses fasst den gesamten Externspeicher aller Rechner des Clusters in einem Dateisystem zusammen und macht ihn einheitlich für alle zugänglich. Für das darauf aufsetzende Betriebssystem und somit auch für die im Cluster betriebene Datenbank geschieht dies transparent. Dadurch wird erreicht, dass alle Server im Cluster über dieses gemeinsame Dateisystem vollen Zugriff auf alle sich im Cluster befindliche Daten haben.
In den folgenden Abschnitten werden verschiedene Ansätze zur Replikation für diese beiden Architekturmöglichkeiten vorgestellt. In 3.2 wurde der mögliche Einfluss der Replikation auf die Migration erläutert. Darauf aufbauend, werden die beiden Architekturen nach ihren Ansätzen zur Replikation untersucht. Dies dient dem Ziel, Anhaltspunkte für eine Systemarchitektur eines Migrationssystem für eine mandantenfähige Datenbanken in einem Rechencluster herauszuarbeiten.

3.3.1. Replikation mit Shared-Nothing-Architektur

Es gibt verschiedene Methoden bei einer Shared-Nothing-Architektur die Replikation durchzuführen. Sie unterscheiden sich meistens in der Art und Weise, in der die Kopie aktualisiert

wird. Die nun folgenden Ansätze wurden in [Pos09] vorgestellt. Diese sind teilweise bereits für PostgreSQL, welches die Grundlage für den Prototypen dieser Arbeit bildet, implementiert. Die nun folgenden Ansätze konkretisieren die in 3.1.1 beschriebenen Ansätze zur Replikation auf Datenbankebene. Dabei gliedern sie sich in drei Gruppen:

Master-Slave -Methoden besitzen eine klare Aufteilung der Aufgaben. Sie unterscheiden sich allerdings in der Durchführung der Replikation voneinander.

Multi-Master -Replikation hat zur Folge, dass jeder beteiligte Server eine Replikation der Daten besitzt. Die Masterserver haben daher vollen Zugang zu allen Daten.

Statement-Based Replication Middleware ist der Datenbank selbst übergeordnet. Diese Stellung wird genutzt, um die Replikation der Datenbank von außen durchzuführen und zu kontrollieren.

3.3.1.1. Master-Slave Replikation

Bei Master-Slave-Ansätzen zur Replikation führt der Master die Transaktionen aus und der Slave verwaltet die Kopie. Dazu muss der Master die Änderungen an den Daten dem Slave mitteilen. Dies kann entweder mittels der Write-Ahead-Log-Einträge oder mittels Trigger geschehen. Write-Ahead-Log-Einträge werden unabhängig von der Replikation zur Datensicherheit bei jeder Schreiboperation erzeugt. Eine Schreiboperation löst bei dem zweiten Ansatz einen Trigger aus, sodass die Änderungen zeilenweise erfasst und übertragen werden können.
Den Ausgangspunkt für die Replikation auf dem Slave bildet im allgemeinen ein Checkpoint oder ein Backup des Masters, welches anschließend kontinuierlich aktualisiert wird. Die Aktualisierung findet meist asynchron statt, also nachdem der Master eine Transaktion bereits erfolgreich mit Commit beendet hat. Dadurch kann es bei einem Ausfall zu einem begrenzten Datenverlust kommen. Diese Einschränkung wird aus Performancegründen in Kauf genommen. Ansonsten müssten die Transaktion warten, bis die Änderungen über das Netzwerk zum Slave übertragen wurden. Darüber hinaus können durch diese Verzögerung mehrerer Änderungen in einem Paket zusammengefasst werden. Damit kann der Overhead durch die Netzwerkprotokolle und somit die Netzwerkbelastung verringert werden.
Der Slave aktualisiert seine Daten laufend anhand der Änderungen, die er vom Master übermittelt bekommt. Mittels Heartbeat-Nachrichtenaustausch wird ermittelt, ob der Master noch aktiv ist. Wenn dies nicht der Fall ist, dann kann der Slave im Rahmen des Failovers direkt die Aufgaben des Masters übernehmen. Somit wird der Betrieb nicht unnötig unterbrochen und wird auf dem Slave fortgesetzt. Änderungen, die der Master vor seinem Ausfall committet hat, aber noch nicht an den Slave kommunizieren konnte, gehen dabei aufgrund der asynchronen Übertragung zunächst verloren. Über Recovery-Maßnahmen können sie allerdings teilweise wieder manuell hergestellt werden.
Im folgenden werden drei Methoden der Master-Slave-Replikation für PostgreSQL vorgestellt. Wie oben erwähnt sind diese aus Performancegründen asynchron und können so beim Ausfall des Masters zu einem Datenverlust führen.

Trigger-basierte Replikation Das in [Wie] vorgestellte Slony-l ist ein PostgreSQL-Projekt mit dem Ziel eine triggerbasierte Replikation zu ermöglichen. Es nutzt AFTER ROW Trigger, um eine SQL-Query zu reproduzieren, welche die Änderungen auf dem Tupel wiederspiegelt. Diese reproduzierte SQL-Query wird dann an den Slave übertragen und dort ausgeführt. Dadurch wird das entsprechende Tupel auf dem Slave aktualisiert.
Auf den Slaves können zusätzlich read-only Anfragen durchgeführt werden. Dies hat das Ziel den Master zu entlasten. Das Slony System kann dazu mehrere Slaves pro Master verwalten. Diese sind kaskadiert, sodass der Master nur einen Slave versorgen muss. Dieser reicht die Änderungen an die anderen Slaves weiter, wodurch der Master entlastet wird. Dies ermöglicht eine höhere Datensicherheit, da beim Failover trotz Ausfall des Masters neben dem neuen Master auch noch mindestens ein Slave existiert, auf den die Daten repliziert wurden.
Das Nutzen der Trigger auf dem Master kann dessen Leistung im Vergleich zu den anderen Methoden allerdings negativ beeinflussen.

Continuous Archiving mittels Point in Time Recovery Für die Master-Slave Replikation mittels Point in Time Recovery ist es zunächst erforderlich, dass ein Backup des Masters auf den Slave kopiert wird. Dieser stellt dann das Backup mittels der Restorefunktionalität wieder her. Anschließend wird dem Slave regelmäßig das neueste Segment des Write-Ahead-Logs des Masters übertragen. Dieses spielt der Slave in seine Datenbank ein, welche dadurch aktualisiert wird. Ein Write-Ahead-Log-Segment enthält die Einträge mehrerer Änderungen, die somit gebündelt übertragen werden.
Diese Methode verursacht beim Master nahezu keinen Overhead, da er die Write-Ahead-Logs sowieso für die Datensicherheit anfertgigt.
Wie bei dem im vorigen Abschnitt beschriebenen Slony können die Slaves read-only Anfragen beantworten, um den Master zu entlasten. Ein Nachteil im Vergleich zur Replikation auf einer höheren Ebene wie über SQL-Statements ist, dass das Übertragen und Einspielen des Write-Ahead-Logs zwischen unterschiedlichen Datenbankversionen nicht möglich ist, wenn sich das Logformat ändert.

Streaming Replication Bei der Streaming Replication werden im Gegensatz zum Continuous Archiving nicht die ganzen Segmente des Write-Ahead-Logs vom Master an den Slave übertragen, sondern die einzelnen Write-Ahead-Log-Einträge. Dieses Verfahren ist auch asynchron, da das Commit schon vor der vollständigen Aktualisierung des Slaves erfolgt. Die Zeitspanne zwischen dem Commit auf dem Master und der Aktualisierung des Slaves ist aber im Vergleich zum Continuous Archiving erheblich kürzer. Dadurch sinkt der Umfang des möglichen Datenverlust durch den Ausfall des Masters. Dieses Verfahren wurde wie in [Pos10] beschrieben in PostgreSQL 9.0 implementiert.

3.3.1.2. Synchrone Multi-Master Replikation

Bei der Multi-Master Architektur haben alle Server des Clusters die vollen Rechte. Jeder Server darf somit auf die gesamte Datenbank lesend und schreibend zugreifen. Zur Replikation werden alle Änderungen eines Servers auf alle anderen Masterserver übertragen. Dadurch hat jeder Master trotz Shared-Nothing-Architektur den vollen Datenbestand lokal vorliegen und kann auf ihn zugreifen.
Da auf alle Replikate geschrieben werden darf, kommt es zu Schreibkonflikten. Um diese aufzulösen, muss ein clusterweites Sperrverfahren implementiert werden. Dies führt zu einer schlechten Performance bei der Bearbeitung von Schreibanfragen, da jeder Master für die zu bearbeitenden Daten ein globale Sperre auf allen Servern des Netzwerks erhalten muss. Zusätzlich ist die Netzwerkbelastung hoch, da alle Änderungen an alle Server weitergegeben werden müssen.
Diese Lösung eignet sich ausschließlich für eine Datenbanklast, die beinahe ausschließlich aus Leseanfragen besteht.

3.3.1.3. Statement-Based Replication Middleware

Ein anderer Ansatz zur Replikation für Shared-Nothing-Systeme besteht darin, über der Datenbank eine weitere Middlewareschicht einzuziehen. Diese Middleware übernimmt dann die Verwaltung und Anfertigung der Replikation des Datenbestandes.
Bei Statement-Based Replication Middleware fängt diese Schicht die SQL-Anfragen ab und leitet sie an den entsprechenden Datenbankserver im Cluster weiter. Die Replikation wird in diesem Fall dadurch erreicht, dass jede SQL-Anfrage an mehrere Datenbankserver übergeben wird. Die Datenbankinstanzen führen die Anfragen dann unabhängig voneinander aus und aktualisieren so ihre lokalen Daten. Da sie die gleichen Anfragen auf den gleichen Ausgangsdaten ausführen, erhalten sie auch die gleichen Ergebnisse. Dies führt zu einer Replikation der Datenbank.
Spezielle lokale Berechnungen, wie ein Zufallswert oder ein Zeitstempel, müssen allerdings gesondert behandelt werden. Dazu sind beispielsweise Rückfragen an die Middleware nötig, die diese Werte dann global festlegt.
Dieses Verfahren ermöglicht durch die Redundanz Datensicherheit und Hochverfügbarkeit. Es bietet allerdings kein direktes Verfahren zur Skalierbarkeit. Lediglich über die Partitionierung der Datenbank in mehrere getrennte Bereiche, kann Skalierbarkeit ermöglicht werden. Diese Trennung würde aber darauf hinauslaufen, dass jede Partition einen eigenen Server nutzt. Dies verringert wiederum die Möglichkeiten zur Konsolidierung.

3.3.2. Shared-Disk-Architekturen

Die Shared-Disk-Architektur unterscheidet sich von der Shared-Nothing-Architektur generell dadurch, dass neben dem physischen Server noch eine weitere Komponente die Grundlage für den Betrieb des Datenbanksystems bildet. Diese Komponente ist dafür verantwortlich, dass alle Server im Cluster Zugriff auf die gesamten persistenten Daten haben. Die Daten

werden also in der Shared-Disk-Komponente gebündelt gespeichert und verwaltet. Das Shared-Disk-System besteht, wie in 3.1.3 beschrieben, entweder aus einer Hardware- oder einer Softwarekomponente. Die Hardwarekomponente beinhaltet physisch die Festplatten des Systems und macht diese über das Netzwerk dem Cluster zugänglich. Ein Clustered File System ist ein Beispiel für eine Shared-Disk-Softwarekomponente, welche die Externspeicherressourcen der Rechner des Clusters bündelt und zur Verfügung stellt.
Die Datensicherheit und Hochverfügbarkeit auf Speicherebene kann bei Shared-Disk-Systemen durch einen Mirrored Storage erfolgen. Dabei wird das Shared-Disk File System auf einen zweiten Satz an Externspeichermedien gespiegelt. Die dadurch entstehende Replikation wird auf Blockebene aktualisiert, sodass Original und Spiegel immer den gleichen Zustand besitzen. Bei einem Ausfall steht also immer noch der Spiegel zur Verfügung. Dadurch ist der Zugriff auf die persistenten Daten durch das Datenbanksystem weiterhin möglich.
Dadurch dass die gesamten persistenten Daten des Clusters für alle Server zugänglich sind und diese aus Datenbanksicht extern verwaltet werden, ergeben sich für die Replikation und Migration andere Voraussetzungen als bei Shared-Nothing-Systemarchitekturen. Diese Unterschiede werden nun im folgenden erläutert.

Vorteile Der Shared-Disk-Ansatz kann dadurch überzeugen, dass automatisch jeder Knoten Zugriff auf die gesamte Datenbank besitzt. Mechanismen für den Failover und die Migration sind deshalb unabhängig von den persistenten Daten auf dem Shared-Disk-Externspeicher. Darüber hinaus wird die Zugriffsisolation durch das Shared-Disk-System gewährleistet. Sperren für den Zugriff müssen also nicht extern verwaltet werden und sind deshalb unabhängig von der Datenbank. Des Weiteren entsteht im Cluster keine Overhead durch Synchronisation der einzelnen Server, da nur eine Version der Daten existiert.
In [Hog] werden außerdem auf die Vorteile der Shared-Disk-Architektur für die Verfügbarkeit des Systems hingewiesen. Da der Externspeicher getrennt von den Servern verwaltet werden kann, ist dieser beim Ausfall eines Knotens immer noch für die andere Server im Cluster erreichbar. Im Rahmen des Failovers übernimmt dann direkt ein anderer Knoten die Anfragen.

Nachteile Allerdings gibt es neben den genannten Vorteilen auch Schwächen des Shared-Disk-Ansatzes. Diese entstehen dadurch, dass die einzelnen Knoten die Zugriffe auf persistente Daten über das Netzwerk ausführen müssen. Dieser Zugriff auf entfernte Ressourcen kann laut [Ous90] und [CTL+05] zu einer Verlangsamung des gesamten Systems führen. Außerdem hat, wie in [Sah05] beschrieben, der Protokolloverhead ebenfalls starken Einfluss auf die Leistung des Systems. Bei Datenbanksystemen, die ein solches System als Basis haben, kann es dadurch zu erheblichen Leistungseinbußen beim Zugriff auf Datenbankseiten auf dem Externspeicher kommen.
Demgegenüber stehen Ansätze, die durch Lokalisation der Daten diesen Nachteil ausgleichen wollen. Ein solcher Ansatz, der durch Prefetching versucht die Daten lokal vorzuhalten, wurde in [GA94] vorgestellt.

Darüber hinaus ist das Nutzen einer zusätzlichen Komponente, welche Shared-Disk ermöglicht, relativ teuer und aufwändig. Dies trifft beispielsweise auf die Beschaffung dedizierter Hardware für die Externspeicherverwaltung zu. Clustered File Systeme können aufwändig zu integrieren und zu warten sein. Dies führt dazu, dass Shared-Disk-Ansätze zusätzliche Voraussetzungen für den Betrieb mandantenfähiger Datenbanksysteme schaffen. In der Praxis könnte dies gegen einen Einsatz eines darauf basierenden Datenbanksystems sprechen.

3.4. Auswahl eines Ansatzes

Mit Shared-Nothing und Shared-Disk stehen zwei unterschiedliche Ansätze zur Systemarchitektur eines Rechenclusters für ein mandantenfähiges Datenbanksystem zur Auswahl. Für diese Arbeit ist neben den allgemeinen Anforderungen an ein mandantenfähiges Datenbanksystem insbesondere die Eignung zur effizienten Migration im Rahmen der horizontalen Skalierbarkeit entscheidend. Als Grundlage für den Entwurf und die Implementierung des Prototyps als Teil dieser Arbeit wurde eine Shared-Nothing-Systemarchitektur ausgewählt. Die Migration selbst wiederum soll auf einer Master-Slave-Lösung basieren.
Die Entscheidung einen Shared-Nothing-Cluster als Grundlage vorauszusetzen hat mehrere Gründe. Der erste Punkt ist, dass sich ein Shared-Nothing-System gut skalieren lässt. Dies liegt zum einen daran, dass das Netzwerk nicht durch Datenzugriffe auf entfernte Ressourcen belastet wird. Es entsteht also kein Flaschenhals im Netzwerk, welches die maximale Skalierbarkeit einschränkt. Zum anderen können die dem Cluster zur Verfügung stehenden Ressourcen leicht erweitert werden. Es reicht aus einen neuen Server an den Cluster anzuschließen. Die Datenbanklogik übernimmt dann die Integration des neuen Knoten.
Des Weiteren ist es nicht notwendig bei einem Shared-Nothing-Cluster in spezielle Hardware oder clusterfähige Software zu investieren, da die die Datenbanklogik diese Aufgaben übernimmt. Dadurch besteht hohe Flexibilität beim Einsatz der zugrunde liegenden Hardware und Software.
Darüber hinaus hat eine Shared-Nothing-Architektur zur Folge, dass die Datenbanklogik sich von den Systemvoraussetzungen trennen lässt. Die Datenbanklogik übernimmt die Aufgaben zur Verwaltung und Sicherung der Daten und erfüllt so die Anforderungen an das System. Die Replikation und Migration können direkt durch das Datenbankmanagementsystem ermöglicht werden und es muss nicht auf proprietäre Lösungen zurückgegriffen werden. Durch diese klare Trennung wird die Implementierung und die Wartung des Systems vereinfacht.
Als letzter Punkt spricht die Möglichkeit zur Partitionierung für einen Shared-Nothing-Ansatz als Systemarchitektur. Die Partitionierung lässt sich aufgrund der Auftrennung der Daten in Mandanten und Mandantengruppen relativ leicht erreichen. Im Zusammenhang mit Shared-Nothing-Servern lässt sich diese Mandantenpartitionierung auch auf eine Partitionierung des Clusters übertragen, was weitere Vorteile mit sich bringt. Diese werden im folgenden Abschnitt vorgestellt und im Bezug auf mandantenfähige Datenbanksysteme erläutert.

3.4.0.1. Partitionierte Shared-Nothing-Systeme

Eine Partitionierung von Shared-Nothing-Systemen bringt einige Vorteile, welche beispielsweise in [Sha10] vorgestellt wurden. Die Vorteile basieren darauf, dass eine effiziente Partitionierung der Daten möglich ist. Im Rahmen von mandantenfähiger Datenbanken ist dies durch die in 2.3.5.2 vorgestellten Tenantspaces möglich. Durch Tenantspaces lassen sich zum einen die Daten nach Mandantengruppen partitionieren und zum anderen existiert eine effiziente Möglichkeit, diese Partition zu ändern und anzupassen. Durch eine Anpassung kann auf aktuelle Nutzungsmuster eingegangen und so Lastspitzen abgefangen werden. Diese Vorteile für die Skalierbarkeit durch die Partitionierung wurden auch in [DG90] betrachtet. Darüber hinaus hat diese Partitionierung zur Folge, dass die Leistung des gesamten Systems steigt, da nicht ein einzelner Server zum Flaschenhals wird.
Ein Nachteil ist, dass Operationen aufwändiger werden, die mehrere Partitionen umspannen. Das liegt daran, dass die Partitionierung auch die Zugriffsrechte für diese Daten im Cluster auf einzelne Server einschränkt. Aufgrund der geforderten Isolation zwischen den Mandanten sind solche Anfragen für mandantenfähige Datenbanksysteme im Allgemeinen nicht vorgesehen.
Ebenfalls entscheidend ist, dass durch die Partitionierung des Shared-Nothing-Clusters eine sehr große Menge an Nutzern verwaltet werden kann. Diese Skalierbarkeit folgt daraus, dass die Bearbeitung durch die Partitionierung in größtenteils unabhängigen Bereichen erfolgt. Dies ist wichtig für den Betrieb eines mandantenfähigen Datenbanksystems in einem Cluster, da damit erreicht werden kann, viele Nutzer auf einem Datenbanksystem zu konsolidieren.

4. Entwurf

Die in den vorherigen Kapiteln erläuterten Grundlagen, Voraussetzungen und Anforderungen werden nun in einem Entwurf zur Pufferskalierung und Live Migration umgesetzt. In diesem Kapitel wird zunächst ein Ansatz ausgewählt und vorgestellt. Darauf folgt die Beschreibung dessen einzelner Module und die Erläuterung deren Funktionsweise. Die konkrete Implementierung des Entwurfes in Form eines Prototyps erfolgt dann im nächsten Kapitel.

4.1. Einordnung des Entwurfs

Die Grundlage für die Auswahl eines Entwurfs zur horizontalen Skalierung stellen Mechanismen zur Migration und Synchronisation dar, welche in den folgenden Abschnitten erläutert werden. Ausgangspunkt ist dabei, dass das Datenbanksystem in einer Clusterumgebung arbeitet. Dies bedeutet, es stehen mehrere Server zur Verfügung, mit deren Hilfe Skalierung ermöglicht werden soll.
Zunächst ist allerdings festzustellen, dass die Pufferskalierung selbst einen Teilschritt der Live Migration einer Datenbank bilden kann. Bezogen auf ein mandantenfähiges Datenbanksystem ergibt sich hier der Vorteil, dass sich sowohl die Pufferskalierung als auch die Live Migration effektiv auf einen Teilbereich der Datenbank eingrenzen lassen. In Bezug dieses Entwurfs werden für diese Teilbereiche die Tenantspaces angenommen. Ziel des Entwurfs ist es, dass die Daten ohne größere Unterbrechung von Transaktionen und somit der Erreichbarkeit des Services, sowie ohne gravierende Verschlechterung der Antwortzeiten, von einem System auf ein anderes migriert werden können. Die Migration findet dabei vom sogenannten Quell- auf das Zielsystem statt.
Für die Migration gibt es zwei Anwendungsformen für eine Datenbank. Das Ziel des Scale-Outs ist die Entlastung des Quellsystems. Beim Scale-Down hingegen soll die Konsolidierung zurück auf ein einzelnes System durchgeführt werden. Während die Skalierung des Puffers zunächst das Quellsystem entlasten kann, müssen zur dauerhaften Entlastung auch die persistenten Daten übertragen werden. Damit wird dann die Quelle vollständig von Disk-I/O dieser Daten entlastet. Diese Übertragung der persistenten Daten wird als Migration bezeichnet und kann durch verschiedene Ansätze, welche im folgenden Abschnitt erläutert werden, realisiert werden.
Im darauf folgenden Abschnitt werden verschiedene Ansätze zur Synchronisation behandelt. Synchronisation ist für eine Live Migration notwendig, da wie im vorliegenden Fall der Querybetrieb während der Migration nicht gestoppt werden soll.

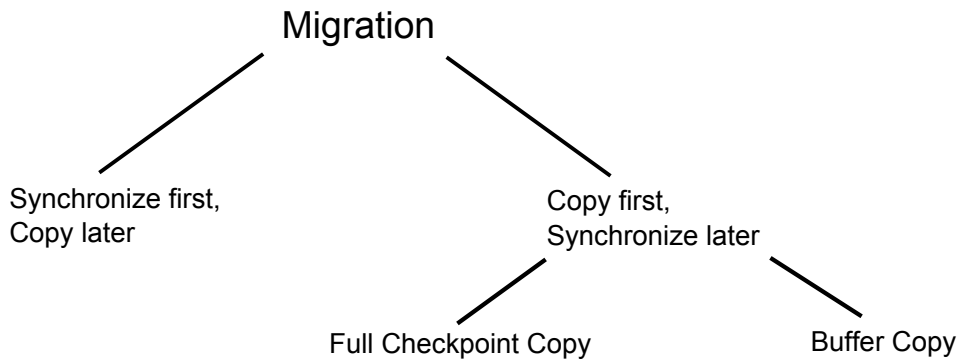

Abbildung 4.1.: Übersicht unterschiedlicher Ansätze zur Migration

4.1.1. Migration

Die in Abbildung 4.1 eingeordneten Ansätze zur Live Migration umfassen im Groben zwei logische Schritte. Der eine ist die physische Kopie der Daten. Dem gegenüber steht die Synchronisation der aktuellen Transaktionen. Die Synchronisation wird notwendig, da der Datenbankservice während der Migration nicht unterbrochen werden soll und folglich Transaktionen auf dem Quell- und Zielsystem gleichzeitig ablaufen können. Damit diese Transaktionen serialisierbar sind und die Konsistenz der Datenbank erhalten bleibt, müssen diese synchronisiert werden.
Die Ansätze zur Live Migration unterscheiden sich zunächst in der Reihenfolge dieser beiden Schritte.
Im ersten Ansatz wird zuerst mit der Synchronisation zwischen Quelle und Ziel begonnen, um anschließend die Daten vollständig zu migrieren. Es werden also schon zu Beginn Transaktionen auf dem Zielsystem gestartet, welche sich mit den Transaktionen auf der Quelle synchronisieren. Durch diesen Ansatz wird die Quelle schneller entlastet, da die Transaktionen früher auf dem Zielsystem beginnen können. Mit Zephyr wurde in [EDAA11] ein zu diesem Vorgehen ähnlicher Ansatz vorgestellt.
Sobald bei diesen Ansätzen nun alle Transaktionen auf dem Quellsystem beendet sind und somit die direkte Synchronisation abgeschlossen ist, werden im zweiten Schritt nun die fehlenden Daten von der Quelle auf das Ziel kopiert. Dabei werden sowohl die persistenten Daten, als auch die noch nicht synchronisierten Daten, die sich noch im Puffer des Quellsystems befinden, übertragen. Sobald diese Daten komplett auf das Ziel kopiert wurden, ist die Migration abgeschlossen und der normale Betrieb geht auf dem Ziel weiter.
Im Gegensatz dazu stehen Ansätze, die zunächst mit der Kopie der Daten beginnen und anschließend die Synchronisation durchführen. Dies ähnelt dem in [DNAA10] als `Iterative Copy` vorgestellten Ansatz.
Zur physischen Kopie der Daten können mittels eines Checkpoints die Daten zunächst auf Festplatte ausgeschrieben werden, um diese dann anschließend auf das Zielsystem zu kopieren. Dieser Ansatz wurde in der Abbildung als `Full Checkpoint Copy` bezeichnet.
Beim `Buffer Copy` Ansatz werden nicht die kompletten Daten im Rahmen eines Check-

points, sondern lediglich die Daten, die sich im Puffer des Quellsystems befinden, kopiert. Da während des Kopiervorgangs weiterhin Datenänderungen auf dem Quellsystem anfallen, müssen auch diese geänderten Daten anschließend auf das Zielsystem übertragen werden. Unter der Annahme, dass das Einbringen der Änderungen auf dem Ziel schneller abläuft als das Erzeugen dieser, liegt der Zustand des Zielsystems nach wenigen Iterationen nur noch um wenige Änderungen hinter der Quelle zurück.
Im zweiten Schritt werden alle neuen Transaktionen ausschließlich auf dem Zielsystem gestartet. Beide Ansätze haben hier im Vergleich zur direkten Synchronisation mit anschließender Kopie, den Vorteil, dass neue Transaktionen auf dem Ziel direkt mit einem **warmen Cache** starten. Die Transaktionen synchronisieren sich nun während der Ausführung mit den Transaktionen auf der Quelle. Die Synchronisation dauert solange, bis alle noch während des Kopiervorgangs gestarteten Transaktionen auf der Quelle abgeschlossen sind. Es liegen nun alle Daten und Transaktionen auf dem Ziel vor und die Migration ist abgeschlossen.

4.1.2. Synchronisation

In Abbildung 4.2 sind unterschiedliche Methoden zur Durchführung der Synchronisation dargestellt. Die Unterteilung erfolgt zunächst in einen `eager`- und in einen `lazy`-Ansatz. Der `eager`-Ansatz zur Synchronisation sieht vor, dass das Quell- und Zielsystem immer den gleichen physischen Zustand der Daten haben. Jegliche Änderungen der Daten müssen also synchron auf beiden Systemen ausgeführt werden. Dieser eher pessimistische Ansatz geht davon aus, dass die Transaktionen auf Quelle und Ziel oftmals auf die gleichen Datenbereiche zugreifen und somit eine sofortige und direkte Synchronisation zwischen den beiden Systemen von Vorteil ist.
Der `lazy`-Ansatz führt im Gegensatz dazu die Synchronisation zwischen Quelle und Ziel erst auf Anforderung aus. Dass heißt, bevor eine Transaktion auf Daten zugreift muss sie überprüfen, ob auf ihrem System die Daten in der aktuellsten Version vorliegen. Ist dies nicht der Fall müssen diese nun zwischen den beiden Systemen synchronisiert werden. Der Vorteil dieses Ansatzes ist es, dass nicht alle Änderungen sofort synchronisiert werden müssen und somit der dabei entstehende Overhead verringert wird. Diese nach Bedarf ausgeführte Synchronisation lohnt sich besonders, wenn man davon ausgeht, dass Konflikte selten auftreten. Ein Konflikt ist hierbei, wenn auf den selben Datenbereich von beiden Transaktionen auf

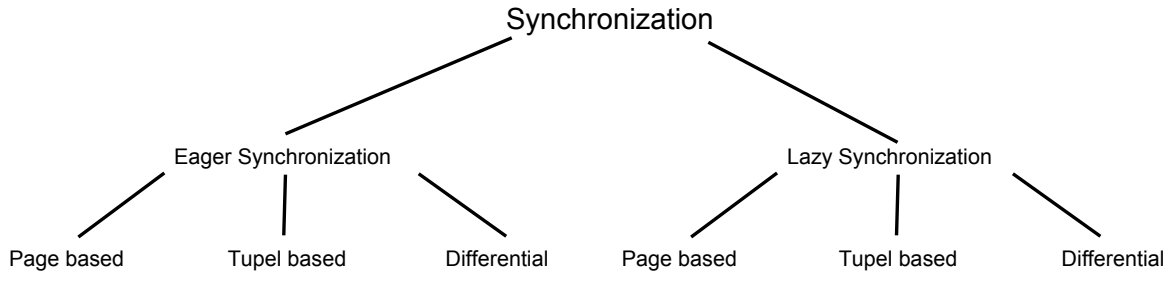

Abbildung 4.2.: Übersicht unterschiedlicher Ansätze zur Synchronisation

beiden Systemen während der Migration zugegriffen wird. Es handelt sich somit um einen optimistischen Ansatz, der davon ausgeht, dass die eigentliche Synchronisation nur in relativ wenigen Fällen ausgeführt werden muss.
Die Synchronisation selbst kann wiederum auf verschiedenen Ebenen stattfinden.
Wird die Synchronisation auf Ebene der Seiten ausgeführt, müssen diese zunächst auf beiden Systemen mit einem Lock gesperrt werden, sodass nur noch ein System Zugriff darauf hat. Die Synchronisation selbst findet dann durch Übertragen der gesamten Seite von einem auf das andere System statt. Durch Überschreiben der alten Seite mit der geänderten Seite haben beide Systeme nun wieder den aktuellen Zustand und die Synchronisation ist abgeschlossen.
Auf Ebene der Tupel wird die Synchronisation durch Übertragung des geänderten oder neuen Tupels durchgeführt. Durch Sperren wird wiederum die physische Integrität gewährleistet, sodass das übertragene Tupel in die entsprechende Seite eingefügt werden kann. Durch dieses Einfügen wird somit die Synchronisation vollzogen.
Als dritte Möglichkeit können zur Synchronisation die Differenz einer Seite zu ihrer Version vor der Änderung genutzt werden. Diese Differenz stellt im allgemeinen den physischen Unterschied, der durch die Änderung der Seite entstanden ist, dar. Diese Differenz wird in Datenbanksystemen häufig, zum Beispiel in Form von Log-Einträgen, zur Recovery erzeugt. Durch Übertragung und Einspielen dieser Differenzwerte kann die entsprechende Seite auf dem anderen System aktualisiert und somit synchronisiert werden.
Natürlich besteht auch die Möglichkeit diese Synchronisationsansätze miteinander zu kombinieren. So kann es sinnvoll sein, Seiten die von beiden Systemen genutzt werden, `eager` auf Ebene der Seitendifferenzen zu synchronisieren. Wohingehen Seiten, die momentan nur von einem System genutzt werden, erst nach Bedarf, also `lazy`, und dafür auf Seitenebene synchronisiert werden.

4.1.3. Mathematische Abschätzung

Für die Durchführung der Pufferskalierung beziehungsweise Live Migration gibt es, wie in den vorherigen Abschnitten gesehen, viele unterschiedliche Ansätze. Eine mathematische Abschätzung des Migrationsaufwandes soll als Grundlage für den Entwurf und die Auswahl eines Ansatzes dienen. Dazu werden zunächst die Anforderungen aufgezählt und daraus die Folgen für die Leistung des Systems abgeschätzt.
Es ist von einer Datenbank mit einer Größe in der Größenordnung von 100 Gigabyte auszugehen. Diese Daten stammen von mehreren tausend, relativ kleinen Mandanten. Jeder Mandant besitzt somit im Schnitt jeweils Daten im zweistelligen Megabytebereich. Diese Mandanten sind in mehreren Tenantspaces organisiert. Es ist davon auszugehen, dass ein Tenantspace zwischen 5-10 Gigabyte groß ist und somit mehrere hundert Mandanten umfasst.
Die Server besitzen einen großen Hauptspeicher im Bereich von 16 bis 32 Gigabyte, der unter anderem von der PostgreSQL-Datenbank direkt als `Shared Buffer` (im nachfolgenden Puffer genannt) und auf Betriebssystemebene als Cache genutzt wird. Es wird empfohlen, dass das Volumen des Puffers 50% des zur Verfügung stehenden Hauptspeichers ausmachen sollte, da ansonsten für die anderen Aufgaben des Datenbank- oder Betriebssystems nicht mehr genug Ressourcen übrig bleiben. Zu diesen Aufgaben zählen unter anderem das Caching, aber auch temporär erzeugte Daten, die zum Beispiel bei der Ausführung von Joins entstehen. Da

die angenommene OLTP-Last nur verhältnismäßig kleine Joins verursacht, sollte der Puffer im oberen Bereich dieser Empfehlungen liegen. Es ergeben sich somit für dieses System ein Puffer mit einer Größe zwischen 8 GB und 16 GB.
Unter diesen Voraussetzungen befinden sich grob überschlagen also ein Zehntel der Datenbank im Puffer. Eine PostgreSQL-Seite besitzt normalerweise eine Größe von 8 KB, sodass der Puffer ca. 2.000.000 Seiten umfasst. Auf einen Tenantspace bezogen wären das also 200.000 Seiten im Puffer.
Die Anzahl an Verbindungen zum Datenbanksystem sollte maximal in der Größenordnung der Anzahl der Mandanten sein. Es ist allerdings davon auszugehen, dass nicht alle Mandanten gleichzeitig aktiv sind beziehungsweise, dass durch die vorgeschalteten Applicationserver oder durch Nutzung des in [PgP11] vorgestellten `pgpool` die Anfragen gebündelt werden. Um die zur Verfügung stehenden Cores nicht zu überlasten, sollte die Anzahl der parallelen Verbindungen 100 nicht massiv überschreiten.
Der Workload besteht aus vielen kurzen Transaktionen mit geringem Schreibanteil. Als Anhaltspunkt seien **5.000 Transaktionen pro Sekunde** mit **10% Schreibanteil** gegeben. Die Transaktionen umfassen im Schnitt nur ca. 20 Operationen [SB11]. Daraus kann man schließen, dass sich pro Sekunde ungefähr 10.000 Änderungen durch Schreiboperationen ergeben. Diese Schreiboperation finden auf maximal ebenso vielen Seiten statt. Auf einen Tenantspace umgerechnet ergibt dies maximal **1.000 geänderte Seiten pro Sekunde**. Da PostgreSQL ein HOT-Tuple-Verfahren für Updates nutzt, sollte die tatsächliche Anzahl der geänderten Seiten deutlich geringer sein.
Wenn man darüber hinaus eine HIT-Ratio von ca. 90 % annimmt, werden bei 100.000 Operationen, die unter Annahme einer geringen Lokalität auf 50.000 Seiten ausgeführt werden, 5.000 Seiten pro Sekunde vom Datenbanksystem ausgeschrieben. Umgerechnet auf einen Tenantspace entspricht dies **500 Pufferplätzen**, deren Seiten durch einen MISS ausgeschrieben werden. Gleichzeitig wird für jede ausgeschriebene Seite auch wieder eine neue eingelesen. Diese auf das System gesehenen 5.000 I/O-Operationen mit einem Volumen von 5.000 Seiten * 8 KB/Seite = 40 MB müssen pro Sekunde auf die Festplatten übertragen werden. Bei SATA 3.0 Gbit/s Festplatten, die mit 500 MByte/s angebunden sind, ist das von der Bandbreite absolut kein Problem. Das Problem ist allerdings, dass diese Festplatten nur maximal 150 I/O-Operationen pro Sekunde (IOPS) durchführen können. Dies entspräche 150 Seiten mit einem gesamt Volumen von ca. 1 MB. Dies entspricht wiederum nicht den Anforderungen, die durch die MISSs im Puffer entstehen. Um dennoch die gewünschte Leistung zu erhalten, spielt hier bei Postgres der Betriebssystemcache eine wichtige Rolle. Da dieser nicht nur einzelne Seiten auf die Festplatte überträgt oder von ihr einliest, sondern dies gleich für größere Bereiche übernimmt, können die begrenzten I/O-Operationen besser ausgenutzt werden. Mit Hilfe dieses Caches lässt sich effektiv die angenommene HIT-Ratio erreichen.
Unter der Annahme, dass im Netzwerk Gigabit-Ethernet zur Verfügung steht, wäre die maximal möglichen Transferraten 125 MB/s. Nehmen wir weiter an, dass mit 50 MB/s ca. die Hälfte der Bandbreite des Netzwerks für eine Migration auf Datenbankebene zur Verfügung steht. Ein TCP/IP-Paket kann maximal 1500 Bytes umfassen, wobei pro Paket Header-Informationen von ca. 50 Byte anfallen. Versendet man Seiten mit einer Größe von 8 KB, umfassen diese 6 Pakete und man hat somit einen Overhead von 3%. Die minimale Paketgröße im Ethernet beträgt 84 Bytes, wovon 46 Bytes Nutzlast sind. Der Netzwerk-Overhead in diesem Fall ist ca. 100%.

Möchte man nun den Pufferinhalt eines Tenantspaces durch das Netzwerk kopieren, müssten ungefähr 2 GB an Seiten übertragen werden. Dies würde bei der angenommenen Transferrate von **50 MB/s** etwa 40 Sekunden dauern. In dieser Zeit würden durch die aktiven Transaktionen auf diesem Tenantspace weiterhin Änderungen durchgeführt. In 40 Sekunden werden laut den oben getroffenen Annahmen auf dem Tenantspace 60.000 Seiten auf dem Tenantspace geändert. 40.000 dieser Seiten wurden durch Änderungen aktualisiert und 20.000 neu in den Puffer eingelesen. Die Hälfte dieser Änderungen sollte im Mittel schon vor der Kopie in die Seite eingebracht worden sein, sodass nach der ursprünglichen Kopie noch 40.000 geänderte Seiten mit einem Volumen von ca. 320 MB übertragen werden müssen. Dies wird in etwa nochmal 8 Sekunden in Anspruch nehmen, wobei zu beachten ist, dass in dieser Zeit wiederum neue Änderungen anfallen. Die Dauer für den Transfer hat sich in etwa um den Faktor 5 verringert, sodass nach einer weiteren Iteration davon auszugehen ist, dass die Transferzeit für die Änderungen unter einer Sekunde liegen wird.
Es ist allerdings nicht nötig, Seiten komplett zu übertragen. Seiten enthalten einen Bereich, der momentan nicht genutzt wird und für zukünftige Tupel zur Verfügung steht. Durch Komprimierung dieses Platzes vor dem Versenden sinkt das Volumen der zu übermittelnden Daten. Da die Löschoperationen in der angenommenen OLTP-Last nur einen sehr geringen Anteil ausmachen werden, wird dadurch allerdings auch nur relativ wenig Platz in den Seiten wieder frei. Die Komprimierung des freien Platzes wird also vermutlich nur einen geringen Performancevorteil bringen.
Eine weitere Optimierung ist es, bei Seiten die sich nach der initialen Übertragung geändert haben, diese nicht komplett zu ersetzen sondern lediglich die Änderungen zu übertragen und wieder einzufügen. Diese Änderungen werden bereits im Rahmen eines WAL-Eintrags erzeugt. Diese WAL-Einträge können auch zu Synchronisationszwecken genutzt werden. Die Größe eines WAL-Eintrags hängt entscheidet von der Größe des geänderten Tupels ab. Die Größe des Tupel selbst ist natürlich abhängig von der entsprechenden Definition der Relation. Unter der Annahme, dass ein Tupel in der Größenordnung von < 100 Byte liegt, kann durch diesen Ansatz bei Änderungen ungefähr $1/50$ der Übertragungslast eingespart werden. Konkret am obigen Beispiel würde das bedeuten, dass bei 1.000 reinen Änderungen pro Sekunde in den 40 Sekunden der ursprünglichen Übertragung 40.000 Seiten aktualisiert werden müssen. Wiederum unter der Annahme, dass die Hälfte bereits übertragen worden ist, müssen also 20.000 WAL-Einträge dieser Änderungen übertragen werden. Bei 100 Byte pro Eintrag entspräche das einem Volumen von 2 MB. Dies könnte in sehr kurzer Zeit übertragen werden. Wichtig ist hierbei, dass das Versenden dieser kurzen WAL-Einträge nicht einzeln erfolgt, da diese sonst jeweils ein einzelnes Netzwerkpaket verbrauchen würden. Dabei entstünde sonst ein **Overhead von 50 Bytes pro Paket**, was in etwa 50% Overhead entspräche. Es müssten unter diesen Annahmen also insgesamt 6 MB an Daten zur Synchronisation der Änderungen übertragen werden. Allgemein hängt bei diesem Verfahren dessen Effizienz von der Größe der Tupel ab. Je größer die Tupel sind, desto geringer ist der relative Overhead durch das Netzwerkprotokoll, aber desto mehr Daten müssen insgesamt übertragen werden.
Darüber hinaus ist es allerdings weiterhin notwendig auch die in den 40 Sekunden neu in den Puffer geladenen Seiten zu übertragen. Nach den oben gemachten Annahmen werden pro Tenantspace und pro Sekunde maximal 500 Seiten ausgetauscht. Es müssen also trotzdem 20.000 Seiten mit einem Volumen von 160 MB übertragen werden. Diese würde in etwa wei-

tere 4 Sekunden in Anspruch nehmen. Insgesamt kann durch die WAL-Synchronisation die Migrationszeit halbiert werden. Eine Iteration verringert die Transferzeit nun um den Faktor 10. Da in den 4 Sekunden weiterhin Änderungen anfallen, ist eine weitere Iteration dennoch nötig. Werden hingegen weniger Seiten neu in den Puffer eingelesen als hier angenommen, dann wirkt sich diese Optimierung auch deutlich stärker auf die Migrationszeit aus.
Für die Performance sind somit die Anzahl aktiver Transaktionen, die Anzahl an Operationen pro Transaktion, die Änderungsrate, die HIT-Ratio, das Volumen des Puffers sowie die Netzwerkkapazität die entscheidenden Faktoren.

4.1.4. Vorstellung des Lösungsansatzes

Aus den in den vorherigen Abschnitten erläuterten Möglichkeiten wurde das in Abbildung 4.3 dargestellte Schema zur Live Migration entwickelt. Dieser Ansatz umfasst eine direkte Kopie des Puffers und eine anschließende Synchronisation mittels des Write-Ahead-Logs und wird kurz als `BufferCopyMigration` bezeichnet. Der Ablauf des Migrationsprozesses gliedert sich dabei in mehrere Phasen, welche gleich kurz vorgestellt werden. Diese Phasen bilden die Grundlage des Entwurfs, welcher zusammen mit den in den Phasen notwendigen Modulen im Rest dieses Kapitels näher erläutert wird.
Zunächst folgt nun eine Übersicht über die Entwurfsentscheidungen, die zu diesem Ansatz geführt haben. Zentral war zunächst die Feststellung, dass eine reine Pufferskalierung nicht die erwünschten Ergebnisse erbringen kann und deswegen ein Live Migration-Ansatz verfolgt wird. Bei der reinen Pufferskalierung werden lediglich einzelne Pufferseiten auf ein zweites System übertragen. Der dadurch theoretisch erreichbaren Vergrößerung des Gesamtpuffers steht entgegen, dass sich bei Seitenfehlern die Zugriffszeit verschlechtern würde. Bei einem Seitenfehler müsste die Seite zum einen von der Disk gelesen und zum anderen über das Netzwerk übertragen werden. Dieser Vorgang würde in einer dauerhaften Verdoppelung der Zugriffsdauer für diese Fälle münden. Das Ziel der Live Migration ist demgegenüber, dass nach Abschluss der Migration die Anfragen lokal und deshalb ohne Overhead erfolgen können.
Wie eingangs erwähnt gliedert sich die Migration in mehrere Phasen. Die erste Phase der Migration beinhaltet dabei den Aufbau der Verbindung zwischen Migrationsquelle und -ziel. Hinzu kommt Funktionalität zur Initialisierung der Migration.
Für die zweite Phase wurde aus den in 4.1.1 vorgestellten Ansätzen zur Migration der `Copy first, Syncronize later`-Ansatz ausgewählt. Dieser Ansatz bietet den Vorteil, dass die Quelle nur geringfügig belastet wird, während anschließend neue Transaktionen auf dem Zielsystem direkt auf einem warmen Cache operieren können. Ein warmer Cache ist für eine gute Performance der Datenbankanfragen wichtig. Damit ein warmer Cache auf dem Ziel entsteht, muss der Pufferinhalt des Quellsystems an das Ziel übertragen werden. Für den Betrieb mit warmen Cache ist es erstmal nicht notwendig, dass die komplette Datenbank übertragen wird. Dadurch dass man eine Kopie des Puffers der Quelle anfertigt, nutzt man automatisch die Vorteile, welche durch Pufferersetzungsstrategien entstanden sind. Obwohl man den parallelen Betrieb erst mit kurzer Verzögerung startet, bietet diese Maßnahme eine verhältnismäßig kurze Reaktionszeit.
An diese Phase schließt sich eine Phase des dualen Querybetriebs auf Quelle und Ziel an.

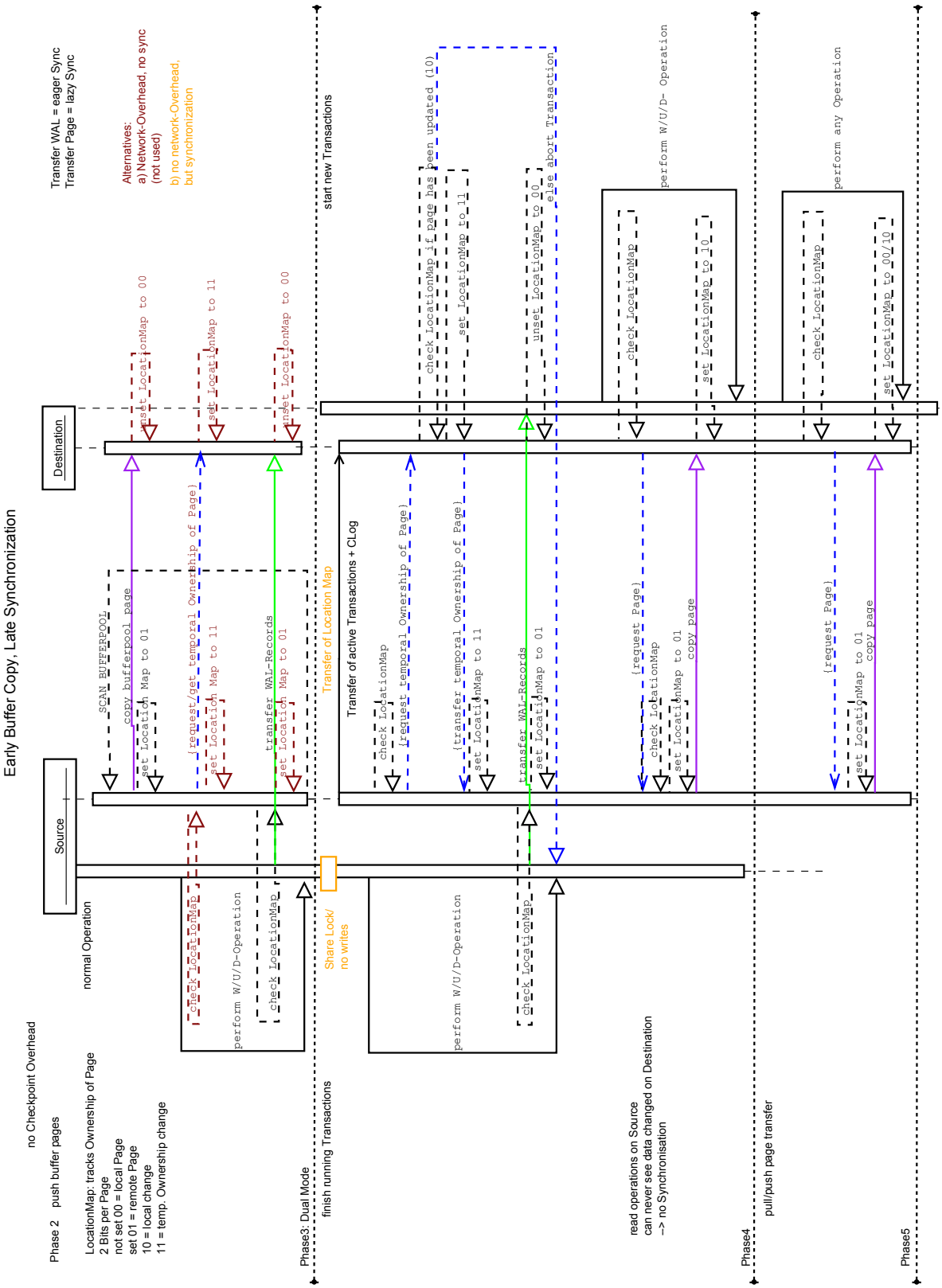

Abbildung 4.3.: Migration durch direkte Kopie des Puffers und Synchronisation mittels des Write-Ahead-Logs

Der duale Betrieb bedingt eine Synchronisation zwischen den Systemen, um die Konsistenz der Datenbank sicherzustellen. Zunächst einmal ist anzumerken, dass der Entwurf nur eine partielle Synchronisation vorsieht. Dieser partielle Synchronisationsansatz beinhaltet, dass Änderungen auf dem Ziel nicht auf die Quelle zurück übermittelt werden. Diese Entscheidung wurde aus den in 4.1.3 vorgestellten Abschätzungen abgeleitet mit dem Ziel die Netzwerkbelastung möglichst gering zu halten. Darüber hinaus wird damit auch verhindert, dass durch mehrmaliges hin und her übertragen einer Seite die gesamte Migrationsdauer verlängert wird. Eine längere Migrationsdauer hat durch den Migrationsoverhead einen negativen Einfluss auf die Leistung der Datenbank.
Für den Übergang in die nächste Phase und die damit beginnende Synchronisation sind noch Zwischenarbeiten nötig. Diese beinhalten die Übertragung der Zustandsinformationen, mit deren Hilfe die Ausgangssituation für die Synchronisation geschaffen wird. Diese ist nötig damit die Konsistenz der Datenbank erhalten bleibt. Für die Erstellung und Übertragung dieser Zustandsinformationen muss die Transaktionsbearbeitung auf der Quelle kurz angehalten werden. Da es sich aber nur um vergleichsweise sehr geringe Datenmengen handelt, ist davon auszugehen, dass diese Unterbrechung die Gesamtperformance der Datenbank nicht wesentlich beeinflusst.
Für diese Phase des dualen Betriebs standen die in 4.1.2 vorgestellten Ansätze zur Verfügung. Die Wahl für den Entwurf fiel auf einen gemischten Ansatz, der mehrere Konzepte miteinander verbindet. Der `lazy`-Ansatz der Migration wird insofern umgesetzt, dass das Ziel einzelne Seite anfordern muss und diese erst bei dieser Anforderung durch deren Übertragung synchronisiert werden. In dem Spezialfall, dass eine Seite, welche bereits auf dem Ziel existiert, von der Quelle geändert wird, tritt der `eager`-Ansatz zur Synchronisation in Kraft. In diesem Fall sendet die Quelle von sich aus die zur Synchronisation notwendigen Daten an das Ziel. Dieser gemischte Ansatz ermöglicht die Verwirklichung des Ownershipkonzeptes ohne zentralen Lockmanager.
Die zweite Entscheidung zur Synchronisation betrifft die Ebene, auf welcher diese stattfindet. Auch hier wurde ein hybrider Ansatz verfolgt. Grundsätzlich findet die Synchronisation auf Ebene der Seiten statt. Diese Entscheidung ist durch das Ownershipkonzept geprägt, welches dadurch die aufwändige Tupelebene umgehen kann. Um allerdings die Netzwerklast einzugrenzen, findet die Synchronisation von Änderungen auf der Quelle über einen differentiellen Ansatz statt. Dabei werden die auf der Quelle sowieso erstellten WAL-Einträge an das Ziel übertragen. Dort können die in den WAL-Einträgen beinhalteten Änderungen in die Datenbank eingebracht werden und sorgen somit für die Synchronisation. Die WAL-Einträge selbst beinhalten gerade die Differenz zwischen zwei Versionen einer Seite.
Mit dem Abschluss aller Transaktionen auf der Quelle beginnt die letzte Phase der Migration. Diese Phase zeichnet sich dadurch aus, dass die Quelle die restlichen persistenten Daten von sich aus an das Ziel überträgt. Gleichzeitig läuft die Transaktionsverarbeitung auf dem Ziel, wie in der vorherigen Phase beschrieben, weiter. Dieses als `Push` bezeichnete Verfahren auf der Quelle dient dazu, die restlichen Daten zu kopieren und somit die Migration abschließen zu können. Die Belastung der Quelle durch den `Push` der Daten wird dadurch aufgehoben, dass sie von der Bearbeitung neuer Transaktionen nun ausgeschlossen ist.
Anschließend finden lediglich noch Aufräumarbeiten statt, um den gesamten Migrationsvorgang als abgeschlossen betrachten zu können. Die Transaktionsbearbeitung für den

migrierten Datenbereich findet nun ausschließlich auf dem Zielsystem statt. Diese Bearbeitung ist dank der vollständigen Migration nun vollkommen unabhängig von der Quelle.

4.1.4.1. Alternative Ansätze

Zu dem ausgewählten Ansatz für die Migration gibt es noch Alternativen. Vier davon werden an dieser Stelle kurz vorgestellt und begründet, weshalb sie nicht ausgewählt wurden. Die Ersten drei Ansätze orientieren sich an dem letztendlich ausgewählten Ansatz. Sie besitzen einen ähnlichen Ablauf der Migration in Phasen. Sie unterscheiden sich im Wesentlichen durch die Art der Synchronisation. Der vierte und letzte Ansatz basiert auf einem anderen Modell. Diese Modell entspricht dem in 4.1.1 vorgestellten Mechanismus zur Migration, welcher zunächst die Kopie der Daten und eine anschließende Synchronisation vorsieht.

MigrationWALSynchronisation Dieser im Anhang in der Abbildung A.1 schematisch dargestellte Ansatz ähnelt dem letztendlich ausgewählten Ansatz in einigen Punkten. Der Unterschied liegt darin, dass das verteilte Ownershipkonzept hier über einen zentralen globalen Lockmanager erfolgen würde. Neben dem Aufwand für die Umsetzung dieses Lockmanagers spricht vor allem der durch das Anfordern der Sperren entstehende Overhead gegen diesen Ansatz. Jede Operation müsste nach diesem Modell eine Anfrage über das Netzwerk an diesen globalen Lockmanager stellen und auf eine Antwort warten um mit die Operation fortführen zu können. Dieser Overhead würde die Leistung des Systems während der Migration negativ beeinflussen.

MigrationOperationSynchronisation Dieser im Anhang unter A.2 illustrierte Ansatz hat seinen Schwerpunkt auf der Synchronisation auf Operationenebene. Dabei ist vorgesehen, dass sämtliche Änderungsoperationen auf das jeweilig andere System dupliziert werden. Dieser Ansatz beinhaltet ähnliche negative Elemente wie das vorherigen Modell. Der globale Lockmanager müsste ebenfalls aufwendig auf Tupelebene operieren. Außerdem stellt die Übertragung von Operationen eine aufwändige Methode zur Synchronisation dar, da zum einen viele Schnittstellen über das Netzwerk notwendig sind und zum anderen für jede Operation Netzwerkoverhead entsteht.

MigrationPageSynchronisation Dieser ebenfalls im Anhang unter A.3 skizzierte Ansatz führt die Synchronisation ausschließlich auf Seitenebene aus. Während dies die Kommunikation mit dem globalen Lockmanager erleichtert, hat diese Methode den Nachteil, dass das Netzwerk stärker belastet wird. Diese Belastung resultiert daher, dass zur Synchronisation grundsätzlich die gesamte Seite übertragen werden müsste. Dieser Aufwand würde die gesamte Migrationszeit verlängern und somit die Systemleistung negativ beeinflussen.

MigrationStreamingReplication Dieser im Anhang unter A.4 illustrierte Ansatz basiert auf dem in PostgreSQL 9.0 vorhanden Ansatz der `Streaming Replication`. Bei diesem Ansatz wird zuerst ein Checkpoint der Datenbank auf der Quelle erstellt. Dieser wird anschießend an das Ziel übertragen. Nun müssen die Änderungen, die in der Zwischenzeit anfielen, auf das Ziel übertragen werden. Wenn das Ziel wieder nahezu auf dem aktuellen Stand ist, werden keine neuen Transaktionen mehr akzeptiert. Erst nachdem alle Transaktionen auf der Quelle abgeschlossen sind, können neue Transaktionen auf dem Ziel gestartet werden. Dieser Ansatz hat zum Nachteil, dass die Quelle durch den zusätzlichen Checkpoint belastet wird. Dies würde in einer Situation geschehen, in der sie sowieso unter schwerer Last ist, da ansonsten eine Migration gar nicht notwendig wäre. Außerdem führt die Übergabe der Transaktionsausführung von der Quelle auf das Ziel zu einer Unterbrechung der Transaktionsbearbeitung, welche die Antwortzeit negativ beeinflusst.

4.2. Initialisierung der Migration

Der Entwurf zur Live Migration teilt sich, wie in 4.1.4 vorgestellt, in mehrere aufeinander aufbauende Phasen. In der ersten Phase werden die für die Migration notwendigen Datenstrukturen initialisiert. Des Weiteren werden die Kommunikationswege zwischen Quell- und Zielsystem aufgebaut. In diesem Abschnitt werden zunächst die Voraussetzungen für die Migration erläutert. Darauf folgen die Beschreibung der beteiligten Komponenten. Anschließend wird dann im nächsten Abschnitt die folgende Phase der Erweiterung des Bufferpools beschrieben.

4.2.1. Voraussetzungen

Zur Migration werden zwei Systeme benötigt, die über Netzwerkinfrastruktur verbunden sind und auf denen jeweils eine Postgres-Datenbankinstanz läuft. Die Quelle bildet das momentan aktive System, auf dem die aktuellen Daten vorliegen und welches momentan unter Last steht. Das Zielsystem der Migration muss zu Beginn freie Kapazitäten besitzen. Als Ausgangssituation ist gegeben, dass sich Quell- und Zielsystem in einem Clusterumfeld befinden. Dies bedeutet, dass beide Systeme die selben Metadaten über das globale Datenbankmodell besitzen. Folglich muss die Information der Systemkataloge identisch sein und clusterweit eindeutige Mechanismen zur Identifikation vorliegen. Dies impliziert, dass auch auf beiden Systemen die gleichen Nutzer und Nutzerrechte vorliegen, welche notwendig für einen Verbindungsaufbau zwischen den System sind.
Um Teilbereiche einer Datenbank migrieren zu können, müssen die Daten logisch unterteilbar sein. Dies kann durch Tenantspaces erfolgen, in denen die Daten mehrere Mandanten zusammengefasst sind. Dem Datenbankcluster vorgestellt ist ein globaler Koordinator, welche zentrale Aufgaben wie Queryrouting oder die Verwaltung der clusterweiten Metadaten übernimmt.

4.2.2. Erweiterung des Befehlssatzes

Damit die Migration gestartet werden kann, muss der Befehlssatz erweitert werden. Eine Datenbankinstanz kann nicht von sich aus die Migration starten, da ihr der Überblick über den Cluster fehlt. Die Migration wird also entweder durch den Datenbankadministrator oder von einem übergeordneten System zur Koordination gestartet.

4.2.2.1. Befehl zur Migration

Bevor die Migration durchgeführt werden darf, müssen zunächst die Rechte des Absenders des Befehls überprüft werden. Die `superuser()`-Funktion gibt dabei an, ob der aktuelle Benutzer die entsprechenden Rechte hat.
Darüber hinaus muss die SQL-Schnittstelle angepasst werden, damit der Migrationsbefehl akzeptiert werden kann. Der Migrationsbefehl hat dabei die Form:

```
MIGRATE TENANTSPACE <name> TO <system name>
```

Dabei muss <name> ein Tenantspace, der in der Systemtabelle `pg\_mttenantspaces` eingetragen ist, beinhalten. Mit Hilfe des <system name> wird über die Konfigurationsdatei die IP-Adresse des Zielsystems ermittelt. Mit diesen Informationen kann nun im nächsten Schritt begonnen werden eine Verbindung zwischen den Systemen aufzubauen.

4.2.3. Backends für Quelle und Ziel

Für die Kommunikation zwischen Quelle und Ziel werden in den jeweiligen Datenbankinstanzen Backends gestartet. Backends übernehmen die Interpretation und Ausführung von Befehlen, die der Datenbankinstanz von außen über ein Frontend zugesendet wurden. Für jede eingehende Verbindung vom Frontend zum Datenbankserver wird ein lokales Backend gestartet. Da für die Migration Kommunikation in beide Richtungen notwendig ist, übernehmen Quelle und Ziel jeweils Funktionen als Frontend und Backend.
Zur Kommunikation selbst wird der Austausch von Nachrichten genutzt. Um eine einzelne Funktionen des Nachrichtenprotokolls zu definieren, werden einzelne Zeichen am Anfang der Nachricht genutzt. Mit Hilfe dieses Zeichens kann der Typ der Nachricht bestimmt und ihre korrekte Verarbeitung durchgeführt werden.

4.2.4. Verbindungsaufbau

Nachdem auf der Quelle der Migrationsbefehl eingegangen ist, startet diese den Verbindungsaufbau. Dazu erstellt sie ein `StartupPacket`, welches an das Ziel gesendet wird. Das Zielsystem interpretiert dieses `StartupPacket` und startet dementsprechend ein lokales Backend. Das `StartupPacket` enthält dazu wichtige Informationen wie Name des Nutzers und der genutzten Datenbank. Mit dessen Hilfe können nun Nachrichten von der Quelle

empfangen und verarbeitet werden. Um jedoch auch Nachrichten vom Ziel an die Quelle zu schicken, muss ein Rückkanal eingerichtet werden. Folglich nutzt das Ziel den selben Mechanismus mit dem `StartupPacket`, um eine Backend auf der Quelle zu starten und somit eine Rückverbindung zu ermöglichen. Nun kann auch das Ziel eine Nachrichtenübertragung initiieren.

4.2.5. Unterbindung von Verwaltungsoperationen

Für die Dauer der Migration dürfen keine Verwaltungsoperationen ausgeführt werden. Zu Verwaltungsoperationen zählen unter anderem das Anlegen und Löschen von Datenbankobjekten, wie Relationen oder Mandanten. Darüber hinaus müssen auch Operationen, wie das Anlegen eines Checkpoints oder die Durchführung des `Vacuum`-Vorgangs, unterbunden werden. Diese Einschränkungen dienen dazu, die Migrationslogik zu entlasten und die Konsistenz der Datenbank nicht zu gefährden. Da ein Ziel ist den Migrationsvorgang möglichst schnell abzuschließen, sollten diese Einschränkungen eingehalten werden können ohne eine negative Auswirkungen auf die Gesamtleistung des Systems zu verursachen.
Verantwortlich für die Einhaltung der Einschränkungen kann sowohl der Administrator als auch das übergeordnete Koordinatorsystem sein. Eine Umsetzung durch direkte Sperren innerhalb der Datenbank wäre auch möglich.

4.3. Erweiterung des Bufferpools auf das Zielsystem

Nachdem die Grundlagen für die Migration in der ersten Phase erläutert wurden, beginnt nun die eigentliche Migrationslogik. In dieser Phase findet die Transaktionsverarbeitung noch regulär auf der Quelle statt. Zusätzlich werden im Rahmen des `EarlyBufferKopie`-Mechanismus bereits erste Daten auf das Zielsystem kopiert. Dieser Mechanismus führt zu einer Art Pufferskalierung, da Seiten aus dem Puffer der Quelle an den Puffer des Ziels übergeben werden.
Der Entwurf enthält die Beschreibung mehrerer Komponenten, welche in dieser Phase zum Einsatz kommen und teilweise auch in den nächsten Phasen weitergenutzt werden. Zentrale Komponente ist dabei die Umsetzung eines Ownershipkonzeptes für Datenbankseiten. Dieses Modul übernimmt Funktionalität eines Lockmanagers und ermöglicht so den späteren dualen Betrieb von Quelle und Ziel. Darauf folgt die Erstellung der Kopie des Pufferinhalts von der Quelle auf das Ziel. Diese dient dazu, dass das Ziel einen warmen Cache zur Verfügung hat. Zum Abschluss dieser Phase ist noch weitere Funktionalität nötig, die eine Übergabe des Transaktionsstandes an das Ziel ermöglicht. Diese Übergabe ist Voraussetzung für den Beginn der Synchronisation zwischen Quelle und Ziel unter Erhaltung der Korrektheit der Datenbank in der nächsten Phase.

4.3.1. Ownershipkonzept für Datenbankseiten

Ein Ziel des Migrationsansatzes ist es die Daten ohne Unterbrechung von Transaktionen zu übertragen. Das hat zur Folge, dass die Daten zwischen Quelle- und Ziel im laufenden Betrieb synchronisiert werden müssen. Für diese Synchronisation ist ein Ownershipkonzept für einzelne Seiten notwendig. Umgesetzt wird dieses dadurch, dass beide beteiligte Systeme jeweils eine lokale Datenstruktur besitzen. Diese gibt für jede Seite an, wer die Ownership über diese Seite hat. Besitzt ein System die Ownership einer Seite, so hat es das Recht zu unbeschränkten Lese- und Schreibzugriffen. Dabei muss diese Struktur für jede Relation des Tenantspace vorgehalten werden, also auch für die entsprechenden Indexrelationen. Für jede Seite der Relation werden in dieser Struktur zwei Bit genutzt um die Ownership zu repräsentieren.
Diese zwei Bits ermöglichen das Setzen der folgenden vier Modi für eine Seite.

- **lokale** Ownership bedeutet, dass das aktuelle System die volle Ownership über die Seite hat.
- **dirty** Ownership bedeutet, dass die Ownership auf dem aktuellen System ist und dass die Daten auf diesem System bereits geändert wurden.
- **remote** Ownership bedeutet, dass die Ownership für diese Seite auf dem komplementären System liegt.
- **temporäre** Ownership bedeutet, dass die Ownership grundsätzlich auf dem Zielsystem liegt, aber momentan an die Quelle zurück gegeben wurde.

Initial ist die Ownership auf der Quelle für alle Seiten lokal vorhanden, da zu Beginn der Migration alle Seiten nur lokal auf der Quelle vorliegen. Aus diesem Grund wird auf dem Ziel bei der Initialisierung die Ownership auf `remote` gesetzt, dass heißt das Ziel besitzt zu Beginn keine Datenbankseiten des Tenantspaces.
Führt im Laufe der Migration ein System eine Änderung auf einer Seite aus, für die es die lokale Ownership besitzt, so setzt das System diese anschließend auf `dirty`. Ein Spezialfall bildet die temporäre Ownership. Die Quelle erhält genau dann die temporäre Ownership über eine Seite, wenn diese zuvor bereits an das Ziel übertragen wurde und dort die Ownership noch nicht auf `dirty` gesetzt wurde. Nachdem die Quelle die Änderungen an der Seite unter temporärer Ownership durchgeführt hat, werden die Änderungen sofort an das Ziel übertragen. Die Quelle gibt nun durch Zurücksetzen der Ownership wieder diese an das Ziel ab. Das Ziel hat nun wieder die lokale Ownership. Dieser Mechanismus ist Teil des `eager`-Synchronisationsansatzes.
Die zur Verwaltung der Ownership verwendete Struktur wird `LocationMap` genannt. Ihre Funktionalität ist angelehnt an die der in PostgreSQL vorhandenen `VisibilityMap`. Es werden allerdings im Gegensatz dazu zwei Bits pro Seite genutzt, da vier Modi möglich sind. Die einzelnen Ownershipbits für die Seiten werden, auch wie in der `VisibilityMap` nach Relationen getrennt, in Datenbankseiten angelegt und verwaltet. Pro Seite der `LocationMap` kann demnach die Ownership über etwa 64 MB an Daten in den ursprünglichen Relationen organisiert werden.
Die Zugriffssynchronisation zwischen den beiden Systemen wird über diese Datenstruktur

abgewickelt. Dazu gibt es die oben beschriebenen Fälle. Die Ownership für eine Seite ist zum jeweils anderen System komplementär. Konkret bedeutet dies, dass wenn ein System für eine Seite lokale oder `dirty` Ownership besitzt muss das andere zwingend `remote` Ownership eingetragen haben. Ausnahme bildet die `temporal` Ownership, welche immer auf beiden System gleichzeitig gesetzt ist.

4.3.2. Early Buffer Copy

Zentrale Aufgabe dieser Phase ist die Durchführung der `Early Buffer Copy`. Deren Zweck ist es einen warmen Cache auf dem Zielsystem zu erreichen. Dafür soll der Pufferinhalt des entsprechenden Tenantspace auf der Quelle auf das Zielsystem kopiert werden. Da es keine gesonderten Informationen über die Pufferseiten eines Tenantspaces gibt, muss ein Scan über den Bufferpool durchgeführt werden. Der Migrationsprozess iteriert dazu über die Pufferplätze und überprüft die einzelnen Pufferseiten. Bei diesen Seiten kann es sich um Heap- oder Indexseiten handeln. Für jede Seite muss dann die Tenantspacezugehörigkeit und anschließend die Ownership überprüft werden.

Erkennen der Tenantspacezugehörigkeit Die Überprüfung auf Tenantspacezugehörigkeit ist wichtig um die Migration auf den gewollten Bereich einzugrenzen. Es wird ausgenutzt, dass die Relationen in ihrem Deskriptor einen Eintrag, der den `RelFileNode` angibt, besitzen. Der `RelFileNode` identifiziert die zugehörige physische Datei. Dieser `RelFileNode` enthält auch den `suffix`-Eintrag, mit dessen Hilfe der zugehörige Tenantspace identifiziert werden kann. Dieser `RelFileNode` ist auch in jedem Bufferheader als Teil des `tag`-Eintrag verlinkt. Dadurch kann auch von einer Pufferseite ausgehend der zugehörige Tenantspace eindeutig bestimmt werden.

Übeprüfen der Ownership Ist die Seite Teil des Tenantspaces, dann wird für diese Seite mittels der `LocationMap` überprüft, ob sie überhaupt noch im Besitz der Quelle ist. Ist dies nicht der Fall, dann wurde die Seite bereits übertragen und sie wird übersprungen.

Wurde nun eine Pufferseite gefunden, die auf das Ziel kopiert werden soll, wird diese wie später in 4.3.4.1 erläutert für Schreibzugriffe gesperrt und übertragen. Anschließend wird die Seite wie in Abschnitt 4.3.4.2 beschrieben in die Zieldatenbank eingebracht. Bei diesem Vorgang werden dann auch die jeweiligen `LocationMaps` des Quell- und Zielsystems aktualisiert.

4.3.3. Kommunikationsprozesse

Ein wichtiger Bestandteil der Migration ist die Kommunikation zwischen dem Quell- und Zielsystem. In 4.2.3 wurden schon die generelle Funktionalität eines Verbindungsaufbau über Backends zwischen den Datenbanksystemen erläutert. Damit die Kommunikation zwischen

den Systemen zentralisiert und gebündelt ablaufen kann, muss ein neuer Kommunikationsprozess eingeführt werden. Dieser als `MessageWorker` bezeichnete Prozess übernimmt auf jedem System die Kommunikation durch das Verschicken von Nachrichten. Dieser für das jeweilige System zuständige `MessageWorker`-Prozess sammelt die auf diesem System im Rahmen des Ownershipkonzeptes und durch die Synchronisation von Seiten und WAL-Einträgen entstehenden Nachrichten und übermittelt diese an das komplementäre System. Als zentraler Prozess muss der `MessageWorker` in der Lage sein mit den anderen Prozessen des Systems zu kommunizieren. Dazu können Signale und der Postgres interne `Shared Memory` genutzt werden.

4.3.3.1. Protokoll zur Übertragung der Seiten

Ein wichtiger Bestandteil der Migration ist die Übertragung von Seiten über die Kommunikationsverbindungen. Dazu wird eine Nachrichtenklasse definiert, die neben dem Seiteninhalt auch dessen Metadaten enthält. Dazu zählen unter anderem die RelationenID und die Blocknummer. Mit Hilfe dieser Daten kann die übertragene Seite dann korrekt in das Zielsystem eingefügt und weiter genutzt werden. Um Netzwerkbandbreite zu sparen, kann der Seiteninhalt komprimiert werden. Die Seiten sind so organisiert, dass sie einen zusammenhängenden leeren Teil unterschiedlicher Größe beinhalten. Dieser kann zur Übertragung ausgelassen und nachträglich wieder eingefügt werden.

4.3.3.2. Protokoll zur Übertragung und Einbringen der WAL-Einträge

Seiten, die bereits kopiert und anschließend wieder geändert wurden, müssen auf das Ziel synchronisiert werden. Für diese Aktualisierung werden die WAL-Einträge genutzt. Dafür muss die Übertragung über die existierenden Kommunikationsverbindungen zwischen Quelle und Ziel organisiert werden. Das Backend auf der Quelle, welches einen WAL-Eintrag zu einer bereits kopierten Seite des zu migrierenden Tenantspace erstellt, muss innerhalb der `XLogInsert`-Funktion dem `MessageWorker` eine Kopie des Eintrags zukommen lassen. Der `MessageWorker` sammelt mehrere Nachrichten und überträgt diese, in einem Paket gebündelt, an das Backend auf dem Zielsystem. Dieses Backend sorgt wiederum dafür, dass die einzelnen Nachrichten und somit die WAL-Einträge aus dem Paket ausgelesen werden. Anschließend müssen diese Einträge mit den entsprechenden Ressourcenmanager durch den Aufruf von `rm_redo` wieder eingespielt werden. Wichtig ist hierbei, dass während dem Einbringen der Änderungen auf dem Heap die LSN dieser Seiten auf `InvalidXLogRecPtr` gesetzt wird. Dies ist nötig, da im Falle der Recovery die von der Quelle stammenden WAL-Einträge nicht nochmal auf dem Zielsystem in dessen WAL geschrieben werden, um Duplikate im Gesamtsystem zu vermeiden. Danach übernimmt das Ziel durch Setzen der Bits in der `LocationMap` wieder die volle Ownership über die Seite, welche mittels `rm_redo` aktualisiert wurde.

4.3.4. Seitenzugriffe

Es folgt nun die Beschreibung des Ablaufs von Seitenzugriffen im Rahmen des Ownershipkonzeptes. Für die Durchführung von Seitenzugriffen als Teil der Migration ist es notwendig die in 4.3.3 beschriebenen Kommunikationsprozesse einzubinden. Diese übernehmen die Übermittlung von Nachrichten zur Aushandlung der Ownership oder zur Übertragung von Daten.

4.3.4.1. Auslesen einer Seite zur Übertragung

Auf der Quelle wird im Rahmen der `Early Buffer Copy` durch den Scanner Seiten ausgelesen und zur Übertragung bereit gestellt. Der Scanner ermittelt die zu kopierenden Seiten des Puffers und liest diese dann aus, um sie an das Ziel zu übertragen. In der anschließenden dritten Phase wird im Rahmen des dualen Betriebs diese Methode zur Übertragung von Seiten dann vom Ziel aufgerufen. Dies geschieht, wenn das Ziel einzelne Seiten anfordert, die noch nicht auf das Ziel kopiert wurden.
Für das Auslesen und zur Übertragung muss zunächst einmal das `BUFFER_LOCK_SHARE` für die jeweilige Seite erworben werden. Wenn die Sperre zugesichert ist, wird die `LocationMap` auf der Quelle angepasst. Dadurch gibt die Quelle die Ownership über diese Seite ab. Die Seite wird nun mittels dem in 4.3.3.1 vorgestellten Seitenübertragungsprotokoll auf das Ziel kopiert. Anschließend kann das `BUFFER_LOCK_SHARE` auf der Quelle wieder freigegeben werden. Die Quelle ist jetzt nicht mehr im Besitz der Seite und kann deswegen zunächst nicht mehr schreibend auf sie zugreifen.
Beim Einbringen der Seite im nächsten Schritt (siehe 4.3.4.2) aktualisiert das Ziel wiederum seine lokale `LocationMap`. Die Ownership der Seite ist nun komplett auf das Zielsystem übergegangen.
Bei einer vollständigen Synchronisation könnte diese Methode von beiden Systemen genutzt werden.

4.3.4.2. Einbringen der Seite in den Puffer des Ziels

Seiten werden in dem partiellen Synchronisationsverfahren ausschließlich von der Quelle kopiert und anschließend in den Puffer des Ziels eingebracht. Um eine Seite auf dem Ziel einbringen zu können, sind mehrere Schritte notwendig.
Zunächst muss `BufferAlloc(smgr,forkNum,blockNum...)` mit den Informationen über die übertragene Seite aufgerufen werden. Diese Methode stellt nun einen leeren Pufferplatz bereit, da die Seite auf dem Ziel noch nicht existiert bzw. leer ist. Der `BufferDesc` dieses Pufferplatzes bekommt den `tag` der übertragenen Seite, um sie eindeutig identifizieren zu können. Darüber hinaus wird die lokale `LocationMap` auf dem Ziel aktualisiert. Dadurch geht die Ownership über diese Seite an das Ziel über. Anschließend wird der Pufferinhalt mit der übertragenen Seite überschrieben. Um deutlich zu machen, dass diese Seite ursprünglich von einem anderen System stammt, wird die LSN im Header der Seite auf den

`InvalidXLogRecPtr` gesetzt. Dieser wird erst bei lokal ins Write-Ahead-Log geschriebenen Änderungen erneuert.

4.3.4.3. Seitenzugriff auf der Quelle

Es folgt nun die Beschreibung des Seitenzugriffs auf der Quelle. Die Seite wird dabei mittels `ReadBuffer` in den Puffer geladen. Die Quelle darf immer lesend auf Seiten zugreifen, da sie sich für Leseoperationen nicht mit dem Ziel synchronisieren muss. Das liegt darin begründet, dass dank des Multi-Version-Concurrency-Control-Ansatzes nur Änderungen von Transaktionen sichtbar sind, die vor dem eigenen Transaktionsbeginn gestartet wurden. Die Migrationslogik stellt sicher, dass alle Zieltransaktionen nach den Transaktionen auf der Quelle beginnen. Möchte die Quelle allerdings schreibend zugreifen, so muss die Ownership überprüft werden.
Wird auf die Pufferseite lediglich zum Lesen das `BUFFER_LOCK_SHARE` erworben, muss dafür nicht die Ownership überprüft werden. Wird allerdings ein `BUFFER_LOCK_EXCLUSIVE` zur Durchführung von Änderungen erworben, muss die Ownership überprüft werden. Die Ownership wird hier mittels der `LocationMap` ermittelt und, falls die Quelle die Ownership noch nicht besitzt, beantragt.
Wenn die Quelle die Ownership über diese Seite nicht besitzt, bedeutet dies, dass die zugehörige Seite bereits an das Ziel übertragen wurde. In diesem Fall muss die Quelle die Ownership temporär zurückfordern. Ist dies nicht der Fall, dann kann die Operation regulär ausgeführt werden. Während der `Early Buffer Copy`-Phase bekommt die Quelle die Ownership immer zurück übergeben. Dies liegt daran, dass auf dem Ziel noch keine Transaktionen ausgeführt werden.
Die Quelle setzt nun die temporäre Ownership über diese Seite in ihrer lokalen `LocationMap`. Sie erhält dadurch nur ein zeitlich begrenztes Recht diese Seite zu nutzen. Ändert die Quelle Daten auf der Seite, schickt sie diese Änderungen umgehend an das Ziel. Das Zielsystem pflegt diese Änderungen ein und übernimmt somit wieder die volle Ownership über diese Seite.
Die dazu genutzten Nachrichten werden im folgenden definiert. Eine Bündelung dieser Anfragen ist im Bezug auf den Netzwerkoverhead sinnvoll.

Protokoll zur Anforderung der temporären Ownership

Eine Anfrage wird an das Ziel geschickt. Diese Nachricht muss die Seite eindeutig identifizieren. Dazu wird die Blocknummer und die RelationenID der Seite übertragen. Außerdem muss der die Ownership anfordernde Prozess über die ProzessID eindeutig identifizierbar sein, damit diesem die Antwort zugeordnet werden kann.

Protokoll zur Übergabe der temporären Ownership

Das Ziel schickt eine Antwortnachricht zurück an die Quelle mit den Identifikationsdaten der Seite. Die Seite wird wiederum mittels Blocknummer und RelationenID identifiziert. Zusätzlich muss mitgeteilt werden, ob die Ownership erfolgreich übergeben werden konnte

oder nicht. Wurde die Ownership nicht übergeben, so muss die entsprechende Transaktion auf der Quelle abgebrochen werden.
Wurde die Ownership übergeben, wird die entsprechende Transaktion auf der Quelle benachrichtigt, sodass sie ihre Operation ausführen kann.

4.3.4.4. Änderungen auf bereits kopierten Seiten

Ein Spezialfall existiert für Seiten, die durch den Scan schon auf das Ziel kopiert wurden. Wenn diese Seiten in der Zwischenzeit auf der Quelle wieder geändert werden, bedarf es einer Synchronisation mit dem Ziel. Da eine Version dieser Seite bereits auf dem Ziel vorhanden ist, reicht es die Differenz zu übermitteln. Zur Synchronisation ist es also in diesem Fall nicht notwendig die ganze Seite zu übertragen.
Aus diesem Grund wird auch nicht die volle Ownership vom Ziel zur Quelle zurück übertragen, sondern nur eine temporäre Ownership. Wenn auf der Quelle nun eine Änderung auf einer Seite mit temporären Ownership durchgeführt wird, dann wird dafür automatisch ein WAL-Eintrag erstellt. Dieser WAL-Eintrag kann zur Synchronisation des Ziels genutzt werden. Also sendet die Quelle diesen WAL-Eintrag wie im Abschnitt 4.3.3.2 beschrieben an das Ziel. Das Ziel spielt nun den WAL-Eintrag wieder in die schon vorher kopierte, alte Version der Seite ein. Dadurch hat das Ziel die gleiche Version wie die Quelle und damit auch die aktuellste Version. Beim Wiedereinbringen der Änderungen auf dem Ziel ist darauf zu achten, dass die LSN der Seite wieder auf den `InvalidXLogRecPtr` gesetzt wird. Anschließend aktualisieren Quelle und Ziel ihre lokalen `LocationMaps`, wodurch das Ziel wieder die volle Ownership übernimmt.

4.3.5. Übergang in die nächste Phase

Nachdem die `Early Buffer Copy`-Logik ausgeführt wurde, kann die Übergabe für die nächste Phase beginnen. Da nun auch Transaktionen auf dem Ziel durchgeführt werden können und somit beide System synchronisiert werden müssen, gilt es während des Übergangs einen konsistenten Ausgangspunkt zu erstellen. Dazu werden zunächst auf der Quelle alle Transaktionen unterbrochen. Somit wird verhindert, dass die Daten, die an das Ziel übertragen werden sollen, während dieser Übergangsphase geändert werden. Um den Ausgangspunkt für den dualen Betrieb festzulegen, müssen von der Quelle der aktuelle Transaktionszustand, der aktuelle Ownershipstand der `LocationMap` und das Commit-Log auf das Ziel übertragen werden. Mit diesen Informationen kann dann die Transaktionsverarbeitung auf dem Ziel und die damit einhergehende partielle Synchronisation durchgeführt werden.

4.3.5.1. Protokoll zur Übergabe des Transaktionszustands

Die Quelle muss dem Ziel zum einen die aktuell höchste beendete TransaktionsID und zum anderen eine Liste aller momentan auf der Quelle aktiven Transaktionen übermitteln.

Dies ist für die Transaktionsausführung und die dabei notwendige Erstellung von Multi-Version-Concurrency-Control-Snapshots auf dem Ziel notwendig. Die aktiven Transaktionen des Tenantspaces können über die aktiven Backends ermittelt werden. Das Zielsystem speichert nach Erhalt die aktiven Transaktionen in dem Array `ActiveSourceXacts`, um diese beim Snapshot für neue Transaktionen zu nutzen. Dieses Array muss beim Abschluss einer Transaktion auf der Quelle, durch die Übermittlung des WAL-Eintrags aktualisiert werden. Dies wird erreicht, indem die betroffene Transaktion in dem Array der aktiven Transaktionen mittels einer Schleife gefunden und durch einen ungültigen Wert überschrieben wird. Das Array kann im `Shared Memory` vorgehalten werden, da es nicht mehr wachsen kann, da keine neuen Transaktionen auf der Quelle mehr gestartet werden.

4.3.5.2. Übermittlung der LocationMap

Um das Netzwerk während dieser Phase zu entlasten, wurde die Entscheidung getroffen die `LocationMap` auf dem Ziel nicht von Beginn an zu pflegen. Das Ziel erhält erst in diesem Schritt die nötigen Informationen gebündelt.
Damit nun das Zielsystem eine Übersicht über die Seiten hat, die aktuell in seinem Besitz sind, muss die `LocationMap` von der Quelle auf das Ziel kopiert werden. Die einzelnen Einträge der `LocationMap` müssen dabei umgewandelt werden. Dies liegt daran, dass Seiten, die auf der Quelle als lokale Ownership gekennzeichnet waren, jetzt auf dem Ziel `remote` sein müssen und umgekehrt.

4.3.5.3. Übertragung des Commit-Logs

Damit neue Transaktionen in der nächsten Phase die Multi-Version-Concurrency-Control-Daten korrekt auswerten können, müssen die Commit-Logs (CLog) an das Zielsystem übertragen werden. Diese Commit-Logs umfassen nur einige Seiten, sodass ihre Übertagung nur eine kurze Zeitspanne in Anspruch nimmt. Um diese CLog-Seiten übertragen und auf dem Ziel einfügen zu können, muss zunächst die entsprechende CLog-Datei auf dem Zielsystem angelegt werden. Anschließend werden die Seiten als Nachrichten an das Ziel verschickt.
Neue Commit/Abort-Einträge werden anschließend in der nächsten Phase mit Hilfe der WAL-Einträge auf das Zielsystem übertragen.

4.3.6. Recovery

Da die Quelle alle Operationen in ihrem regulären WAL mitgeloggt hat, reicht bei einem Ausfall der Quelle in dieser Phase die Ausführung der lokalen Recovery. Alle Seiten liegen auch durchgehend lokal auf der Quelle vor, da das reguläre Ausschreiben von Seiten auf der Quelle nach wie vor unterstützt wird. Beim Ausfall des Ziels hingegen kann einfach die Migration neu gestartet werden. Dieser Ansatz verringert den Recoveryaufwand im Vergleich zu einer kompletten Recovery des aktuellen Migrationszustandes.

4.4. Dualer Betrieb von Quelle und Ziel

Die Phase des dualen Betriebs zeichnet sich dadurch aus, dass nun auch Transaktionen auf dem Ziel zugelassen werden. Im Gegenzug dazu werden auf der Quelle keine neuen Transaktionen mehr akzeptiert. Die bereits gestarteten Transaktionen werden aber dennoch ausgeführt. Dabei ist der clusterweite Koordinator dafür zuständig, dass ab diesem Zeitpunkt neue Transaktionen für diesen Tenantspace ausschließlich auf das Zielsystem geleitet werden. Der duale Betrieb auf Quelle und Ziel bedingt eine Synchronisation zwischen den Systemen, da auf den gleichen Datenbereich zugegriffen werden können muss. Der Entwurf sieht hierbei eine partielle Synchronisation vor, mit dem Ziel die Migrationsdauer möglichst kurz zu halten. Dies hat zur Folge, dass im Falle eines Schreibkonflikts die Transaktion auf der Quelle abgebrochen werden muss. Es spricht allerdings nichts gegen einen durch den Koordinator bedingten Neustart dieser Transaktion auf dem Ziel. Für die partielle Synchronisation wird das in 4.3.1 vorgestellte Ownershipkonzept für Seiten angewandt. Darüber hinaus sorgt der in 4.3.2 vorgestellte Mechanismus für einen warmen Cache auf dem Ziel. In dem Puffer des Zieles befinden sich also schon die momentan am häufigsten genutzten Seiten, sodass die neuen Transaktionen teilweise lokal auf dem Ziel ausgeführt werden können.
Dieser Abschnitt befasst sich im Rahmen der partiellen Synchronisation zunächst mit dem Starten neuer Transaktionen auf dem Ziel. Danach folgt die Beschreibung von Synchronisationsmechanismen auf Grundlage des Ownershipkonzeptes.

4.4.1. Partielle Synchronisation

Die partielle Synchronisation umfasst die zentrale Logik zur parallelen Durchführung von Transatkionen auf dem Quell- und Zielsystem im Rahmen dieses Migrationsansatzes. Die partielle Synchronisation wird wie bereits erwähnt durch das Ownershipkonzept umgesetzt. Dieses sorgt für die Erhaltung der Konsistenz der Datenbank, obwohl Seiten auf der Quelle und dem Ziel teilweise unterschiedlichen Inhalt haben können. Diese unterschiedlichen Inhalte der Seiten treten aufgrund der `lazy` Synchronisation auf. Diese `lazy` Synchronisationsmethode sorgt dafür, dass Unterschiede erst bei Lese- oder Schreibanfragen auf diese Seiten synchronisiert werden. Dadurch wird einem durch eine `eager` Synchronisation entstehender Synchronisationsoverhead entgegen gewirkt.
Für die `lazy` Synchronisation spricht auch, dass die Quelle aufgrund der Multi-Version-Concurrency-Control sowieso niemals die auf dem Ziel durchgeführten Änderungen sehen kann. Es kann allerdings zu Schreibkonflikten kommen, wenn nach dem Ziel die Quelle auf die selbe Seite schreiben möchte. Diese Konflikte werden nicht über die Synchronisation, sondern durch den Abbruch der Quelltransaktion aufgelöst. Daher auch die Bezeichnung als partielle Synchronisation.
Die Synchronisation wird automatisch auf Heap- und Indexseiten angewandt. Deshalb werden mit diesem Ansatz auch die Zugriffsstrukturen synchronisiert, wodurch die korrekte und effiziente Querybearbeitung auf dem Ziel ermöglicht wird. Lediglich spezielle Indexmodelle, die nicht über Seiten und Write-Ahead-Logging gesichert sind, können nicht synchronisiert

werden. Diese müssen, falls notwendig, nach der Synchronisation auf dem Zielsystem neu aufgebaut werden.

4.4.1.1. Starten neuer Transaktionen auf dem Ziel

Die Vorbedingung für das Eintreten in die Synchronisation ist das Starten neuer Transaktionen auf dem Ziel. Dabei wird zunächst beim Aufruf von `GetCurrentTransactionId` eine neue TransaktionsID erzeugt. Dies geschieht mittels `ShmemVariableCache->nextXid`, welche eine global neue TransactionsID aus dem `ShmemVariableCache` im `Shared Memory` erwirbt. Um dies auf dem Ziel ohne Widerspruch zu TransaktionsIDs auf der Quelle durchzuführen, wurde in 4.3.5.1 der `ShmemVariableCache` des Ziels aktualisiert. Die TransaktionsID wird dann im Rahmen des `CurrentTransactionState` für die Durchführung von Operationen backendweise vorgehalten.
Außerdem müssen `Snapshots` erzeugt werden, die die Multi-Version-Concurrency-Control-Sichtbarkeiten regeln. In `GetSnapshotData` wird ein Snapshot angelegt. Dabei wird aus dem `ShmemVariableCache` unter anderem `xmax` mit der aktuell höchsten beendeten TransaktionsID gefüllt. Dann werden alle aktiven TransaktionsIDs gesammelt. Diese werden im `xip`-Array des Snapshots gespeichert. Um diese Daten korrekt zu ermitteln werden die in 4.3.5.1 übermittelten Daten genutzt.

4.4.1.2. Synchronisation von Commit und Abort

Für die korrekte Durchführung der Transaktionen ist für die Ermittlung der Multi-Version-Concurrency-Control-Sichtbarkeit auch die Synchronisation von auf der Quelle durchgeführten Commits und Aborts von Transaktionen notwendig. Dazu wird das selbe Verfahren wie in 4.3.3.2 auch für WAL-Einträge, die im Rahmen des Commmits oder Aborts einer Transaktion entstehen, eingesetzt. Diese WAL-Einträge werden auf der Quelle durch die Methoden `RecordTransactionCommit` bzw. `RecordTransactionAbort` erstellt, wenn eine Transaktion mit `CommitTransaction` oder `AbortTransaction` beendet wird. Durch das Übertragen und Einbringen mittels `xact_redo` auf dem Ziel wird die Synchronisation des Commit-Logs automatisch mit ausgeführt. Wichtig ist hierbei, dass auf der Quelle nur die WAL-Einträge der Transaktionen des Tenantspaces übertragen werden. Der jeweilige Tenantspace kann über den aktuellen Mandanten des Backends identifiziert werden.
Außerdem müssen diese Transaktionen auch aus dem in 4.3.5.1 eingeführtem Array `ActiveSourceXacts` der aktiven Transaktionen gelöscht werden. Durch diese Aktualisierung werden die Ergebnisse einer Transaktion nach deren Commit auf dem Ziel sichtbar. Dies liegt daran, dass alle neue Transaktionen auf dem Ziel aus Sicht der Serialisierbarkeit nach dieser eben auf der Quelle beendeten Transaktion statt finden. Folglich muss diese Aktualisierung des `ActiveSourceXacts`-Arrays direkt nach der Aktualisierung des CLogs durchgeführt werden.

4.4.1.3. Seitenzugriff auf dem Ziel

Durch das Durchführen von Transaktionen auf dem Ziel erfolgen nun dort auch Zugriffe auf Datenbankseiten. Diese Zugriffe haben zur Folge, dass nun einzelne Seiten explizit von der Quelle angefordert werden.
Wenn das Ziel eine Seite öffnen will, dann muss es sicher sein die aktuellste Version der Seite zu besitzen. Bei der Aufforderung eine Seite mittels `ReadBuffer` in den Puffer zu hohlen, muss also mit Hilfe der `LocationMap` überprüft werden, ob die Seite bereits im Besitz des Ziels ist. Ist dies der Fall wird die weitere Nutzung der Seite regulär auf dem Ziel ausgeführt. Ist die Seite noch nicht im Besitz des Ziels, muss die Seite von der Quelle angefordert werden. Das Ziel sendet dazu eine Anfrage über diese Seite an die Quelle. Es kann nun zwei verschieden Fälle geben.
Im regulären Fall überprüft die Quelle ihre lokale `LocationMap` und stellt fest, dass sie die Ownership über diese Seite hat. Also führt die Quelle die in 4.3.4.1 beschriebenen Operationen durch und kopiert so die Seite mittels des in 4.3.3.1 beschrieben Protokolls auf das Ziel. Die Quelle erwirbt dabei ein `BUFFER_LOCK_SHARE` auf die Seite, um zur Übertragung parallele Schreibzugriffe auf diese Seite zu verhindern und gibt die Ownership mittels der Anpassung der `LocationMap` an das Ziel ab.
Bei der Überprüfung der Ownership auf der Quelle kann auch festgestellt werden, dass die Quelle nur die temporäre Ownership hat (siehe 4.3.1). In diesem Fall wird, nach Beendigung der Änderungen auf der Quelle und dem Aktualisieren der Seite auf dem Ziel, die Ownership wieder automatisch an das Ziel übergeben. Die Transaktion auf dem Ziel muss benachrichtigt werden, solange auf dieses Ereignis zu warten. Der Zugriff auf dem Ziel wird anschließend nach einem erneuten Check der `LocationMap` gestattet.
Wenn das Ziel eine Änderung auf einer Seite durchführt, dann trägt es das durch Setzen der Bits auf den `dirty`-Zustand in die `LocationMap` ein. Dadurch ist sichergestellt, dass die Ownership der Seite nicht mehr an die Quelle zurückgehen kann. Eine dementsprechende Anfrage wird nun vom Ziel abgelehnt werden.

4.4.1.4. Erzeugen neuer Seiten

Zusätzlich zu den bereits in der `Early Buffer Copy`-Phase auftretenden Fällen des Lesens und Übertragens von Seiten (siehe 4.3.3.1) muss nun auch das Erzeugen von neuen Seiten für eine Relation synchronisiert werden. Dabei bekommt die Methode `ReadBuffer_common` mitgeteilt, dass eine neue Seite angefordert wird. Dieser neuen Seite wird ein neuer Block am Ende der Relation zugewiesen. Durch die Initialisierung der `LocationMap` ist die Ownership von jeder neuen Seite zunächst automatisch auf der Quelle.
Auf dem Ziel muss daher vor dem Anlegen eines neuen physischen Blocks mittels der Ownership überprüft werden, ob dieser Block nicht schon auf der Quelle angelegt wurde. Dazu wird für den zu erstellenden Block gemäß 4.4.1.3 eine ganz normale Anfrage gestellt, die Seite von der Quelle zu übertragen. Die Ownership für diese Seite wird bei durch die Antwort auf diese Anfrage, wie bei dem normalen Ablauf, mit an das Ziel übergeben.
Wird bei dieser Anfrage eine leere Seite übertragen, bedeutet dies, dass die Seite vor der Anfrage noch nicht auf der Quelle existiert hat. Die Übertragung einer leeren Seite verbraucht

nur sehr wenig Kapazität. In diesem Fall kann das Ziel ganz normal fortfahren und lokal die Relation erweitern. Wird eine nicht leere Seite zurückgegeben, muss das Ziel diese Seite einfügen. Anschließend kann das Ziel diese Seite nutzen. Mittels des Resource Managers kann über die Methode `mdnblocks` die momentane Anzahl an Seiten einer Relation ermittelt werden.

4.4.1.5. Logging

Ein wichtiger Aspekt für die Konsistenz und Dauerhaftigkeit der Datenbank ist das Logging. In PostgreSQL wird dazu ein Write-Ahead-Logging-Mechnismus genutzt. Dieser wird wie folgt eingebunden.
Nachdem eine Seite im Puffer geändert und somit als `dirty` markiert wurde, muss ein WAL-Eintrag erzeugt und im Rahmen des Write-Ahead-Logs auf die Festplatte in das Logfile geschrieben werden. Dies muss vor dem Abschluss der Transaktion oder dem Ausschreiben der Pufferseite selbst geschehen. Der Verweis auf diesen WAL-Eintrag enthält den Relationenknoten sowie die TupleID. Der Inhalt des Tupels wird dann der Funktion `XLogInsert` übergeben. Für das Replay des Logs wird die Log-Sequence-Number (LSN) genutzt. Diese stellt die korrekte Reihenfolge beim Wiedereinbringen von Daten aus dem Log sicher.
Das Logging wird nur lokal auf dem jeweiligen System ausgeführt. Für die Quelle bedeutet dies, dass das Logging unabhängig von der Migration ausgeführt werden kann. Da die Quelle auf Seiten, die auf dem Ziel geändert wurden, nicht mehr zugreifen kann, gibt es für diesen Fall auch keine WAL-Einträge. Daraus folgt, dass die Logs auf der Quelle auch direkt für die Recovery genutzt werden können.
Auf dem Ziel muss für korrektes Logging und Recovery beachtet werden, dass die von der Quelle übertragenen Seiten im Rahmen ihrer LSN eine andere Nummerierung der Logeinträge haben. Deswegen wird die LSN von übertragenen Seiten direkt mit einem ungültigen Wert überschrieben. Grundsätzlich werden aber die Operationen auf dem Ziel auch hier auf normalen Wege in das lokale Write-Ahead-Log übertragen.
Die WAL-Einträge jedoch, die wie in 4.3.3.2 beschrieben das Ziel zur Synchronisation von der Quelle übermittelt bekommt, soll das Ziel allerdings nicht für die Recovery nutzen und darf sie deshalb auch nicht in das eigene Write-Ahead-Log schreiben. Ziel ist es, dass keine Überschneidung in den WALs von Quelle und Ziel entsteht. Dies würde die Recovery unnötig verkomplizieren.
Das Ziel loggt seine Änderungen zunächst im `full_page_write`-Verfahren, um unvollständig ausgeschriebene Seiten wiederherstellen zu können. Abgeleitet aus dem Ownershipkonzept und der partiellen Synchronisation lässt sich für das Logging folgende Regel aufstellen: Das System welches die Ownership über die Seite hat und eine Änderungsoperation durchführt, loggt auch diese. Die temporäre Ownership ist lediglich die Fortsetzung dieses Ansatzes. Seiten die einmal auf dem Ziel geändert und somit geloggt wurden, werden anschließend nicht mehr auf der Quelle geloggt. Dadurch entsteht eine eindeutige zeitliche Abfolge der Logeinträge.

4.4.2. Recovery

Wie eben in 4.4.1.5 beschrieben werden WAL-Einträge in dieser Phase in Abhängigkeit der Ownership einer Seite auf der Quelle oder auf dem Ziel geloggt.

Ausfall der Quelle Fällt nun die Quelle aus, dann werden alle Transaktionen auf der Quelle automatisch beendet. Die Quelle hat lediglich WAL-Einträge von Seiten, die vor einer möglichen Änderung auf dem Ziel verfasst wurden, und kann deshalb lokal recovern. Das Ziel muss nun warten bis die Quelle lokal recovert hat und kann anschließend wieder Seiten anfordern.

Ausfall des Ziels Fällt jedoch das Zielsystem aus, werden wiederum alle laufenden Transaktionen auf dem Ziel zwangsläufig abgebrochen. Laufenden Transaktionen auf der Quelle, welche Schreiboperationen auf dem Tenantspace ausführen wollen, werden ebenfalls abgebrochen, da die Ownership nicht mehr vom Ziel zurück übertragen werden kann. Ansonsten käme es zu Widersprüchen in Seiten, die auch auf dem Ziel geändert wurden. Anschließend recovert das Ziel alle Einträge, die vom Ziel verfasst wurden, mit Hilfe des lokalen Write-Ahead-Logs. Da das Ziel die Information über die Ownership verloren hat, muss sie dabei ihre lokale `LocationMap` wieder mit entsprechend gesetzten Bits neu initialisieren und danach neu befüllen. Das Ziel erkennt durch das Auffinden eines entsprechenden WAL-Eintrags, ob eine Seite überhaupt an das Ziel übertragen wurde und deshalb die Ownership angepasst werden muss. In diesem Fall wird dann folglich die `LocationMap` aktualisiert.
Entscheidend bei der Recovery ist die Ausführung von `XLogReadBufferExtended`, bei der die Ausgangsseite für das Replay eingelesen wird. Es wird also überprüft, ob eine physische Kopie der Seite auf der Disk des Ziels liegt. Wenn nicht, wird diese Seite von der Quelle angefordert und die `LocationMap` wird aktualisiert. Im Falle, dass eine gültige Version der Seite bereits auf Disk gefunden wurde, wird natürlich ebenfalls die `LocationMap` aktualisiert. Nun findet eine Überprüfung, ob ein WAL-Eintrag bereits auf die physische Seite angewandt wurde, statt. Da Seiten ausgeschrieben werden, kann die Version auf Disk neuer sein als der WAL-Eintrag. Diese Überprüfung ist für alle Seiten korrekt, da selbst von der Quelle übertragene Seiten, die ungeändert auf Disk ausgeschrieben wurden, korrekt eingeordnet werden. Dies liegt daran, dass die von der Quelle übertragenen Seiten künstlich eine ungültige LSN zugewiesen bekommen haben. Diese ist so gewählt, dass sie immer kleiner als die LSN des zu wiederherstellenden WAL-Eintrags ist. Der entsprechende WAL-Eintrag wird deshalb auch für einmalig übertragene Seiten korrekt zurückgespielt.

4.4.3. Übergang in die nächste Phase

Der Phasenübergang in die vierte Phase wird auf der Quelle eingeleitet. Sie stellt fest, dass alle Transaktionen ihrer Backends auf dem Tenantspace beendet wurden. Dies ist nach relativ kurzer Zeit der Fall, da in dieser Phase keine neuen Transaktionen auf der Quelle mehr angenommen wurden.

4.5. Abschluss der Migration durch Übertragung der persistenten Daten

In dieser Phase werden nun alle Transaktionen ausschließlich auf dem Ziel ausgeführt. Damit die Migration abgeschlossen werden kann, müssen allerdings noch die restlichen Daten von der Quelle an das Ziel übertragen werden. Ist auch diese Übertragung abgeschlossen, so können Quelle und Ziel unabhängig voneinander weiterarbeiten.

4.5.1. Transaktionsbearbeitung auf dem Ziel

Grundsätzlich laufen in dieser Phase die selben Schritte wie in den vorherigen Phasen ab. Der Unterschied besteht darin, dass nun nicht mehr synchronisiert werden muss, da keine Transaktionen mehr auf der Quelle bearbeitet werden. Bei Anforderung einer Seite durch das Ziel, muss diese wie in 4.4.1.3 überprüfen, ob sich die Ownership und somit die Seite generell schon in ihrem Besitz befindet. Wenn dies nicht der Fall ist, dann muss die Seite von der Quelle wie in 4.3.3.1 beschrieben angefordert werden.

4.5.2. Push von Seiten

In dieser Phase wird nun zusätzlich der sogenannte `Push` von Seiten durchgeführt. Dieser Mechanismus dient dazu die persistenten Daten von der Quelle an das Ziel zu kopieren. Um diesen `Push` auszuführen, muss die Quelle mit Hilfe der Ownership alle Seiten des Tenantspace überprüfen. Dabei übermittelt die Quelle die Seiten, die noch lokal in ihrem Besitz sind, an das Ziel. Das Ziel reagiert darauf wie in der zweiten Phase. In 4.3.3.1 wurde die Verarbeitung von eintreffenden Seitennachrichten beschrieben. Mit Abschluss dieses `Push`-Verfahrens ist auch die Migration abgeschlossen, da sich nun der komplette Datensatz auf dem Ziel befindet.

4.5.3. Recovery

Die Recovery läuft nach dem selben Muster, wie in 4.4.2 für die Phase des dualen Betriebs beschrieben, ab.

4.5.4. Übergang in nächste Phase

Sobald die Quelle alle Seiten an das Ziel übergeben hat, ist diese Phase beendet. Die Quelle stellt dies mit Hilfe der in der `LocationMap` gespeicherten Ownershipinformationen fest.

4.6. Aufräumen der temporären Daten

Die Migration an sich ist bereits mit der letzten Phase beendet worden. In dieser Phase müssen nun noch die temporären Daten entfernt und die für die Migration notwendigen Prozesse beendet werden. Anschließend befindet sich die Datenbank wieder in der Ausgangslage und eine neue Migration könnte beginnen.
Neben der Freigabe der Sperren und temporären Daten durch das Ende des Migrationskontextes können nun auch die `LocationMap`-Dateien aus den Systemen entfernt werden. Darüber hinaus besteht die Möglichkeit die Daten des Tenantspaces auf der Quelle komplett zu löschen. Dies könnte mittels des Befehls `DROP TENANTSPACE <name>` erfolgen, wodurch die entsprechenden Dateien und der Tenantspace gelöscht werden. Dies könnte dazu dienen auf der Quelle Festplattenspeicher freizugeben. Unabhängig davon wird der aktive Betrieb der Quelle mit Abschluss der Migration nicht mehr durch den migrierten Tenantspace belastet.

5. Implementierung

In diesem Kapitel wird die prototypische Realisation des Entwurfs zur Migration eines Tenantspaces vorgestellt. Der Prototyp basiert auf den Ergebnissen der in [Sch11] vorgestellten Arbeit. Als zu Grunde liegende Datenbank dient PostgreSQL in der Version 8.4.
Die folgenden Abschnitte orientieren sich an dem Entwurf und erläutern die konkrete Realisation des Entwurfs. Dazu zählen insbesondere die Umsetzung der einzelnen Phasen der Migration und der dafür notwendigen Funktionen und Datenstrukturen. Die Implementierung von im Entwurf vorgestellten Recoverymechanismen im Fehlerfall ist nicht Teil des Prototyps. Ebenso fehlt der dem Datenbankcluster vorgeschaltete Koordinator, welcher das Queryrouting und die Verwaltung der clusterweiten Metadaten sicherstellt.
Die Schwerpunkte der Implementierung des Prototyps liegen in der Umsetzung der Funktionalität des Entwurfs unter Berücksichtigung der Erhaltung und Sicherstellung der Konsistenz der Datenbank. Der Prototyp bietet noch Potential zur Optimierung der Leistung. Erste Optimierungsansätze, wie die Kompression von Seiten zur Übertragung, wurden aber bereits umgesetzt.

5.1. Initialisierung der Migration

Die Migrationslogik teilt sich in mehrere aufeinander aufbauende Phasen. In der ersten Phase werden die für die Migration notwendigen Datenstrukturen initialisiert und die Kommunikationswege zwischen Quell- und Zielsystem aufgebaut. In diesem Abschnitt werden zunächst die Voraussetzungen für die Migration erläutert. Nach einer Übersicht über die an der Migration beteiligten Prozesse folgt die Beschreibung der Implementierung der für diese Phase notwendigen Komponenten.

5.1.1. Voraussetzung für die Migration

Bevor die Migration eines Tenantspaces stattfinden kann, muss die Umgebung bestimmte Voraussetzungen erfüllen. Das Schaffen dieser Voraussetzungen für die Migration ist nicht Teil des Prototyps. Da sie allerdings für dessen korrekte Ausführung notwendig sind, werden sie an dieser Stelle kurz vorgestellt.
Zur Migration eines Tenantspaces wird von einem Postgrescluster-Umfeld ausgegangen. Eine sich daraus ableitende Voraussetzung für die Migration ist, dass Objektidentifkatoren (Oid) clusterweit einzigartig sind. Es müssen also die Metadaten aus den Systemkatalogen der Postgresinstanzen auf dem Quell- sowie auf dem Zielsystem die gleichen clusterweit

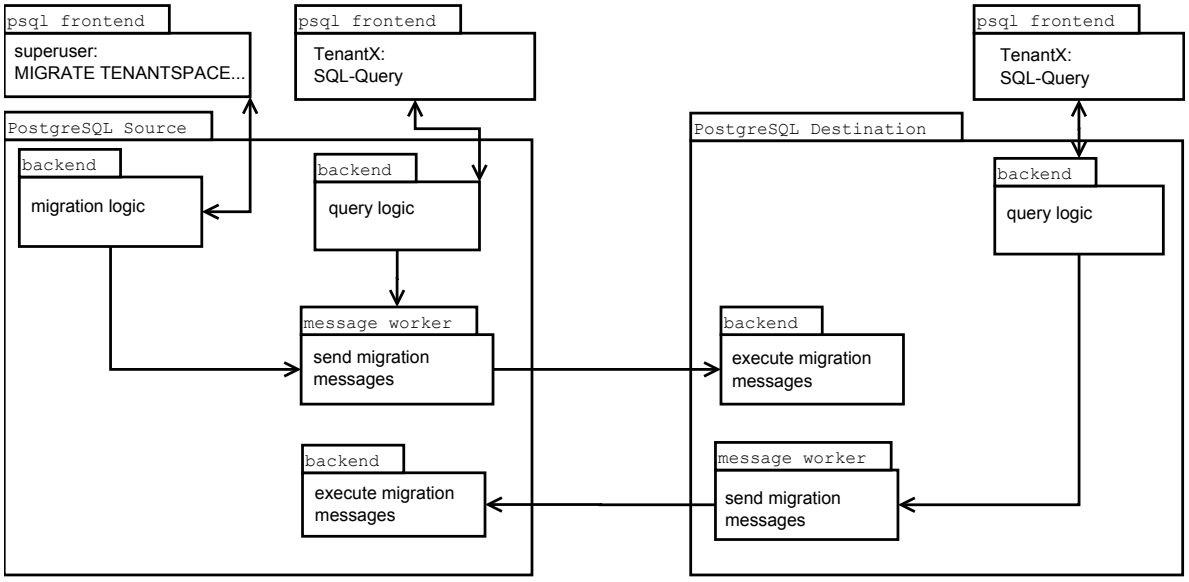

Abbildung 5.1.: Übersicht der an der Migration beteiligten Prozesse

eindeutig vergebenen Identifikationsnummern und daraus folgend auch die gleichen Werte für die zugehörigen Attribute haben. Dies gilt insbesondere für die Systeminformationen zur Datenbank, deren Nutzer und Mandanten und den zugehörigen Tenantspaces. Beispielsweise muss ein Mandant, der auf der Quelle existiert, auch auf dem Ziel mit gleicher ID und gleichen Namen eingetragen sein. Das gleiche gilt natürlich auch für Relationen und deren logischen Schemata.
Eine weitere Voraussetzung ist, dass ein gesonderter Nutzer mit `superuser`-Rechten im System eingetragen ist. Dieser Nutzer wird für die interne Kommunikation zwischen Quell- und Zielsystem eingesetzt.
Zum Aufbau einer Verbindung zwischen Quell- und Zielsystem wird vom Prototyp nur die Authentifikation über Einträge in der Konfigurationsdatei `hba_config` unterstützt. Diese müssen vor dem Start der jeweiligen Postgresinstanz auf dem Quell- oder Zielknoten korrekt gesetzt sein, um eine Verbindung zwischen den Systemen zu ermöglichen.

5.1.2. Übersicht über die beteiligten Prozesse

Die Abbildung 5.1 stellt eine Übersicht über die an der Migration beteiligten Prozesse und Kommunikationswege dar. Neben der Erweiterung der SQL-Schnittstelle zwischen `Frontend` und `backend` zählen insbesondere der für die Kommunikation verantwortliche `MessageWorker` und die Umsetzung der Migrationslogik zu den für den Prototyp implementierten Komponenten. Auf diese wird nun in den nächsten Abschnitten näher eingegangen.

5.1.3. Erweiterung des Befehlsatzes

PostgreSQL bietet mit dem Tool psql eine Benutzerschnittstelle zur Datenbank an. Über diese wird die Kommunikation von Befehlen und Antworten mit dem Datenbanknutzer durchgeführt. Die Befehlsschnittstelle dieses Tools wird so erweitert, dass die Migration mittels eines Befehls eingeleitet werden kann. Durch das Aufbauen der Verbindung des psql-Frontends mit der Postgresdatenbank wird in dieser automatisch ein neuen Backend erzeugt, welches die Kommunikation und die Verarbeitung der Befehle dieses Frontends übernimmt.

5.1.3.1. Befehl zur Migration

Der neu implementierte Befehl hat eine SQL-ähnliche Syntax und lautet wie folgt:

```
MIGRATE TENANTSPACE <name> TO <system>
```

Der erste Parameter <name> muss dabei ein bereits existierender Tenantspace sein, der automatisch gegen die Systemtabelle `pg_mttenantspaces` geprüft wird.
Der zweite Parameter <system> spezifiziert das Ziel System der Migration. Er ist ein zusammengesetzter Identifikationsstring, dessen einzelne Werte durch „+“ getrennt angegeben werden. In diesem String werden zunächst der Nutzer und die Datenbank sowie der Name des Zielsystems und der zugehörige Port angegeben. Der Nutzer und die Datenbank müssen auf dem Zielsystem bereits angelegt sein, da über sie die Verbindung aufgebaut wird. Der Befehl kann somit für den Nutzer `migrationrole` und die Datenbank `production` wie folgt aussehen:

```
MIGRATE TENANTSPACE ts1 TO "migrationrole+production+test.com+8888"
```

Der Name des Zielsystems wird intern für den Aufbau der Verbindung in die entsprechende IP-Adresse aufgelöst.

5.1.3.2. Anpassung des Parsers

Um den Migrationsbefehl in PostgreSQL einzufügen, ist die Anpassung des Parsers notwendig. Der Parser wird einerseits genutzt, um die Eingabe am psql-Frontend zu überprüfen und andererseits vom Datenbankserver selbst. Das Frontend erzeugt aus dem Befehl einen Querystring, welchen es an das Backend übermittelt. Das Backend parsed wiederum diesen String, um einen Befehlsbaum zu erzeugen und diesen auszuführen. Durch Einfügen der Syntax des neuen Befehls wird dieser beim Parsen korrekt erkannt und die Funktion `MigrateTenantspace` wird mit den entsprechenden Parametern aus dem Befehl aufgerufen. Diese Funktion bildet den Einstiegspunkt in die Migrationslogik.

5.1.3.3. Überprüfen der Rechte

Die Rechte zur Migration eines Tenantspace sollen nur Administratoren oder ggf. ein übergeordnetes Koordinationssystem besitzen. Diese sollen die Migration von Tenantspace zur Skalierung und Lastverteilung nutzen können. Daher wird zunächst überprüft, ob der Nutzer, der den Migrationsbefehl abgesetzt hat, auch die entsprechenden `superuser`-Rechte hat. Es ist also nicht vorgesehen, dass Kunden und deren Applikation diese Rechte der im Cloud Computing-Umfeld laufenden Datenbank bekommen und nutzen.

5.1.4. Interprozesskommunikation mittels Shared Memory

Da Postgres keine Threads einsetzt, sondern eigene Prozesse für jedes Backend und für einige interne Funktionen einsetzt, ist eine gesonderte Möglichkeit zur Interprozesskommunikation notwendig. Neben dem Einsatz von Signalen dient dazu vor allem der `Shared Memory`.

5.1.4.1. Shared Memory

Bei der Initialisierung von Postgres wird im Hauptspeicher der gesonderte `Shared Memory`-Bereich angelegt. Objekte im `Shared Memory` besitzen im Allgemeinen eine nach der Initialisierung feste Größe, da ein beim Nachallokieren notwendiges Verschieben dieser Objekte nicht möglich ist. Dies liegt daran, dass Prozesse, die nachträglich vom Hauptprozess geforkt werden, direkt Zeiger in den `Shared Memory` übernehmen und darüber auf diese Objekte zugreifen können.
Der `Shared Memory` wird bei der Migration zunächst einmal dazu genutzt den Migrationszustand global zugänglich zu machen. Dazu wird die `global_migration_state` Struktur angelegt. Sie gibt die aktuelle Phase der Migration, den zu migrierenden Tenantspace und ob

Ausschnitt 5.1 Struktur für den Migrationszustand

```
typedef struct global_migration_state_t
{
   bool          isDestination;
   bool          isSource;
   bool          finished;               /* used to signal MessageWorker */
   TenantspaceId migratingTenantspaceId; /* only changed at the beginning */
   MigrationPhaseData    migrationPhase; /* protected by MTTransitionLock */
} global_migration_state_t;
```

es sich um das Quell- oder Zielsystem handelt, an. Zur Interprozesssynchronisation dieser Struktur wurde das `MTTransitionLock` eingeführt, welches durch Postgres Lockmanagerkomponente verwaltet wird. Mittels dieser Sperre wird verhindert, dass ein Lesezugriff insbesondere auf das `migrationPhase`-Attribut einen falschen Wert zurück liefert. Das Erwerben

der Sperre und das Auslesen der Migrationsphase wurde in der `getGlobalMigrationState`-Methode realisiert.
Ein weiteres `Shared Memory`-Objekt bilden die in `global_migration_connection` vorgehaltenen Daten zur Verbindung zwischen Quell- und Zielsystem. In dieser Struktur sind auch die Nachrichtenpuffer, die zum Versenden und Empfangen von Nachrichten genutzt werden, enthalten. Nach der Initialisierung dieser Datenstruktur durch den Migrationsprozess werden diese Daten vor allem durch den im nächsten Abschnitt vorgestellten `MessageWorker` zur Kommunikation genutzt.

5.1.5. Aufbau der Verbindung zwischen Quell- und Zielsystem

Für die Kommunikation zwischen Quell- und Zielsystem wird eine neue zentrale Komponente eingeführt. Diese `MessageWorker` genannte Komponente ist ein eigener Prozess, der bei Beginn der Migration von dem Postmaster-Hauptprozess geforkt wird. Dieser Prozess nutzt die Informationen über Quell- und Zielsystem, die der Migrationsprozess in die `global_migration_connection`-Struktur im `Shared Memory` eingetragen hat. Mit diesen Informationen kann der `MessageWorker` nun zunächst eine Verbindung zum Zielsystem aufbauen. Dazu wird die zur Verbindung des psql-Frontends zum Postgresbackend genutzte Methodik wiederverwendet und angepasst.
Grundsätzlich wird über das Netzwerk eine TCP/IP-Verbindung aufgebaut. Genutzt werden kann je nach Bedarf das Internet Protokoll (IP) der Version 4 oder 6. Die Nachrichten werden in Paketen zwischen den beiden Systemen übertragen. Da eine Nachricht in mehrere Pakete aufgespalten werden kann, wird dies anschließend durch einen Nachrichtenpuffer wieder ausgeglichen.
Steht die Verbindung, so ist der `MessageWorker` für die Kommunikation verantwortlich. Dazu gibt es im `Shared Memory` als Teil der `global_migration_connection`-Struktur Nachrichtenpuffer, auf die alle an der Migration beteiligten Prozesse zugreifen können. Für ein- und ausgehende Nachrichten gibt es jeweils einen eigenen Puffer mit fester Kapazität. Damit es nicht zu Überschneidungen beim Zugriff zu diesen Puffern kommt, sind diese mittels eines `LightWeightLocks` geschützt. Nachrichten, die versendet werden sollen, werden also nach dem Erwerb der Sperre von dem jeweiligen Prozess in den Nachrichtenpuffer geschrieben. Der `MessageWorker` überprüft regelmäßig, ob im Nachrichtenpuffer neue Nachrichten vorhanden sind und verschickt diese dann anschließend über die Verbindung. Auf diese Weise ist die asynchrone Kommunikation vom Quell- zum Zielsystem mit nur einer Verbindung möglich.
Synchrone Kommunikation ist darauf beschränkt, dass auf eine Anfrage genau eine Antwortnachricht erwartet wird. Dazu muss der `MessageWorker` nach dem Versenden des Pufferinhalts wissen, wie viele Antworten er insgesamt zu erwarten hat. Dies müssen die einzelnen Prozesse beim Eintragen einer Nachricht in den Nachrichtenpuffer festhalten. Nach dem Senden der Nachricht wartet der `MessageWorker` darauf, die Antwort zu Empfangen und schreibt diese in den entsprechenden Puffer im `Shared Memory`. Des weiteren zählt der `MessageWorker` mit, welche Nachrichten er versendet hat, sodass andere Prozesse überprüfen können, ob ihre Nachricht bereits versendet wurde. Dadurch können diese Prozesse in kritischen Phasen sicher gehen, dass das komplementäre System die Nachricht

empfangen hat und somit auf dem gleichen Stand ist.
Die Verbindung zwischen dem Quell- und Zielsystem der Migration ähnelt der Verbindung mit dem psql-Frontend insofern, dass nachdem auf unterster Ebene eine Socketverbindung zwischen Quell- und Zielsystem aufgebaut wurde, auf dem Ziel ein neues Backend für diese Verbindung gestartet wird. Dies geschieht dadurch, dass das Quellsystem ein `StartupPacket` an das Ziel verschickt. Dieses `StartupPacket` enthält den Nutzernamen und die Datenbank, für die auf dem Ziel das Backend gestartet werden soll. Nachdem dieses `StartupPacket` erfolgreich verarbeitet wurde, steht der Migration nun ein Backend mit vollem Funktionsumfang und Zugriff auf das Zieldatenbanksystem zur Verfügung. Das Ziel teilt der Quelle durch eine Nachricht die erfolgreiche Verarbeitung des `StartupPacket` mit. Nun können von der Quelle an das Ziel Nachrichten versandt und Antworten empfangen werden.
Auf dem Zielsystem wird wiederum nach dem selben Schema ein eigener `MessageWorker`-Prozess gestartet. Die Rückverbindung wird ebenfalls mittels des `StartupPacket` aufgebaut. Dadurch ist nun auch Kommunikation, die vom Ziel ausgeht und sich an die Quelle richtet, möglich.

5.1.5.1. Nachrichtenformat

Das Nachrichtenformat für Mitteilungen zwischen dem `MessageWorker`-Prozess und dem Backend auf dem komplementären System fügt sich in das in Postgres bestehende Nachrichtenprotokoll ein. Eine Nachricht setzt sich somit aus drei Teilen zusammen. Die ersten zwei Byte spezifizieren die Nachrichtenklasse. Für Nachrichten, die sich auf die Migration beziehen, wird für das erste Byte die Klasse `'M'` eingeführt. Das zweite Byte spezifiziert eine Subklasse der Migrationsnachrichten. Anschließend wird die Länge der Nachricht angegeben, worauf der eigentliche Inhalt folgt. Empfängt das Backend eine Nachricht, ermittelt es zuerst mittels des `'M'` die korrekte Nachrichtenklasse und kann anschließend die Subklasse und den Nachrichteninhalt ermitteln und zur Weiterverarbeitung de-serialisieren.

5.1.6. Interne Verwaltung von Metadaten über Relationen

Zur Durchführung der Migration eines Tenantspaces ist es notwendig, die entsprechenden Relationen zu identifizieren. Dazu werden die Metadaten einer Relation genutzt. Diese werden intern in so genannten `Relation Descriptors` vorgehalten. Zur Identifikation von Daten einer Relation dient neben der clusterweit eindeutigen ObjektID auch der zugehörige `RelFileNode`. Der `RelFileNode` stellt dabei die Verbindung von der logischen Relation zu ihrer physischen Repräsentation in einer Datei auf dem Externspeicher dar.

Erkennen der Tenantspacezugehörigkeit Durch die Einführung von Tenantspaces und nach Tenantspace getrennten Tabellen in der vorherigen Arbeit [Sch11] entstand die Besonderheit, dass es zu jeder Relation mehrere `RelFileNodes` und somit auch mehrere physische Dateien geben kann. Die `RelFileNode`-Datenstruktur wurde dazu um ein `suffix`-Attribut erweitert. Für die Umsetzung des Prototyps wird davon ausgegangen, dass

der Wert des `suffix`-Attributs immer gleich der ID des zugehörigen Tenantspaces ist. Mittels einer Mappingtabelle könnten auch andere Fälle behandelt werden. Dieses `suffix`-Attribut dient dazu, die physische Datei, die zu dieser Relation und diesem Tenantspace gehört eindeutig zu identifizieren. Dies geschieht dadurch, dass der Name dieser Datei um diesen `suffix` erweitert wird.
Die Zuordnung von `RelFileNodes` und Relationen kann mit Hilfe des `pg_class`-Systemkatalogs erfolgen. Implementiert wurde dies als Teil der `getRelIdsfromTenantspaceId`-Methode, welche zu einem Tenantspace die Objekt-IDs aller zugehörigen Relationen und die der entsprechenden `RelFileNodes` zurück gibt.
Während die ObjektID der Relation auf Quell- und Zielsystem der Migration gleich sein muss, gilt dies nicht zwangsläufig auch für die ObjektID des zugehörigen `RelFileNodes`. Daher muss beim Versand von Daten darauf geachtet werden, dass die ObjektID der Relation mit gesendet wird. Mit deren Hilfe kann das andere System den zugehörigen `RelFileNode` ermitteln. Somit ist eine eindeutige Identifikation von Daten auf beiden Systemen gewährleistet.

5.1.7. Start der Migration

Nach dem die Verbindung zwischen Quell- und Zielsystem aufgebaut wurde, wird mittels des in 5.1.5 vorgestellten Nachrichtenprotokolls der Start der Migration kommuniziert. Dazu wird dem Ziel die entsprechende Tenantspace ID des zu migrierenden Tenantspace übermittelt. Dafür wurde in das in 5.1.5.1 vorgestellte Nachrichtenprotokoll die Subklasse `'S'` eingeführt.
Anschließend setzen Quelle und Ziel den globalen Zustand im in 5.1.4 beschriebenen `Shared Memory`. Dadurch ist der Tenantspace und ob es sich um das Ziel- oder Quellsystem der Migration handelt, in der Postgresinstanz global bekannt. Zusätzlich wird dort in die Statusvariable die Phase `earlyBufferCopy` eingetragen, womit die nächste Phase und damit die eigentlich Migration beginnt.

5.2. Erweiterung des Bufferpools

In dieser Phase läuft auf der Quelle der normale Transaktionsbetrieb weiter. Im Rahmen der Migration wird auf der Quelle eine `LocationMap` aufgebaut und gewartet. Diese Umsetzung des im Entwurf vorgestellten Ownershipkonzepts wird im nächsten Abschnitt erläutert.
Darauf folgt die Implementierung der Übertragung einzelner Seiten des zu migrierenden Tenantspace an das Ziel. Dieser Vorgang wird als `Early Buffer Copy` bezeichnet und findet parallel zum normalen Transaktionsbetrieb statt. Einmal übertragene Seiten werden bei Änderungen anschließend auch auf dem Ziel aktualisiert. Zur Durchführung von Leistungsvergleichen kann die Übertragung von Seiten während dieser Phase übersprungen werden. Dazu muss eine Compiler-Direktive bei der Kompilierung des Prototyps gesetzt werden. Sinn des Überspringens der `Early Buffer Copy`-Phase ist es, die Auswirkungen eines warmen Caches auf dem Ziel zu ermitteln.

Zum Abschluss werden die zur Synchronisation notwendigen Komponenten vorgestellt. Deren Ausführung ist Teil des Übergangs in die nächste Phase.

5.2.1. Umsetzung des Ownershipkonzepts

Das Ownershipkonzept sorgt für die nötige Synchronisierung von Quell- und Zielsystem. Die Ownership wird dabei für einzelne Seiten einer Relation jeweils auf dem Quell- und Zielsystem mitprotokolliert. Dazu wird die Ownership in der neu eingeführten `LocationMap` verwaltet. Die dafür notwendige Datenstruktur wird in einer mittels Seiten organisierten Datei angelegt. Jeder physische `RelFileNode` erhält eine eigene Datei, welche die `LocationMap` beinhaltet. Der `LocationMap`-Dateinamen wird aus dem Namen des zugehörigen `RelFileNodes` und der Endung `_mtlm` gebildet. Intern wird diese Datei in der Datenstruktur `RelNode` durch das Einfügen eines eindeutig identifizierbaren Eintrags repräsentiert. Dieser Eintrag lautet `MTLOCATIONMAP_FORKNUM` und dient zur Identifikation der `LocationMap`-Daten einer Relation.
Die interne Funktionalität der `LocationMap` ist an die bereits in Postgres existierende `VisibilityMap` angelehnt. Beide Module speichern für jede Seite einer Relation kurze Informationen über deren Zustand. Im Gegensatz zur `VisibilityMap` werden für die `LocationMap` allerdings zwei Bits pro Seite benötigt. In diesen zwei Bits wird der aktuelle Modus der zugehörigen Seite und somit die Ownershipinformation gespeichert. Die im Entwurf beschriebenen Modi werden durch diese vier möglichen Zustände umgesetzt:

- `LOCAL_CLEAN_OWNERSHIP`
- `LOCAL_DIRTY_OWNERSHIP`
- `REMOTE_OWNERSHIP`
- `TEMPORAL_OWNERSHIP`

Jeder dieser Zustände entspricht einer physischen Repräsentation von zwei Bits für diese Seite.
Da die `LocationMap` über Seiten organisiert ist, erfolgt die Zugriffssynchronisation über die üblichen Sperrmechanismen für Pufferseiten. Dadurch wird automatisch ein Zugriff auf die `LocationMap` durch alle Prozesse ermöglicht.

5.2.1.1. Operationen auf der LocationMap

Die Operationen auf der `LocationMap` teilen sich in zwei Schritte. Zunächst wird mittels `locationmap_pin` die der Seite der Relation entsprechende Seite der `LocationMap` mit einem Pin für den Zugriff festgehalten und mit `locationmap_unpin` dementsprechend wieder freigegeben. Befindet sich diese Seite noch nicht im Puffer, so wird diese Seite dadurch vom Externspeicher eingelesen. Existiert diese Seite überhaupt noch nicht, dann wird die `LocationMap` automatisch auf diese Seite erweitert. Dabei wird die Seite entsprechend

initialisiert.
Im zweiten Schritt kann dann mittels `locationmap_request` die Ownership über die Seite angefordert werden. Mittels `locationmap_test` wird dabei zunächst der aktuelle Modus für die Seite der Relation abgefragt. Besitzt das System bereits die Ownership der Seite, dann kann die Verarbeitung normal fortgeführt werden. Wenn dies nicht der Fall ist, dann muss die Ownership zunächst vom anderen System beantragt werden. Anschließend wird mittels `locationmap_set` der entsprechende neue Modus gesetzt.
Zur Durchführung der Operationen wird jeweils die Blocknummer der Seite der Relation auf die entsprechende `LocationMap`-Seite und die genaue Position innerhalb der Seite umgerechnet. Dadurch kann auf die der Blocknummer entsprechenden zwei Bits zugegriffen werden.

5.2.1.2. Besonderheiten auf dem Ziel

Während auf der Quelle die `LocationMap` automatisch mit `'00'`, dass heißt mit der `LOCAL_CLEAN_OWNERSHIP` initialisiert wird, darf dies auf dem Zielsystem der Migration nicht der Fall sein. Neue Seiten auf dem Ziel müssen mit `REMOTE_OWNERSHIP` initialisiert werden, da zunächst einmal die Ownership für alle Seiten auf der Quelle liegt. Erst bei der Übertragung von Seiten wird dann auf dem Ziel die Ownership angepasst und dadurch entsprechend auf `LOCAL_CLEAN_OWNERSHIP` gesetzt.
Bekommt jedoch das Zielsystem im Rahmen des Phasenwechsels eine Locationmapseite von der Quelle, dann muss sie diese mittels `locationmap_convert` zunächst einmal umwandeln. Dies ist notwendig, da das Quellsystem die Ownershipverhältnisse aus seiner lokalen Sicht aufgezeichnet hat. Wenn also das Quellsystem bei sich für eine Seite lokal die Ownership eingetragen hat, muss dementsprechend auf den Ziel für diese Seite eine `remote` Ownership eingetragen werden.

5.2.1.3. Einbinden der LocationMap

Die Einbindung der `LocationMap` bildet den zentralen Mechanismus zur Synchronisation zwischen Quell- und Zielsystem während der Migration. Dafür muss die `LocationMap` einerseits bei Änderungsoperationen und andererseits bei Leseoperationen abgefragt werden. Für diese Operationen muss zunächst jeweils überprüft werden, ob das System die Ownership über die entsprechende Seite besitzt oder, wenn nicht, diese anfordern.
Bei Änderungsoperationen muss dann im zweiten Schritt die `LocationMap` entsprechend angepasst werden, sodass der Zustand für die Seite auf `LOCAL_DIRTY_OWNERSHIP` beziehungsweise auf `TEMPORAL_OWNERSHIP` gesetzt wird. Dieser Zustand wird direkt nach dem Anfragen der Ownership gesetzt, sodass Race Conditions vermieden werden. Außerdem halten diese Methoden eine Sperre auf die entsprechende Heap- bzw. Indexseite, sodass eine Seite nicht von zwei Prozessen gleichzeitig geändert werden kann.
Für Heapseiten geschieht die Einbindung der Ownershipüberprüfung in den Methoden `heap_insert`, `heap_update` und `heap_delete` sowie für Indexseiten in den Methoden `_bt_getbuf`, `_bt_insertonpg`, `_bt_newroot` und `bt_split`. Dies entspricht den Operationen,

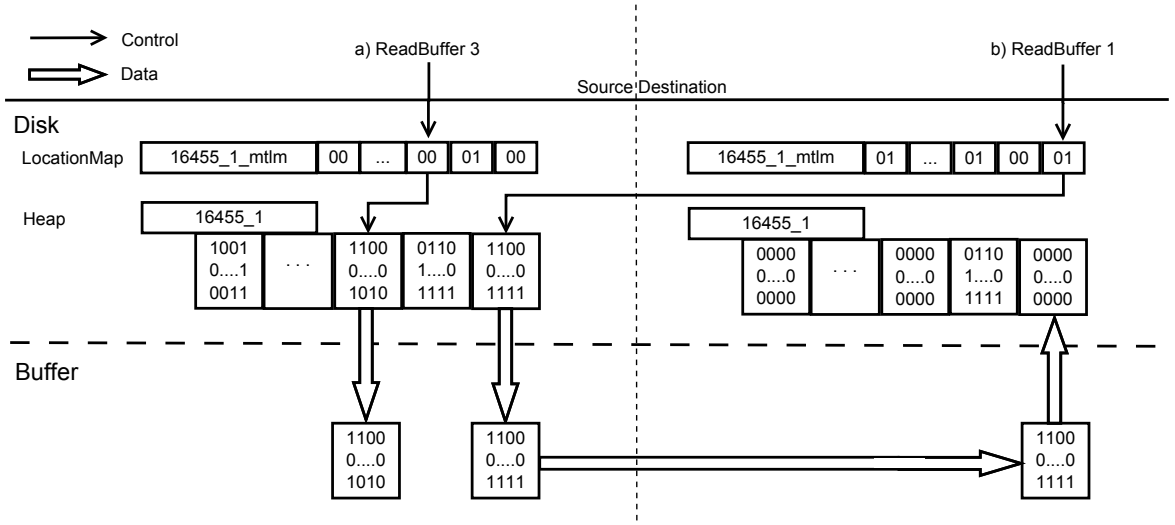

Abbildung 5.2.: Einbinden der LocationMap

welche die Ursprungsseiten physisch ändern.
In Abbildung 5.2 ist die Einbindung der `LocationMap` in eine Abfrage schematisch dargestellt. Bei Abfrage a) signalisiert `'00'` in der `LocationMap`, dass die Ownership lokal vorliegt. Deshalb kann die zugehörige Seite direkt im Bufferpool zur Verfügung gestellt werden.
In Fall b) liefert die Abfrage der `LocationMap` `'01'` zurück - die Ownership über diese Seite ist also auf dem anderen System. Folglich muss an dieses System eine Anfrage gestellt werden. Daraufhin wird die entsprechende Seite übertragen, sodass auf sie zugegriffen werden kann. Das anschließende Anpassen der Ownership in der `LocationMap` für diese Seite ist in der Abbildung nicht dargestellt.

5.2.1.4. Besonderheiten auf der Quelle

Aufgrund der nur partiellen Synchronisation zwischen Quelle und Ziel ergeben sich zwei Besonderheiten. Zum einen kann das Quellsystem nie mehr die volle Ownership von einmal an das Ziel übertragenen Seiten zurückbekommen. Zum anderen braucht die Quelle für Leseoperationen nicht die Ownership anfragen. Sie hat auf vom Zielsystem geänderte Daten aufgrund der Multi-Version-Concurrency-Control keinen Zugriff, da alle Transaktionen auf dem Ziel nach den Transaktionen auf der Quelle gestartet werden. Für Schreiboperationen auf bereits übermittelten Seiten muss die Quelle eine temporäre Ownership beim Ziel beantragen. Nach der Operation aktualisiert die Quelle automatisch mithilfe des WAL-Eintrags die Seite auf dem Ziel. In der `Early Buffer Copy`-Phase wird der Quelle die temporäre Ownership einer Seite immer automatisch zugestanden, während in der Phase des dualen Betriebs dies nur möglich ist, wenn das Ziel die Daten auf dieser Seite noch nicht geändert hat.

5.2.2. Early Buffer Copy

Die eigentliche Migration beginnt mit der Durchführung der `Early Buffer Copy`. Dabei werden die Seiten aus dem Puffer auf dem Quellsystem ausgelesen und an das Zielsystem übertragen. Dadurch ist gewährleistet, dass wenn in der nächsten Phase Anfragen auf dem Ziel zugelassen werden, diese mit einem warmen Puffer starten und deswegen erstmal keine Performanceverluste auftreten.
Um die `Early Buffer Copy` durchzuführen werden auf der Quelle die durchnummerierten Pufferplätze durchgegangen und überprüft, ob sie an das Ziel versendet werden sollen. Diese Überprüfung testet im ersten Schritt, ob die Seite des Puffers überhaupt zum migrierenden Tenantspace gehört. Dazu muss zunächst die Relation, zu der die Pufferseite gehört, festgestellt werden. Dazu werden die Informationen über den `RelNode` aus dem `Buffer Descriptor` genutzt. Dieser Wert ist identisch mit dem `RelFileNode`-Wert aus dem Systemkatalog `pg_class`. Aus `pg_class` lässt sich letztendlich rückschließen, um welche Relation es sich handelt und ob diese zu einem Tenantspace gehört. Relationen, die zu Tenantspaces gehören, haben im Attribut `relstoragetype` die Werte 't' und für das Attribut `relmtkind` die Werte 'b' oder 'c' und können anhand dessen identifiziert werden.
Steht also nun fest, dass der gewählte Puffer zu einer Tenantspace getrennten Relation gehört, dann muss noch überprüft werden, ob es sich um den korrekten Tenantspace handelt. Dies wird wiederum mittels des `Buffer Descriptors` erreicht, da das Attribut `suffix` wie in 5.1.6 beschrieben Rückschluss auf den Tenantspace ziehen lässt.
Trifft beides auf die gewählte Seite im Puffer zu, wird nun die Ownership mittels der `LocationMap` ermittelt. Die Ownership über diese Seite muss noch lokal auf der Quelle vorliegen, dass heißt der Modus muss entweder `LOCAL_CLEAN_OWNERSHIP` oder `LOCAL_DIRTY_OWNERSHIP` sein.
Wenn auch dies korrekt ist, hat die Seite alle Voraussetzungen im Rahmen der `Early Buffer Copy` an das Ziel übertragen zu werden.

5.2.2.1. Serialisierung einer Pufferseite

Um die Seite zum Ziel übertragen zu können, muss zunächst der `Buffer Descriptor` serialisiert werden. Dieser dient dazu, die Seite anschließend auf dem Ziel wieder korrekt einfügen zu können. Dazu werden neben der clusterweit eindeutigen RelationenID auch der Buffertag des `Buffer Descriptors` in einen String serialisiert. Dabei wird jedem Wert eine feste Zahl an Bytes im String zugewiesen, sodass die Nachricht anschließend wieder korrekt de-serialisiert werden kann.

5.2.2.2. Nachrichtenprotokoll zur Übertragung der Seiten

Zur Übertragung von Seiten wird die Subklasse `'P'` für die Migrationsnachrichten wie in 5.1.5.1 beschrieben eingeführt. Diese Nachrichten setzen sich aus dem serialisierten `Buffer Descriptor` und den Bytes der eigentlich zu übertragenden Seite zusammen. Versendet werden die Nachrichten anschließend über den in 5.1.5 eingeführten Nachrichtenpuffer.

Durch die Übertragung der Seite gibt die Quelle die Ownership über diese Seite an das Ziel ab, sodass die `LocationMap` entsprechend auf `REMOTE_OWNERSHIP` angepasst werden muss. Ab jetzt muss das Quellsystem für Zugriffe auf die Seite im Allgemeinen zunächst die Rückgabe der Ownership vom Zielsystem beantragen.
Um das Volumen der Datenübertragung zu verringern, wird für die Übertragung der Seiten eine Komprimierung eingesetzt. Dazu wird ausgenutzt, dass im `PageHeader` einer Seite Verweise auf den freien Teil der Seite gespeichert sind. Dieser freie Teil kann bei der Übertragung ausgelassen und anschließend auf dem Zielsystem vor dem Einfügen in die Relation wieder erstellt werden.

5.2.2.3. Deserialisierung und Verarbeitung einer Seitennachricht

Das Ziel empfängt die von der Quelle versendeten Seitennachrichten und verarbeitet sie weiter. Dazu werden die Nachrichten zunächst deserialisiert und anschließen wird die in die Nachricht eingebundene Seite extrahiert.
Erhält nun das Ziel eine solche Nachricht wird als erstes ein Pufferplatz ermittelt, in den die übertragene Seite geschrieben werden kann. In der Methode `rebuildBufferDesc` wird dazu zunächst die Informationen über den `Buffer Descriptor` deserialisiert. Mit Hilfe der am Anfang der Nachricht übertragenen clusterweit eindeutigen RelationID wird die entsprechende Relation geöffnet. Nun muss ein Pufferplatz im Bufferpool ermittelt werden. Dazu wurde die Methode `ReadRandomBuffer` geschrieben. Sie veranlasst Postgres eine nicht existierende, leere Pufferseite zur Verfügung zu stellen. Diese kann dann im weiteren Verlauf überschrieben werden.
Ist ein freier Pufferplatz gefunden wird zunächst dessen `Buffer Descriptor` aktualisiert. Dazu muss sein Buffertag angepasst werden. Aus den übertragenen Daten kann die Blocknummer, die Forknummer sowie das Suffix des alten `Buffer Descriptor` auf der Quelle übernommen werden. Der `relNode`-Wert kann nicht vom alten Puffer übernommen werden, sondern muss auf dem Zielsystem neu ermittelt werden. Da die entsprechende Relation bereits geöffnet ist, kann aus deren `Relation Descriptor` der `relNode`-Wert für die physischen Teilrepräsentationen dieser Relation übernommen werden. Dabei haben alle physischen `RelFileNodes`, die sich wie in 5.1.6 beschrieben unter anderem aus dem `Suffix` für den Tenantspace zusammensetzen, den gleichen `relNode`-Wert. Dieser kann deshalb direkt aus dem `Relation Descriptor` übernommen werden.
Dieser neue Buffertag muss nun mitsamt seiner ID in den entsprechenden `SharedBufHash` eingetragen werden, damit der neu angelegte Puffer auch wieder aufgefunden werden kann. Darüber hinaus wird die `LocationMap` für diese Seite auf dem Ziel angepasst, sodass die Ownership über diese Seite nun lokal auf dem Ziel vorliegt.
Anschließend muss nur noch die Seite selbst in den Puffer kopiert werden und die entsprechenden Flags im `Buffer Descriptor` gesetzt werden. Die Flags zeigen an, dass es sich nun um einen validen Buffer handelt und dass er `dirty` ist. Eine `dirty` Pufferseite muss, bevor sie ersetzt werden kann, auf den Externspeicher ausgeschrieben werden. Obwohl diese Pufferseite auf dem Zielsystem nicht nativ allokiert wurde, sondern ein anderer Puffer überschrieben wurde, ist das Ausschreiben dieser Seite möglich. Dies ist darauf zurückzuführen, dass die Relation vor Eintritt in diese Phase bereits die entsprechende physische Größe

besitzt, sodass die neue Seite in die Datei geschrieben werden kann.
Zusätzlich wird die LSN der Seite auf Null gesetzt um zu verdeutlichen, dass diese Seite auf dem Ziel noch nicht in einem WAL-Eintrag geloggt wurde. Anschließend werden die Sperren und Pins auf den Puffer aufgehoben und Postgres übernimmt seine weitere Verwaltung.

5.2.3. Übergang in die nächste Phase

Der Übergang in die dritte Phase mit dualem Betrieb des Quell- und Zielsystems beinhaltet mehrere Verwaltungsoperationen, die zur Synchronisation von Quelle und Ziel erfolgen müssen. Diese beinhalten in der Regel eine Nachrichtenübermittlung von der Quelle zum Ziel. Damit das Ziel die Nachrichten korrekt interpretieren kann, wurden für die einzelnen Operationen weitere Subklassen (siehe 5.1.5.1) eingeführt.
Die gesamte Phase der Erweiterung des Bufferpools wird als abgeschlossen betrachtet, nachdem diese zum Übergang notwendigen Operationen ausgeführt und die dabei entstehenden Nachrichten übermittelt wurden.

5.2.3.1. Übermittlung der FreespaceMap

PostgreSQL nutzt intern eine `FreespaceMap` genannte Datenstruktur. Diese dient dazu, bei Einfüge- oder Änderungsoperationen eine Seite in der Relation zu finden, welche noch genügend Platz für die neuen Daten hat. Die `FreespaceMap` wird dabei für jeden physischen `RelfileNode` einer Relation getrennt angelegt. Wenn die `FreespaceMap` nicht vorhanden oder leer ist, nutzt PostgreSQL in der Regel die letzte Seite einer Relation um dort die Daten einzufügen. Damit Postgres auf dem Zielsystem nicht grundsätzlich auf diese platzverschwenderische Methode zurückgreifen muss, wird die `FreespaceMap` der Quelle kopiert. Da die `FreespaceMap` auch in Datenbankseiten organisiert ist, kann zu ihrer Übertragung der in 5.2.2.1 eingeführte Mechanismus zur Seitenübertragung genutzt werden. Aufgrund dessen, dass der Zustand der `FreespaceMap` nicht relevant für das korrekte Ausführen von Transaktionen ist, kann ihre Übertragung ohne Schreibsperre auf parallel ausgeführte Transaktionen erfolgen.

5.2.3.2. Sperre der Transaktionsverarbeitung

Während des Übergangs in die nächste Phase dürfen keine Schreiboperation ausgeführt oder Transaktionen abgeschlossen werden. Dies ist notwendig, um einen konsistenten Zustand an das Zielsystem zu übermitteln, auf welchen anschließend die Synchronisation aufsetzen kann. Um diese Sperre durchzuführen, wurde mit `MTTransitionLock` ein neues `LightweightLock` eingeführt, welches vom entsprechenden Postgres Lockmanager verwaltet wird.
Der Migrationsprozess erwirbt diese Sperre am Anfang des Phasenübergangs. Da Schreiboperationen diese Sperre im Rahmen der `getGlobalMigrationState`-Operation zur Überprüfung der Ownership von Seiten erwerben, sind diese bis zur Freigabe der Sperre blockiert. Dadurch werden Änderungen auf diesem Tenantspace in dieser Übergangsphase

unterbunden. Nach Freigabe der Sperre werden dann die Operationen der nächsten Phase entsprechend ausgeführt.

5.2.3.3. Übermittlung der Relationengröße

Obwohl die clusterweiten Systeminformationen auf Quelle und Ziel gleich sind, müssen gewisse Informationen übermittelt werden, bevor die nächste Phase beginnen kann. Für jede Relation des Tenantspace wird auf der Quelle, dessen Größe und die Größe der zugehörigen `LocationMap` ermittelt. Diese Werte werden dann an den Zielknoten übermittelt. Zur Übertragung der Anzahl an Blöcken einer Relation wird die Subklasse `'B'` für die in 5.1.5.1 beschriebenen Migrationsnachrichten eingeführt.
Dieser legt bei sich die physischen Dateien mit der entsprechenden Größe an. Diese Dateien sind grundsätzlich erst einmal leer. Trotzdem sind sie für den korrekten Betrieb auf dem Zielknoten in der nächsten Phase essentiell. Dies liegt daran, dass Anfragen zunächst einmal die physische Größe einer Relation ermitteln, bevor sie einzelne Seiten in den Puffer holen. Damit diese Anfrage das korrekte Ergebnis liefert, muss also die Relation mit der entsprechenden Größe physisch vorhanden sein.

5.2.3.4. Übermittlung der LocationMap

Als nächsten Schritt müssen die `LocationMap`-Einträge, welche die Quelle zur Erfassung der Ownership einzelner Seiten angelegt hat, an das Ziel übertragen werden. Die Quelle überprüft dazu alle Transaktionen des Tenantspace, ob `LocationMap`-Einträge vorliegen und übermittelt diese dann ebenfalls als Pufferseite an den Zielknoten. Der Ablauf ist hier der selbe wie der in 5.2.2.1 für normale Pufferseiten erläuterte. Bevor der Zielknoten die erhaltenen `LocationMap`-Seiten allerdings bei sich einfügen kann, muss er diese zunächst, wie in 5.2.1.2 beschrieben, konvertieren.

5.2.3.5. Übermittlung des Commit-Logs

Die nächste Operation ist notwendig um den Multi-Version-Concurrency-Control-Mechanismus auf dem Ziel zu ermöglichen. Dazu müssen die Einträge des Commit-Logs (auch `CLog` genannt) übertragen werden. In der `CLog`-Datei wird der Transaktionszustand für jede aktive und abgeschlossene Transaktion gespeichert. Darüber kann ausgelesen werden, ob die Transaktion, welche ein Tupel in die Relation geschrieben hat, noch aktiv ist, committed oder aborted hat. Dementsprechend kann das Tupel dann von anderen Transaktionen gesehen und verarbeitet werden oder nicht. Das `CLog` ist ebenfalls über Seiten organisiert, sodass diese in den `Shared Buffer` eingelesen werden können. In eine Nachricht verpackt werden diese Seiten einzeln an den Zielknoten verschickt. Für diese Nachricht wird die Subklasse `'C'` in das in 5.1.5.1 beschriebenen Nachrichtenprotokoll eingeführt. Das Ziel überschreibt mit den erhaltenen `CLog`-Seiten sein eigenes, bis dato leeres `CLog`. Dadurch kann der Zielknoten die Anfragen in der nächsten Phase korrekt ausführen.

5.2.3.6. Übermittlung des Transaktionszustandes

Damit PostgreSQLs Multi-Version-Concurrency-Control auch auf dem Zielknoten korrekt arbeiten kann, müssen nun auch die Informationen zum globalen Transaktionsstand von der Quelle auf das Ziel übertragen werden. Die Übertragung des Transaktionszustandes gliedert sich dabei in zwei Abschnitte. Zunächst werden die aktuell auf der Quelle aktiven Transaktionen übertragen und anschließend noch globale TransaktionsIDs.
Auf der Quelle werden zunächst einmal die momentan aktiven Transaktionen in einer Liste gesammelt. Dazu werden ähnlich wie bei der Erstellung eines Multi-Version-Concurrency-Control-Snapshots alle Backends überprüft und deren aktuelle TransaktionsIDs gesammelt. In Postgres wird zur Verwaltung der Backends die `ProcArrayStruct`-Datenstruktur eingesetzt, welche nun auch zur Erstellung der Liste der auf der Quelle aktiven Transaktionen genutzt wird. Diese Liste wird anschließend serialisiert, indem jede TransaktionsID, durch `':'` getrennt, aneinandergereiht wird. Die serialisierte Liste der aktiven Transaktionen wird als Nachricht der neuen Subklasse `'A'` des Nachrichtenprotokolls (5.1.5.1) verschickt. Das Ziel deserialisiert die Liste und stellt diese zusammen mit den Daten aus dem zweiten Teil der Übertragung des Transaktionszustandes zur Durchführung von neuen Transaktionen auf dem Ziel zur Verfügung.
Der zweite Teil der Übertragung des globalen TransaktionsIDs ist notwendig, damit neue TransaktionsIDs auf dem Zielsystem vergeben werden können. Die Quelle verschickt die aktuell höchste TransaktionsID, damit das Erzeugen der TransaktionsIDs auf dem Ziel korrekt erfolgen kann. Zusätzlich werden auch die ID der letzten beendeten Transaktion und die älteste gültige TransaktionsID mit verschickt. Diese sind zusammen mit der bereits im ersten Teil verschickten Liste der aktiven Transaktionen für Verwaltungsaufgaben, wie das Erstellen eines Multi-Version-Concurrency-Control-Snapshots zur Auswertung der Sichtbarkeit von Tupeln, notwendig. Für diese Nachricht wurde ebenfalls eine eigene Subklasse mit der Bezeichnung `'T'` in das Nachrichtenprotokoll eingefügt. Der Zielknoten kann mit diesen Informationen seinen internen Transaktionsstand synchronisieren und die anfallenden Aufgaben, wie das Starten neuer Transaktionen, ausführen.
Anschließend darf nur noch das Zielsystem neue Transaktionen starten und dafür neue TransaktionsIDs generieren. Das Quellsystem erwirbt deshalb die `XidGenLock`-Sperre, wodurch verhindert wird, dass nach dieser Phase dort noch neue Transaktionen gestartet werden können.
Mit Abschluss dieser Verwaltungsoperation sind die Zustände von Quell- und Zielknoten synchron und die duale Phase der Migration kann beginnen. Damit der Quellknoten sicher gehen kann, dass auch die gesamten Daten an das Ziel übertragen wurden, wartet der Migrationsprozess solange, bis der `MessageWorker`-Prozess die Übertragung der letzten Nachricht bestätigt hat.

5.3. Dualer Betrieb von Quelle und Ziel

In der nun folgenden Phase werden auch Transaktionen auf dem Zielsystem zugelassen. Das Zielsystem erkennt den Eintritt in diese Phase mit dem eintreffen der Nachricht aus 5.2.3.6.

Bei deren Verarbeitung gibt das Ziel die `XidGenLock`-Sperre frei, wodurch nun Transaktionen zugelassen werden. Mithilfe des in 5.2.1 vorgestellten Ownershipkonzeptes und der Übertragung von Seiten und WAL-Einträgen findet die partielle Synchronisation während dieser Phase statt. Die Implementierung der partiellen Synchronisation ist in den Grundzügen bereits in den vorherigen Phasen vorhanden. Die Funktionalität des Ownershipkonzeptes wurde insofern erweitert, dass nun auch Operationen auf dem Zielsystem unterstützt werden. In diesem Abschnitt werden somit die durch den dualen Betrieb von Transaktionen auf Quelle und Ziel auftretenden Fälle behandelt. Die Phase endet, wenn alle Transaktionen auf dem Quellsystem beendet wurden.

5.3.1. Partielle Synchronisation

Die partielle Synchronisation wird genutzt um den dualen Betrieb von Quelle und Ziel zu ermöglichen. Obwohl durch das Ownershipkonzepts die Grundlagen für die partielle Synchronisation bereits bestehen, sind in mehreren Bereichen Erweiterungen notwendig. Zum einen muss nun das Durchführen von Transaktionen und damit auch das Anfordern von Seiten vom Ziel aus eingebunden werden. Des Weiteren bedingt das Konzept der temporären Ownership den Bedarf, Seiten mittels WAL-Einträgen direkt zu synchronisieren. Darüber hinaus entsteht im Rahmen der partiellen Synchronisation die Situation, dass Anfragen der Quelle abgelehnt werden können. Diese im Entwurf eingebrachte Einschränkung dient dazu zu verhindern, dass eine Seite bei ungünstigen Queries mehrmals zwischen Quelle und Ziel hin und her übertragen werden muss.

5.3.1.1. Starten neuer Transaktionen auf dem Ziel

Da auf dem Zielsystem nun neue Transaktionen möglich sind, werden die entsprechenden Abläufe zur Generierung neuer Transaktionen ausgeführt. Dabei werden die in 5.2.3.6 übermittelten Daten des Transaktionszustands genutzt. Mit deren Hilfe kann zum einen eine neue TransaktionsID bestimmt werden und zum anderen kann der Snapshot für die Multi-Version-Concurrency-Control erstellt werden. Für diesen Snapshot wird insbesondere auch die Liste der noch auf der Quelle aktiven Transaktionen genutzt, welche beim Phasenübergang übertragen wurde. Die im Snapshot hinterlegte Liste aller aktiven Transaktionen beider Systeme ist notwendig, damit für jedes Tupel die Sichtbarkeit eindeutig bestimmt werden kann.

5.3.1.2. Synchronisation von Commit- und Abort

Wenn Transaktionen auf der Quelle durch Commit oder Abort beendet werden, wird ein WAL-Eintrag erstellt. Dieser WAL-Eintrag wird mittels des Nachrichtenprotokolls an das Ziel geschickt. Zur Kommunikation von WAL-Einträgen wurde die neue Subklasse `'X'` erstellt. Das Zielsystem führt ein reguläres `Replay` dieses WAL-Eintrags aus, wie es auch bei der Recovery der Fall wäre. Dadurch wird das lokale Commit-Log auf dem Ziel für die im

WAL-Eintrag angegebene Transaktion aktualisiert. Zusätzlich wird die TransaktionsID dieser Transaktion auf dem Ziel aus der Liste der aktiven Transaktionen gestrichen. Diese Liste wurde im Rahmen der in 5.2.3.6 beschriebenen Übermittlung des Transaktionszustandes angelegt. Diese Liste der auf der Quelle aktiven Transaktionen wird wie in 5.3.1.1 beschrieben zur Erzeugung eines Snapshots für neue Transaktionen genutzt und muss deshalb aktuell gehalten werden, um die Konsistenz der Datenbank zu sichern.

5.3.1.3. Seitenzugriff des Quellsystems

Auf dem Quellsystem muss nun für Schreiboperationen die Ownership der betroffenen Seiten erworben werden. Dadurch wird das in 5.2.1.3 vorgestellte Konzept zur Einbindung der Ownership erweitert. Dabei kann es zu zwei Fällen kommen. Der erste Fall besteht darin, dass im Unterschied zu den vorherigen Phasen diese Ownershipanfrage nun vom Zielsystem abgelehnt werden kann. Dies geschieht, wenn das Ziel diese Seite bereits geändert hat. Im zweiten Fall erhält das Quellsystem die temporäre Ownership für die Seite und kann die Änderungsoperation ausführen.

Ablauf einer Ownershipanfrage Die Anfrage, die Ownership vom komplementären Migrationssystem zu erhalten, läuft wie folgt ab. Über die neue Subklasse `'O'` wird eine Anfrage über eine Seite verschickt. Diese Anfrage enthält neben den Informationen zur Identifikation der Seite auch die ProzessID des anfragenden Prozesses. Diese ProzessID ist notwendig um das (De-)Multiplexing von Anfragen und Antworten im Rahmen des `MessageWorker`-Prozess korrekt durchzuführen.
Die Beantwortung der Anfrage erfolgt in zwei Phasen. In der ersten Phase wird eine Zu- oder Absage verschickt. In der zweiten Phase wird, wenn nötig, auch die angefragte Seite an das andere System übermittelt.
Da die bei der Bearbeitung der Anfrage entstehende Antwort generell entweder die Zusage oder die Ablehnung der Ownershipanfrage beinhalten kann, wird dieses in der ersten Phase dem komplementären System mitgeteilt. Die Antwortnachricht selbst wird synchron über das Postgres interne `pq-interface` zum Nachrichtenaustausch zurückgesandt. Diese Antwort enthält wiederum neben den Daten zur Identifizierung der Seite auch die ProzessID des Prozesses, der ursprünglich die Anfrage gestellt hatte. Damit kann die Antwort, die an den `MessageWorker`-Prozess zurückgeschickt wurde, dem entsprechenden Prozess zugeordnet werden.
Wenn diesem Prozess die Ownership verweigert wurde, bricht er die laufende Transaktion durch ein `abort` ab. Dies kann laut des Ownershipkonzeptes nur auf der Quelle geschehen. Durch den Abbruch ist sichergestellt, dass kein Deadlock entsteht und alle Transaktionen auf dem Quellsystem in kurzer Zeit zum Ende kommen.
Wurde die Ownership gewährt, kann der Prozess entweder die Bearbeitung direkt durchführen oder er muss darauf warten, dass das antwortende System in der jetzt folgenden zweiten Phase die angefragte Seite übermittelt. Zur Übertragung dieser Seite wird der in 5.2.2.1 eingeführte Mechanismus genutzt.

5.3.1.4. WAL-Synchronisation

Ein Spezialfall der Ownershipanfrage ist das Erhalten einer temporären Ownership. Dies kann aufgrund des Ownershipkonzepts nur auf dem Quellsystem auftreten. Besitzt die Quelle also nur die temporäre Ownership über eine Seite, so synchronisiert sie nach dem Durchführen der Operation automatisch das Ziel. Dazu werden die für jede Schreiboperation entstehenden WAL-Einträge genutzt. Dieser WAL-Eintrag für Änderungen an Heap- oder Indexseiten wird im Rahmen der Migration ebenfalls wie in 5.3.1.2 für die WAL-Einträge von Commit und Abort an das Zielsystem übermittelt. Das Zielsystem führt ebenfalls das Replay dieser WAL-Einträge durch und synchronisiert sich somit mit dem Zustand dieser Seite auf der Quelle.
Die darauf folgende Rückgabe der Ownership für diese Seite von der Quelle zum Ziel wird über eine weitere Nachrichtenübermittlung durchgeführt. Dazu wurde die Nachrichtensubklasse `'G'` eingeführt, welche das Zielsystem dazu veranlasst seine `LocationMap` zu aktualisieren und somit die Ownership über diese Seite wieder zu übernehmen.

5.3.1.5. Seitenzugriff des Zielsystems

Dadurch dass nun Transaktionen auf dem Ziel ausgeführt werden, muss das Zielsystem nun auch bei jeder Operation die Ownership überprüfen. Dazu werden die in 5.2.1.3 und in 5.3.1.3 vorgestellten Mechanismen zur Abfrage der Ownership für Schreiboperationen ebenso auf dem Ziel ausgeführt. Zusätzlich wird allerdings auch bei jeder Aufforderung eine Datenbankseite lesen zu wollen die Ownership überprüft. Die Aufforderung zur Nutzung einer Datenbankseite geschieht in Postgres in der `ReadBuffer`-Methode, welche die Seite bereitstellt. Wird diese Methode zum Beispiel als Teil einer Leseoperation auf dem Ziel aufgerufen, wird zunächst mittels der `LocationMap` der aktuelle Zustand der Ownership abgefragt.
Dabei können verschiedene Fälle auftreten. Ist die Seite bereits im Besitz des Zielsystems, kann die Verarbeitung normal fortgeführt werden.
Ist der Zustand für diese Seite `TEMPORAL_OWNERSHIP`, so wurde die Ownership über diese Seite an die Quelle zurückgegeben. Das Zielsystem muss nun darauf warten, dass im Rahmen des in 5.3.1.4 vorgestellten Ablaufs diese Seite mittels des WAL-Eintrags aktualisiert wird. Nach dem Erhalt des WAL-Eintrags wird automatisch auch die Ownership wieder an das Ziel zurückübertragen.
Den letzten Fall stellt der Zustand `REMOTE_OWNERSHIP` dar. Dies bedeutet, das Ziel besitzt noch keine Daten dieser Seite. Das Zielsystem sendet nun eine Anfrage über diese Seite an das Quellsystem. Die Quelle wiederum stellt nun diese Seite bereit und übermittelt sie zusammen mit der Ownership zurück an das Ziel. Zur Seitenübertragung wird der bereits in 5.2.2.1 vorgestellte Mechanismus genutzt.

5.3.2. Übergang in die nächste Phase

Sobald alle Transaktionen auf der Quelle beendet sind, ist diese Phase abgeschlossen und die darauf folgende Phase beginnt. Im Verhalten des Zielsystems gibt es keine Unterschiede zwischen der aktuellen Phase des dualen Betriebs und der folgenden Phase.
Der Migrationsprozess überprüft nun auf dem Quellsystem die Backends, welche noch

Ausschnitt 5.2 Erweiterung der PGPROC-Struktur

```
struct PGPROC
{
    ...
    BackendId   backendId;      /* This backend's backend ID (if assigned) */
    Oid         databaseId;     /* OID of database this backend is using */
    Oid         roleId;         /* OID of role using this backend */
    TenantId    tenantId;       /* MT Id of tenant using this backend */
    ...
};
```

Transaktionen ausführen. Dazu wird die von Postgres zur Verwaltung der Backends genutzten `ProcArrayStruct`-Datenstruktur eingesetzt. Mit deren Hilfe kann nun die Anzahl an Backends, welche momentan noch eine Transaktion ausführen, ermittelt werden. Um generell Transaktionen, die nicht auf den zu migrierenden Tenantspace zugreifen, ausschließen zu können, wurde die im `ProcArrayStruct` verwaltete `PGPROC`-Datenstruktur erweitert. Sie enthält nun auch einen Hinweis auf den aktuell gesetzten Tenant. Sind keine Tenants mehr aktiv, so kann die nächste Phase eingeleitet werden.

5.4. Abschluss der Migration durch Übertragung der persistenten Daten

Auf dem Quellsystem sind nun alle Transaktionen beendet. Damit der für die Migration nötige Overhead nur so kurz wie möglich entsteht, ist das Ziel, nun die Migration möglichst schnell abzuschließen. Nach dem Abschluss der Migration erfolgt der reguläre Betrieb auf dem Ziel vollkommen unabhängig von dem Quellsystem. Um die Migration abschließen zu können, müssen in dieser Phase die restlichen persistenten Daten von der Quelle auf das Ziel übertragen werden. Gleichzeitig besteht weiterhin wie in der vorherigen Phase die Möglichkeit für das Zielsystem, einzelne Datenbankseiten von der Quelle anzufordern. Sobald alle Daten von der Quelle kopiert und übertragen wurden, kann die Migration beendet werden.

5.4.1. Push von Seiten

Das wichtigste Instrument in dieser Phase ist der sogenante `Push` von Datenbankseiten. Dazu wurde die Funktion `pushRelations` implementiert. `pushRelations` wird auf der Quelle aufgerufen und öffnet alle Relationen und deren Indizes des Tenantspaces nacheinander. Für jede Datenbankseite jeder Relation überprüft die Funktion die Ownership. Ist die Ownership noch lokal auf der Quelle, dann bedeutet dies, dass die entsprechende Seite der Relation bis jetzt noch nicht auf das Ziel übertragen wurde. Folglich wird die entsprechende Seite über einen Aufruf von `ReadBuffer` in den Puffer der Quelle eingelesen und so zur Übertragung bereitgestellt. Die Übertragung und das anschließende Einfügen der Seite auf dem Ziel erfolgt nach dem bereits in 5.2.2.1 vorgestellten Protokoll zur Übertragung von Pufferseiten. Nach der Iteration über alle Seiten aller betroffenen Relationen besitzt nun das Ziel die Ownership über all diese Seiten und somit auch die kompletten Daten des Tenantspaces. Dies erfüllt somit die Bedingung für den Abschluss dieser Phase und damit auch der Migration. Es folgt nun noch die letzte Phase, welche Aufräumfunktionen durchführt.

5.5. Aufräumen temporärer Daten

In dieser Phase werden die für die Migration benötigten temporären Daten zurückgesetzt und somit der reguläre Datenbankbetrieb wieder ermöglicht. Die Durchführung dieser Phase wird von der Quelle aus an das Ziel kommuniziert, sodass beide Systeme die Migration abschließen können.

Aufräumen auf dem Quellsystem

Die wichtigste Aufgabe des Quellsystems ist es dem Ziel das Erreichen des Endes der Migration mitzuteilen. Für diese Nachricht wurde die Subklasse `'F'` für das in 5.1.5.1 beschrieben Nachrichtenprotokoll eingeführt. Nach dem versenden dieser Nachricht räumt die Quelle ihre temporären Daten auf und beendet den Kontext der Migrationsoperation. Zum Aufräumen gehört die `LocationMap`, welche die Ownership während der Migration beinhaltete, zu löschen. Dazu werden die physischen `LocationMap`-Dateien für alle Relationen des Tenantspaces gelöscht. Anschließend wird der in 5.1.4 beschriebene globale Migrationszustand `global_migration_state` wieder auf seine Ausgangswerte zurückgesetzt. Des Weiteren wird auch dem lokalen `MessageWorker`-Prozess das Ende der Migration mitgeteilt, sodass dieser sich beendet und seine Verbindung zurücksetzt. Außerdem wird die Sperre für neue Transaktionen aufgehoben. Dadurch ist sichergestellt, dass die Ausgangslage wiederhergestellt ist. Somit besteht nun die Möglichkeit die Migration eines anderen Tenantspaces auszuführen.

Aufräumen auf dem Zielsystem

Aus dem Zielsystem müssen ebenfalls der Migrationszustand `global_migration_state` zurückgesetzt, der `MessageWorker`-Prozess beendet und die `LocationMap` gelöscht werden. Durch das Beenden des `MessageWorker`-Prozess wird auch die zweite Verbindung zwischen Quelle und Ziel zurückgesetzt. Mit dem Zurücksetzen des Migrationszustand und das Löschen der `LocationMap` ist somit ebenfalls wieder die Ausgangslage hergestellt.

6. Abschließende Betrachtung

In diesem Kapitel wird zunächst eine Evaluation der Komplexität des Lösungsansatzes zur Migration eines mandantenfähigen Datenbanksystems durchgeführt. Dazu wird der Aufwand, welcher durch die Migration entsteht, in Bezug auf verschiedene Ressourcen abgeschätzt. Dabei wird neben dem Rechenaufwand und der Netzwerkbelastung insbesondere auf die durch die Migration bedingten Externspeicherzugriffe eingegangen.
Darauf folgt eine Betrachtung des Prototyps unter funktionalen Gesichtspunkten. Dabei wird insbesondere auf die Migration im Bezug auf die physischen Daten eingegangen. Zusätzlich werden die im Rahmen eines Versuchs erstellten Messungen vorgestellt und eingeordnet.
Dieses Kapitel schließt mit der Vorstellung weiterer Anknüpfungspunkte für die Migration von mandantenfähigen Datenbanksystemen. Dazu gehören sowohl Optimierungsansätze als auch Auswirkungen auf weiterführende Themengebiete.

6.1. Aufwandsabschätzung

Der Aufwand für die Durchführung der Live Migration von mandantenfähigen Datenbanken bildet einen wichtigen Faktor, um die Anforderungen zu erfüllen. Die Komplexität dieses Aufwands ist entscheidend, damit der Migrationsansatz auch das Ziel erfüllt, effizient und dynamisch auf Laständerungen zu reagieren. In diesem Abschnitt wird nun die Komplexität der für die Live Migration genutzten Algorithmen diskutiert. Dabei liegt der Schwerpunkt auf den drei Größen Rechenaufwand, Netzwerkbelastung und Externspeicherzugriff. Mit diesen wird die Belastung des Systems durch die Migration eingeschätzt. Darüber hinaus sind noch die Seiteneffekte auf den zur Migration parallel stattfindenden Transaktionsbetrieb zu beachten.
Die Komplexität der Migration unterscheidet sich in den einzelnen Phasen, da dort unterschiedliche Algorithmen zur Anwendung kommen. Des Weiteren sind der Aufwand auf der Quelle und auf dem Ziel unterschiedlich. Dies folgt aus dem Aspekt , dass die Quelle möglichst schnell entlastet und möglichst wenig durch die Migration zusätzlich belastet wird. Auf diese Aspekte wird nun in der Komplexitätsabschätzung für die einzelnen Dimensionen eingegangen.

6.1.1. Rechenaufwand

Der Rechenaufwand bildet eine Größe, mit der die Belastung des Systems eingeschätzt werden kann. Die CPU ist allerdings als Ressource nicht so kritisch wie das Netzwerk oder die

Externspeicherzugriffe. Der Rechenaufwand beeinflusst aber die Gesamtdauer der Migration und somit auch die Zeitspanne, in der der Migrationsoverhead das System beeinträchtigt.
Der Rechenaufwand während der Initialisierungsphase ist sowohl auf der Quelle als auch auf dem Ziel vernachlässigbar, da hauptsächlich die Logik für den Verbindungsaufbau ausgeführt wird.
Während der darauf folgenden Phase der Erweiterung des Bufferpools setzt sich der Rechenaufwand auf der Quelle aus zwei Teilschritten zusammen. Zum einen müssen die Relationen des zu migrierenden Tenantspaces bestimmt werden und zum anderen werden die Seiten im Bufferpool überprüft. Zur Bestimmung der Relationen des Tenantspaces werden mit Hilfe eines Systemkatalogs alle Relationen des Tenantspace ermittelt. Der Aufwand dafür hängt von der gesamten Anzahl der Relationen und der Anzahl an Relationen des Tenantspaces ab. Dieser Aufwand ist aber im Verhältnis zur gesamten Migration zu vernachlässigen.
Entscheidend in dieser Phase ist hingegen der Aufwand für die Ermittlung der Hauptspeicherseiten. Dieser hängt linear von der Größe des Bufferpools ab. Der Bufferpool befindet sich im Hauptspeicher und seine Größe wird beim Start der Datenbankinstanz festgelegt. Für jeden Pufferplatz wird überprüft, ob die sich darin befindende Seite Teil des Tenantspaces ist. Seiten, auf die dies zutrifft, werden dann über das Netzwerk verschickt. Allgemein entsteht bei der Übertragung Rechenaufwand durch das Anfordern von Sperren und durch die Aktualisierung der LocationMap. Der Gesamtaufwand ist somit abhängig von der Größe des Hauptspeichers und der Anteil an Datenbankseiten, die Teil des zu migrierenden Tenantspaces sind.
Zum Ende der Phase der Erweiterung des Bufferpools müssen auf der Quelle Verwaltungsinformationen gesammelt und an das Ziel übertragen werden. Diese dienen dazu, in der nächsten Phase den dualen Betrieb und die Synchronisation durchführen zu können. Dazu werden die FreespaceMap und die LocationMap der Relationen des Tenantspaces übertragen. Deren Anzahl ist linear abhängig von der Anzahl an Relationen des Tenantspaces. Allgemein gilt somit, dass der Übertragungsaufwand linear von der Größe der Relationen des Tenantspace abhängt.
Außerdem wird der Transaktionszustand ermittelt und übertragen. Dazu müssen alle aktiven Transaktionen ermittelt werden. Deren Anzahl ist auf die maximale Anzahl an aktiven Backends beschränkt. Da diese Anzahl für jede Datenbankinstanz fest definiert ist, ist der Aufwand für die Ermittlung des Transaktionszustands nahezu konstant.
Das Zielsystem muss während dieser Phase lediglich für die ankommenden Seiten Pufferplätze zur Verfügung stellen und diese dann mit den neuen Seiten füllen.
In der Phase des dualen Betriebs von Quelle und Ziel ist der Aufwand unabhängig von der Berechnungslogik und wird über die aktiven Transaktionen auf Quelle und Ziel bestimmt. Der Rechenaufwand hängt somit von der Anzahl der Lese- und Schreiboperationen, die während dieser Phase auf dem migrierenden Tenantspace ausgeführt werden, ab. Auf der Quelle betrifft dies nur die Schreiboperationen, während auf dem Ziel beide Operationstypen den Rechenaufwand, welcher zur Sicherstellung der Synchronität nötig ist, verursachen. Die Sicherstellung der Synchronität findet über die LocationMap statt. Der Rechenaufwand entsteht hier durch das Erwerben von Sperren und durch die notwendige Kommunikation zwischen den Systemen.
In der darauf folgenden die Migration abschließenden Phase werden die restlichen persistenten Daten des Tenantspaces an das Ziel übertragen. Der Rechenaufwand entsteht dabei, wie

bei den bisherigen Phasen, durch das Bereitstellen und Transferieren der Datenbankseiten. Der Aufwand hängt hierbei linear von der Größe der Relationen des Tenantspaces ab, da alle Seiten einmal eingelesen und an das Ziel übertragen werden müssen.
Die Aufräumarbeiten der letzten Phase beziehen sich hauptsächlich auf das Freigeben von nicht mehr genutzten Objekten. Dies hat nahezu keinen Einfluss auf die Leistung des Systems.
Insgesamt ist der Rechenaufwand relativ gering. Dies liegt daran, dass die meisten der hier vorgenommenen Berechnungen für das Datenbanksystem Routine sind und deshalb effizient implementiert wurden. Die Netzwerkbelastung und die Externspeicherzugriffe haben eine deutlich stärkeren Einfluss auf die Komplexität der Migration.

6.1.2. Netzwerkbelastung

Die Netzwerkbelastung durch die Migration ist aus mehreren Gründen ein wichtiger Faktor. Zum einen beeinflusst das zu übertragende Datenvolumen direkt die Dauer der Migration und somit die Zeitspanne, in der der Migrationsoverhead das System beeinträchtigt. Zum anderen müssen parallel zur Migration noch weitere Funktionen über das Netzwerk möglich sein. Dazu zählen beispielsweise der Empfang von Anfragen oder das Versenden der Antworten. Auf das gesamte Netzwerk des Clusters gesehen, spielen natürlich auch das Verhalten der anderen Server eine Rolle. Wenn diese beispielsweise, während auf zwei Servern des Clusters eine Migration durchgeführt wird, ein Backup über das Netzwerk erstellen, dann kann dies zu einer Überlastung des Netzwerks führen. Daher ist die Betrachtung der durch die Migration entstehenden Netzwerkbelastung wichtig, um die Folgen abschätzen und die Koordination des Clusters daran auslegen zu können.
Während der Migration tritt die erste signifikante Netzwerkbelastung in der zweiten Migrationsphase auf. Diese Phase erweitert den Bufferpool auf das Zielsystem. Dazu werden zunächst Daten mit einem Volumen, welches dem Anteil an Seiten des zu migrierenden Tenantspaces an dem Bufferpool entspricht, übertragen. Die Zeitdauer, welche für die Übertragung dieser Seiten benötigt wird, hängt von der zur Verfügung stehenden Netzwerkbandbreite und der Last durch parallele Transaktionen auf diesem Tenantspace ab. Diese Transaktionen haben einen Einfluss darauf, wie schnell die für die Übertragung nötigen Sperren der Seiten dem Migrationsprozess zugestanden werden.
Zum Ende dieser Phase wird der Migrationszustands an das Ziel übertragen. Dabei wird als erstes die FreespaceMap übertragen. Das Volumen der FreespaceMap hängt direkt von der Größe der Relationen des Tenantspaces ab. Mit jedem Byte der FreespaceMap wird eine Datenbankseite adressiert. Folglich hat die FreespaceMap nur ein sehr geringes Volumen. Dies bedeutet, dass nur ein Bruchteil des Tenantspacevolumen hier übertragen werden müssen. Anschließend folgt die Übertragung der LocationMap. Diese benötigt nur 2 Bit pro Relationenseite, sodass sie nur ca. $1/4$ des Volumens der FreespaceMap umfasst. Beide Größen hängen direkt vom Tenantspacevolumen ab, sind aber sehr gering. Die noch zusätzlich nötige Übertragung des Transaktionszustandes hat eine konstante Größe. Zusammenfassend sind diese Datenmengen für die Netzwerkbelastung vernachlässigbar.
In den nun folgenden Phasen findet die Netzwerkkommunikation in beide Richtungen statt. Dass heißt es werden sowohl, wie bisher, Daten von der Quelle an das Ziel als auch

Daten von dem Ziel an die Quelle verschickt. Das Volumen der versandten Daten hängt hier direkt von den zur Migration parallel ausgeführten Transaktionen ab. Dies liegt daran, dass diese die Ownership anfordern, Seiten übertragen oder Write-Ahead-Log-Einträge zur Synchronisation erzeugen. Der Netzwerkaufwand für diese drei Bereiche wird nun für alle Phasen übergreifend betrachtet.
Die Hauptlast entsteht hierbei durch die Übertragung der Datenbankseiten des Tenantspaces, welche noch nicht im Rahmen der Erweiterung des Bufferpools in der zweiten Phase übertragen wurden. Der Aufwand hierfür hängt direkt von dem Volumen des Tenantspaces ab, da diese Daten genau einmal von der Quelle an das Ziel übertragen werden müssen.
Zusätzlich müssen Anfragen über die Ownership von Seiten übermittelt werden. Die dabei entstehenden Nachrichten haben ein relativ geringes Volumen und werden deshalb in dieser Betrachtung vernachlässigt. Als drittes gibt es noch den Fall, dass die Write-Ahead-Log-Einträge zur Synchronisation an das Ziel übermittelt werden. Diese Situation entsteht, wenn die Quelle Änderungen an Datenbankseiten, die bereits an das Ziel übermittelt wurden, durchführt. Da die Quelle nur in den ersten Phasen Transaktionen ausführt, wurden bis dahin auch nur ein Teilbereich der Datenbankseiten an das Ziel übertragen. Es sind also nur ein Teil der Operationen auf der Quelle betroffen. Dies sind diejenigen Seiten, welche im Rahmen der Erweiterung des Bufferpools auf das Zielsystem übertragen wurden, oder die explizit durch das Ziel angefordert wurden. Die Menge der im Rahmen dieser Synchronisation zu übertragenen Daten hängt neben dem Anteil an Schreiboperationen auf der Quelle insbesondere von der Lokalität dieser Zugriffe ab. Greifen die Transaktionen auf der Quelle und dem Ziel häufig auf den gleichen Datenbereich zu, so steigt auch die Menge der zur Synchronisation notwendigen Daten, da damit auch die Wahrscheinlichkeit, dass die Seite bereits auf dem Ziel ist, steigt. Da ein Write-Ahead-Log-Eintrag im Verhältnis zu einer Datenbankseite im Allgemeinen relativ klein ist, ist davon auszugehen, dass die Netwerkbelastung hauptsächlich durch das Übertragen der Datenbankseiten entsteht. Übergreifend wird daraus ersichtlich, dass die gesamte Netzwerkbelastung hauptsächlich von der Größe des Tenantspaces abhängt und durch Kommunikation von der Quelle an das Ziel geprägt ist.

6.1.3. Externspeicherzugriffe

Die Externspeicherzugriffe sind für Datenbanksysteme der kritische Leistungsfaktor. Sie bilden die Ressource, welche die Leistung des Systems am stärksten begrenzt und somit auch die Grenze für die Skalierbarkeit eines Servers bilden. Das Ziel der Migration ist deshalb unter anderem die Quelle mittelfristig von Externspeicherzugriffen zu entlasten. Die Migrationslogik selbst sollte daher die Quelle möglichst wenig belasten. Deshalb stellen die Externspeicherzugriffe einen wichtigen Aspekt bei der Einschätzung der Komplexität der Migration dar.
Für die Pufferverwaltung, welche für einen Großteil der Externspeicherzugriffe verantwortlich ist, gilt, dass die Hauptlast durch Lesevorgänge entsteht. Schreibzugriffe auf den Externspeicher finden meistens nur im Rahmen eines Lesevorgangs statt. Dies geschieht dadurch, dass für einen Lesevorgang eine geänderte Seite aus dem Puffer entfernt werden

muss, um Platz zu schaffen. Diese geänderte Seite wird dann auf den Externspeicher geschrieben. Auf diesem Zusammenhang basierend werden in der folgenden Betrachtung nur die Lesezugriffe einbezogen.
Im folgenden werden die Zugriffe auf Systemkataloge vernachlässigt, da davon auszugehen ist, dass diese sich meistens bereits im Hauptspeicher befinden. Abgesehen davon ist der Umfang der Systemkataloge sehr gering. Außerdem werden die Externspeicherzugriffe auf der Quelle, welche durch die Transaktionsverarbeitung entstehen, ebenfalls ignoriert. Dies ist damit begründet, dass diese Zugriffe auch ohne die Durchführung der Migration ausgeführt werden müssten und somit das System nicht zusätzlich belasten. Darüber hinaus wird auch nicht auf die Externspeicherzugriffe auf dem Ziel eingegangen. Dies folgt daraus, dass das Ziel unter dem Aspekt ausgewählt wird, dass es momentan unter geringer Last steht. Abgesehen davon ist die Zahl an Externspeicherzugriffen auf dem Zielsystem in der Größenordnung der Quelle, da sie hauptsächlich durch die Übertragung von Seiten entsteht. Dies sind zuvor auf der Quelle vom Externspeicher eingelesen worden.
Externspeicherzugriffe entstehen während der Migration hauptsächlich durch das Einlesen von Datenbankseiten. Darüber hinaus entstehen allerdings auch Externspeicherzugriffe bei der Nutzung von Verwaltungsinformationen. Neben den Systemkatalogen spielt bei der Migration insbesondere die Nutzung der LocationMap für die Ownershipinformationen eine Rolle.
Während der Phase der Erweiterung des Bufferpools entstehen auf der Quelle durch die Migration keine zusätzlichen Externspeicherzugriffe zum Auslesen von Seiten, da sich diese bereits im Puffer befinden. Für die zu übertragenden Seiten muss allerdings die LocationMap angepasst werden. Dazu müssen die entsprechenden LocationMap-Seiten in den Hauptspeicher eingelesen werden. Wie in 6.1.2 beschrieben hängt die Anzahl dieser Seiten von der Größe des Tenantspaces ab. Da eine LocationMap-Seite die Ownership vieler Datenbankseiten verwaltet, wird auf sie häufig zugegriffen. Deshalb ist die Wahrscheinlichkeit gering, dass LocationMap-Seiten wieder aus dem Bufferpool herausgeworfen werden. Somit ist im Mittel während der Migration für jede LocationMap-Seite nur ein wenig mehr als ein Externspeicherzugriff notwendig. Zusammen mit dem geringen Volumen der LocationMap sind diese Zugriffe vernachlässigbar.
Am Ende dieser zweiten Phase wird die LocationMap und die FreespaceMap an das Ziel übertragen. Beide haben ein relativ geringes Volumen und befinden sich teilweise schon im Hauptspeicher, sodass die daraus resultierenden Externspeicherzugriffe sehr gering sind.
In der dualen Phase werden nun auch Transaktionen auf dem Ziel gestartet. Diese können Seiten auf der Quelle anfordern, welche die Quelle teilweise erst vom Externspeicher einlesen muss. Würde man die Migration nicht durchführen, dann würde die Quelle diese Transaktion selbst ausführen und müsste deswegen diese Seiten ebenfalls vom Externspeicher einlesen. Deshalb entsteht hier kein zusätzlicher Aufwand durch die Migration.
Die sich anschließende Phase zur Übertragung der persistenten Daten von der Quelle an das Ziel ist bezüglich der Externspeicherzugriffe am kritischsten. Hier müssen auf der Quelle sequentiell alle Daten der Relationen des Tenantspaces vom Externspeicher eingelesen werden, damit sie über das Netzwerk verschickt werden können. Der Gesamtaufwand ergibt sich aus dem Volumen des Tenantspaces abzüglich der bereits an das Ziel übertragenen Seiten. Diese hier durch die Migration zusätzlich entstehenden Externspeicherzugriffe können auf der Quelle die parallel auf anderen Tenantspaces laufenden Transaktionen negativ beeinflussen,

da sie die Bandbreite der Externspeicherzugiffe auslasten. Dem steht gegenüber, dass die Quelle bereits von der direkten Transaktionslast des migrierenden Tenantspaces entlastet ist.

6.1.4. Hauptspeicherbelastung

Die zusätzliche Belastung des Hauptspeichers durch die Migrationslogik ist konstant und somit unabhängig von der Ausführung der Migration. Dies liegt daran, dass jede Datenbankinstanz für die Migrationslogik einen festen Bereich im Hauptspeicher allokiert. Dies geschieht beim Start der Datenbankinstanz und somit unabhängig von der Durchführung der Migration. Dieser Hauptspeicherbereich dient dann während der Migration für das Speichern von Daten, die zur Interprozesskomunikation dienen. Dazu gehören beispielsweise Nachrichten, die für die Kommunikation zwischen der Quelle und dem Ziel in diesem Hauptspeicherbereich gepuffert werden. Dieser Puffer hat aufgrund des fest allokierten Hauptspeicherbereichs eine unveränderliche Größe. Somit bildet das Volumen dieses Puffers die einzige Hauptspeicherbelastung des Systems. Dieser Puffer hat allerdings im Verhältnis zum gesamten Hauptspeicher ein geringes Volumen, sodass dadurch keine Beeinträchtigung des Systems entsteht.

6.1.5. Seiteneffekte

Dieser Abschnitt beschäftigt sich mit den Seiteneffekten, die durch die Integration und Ausführung der Migrationslogik für den Datenbankbetrieb entstehen und noch nicht in den bisherigen Abschnitten behandelt wurden. Die Seiteneffekte bilden Auswirkungen auf die Leistung des Datenbanksystems, welche über die direkten Migrationsziele hinausgehen. Generell lassen sich diese Auswirkungen in drei Bereiche gliedern. Dazu zählen zum einen Effekte, die unabhängig von der Ausführung der Migration auftreten. Zum anderen kann die Ausführung der Migration selbst Seiteneffekte verursachen. Als letztes ist die Situation nach der Durchführung der Migration zu betrachten.
Unabhängig von der Ausführung der Migration entstehen nur sehr geringe Seiteneffekte, wie beispielsweise der in 6.1.4 erwähnte leicht erhöhte Hauptspeicherbedarf. Eine größere Rolle spielen hingegen die direkt durch die Migration ausgelösten Seiteneffekte. Diese zeigen sich insbesondere auf der Quelle der Migration, da diese laut Annahme bereits unter einer besonders hohen Last steht. Auf der Quelle ist somit zu beachten, dass neben den Transaktionen auf dem zu migrierenden Tenantspace auch die anderen Tenantspaces unter Transaktionslast stehen. Als Seiteneffekt der Migrationslogik auf diese parallelen Transaktionen ist das kurze Aufschieben aller Transaktionen während des Übergangs am Ende der zweiten Phase zum dualen Betrieb zu nennen. Während bei diesem Übergang das Commit-Log ausgelesen und von der Quelle auf das Ziel übertragen wird, können keine neuen Einträge in das Commit-Log geschrieben werden. Somit müssen alle Transaktionen mit ihrer Beendigung warten, bis die Seiten des Commit-Logs übertragen sind. Da diese Sperren auf das Commit-Log nur sehr kurz bestehen, ist dieser Einfluss auf die parallelen Transaktionen jedoch gering.

Die Leistung im Transaktionsbetrieb auf den von der Migration unabhängigen Tenantspaces auf der Quelle wird allerdings durch die letzte Migrationsphase, in der die persistenten Daten von der Quelle an das Ziel übertragen werden, beeinflusst. Insbesondere die in 6.1.3 beschriebenen Externspeicherzugriffe zum Auslesen der persistenten Daten für die Migration können die für den normalen Transaktionsbetrieb zur Verfügung stehende Bandbreite an Leseoperationen auf der Quelle beeinträchtigen. Da die Datenbankseiten für die Migration durch die normale Pufferlogik in den Bufferpool geladen werden, besteht die Gefahr, dass Pufferseiten verdrängt werden, welche von den Transaktionen den anderen Tenantspaces genutzt werden. Da auf die Pufferseiten für die Migration nur ein einziger Lesezugriff erfolgt, werden diese allerdings durch die Pufferverwaltung relativ schnell wieder für die Elimination bestimmt. Insgesamt kann diesem Einfluss auf die Transaktionsperformance auf der Quelle durch gesonderte Maßnahmen, wie die Begrenzung der Externspeicherzugriffe durch die Migrationslogik in dieser Phase, entgegen gewirkt werden.
Nach der Migration operieren die Systeme unabhängig, sodass keine weiteren Seiteneffekte mehr beachtet werden müssen.

6.2. Funktionstest

In diesem Abschnitt wird nun die Funktionalität des Prototypen vorgestellt. Der Funktionsumfang wurde im Kapitel Entwurf festgelegt. Die entsprechende Implementierung wurde in Kapitel Implementierung vorgestellt. Hier folgen nun zunächst eine Erläuterung der Migrationsfunktionalität anhand eines Beispiels. Darauf folgt ein Experiment zur Einschätzung der Leistungsfähigkeit des Prototyps.

6.2.1. Migration

Die in dieser Arbeit vorgestellte Migrationslogik sieht vor, dass eine Kopie eines Tenantspaces erstellt wird. Im Folgenden wird nun ein Beispiel eingeführt, anhand dessen die Migration durchgeführt wird. Dieses Beispiel ist nicht repräsentativ für den produktiven Betrieb der mandantenfähigen Datenbank. Es dient nur zur Verdeutlichung der Funktionalität des Prototypen.
Als Voraussetzung für die Migration gilt, dass auf Quelle und Ziel die selben Metadaten genutzt werden. Dazu zählt auch, dass die Nutzer auf beiden Systemen gleich sind. In dem für diese Arbeit ausgewählten Beispiel existieren die in Ausschnitt 6.1 auf Seite 94 dargestellten Metadaten für Nutzer.
Diese Datenbanknutzer lassen sich in drei Gruppen gliedern. Zum einen gibt es Mandanten, welche für das Attribut `rolmtkind 't'` stehen haben. Diese Mandanten haben Daten auf der Quelle in die Datenbank eingefügt und können auf diese zugreifen. Die Daten der Mandaten sind in zwei Tenantspaces organisiert. Die Zuordnung von Mandant zu Tenantspace findet über `rolmttenantspaceid` statt. Tenantspaces selbst haben für die interne Verwaltung mit `'g'` eine eigene `rolmtkind`-Gruppe.
Der `migrationuser` bildet eine weitere Voraussetzung für die Migration. Dieser Nutzer wird

Ausschnitt 6.1 Vorhandene Nutzer

```
postgres= select rolname, rolmtkind, rolmttenantid, rolmttenantspaceid
>>        from pg_authid;
       rolname        | rolmtkind  | rolmttenantid  | rolmttenantspaceid
 ---------------------+-----------+---------------+--------------------
 NOTENANTGROUP_GROUP  | g          |             0  |                  0
 migrationuser        | n          |             0  |                  0
 ts1_TS               | g          |             1  |                  1
 ts2_TS               | g          |             2  |                  2
 tenant1_1            | t          |             3  |                  1
 tenant1_2            | t          |             4  |                  1
 tenant1_3            | t          |             5  |                  1
 tenant2_1            | t          |             6  |                  2
 tenant2_2            | t          |             7  |                  2
 tenant2_3            | t          |             8  |                  2
```

Ausschnitt 6.2 Vorhandene Relationen

```
postgres= select relname,oid,relmtkind from pg_class where relmtkind != 'n';
    relname    |  oid  | relmtkind
 --------------+-------+-----------
 order         | 16394 | b
 neworder      | 16397 | b
 order_pkey    | 16400 | c
 neworder_pkey | 16404 | c
 order_c_id    | 16408 | c
```

benötigt, um für die Migration eine Verbindung zwischen Quelle und Ziel aufbauen zu können.
Das Beispiel beinhaltet eine Datenbank mit den in Ausschnitt 6.2 angegebenen Relationen. Die Datenbank besteht dabei aus den zwei nach Tenantspace getrennten Tabellen `order` und `neworder`, welche jeweils einen Primärschlüsselindex für den Zugriff besitzen. Zusätzlich besitzt die `order`-Relation noch einen weiteren Index. Diese Relationen sind Teil des Schemas `benchmark`.
Die drei Mandanten haben auf der Quelle jeweils 99.999 Zeilen in die `order`-Relation eingetragen. In Ausschnitt 6.3 auf Seite 95 ist eine beispielhafte Anfrage dargestellt, die dies für Mandant `tenant1_1` aufzeigt. Auf dem Ziel existieren, wie aus Ausschnitt 6.4 auf Seite 95 ableitbar, bereits die Mandanten und Relationen. Sie enthalten jedoch noch keine Einträge. Die Übermittlung der Daten geschieht erst durch die Ausführung der Migrationslogik.
Bei der Ausführung der Migration werden nun sämtliche Daten des Tenantspaces von der Quelle auf das Ziel kopiert. In Ausschnitt 6.5 auf Seite 96 werden die auf der Quelle physisch vorhanden Relationen aufgelistet. Der Dateiname der Relation setzt sich aus der RelationID

und einem Suffix zusammen. Die Zuordnung von RelationenID zur Relation lässt sich aus Ausschnitt 6.2 auf Seite 94 ablesen. Teil des zu migrierenden Tenantspaces sind alle Dateien mit dem Suffix '_1'. Die anderen Dateien sind nicht Teil des Tenantspaces und größtenteils leer, da diese Relationen momentan keine Daten enthalten. Der zu migrierende Tenantspace hat eine Größe von ca. 56 Megabyte. Diese setzt sich aus 27 Megabyte für die Relation `order`, sowie 13 beziehungsweise 16 Megabyte für die entsprechenden Indizes zusammen. Der Umfang der Relation `neworder` kann in diesem Beispiel vernachlässigt werden.

Ausschnitt 6.3 Daten auf der Quelle

```
postgres= set tenant tenant1_1;
 SET
postgres= select count(*) from benchmark.order;
  count
 -------
  99999
 (1 row)
```

Ausschnitt 6.4 Daten auf dem Ziel

```
postgres= set tenant tenant1_1;
 SET
postgres= select count(*) from benchmark.order;
  count
 -------
      0
 (1 row)
```

In Ausschnitt 6.5 auf Seite 96 ist ein Ausschnitt der physischen Daten auf der Quelle während der Migration zu sehen. Neben denen nach Tenantspace getrennten Relationen existieren Dateien mit dem Suffix '_1_mtlm'. Diese Dateien beinhalten die `LocationMap`, welche im Rahmen des Ownershipkonzeptes für die Synchronisation während der Migration benötigt wird. Nach der erfolgreichen Migration werden diese `LocationMap`-Dateien auf der Quelle als auch auf dem Ziel wieder gelöscht, da sie nicht mehr benötigt werden.
In Ausschnitt 6.6 auf Seite 97 sind nun die Dateien auf dem Ziel vor und nach der Migration gegenübergestellt. Vor der Migration kann man deutlich erkennen, dass alle Relationen noch leer sind. Dateien von Indexrelationen, die eine momentan leere Relationen indizieren, haben die Größe von 8 Kilobyte, da sie immer zumindest eine Seite enthalten.
Nach der Migration kann man erkennen, dass die Dateien, die zum migrierten Tenantspace gehören, die gleiche Größe wie auf der Quelle angenommen haben. Sie enthalten jetzt die kompletten Daten des Tenantspaces. Somit kann auf diese jetzt auf dem Ziel zugegriffen werden. Des Weiteren wurde die FreespaceMap für die Relation `order` übertragen. Diese

Ausschnitt 6.5 Dateien während der Migration auf der Quelle

```
ls -lh source.data/base/11523
  size | filename
 ------+------------
 ...
     0   16394_0
   27M   16394_1
   24K   16394_1_fsm
    8K   16394_1_mtlm
     0   16394_2
     0   16397_0
    8K   16397_1
    8K   16397_1_mtlm
     0   16397_2
    8K   16400_0
   13M   16401_1
    8K   16401_1_mtlm
    8K   16402_2
    8K   16404_0
   16K   16405_1
    8K   16405_1_mtlm
    8K   16406_2
    8K   16408_0
   16M   16409_1
    8K   16409_1_mtlm
    8K   16410_2
  ...
```

ist am Suffix `_fsm` zu erkennen. Die auch auf dem Ziel während der Migration erzeugten `LocationMap`-Dateien sind direkt nach der Migration gelöscht worden.
In Ausschnitt 6.7 auf Seite 97 sind nun zwei Anfragen auf dem Ziel durch den Mandant `tenant1_1` dargestellt. Dieser fügt zunächst einen weiteren Wert in die Relation `order` ein. Dies ergibt dann zusammen mit den von der Quelle übertragenen Werten 100.000 Zeilen des Mandanten in dieser Relation.
Die Migration wurde also erfolgreich durchgeführt. Konkret sieht man das im Vergleich zu der Anfrage aus 6.4 auf Seite 95. Vor der Migration waren bei dieser Anfrage natürlich noch keine Daten auf dem Ziel zugänglich.

Ausschnitt 6.6 Dateien vor und nach der Migration auf dem Ziel

```
ls -lh destination.data/base/11523
     before                  after
  size | filename       size | filename
 ------+------------   ------+------------
  ...                   ...
     0   16394_0           0   16394_0
     0   16394_1          27M   16394_1
                          24K   16394_1_fsm
     0   16394_2           0   16394_2
     0   16397_0           0   16397_0
     0   16397_1          8K   16397_1
     0   16397_2           0   16397_2
    8K   16400_0          8K   16400_0
    8K   16401_1          13M   16401_1
    8K   16402_2          8K   16402_2
    8K   16404_0          8K   16404_0
    8K   16405_1          16K   16405_1
    8K   16406_2          8K   16406_2
    8K   16408_0          8K   16408_0
    8K   16409_1          16M   16409_1
    8K   16410_2          8K   16410_2
  ...                   ...
```

Ausschnitt 6.7 Anfragen nach der Migration auf dem Ziel

```
postgres= set tenant tenant1_1;
 SET
postgres= INSERT INTO benchmark.order VALUES (0, 2, 2, 2, 2,
>>        CURRENT_TIMESTAMP, 2, 2, 2);
 INSERT 0 1
postgres= select count(*) from benchmark.order;
  count
 --------
  100000
 (1 row)
```

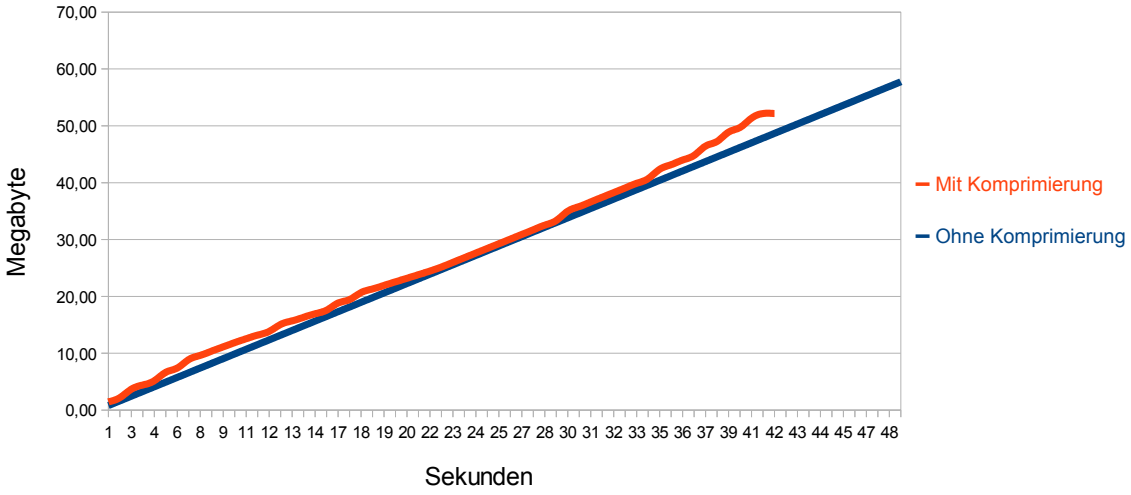

Abbildung 6.1.: Migration mit und ohne Seitenkomprimierung

6.3. Versuch

In diesem Abschnitt wird eine Messung bei der Durchführung der Migration vorgestellt. Es handelt sich dabei um ein nicht repräsentatives Experiment. Das System befand sich parallel zu den Messungen nicht unter Last, sodass die Migration diesbezüglich unter optimalen Bedingungen durchgeführt werden konnte. Abgesehen davon ist auch die Größe und die Struktur der Datenbank nur ein beispielhaftes Szenario. Deshalb lassen sich allgemeine Rückschlüsse nur in gewissen Punkten ziehen.
Die Basis für den Versuch bildet dabei das in 6.2.1 eingeführte Beispiel. Dieses Beispiel umfasst eine Datenbank mit einem Tenantspace, welcher eine Größe von ca. 56 Megabyte hat. Dieser Tenantspace wird mit Hilfe des Prototypen migriert.
In dem Diagramm 6.1 werden die Ergebnisse der Messung veranschaulicht. Dabei wurden mit der Beispieldatenbank zwei Versuche durchgeführt. Der erste Versuch ist eine Migration, bei der die in 5.2.2.2 vorgestellte Komprimierung der Seiten nicht angewandt wurde. Bei der Migration wurden in diesem Versuch insgesamt ca. 58 Megabyte an Daten übertragen. Die Migration dauerte 49 Sekunden.
Für den zweiten Versuch wurde die Seitenkomprimierung aktiviert. Dabei werden bei der Übertragung der Seiten von der Quelle an das Ziel die leeren Bereiche der Seiten ausgelassen. Diese Leerstellen werden anschließend wieder auf dem Ziel eingefügt, sodass die Seite wieder ihr ursprüngliche Größe hat. Bei der Migration im Rahmen dieses Versuchs wurden in 42 Sekunden 52 Megabyte an Daten übertragen.
Aus diesen beiden unter den genannten Bedingungen durchgeführten Versuchen lassen sich nun Rückschlüsse ziehen. Zum einen lässt sich aus dem ersten Versuch ablesen, dass ca. 2 Megabyte an Migrationsoverhead übertragen werden musste. Die Migrationslogik erzeugt

diesen Overhead einerseits im Rahmen des internen Kommunikationsprotokolls und andererseits durch die Übertragung von Verwaltungsinformationen, die für die Migrationslogik notwendig sind. Dieser Overhead lässt sich daher auch auf den zweiten Versuch übertragen, da der Overhead unabhängig von der Seitenkomprimierung entsteht. Für dieses Beispiel beläuft sich der Overhead bezogen auf die Netzwerkbelastung somit auf 3-5%.
Im Vergleich der beiden Messungen kann man erkennen, dass die Seitenkomprimierung ca. 6 Megabyte oder 10% des Volumens eingespart hat. Dadurch hat sich auch die gesamte Dauer der Migration um 7 Sekunden verkürzt. Diese Werte zeigen, dass die Komprimierung von Leerstellen einen erheblichen Vorteil mit sich bringt.
Außerdem lässt sich aus den Messungen ableiten, dass unter diesen Voraussetzungen das Volumen der zu übertragenen Daten der entscheidende Faktor für die Dauer der Migration ist. Somit hängt von dieser Größe auch die Dauer der durch die Migration entstehenden Beeinträchtigungen und Seiteneffekten ab.

6.4. Erweiterungen

In diesem Abschnitt folgt nun ein kurzer Überblick über weitere Themengebiete, die sich an diese Arbeit anschließen. Dabei werden sowohl Optimierungsvorschläge für die Migration, als auch Erweiterungen auf Basis des Migrationsalgorithmus vorgeschlagen. Darüber hinaus wird noch auf weitere Gebiete eingegangen, die sich aus den Anforderungen an ein mandantenfähiges Datenbanksystem ergeben und im Zusammenhang mit dieser Arbeit stehen.

6.4.1. Optimierungsansätze

Für die Durchführung der Migration gibt es noch weitere Optimierungsmöglichkeiten, die auf den in dieser Arbeit vorgestellten Entwurf zur Migration aufbauen. Die nun folgenden Ansätze trennen sich in drei Gruppen. Zunächst soll die Synchronisation zwischen Quelle und Ziel verbessert werden. Anschließend folgt ein Ansatz, der sich auf die Transaktionsleistung des Systems während der Migration positiv auswirkt, und ein Ansatz zur Verringerung der durch die Migration entstehenden Seiteneffekte.

6.4.1.1. Volle Synchronisation

Der in dieser Arbeit vorgestellte Ansatz zur Migration sieht eine partielle Synchronisation zwischen Quelle und Ziel vor. Dies bedeutet, dass nur die Änderungen, die auf der Quelle durchgeführt werden, auf das Ziel synchronisiert werden. Dies hat zur Folge, dass Transaktionen auf der Quelle nicht mehr auf auf dem Ziel geänderte Daten zugreifen können. Deswegen müssen nach dem bisherigen Ansatz diese Transaktionen auf der Quelle abgebrochen werden. Dieser Abbruch kann Auswirkungen auf die Applikationslogik haben oder muss durch die Clusterverwaltung abgefangen werden.

Ein erweiterter Migrationsansatz beinhaltet nun eine volle Synchronisation zwischen Quelle und Ziel, sodass keine Transaktionen mehr abgebrochen werden müssen. Hierbei kann das im Entwurf in 4.3.1 vorgestellte Ownershipkonzept genutzt werden. Das Ownershipkonzept enthält bereits die grundsätzliche Fähigkeit zur vollen Synchronisation. Auch die Integration des Ownershipkonzeptes in die Datenbanklogik kann für die volle Synchronisation nach den selben Mustern erfolgen, wie es bisher für die partielle Synchronisation der Fall ist.
Zur Umsetzung der vollen Synchronisation müsste das bestehende Ownershipkonzept in den nun folgenden Bereichen erweitert werden. Auf der Quelle ändert sich nur etwas bei Schreibanfragen. Wenn die Seite bereits an das Ziel übertragen und dort geändert wurde, dann müsste auf ein erfolgreiches Rückübertragen der Seite gewartet werden, anstatt die Transaktion abzubrechen. Auf dem Ziel würde das bedeuten, dass die Ownership über eine Seite unabhängig davon, ob sie bereits auf dem Ziel geändert wurde, wieder an die Quelle zurückgegeben werden können müsste. Die Quelle würde auch in diesem Fall nur die temporäre Ownership zurückerhalten, sodass das Ziel mittels des Write-Ahead-Log-Eintrags aktualisiert werden könnte.
Diese relativ einfache Erweiterung der Migrationslogik mittels der vollen Synchronisation hat zum Nachteil, dass die Recovery erschwert wird. Quelle und Ziel können nun nicht mehr unabhängig voneinander ihre lokalen Write-Ahead-Log-Einträge im Rahmen der Recovery zurückspielen. Dies liegt daran, dass sich diese Einträge nun teilweise auf Änderungen, welche auf dem komplementären System durchgeführt wurden, beziehen. Beispielsweise führt die Quelle eine Änderung auf einer Seite aus, die auf dem Ziel neu erstellt wurde. Das Erstellen dieser Seite wurde aber nur auf dem Ziel in das Write-Ahead-Log eingetragen, sodass diese Seite bei der lokalen Recovery auf der Quelle nicht zur Verfügung steht. Aus diesem Grund müssen für dieses Verfahren auch die Write-Ahead-Logs von Quelle und Ziel synchronisiert werden. Damit die Recovery nicht unnötig verkompliziert wird, bietet sich an, Write-Ahead-Log-Einträge für Seiten, die einmal auf dem Ziel geändert wurden, auch nur noch auf dem Ziel in das physische Write-Ahead-Log zuschreiben. Die Quelle sendet diese Einträge im Rahmen der Synchronisation an das Ziel, schreibt sie aber nicht mehr in sein eigenes Log. Diese Vorgehensweise bietet sich auch an, da die Write-Ahead-Log-Einträge zur Synchronisation sowieso an das Ziel übertragen werden. Das Ziel muss für diese Einträge noch seine eigene Log-Sequence-Number generieren und die erhaltenen Einträge in sein Write-Ahead-Log ausschreiben. Dadurch kann die Quelle komplett unabhängig vom Ziel recovern, da sie keine Einträge mehr besitzt, die sich auf Zielseiten beziehen. Das Ziel selbst führt den selben Recoverymechanismus aus, wie im Entwurf dieser Arbeit vorgestellt.
Ein weiteres Problem entsteht dadurch, dass zur Sicherstellung der Dauerhaftigkeit von Transaktionen die Write-Ahead-Log-Einträge vor dem Commit einer Transaktion auf den Externspeicher ausgeschrieben werden müssen. Diese hier vorgestellte Erweiterung führt zu der Situation, dass Änderungen von Transaktionen auf der Quelle ausschließlich auf dem Ziel in dessen Write-Ahead-Log eingetragen werden. Damit trotzdem die Dauerhaftigkeit gesichert ist, müsste der Befehl, auf der Quelle die aktuellen Write-Ahead-Log-Einträge auf den Externspeicher auszuschreiben, an das Ziel weitergegeben werden. Erst wenn das Ziel dann das Ausschreiben des Write-Ahead-Logs bestätigt, kann die Transaktion auf der Quelle durch das Commit beendet werden.
Durch diese Erweiterung wird verhindert, dass Transaktionen auf der Quelle abgebrochen werden müssen. Der Preis dafür ist neben der erhöhten Komplexität der steigende

Netzwerkaufwand. Dies liegt daran, dass nun Seiten auch von dem Ziel an die Quelle zurück übertragen werden. Außerdem verzögert sich das Commit der Transaktionen auf der Quelle, da über Netzwerkkommunikation sichergestellt werden muss, dass das Ziel die Write-Ahead-Log-Einträge ausgeschrieben hat.

Synchronisation auf Tupelebene Während des dualen Betriebs von Quelle und Ziel ist eine Synchronisation bei Schreibkonflikten notwendig. Insbesondere wenn beide Systeme auf die gleiche Seite schreibend zugreifen wollen, muss der Schreibkonflikt aufgelöst werden. Dies erfolgt bei der hier vorgestellten Migration durch das Ownershipkonzept. Bisher bedarf die Synchronisation die Übertragung der ganzen Seite beziehungsweise des Write-Ahead-Log-Eintrags. Dies geht immer mit der Übergabe der Ownership einher. Eine Möglichkeit die Schreibkonflikte bei einem erweiterten Ansatz zu lösen, ist eine Synchronisation auf Tupelebene durchzuführen. Dabei werden nur die geänderten Tupel an das komplementäre System übergeben und diese dann dementsprechend eingefügt. Die Übertragung eines Tupel verbraucht nur wenig Bandbreite und kann deshalb schnell durchgeführt werden. Durch die Einführung dieser neuen Ebene der Synchronisation für Schreibkonflikte könnte auch eine Trennung der Synchronisation von der Übergabe der Ownership erfolgen. Somit verspricht diese hier skizzierte Optimierung eine Verbesserung der Leistung bei in Konflikt stehenden Transaktionen.

6.4.1.2. Optimierung des Warm Caches

Im Rahmen der Phase der Erweiterung des Bufferpools auf dem Ziel werden Seiten des zu migrierenden Tenantspaces ausgewählt. Diese Seiten, die sich momentan im Puffer der Quelle befinden, werden auf das Ziel kopiert, um dort einen warmen Cache zu erzeugen. Nach dem bisherigen Verfahren wird dabei der Bufferpool der Quelle nur einmal nach Seiten des Tenantspaces durchsucht. Unter der Annahme, dass der Bufferpool des Ziels die gleiche Größe hat, bedeutet dies, dass der Bufferpool des Ziels noch nicht voll gefüllt ist. Dies liegt daran, dass sich im Bufferpool der Quelle auch viele Seiten von Tenantspaces, die nicht migriert werden, befinden. Durch eine Verlängerung dieser Phase bis nahezu alle Pufferplätze auf dem Ziel besetzt sind, könnte sich der Vorteil eines warmen Caches auf dem Ziel noch ausbauen lassen. Auf der Quelle werden durch den Transaktionsbetrieb sowieso regelmäßig neue Seiten des zu migrierenden Tenantspaces in den Puffer geladen. Folglich können diese durch eine Verlängerung dieser Phase ohne weitere Externspeicherzugriffe auf das Ziel übertragen werden.

6.4.1.3. Throtteling

Während der Phase der Übertragung der persistenten Daten von der Quelle an das Ziel kommt es zu verstärkten Externspeicherzugriffen durch die Migrationslogik. Dies kann, wie in 6.1.5 beschrieben, die Leistung der Quelle negativ beeinflussen. Ein Verfahren zum Throtteling dieser Externspeicherzugriffe kann diesen Einfluss abmildern. Dazu werden die

zu übertragenden Seiten nur in Abhängigkeit von der aktuellen Last auf der Quelle vom Externspeicher geholt. Dadurch wird zwar die Migrationsdauer verlängert, aber es werden gleichzeitig die Seiteneffekte gemildert.

6.4.2. Erstellen einer weiteren Kopie

Durch eine Erweiterung der Migrationslogik wird das Erstellen einer Kopie des Tenantspaces auf einen Server im Netzwerk ermöglicht. Diese Kopie kann dabei zu Backup- oder Testzwecken genutzt werden. Sie kann auch die Grundlage für einen Replikationsansatz bilden. Die Replikation dient dazu, die Anforderungen an ein mandantenfähiges Datenbanksystem bezüglich der Verfügbarkeit zu gewährleisten. In diesem Abschnitt wird nun ein Lösungsansatz skizziert, der auf die in dieser Arbeit vorgestellten Migrationslogik aufbaut.
Die Erstellung der Kopie des Tenantspaces soll bei der Migration als Erweiterung mit ausgeführt werden. Ziel ist es auszunutzen, dass für die Migration alle persistenten Daten in den Hauptspeicher geladen werden müssen. Andererseits muss im Gegensatz zu normalen Kopie- beziehungsweise Replikationsmechanismen kein Checkpoint durchgeführt werden, welcher alle Daten aus dem Hautspeicher auf den Externspeicher ausschreiben würde. Durch dieses Erstellen der Kopie im laufenden Betrieb entsteht somit kein Overhead bei den leistungskritischen Externspeicherzugriffen. Darüber hinaus werden diese Daten sowieso bereits bei der Migration über das Netzwerk übertragen. Diese Übertragung ist auch bei der Erstellung der Kopie auf einem entfernten System notwendig. Deshalb können diese für die Migration übertragene Daten auch für die Kopie genutzt werden, indem sie an den Kopie-Server weitergeleitet werden.
Die Kopie des Tenantspaces selbst stellt logisch gesehen den Zeitpunkt dar, bevor die Transaktionen auf dem Ziel der Migration beginnen. Die Kopie beinhaltet also alle Transaktionen, die auf der Quelle ausgeführt werden.
Die Kopie selbst könnte erstellt werden, indem alle Seiten, die im Rahmen der Migration an das Ziel übertragen werden, im selben Schritt auch an den Server für die Kopie übertragen werden. Aufgrund der Migration werden die kompletten Daten des Tenantspaces übertragen und mit sehr begrenztem zusätzlichen Aufwand können diese auch für die Kopie genutzt werden.
Damit die Kopie auch beispielsweise für Testanfragen genutzt werden kann, müssen neben den persisten Daten auch die Verwaltungsinformationen und Metadaten von der Quelle an den Server für die Kopie übertragen werden. Diese Daten werden ebenfalls bereits bei der Migration erstellt und an das Ziel übertragen. Diese können ebenfalls mit geringem Aufwand auch an den Kopie-Server übertragen und dort genutzt werden.

6.4.3. Koordinatorkomponente im Rechencluster

Wie bereits in dieser Arbeit erwähnt, wird für den effizienten Betrieb des mandantenfähigen Rechenclusters eine zentrale Koordinatorstelle benötigt. Diese ist für die Verwaltung des Clusters zuständig und startet somit Administrationsaufgaben, wie die Migration eines Tenantspaces.

Dieser Koordinator stellt eine zentrale Softwarekomponente dar, welche das System überwacht. Dazu sammelt es Leistungsdaten der einzelnen Server des Clusters. Mit diesen kann der Koordinator Leistungsengpässe erkennen und kurzfristig darauf reagieren. Eine Reaktion kann beispielsweise die Durchführung einer Migration sein. In diesem Fall muss er den Quell- und Zielserver auswählen und den zu migrierenden Tenantspace bestimmen. Darüber hinaus kann der Koordinator auch die Zusammensetzung der Mandanten eines Tenantspaces durch Split- und Mergeoperationen festlegen und so eine möglichst optimale Aufteilung finden. Heuristiken über die optimale Größe eines Tenantspaces im Verhältnis zur notwendige Flexibilität können den Koordinator dabei unterstützen.
Unabhängig davon ist der Koordinator für das zentrale Metadatenmanagement des Datenbankclusters verantwortlich. Dadurch dient er als Ansprechpunkt für administrative Aufgaben. Außerdem ist der Koordinator für das Routing der Anfragen im Cluster verantwortlich. Er besitzt das Wissen darüber, welcher Server momentan welche Mandanten verwaltet und teilt dies nach außen den Applikation mit. Diese können dann eine direkte Verbindung zu dem jeweiligen Datenbankserver im Cluster aufbauen.

7. Zusammenfassung

Diese Arbeit beschäftigt sich mit der Skalierbarkeit von mandantenfähigen Datenbanksystemen. Dazu wurde ein Migrationsansatz für mandantenfähige Datenbanksysteme entwickelt. Einen weiteren Schwerpunkt dieser Arbeit bildet die Implementation eines Prototypen, welcher die Migrationslogik umsetzt.
Mit dem Ziel, die Anforderungen und die Funktionalität des Systems zu bestimmen, wurden zunächst die allgemeinen Eigenschaften eines mandantenfähigen Datenbanksystems untersucht. Ein mandantenfähiges Datenbanksystem wird in der Praxis von einem Software as a Service-Dienstleister in einem Rechnercluster betrieben. Dabei wird das mandantenfähige Datenbanksystem zur Konsolidierung eingesetzt, woraus sich die Anforderung, dynamisch und effizient auf Laständerungen reagieren zu können, ergibt. In diesem Sinne wurde in dieser Arbeit eine Methode zur Migration als Erweiterung der Pufferskalierung entwickelt. Dieser Ansatz zeichnet sich unter anderem dadurch aus, dass er ohne Unterbrechung des Datenbankbetriebs die Migration durchführt.
Dem Migrationsansatz liegt eine Shared-Nothing-Architektur des Rechnerclusters zu Grunde. Diese Systemarchitektur wurde für das Design des Clusters ausgewählt, da sie die beste Skalierbarkeit und eine saubere Trennung zwischen System- und Datenbanklogik ermöglicht. Darüber hinaus zeichnet sich die Shared-Nothing-Architektur durch eine hohe Flexibilität aufgrund ihrer wenigen Voraussetzungen aus.
Darauf aufbauend wurde ein Entwurf für die Migration des Datenbanksystems erarbeitet. Der Entwurf sieht vor, einen Tenantspace, der die Daten mehrerer Mandanten umfassen kann, von dem Quellserver über das Netzwerk auf den Zielserver zu migrieren. Da parallel zu der Migration weiterhin Transaktionen auf diesem Tenantspace ausgeführt werden sollen, wurde im Rahmen des Entwurfs ein Ownershipkonzept auf Seitenebene entwickelt. Dieses ermöglicht die Synchronisation von Quelle und Ziel. Ein Schwerpunkt der Migration ist dabei, die Quelle möglichst schnell zu entlasten. Deswegen werden neue Transaktionen schon während der Migration auf dem Ziel gestartet. Damit diese auf dem Ziel ausgeführten Transaktionen auf einem warmen Cache operieren können, wird zunächst der Pufferinhalt der Quelle auf das Zielsystem übertragen. Ein Aspekt dieser Maßnahmen ist, die gesamte Transaktionsleistung des Systems möglichst wenig durch die Migration zu beeinflussen.
Auf Basis dieses Entwurfs wurde dann ein Prototyp implementiert. Dieser baut auf das PostgreSQL-Datenbanksystem auf, mit welchem bereits die Prototypen für Mandantenfähigkeit und Tenantspaces umgesetzt wurden.
Abschließend wurde die Funktionalität und der Berechnungsaufwand des Prototypen überprüft. Außerdem wurden weiterführende Themengebiete vorgestellt. Diese behandeln zum einen Optimierungsstrategien und zum anderen Erweiterungen, wie ein globales Koordinatorsystem, welches administrative Aufgaben im mandantenfähigen Datenbankcluster automatisiert.

A. Anhang

A.1. Weitere Illustrationen

Abbildungen A.1, A.2, A.3 und A.4 zeigen alternative Ansätze für einen Entwurf zur Migration.

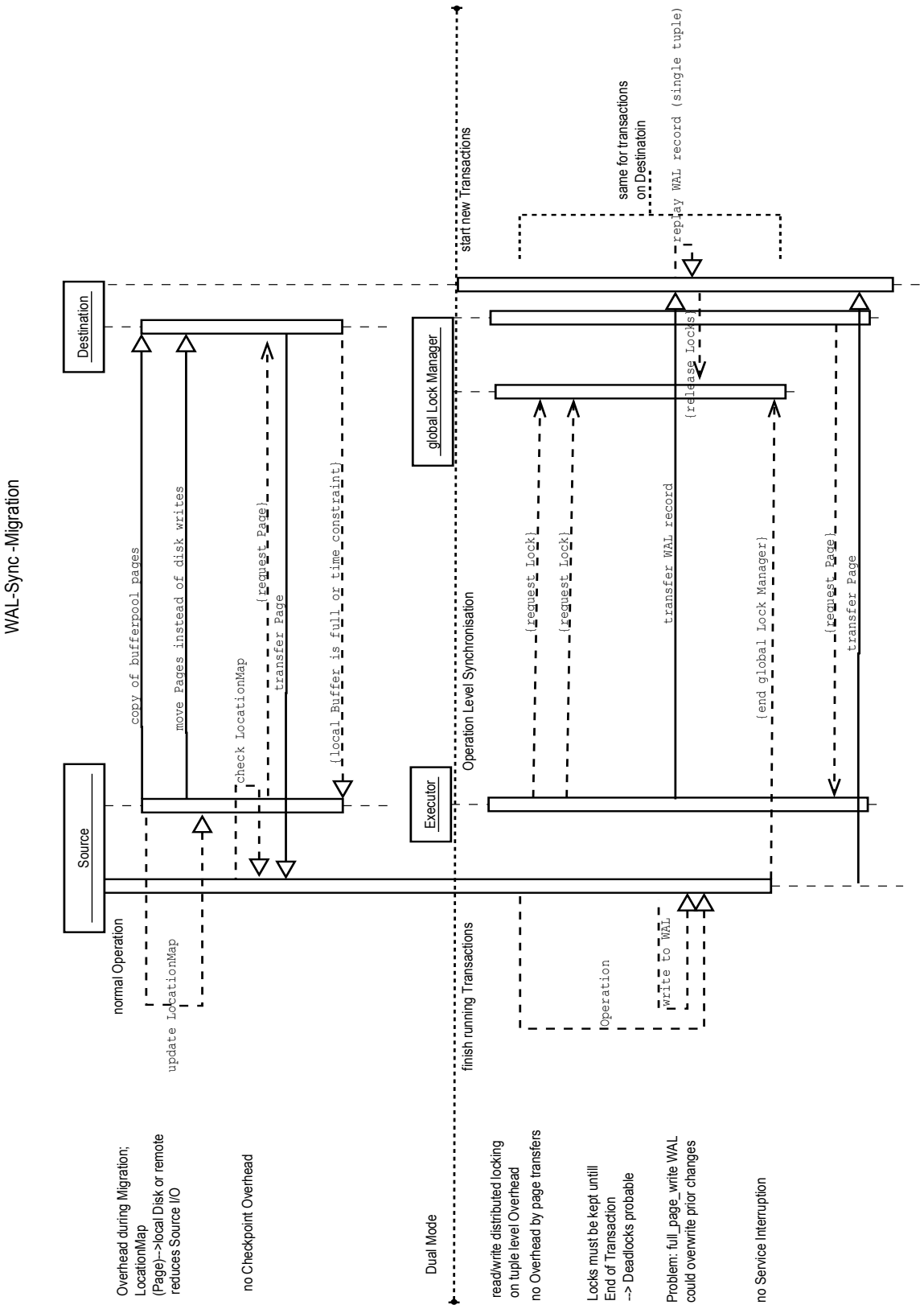

Abbildung A.1.: Migrationsansatz mit Synchronisation durch Nutzung des Write-Ahead-Logs

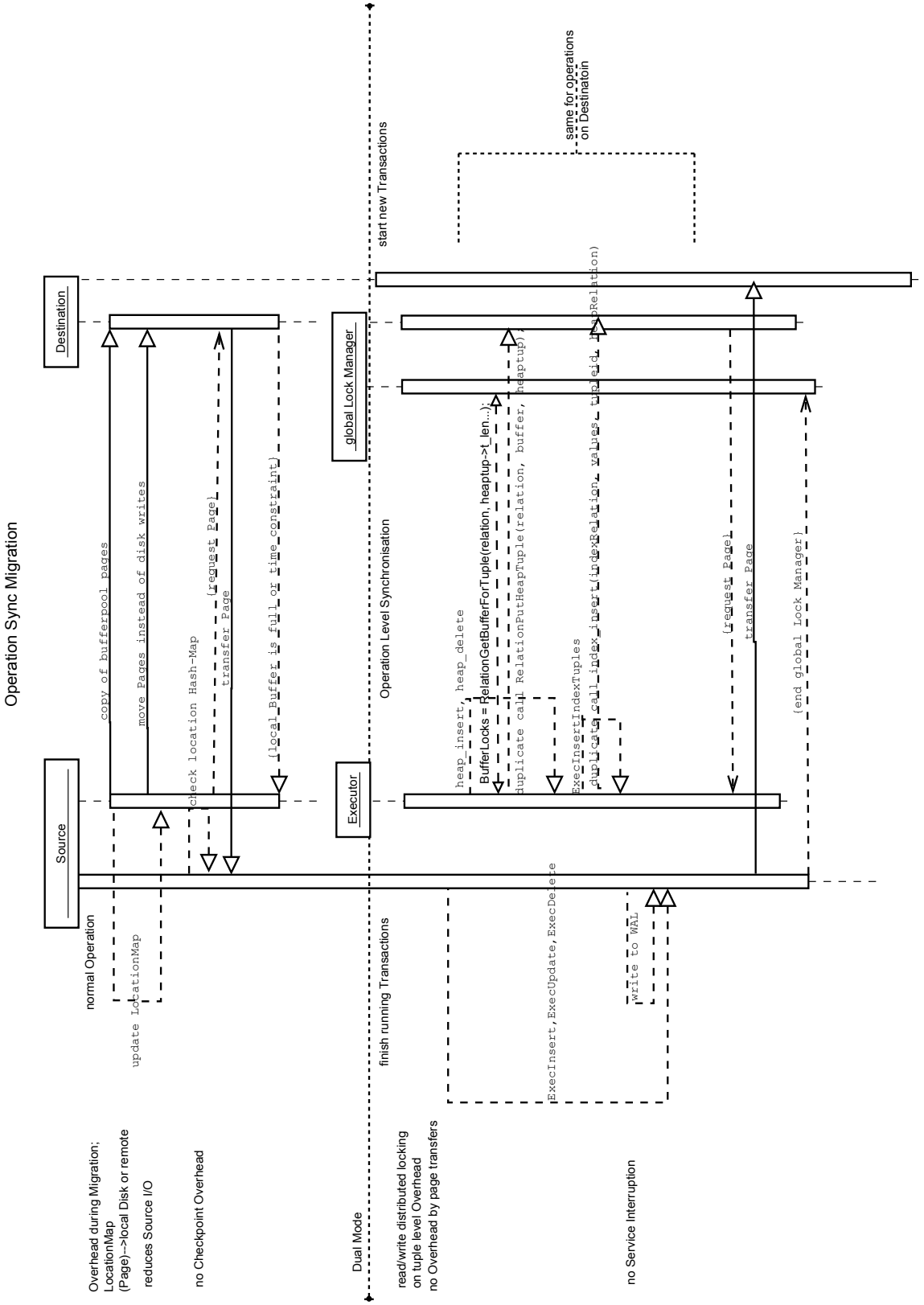

Abbildung A.2.: Migrationsansatz mit Synchronisation auf Operationenebene

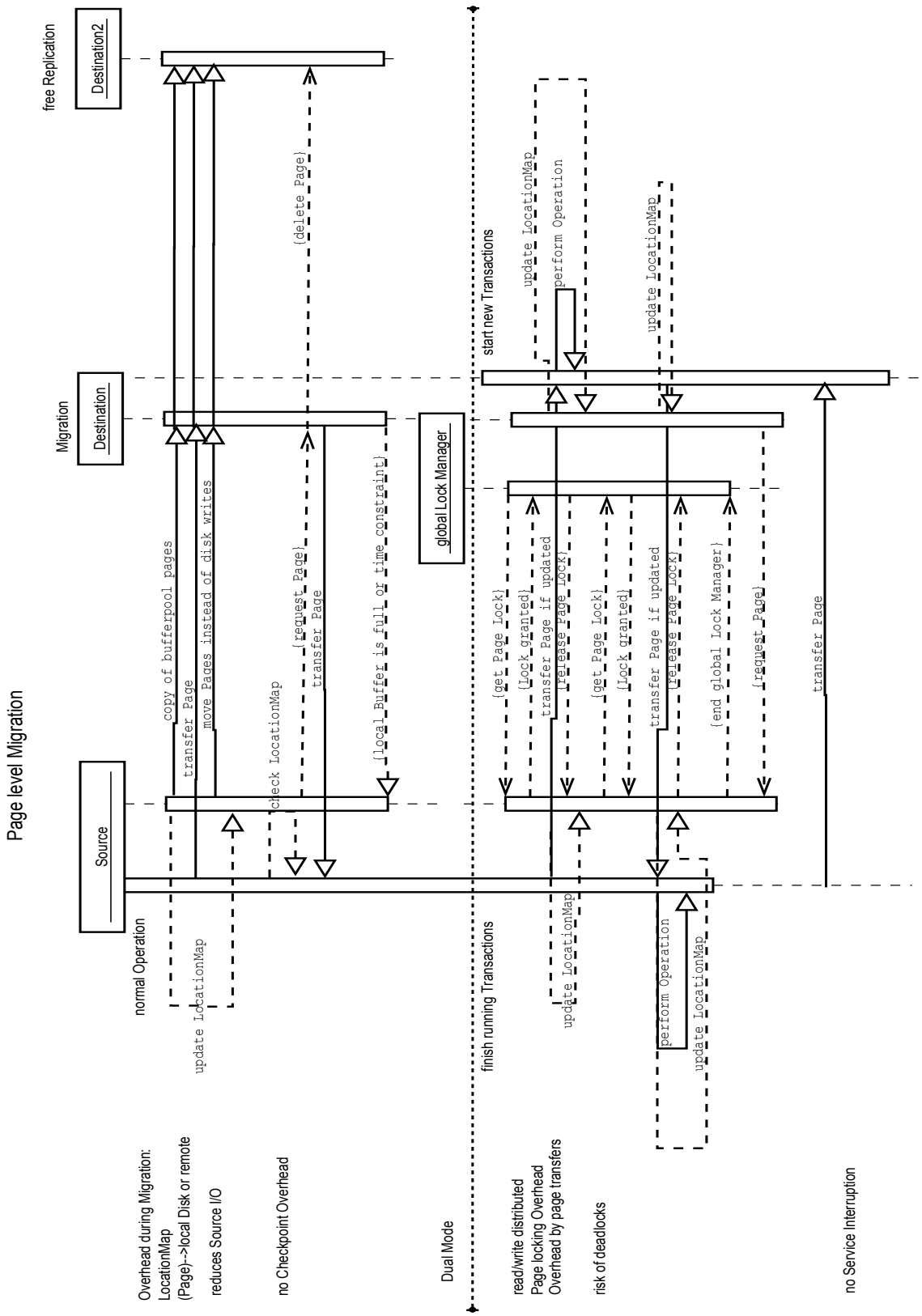

Abbildung A.3.: Migrationsansatz mit Synchronisation auf Seitenebene

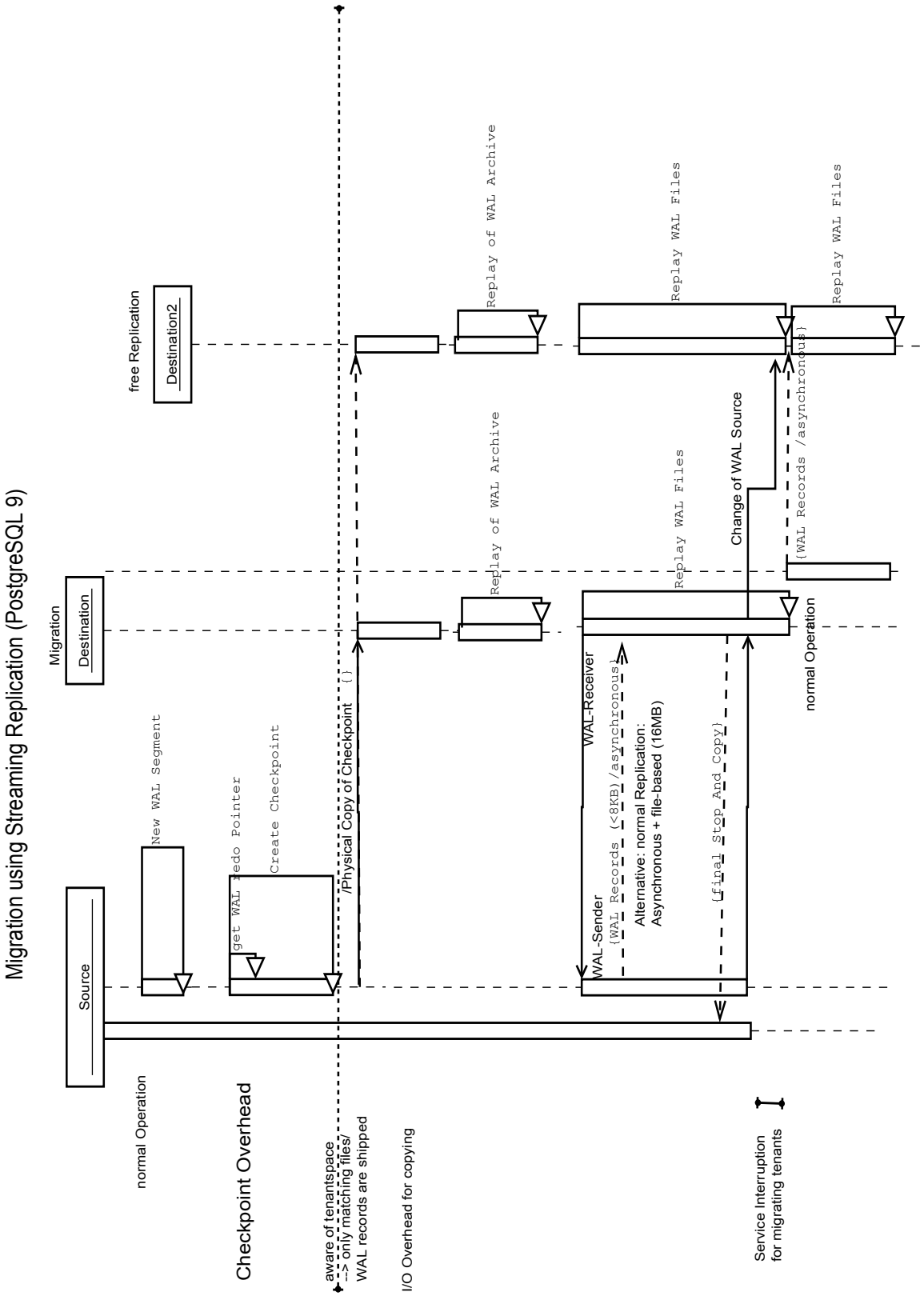

Abbildung A.4.: Migrationsansatz auf Basis von Streaming Replication

Literaturverzeichnis

[AJPK09] S. Aulbach, D. Jacobs, J. Primsch, A. Kemper. Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen. In J. C. Freytag, T. Ruf, W. Lehner, G. Vossen, editors, *BTW*, volume 144 of *LNI*, pp. 544–555. GI, 2009. (Zitiert auf Seite 9)

[BZ10] C.-P. Bezemer, A. Zaidman. Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pp. 88–92. ACM, New York, NY, USA, 2010. URL http://doi.acm.org/10.1145/1862372.1862393. (Zitiert auf Seite 13)

[CTL+05] H.-H. Chiou, P.-L. Tsai, J.-J. Lee, H.-F. Tung, C.-Y. Huang, C.-L. Lei. Extending Cluster File Systems beyond Last Miles. In *Proceedings of IEEE TENCON 2005*. 2005. (Zitiert auf Seite 33)

[DG90] D. J. DeWitt, J. Gray. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec.*, 19:104–112, 1990. URL http://doi.acm.org/10.1145/122058.122071. (Zitiert auf Seite 35)

[DNAA10] S. Das, S. Nishimura, D. Agrawal, A. E. Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Technical Report 2010-09, UCSB CS, 2010. URL http://www.cs.ucsb.edu/research/tech_reports/reports/2010-09.pdf. (Zitiert auf den Seiten 17 und 38)

[EDAA11] A. Elmore, S. Das, D. Agrawal, A. E. Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*. 2011. URL www.cs.ucsb.edu/~sudipto/papers/zephyr.pdf. (Zitiert auf den Seiten 18 und 38)

[GA94] J. Griffioen, R. Appleton. Reducing file system latency using a predictive approach. pp. 197–207. 1994. (Zitiert auf Seite 33)

[Hog] M. Hogan. Shared-Disk vs. Shared-Nothing - Comparing Architectures for Clustered Databases. URL http://www.scaledb.com/pdfs/WP_SDvSN.pdf. (Zitiert auf Seite 33)

[JA07] D. Jacobs, S. Aulbach. Ruminations on Multi-Tenant Databases. In A. Kemper, H. Schöning, T. Rose, M. Jarke, T. Seidl, C. Quix, C. Brochhaus, editors, *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS), Proceedings, 7.-9. März*

2007, Aachen, Germany, volume 103 of *LNI*, pp. 514–521. GI, 2007. (Zitiert auf Seite 13)

[KCF+05] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield. Live Migration of Virtual Machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, pp. 273–286. 2005. (Zitiert auf Seite 16)

[Ous90] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *USENIX Summer*, pp. 247–256. 1990. (Zitiert auf Seite 33)

[PgP11] PgPool Global Development Group. pgpool-II user manual, 2011. URL http://pgpool.projects.postgresql.org/pgpool-II/doc/pgpool-en.html. (Zitiert auf Seite 41)

[Pos09] PostgreSQL Global Development Group. PostgreSQL 8.4.8 Documentation, 2009. URL http://www.postgresql.org/docs/8.4/interactive/high-availability.html. (Zitiert auf Seite 30)

[Pos10] PostgreSQL Global Development Group. PostgreSQL 9.0.4 Documentation, 2010. URL http://www.postgresql.org/docs/9.0/static/index.html. (Zitiert auf Seite 31)

[Pos11] PostgreSQL Global Development Group. PostgreSQL - About, 2011. URL http://www.postgresql.org/about/. (Zitiert auf Seite 19)

[Sah05] A. Sahoo. Clustered DBMS Scalability under Unified Ethernet Fabric. In *Proceedings of the 2005 International Conference on Parallel Processing*, pp. 416–423. IEEE Computer Society, Washington, DC, USA, 2005. doi:10.1109/ICPP.2005.24. URL http://portal.acm.org/citation.cfm?id=1078032.1079416. (Zitiert auf Seite 33)

[SB11] O. Schiller, A. Brodt. Partitioned or Non-Partitioned Table Storage? Concepts and Performance for. Multi-tenancy in RDBMS. In *SEDE*. 2011. (Zitiert auf den Seiten 6, 14, 15 und 41)

[Sch10] B. Schiller. Mandantenpartitionierung in einem RDBMS, Studienarbeit Nr. 2241. Universität Stuttgart, Fakultät Informatik, 2010. (Zitiert auf den Seiten 11 und 19)

[Sch11] B. Schiller. Tenantspace - Ein Raum für Mandanten, Diplomarbeit Nr. 3110. Universität Stuttgart, Fakultät Informatik, 2011. (Zitiert auf den Seiten 11, 15, 19, 23, 65 und 70)

[Sha10] M. Shadmon. Clustered Database Storage Engine - Scaling MySQL in the Cloud, 2010. URL http://www.scaledb.com/pdfs/ScaleDB_Cloud_DBMS.pdf. (Zitiert auf Seite 35)

[SI09] G. H. Sockut, B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41:14:1–14:136, 2009. doi:http://doi.acm.org/10.1145/1541880.1541881. URL http://doi.acm.org/10.1145/1541880.1541881. (Zitiert auf Seite 16)

[SSBM11] O. Schiller, B. Schiller, A. Brodt, B. Mitschang. Native support of multi-tenancy in RDBMS for software as a service. In *EDBT*, pp. 117–128. ACM, 2011. URL http://doi.acm.org/10.1145/1951365.1951382. (Zitiert auf Seite 11)

[Wie] J. Wieck. Slony-I - A replication system for PostgreSQL. URL http://developer.postgresql.org/~wieck/slony1/Slony-I-concept.pdf. (Zitiert auf Seite 31)

Alle URLs wurden zuletzt am 20.11.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Urban Ficht)