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

Studienarbeit Nr. 2241

Mandantenpartitionierung in einem RDBMS

Benjamin Schiller

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Oliver Schiller
begonnen am:	26. November 2009
beendet am:	28. Mai 2010
CR-Klassifikation:	H.3.1, H.3.2, H.3.3

Kurzfassung

Der Bereitstellung von Software durch einen Dienstbereitsteller nach dem Vertriebsmodell Software as a Service wird eine zunehmends größere Bedeutung zuteil. Eine Aufgabe des Dienstbereitsteller besteht aus der Spezialisierung und Konsolidierung, um die Kosten pro Mandant zu senken. So können beispielsweise Ressourcen von mehreren Mandanten gemeinsam, Mandantenfähigkeit genannt, genutzt werden. In der Arbeit werden die sich daraus ergebenden Anforderungen an das Datenmanagement erläutert. Ein Konzept zur Partitionierung eines relationalen Datenbanksystems nach Mandanten wird vorgestellt. Ein auf PostgreSQL basierender Entwurf des vorgestellten Konzepts wird erläutert. Darauf folgt eine Evaluation des implementierten Prototyps.

Inhaltsverzeichnis

Abbildungsverzeichnis

Ausschnittsverzeichnis

1 Einleitung

In dem heutigen durch Schnelllebigkeit geprägten Zeitalter muss sich die IT-Infrastruktur immer wieder neu anpassen. So werden laufend neue nützliche Technologien entwickelt, die man schnellstmöglich einsetzen möchte, sofern sie einen Wettbewerbsvorteil anderen gegenüber darstellen. Ständig werden neue gesetzliche Richtlinien herausgegeben, zu deren Einhaltung man unter Umständen gezwungen ist, neue Hardware oder neue Software anzuschaffen. Von jetzt auf nachher kann es nötig sein, große Projekte aufzuziehen, um sich in einem gerade auftuenden Markt einen möglichst großen Anteil zu sichern, für die man neue IT-Ressourcen bereitstellen muss. Und im gleichen Moment muss man mit der Möglichkeit rechnen, dass man in diesem Markt keinen Fuß fassen kann, wodurch das Projekt aufgegeben werden muss und die teilweise teuren IT-Ressourcen übrig bleiben. Zudem hat die IT längst einen so hohen Stellenwert erreicht, dass sich viele einen Ausfall ihrer IT nicht mehr leisten können. Um eine hohe Verfügbarkeit zu erreichen, benötigt man natürlich redundante Systeme. Jedoch ist es ebenso wichtig, immer mit möglichen Angreifern Schritt zu halten, um zu verhindern, dass die Infrastruktur bei Angriffen in Mitleidenschaft gerät, oder genauso schlimm, dass Schäden durch Industriespionage entstehen.

Bei allen genannten Punkte fallen Arbeiten an, für die man qualifiziertes und dementsprechend teures Personal benötigt. Darüber hinaus muss ein IT-Entscheider die Punkte kennen und sich deren Bedeutung bewusst sein. Und selbst wenn letzteres der Fall ist, so sind einerseits einige Firmen auf Grund der Kosten oder des fehlenden Personals damit überfordert, und andererseits sehen viele andere in diesem großen Posten, den die IT verursacht, ein großes Einsparpotential. Eine gängige Möglichkeit dem Problem zu begegnen, ist die Auslagerung, indem man eine entsprechende Dienstleistung bezieht, die durch die Spezialisierung kosteneffizienter gestaltet ist.

Der Bezug von Fachkräften als Dienstleistung ist allgegenwärtig und daher nichts besonderes mehr. Anstatt jedoch nur die Fachkräfte zu beziehen, um die Infrastruktur zu warten, gibt es mittlerweile einige Anbieter, die die Software bei sich aufsetzen und dem Kunden zur Verfügung stellen. Diese Form der Bereitstellung nennt sich Software as a Service (Abk. SaaS). Dabei ist der Anbieter dafür verantwortlich, dass die Software zu jeder Zeit einwandfrei nutzbar ist. Um eine bestimmte Software anbieten zu können, muss diese jedoch einige Kriterien erfüllen. Zum Beispiel sollten ihre Funktionen über das Netzwerk erreichbar sein, damit diese vom Computer des Kunden aus genutzt werden können.

Als Beispiel für eine Software, die problemlos per SaaS bereitgestellt werden kann, kann

man sich einen Onlineshop vorstellen. Diese sind in der Regel über das HTTP-Protokoll erreichbar und können darüber mit Warenbeschreibungen, Preisen, usw. befüllt werden.

Für den Anbieter stellt sich die Frage, wie er die Kunden bzw. die für sie bereitgestellte Software auf seine Hosts verteilen kann. Würde er jedem Kunden einen eigenen Host zuweisen, müsste jeder Kunde den Host voll bezahlen, selbst wenn er eigentlich viel weniger Leistung benötigen würde. Daher macht es in der Regel mehr Sinn, dafür zu sorgen, dass die Leistung eines Hosts auf mehrere Kunden aufgeteilt werden kann.

Damit man die Aufteilung auf mehrere Hosts vornehmen kann, benötigt man ein mandantenfähiges System. In so einem System stellt die Firma bzw. der Kunde einen sogenannten Mandanten dar. Es wird garantiert, dass die zu einem Mandanten gehörenden Daten von denen anderer isoliert werden und er den alleinigen Zugriff darauf erhält. Dadurch wird eine Partitionierung nach Mandanten erreicht.

Ein Beispiel dafür wäre eine Virtualisierungssoftware, die einen Host in mehrere virtuelle Server aufteilt. Im Idealfall sieht jeder Mandant nur seine eigene Umgebung, von der aus er nicht ausbrechen kann, um zu denen der anderen Mandanten zu gelangen. Durch dieses Beispiel wird deutlich, dass es nicht die Software sein muss, die eigentlich angeboten werden soll, die mandantenfähig sein muss, sondern, dass die Partitionierung auch durch eine darunterliegende Schicht stattfinden kann.

Der vollständigkeithalber sei an dieser Stelle noch der umgekehrte Fall erwähnt, dass ein Kunde zeitweise mehr Leistung benötigt, als durch einen Host zur Verfügung gestellt werden kann. Die erforderliche Leistung muss dann von mehreren Hosts zur Verfügung gestellt werden. Eine Infrastruktur, die eine beliebige, teilweise kurzfristige Erweiterung der benutzten Ressourcen erlaubt, wird heute häufig mit dem Modewort Cloud bezeichnet. Die Software muss sich dazu dynamisch partitionieren und über mehrere Hosts verteilt laufen können.

Microsoft bietet beispielsweise seit einiger Zeit Exchange 2007 als Software as a Service in einer Cloud-Umgebung an. Beim einfachsten Lizenzmodell werden Gebühren je Benutzerkonto fällig, wodurch man nur so viel bezahlt, wie man tatsächlich benötigt, und man kann kurzfristig Konten kündigen und neue anlegen.

Einer in [1] nachzulesenden Analyse zufolge, wird bis zum Jahr 2011 ein Viertel der Software mittels dem Konzept SaaS bereitgestellt. Daher lohnt es sich, die damit verbundenen Themenbereiche zu erörtern. Beispielsweise ist ein großes Thema die Sicherheit der Daten. Dazu muss zum Einen eine unüberwindbare Trennung zwischen den Mandanten erfolgen. Zum Anderen gehört dazu, die Daten eines Mandanten gegenüber einem Zugriff einzelner Mitarbeiter der Firma, die die Software bereitstellt, zu schützen.

Ein weiterer Themenbereich ist es, eine geeignete Architektur für die Software zu finden. Zum Beispiel stellt sich die Frage, ob für jeden Mandant eine eigene Prozessinstanz gestartet werden sollte. Des Weiteren ist ein Punkt, wie die Software feststellt, zu welchem

Mandant eine eingehende Verbindung gehört. Da ein Mandant konzeptuell auf der obersten Ebene einer Anwendung samt deren Daten anzusiedeln ist, von dem alles darunterliegende abhängen kann, muss der Mandant vor einem Zugriff auf Teile der darunterliegenden Ebene und daher zu einem frühen Zeitpunkt bekannt sein. Beim eingangs erwähnten Onlineshop könnte der Mandant beispielsweise anhand der in der HTTP-Anfrage übermittelten Domain festgelegt werden.

Der Themenbereich, mit dem sich die folgende Arbeit befasst, ist das Datenmanagement für ein mandantenfähiges System. Das Datenmanagement muss dazu die Modellierung, das Abspeichern von Daten, die darauf auszuführenden Operationen, die Trennung zwischen den Mandanten und ein Sicherheitskonzept umfassen. Einige Ansätze, das Datenmanagement mandantenfähig zu gestalten, werden im folgenden Abschnitt beschrieben.

1.1 Ansätze

Nachdem erklärt wurde, wozu mandantenfähige Anwendungen benötigt werden, soll in diesem Abschnitt darauf eingegangen werden, welche Ansätze es gibt, um eine bestehende Anwendung mandantenfähig zu machen.

Dabei wird im Folgenden von einer Datenbankanwendung ausgegangen, die nach der in 1.1 beschrieben Architektur, also bestehend aus einer Anwendung, einer Middleware und eines relationalen Datenbanksystems (Abk. RDBMS), aufgebaut ist. Dadurch lassen sich die Ansätze in drei Gruppen untergliedern.

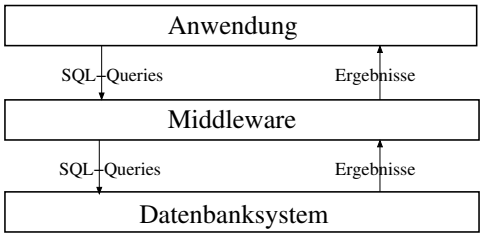

Abbildung 1.1: Architektur einer Datenbankanwendung

Eine Möglichkeit ist es, die Mandantenfähigkeit im Wesentlichen in der Anwendung zu organisieren. Das hat den Nachteil, dass dazu eine Menge an zusätzlichem Code je Anwendung benötigt wird. Dabei ist ein Großteil davon bei allen Anwendungen gleich. Zum Beispiel wird sich die Verwaltung der Mandanten kaum ändern.

Des Weiteren gibt es die Möglichkeit, die notwendige Logik in der Middleware unterzubringen, um zwischen mehreren Mandanten zu unterscheiden. Ein Nachteil hierbei ist, dass bei allen von der Anwendung an die Datenbank übermittelten Anfragen der Mandant eingefügt

und bei allen zurückgelieferten Ergebnissen der Mandant entfernt werden muss. Hierzu ist meist eine aufwändige Manipulation der Nachrichten notwendig.

Außerdem wird eine Middleware möglichst schlank gehalten, damit die durchzureichenden Anfragen schnell weitergegeben werden können. Dagegen spricht, dass für die notwendige Logik einiges an Zustandsinformationen gesammelt werden müssten. Zum Beispiel müssen je Mandant Statistiken über die abgesetzten Anfragen und über die abgespeicherten Daten angefertigt werden, damit die Datenbank darauf basierend optimiert werden kann.

Die zuvor genannten Ansätze haben einige gemeinsame Nachteile. Beispielsweise muss die Modellierung weiterhin in der Datenbank vorgenommen werden. Die Modellierung erfolgt durch das Anlegen von Datenbankobjekten in einem sogenannten Schema. Ein Objekt kann hierbei zum Beispiel eine Tabelle, eine View oder ein Trigger sein. Beim Anlegen dieser Objekte muss die Trennung der Daten zwischen den Mandanten vorgesehen werden. Dazu werden in [2] die folgenden zwei Vorgehensweisen vorgestellt:

1. Bei der Modellierung kann ein sogenanntes, in [3] erklärtes Shared Schema verwendet werden, um die Trennung zu erreichen. Bei diesem Schema erhält jede Tabelle eine zusätzliche Spalte, die angibt, für welchen Mandanten das jeweilige Tupel gespeichert werden soll. Es ist daher nicht möglich, dass ein Mandant Änderungen am Schema vornimmt. Dieser Ansatz wird zum Beispiel in [6] verwendet.

2. Außerdem kann die Modellierung für jeden Mandanten in einem eigenständigen Schema hinterlegt werden. In diesem Fall wird das Schema eines Mandanten auch als Private Schema bezeichnet. Der Nachteil ist offenbar, dass der Großteil des Schemas mit dem der anderen Mandanten übereinstimmt und diese Informationen daher redundant sind.

Damit bleibt noch die Möglichkeit, die Daten vom Datenbanksystem nach Mandanten trennen zu lassen. Da das Konzept der Mandanten im Wesentlichen die Datenorganisation betrifft, ist diese Möglichkeit offenbar passender als die anderen. Im Gegensatz zu den zuvor genannten sind damit Mandanten und Daten zusammen untergebracht und das Datenbanksystem kennt den Bezug zueinander, wodurch einige Optimierungen möglich werden.

Eine Optimierung wäre beispielsweise, jedem Mandant ein eigenes Schema zuzuordnen, in dem er Objekte anlegen kann, und die redundanten Teile der Schemata der Mandanten zusammenzufassen. Dadurch bietet sich ein Vorteil gegenüber den Ansätzen, ein Shared Schema oder ein Private Schema zu verwenden.

Des Weiteren können die ohnehin vom Datenbanksystem erstellten Statistiken verwendet werden, indem diese pro Mandant erhoben und gespeichert werden. Zudem kann ein mandantenfähiges Datenbanksystem so ausgerichtet werden, dass mandantenweise Operationen, wie das Migrieren aller Tupel eines Mandanten auf einen anderen Host effizient

gestaltet werden können. Die Anforderungen an ein mandantenfähiges Datenbanksystem werden im folgenden Abschnitt erläutert.

1.2 Anforderungen

Die Anforderungen eines mandantenfähigen Datenbanksystems unterscheiden sich in einigen Punkten von denen eines normalen Datenbanksystems. Sie werden im Folgenden anhand dem eingangs erwähnten Beispiel, dem Onlineshop, verdeutlicht.

Konsolidierungseffizienz muss genutzt werden, um einen Kostenvorteil bei gleichbleibender Geschwindigkeit zu bieten. Bei einem Onlineshop wäre es zum Beispiel denkbar, dass eine Tabelle mit Postleitzahlen von allen Mandanten gemeinsam genutzt werden kann und somit kostbare Ressourcen eingespart werden können.

Isolation muss Trennung zwischen Mandanten gewährleisten. D.h. ein Mandant muss Artikel, Kunden und Bestellungen hinzufügen können, die von einem anderen Mandanten weder gelesen noch gelöscht werden können.

Migrationskonzepte werden benötigt, um Mandanten bei höher werdenden Leistungsanforderungen auf freie Ressourcen zu migrieren. Für den Onlineshop bedeutet das, dass alle Daten auf ein neues System kopiert werden, ohne dass eine Bestellung eines Kunden während der Migration verloren geht, und der Shop weiterhin unter der gleichen Adresse erreichbar sein können muss.

Verfügbarkeit muss ununterbrochen gewährleistet sein. Beispielsweise darf ein Onlineshop eines Mandanten nicht gestoppt werden müssen, damit der eines anderen Mandanten migriert werden kann.

Datensicherheit in Form von redundanten Systemen und täglichen Sicherungen muss möglich sein. D.h. bei einem Ausfall muss es möglich sein, dass die Anfragen an den Shop eines Mandanten an ein anderes System weitergeleitet werden, das Zugriff auf die aktuellste Version der Daten hat. Weiterhin müssen Fehler, wie das versehentliche Löschen von Artikeln oder Kunden mithilfe von Sicherungen rückgängig gemacht werden können.

Dynamische Buchung der garantierten Leistung muss möglich sein, damit der Mandant diese jederzeit an seine tatsächlichen Anforderungen anpassen kann. Im Fall des Onlineshops könnte eine garantierte Leistung die Anzahl der Bestellungen pro Sekunde sein, die das System bewältigen können muss.

Skalierbarkeit bezüglich der Anzahl der Mandanten muss erreicht werden können. Beispielsweise darf es nicht sein, dass für jeden Shop ein eigener Prozess gestartet wird, falls der Kontextwechsel zwischen den Prozessen der Mandanten die durchführbaren Bestellungen pro Sekunde zu sehr beeinflusst.

Wartbarkeit durch einen Administrator muss nahtlos zum normalen Betrieb gewährleistet sein. D.h. zum Beispiel, dass der Administrator ein Attribut hinzufügen können muss, ohne dass alle Daten exportiert und anschließend importiert werden müssen.

1.3 Inhalt der Arbeit

Die Arbeit ist in mehrere Kapitel untergliedert. Auf dieses Kapitel, der Einleitung, folgt ein Kapitel über die Aufgabenstellung, das ein Konzept beschreibt, wie ein mandantenfähiges Datenbanksystem aufgebaut sein kann, und zudem erklärt, inwieweit dieses Konzept im Rahmen dieser Arbeit umgesetzt werden soll.

Das dritte Kapitel erklärt einige Grundlagen des Datenbanksystems PostgreSQL. Dessen Quellcode soll als Basis für die Implementierung des Konzepts dienen. Damit wird das notwendige Verständnis für das vierte Kapitel vermittelt.

Im vierten Kapitel wird der Entwurf präsentiert, der ausgearbeitet wurde, um PostgreSQL entsprechend dem Konzept zu erweitern.

Das fünfte Kapitel demonstriert die Implementierung des Entwurfs. Damit soll gezeigt werden, dass die Aufgabenstellung erfüllt wurde.

Im sechsten und letzten Kapitel folgt eine Zusammenfassung der Arbeit und ein kurzer Ausblick auf mögliche weiterführende Arbeiten.

2 Aufgabenstellung

2.1 Konzept

In diesem Abschnitt wird ein allgemeines Konzept erläutert, mit dem ein Datenbanksystem mandantenfähig gemacht werden kann. Das Konzept beschränkt sich dabei auf Datenbanksysteme, die über die Sprache SQL angesprochen werden.

2.1.1 Ansatz

Der Ansatz dieses Konzepts ist es, einen Großteil der Logik, die für einen Mandanten notwendig ist, schon auf der untersten Ebene, der Datenbank, abzubilden. D.h. das Datenbanksystem ist sich der Einheit Mandant bewusst, und dadurch können Vorteile ausgenutzt werden, die im Folgenden näher erläutert werden:

- Eine vom Administrator angelegte Definition einer Tabelle kann von allen Mandanten genutzt werden, die in der Tabelle von einem Mandanten eingefügten Daten können jedoch trotzdem gegenüber den anderen Mandanten isoliert werden, damit sie für andere Mandanten nicht sichtbar sind.
- Es können Statistiken über gespeicherte Daten und über ausgeführte Anfragen pro Mandant angefertigt werden.
- Die zur Ausführung einer Anfrage angereicherten Informationen können von allen Mandanten benutzt werden, die dieselbe Anfrage an das Datenbanksystem übermitteln.
- Im Gegensatz zum Ansatz, ein Shared Schema zu verwenden, kann jedem Mandant der direkte Datenbankzugriff gewährt werden.
- Mandantenweise Operationen können beschleunigt werden.
- Einmalige größere Anpassung des Datenbanksystems ist notwendig, dafür sind weniger Anpassungen an den Anwendungen zu machen.

Den genannten Vorteilen stehen folgende Nachteile gegenüber:

- Die vollständige Implementierung ist aufwändig und erfordert an einigen Stellen schwerwiegende Änderungen.
- Die zusätzliche für Mandanten notwendige Logik erhöht die Komplexität bzgl. des Quellcodes und der Ausführungsgeschwindigkeit. Bei letzterem ist bisher unklar, ob diese durch die bei den Vorteilen genannten Punkten ausgeglichen werden kann.

2.1.2 Eingliederung der Mandanten in bisheriges Konzept

Um die im Ansatz erwähnten Vorteile ausnützen zu können, wird zunächst eine neue Einheit innerhalb der Datenbank benötigt, der Mandant. Ein Mandant darf nicht mit einem Benutzer verwechselt werden, da diese konzeptionell vollständig verschiedene Einheiten bilden. Beispielsweise ist es konzeptionell so, dass ein Mandant mehrere Benutzer haben kann, aber ein Benutzer nur zu einem Mandanten gehört. Die Mandanten werden in einem Katalog der Datenbank hinterlegt, bekommen einen global eindeutigen Namen und können zur Laufzeit je Datenbankverbindung gesetzt werden. Über den Namen hinaus sind natürlich weitere Attribute, wie beispielsweise die Adresse, die Telefonnummer oder die Kontaktdaten eines technischen Ansprechpartners des Mandanten zur Modellierung des Mandanten denkbar, die für diese Arbeit jedoch keine Rolle spielen.

Eine Funktion des Mandanten ist es, für ihn ein Datenbankschema herzuleiten, das alle vom Administrator erstellten Definitionen enthält, sowie eigene vom Mandanten durchgeführte Anpassungen. Das vom Administrator erstellte Schema wird nachfolgend als Basisschema und das hergeleitete als Mandantenschema bezeichnet. Eine der vielen denkbaren Anpassungen wäre eine eigene Tabelle, die nur der Mandant sieht.

Die Herleitung eines Mandantenschemas ist in Abbildung 2.1 zu sehen. Im Basisschema wurde eine Tabelle `articles` angelegt. Diese Tabelle kann keine Daten enthalten, daher ist der untere Teil ausgegraut. Die Pfeile vom Basisschema zu den Mandantenschemata deuten an, dass die Tabelle auch im jeweiligen Mandantenschema sichtbar ist. Im Schema des ersten Mandanten enthält die Tabelle zusätzlich ein vom Mandant angelegtes Attribut `Tax`. Dem Schema des zweiten Mandanten wurde eine Tabelle `warehousing` hinzugefügt.

Darüber hinaus soll es möglich sein, mehrere Mandanten zu einer Gruppe zusammenzufassen. Dabei soll ein Mandant maximal einer Gruppe angehören können, da der Administrator für jede Gruppe Einstellungen vergeben können soll, die auf deren Mandanten angewandt werden sollen, und somit gewährleistet ist, dass keine Konflikte durch eine widersprüchliche Einstellung zweier Gruppen möglich sind. Beispielsweise sind Einstellungen für eine Gruppe Premium denkbar, die einen Geschwindkeitsvorteil gegenüber einer Gruppe Basic bewirken.

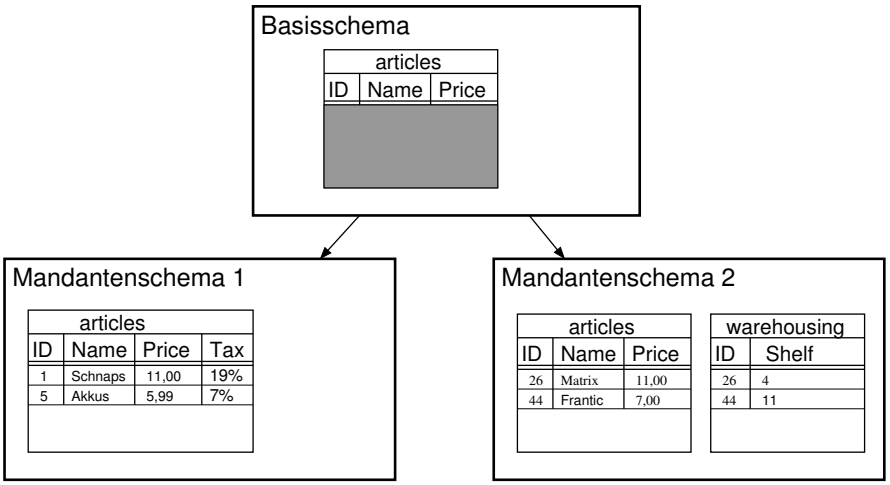

Abbildung 2.1: Mandantenschema aus Basisschema herleiten

2.1.3 Gemeinsame Tabellendefinition und eigene Dateien

Wie schon bei den Vorteilen des Ansatzes erwähnt wurde, bietet sich die Möglichkeit an, den Administrator eine Tabelle für alle Mandanten im Basisschema anlegen lassen zu können, bei der jeweils nur die eigenen Tupel des Mandanten sichtbar sind. Bei den bisherigen unangepassten Datenbanksystemen müsste man dazu für jeden Mandant eine eigene Datenbank einrichten und dieselbe Tabelle in jeder Datenbank anlegen. Dagegen ermöglicht das vorgestellte Konzept eine platzsparende Darstellung und korrekte Modellierung, da hierbei der gemeinsame Ursprung der Definition erhalten bleibt. Bezeichnet man nun mit einer Instanz einer Tabelle den eigentlichen Inhalt, so muss eine derartige vom Administrator erstellte Tabelle für jeden Mandant eine eigene Instanz besitzen können. In Datenbanken wird der Inhalt einer Tabelle meist in einem eigenen Speicherbereich abgespeichert. Daher wird die Isolierung der Daten dadurch erreicht, dass jeder Instanz einer Tabelle ein eigener Speicherbereich zugewiesen wird. D.h. je isolierter Tabelle und Mandant wird ein eigener Speicherbereich reserviert, in der die Instanz abgespeichert wird.

In diesem Zusammenhang ergibt sich die Möglichkeit, eine weitere der im vorherigen Abschnitt erwähnten Anpassungen zuzulassen, die ein Mandant am Schema vornehmen kann. So ist es bei diesem Typ von Tabellen möglich, dass ein Mandant ein eigenes Attribut hinzufügt. Dabei spielt es durch die Trennung der Tupel keine Rolle, dass die Tupel der anderen Mandanten nicht zu dem Mandantenschema mit dem zusätzlichen Attribut passen.

2.1.4 Gemeinsame Indexdateien

Bei einem Index für eine isolierte Tabelle stellt sich wie bei den Daten der isolierten Tabelle trotz gemeinsamer Nutzung der Definition die Frage, wie die Daten des Index eines Mandants gegenüber den anderen Mandanten isoliert werden können. Anstatt wieder den Inhalt je Mandant in eine eigene Indexdatei abzuspeichern, sollen mehrere Mandanten die gleiche Indexdatei benutzen können. Die Idee, die dahinter steckt, ist die wahrscheinlich steigende Auslastung der B-Baum-Knoten und damit einer höheren Trefferquote im DB-Puffer. Mit anderen Worten, die Wahrscheinlichkeit, dass weniger Seiten vom externen Speicher nachgeladen werden müssen, steigt. Im später folgenden Entwurf muss es also einen Mechanismus geben, der aus den gelesenen Daten der gemeinsamen Indexdatei die des aktuellen Mandants herausfiltert.

Dennoch soll der Administrator die Möglichkeit haben, für Mandantengruppen und für einzelne Mandanten eine eigene Indexdatei zu erstellen. Damit hat der Administrator beispielsweise die Möglichkeit nach zeitlichen Zugriffsmustern zu gruppieren und diesen Gruppen jeweils eigene Indexdateien zuzuweisen. In einem Extremfall greift eine Gruppe nur zwischen 4 und 10 Uhr zu und eine zwischen 18 und 22 Uhr. Dann wird zum Einen durch die Trennung in zwei Indexdateien die Anzahl der Seiten kleiner und dadurch passt ein größerer Teil des B-Baums in den Cache als des B-Baums, der die Indexdaten beider Gruppen enthalten würde, und zum Anderen werden die Seiten während ihrer Nutzung nicht von denen der anderen Indexdatei aus dem DB-Puffer verdrängt.

2.1.5 Tablespace je Gruppe von Mandanten

Meist kann man in einem Datenbanksystem mehrere sogenannte Tablespaces anlegen, die jeweils einen Speicherort, zum Beispiel ein Verzeichnis im Dateisystem, beschreiben, von dem man dann beim Anlegen einer Tabelle einen auswählen kann. Da eine isolierte Tabelle je Mandant eine eigene Datei besitzt und die Möglichkeit bestehen soll, dass für einen Index einer isolierten Tabelle eine eigene Indexdatei je Mandantengruppe zugewiesen werden kann, bietet es sich an, für diese Dateien unterschiedliche Tablespaces angeben lassen zu können. Für diesen Zweck sollen die Gruppen verwendet werden, so dass jeder Gruppe und damit allen zugehörigen Dateien ihrer Mandanten ein eigener Tablespace zugewiesen werden kann.

2.1.6 Gemeinsame Tabellen

Zusätzlich zu den isolierten Tabellen sind Tabellen vorgesehen, die von allen Mandanten gemeinsam genutzt werden. Bei so einer Tabelle wird sowohl die Definition als auch der Inhalt geteilt. Ein Mandant hat darauf allerdings keinen Schreibzugriff. Stattdessen wird

die Tabelle vom Administrator befüllt. Eine sinnvolle Anwendung könnte eine Tabelle für Postleitzahlen und Orte sein, die vom Dienstbereitsteller aktuell gehalten wird.

2.1.7 Grad der Virtualisierung

Der hier vorgestellte Ansatz kann als eine Art der Virtualisierung aufgefasst werden. Das wird vor allem durch das Konzept eines Basisschemas deutlich, das von jedem Mandanten angepasst werden kann, um ein eigenes gegenüber anderen Mandanten abgeschottetes Mandantenschema zu erzeugen. Darüber hinaus gibt es mit den isolierten Tabellen eine Möglichkeit den eigenen Inhalt einer Tabelle von dem der anderen Mandanten zu trennen. Dadurch können die Mandanten mithilfe dieser Mittel in einer bestehenden vom Administrator bereitgestellten Datenbank arbeiten, ohne die anderen Mandanten zu beeinflussen. Damit erfüllt das Konzept einen wesentlichen Aspekt, der für die Virtualisierung grundlegend ist. Würde man jedoch zu 100% virtualisieren wollen, so müssten alle Funktionen der Datenbank wie gewohnt zur Verfügung stehen. Das ist nicht Ziel des Konzepts. So soll es zum Beispiel nicht erlaubt sein soll, vom Administrator definierte Attribute einer isolierten Tabelle oder gar eine ganze Tabelle zu löschen. Damit soll verhindert werden, dass ein Mandant für eine vom Administrator für alle Mandanten aufgesetzte Anwendung wichtige Attribute bzw. Tabellen löschen oder durch andere ersetzen kann und somit bezüglich der Anwendung undefinierte Zustände möglich wären. Zudem können vom Administrator weiterhin normale Tabellen angelegt werden, die nicht virtualisiert werden. D.h. diese Tabellen können durch Mandanten nicht angepasst und deren Inhalt nicht isoliert werden.

2.1.8 Syntax

Im Folgenden sind einige für den Administrator gedachte Syntaxänderungen aufgelistet. Der Mandant hingegen benötigt keine. Er kann für seine Anpassungen die `ALTER`-Befehle verwenden.

```
CREATE TENANT name [ IN TENANTGROUP name ]
```

Dieser Befehl wird benötigt, damit datenbankweite Mandanten angelegt werden können. Außerdem ist es mit dem optionalen Teil möglich eine Gruppe von Mandanten anzugeben, zu der der Mandant hinzugefügt werden soll.

```
DROP TENANT name
```

Hiermit lässt sich ein zuvor mittels `CREATE TENANT name` angelegter Mandant löschen.

```
ALTER TENANT name SET TENANTGROUP name
```

Mit diesem Befehl kann man einem bestehenden Mandanten eine neue Gruppe zuweisen.

```
SET TENANT name
```

Mithilfe dieser Anweisung kann man der aktuellen Datenbankverbindung einen Mandanten zuweisen, ähnlich wie man ihr mithilfe der geläufigen Anweisung `SET ROLE name` einen Benutzer zuweist.

```
UNSET TENANT
```

Hiermit kann der durch `SET TENANT` gesetzte Mandant der Datenbankverbindung zurückgesetzt werden, nach der Ausführung dieses Befehls werden also alle nachfolgende Anfragen wieder ohne einen gesetzten Mandanten ausgeführt.

```
CREATE TABLE name '(' OptTableElementList ')' SEGREGATED BETWEEN TENANTS
```

Legt eine isolierte Tabelle an, die für jeden Mandant sichtbar ist und auf die jeder Mandant Schreibzugriff hat. Es wird garantiert, dass beim Lesen aus der Tabelle nur die zuvor vom Mandanten in der Tabelle abgespeicherten Tupel sichtbar sind. Damit teilen sich alle Mandanten das logische Schema der Tabelle, deren Inhalt wird jedoch gegenüber den anderen Mandanten isoliert, was durch das Schlüsselwort `SEGREGATED` angedeutet wird.

```
TABLESPACE name [ FOR '(' name, ... ')' ], name [ FOR '(' name, ... ')' ],
...
```

Das Schlüsselwort `TABLESPACE` wird beim Anlegen von Tabellen und Indizes verwendet, um den Tablespace anzugeben. Zum Anlegen von isolierten Tabellen und gemeinsamen Indizes wird der bisherige durch diesen neuen Syntax ersetzt. Der Name vor dem optionalen Schlüsselwort `FOR` benennt einen Tablespace und über die Namen in den Klammern können Mandanten bzw. Mandantengruppen angegeben werden, für die der vorstehende Tablespace verwendet werden soll.

```
CREATE TABLE name '(' OptTableElementList ')' SHARED BETWEEN TENANTS
```

Eine so angelegte Tabelle soll ebenfalls für jeden Mandant sichtbar sein, jedoch soll kein Mandant Schreibzugriff darauf erhalten. Stattdessen sollen beim Lesen alle vom Administrator eingefügten Tupel sichtbar sein, daher die Verwendung des Schlüsselworts `SHARED` im Syntax.

2.2 Umsetzung

Das Ziel der Arbeit, in dessen Rahmen diese Ausarbeitung verfasst wurde, ist die Umsetzung des in den vorherigen Abschnitten erläuterten Konzepts durch Ausarbeitung eines Entwurfs der notwendigen Änderungen an dem relationalen Datenbanksystem PostgreSQL und die anschließende Implementierung. Damit dient sie als Beweis dafür, dass das Konzept umgesetzt werden kann, sie stellt also den sogenannten proof-of-concept dar. Es können

jedoch mangels Zeit nicht alle wünschenswerten Funktionen umgesetzt werden, weshalb es sich bei der Implementierung dieser Arbeit um einen Prototyp handelt, der jedoch als Plattform überhaupt erst die Anpassung weiterer Teile des Datenbanksystems an mehrere Mandanten ermöglicht.

3 PostgreSQL

3.1 Warum PostgreSQL?

Es gibt viele relationale Datenbanksysteme, worunter IBM DB2, Microsoft SQL Server, MySQL, Oracle Database und PostgreSQL zu den bekanntesten zählen. Da es sich bei dem hier behandelten Konzept um eines handelt, zu dessen Umsetzung sehr tiefgreifende Änderungen notwendig sind, die durch ein Add-On, welche man bei den meisten Datenbanksystemen zur Laufzeit einbinden kann, nicht vorgenommen werden können, scheiden die proprietären Varianten aus. Es bleiben lediglich PostgreSQL und MySQL übrig, die beide OpenSource sind und dadurch die notwendigen Änderungen direkt am bestehenden Quellcode vorgenommen werden können.

MySQL wird nach den Angaben in [8] unter diesem Namen schon seit 1994 entwickelt, startete jedoch als Klon des älteren mSQLs. Das Datenbanksystem war ursprünglich für die Benutzung zusammen mit einer Webanwendung gedacht. Seitdem wurde MySQL stetig weiterentwickelt und hat mittlerweile einen sehr hohen Bekanntheitsgrad erreicht. Letzteres hängt sicherlich damit zusammen, dass bei vielen Hosting-Angeboten zur Bereitstellung von Webanwendungen standardmäßig ein MySQL-Server installiert ist.

Darüber hinaus werden von MySQL seit der aktuellen Version 5.x alle wichtigen vom Standard SQL-3 definierten Funktionen wie Stored Procedures, User Defined Functions, Triggers und Views implementiert und eignet sich daher bzgl. des Funktionsumfangs längst auch für größere Projekte.

Im Gegensatz zu MySQL handelt es sich bei PostgreSQL um ein relationales Datenbanksystem, das die vom SQL-Standard definierten objektrelationalen Erweiterungen implementiert und somit das Datenmanagement von Objekten im Sinne der Objektorientierung übernehmen kann. Daher werden einige andere Begriffe verwendet, wie bspw. Objekt statt Entität, Klasse statt Entitätstyp und Namespace statt Schema. Ein System, das diese Erweiterungen implementiert, wird häufig als objektrelationales Datenbanksystem bezeichnet.

Mit der Entwicklung von PostgreSQL wurde laut [9] 1997 begonnen, ebenfalls auf Basis einer wesentlich älteren Datenbank, und wird seitdem immer noch sehr aktiv weiterentwickelt. Die Community selbst, die für die Entwicklung zuständig ist, bezeichnet PostgreSQL in [7] als sogenannte Enterprise-Datenbank, die auch in großen produktiv genutzten Um-

gebungen eingesetzt wird. Zudem wird im Wesentlichen der Standard ANSI-SQL:2008 unterstützt.

Da beide Datenbanksysteme sich von dem Funktionsumfang nicht viel schenken und bei beiden keine Schwächen bzgl. Stabilität oder Performanz bekannt sind, wurde PostgreSQL auf Grund des Quellcodes und der Dokumentation gewählt. Der Quellcode ist gut lesbar und fördert daher ein schnelles Verständnis, insbesondere durch teilweise längere Kommentare an wichtigen Stellen. Außerdem gibt es zu komplizierteren Modulen jeweils ein technisch fundiertes Dokument, das die zugrundeliegende Ideen sehr ausführlich erklärt.

Ein weiterer Grund ist die Tatsache, dass die Architektur von PostgreSQL einem allgemeinen Modell eines Datenbanksystems, dem ANSI-Schichten-Modell, folgt, wodurch es wahrscheinlich ist, dass der im späteren Verlauf der Arbeit vorgestellte Entwurf auch auf andere Datenbanksysteme angewandt werden kann, die diesem Modell folgen.

3.2 Übersicht über PostgreSQLs Aufbau

PostgreSQL ist ausschließlich in der Sprache C geschrieben und wurde für unzählige Plattformen portiert, darunter Windows und die meisten unixbasierten Betriebsysteme mit den bekanntesten CPUs. Die Entwicklung erfolgt hauptsächlich durch eine Community, die sich über eine Mailingliste öffentlich austauscht und jeden am Entwicklungsprozess teilhaben lässt. Dort können Änderungen vorgeschlagen werden, die dann diskutiert und gegebenenfalls als sogenannte Patches mittels der Versionskontrollsoftware Git in den Quellcode eingepflegt werden. Die Lizenzierung des Quellcodes erfolgt über die eigene Lizenz namens The PostgreSQL License (Abk. TPL), die aber kaum Einschränkungen macht und PostgreSQL daher als freie Software bezeichnet werden kann.

Wie es für Datenbanksysteme, die für den Mehrbenutzerbetrieb gedacht sind, typisch ist, folgt die Kommunikation zwischen dem Benutzer und PostgreSQL einem Client-Server-Modell. Dazu wurde eine eigene Schnittstelle zwischen Client und Server entworfen, für die ein eigener Kommandozeilenclient und ein grafischer Client namens pgAdmin entwickelt wurden. Auf der Seite des Servers wird der Pool, der die Anfragen der Clients bearbeitet, durch mehrere in diesem Fall Backend genannte Prozesse realisiert. Dazu wird für jede Verbindung ein Backend gestartet, das die Anfrage vom Client sequenziell abarbeitet, insbesondere also selbst keine Threads benutzt. Die für den Mehrbenutzerbetrieb notwendige Synchronisation und Interprozesskommunikation zwischen den Prozessen wird über einen gemeinsamen Speicherbereich, dem Shared Memory, erreicht. Der Shared Memory enthält außerdem den DB-Puffer, auch Bufferpool genannt.

Bei der Entwicklung von PostgreSQL wurde viel Wert auf die Erweiterbarkeit gelegt. So kann man problemlos, ohne Prozeduren im Quellcode anpassen zu müssen, eigene Indexstrukturen oder eine neue Speicherverwaltung hinzufügen. Das funktioniert insbesondere deshalb, weil bei PostgreSQL möglichst viel über die Datenbank selbst modelliert wur-

de. Zum Beispiel wurden Informationen über einige interne Funktionen des Quellcodes in der Datenbank hinterlegt, so dass diese gegenüber der SQL-Schnittstelle sichtbar sind und daher vom Benutzer mittels SQL überschrieben werden können. Darüber hinaus lassen sich eigenständige Datentypen modellieren, indem man alle notwendigen Operatoren zur Verfügung stellt. Die Operatoren können hierbei durch C Funktionen zur Laufzeit geladen werden. Und falls man dann doch den Quellcode selbst ändern muss, kommt einem sehr entgegen, dass viele Mechanismen so generisch wie möglich gelöst wurden und dementsprechend leicht zu neuen Zwecken genutzt werden können.

3.3 Prozessarchitektur von PostgreSQL

In Abbildung 3.1 ist das Schema dargestellt, mit dem PostgreSQL mehrere Anfragen verschiedener Clients parallel abarbeitet. Zum Einen gibt es hierzu einen Postmaster, der auf allen Kanälen lauscht, über die sich Clients mit dem Server verbinden können sollen und der den gemeinsam genutzten Speicherbereich initialisiert. Und zum Anderen gibt es die vom Postmaster gestarteten Backends, die eine eingehende Verbindung vom Postmaster zugewiesen bekommen, so dass sie die darüber vom Client empfangenen Anfragen ausführen können.

Der eigentliche arbeitende Prozess ist daher das Backend und hat dementsprechend die meisten Aufgaben. Er ist unter Anderem für das Parsen und das Ausführen der mittels SQL formulierten Anfragen zuständig. Nebenbei muss er sich über den gemeinsamen Speicherbereich mit den anderen Backends koordinieren, da alle Backends auf den gleichen Bufferpool und einige andere gemeinsame Mechanismen zugreifen, sowie gleichzeitig auf einer Relation arbeiten können müssen.

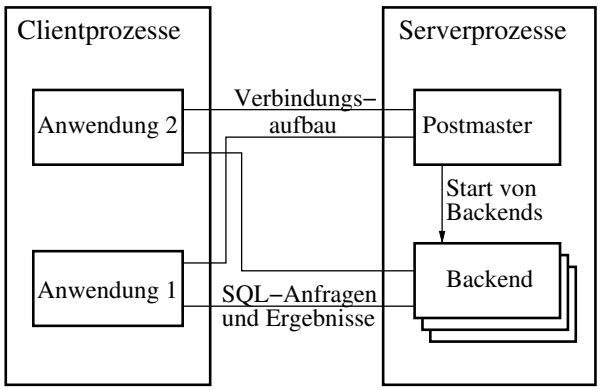

Abbildung 3.1: Serveraufbau zur gleichzeitigen Abarbeitung mehrerer Anfragen, aus [4]

3.4 Aufbau eines Backends

Die Backends sind relativ komplexe Prozesse und bestehen daher aus einer Vielzahl von Modulen. Diese können zum besseren Verständnis auf Grund ihrer Aufgabe bzw. Verwendung zu unterschiedlichen Gruppen aufgegliedert werden. An dieser Stelle sollen die in der Abbildung 3.2 gezeigten Module erklärt werden, die für jede eingehende Anfrage nacheinander aufgerufen werden, um ihrerseits etwas dazu beizutragen, die Anfrage vollends zu bearbeiten.

Parser ist dafür zuständig aus der mittels SQL formulierten Anfrage einen bestimmten zum Typ der Anfrage passenden Syntaxbaum zu erstellen, der alle in der Anfrage angegebenen Informationen enthält.

Traffic Cop überprüft, ob es sich um eine Anfrage handelt, deren Abarbeitung immer auf gleiche Art und Weise erfolgt, und gibt diese in diesem Fall an das nachfolgende Modul Utility weiter.

Utility ruft zur Abarbeitung einer Anfrage direkt die jeweilige dafür zuständige Funktion auf, die im Prinzip einen festverdrahteten Plan ausführt.

Optimizer führt eine Suche aus, um den optimalen Pfad bezüglich eines Kostenmodells zu ermitteln, aus dem sich der optimale Plan erstellen lässt. Ein Plan ist hierbei ein etwas feinerer Baum als der aus Knoten der Relationenalgebra bestehende.

Planer baut den Plan nach den Angaben des optimalen Pfads auf.

Executor führt den Plan aus.

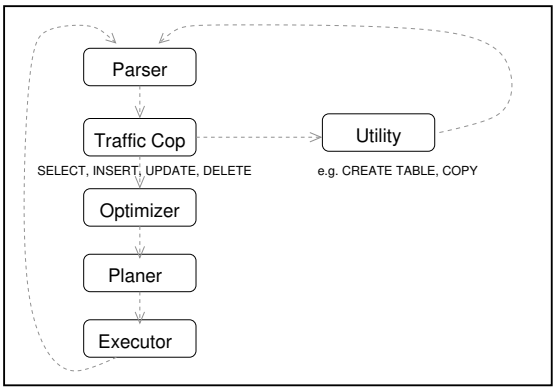

Abbildung 3.2: Flussdiagramm der Anfragebearbeitung durch ein Backend, aus [5]

3.5 Darstellung eines Tupels

Die wesentliche Aufgabe eines relationen Datenbanksystems ist die Verwaltung von Tupeln. Dazu muss es die Tupel abspeichern und bearbeiten können. Das Abspeichern der Attribute eines Tupels erfolgt bei PostgreSQL in einem zusammenhängenden Bereich des externen Speichers. Vor den Attributen steht ein Header, der beispielweise angibt, welche Attribute des Tupels `NULL` sind und daher kein Platz auf dem externen Speicher dafür reserviert wurde.

Damit das Datenbanksystem auf ein in diesem Format vom externen Speicher geladenen Tupel zugreifen kann, wird eine Datenstruktur benötigt. Zu dessen Definition wird ein von der Sprache C bereitgestelltes Konstrukt namens Struct verwendet, um mehrere Variablen zusammenzufassen. Die Größe eines für eine Variable notwendigen Speicherplatz muss fest sein. Die Variablen werden auch häufig als Felder bezeichnet. Durch so ein Struct wird die in PostgreSQL für Tupel verwendete Datenstruktur mit dem Bezeichner `HeapTuple` definiert, die alle Werte des Headers enthält, die bzgl. des relativen Offsets immer die gleiche Position haben. Dadurch kann auf die Werte des Headers direkt über die Felder des Structs zugegriffen werden. Da die einzelnen Attribute des Tupels unterschiedlich viel Speicherplatz benötigen, ist die Struktur des dahinterliegenden Teils variabel, und daher wird über ein Feld der Datenstruktur `HeapTuple` auf den kompletten Bereich zugegriffen.

Offensichtlich muss es einen weiteren Mechanismus geben, der die einzelnen Attribute aus dem soeben erwähnten Feld des Structs `HeapTuple` extrahiert. Da sich die Daten in dem Feld noch in dem Format des externen Speichers befinden, wird dazu das physikalische Schema benötigt, für das im nachfolgenden Abschnitt die entsprechende zur Laufzeit genutzte Datenstruktur vorgestellt wird.

3.6 Tupel Deskriptor

Ein Datenbanksystem muss naturgemäß generisch gestaltet werden, um an die Anwendung angepasst zu werden. Das wird erreicht, indem die grundlegenden Datentypen und die zu einem Tupel zusammengesetzten und in der Regel zu einer Tabelle gehörenden Datentypen mittels Metainformationen beschrieben werden. Diese müssen zum Einen die Sicht des Anwenders auf die Daten enthalten. Dieser Teil wird als logisches Schema bezeichnet. Und zum Anderen müssen sie beschreiben, wie die Daten auf dem physikalischen Speicher abgelegt werden. Diese Informationen werden entsprechend als physikalisches Schema bezeichnet.

Die genannten Metainformationen werden an sehr vielen Stellen im Quellcode benötigt. Dazu müssen sie von Modul zu Modul weitergegeben werden können. Zu diesem Zweck gibt es eine Datenstruktur namens `TupleDesc`, die aus den Metainformationen aufgebaut

wird. Dadurch genügt ein Parameter, um alle Informationen weiterzugeben, und der teure Aufbau muss von den nachfolgenden Modulen nicht nochmals durchgeführt werden.

Ein typischer Anwendungsfall, für den ein Modul den Deskriptor benötigt, ist das Auslesen eines einzelnen Attributs aus dem für Attribute in dem `HeapTuple`-Struct vorgesehenen Feld. In diesem Zusammenhang übernimmt der Deskriptor eine weitere wichtige Aufgabe. Das Offset bei dem ein Attribut anfängt, hängt von der Größe aller davor stehender Attribute ab. Falls diese Attribute alle vorhanden sind, also im Header des `HeapTuple`-Struct nicht mit `NULL` markiert worden sein, und alle eine feste Größe haben, zum Beispiel falls es sich um den ein Byte großen Typ char handelt, dann ist das Offset des Attributs für alle so gearteten Tupel der Tabelle gleich. Es reicht daher aus, einmalig dieses Offset für jedes Attribut im Deskriptor zu berechnen, anstatt das Offset für jedes Tupel erneut berechnen zu müssen.

3.7 Index

Um das Auslesen von Tupeln aus einer Tabelle zu realisieren, hat der Planer wenigstens zwei Möglichkeiten. Zum Einen kann er einen sogenannten HeapScan verwenden, bei dem der komplette für diese Tabelle zuständige externe Speicherbereich gelesen wird. Sollen die Tupel beim Auslesen jedoch selektiert werden, d.h. existiert ein Prädikat, das die Tupel erfüllen müssen, dann müssen nur die Seiten des externen Speichers gelesen werden, die ein selektiertes Tupel enthalten. Daher kann der Planer zum Anderen einen IndexScan durchführen, damit mithilfe eines passenden Index bestimmt wird, welche Seiten ein selektiertes Tupel enthalten, und diese dann geladen werden.

Dazu muss ein Benutzer für eine Tabelle im Vorfeld einen Index anlegen. Er muss dabei angeben, welche Attribute in den Index aufgenommen werden sollen, und welche Art der Suche verwendet werden soll. Als Suchmethode bietet PostgreSQL standardmäßig einen B-Baum und eine Hashtabelle. Der Index bildet jedoch eine eigene abgeschlossene Einheit, die über Schnittstellen angesprochen wird, die unabhängig von der verwendeten Suche sind und dementsprechend einfach kann PostgreSQL um einen neuen Typ von Index erweitert werden.

Damit der Index ein Tupel indizieren kann, wird beim Abspeichern eines `HeapTuple`-Structs ein dem `HeapTuple`-Struct sehr ähnliches `IndexTuple`-Struct aufgebaut und dem Index übergeben. Für den Aufbau werden im Normalfall alle Attribute, die indiziert werden sollen, aus dem eingefügten Tupel in das `IndexTuple`-Struct kopiert. Wie beim `HeapTuple`-Struct wird ein Tupel Deskriptor benötigt, weswegen sich die Modellierung der für einen Index benötigten Metainformationen kaum von denen einer normalen Tabelle unterscheidet. Zusätzlich zu diesem Struct erhält der Index eine ID des Tupels (Abk. TID), die auf den Speicherbereich des externen Speicherbereichs, in dem das Tupel abgespeichert wurde, und damit auch auf die zu ladende Seite verweist.

Bei einer lesenden Anfrage, die ein Prädikat enthält, dessen Attribute inidiziert wurden, kann der Index die TIDs aller Tupel benennen, die das Prädikat erfüllen. Dazu muss man dem Index ein Prädikat, etwa $Gehalt > 60.000$, übergeben können. Die Angaben eines Prädikats, die im Wesentlichen aus einem Attribut, einem Operator und einem Konstante bestehen, werden in einem Struct namens `ScanKey` verpackt. Es können mehrere `ScanKey`-Structs übergeben werden, wobei die Prädikate alle mit dem logischen UND verknüpft werden.

3.8 Relationen Deskriptor

Bei PostgreSQL bezeichnet eine Relation ein Objekt, das sich insbesondere durch die dazugehörenden Attribute definiert. Darunter fallen natürlich Tabellen, sowie alle Views und, durch die Attribute, die für das `IndexTuple`-Struct benötigt werden, auch alle Indizes.

Der Relationen Deskriptor ist die wohl am häufigsten verwendete Datenstruktur in PostgreSQL, abgesehen natürlich von den simplen Datentypen. Sie wird über ein Struct definiert, das alle Informationen vereint, die zur Laufzeit für eine Relation relevant sind. Dazu gehört natürlich der Tupel Deskriptor. Außerdem ein Deskriptor, der auf den für die Relation reservierten Speicherbereich des externen Speichers verweist, und sofern es sich um eine Tabelle handelt, Informationen über alle Indizes, die auf dieser Tabelle anlegt wurden. Das Struct wird von jedem Backend selbst aufgebaut, also nicht im gemeinsamen Speicher abgelegt, damit keine Locks beim Zugriff darauf benötigt werden.

Da der Aufbau des Relationen Deskriptors sehr teuer ist und er so oft verwendet wird, gibt es einen Cache, der alle Deskriptoren verwaltet. Wird von einem Modul ein Relationen Deskriptor benötigt, ruft es eine Funktion vom Cache auf, die in einer Hashtabelle nachschlagen und dem Modul gegebenenfalls einen Zeiger auf die gefundene Instanz zurückgeben kann. Falls keine Instanz gefunden wurde, wird der Deskriptor aufgebaut, die Instanz in der Hashtabelle abgelegt und der Zeiger darauf zurückgegeben.

Falls zwischendurch Änderungen an der Relation vorgenommen werden, wie beispielsweise das Löschen eines Attributs, dann werden die Metainformationen aktualisiert und daher wird der daraus aufgebaute Relationen Deskriptor als invalide markiert. Dieser muss von jedem Backend vor der Ausführung der nächsten Anfrage abgebaut und neu aufgebaut werden. Damit der Cache davon Kenntnis erlangt, gibt es einen Mechanismus, der gestartet wird, sobald es neue Informationen über eine Relation gibt, die für den Aufbau eines Relationen Deskriptors verwendet werden müssen, also zum Beispiel Informationen über ein hinzugefügtes Attribut. Dieser Mechanismus merkt sich die dazugehörige Relation und führt die Invalidierung des Relationen Deskriptors vor der Ausführung der nächsten Anfrage aus.

3.9 Systemkataloge

Ein Tupel Deskriptor sowie ein Relationen Deskriptor wird von jedem Backend in einem privaten Teil des Arbeitsspeichers aufgebaut, sobald sie den Deskriptor benötigen, um eine Operation auszuführen. Die für den Aufbau notwendigen Metainformationen müssen daher dauerhaft abgespeichert werden und alle Backends müssen darauf zugreifen können. Für diese Aufgabe sind die Systemkataloge zuständig.

Ein Systemkatalog ist in PostgreSQL eine vordefinierte Tabelle, die in einem speziell dafür vorgesehenen Schema angelegt worden ist. Die darin enthaltenen Daten können vom Datenbankbenutzer direkt ausgelesen werden, um beispielsweise festzustellen, welche Tabellen angelegt worden sind, und welche Spalten eine bestimmte Tabelle besitzt. Ein besonderes Merkmal von PostgreSQL ist, dass die Systemkataloge wesentlich mehr Daten enthalten, als das Konzept des klassischen Datenbanksystems vorschreibt.

Jede Ausführung eines Befehls ist mit einer Vielzahl von Anfragen an den Systemkatalog verbunden. Unter Anderem kommt es häufiger vor, dass ein und derselbe Eintrag bei der Ausführung eines Befehls mehrmals nachgeschlagen werden muss, da der Eintrag nicht im Quellcode weitergegeben wird. Deshalb ist es für die Effizienz entscheidend, dass diese Abfragen sehr schnell durchgeführt werden können. Aus diesem Grund gibt es für jeden Systemkatalog einen oder mehrere Caches. Die Suche nach einem Eintrag im Cache erfolgt über Attribute der Tabelle des Systemkatalogs, die einen Eintrag eindeutig identifizieren. Falls ein Eintrag nicht vorhanden ist, wird dieser nachgeladen und in einer Hashtabelle abgelegt. Der Quellcode, der die Caches implementiert, ist so generisch gehalten, dass die Angabe des Index, der beim Laden des Tupels verwendet werden soll, im Wesentlichen ausreicht, um einen neuen Cache zu definieren.

Es gibt eine häufig vorkommende Ausnahme, für die beim Zugriff auf die Attribute eines `HeapTuple`-Structs der zuvor erläuterte Tupel Deskriptor nicht notwendig ist. Das Schema eines Systemkatalogs kann nicht verändert werden und steht daher schon beim Kompilieren fest. D.h. Attribute, deren Datentypen und die Reihenfolge beim Abspeichern sind bekannt. Weiterhin wurden alle Attribute, dessen Werte eine feste Größe haben, was bei den Attributen der Systemkataloge meistens der Fall ist, bzgl. der Reihenfolge vorne eingereiht. Daher lässt sich das Format mit einem Struct nachbilden, wodurch über die Felder des Structs direkt auf die Attribute fester Größe zugegriffen werden können.

4 Entwurf

4.1 Systemkataloge

Als Speicherort von Metainformationen über die in der Datenbank angelegten Objekte, müssen die Systemkataloge angepasst werden, damit mandantenspezifische Änderungen möglich sind. Zum Einen muss repräsentiert werden können, zu welchem Mandanten die Metainformationen gehören, was häufig mit einer zusätzlichen Spalte für die ID des Mandanten gelöst werden kann, und zum Anderen ist es notwendig die Informationen so abzuspeichern, dass redundante Teile im Arbeitsspeicher zusammengefasst werden können.

Die Abbildung 4.1 gibt eine kurze Übersicht über die Systemkataloge, die angepasst werden müssen. Der Systemkatalog `pg_authid` enthält alle Rollen. Eine Rolle kann als Benutzer verstanden werden. In `pg_class` werden alle Relationen und in `pg_attribute` die zu einer Relation gehörenden Attribute abgelegt. Der Primärschlüssel von `pg_attribute` enthält eine Referenz auf `pg_class` und dadurch wird durch eine Referenz auf `pg_attribute` gleichzeitig `pg_class` referenziert. Der Systemkatalog `pg_attrdef` enthält den zu einem Attribut gehörenden `DEFAULT`-Wert und `pg_constraint` eine für ein Attribut zu prüfende Bedingung. In `pg_index` werden alle Indizes gespeichert. Ein Eintrag darin verweist zum Einen auf `pg_class`, da ein Index auch eine Relation ist. Und zum Anderen werden auf die Attribute der zu indizierenden Relation verwiesen, die in den Index aufgenommen werden sollen.

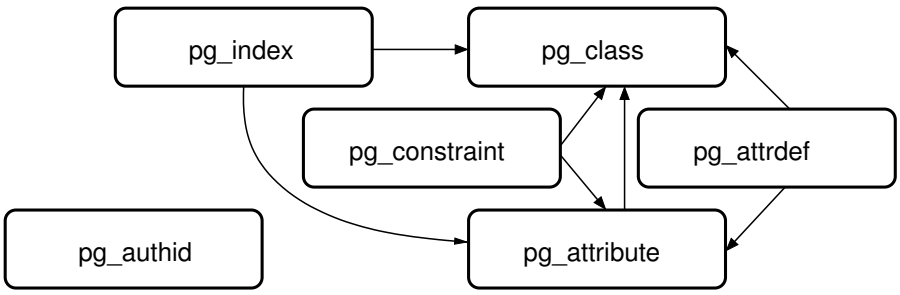

Abbildung 4.1: Übersicht über Systemkataloge

4.1.1 Informationen über Relationen in pg_class

Das Konzept sieht einige neue Typen von Relationen vor, die in die bisherige Modellierung der Systemkataloge eingebettet werden müssen. Zum Einen wird eine Relation benötigt, die lediglich benutzt werden soll, um zusammen mit anderswo abgespeicherten mandantenspezifischen Änderungen die eigentliche Relation des Mandanten abzuleiten. Der Eintrag für diese Relation darf der Mandant also gar nicht sehen, sondern nur die abgeleitete. Und zum Anderen muss es eine Relation geben, auf die Mandanten nur lesend zugreifen können. Da Mandanten konzeptuell keine Benutzer sind, können dafür keine ACLs verwendet werden. Es wird daher eine neue Spalte `relmtkind` benötigt, die den Typ der Relation abspeichert.

Über die Einträge in `pg_class` wird der Namensraum der Relationen eines Schemas verwaltet. Bisher sorgt ein Index über die Spalten `relname` und `relnamespace` dafür, dass kein Name doppelt in einem Schema vorkommen kann. Damit wären für jeden Mandanten die Namen von Relationen anderer Mandanten sichtbar und dementsprechend könnten zwei Mandanten nicht den gleichen Namen für eigene Relationen verwenden. Relationen, die ausschließlich der Mandant sehen soll, können beispielsweise notwendig sein, falls ein Mandant auf einer isolierten Tabelle einen eigenen Index anlegt, der ebenfalls einen Eintrag samt Namen in `pg_class` bekommen muss, aber für andere Mandanten nicht sichtbar sein darf. Es wird also eine neue Spalte `relmttenantid` benötigt, damit Einträge nur für einen Mandanten sichtbar sind und der Index muss auf diese Spalte ausgedehnt werden. Da weder ein eigener Index noch eine eigene Tabelle Teil der Aufgabenstellung waren, wird das Attribut im weiteren Verlauf des Entwurfs nicht mehr verwendet.

Durch die neue Spalte `relmttenantid` ergibt sich ein weiteres Problem. Es sind nun Einträge mit dem gleichen Namen denkbar, bei denen ein Mal kein Wert für die Spalte gesetzt ist und der Eintrag daher für alle sichtbar sein soll, und beim anderen Mal ein Wert gesetzt ist. Da der Index um die Spalte `relmttenantid` erweitert wurde, kann er diesen Fall nicht mehr verhindern. Da ein Mandant nicht die Möglichkeit haben soll eine Relation durch seine eigene zu überdecken, damit er die vom Administrator installierte und gewartete Anwendung nicht stören kann, muss dieser Fall beim Anlegen einer Relation im Quellcode explizit abgefangen werden.

4.1.2 Anpassungen eines Mandanten von pg_class in pg_mtclass

Zu den Änderungen, die der Mandant vornehmen können soll, gehören insbesondere Änderungen an isolierten Tabellen. Durch diese Änderungen werden einige der in `pg_class` gespeicherten Werte beeinflusst, wodurch es notwendig wird, dass diese Werte per Mandant gespeichert werden können. Beispielsweise erhöht sich aus Sicht des Mandanten der Wert für die Anzahl der Attribute `relnatts` um eins, falls der Mandant einer isolierten Tabelle eine eigene Spalte hinzufügt. Außerdem werden in `pg_class` Informationen über den in der Tabelle gespeicherten Inhalt abgelegt, wie beispielsweise die im Attribut `reltuples`

hinterlegte Anzahl der Tupel der Tabelle, die sich durch die isolierten Tabellen ebenfalls unterscheiden können.

Um diese Abweichungen abspeichern zu können, wird ein neuer Systemkatalog `pg_mtclass` angelegt, der eine Referenz namens `relrelid` auf den Eintrag in `pg_class` enthält, eine Spalte `relmttenantid` und alle Spalten aus `pg_class`, die sich durch Operationen auf der Tabelle ändern können müssen. Hierbei sei nochmals betont, dass es sich bei dem Attribut `relmttenantid` um keinen Fremdschlüssel auf das gleichnamige Attribut von `pg_class` handelt.

Eine weitere Spalte, die bei isolierten Tabellen abweichen können soll, ist das Attribut `relfilenode`, das auf die Datei verweist, in der die Daten gespeichert werden sollen. Diese Spalte muss darüber hinaus bei einem gemeinsamen Index abweichen können, damit die im Konzept vorgesehenen eigenen Indexdateien realisiert werden können, weshalb diese Abweichung ebenfalls in diesem Systemkatalog abgespeichert werden muss. Soll eine Mandantengruppe eine Indexdatei bekommen, wird ebenso ein Eintrag mit der ID der Mandantengruppe angelegt.

Da es sich hierbei um einen neuen Systemkatalog handelt, existiert dafür noch kein Cache. Als Primärschlüssel, der für die Suche im Cache verwendet werden kann, wird die Referenz auf die Relation und `relmttenantid` verwendet. Auf diesem wird ein Index angelegt und darauf basierend kann ein Cache namens `MTRELOID` definiert werden.

4.1.3 Informationen über Attribute in pg_attribute

Wie die Attribute von gemeinsamen Tabellen und die Attribute des Basisschemas der isolierten Tabellen, sollen auch die Attribute der Mandanten in diesem Systemkatalog abgespeichert werden. Dazu wird eine neue Spalte `attmttenantid` benötigt, um die auch hier die Indizes erweitert werden müssen.

Die Einträge in `pg_attribute` enthalten eine Attributnummer, wodurch die Attribute einer Relation zum Einen eindeutig identifizierbar sind und zum Anderen deren Reihenfolge dokumentiert wird. Bei einer isolierten Tabelle kann die Reihenfolge der Attribute des Basisschemas und die der Attribute des Mandanten jeweils anhand der Attributnummern rekonstruiert werden. Damit ein Mandant beide Reihenfolgen zur kompletten rekonstruieren kann, benötigt er für die selbst angelegten Attribute eine Spalte `attmtlatestbaseatt`, das angibt, dass das jeweils selbst angelegte Attribut zwischen dem Attribut des Basisschemas mit dieser Attributnummer und dem Attribut des Basisschemas mit der nächstgrößeren Attributnummer einzureihen ist.

4.1.4 Informationen über DEFAULT-Werte der Attribute

Der Mandant möchte beim Anlegen eines Attributs einer isolierten Tabelle möglicherweise einen `DEFAULT`-Wert angeben. Wie alle anderen soll der Wert in `pg_attrdef` abgespeichert werden. Damit angegeben werden kann, dass dieser Wert zu einem Attribut eines Mandanten gehört, muss der Fremdschlüssel, der bisher nur aus der Referenz auf die Relation und aus der Attributnummer bestand, entsprechend dem Primärschlüssel von `pg_attribute` um eine Spalte `admttenantid` für die ID des Mandanten ergänzt werden.

4.1.5 Informationen über zu prüfende Bedingungen in pg_constraint

Der Mandant kann beim Hinzufügen eines Attributs eine zu prüfende Bedingung angeben, die in diesem Systemkatalog abgespeichert werden muss. Im Gegensatz zu `pg_attrdef` gibt es zwar keinen Fremdschlüssel der an den Primärschlüssel von `pg_attribute` angepasst werden müsste, jedoch kann es einerseits wie bei `pg_class` zu Namenskonflikten kommen und andererseits muss vermerkt werden, dass die Bedingung nur bei diesem Mandanten geprüft werden darf, weshalb auch hier eine neue Spalte `conmttenantid` für die ID des Mandanten benötigt wird.

4.1.6 Informationen über Rollen in pg_authid

Obwohl es sich bei einem Mandanten um eine Einheit handelt, die konzeptionell unabhängig von einem Benutzer ist, soll ein Mandant vorerst als Rolle modelliert werden. Das hat den Vorteil, dass alle Operationen und Mechanismen für Rollen auch auf Mandanten anwendbar sind, was sich bei späteren Tests mit dem Prototyp als nützlich erweisen kann. Zwar wäre es andererseits naheliegend die Mandanten dazu einfach mit Benutzern über Referenzen zu verknüpfen, jedoch müsste man dann einen Aufwand betreiben, wie neue Kommandos und zusätzliche Anwendungslogik zu implementieren, durch den für den Prototyp kein echter Gewinn entsteht.

Damit die normalen Rollen, die Rollen, die einen Mandanten repräsentieren, und die Rollen der Mandantengruppen unterschieden werden können, wird eine Spalte `rolmtkind` benötigt. Für die letzteren beiden muss eine neue Spalte `rolmttenantid` auf einen eindeutigen Wert gesetzt sein, der der Identifizierung dient. Der Wert muss größer null sein, damit er im Folgenden als valide bezeichnet wird. Falls es nachfolgend heißt, dass er nicht gesetzt wurde oder invalide ist, dann ist damit der Wert null gemeint. Er wird häufig verwendet, um auszudrücken, dass etwas zum Basisschema gehört. Zur Identifizierung der Mandanten wird absichtlich nicht die ID der Rollen verwendet, damit die Implementierung der Mandanten nicht davon abhängt, dass die Mandanten in der Tabelle der Rollen gespeichert werden.

4.1.7 Informationen über Indizes in pg_index

Ein Mandant kann auf einer isolierten Tabelle einen eigenen Index anlegen. Der dadurch notwendige Eintrag in `pg_index` wird zwar schon durch den korrespondierenden Eintrag in `pg_class` dem entsprechenden Mandanten zugewiesen, da diese Information jedoch aus Effizienzgründen verfügbar sein soll, ohne dass man extra in `pg_class` nachschlagen muss, wird dem Systemkatalog `pg_index` eine eigene Spalte `indmttenantid` hinzugefügt und die Indizes auf dem Systemkatalog entsprechend erweitert.

4.2 Schnittstelle für pg_mtclass

Viele der Operationen auf Tabellen, zu deren Ausführung das Modul Utility jeweils die entsprechende Funktion aufruft, müssen nach getaner Arbeit den dazugehörenden Eintrag in `pg_class` aktualisieren. Diese Aktualisierung ist in den Funktionen fest verdrahtet. Ein typischer etwas vereinfachter Ausschnitt aus dem Quellcode einer Funktion kann man in dem Ausschnitt 4.1 sehen. Zuerst wird der Cache benutzt, um eine Kopie des `HeapTuple`-Structs zu bekommen. Dann wird ein simples Makro `GETSTRUCT` benutzt, um einen Zeiger auf den Bereich des Structs zu erhalten, der die Attribute des Tupels enthält. Das Attribut kann nun mithilfe des Struct `Form_pg_class` direkt aktualisiert werden. In der nächsten Zeile wird das aktualisierte `HeapTuple`-Struct über die Funktion `simple_heap_update` in die bereits geöffnete Relation `pg_class` geschrieben. Die neue TID befindet sich danach in `heapTuple->t_self`.

```
1 heapTuple = SearchSysCacheCopy(RELOID, relid, 0, 0, 0);
2 pg_class = (Form_pg_class *) GETSTRUCT(tup);
3 pg_class->relnatts++;
4 simple_heap_update(pg_class_relation, &heapTuple->t_self,
    heapTuple);
```

Ausschnitt 4.1: Aktualisierung eines Tupels der Tabelle `pg_class`

Offensichtlich müssen alle Funktionen, die auch für Mandanten aufgerufen werden können, angepasst werden, damit die Änderung gegebenenfalls in der Tabelle `pg_mtclass` statt in `pg_class` abgespeichert wird. Dazu sind zum Einen einige neue Variablen notwendig, wie beispielsweise eine für die Relation `pg_mtclass` und eine für das `HeapTuple`-Struct aus `pg_mtclass`. Und zum Anderen müssen Werte, die in dem Eintrag des Systemkatalogs `pg_mtclass` noch nicht vorhanden sind, stattdessen aus dem entsprechenden Attribut des Eintrags des Systemkatalogs `pg_class` gelesen werden. Würde man versuchen das alles direkt in den Funktionen umzusetzen, würde ihr Quellcode wesentlich länger und unübersichtlicher werden. Daher wird eine Schnittstelle benötigt, die selbstständig entscheidet, in welche der zwei Relationen die Aktualisierung gespeichert wird und aus welcher der jeweilige Wert gelesen wird, und die alle dazu notwendigen Variablen verbirgt.

Bei den Attributen der bisherigen Systemkataloge wird immer erwartet, dass alle gesetzt sind, es also kein Attribut gibt, das `NULL` ist, damit die Position eines Attributs im `HeapTuple`-Struct immer gleich ist und daher mithilfe eines C Structs direkt auf die Attribute zugegriffen werden kann. Bei den Attributen eines Eintrags aus `pg_mtclass` können jedoch einige noch nicht gesetzt sein, da der Mandant noch keine Operation ausgeführt hat, durch die sie überschrieben würden. Die Implementierung der Schnittstelle soll daher einen Tupel Deskriptor verwenden, um auf die Attribute zuzugreifen, wodurch die Position trotz fehlender vorheriger Attribute korrekt berechnet wird. Die Schnittstelle selbst soll unabhängig davon sein, damit eine spätere Implementierung denkbar ist, bei der keine Attribute `NULL` sein dürfen, sondern stattdessen vor dem ersten Schreiben für die fehlenden Attribute einfach die Werte aus `pg_class` genommen werden. Im folgenden Abschnitt werden die Datenstrukturen und Funktionen dieser Schnittstelle erläutert.

4.2.1 Datenstrukturen

Wie schon erwähnt wurde, werden einige neue Variablen benötigt. Außerdem muss auf einige schon vorhandene Variablen so häufig zugegriffen werden, dass sie bei jedem Aufruf der Schnittstelle angegeben werden müssten und die Schnittstelle daher unnötig kompliziert werden würde. Es ist daher sinnvoll ein Struct namens `ChangeableMTClass` einzuführen, dass diese Variablen zusammenfasst. Das Struct ist in 4.2 dargestellt. Es enthält im Wesentlichen beide `HeapTuple`-Structs, sowie Variablen für deren Attribute, die TID des Tupels aus `pg_mtclass` und die ID des Mandanten, für den die Änderung vorgenommen werden soll. Insbesondere enthält es die Variablen `is_null` und `values`, in die das Tupel aus `pg_mtclass` nach der Deserialisierung der Attribute des Tupels gespeichert wird.

```
1  typedef struct {
2   Datum values[Natts_pg_mtclass];/* attributes of
3                                     pg_mtclass heap tuple */
4   bool isnull[Natts_pg_mtclass]; /* is attribute null? */
5   Form_pg_class   rd_rel;        /* pg_class struct */
6   HeapTuple       reltup;        /* pg_class heap tuple */
7   HeapTuple       mtreltup;      /* pg_mtclass heap tuple */
8   bool            newtup;        /* is mtreltup new */
9   ItemPointerData mtreltupptr;   /* tid of pg_mtclass
10                                    heap tuple */
11  int16           tenantid;
12 } ChangeableMTClass;
```

Ausschnitt 4.2: Struct `ChangeableMTClass`

4.2.2 Funktionen

Die Funktionen der Schnittstelle sind in Ausschnitt 4.3 zu sehen. Bei den ersten beiden handelt es sich um Prototypen und für die Realisierung der darunterstehenden wurden unter Anderem aus Effizienzgründen Makros verwendet, für die die Definition nur angedeutet wurde.

```
1  extern ChangeableMTClass BeginMTClassChange(Oid relid,
2                                              int16 tenantid,
3                                              HeapTuple reltup);
4  extern EndMTClassChange(ChangeableMTClass *chmtclass,
5                          Relation pg_mtclass_relation,
6                          Relation pg_class_relation,
7                          MemoryContext cxt);
8
9  #define SetMTClassAttr(chmtclass, attr, value) \
10             ...
11 #define GetMTClassAttr(chmtclass, attr) \
12             ...
13 #define notequalMTClassAttrOrNotSet(chmtclass, attr, value) \
14             ...
```

Ausschnitt 4.3: Funktionen der Schnittstelle zu `pg_mtclass`

Ein Beispiel, das auf dem eingangs eingeführten basiert, für die Verwendung der Definitionen ist in Ausschnitt 4.4 zu sehen. Die Funktionen `BeginMTClassChange` und `EndMTClassChange` werden aufgerufen, um eine Änderungen zu beginnen bzw. zu beenden. Der Zugriff auf ein Attribut erfolgt nun mittels Getter- und Setter-Methoden. Der Ausschnitt soll deutlich machen, dass durch die im Folgenden detailierter erklärten Funktionen die Struktur des Quellcodes gleich bleibt, also insbesondere keine Unterscheidung mittels der Kontrollstruktur `if-then-else` erfolgen muss.

```
1 ChangeableMTClass chmtclass;
2 heapTuple = SearchSysCacheCopy(RELOID, relid, 0, 0, 0);
3 chmtclass = BeginMTClassChange(chmtclass, GetTenantId(),
    heapTuple);
4 SetMTClassAttr(chmtclass, relnatts, 1+GetMTClassAttr(chmtclass
    , relnatts));
5 EndMTClassChange(chmtclass, pg_mtclass_relation,
6                  pg_class_relation, cxt);
```

Ausschnitt 4.4: Beispiel für Nutzung der Schnittstelle zu pg_mtclass

4.2.2.1 BeginMTClassChange

Diese Funktion wird zu Beginn einer Änderung aufgerufen. Sie bekommt als Paramter die Relation, für die die Metainformationen geändert werden sollen, die ID des Mandanten und das `HeapTuple`-Struct des Eintrags aus `pg_class`, das nach dem typischen Aufbau des Quellcodes aus 4.1 schon vom Systemkatalog geholt wurde.

Falls das Attribut `tenantid` einen validen Wert enthält, es sich also tatsächlich um einen Mandanten handelt, so wird der passende Eintrag aus `pg_mtclass` nachgeladen, oder falls keiner gefunden werden konnte, ein neuer Eintrag angelegt. Somit sind alle Daten vorhanden, um damit ein neues Struct `ChangeableMTClass` zu befüllen und zurückzugeben.

4.2.2.2 EndMTClassChange

Nachdem der Aufrufende alle Änderungen vorgenommen hat, kann er diese Funktion verwenden, um das aktualisierte `HeapTuple`-Struct abzuspeichern. Dazu müssen als Parameter beide zuvor geöffnete Relationen `pg_class` und `pg_mtclass` übergeben werden. Das Öffnen übernimmt der Aufrufende, da er unter Umständen mehr als einen Eintrag ändern möchte und die Relationen somit nur ein Mal geöffnet werden müssen. Zudem werden das Struct `ChangeableMTClass` und ein sogenannter `MemoryContext` übergeben, der angibt, dass das Tupel für `pg_mtclass` nach dem Aufruf in dem durch den `MemoryContext` beschriebenen Speicherbereich verbleiben soll. Wird stattdessen `NULL` übergeben, wird das Tupel aus dem Speicher gelöscht.

Anhand der in der Variable `tenantid` gespeicherten ID des Mandanten des übergebenen Structs `chmtclass` wird entschieden, ob der Eintrag in `pg_class` oder in `pg_mtclass` abgespeichert werden soll. In letzterem Fall wird zuvor aus den Arrays `values` und `is_null` des Structs ein neues `HeapTuple`-Struct geformt.

4.2.2.3 GetMTClassAttr

Das Makro `GetMTClassAttr` soll für den Zugriff auf ein Attribut eines durch das Struct `ChangeableMTClass` gekapseltes `HeapTuple`-Struct benutzt werden. Dazu benötigt es natürlich den Attributnamen und das Struct. Das Makro ist derart definiert, dass es quasi als Rückgabewert den Wert des Attributs zurückgibt.

Ist der Wert des Attributs `tenantid` des Structs valide, dann wird zuerst versucht, den Wert aus dem entsprechenden Eintrag des Array `values` auszulesen. Falls das Array `is_null` allerdings angibt, dass das Attribut `NULL` ist, wird der Wert von dem im Struct hinterlegten Eintrag namens `rd_rel` der Relation `pg_class` verwendet. Dieser wird ebenso dann verwendet, falls der Wert des Attributs `tenantid` invalide ist.

4.2.2.4 SetMTClassAttr

Mit diesem Makro kann ein Attribut auf einen neuen Wert gesetzt werden. Neben dem Struct muss dazu der Attributname und der Wert übergeben werden. Das Makro wird als Prozedur, also ohne Rückgabewert, verwendet.

Bei einer validen ID des Mandanten im Struct wird der Wert an entsprechender Stelle im Array `values` abgelegt und die zu diesem Attribut gehörende Booleanvariable in dem Array `is_null` auf den Wert `false` gesetzt. Bei einem invaliden Wert in dem Attribut `tenantid` dagegen wird der Wert in dem Eintrag `rd_rel` des Structs abgespeichert.

4.2.2.5 notequalMTClassAttrOrNotSet

Um zu prüfen, ob ein Attribut auf einen angegebenen Wert oder gar nicht gesetzt ist, kann dieses Makro verwendet werden. Dazu benötigt es das Struct, den Attributnamen und den Vergleichswert. Das Ergebnis des Vergleichs wird direkt als Boolean zurückgegeben.

Ist die ID des Mandanten valide, dann kann in `is_null` geprüft werden, ob das Attribut nicht gesetzt ist, und ansonsten das Attribut aus dem Array `values` mit dem übergebenen Wert verglichen werden. Falls die ID allerdings invalide ist, muss das Attribut per Design gesetzt sein und man braucht den Wert daher nur mit dem Attribut aus `rd_rel` zu vergleichen.

4.3 Mandanten

Die Mandanten und deren Gruppen werden als Objekte in dem Systemkatalog `pg_authid` modelliert, siehe 4.1.6. Für diese Objekte sieht das Konzept bestimmte, darauf arbeitende Operationen vor. Der Entwurf für die Umsetzung dieser Operationen wird in den nachfolgenden Abschnitten erläutert. Er orientiert sich auf Grund der ähnlichen Anforderungen an Mandanten und an Benutzer stark an die Implementierung der Rollen. Diese lässt sich in drei Teile untergliedern:

1. Zum Einen der, mit dem sich die Liste der dem System bekannten Mandanten verwalten lässt.
2. Dann der, der abspeichert, welches der aktuell gesetzte Mandant ist und diese Information allen anderen Module bereitstellt.
3. Und zuletzt der Teil, der dem Benutzer eine Schnittstelle zur Verfügung stellt, mit der er den Mandanten wechselt, unter dem alle nachfolgend abgesetzten Befehle ausgeführt werden sollen.

4.3.1 Verwaltung der Mandanten

Im Konzept sind die drei Operationen `CREATE TENANT`, `DROP TENANT` und `ALTER TENANT` vorgesehen, um die Liste der Mandanten zu verwalten. Für die beiden ersteren gibt es noch ein Pendant mit dem Schlüsselwort `TENANTGROUP` für Mandantengruppen.

Nachdem einer dieser Befehle geparst wurde, müssen die dabei eingelesenen Daten in einem Struct abgelegt werden. Da sich die geparsten Daten für Mandant und Mandantengruppe kaum unterscheiden, reicht jeweils ein Struct für beide Fälle aus. Ein vereinfachtes Beispiel für ein Struct, das zum Anlegen verwendet wird, ist in Ausschnitt 4.5 zu sehen. Es fasst den Namen, optional die Mandantengruppe im Feld `options`, die einem Mandant zugewiesen werden soll, und den anzulegenden Typ, also Gruppe oder Mandant, zusammen. Das jeweilige Struct wird dann an eine der drei folgend erläuterten Funktionen zur eigentlichen Ausführung des Befehls übergeben.

```
1 typedef struct CreateTenantStmt
2 {
3   char  *tenant;     /* tenant name */
4   List  *options;    /* List of DefElem nodes */
5   char  rolmtkind;   /* t=tenant, g=tenant group */
6 } CreateTenantStmt;
```

Ausschnitt 4.5: Struct zum Anlegen eines Mandanten bzw. einer Mandantengruppe

4.3.1.1 Erstellen eines Mandanten

Die Funktion `CreateTenant` bekommt das Struct aus Ausschnitt 4.5 übergeben. Sie versichert sich über die Tabelle `pg_authid`, dass der angegebene Name noch nicht existiert, und dass der optional angegebene Name einer Mandantengruppe existiert. Falls ein Eintrag für den als Mandantengruppe angegebenen Namen gefunden wurde, so muss zusätzlich über dessen Attribut `rolmtkind` geprüft werden, dass es sich tatsächlich um eine Mandantengruppe handelt. Sind alle genannten Bedingungen erfüllt, wird dem Systemkatalog `pg_authid` der entsprechende neue Eintrag hinzugefügt und gegebenenfalls die Beziehung zwischen Mandant und Mandantengruppe in `pg_auth_members` eingetragen.

An dieser Stelle sei darauf hingewiesen, dass die Funktion keine isolierten Tabellen für den Mandanten instanziiert. D.h. für die isolierten Tabellen werden keine Einträge in `pg_mtclass` für den Mandanten angelegt und keine eigenen Speicherbereiche auf dem externen Speicher reserviert. Dazu müsste im schlimmsten Fall die Datenbank gesperrt werden, damit, während die Liste der isolierten Tabellen abgearbeitet wird, keine neuen hinzukommen, die dann möglicherweise nicht beachtet würden. Deshalb erledigen andere Module diese Aufgaben, sobald es nötig wird.

4.3.1.2 Löschen eines Mandanten

Mithilfe der Funktion `DropTenant` wird ein Mandant oder eine Mandantengruppe gelöscht. Sie muss vorher prüfen, ob ein Eintrag in `pg_authid` mit diesem Namen tatsächlich existiert, und dass dessen in `rolmtkind` gespeicherter Typ mit dem angegebenen, also einem Mandant oder einer Gruppe, übereinstimmt. Nach erfolgreicher Prüfung, kann der Eintrag und alle Beziehungen zwischen Gruppe und Mandant aus `pg_auth_members` gelöscht werden.

4.3.1.3 Änderungen an einem Mandanten vornehmen

Die Funktion `AlterTenant` wird verwendet, um Änderungen an einem Mandanten vorzunehmen. Bisher sieht das Konzept nur vor, dass einem Mandant eine neue Mandantengruppe zugewiesen werden kann. Dazu muss geprüft werden, dass sowohl der Name des Mandanten als auch der der Gruppe existieren, und dass der jeweilige Typ übereinstimmt. Dann kann die bisherige Beziehung aus `pg_auth_members` gelöscht und die neue angelegt werden.

4.3.2 Speicherung des gesetzten Mandanten

Jedes Backend soll unabhängig von den anderen Backends die Möglichkeit haben, den Mandanten zu setzen, unter dem alle Befehle, die dieses Backend danach bekommt, ausgeführt werden. Insbesondere soll ein gesetzter Wert nur über die Laufzeit dieses einen Backends gültig sein, weshalb keine Speicherung auf dem externen Speicher notwendig ist. Daher eignet sich dazu am besten eine globale Variable im eigenen Speicherbereich. Als Datentyp wird entsprechend der neuen Spalte `rolmttenantid` im Systemkatalog `pg_authid` `int16` verwendet. Der Zugriff auf die Variable kann an jeder Stelle des Quellcodes über die entsprechenden Getter und Setter-Methoden namens `GetTenantId` und `SetCurrentTenantId` erfolgen. Wie im Systemkatalog `pg_authid` wird als invalider Wert die null verwendet. Im Quellcode wird statt der null das Makro `InvalidTenantId` verwendet.

4.3.3 Schnittstelle zum Wechseln des Mandanten

Der erste Abschnitt hat erklärt, wie Mandanten verwaltet werden. Der folgende Abschnitt soll erläutern, wie diese Mandanten genutzt werden können. Dazu sieht das Konzept einen Befehl `SET TENANT` vor, mit dem sich der aktuelle Mandant, unter dem alle nachfolgenden über die aktuelle Datenbankverbindung abgesetzten Befehle ausgeführt werden sollen, ändern lässt.

Dieser Befehl darf in PostgreSQL nicht als eigenständige Anweisung gesehen werden, die

durch ein eigenes Struct und eine eigene Funktion realisiert wird. Stattdessen handelt es sich hierbei um eine Anweisung, eine Variable namens `TENANT` auf einen Wert zu setzen. Das Modul, das solche Abfragen abarbeitet, nennt sich Global User Configuration (Abk. GUC). Es ist dafür zuständig, den Wert an das eigentliche Modul weiterzugeben, für das die Variable bestimmt ist, und den Wert gegebenenfalls zurückzusetzen, falls die aktuelle Transaktion fehlschlägt und daher ein sogenanntes Rollback durchgeführt werden muss.

Bei der Vielzahl an Variablen, die durch GUC verwaltet werden, ist es wichtig, dass das Modul sehr generisch ist. Daher muss man für eine neue Variable nicht viel mehr als einen Namen, den Datentyp der Variable, in diesem Fall ein String, und zwei Funktionsnamen angeben. Mithilfe der zwei Funktionen wird das Modul angesprochen, das eigentlich für die Variable zuständig ist. Eine namens `assign_tenant` wird benutzt, um dem Modul den neuen Wert mitzuteilen, und die andere namens `show_tenant`, um den aktuell gesetzten Mandanten zu erhalten.

4.3.3.1 assign_tenant

Diese Funktion bekommt den geparsten Namen des Befehls SET TENANT übergeben. Es wurde bis dahin noch nicht überprüft, ob der Name tatsächlich existiert, was daher an dieser Stelle erfolgt. Wurde ein Eintrag gefunden, kann das Attribut `rolmttenantid` ausgelesen werden und der Funktion `SetCurrentTenantId` übergeben werden. Zudem wird der Name in einer globalen Variable abgespeichert, die für die nächste Funktion benötigt wird.

4.3.3.2 show_tenant

Bei einem Aufruf der Funktion `assign_tenant` wird neben der ID des Mandanten auch der Name des Mandanten in einer globalen Variable gespeichert. Diese Variable wird von `show_tenant` ausgelesen und zurückgegeben.

4.4 Speicherungsschicht

Bei dieser Schicht handelt es sich um die unterste Schicht des Ansi-Schichten-Modells, die der darüberliegenden Schicht unter Anderem mehrere, einzelne Speicherbereiche auf dem externen Speicher zur Verfügung stellt. Es muss also insbesondere ein Adressschema existieren, mit dem die verschiedenen Bereiche unterschieden werden können, das durch eindeutige Bezeichner realisiert wird. Der Bezeichner wird von der Speicherungsschicht verwendet, um einen eindeutigen Dateinamen zu generieren. Die entsprechende Datei stellt dann einen eigenen, über den Bezeichner eindeutig identifizierbaren Speicherbereich dar.

Einer Tabelle wird ein Speicherbereich zugewiesen, indem das Attribut `relfilenode` auf den Bezeichner des Speicherbereichs gesetzt wird.

Laut dem Abschnitt 4.3.1.1 werden isolierte Tabellen nicht sofort instanziiert. Es kann daher vorkommen, dass für einen Mandanten noch kein Eintrag in `pg_mtclass` existiert, der auf einen mandanteneigenen Speicherbereich verweisen könnte. D.h. obwohl der Eintrag in `pg_class` über das Attribut `relfilenode` für alle Mandanten ohne Eintrag in `pg_mtclass` auf den gleichen Speicherbereich verweist, müssen unterschiedliche Speicherbereiche verwendet werden.

Um das genannte Problem zu lösen, muss die Adressierung der Speicherungsschicht erweitert werden. Eine Adresse soll daher im Folgenden aus dem ursprünglichen Bezeichner und einer ID eines Mandanten bestehen können. Damit kann die Speicherungsschicht für zwei Adressen mit gleichen Bezeichnern und unterschiedlichen IDs verschiedene Dateinamen generieren, indem die IDs jeweils als Suffix an den Dateinamen angehängt werden, der bei einer Adresse generiert worden wäre, die nur aus dem Bezeichner besteht. Es wird dadurch garantiert, dass die Dateinamen verschieden sind, und somit beide Adressen auf unterschiedliche Speicherbereiche verweisen.

Die Speicherungsschicht besitzt selbst keine Datenstruktur, in der Informationen über die vorhandenen Speicherbereiche verwaltet werden. Stattdessen wird lediglich erwartet, dass eine über einen Bezeichner angesprochene Datei im Dateisystem existiert, sonst wird ein Fehler geworfen. Bevor die Speicherungsschicht angewiesen wird, eine Datei zu öffnen, muss daher sichergestellt werden, dass die Datei vorhanden ist. Das Öffnen erledigt das Makro `RelationOpenSmgr`, das erweitert werden muss, so dass zu anfangs mittels `smgrexists` die Existenz einer Datei abgefragt wird und gegebenenfalls mittels `smgrcreate` angelegt wird.

4.5 Tablespace je Mandantengruppe

In PostgreSQL lässt sich beim Anlegen jeder Tabelle und jedes Index ein Tablespace angeben, in dem die jeweiligen Daten abzulegen sind. Teil des Konzepts ist es nun, dass man diesen Tablespace beim Anlegen für jede Mandantengruppe separat angeben kann.

4.5.1 Festlegung der Zuweisung

Der Administrator muss beim Anlegen der Tabelle bzw. des Index die Zuweisung festlegen können. Dazu wurde der Syntax beider Anweisungen entsprechend dem Abschnitt 2.1.8 erweitert. Beim Parsen wird aus den Daten eine Liste von Zuweisungen aufgebaut, bei der jede Zuweisung aus einem Namen eines Tablespace und optional mehreren Namen von Mandantengruppen, die diesen Tablespace benutzen sollen, besteht. Die Angabe der Man-

dantengruppen ist optional, da es weiterhin möglich sein muss, einen Standardtablespace anzugeben, der für alle verwendet wird, für die keine andere Zuweisung existiert.

Die Funktionen `DefineIndex` und `DefineRelation`, die einen Index bzw. eine Relation anlegen, müssen die übergebene Liste in den Systemkatalog umsetzen. Eine Möglichkeit wäre es, den Tablespace je Mandant in dem entsprechenden Eintrag in `pg_mtclass` abzuspeichern. Allerdings würde die Zuweisung dann nicht automatisch für neue Mandanten übernommen werden. Andererseits gibt es in den Systemkatalogen für isolierte Tabellen keine Einträge, die je Mandantengruppe abgespeichert werden. Daher wird die komplette Liste so kodiert, dass sie in einem im Abschnitt 4.1.1 beschriebenen Attribut abgespeichert werden kann.

Dazu soll eingelesene Liste mithilfe der Funktion `transformMTTablespace` serialisiert werden, die ihr als Parameter übergeben wird. Dabei werden nicht die Namen abgespeichert, sondern die IDs der Tablespaces und der Mandantengruppen, wobei gleichzeitig geprüft wird, dass die jeweiligen Einträge auch tatsächlich existieren.

Während der Serialisierung wird zudem der Standardtablespace herausgefiltert, der nicht in der Liste gespeichert wird, sondern nach wie vor in einem Attribut in `pg_class`. Des Weiteren ist es durch die Änderungen am Syntax möglich, dass der Benutzer mehr als einen Standardtablespace angeben kann. Daher wird dieser Fall an dieser Stelle abgefangen.

Die Funktion `transformMTTablespace` muss die serialisierte Liste und den Standardtablespace zurückgeben, wofür das Struct in Ausschnitt 4.6 verwendet wird. Das erste Attribut im generischen Datentyp `Datum` enthält die Liste und das zweite ist eine Referenz auf den Eintrag in dem Systemkatalog `pg_tablespace`.

```
1 struct MTTablespace {
2   Datum tenantgroupstablespacenames;
3   Oid defaulttablespace;
4 };
```

Ausschnitt 4.6: Struct für `transformMTTablespace`

4.5.2 Tablespace nachschlagen

Da der zu verwendende Tablespace, wie in dem vorherigen Abschnitt erklärt wurde, statt in einem Eintrag des Mandanten in `pg_mtclass` in dem Eintrag in `pg_class` in einer Liste abgespeichert wird, muss man diese Liste durchgehen, um anhand der ID einer Mandantengruppe den entsprechenden Tablespace zu finden.

Diese Aufgabe übernimmt eine Funktion namens `extractMTTablespace`, die als Parameter das Attribut des Eintrags aus `pg_class` übergeben und die ID der Mandantengruppe übergeben bekommt. Sie startet eine Suche nach einer Zuweisung, die die ID enthält, auf

der im Attribut gespeicherten Liste. Als Rückgabewert wird die ID des Tablespace zurückgegeben, falls einer gefunden wurde. Ansonsten wird ein invalider Wert zurückgegeben und somit dem Aufrufer signalisiert, dass er den Standardtablespace verwenden soll.

4.5.3 Anlegen der Indexdateien

Eine Anforderung an den Entwurf, die im Abschnitt 2.1.4 angeführt wurde, ist, dass es möglich sein soll, einer Mandantengruppe eine eigene Indexdatei zuzuweisen. Der ebenfalls in dem Abschnitt erklärte Gedanke hinter dieser Anforderung hat zwar mit den Tablespaces nichts zu tun, allerdings erfordert die Zuweisung unterschiedlicher Tablespaces je Gruppe zwangsläufig, dass die Daten in eine Indexdatei je Tablespace aufgespalten werden. Aus dem genannten Grund muss ähnlich wie bei den mandanteneigenen Dateien für isolierte Tabellen je Tablespace-Zuweisung eine eigene Indexdatei angelegt werden.

Im Abschnitt 4.3.1.1 wurde erklärt, dass es bei isolierten Tabellen zu Problemen führt, würde man die Dateien sofort anlegen wollen. Bei den Indexdateien treten diese Probleme nicht auf, da die Liste der Mandantengruppen explizit vom Administrator angegeben wurde und sie sich daher während der Ausführung durch das Anlegen einer weiteren Mandantengruppe mittels `CREATE TENANTGROUP` nicht ändern kann.

In einer von `DefineIndex` aufgerufenen Funktion wird die Funktion `createPGMtClassEntriesByMTTablespace` ausgeführt, die das Anlegen der Indexdateien übernimmt. Sie durchläuft die von `transformMTTablespace` erstellte Liste und legt für jede gefundene Mandantengruppe einen Eintrag in `pg_mtclass` an. Des Weiteren wird der Speicherungsschicht durch einen Aufruf von `RelationCreateStorage` mitgeteilt, dass die Datei sofort angelegt werden soll.

4.6 Isolierte Tabellen

Im Abschnitt 2.1.8 wurde die syntaktische Erweiterung eingeführt, mit der eine gegenüber anderen Mandanten isolierte Tabelle angelegt werden soll. Das Anlegen bei normalen Tabellen wird von der Funktion `DefineRelation` und einigen Unterfunktionen übernommen. Um damit isolierte Tabellen anzulegen sind daher einige kleinere Anpassungen notwendig, die im darauffolgenden Abschnitt aufgelistet werden.

Als ein Teil des Konzepts wurde festgelegt, dass die Isolierung der Daten über die Speicherungsschicht erfolgen soll, indem jeder Mandant eine eigene Datei zugewiesen bekommen soll. Zudem erwartet PostgreSQL, dass es zu jederzeit möglich ist, dass einem Mandanten eine neue Datei zugewiesen werden kann. Die dazu notwendige Zuordnung von einem Mandant zu seiner Datei wird im zweiten Abschnitt erläutert.

Die Abbildung 4.2 skizziert den Entwurf. Im Systemkatalog `pg_class` befindet sich ein

Eintrag für eine isolierte Tabelle namens `articles` und im Systemkatalog `pg_mtclass` befinden sich Einträge von zwei Mandanten. Die Pfeile zum Eintrag in `pg_class` bedeuten, dass die Einträge der Mandanten zu dieser isolierten Tabelle gehören. Aus den zwei Einträgen wurde jeweils ein Relationen Deskriptor aufgebaut. Beide verwenden außerdem den Eintrag aus `pg_class`, um die gemeinsamen Metainformationen des Basisschemas auszulesen. Das Attribut `relfilenode` eines Eintrags des Systemkatalogs `pg_mtclass` verweist auf eine Datei der Speicherungsschicht, auf die daher auch der aus diesem Eintrag aufgebaute Relationen Deskriptor verweist. Alle Operationen auf der durch den Deskriptor beschriebenen Relation greifen dadurch auf die Datei zu.

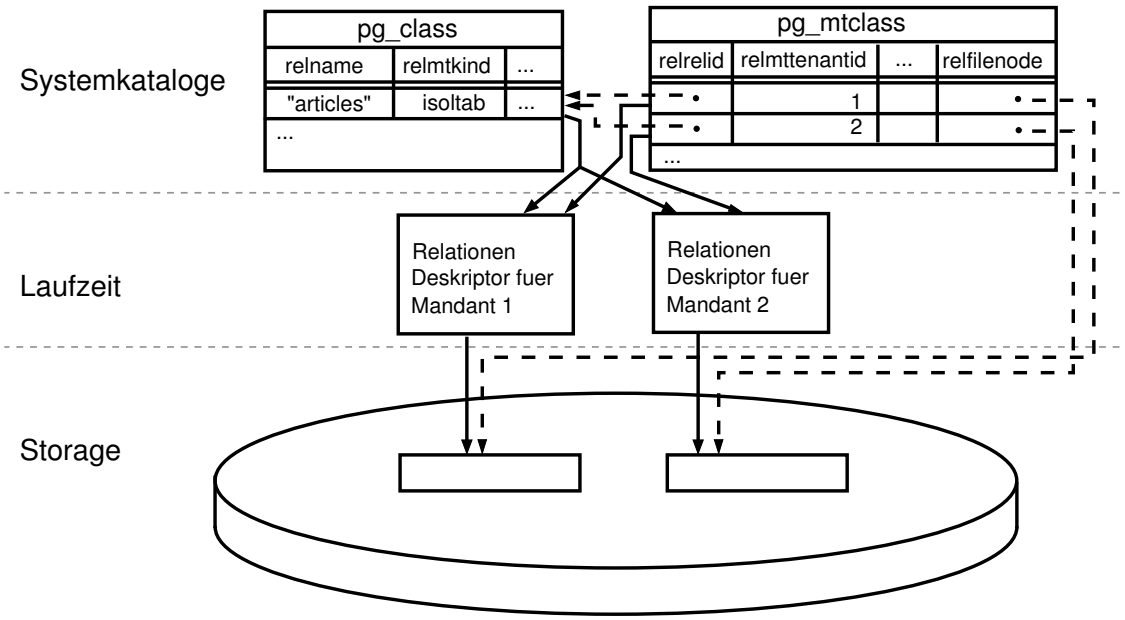

Abbildung 4.2: Entwurf für die isolierte Tabelle

4.6.1 Anlegen einer isolierten Tabelle

Wird eine Tabelle mit den Schlüsselwörten `SEGREGATED BETWEEN TENANTS` angelegt, so wird das der Funktion `DefineRelation` über ein Struct mitgeteilt. In diesem Fall werden einige zusätzliche Schritte ausgeführt. Beispielsweise wird beim Hinzufügen des Eintrags in `pg_class` in dem im Abschnitt 4.1.1 eingeführten Attribut `rolmtkind` vermerkt, dass es sich um eine isolierte Tabelle handelt.

Des Weiteren wird in diesem Fall eine zusätzliche Spalte `pg_tenantid` angelegt, die die ID des Mandanten modelliert. Nach außen wird sie durch den Parser versteckt und kann somit nicht vom Mandanten genutzt werden. Selbst intern wird für diese Spalte kein Wert gesetzt und benötigt insbesondere auf dem externen Speicher keinen zusätzlichen Platz.

Es erleichtert jedoch an einigen Stellen die Implementierung, wenn die Tupel aus logischer Sicht Bezug auf den Mandanten nehmen.

Falls eine Zuordnung bezüglich Tablespaces angegeben wurde, muss diese im Systemkatalog hinterlegt werden. Das wird durch einen Aufruf der im Abschnitt 4.5.1 beschriebenen Funktion `transformMTTablespace` und dem Abspeichern des Rückgabewerts in dem im Abschnitt 4.1.1 eingeführten Attribut `relmttablespaces` des anzulegenden Eintrags in `pg_class` erledigt.

Ebenso wie das in dem Abschnitt 4.3.1.1 erwähnte Problem beim sofortigen Instanziieren einer isolierten Tabelle für einen neuen Mandanten hätte man auch hier das Problem, dass während des Abarbeitens der Liste aller Mandanten zum Instanziieren ein neuer Mandant hinzukommt, der somit unter Umständen nicht beachtet werden würde. Daher wird auch in diesem Fall darauf verzichtet und die Datei daher über den im Abschnitt 4.4 erläuterten Mechanismus beim ersten Zugriff angelegt.

4.6.2 Isolierung durch eigene Dateien

Jedem Mandant muss über die Speicherungsschicht eine eigene Datei zugeordnet werden können. Dazu wird die im Abschnitt 4.4 eingeführte, erweiterte Adressierung verwendet, die sich aus zwei Teilen zusammensetzt. Zum Einen dem ursprünglichen Bezeichner, für den der zu verwendende Wert aus einem Attribut namens `relfilenode` aus einem Systemkatalog geholt wird. Das Attribut kann hierbei aus `pg_class` kommen oder aus `pg_mtclass`, falls bereits eine Änderung vorliegt, damit einem einzelnen Mandant eine neue Datei zugeordnet werden kann. Der zweite Teil ist die ID des Mandanten.

4.7 Relationen Deskriptor

Der Relationen Deskriptor fasst alle zu einer Relation gehörenden und zur Laufzeit benötigten Informationen in einem Struct zusammen. Da bisher allein durch den Systemkatalog `pg_class` bestimmt wurde, welche Relationen existieren, konnte es nur dann einen Relationen Deskriptor geben, falls dafür ein eigener Eintrag in `pg_class` vorhanden war. Damit wäre für eine isolierte Tabelle nur ein Relationen Deskriptor für alle Mandanten möglich. Die Relationen sollen sich jedoch so stark unterscheiden können, dass ein eigener Deskriptor je Mandant möglich sein muss.

So ein eigener Deskriptor eines Mandanten muss Informationen enthalten können, die sowohl vom Basisschema als auch von vom Mandanten gemachten Änderungen stammen können. Daher muss das Auslesen beim Aufbau des Deskriptors beide Teile berücksichtigen und sie so verknüpfen, dass sie im Relationen Deskriptor transparent abgespeichert

werden können, damit andere Teile des Quellcodes nicht oder wenigstens kaum angepasst werden müssen.

Beim Verknüpfen müssen die mandantenspezifischen Teile in den Arbeitsspeicher geladen werden, die gemeinsamen Teile stehen jedoch womöglich schon im Arbeitsspeicher. Damit die gemeinsamen Teile gefunden werden können, werden sie in einem Relationen Deskriptor abgelegt, der ausschließlich aus den Informationen des Basisschemas erstellt wurde. Die Relationen Deskriptoren der Mandanten erhalten einen Verweis auf diesen Relationen Deskriptor des Basisschemas. Dadurch kann über einen Relationen Deskriptor eines Mandanten schnell auf den des Basisschemas zugegriffen werden und damit auch auf den gemeinsamen Teil. Der Verweis ist eines von mehreren Attributen, die dem Relationen Deskriptor hinzugefügt werden.

Des Weiteren wird ein eigener Deskriptor benötigt, falls es laut einem Eintrag des Systemkatalogs `pg_mtclass` eine eigene Indexdatei gibt, da der Deskriptor insbesondere vorgibt, in welche Datei die Speicherungsschicht die Daten der Relation abspeichern soll. Dabei können sich Mandanten ohne eigene Indexdatei, deren Mandantengruppe jedoch eine eigene Indexdatei besitzt, einen Deskriptor teilen. Deswegen gibt es sowohl Deskriptoren für Mandanten als auch für Mandantengruppen.

4.7.1 Änderungen am Relationen Deskriptor

Wie im vorherigen Abschnitt erwähnt wurde, muss der Relationen Deskriptor um einige neue Attribute erweitert werden. Die deshalb zu machenden Änderungen sind in Ausschnitt 4.7 zu sehen. Die einzelnen Attribute werden in folgender Liste in der Reihenfolge der Definition im Struct näher beschrieben.

- Das erste Attribut `rd_mttenantid` ist auf die ID des Mandanten oder der Mandantengruppe gesetzt, für den oder die der Deskriptor aufgebaut wurde, falls es mehrere Instanzen geben kann. Bei einer gemeinsam genutzten Tabelle ist keine eigene Instanz notwendig ist, da die Mandanten die Metainformationen nicht anpassen können und der Deskriptor daher für alle gleich sein muss. Bei einer in der Aufgabenstellung nicht vorgesehenen, aber denkbaren, selbst definierten Tabelle eines Mandanten, kann es nur die eine Instanz des Mandanten geben. In beiden Fällen wird das Attribut daher auf einen invaliden Wert gesetzt.

- Das Attribut `rd_mtbaserelation` verweist auf den Deskriptor, der ausschließlich aus den Informationen des Basisschemas erstellt wurde.

- Als Gegenstück zu dem Attribut `rd_mtbaserelation` gibt es eines mit dem Bezeichner `rd_mttenantsrelations`. Es ist nur beim Deskriptor des Basisschemas gesetzt und enthält eine Liste aller Deskriptoren der Mandanten. Diese Liste wird bei der Invalidierung des Deskriptors des Basisschemas verwendet, um alle Deskriptoren der Mandanten ebenfalls zu invalidieren.

- Das Attribut `rd_mtrebuildsactive` wird während dem Aufbau des Deskriptors gesetzt. Zuerst ist es null, dann wird es beim Beginn des Aufbaus auf eins gesetzt und falls zu einem Zeitpunkt ein zweiter Aufbau gestartet wurde, bevor der erste beendet wurde, dann wird das durch den Wert zwei ausgedrückt.
- Das Attribut `rd_mtrel` verweist auf eine Kopie des Eintrags aus `pg_mtclass`, falls es sich bei diesem Deskriptor um eine für einen Mandanten bzw. Mandantengruppe erstellte Relation einer isolierten Tabelle oder eines gemeinsamen Index handelt. Der Wert darf `NULL` sein, für den Fall, dass noch kein eigener Eintrag in `pg_mtclass` vorhanden ist.
- Das letzte Attribut ist nur gesetzt, falls es sich um den Deskriptor handelt, der ausschließlich aus dem Basisschema erstellt wurde. Es enthält die Liste der Zuordnungen von einer Mandantengruppe zu einem Tablespace.

```
1  typedef struct RelationData {
2    ...
3    int2                    rd_mttenantid;
4    struct RelationData   *rd_mtbaserelation;
5            /* if tenantid is set points to base relation */
6    List                   *rd_mttenantsrelations;
7            /* relation entries of tenants */
8    int2                    rd_mtrebuildsactive;
9    struct HeapTupleData *rd_mtrel; /* pg_mtclass tuple*/
10   Datum                   rd_mttenantgroupstablespace;
11   ...      /* list of tenantgroups and their tablespace */
12 } RelationData;
```

Ausschnitt 4.7: Zusätzliche Attribute für den Relationen Deskriptor

4.7.2 Auf- und Abbau des Relationen Deskriptors

Der folgende Abschnitt erläutert den Aufbau des Deskriptors aus dem Systemkatalog und gegebenenfalls aus den im Deskriptor des Basisschemas gespeicherten gemeinsamen Teilen. Es wird hier insbesondere geklärt, wodurch und wo erkannt wird, dass für eine zu öffnende Tabelle ein eigener Deskriptor aufgebaut werden muss.

Des Weiteren können die Informationen des Deskriptors veraltet sein, weswegen der im Abschnitt 3.8 erwähnte Invalidierungsmechanismus den Abbau des Deskriptors anstößt. Dazu müssen Ressourcen freigegeben werden und bestehende Verknüpfungen zu dem Deskriptor gelöst werden.

Nach dem Abbau ist ein weiterer Schritt notwendig. Der Deskriptor kann von mehreren Modulen gleichzeitig benutzt werden und besitzt daher einen Referenzzähler, der angibt,

wie oft auf ihn verwiesen wird. Ist dieser nach dem Abbau noch auf einen Wert größer als null gesetzt, dann muss der Deskriptor neu aufgebaut werden, damit das referenzierende Modul seine Arbeit vollends verrichten kann.

Die Abbildung 4.3 zeigt das Ergebnis des Aufbaus eines Relationen Deskriptors eines Mandanten. Es ist die Beziehung zwischen dem Deskriptor des Basisschemas und dem des Mandanten über die Attribute `rd_mtbaserelation` und `rd_mttenantsrelations` eingezeichnet. Des Weiteren verweisen beide Deskriptoren auf das selbe Tupel aus `pg_class`, auf das aus `pg_mtclass` verweist dagegen nur der des Mandanten. Das Attribut `rd_att` zeigt jeweils auf einen Tupel Deskriptor, der vereinfacht als Array dargestellt wurde. Jedes Feld des Arrays zeigt auf ein Attribut. Die gemeinsamen werden geteilt.

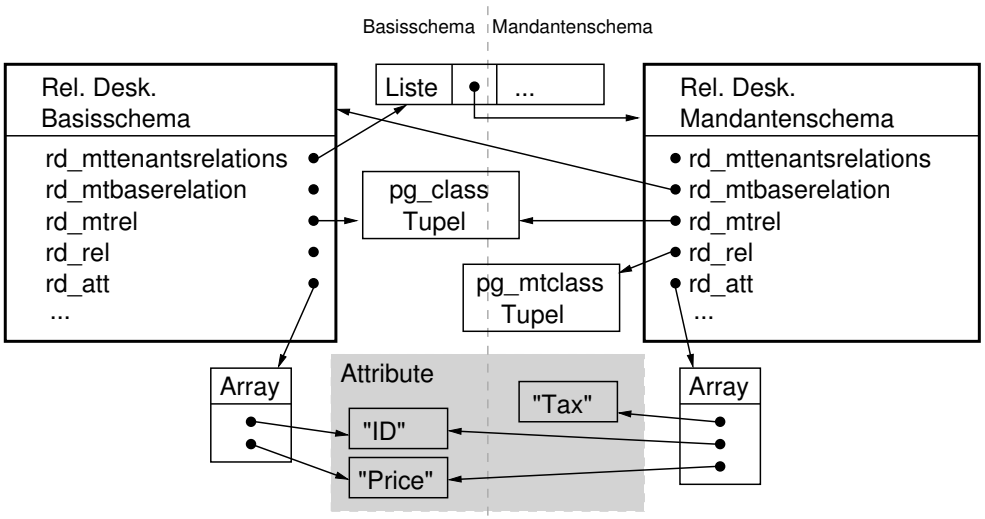

Abbildung 4.3: Verknüpfung der Relationen Deskriptoren

4.7.2.1 Öffnen einer Relation als Mandant

Der Aufbau eines Deskriptors wird in der Regel durch einen Aufruf der Funktion `relation_open` angestoßen. Die Funktion bekommt hierbei lediglich die ID übergeben, die den Eintrag der Tabelle in `pg_class` referenziert. Die aufrufende Funktion bestimmt also nicht selbst, für welchen Mandant der Deskriptor aufgebaut werden soll. Da die Funktion sehr häufig vorkommt, wäre eine Anpassung diesbezüglich viel zu aufwändig. Stattdessen muss die Funktion selbst herausbekommen, ob es sich um eine isolierte Tabelle bzw. einen gemeinsamen Index handelt und für welchen Mandanten der Deskriptor in diesem Fall aufgebaut werden soll.

Für so einen Fall wurde die Funktion `GetTenantId` definiert, mit der an jeder Stelle des Quellcodes die ID des aktuell gesetzten Mandanten bestimmt werden kann. Somit kann

die Funktion `relation_open` als Erstes feststellen, für welchen Mandanten die Relation zu öffnen ist.

Danach wird bestimmt, ob der Typ der Relation einer isolierten Tabelle entspricht. Deshalb wird der Eintrag aus `pg_class` nachgeschlagen und das Attribut `relmtkind` ausgelesen. Fall es eine isolierte Tabelle ist, wird ein Deskriptor für den Mandanten geöffnet.

Sollte der Typ stattdessen ein gemeinsamer Index auf einer isolierten Tabelle sein, so kann hierfür ebenso ein eigener Deskriptor notwendig sein. Ob einer gebraucht wird, hängt von den Einträgen in `pg_mtclass` ab. Falls ein Mandant eine eigene Indexdatei besitzt, so existiert dort ein Eintrag für ihn, der auf die Datei verweist. In dem Fall wird ein eigener Deskriptor für den Mandanten aufgebaut und zurückgegeben. Ansonsten muss die Gruppe des Mandanten bestimmt werden, um zu prüfen, ob hierfür ein Eintrag in `pg_mtclass` existiert, was vorkommen kann, falls alle Mandanten einer Gruppe eine eigene, gemeinsame Indexdatei bekommen sollen. Nur falls weder für den Mandanten noch für seine Gruppe ein Eintrag gefunden wurde, wird der Deskriptor der Basisschema zurückgegeben.

Nachdem die ID des Mandanten feststeht, für den der Deskriptor benötigt wird, wird diese Information an die im folgenden Abschnitt beschriebenen Funktion `RelationIdGetRelation` weitergegeben, die den Deskriptor zurückgibt.

4.7.2.2 Verwaltung in der Hashtabelle

Die Funktion `relation_open` startet den Aufbau nicht selbst. Falls der Deskriptor nämlich schon in Verwendung ist und somit schon aufgebaut wurde, kann dieser zurückgegeben werden. Deswegen wird zuerst die Funktion `RelationIdGetRelation` aufgerufen, die nun insbesondere die ID desjenigen Mandanten bzw. der Mandantengruppe übergeben bekommt, für die der Deskriptor aufgebaut werden soll. Sie versucht zuerst, einen schon aufgebauten Deskriptor aus dem als Hashtabelle realisierten Cache zu bekommen. Die Hashtabelle muss dazu als zusätzlichen Schlüssel zur Identifizierung die ID des Mandanten oder der Gruppe verwenden. Falls der Deskriptor nicht gefunden werden konnte, wird der Aufbau durch die nachfolgend beschriebene Funktion aufgebaut, und der Deskriptor danach in der Hashtabelle eingetragen.

4.7.2.3 Aufbau eines Deskriptors

Falls ein Deskriptor nicht in der Hashtabelle gefunden wurde, wird er durch einen Aufruf der Funktion `RelationBuildDesc` aufgebaut. Sie bekommt als Parameter die Referenz auf den entsprechenden Eintrag in `pg_class` und die ID des Mandanten übergeben.

Erste Schritte in RelationBuildDesc Der Aufbau des Relationen Deskriptors wird von der Funktion `RelationBuildDesc` durchgeführt. Sie ist dafür zuständig alle Informationen aus den Systemkatalogen zusammen zu sammeln und aus ihnen den Deskriptor zu erstellen. Bei einem Deskriptor für einen Mandanten kommen einige zu lesende Informationen hinzu und einige fallen weg. Für letztere werden stattdessen Informationen aus dem Deskriptor des Basisschemas verwendet, um zum Einen unnötige Zugriffe auf die Systemkataloge zu vermeiden und zum Anderen, um direkt auf sie zu verweisen, anstatt sie zu kopieren, was den Arbeitsspeicherverbrauch senkt.

Um an die gemeinsamen Teile im Deskriptor des Basisschemas zu gelangen, erfolgt als erstes ein Aufruf der Funktion `RelationIdGetRelation`, der den Deskriptor gegebenenfalls aufbaut und in jedem Fall zurückliefert. Da die Funktion außerdem einen Referenzzähler erhöht, wird sichergestellt, dass der Deskriptor nicht abgebaut wird, solange der Deskriptor des Mandanten auf Teile des anderen Deskriptors zeigt.

AllocateRelationDesc Als nächstes wird die Funktion `AllocateRelationDesc` aufgerufen, die den Speicherplatz für den Deskriptor reservieren soll. Da ein Attribut des Relationen Deskriptors ein Verweis auf eine Kopie des Eintrags aus `pg_class` ist, wird normalerweise ebenfalls der Speicherplatz für diese Kopie reserviert. Handelt es sich jedoch um einen Deskriptor eines Mandanten wird der Verweis direkt auf die Kopie des Deskriptors des Basisschemas gesetzt und daher wird kein eigener Speicherplatz benötigt.

Statt einer eigener Kopie des Eintrags aus `pg_class` besitzt ein Deskriptor eines Mandant jedoch eine eigene Kopie des Eintrags aus `pg_mtclass`. Daher muss dieser Eintrag geholt und das Attribut `rd_mtrel` auf die Adresse gesetzt werden.

Zudem wird in dieser Funktion der Speicherplatz für den Tupel Deskriptor reserviert. Für einen Mandant kann es notwendig sein, einen eigenen Tupel Deskriptor zu erstellen. Das wäre zum Beispiel der Fall, falls er ein Attribut hinzugefügt oder eine eigene zu prüfende Bedingung für ein Attribut angegeben hat, da diese beiden Informationen im Tupel Deskriptor abgelegt werden. Aufgrund dessen, dass der Tupel Deskriptor zu diesem Zeitpunkt noch nicht aufgebaut wird, ist es jedoch nicht möglich zu wissen, ob für den Mandant ein eigener benötigt, weshalb der Speicherplatz dafür vorerst reserviert werden muss.

Beim Reservieren des Speicherplatzes für den Tupel Deskriptor kann jedoch angegeben werden, für wie viele Attribute er den Speicherplatz reservieren kann. Da der Eintrag aus `pg_mtclass` schon geladen wurde, kann man sehr einfach berechnen, wie viele eigene Attribute der Mandant hat und dementsprechend wie viele im eigenen Tupel Deskriptor abgelegt werden müssen, indem man die Anzahl der Attribute von dem Eintrag aus `pg_mtclass` von dem des Eintrags aus `pg_class` subtrahiert.

Neue Attribute initialisieren Nachdem der Speicherplatz für den neuen Deskriptor reserviert wurde, kann damit begonnen werden die Attribute zu initialisieren. Insbesondere

müssen nun die neuen Attribute gesetzt werden. Die ID des Mandanten wurde als Parameter an die Funktion `RelationBuildDesc` übergeben und kann daher direkt gesetzt werden. Ebenso kann das Attribut `rd_mtrebuildsactive` einfach auf eins gesetzt werden. Außerdem wurde der Deskriptor des Basisschemas zu Beginn nachgeschlagen, kann also auch direkt auf die dabei zurückgelieferte Referenz gesetzt werden. Des Weiteren ist es über diese Referenz leicht möglich den gerade im Aufbau befindlichen Deskriptor in die Liste `rd_mttenantsrelations` des Deskriptors des Basisschemas hinzuzufügen.

Es bleibt das Attribut `rd_tenantgroupstablespace` das lediglich bei dem Deskriptor des Basisschemas gesetzt wird. In diesem Fall wurde das `HeapTuple`-Struct des Eintrags aus `pg_class` geladen, das allerdings nach dem Aufbau wieder freigegeben wird. Danach gibt es nur noch eine im Attribut `rd_rel` des Deskriptors gespeicherte Kopie des Structs, mit dem man direkt auf die Attribute zugreifen kann. Da ein Struct jedoch nur den Zugriff auf Attribute fester Länge zulässt und das Attribut `relmttablespaces` variable Länge besitzt, kann man über `rd_rel` nicht darauf zugreifen. Stattdessen muss das Attribut vom `HeapTuple`-Struct in das Attribut `rd_tenantgroupstablespace` des Deskriptors kopiert werden.

Das bisher nicht erwähnte neue Attribut für das Tupel aus `pg_mtclass` wurde schon in `AllocateRelationDesc` auf eine Kopie des Eintrags gesetzt, weshalb hierfür nichts mehr zu tun ist.

Datendatei festlegen Ein Attribut des Relationen Deskriptors enthält die Adresse auf eine Datei der Speicherungsschicht, die alle Daten der Relation abspeichern soll. Dazu muss bei einer isolierten Tabelle das im Abschnitt 4.6.2 erklärte Schema zur Initialisierung der Adresse verwendet werden, um die Isolierung der Daten zu erreichen. Demnach wird das Attribut `relfilenode` des `HeapTuple`-Structs `rd_mtrel` für den ersten Teil der Adresse verwendet, falls es gesetzt ist, und andernfalls das des `HeapTuple`-Structs `rd_rel`. Der zweite Teil wird auf die ID des Mandanten oder der Mandantengruppe gesetzt, für den oder die der Deskriptor aufgebaut wird.

Der Tablespace, in dem die Datei angelegt wird, muss außerdem je Mandantengruppe definiert werden können. Der Tablespace muss ebenfalls als Teil der Adresse der Speicherungsschicht übergeben werden und muss daher beim Festlegen der zu verwendenden Datendatei ebenfalls bekannt sein. Da der Tablespace für Mandantengruppen definiert wird, muss zuerst die ID der Mandantengruppe des Mandanten nachgeschlagen werden. Mit dieser ID kann der Tablespace in der im Attribut `rd_mttenantgroupstablespace` des Deskriptors des Basisschemas abgespeicherten Liste gesucht werden, indem die im Abschnitt 4.5.2 beschriebene Funktion `extractMTTablespace` aufgerufen wird. Falls kein Tablespace in der Liste gefunden wurde, wird wie der Tablespace aus dem Eintrag `pg_class` als Standard verwendet.

Tupel Deskriptor aufbauen Nachdem durch die Funktion `AllocateRelationDesc` der Speicherplatz für den Tupel Deskriptor reserviert wurde und das Attribut `rd_att` des Relationen Deskriptors darauf zeigt, müssen noch die Informationen aus dem Systemkatalog geladen und dorthin kopiert werden. Im Falle eines Mandanten werden nur die eigenen Attribute aus dem Systemkatalog geladen, die Referenzen der anderen Attribute werden stattdessen auf die Attribute des für das Basisschema erstellten Tupel Deskriptors gesetzt.

Bei den zu prüfenden Bedingungen einzelner Attribute, die ebenfalls im Tupel Deskriptor gespeichert werden, wäre es schön, falls man hier genauso vorgehen könnte. Leider jedoch werden auf diese Einträge des entsprechenden Systemkatalogs nicht einzelne Referenzen im Tupel Deskriptor abgespeichert, wie das bei den Attributen der Fall ist, sondern nur eine Referenz auf ein zusammenhängendes Array. Es ist daher nicht möglich einfach nur auf den gemeinsamen Teil der Einträge zu verweisen, ohne dass einige Änderungen an anderen Stellen des Quellcodes notwendig wären, weshalb vorerst eine komplett eigene Kopie des Arrays angelegt wird.

4.7.2.4 Abbau eines Relationen Deskriptors

Ein über die Funktion `RelationBuildDesc` aufgebauter Deskriptor wird in der Hashtabelle abgelegt und kann dort eingetragen bleiben, bis das Backend beendet wird. Jedoch kann es durch Änderungen am Systemkatalog vorkommen, dass die Informationen im Deskriptor veraltet sind. In diesem Fall wird der Deskriptor abgebaut, indem der Speicherplatz freigegeben wird, Referenzzähler von Instanzen, auf die vom Deskriptor aus verwiesen wurde, dekrementiert werden und die Verbindung durch Verweise auf Deskriptoren der Mandanten und des Basisschemas wieder gelöst werden.

Durch die Verweise auf gemeinsame Teile muss nun geklärt werden, wer diese Teile freigibt. Eine gängige Methode wäre es, für die Teile jeweils einen Referenzzähler zu verwenden. Da die Deskriptoren der Mandanten allerdings auf den des Basisschemas verweisen und dadurch den Referenzzähler erhöhen, ist der des Basisschemas immer der letzte, der abgebaut werden kann. Und daher liegt es nahe, dass nur beim Abbau von dem des Basisschemas die gemeinsamen Teile freigegeben werden, um sich weitere Referenzzähler zu sparen.

4.7.2.5 Erneuter Aufbau eines Relationen Deskriptors

Unter anderem in dem Fall, dass der Deskriptor eines Basisschemas veraltet ist und ein Deskriptor eines Mandanten noch eine Referenz darauf besitzt, kann es vorkommen, dass der Referenzzähler nach dem Abbau größer als null ist. Es gibt also noch eine Referenz auf den Deskriptor, die noch verwendet werden kann. Daher wird in so einem Fall das Struct des Relationen Deskriptors nach dem Abbau nicht freigegeben. Stattdessen wird darauf wieder über die Funktion `RelationBuildDesc` ein neuer Deskriptor aufgebaut.

Gerade beim Abbau eines Deskriptors des Basisschemas entsteht nun ein neues Problem, das es zu lösen gilt. Ist der Grund für den Abbau, dass die Informationen veraltet sind, so können auch die Deskriptoren der Mandanten noch Kopien von diesen veralteten Informationen besitzen. Beispielsweise werden die per Attribut zu prüfenden Bedingungen in den Tupel Deskriptor des Mandanten kopiert. Daher müssen beim Abbau eines Relationen Deskriptors des Basisschemas alle diejenigen der Mandanten ebenfalls abgebaut und gegebenfalls bei einem Referenzzähler größer null wieder aufgebaut werden.

Selbst wenn die Datenstrukturen und der Quellcode soweit angepasst würde, dass nur noch Verweise auf gemeinsame Teile statt eigene Kopien vorhanden wären, bekäme man das Problem, dass sich durch den Abbau und den erneuten Aufbau eines Deskriptors des Basisschemas die Adressen ändern und die Verweise damit ungültig wären. Das ist daher ein weiterer Grund, weshalb alle Deskriptoren der Mandanten neu aufgebaut werden.

Wird der Abbau dagegen für einen Deskriptors eines Mandanten durchgeführt, so passiert das entweder wegen dem soeben erklärten Mechanismus, oder da nur der Teil eines Mandanten verändert wurde. In beiden Fällen genügt es daher nur diesen Deskriptor ab- und gegebenenfalls anschließend wieder aufzubauen.

4.7.2.6 Rekursiver Aufbau verhindern

Während ein Deskriptor aufgebaut wird, muss auf die Systemkataloge zugegriffen werden. Da deren Implementierung im Grunde auf normalen Tabellen basiert, muss beim Laden eines bisher nicht im Cache befindlichen Eintrags ebenfalls ein Relationen Deskriptor für die entsprechende Tabelle des Systemkatalogs geöffnet werden. Bevor jedoch eine Relation geöffnet werden kann, wird ein Mechanismus gestartet, der dafür sorgt, dass die im Cache befindlichen Relationen alle auf dem neuesten Stand sind. D.h. die in einer Liste gehaltenen, zuvor als invalide markierten Deskriptoren werden nach dem im vorherigen Abschnitt beschrieben Schema neu aufgebaut. Damit kann ein Aufbau eines Deskriptors einen Aufbau eines anderen Deskriptors über eine Rekursion nach sich ziehen, ohne dass der erste Aufbau fertig wäre.

Diese Vorgehensweise führte bisher zu keinen Problemen, da die im Arbeitsspeicher gehaltenen Deskriptoren nichts miteinander zu tun hatten. Hingegen können jetzt in der Liste der invaliden Deskriptoren, mehrere vorkommen, die über die neuen Attribute miteinander verknüpft sind. Beispielsweise kann einer eines Mandanten und der des Basisschemas in der Liste enthalten sein. Wird nun der des Mandanten neu aufgebaut, so wird die Liste durch die Rekursion weiter durchlaufen. Kommt die Rekursion bei dem des Basisschemas an, so wird dieser zwangsweise auch aufgebaut, da der des Mandanten noch eine Referenz darauf besitzt. Nach dem im vorherigen Abschnitt beschriebenen Schema werden dadurch alle Deskriptoren der Mandanten ebenfalls neu aufgebaut, inklusive dem ersten der Liste. Der Aufbau eines Deskriptors eines Mandanten kann daher den Aufbau desselben Deskriptors

als Folge haben. Da es hierbei zu unvorhergesehenen Fehlern kommen kann, muss dieser Fall verhindert werden.

Es reicht nicht, die Referenz auf einen Deskriptor eines Mandanten in die im Attribut `rd_mttenantsrelations` des Deskriptors des Basisschemas gespeicherten Liste erst nach dem Aufbau dieses Deskriptors des Mandanten abzuspeichern. Damit würde nach dem Aufbau des Deskriptors des Basisschemas gerade der Deskriptor desjenigen Mandanten nicht nochmals versucht aufzubauen. Damit würde der Aufbau nur ein Mal gestartet, jedoch vor dem des Basisschemas und daher würden dabei veraltete Informationen aus dem des Basisschemas kopiert.

Stattdessen muss dafür gesorgt werden, dass nach der Rekursion der erste Aufbauversuch in so einem Fall verworfen wird und nochmals gestart wird. Dazu ist es jedoch notwendig, dass erkannt wird, dass über die Rekursion ein erneuter Aufbau gestart wurde. Für diese Aufgabe soll das Attribut `rd_mtrebuildsactive` verwendet werden. Ohne laufenden Aufbau ist das Attribut auf null gesetzt. Wird ein Aufbau gestartet, wird das Attribut um eins erhöht, weshalb das Attribut zu Beginn des ersten Aufbaus auf eins gesetzt ist. Ist es nach dem Erhöhen größer als eins, so handelt es sich allerdings schon mindestens um den zweiten Aufbauversuch und dieser wird daher sofort abgebrochen, ohne dass das Attribut `rd_mtrebuildsactive` zurückgesetzt wird. Der erste Aufbau erkennt also zum Einen, dass er der erste ist, dadurch dass bei ihm das Attribut zu Beginn null war und durch ihn auf eins gesetzt wurde. Und zum Anderen erkennt er, dass sich das Attribut `rd_mtrebuildsactive` währenddessen durch eine Rekursion erhöht hat, also mindestens den Wert zwei hat. Es ist daher klar, dass er den letzten noch laufenden Aufbauversuch darstellt, und dass er den jetzigen verwerfen muss, um ihn nochmals starten zu können.

Durch eine einfache Erweiterung des Mechanismus lässt sich außerdem ein weiterer, ungünstiger Fall verhindern, obwohl dieser nicht zwingend verhindert werden müsste. Stellt man sich nämlich den umgekehrten Fall vor, also dass zuerst der Deskriptor des Basisschemas und dann der des Mandanten in der Liste der invaliden Deskriptoren vorkommt, dann würde während des Aufbaus des Deskriptors des Basisschemas durch die Rekursion der des Mandanten aufgebaut. Da der Aufbau desjenigen des Basisschemas noch nicht fertig ist, werden dadurch ebenfalls wieder veraltete Informationen in den des Mandanten übernommen. Nachdem die Rekursion fertig ist und der Deskriptor des Basisschemas vollends aufgebaut werden konnte, werden jedoch automatisch alle Deskriptoren der Mandanten neu aufgebaut und damit würde auch der mit den veralteten Informationen doch noch korrekt aufgebaut. Um sich den ersten fehlerhaften Aufbau in diesem Fall zu sparen, wird vor dem Aufbau zusätzlich geprüft, ob der Deskriptor des Basisschemas sich gerade im Aufbau befindet, indem geprüft wird, ob dessen Attribut `rd_mtrebuildsactive` größer als null ist, und gegebenenfalls der Aufbau sofort abgebrochen.

4.7.2.7 Freigabe des Deskriptors durch Rekursion verhindern

Ein, dem im vorherigen Abschnitt beschriebenen, ähnliches Problem tritt auf, falls der Deskriptor des Basisschemas zwei Mal in der Liste der invaliden Deskriptoren vorkommt. Sei darüber hinaus noch genau ein Deskriptor eines Mandanten offen, dessen Referenzzähler auf null steht, und der in der Liste zwischen den zwei Deskriptoren des Basisschemas vorkommt. Der Referenzzähler des Deskriptors des Basisschemas ist in diesem Fall mindestens auf eins gesetzt, daher wird nach dem Abbau mit dem erneuten Aufbau begonnen. Durch die Rekursion wird zuerst der Deskriptor des Mandanten abgebaut und nicht mehr aufgebaut, da als Referenzzähler null angenommen wurde. Dadurch wurde der Referenzzähler des Deskriptors des Basisschemas dekrementiert, kann nun also insbesondere ebenfalls null sein. Wird nun durch die Rekursion als nächstes mit dessen Abbau begonnen, so erfolgt auch hier kein Aufbau mehr. Wenn allerdings kein Aufbau mehr geplant ist, dann muss der Speicherplatz des Deskriptors freigegeben werden. Kehrt die Rekursion nun zum ersten Versuch, den Deskriptor des Basisschemas aufzubauen, zurück, der immer noch eine Referenz auf den freigegeben Speicherplatz besitzt, so wird das Backend beim ersten Zugriff darauf abstürzen.

Es muss also verhindert werden, dass der Referenzzähler eines Deskriptors null werden kann, während die Liste der invaliden Deskriptoren abgearbeitet wird, damit durch eine Rekursion kein Deskriptor freigegeben werden kann, der nicht schon beim ersten Versuch freigegeben wurde. Für jeden Eintrag in der Liste wird daher wie folgt vorgegangen. Es wird zuerst der Referenzzähler betrachtet, ist er null, wird der Deskriptor nach dem Abbau freigegeben. Sollte jedoch ein erneuter Aufbau notwendig sein, wird der Deskriptor nicht freigegeben, sondern stattdessen der Referenzzähler vor dem Aufbau um eins erhöht, dann der Aufbau vorgenommen, und danach wieder dekrementiert. Somit wird verhindert, dass während eines Aufbaus der Referenzzähler durch eine Rekursion auf null gesetzt und der Deskriptor dadurch freigegeben werden kann.

4.7.3 Invalidierung

Wie schon im Abschnitt 3.8 erklärt wurde, muss bei einem veränderten Eintrag im Systemkatalog der Relationen Deskriptor, der diesen Eintrag beim Aufbau möglicherweise gelesen hat, und der Eintrag im Cache dieses Systemkatalogs gelöscht werden. Dazu ist es notwendig, dass Änderungen am Systemkatalog erkannt werden und diese dem Cache des Systemkatalogs und dem Cache der Relationen Deskriptoren mitgeteilt wurden, damit diese die betreffenden Einträge als invalide markieren können, indem sie in eine Liste aufgenommen werden. Diese Aufgabe wird von einem eigenständigen Modul übernommen.

Durch die für die isolierten Tabellen notwendige Aufspaltung der Relationen, werden auch in diesem Modul einige Änderungen notwendig. Zum Einen müssen die Parameter einiger Funktionen und einige Datenstrukturen um eine Variable für die ID des Mandanten erweitert werden, damit eine einzelne Relation eines Mandanten identifiert und damit allein der

dafür aufgebaute Deskriptor invalidert werden kann. Des Weiteren gibt es mit dem neuen Systemkatalog `pg_mtclass` eine weitere Quelle von Einträgen, bei deren Änderung der Relationen Deskriptor invalidiert werden muss. Es muss daher auch dieser Systemkatalog auf Änderungen überwacht werden und der Relationen Deskriptor bestimmt werden, der auf Informationen des Eintrags basiert.

Die Änderungen an einem Systemkatalog werden erkannt, indem bei jeder Aktualisierung oder Einfügeoperation eines Tupels die Funktion PrepareForTupleInvalidation aufgerufen wird. Als Parameter bekommt die Funktion die ID der Tabelle und das Tupel übergeben.

Die Funktion beginnt damit, die Liste der Caches der Systemkataloge durchzugehen und prüft dabei, ob einer davon Einträge aus der Tabelle enthält, in die das Tupel geschrieben werden soll. Für jeden gefundenen Cache, wird nach einem Eintrag im Cache gesucht und gegebenenfalls in die Liste der invaliden Tupel eingefügt. Da der für den Systemkatalog `pg_mtclass` angelegte Cache ebenfalls in dieser Liste ist, werden diese Einträge also automatisch als invalide markiert.

Danach wird zuerst geprüft, ob der Eintrag überhaupt relevant für einen Relationen Deskriptor sein kann. Das ist nur für Einträge aus `pg_attribute`, `pg_index`, `pg_class` und nun auch für Einträge aus `pg_mtclass` der Fall. Für jeden Systemkatalog muss danach anders vorgegangen werden, um herauszufinden, zu welcher Relation dieser Eintrag gehört. Daher muss dies für den Fall, dass es sich um einen Eintrag aus `pg_mtclass` handelt, noch entsprechender Code hinzugefügt werden. Da sowohl die Referenz auf den Eintrag auf `pg_class` als auch die ID des Mandanten in jedem Eintrag in `pg_mtclass` vorhanden ist, können diese direkt ausgelesen werden, wodurch der zu invalidierende Deskriptor eindeutig bestimmt ist.

4.8 Mandanteneigene Attribute

Dieser und folgende Abschnitte befassen sich mit den Attributen, die ein Mandant einer isolierten Tabelle mittels dem Befehl `ALTER TABLE` hinzufügen kann. Ein solche Tabelle wird vom Administrator angelegt und enthält daher zunächst eine Reihe von vordefinierten Attributen für alle Mandanten. Danach kann es in abwechselnder Reihenfolge vorkommen, dass ein Mandant ein eigenes Attribut hinzufügt, und dass der Administrator ein neues Attribut für alle Mandanten hinzufügen möchte. Dabei entstehen sowohl bezüglich des physikalischen als auch des logischen Schemas eigene Sichten auf die Attribute. Diese Sichten definieren sich unter Anderem durch die Reihenfolge der Attribute, die geklärt werden muss.

Des Weiteren werden einzelne Attribute in diesen Sichten während der Laufzeit häufig nur mit Attributnummern identifiziert. Diese werden von sehr vielen Modulen direkt benutzt, also ohne eine Schnittstelle, an der Anpassungen möglich wären. Deswegen muss für jedes Attribut, also für die des Mandanten und für die des Basisschemas, eine eindeutige Nummer

vergeben werden, was sich in den Folgenden Abschnitten als etwas schwieriger herausstellen wird.

4.8.1 Reihenfolge der Attribute im physikalischen Schema

Das physikalische Schema gibt an, wie die Werte aller Attribute auf dem externen Speicher abgelegt werden sollen. Da die Werte eines Tupels in einem für das jeweilige Tupel reservierten, zusammenhängenden Bereich gespeichert werden, muss das physikalische Schema bei PostgreSQL insbesondere die Reihenfolge vorgeben, in der die Werte abgelegt werden sollen.

Die Sicht auf diese Metainformationen muss natürlich schon deshalb für jeden Mandanten eine eigene sein, da ein Mandant eigene Attribute hinzufügen kann. Es bleibt jedoch zu klären, welche Reihenfolge der Attribute verwendet werden muss. Es stellt sich heraus, dass nur eine mögliche Reihenfolge in Frage kommt. Ein neues, anzulegendes Attribut, egal ob vom Mandanten oder vom Administrator, kann insbesondere `NULL` erlauben. In diesem Fall wird PostgreSQL, und vermutlich viele andere Datenbanksysteme ebenfalls, kein einziges Tupel auf dem externen Speicher anrühren. Stattdessen wird das neue Attribut bzgl. der Reihenfolge des physikalischen Schemas als letztes Attribut angefügt. Wird dann später ein Tupel vom externen Speicher geladen, so verrät einem der im `HeapTuple`-Struct gespeicherte Header die Anzahl der abgespeicherten Attribute. Die Information, dass das neue Attribut `NULL` ist, fehlt hingegen. PostgreSQL setzt dann alle Attribute, die bezüglich der Reihenfolge nach dieser Anzahl folgen, und damit insbesondere das zuletzt angefügte, während der Laufzeit auf `NULL`.

Betrachtet man das Hinzufügen der Attribute als zeitliche Ereignisse, so muss die Reihenfolge der im physikalischen Schema eines Mandanten daher der chronologischen Reihenfolge entsprechen.

4.8.2 Attributnummern für das physikalische Schema

Jeder Eintrag im Systemkatalog `pg_attribute` bekommt eine Attributnummer, durch die die Reihenfolge festgehalten wird, und durch die ein Attribut zur Laufzeit identifiziert wird. Bisher wurde dafür eine fortlaufende Nummer beginnend bei eins vergeben. Gelöschte Einträge werden lediglich als gelöscht markiert, wodurch Attributnummern nie frei werden können. Damit entsprach die Vergabe der Attributnummern exakt der Reihenfolge, in der die Attribute im physikalischen Schema vorkommen müssen. Dadurch waren keine weiteren Informationen notwendig, um die Reihenfolge im physikalischen Schema abzuspeichern.

Die Attribute der isolierten Tabellen werden ebenfalls in diesem Systemkatalog abgespeichert und deshalb müssen für sie ebenfalls Attributnummern vergeben werden. Bei den von den Mandanten angelegten Attributen stellt das kein Problem dar, für sie kann problemlos

die aus Sicht des Mandanten korrekte Attributnummer vergeben werden, die noch nicht für ein anderes Attribut von ihm vergeben wurde, da sie durch die ID des Mandanten im Eintrag nicht mit denen anderer kollidieren können. Bei den Attributen des Basisschemas allerdings kann es zum Problem werden, dass nur eine Attributnummer vergeben werden kann und somit an dieser Stelle nicht auf die Sicht des Mandanten darauf eingegangen werden kann.

Damit können die Attributnummern im Systemkatalog nicht so vergeben werden, dass sie bei allen Mandanten mit ihrem physikalischen Schema übereinstimmen. Dabei könnte es sonst passieren, dass einer der Mandanten seit dem Erstellen zwei eigene Attribute und ein anderer Mandant drei Attribute angelegt hat, und daher für ein neues Attribut des Basisschemas aus Sicht des ersten Mandanten eine um eins kleinere Attributnummer vergeben werden müsste als für den zweiten.

4.8.3 Reihenfolge der Attribute im logischen Schema

Das logische Schema beschreibt die Sicht des Anwenders auf die angelegten Objekte, also insbesondere auf die Attribute einer isolierten Tabelle. Beim Transfer bzw. der Ausgabe von Tupeln aus einer Tabelle wird bei PostgreSQL eine im logischen Schema dokumentierte Reihenfolge verwendet, um die Attribute im Tupel zu ordnen. Diese Reihenfolge ist prinzipiell unabhängig von der Reihenfolge der Attribute im physikalischen Schema und könnte daher frei gewählt werden. Dennoch wird hier ebenfalls die chronologische Reihenfolge verwendet, damit beim Laden eines Tupels die Umordnung der nach dem physikalischen Schema gespeicherten Attribute in die Reihenfolge des logischen Schemas entfällt.

Diese Vorgehensweise ist zwar bei vielen Datenbanksystemen üblich, jedoch besagt der SQL-Standard, dass die Reihenfolge der Attribute eines Tupels vollkommen willkürlich sein darf. Stattdessen wird in der Regel beim Transfer bzw. der Ausgabe für jede Spalte der Name des Attributs angegeben und die Anwendung bzw. der Benutzer die Reihenfolge daher selbst bestimmen kann.

Obwohl die Reihenfolge demnach frei gewählt werden darf, haben viele Entwickler sich daran gewöhnt, dass die Reihenfolge der chronologischen entspricht und vergleichen daher häufig nicht die Namen der Spalten mit der, die sie erwarten würden, sondern greifen häufig nur auf die Spalte mit der Nummer zu, in der sie das Attribut auf Grund der chronologischen Reihenfolge erwarten. Daher ist dennoch wichtig, dass versucht wird, die Kompatibilität mit solcher Software zu erhalten.

Um ein vernünftiges Maß an Kompatibilität zu gewährleisten, das mit akzeptablem Aufwand machbar ist, wird bei isolierten Tabellen sichergestellt, dass eine Anwendung, die nur auf die Attribute des Basisschemas zugreifen möchte, darauf zugreifen kann, als wären keine mandanteneigenen Attribute vorhanden. Das wird erreicht, indem alle Attribute des Basisschemas vor denen der Mandanten gesetzt werden und damit unabhängig von der Ta-

belle des Mandanten immer an der gleichen Stelle steht. Andererseits wird erwartet, dass eine Software, die so dynamisch bzw. angepasst werden kann, dass sie mit neuen Spalten eines Mandant umgehen kann, dass diese nicht nur blind auf eine Spalte zugreifen kann, sondern die Namen korrekt auswertet.

Wenngleich dieses Vorgehen ungewohnt und damit unerwartet sein mag, hat es einen Vorteil. Falls eine Software nur per Position der Spalte auf ein Attribut zugreift, wäre es möglich, dass ein Mandant eigene Attribute anlegt, bevor durch ein Update der Software weitere von ihr genutzte Attribute für alle Mandanten angelegt werden. Damit würde die Software bei der rein chronologischen Reihenfolge auf ein Attribut des Mandanten zugreifen statt auf das eigene. Damit hätte es der Mandant geschafft, das Attribut durch sein eigenes zu ersetzen. Da bei einer Datenbankanwendung im Datenbanksystem modelliert wird, welches valide Daten für die Anwendung sind, also insbesondere eine Bedingung für das ursprüngliche Attribut angegeben worden sein kann, welche der Mandant für seines nicht angegeben hat, werden damit bei der Entwicklung der Anwendung getroffene Annahmen außer Kraft gesetzt und daher stellt diese Möglichkeit ein Risiko bezüglich der Sicherheit und Stabilität der Anwendung dar. Der Vorteil ist demnach, dass so etwas bei dem angestrebten Maß an Kompatibilität nicht möglich ist.

4.8.4 Attributnummern für das logischen Schema

Der Systemkatalog `pg_attribute` enthält eine Spalte für die Attributnummer jedes darin gespeicherten Attributs. Wie schon erwähnt wurde, werden diese Nummern üblicherweise entsprechend der Reihenfolge des physikalischen Schemas vergeben. Da die Reihenfolge der Attribute des logischen Schemas ebenfalls chronologisch ist, entspricht sie jedoch ebenso der des logischen Schemas und daher wurden auch hierfür keine weiteren Informationen als der Attributnummer benötigt.

Das logische Schema stellt während der Laufzeit eine der wichtigsten Informationsquellen dar, die für eine Relation gebraucht werden. Es wird daher unter Anderem im Parser, Planer und im Executor an sehr vielen Stellen benutzt. Manchmal wird hierfür der Tupel Deskriptor benutzt, häufig jedoch wird aber auch nur die Referenz auf die Relation und die Attributnummer übergeben. Daher wird ein Attribut des logischen Schemas während der Laufzeit häufig nur über die Attributnummer des Systemkatalogs identifiziert. Will eine Funktion daher mehr Informationen über das Attribut erhalten, so konnte es diese über die Nummer direkt im Systemkatalog nachschlagen.

Das führt zu einer weiteren Einschränkung der Möglichkeiten, die isolierten Tabellen zu implementieren. Obwohl im vorherigen Abschnitt erklärt wurde, warum es einen gewissen Spielraum bei der Reihenfolge der Attribute im logischen Schema gibt, wäre es erstrebenswert die volle Kompatibilität bieten zu können, also die chronologische zu verwenden. Bei den Attributen des Basisschemas würde dann allerdings die Nummer des logischen Sche-

mas nicht mit der im Systemkatalog übereinstimmen und es würde sehr schwierig werden, den Eintrag zu finden.

4.8.5 Attributnummern und Reihenfolge bei isolierten Tabellen

Wie in den vorherigen Abschnitten erläutert wurde, müssen für isolierte Tabellen eine neue Methode gefunden werden, bei der die Reihenfolge der Attribute bezüglich des physikalischen Schemas weiterhin der chronologischen entspricht, die Reihenfolge der Attribute des logischen Schemas eine gewisse Kompabilität bietet und die beim logischen Schema verwendeten Attributnummern denen des Systemkatalogs entsprechen. Es ist dabei insbesondere nicht mehr möglich, dass für beide Schemata die gleiche Reihenfolge verwendet wird.

4.8.5.1 Vergabe der Attributnummern im Systemkatalog bei isolierten Tabellen

Die Attributnummern werden fortlaufend ab der eins vergeben. Die Attribute des Basisschemas sollen die vorderen Attributnummern bekommen und die der Mandanten die nachfolgenden. Beim Hinzufügen eines Attributs eines Mandanten braucht daher lediglich ein neuer Eintrag mit einer um eins größeren Attributnummer als die Anzahl der bisherigen Attribute aus Sicht des Mandanten eingetragen werden. Falls dagegen ein Attribut des Basisschemas hinzugefügt werden muss, müssen die Attributnummern aller Einträge der Mandanten um eins erhöht werden und die dadurch freigewordene Attributnummer verwendet werden. Es gäbe zwar auch eine etwas komplexere Möglichkeit einige Attributnummern dazwischen für so einen Fall zu reservieren, dadurch würde allerdings der Speicherverbrauch höher werden, da die Attributnummern als Indizes einiger Arrays verwendet werden.

Da die Reihenfolge bezüglich der Attributnummern daher nicht mehr der chronologischen Reihenfolge entspricht, benötigt der Systemkatalog `pg_attribute` ein zusätzliches Attribut `attmtlatestbaseatt`. Es wurde im Abschnitt 4.1.3 eingeführt und gibt bei den mandanteneigenen Attributen das Attribut des Basisschemas an, das bezüglich der chronologischen Reihenfolge direkt davor steht.

4.8.5.2 Logisches Schema bei isolierten Tabellen

Wie im Abschnitt 4.8.4 ausgeführt wurde, können zur Laufzeit Probleme auftreten, falls man eine andere Reihenfolge als die der Attributnummern im Systemkatalog verwendet. Daher richtet sich die bei isolierten Tabellen verwendete Reihenfolge exakt nach der des Systemkatalogs. Es kommen demnach zuerst alle Attribute des Basisschemas und dann alle des Mandanten.

4.8.5.3 Physikalisches Schema wiederherstellen

Um den zuvor erläuterten Problemen zu begegnen, wurde zur Abspeicherung der Attribute im Systemkatalog eine Reihenfolge gewählt, die nicht der chronologischen entspricht. Daher muss beim Einlesen des physikalischen Schemas aus dem Systemkatalog, das beim Aufbau des Tupel Deskriptors erfolgt, die korrekte Reihenfolge wiederhergestellt werden.

Die Reihenfolge wurde im Tupel Deskriptor bisher durch ein Array vorgegeben, dessen gespeicherte Werte jeweils für ein Attribut die Informationen beider Schemata darstellen. Da dadurch also auch die Reihenfolge der Attriubte des logischen Schemas dargestellt wird, kann die Reihenfolge im Array nicht verändert werden. Stattdessen wird eine neue Datenstruktur benötigt, über die auf die Informationen des physikalischen Schemas in chronologischer Reihenfolge zugegriffen werden kann. Dazu bietet sich eine Permutation an. Der Tupel Deskriptor muss daher, wie in Ausschnitt 4.8 zu sehen ist, modifiziert werden. Er enthält außerdem die inverse Permutation, damit darüber die aus einem `HeapTuple`-Struct extrahierten Attribute schnell in das logische Schema überführt werden können.

```
1 typedef struct tupleDesc {
2     ...
3   int2             *mtattnummapping;
4                   /* mapping between runtime attnums and attnums
                        in HeapTuples */
5   int2             *reverseattmap;
6                   /* reverse of mtattnummapping */
7     ...
8 }       *TupleDesc;
```

Ausschnitt 4.8: Zusätzliche Attribute für den Tupel Deskriptor

Beim Einlesen ist daher ein Algorithmus notwendig, der aus den Einträgen des Systemkatalogs die Permutation herstellt. Da sowohl die Attribute der Mandanten als auch die des Basisschemas für sich in chronologischer Reihenfolge eingelesen werden können, reicht dazu ein einfaches MergeSort aus, bei dem die Sortierung eben nicht auf dem Array sondern auf der Permutation stattfindet. Die Sortierung erfolgt dabei über das Attribut `attmtlatestbaseatt`. Daraus kann die inverse ohne weitere Probleme direkt berechnet werden.

In Abbildung 4.4 ist ein Beispiel zu sehen. Es basiert auf dem eingangs im Konzept erwähnten Beispiel, bei dem ein Mandant eine zusätzliche Spalte `Tax` angelegt hat. Nachdem er diese Spalte angelegt hat, soll der Administrator die Spalte `Price` im Basisschema angelegt haben. Daher ist das Attribut `attmtlatestbaseatt` der Spalte `Tax` auf eins gesetzt. Beim Anlegen des Attributs durch den Administrator wurde die Attributnummer des Attributs `Tax` von zwei auf drei erhöht. Auf der rechten Seite ist ein Tupel Deskriptor des Mandanten zu sehen, bei dem das logische Schema durch das Array `attrs` dargestellt wird. Man sieht außerdem, dass für das physikalische Schema die Attribute `Price` und `Tax` durch die in `mtattnummapping` gespeicherte Permutation vertauscht wird.

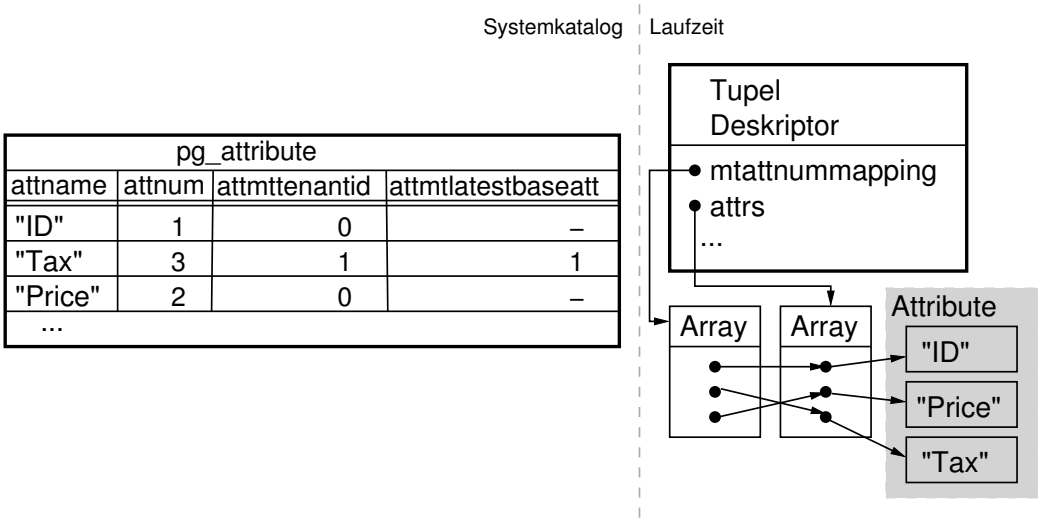

Abbildung 4.4: Physikalisches Schema im Tupel Deskriptor eines Mandanten

4.8.5.4 Zugriff auf Attribute in einem HeapTuple-Struct

Wohingegen bei den Zugriffen auf das logische Schema keine Anpassungen notwendig sind, muss bei Zugriffen auf das physikalische Schema die Permutation verwendet werden. Solche Zugriffe sind dann notwendig, sobald man auf ein Attribut in einem `HeapTuple`-Struct zugreifen muss. Dafür gibt es glücklicherweise eine Schnittstelle zu einem Modul, so dass es genügt, dieses Modul anzupassen.

Da es bei Tabellen, die keine Permutation besitzen, keinen Sinn macht diese extra abzuspeichern, ist der Zeiger bei solchen Tabellen auf `NULL` gesetzt. Daher werden die beiden Makros in Ausschnitt 4.9 eingeführt. Sie erwarten den Tupel Deskriptor und die um eins niedrigere Attributnummer als Parameter. Sofern eine Permutation im Tupel Deskriptor vorhanden ist, verwenden sie diese, um die korrekte Attributnummer zu bestimmen. Andernfalls kann die Attributnummer direkt zurückgegeben werden.

```
1 #define attmap(tupleDesc, attnum) \
2   ((tupleDesc)->mtattnummapping!=NULL ? \
3     (((tupleDesc)->mtattnummapping[attnum])) \
4     : ((attnum)+1))
5 #define reverseattmap(tupleDesc, attnum) \
6   ((tupleDesc)->reverseattmap!=NULL ?
7       (((tupleDesc)->reverseattmap[attnum])) \
8       : ((attnum)+1))
```

Ausschnitt 4.9: Umrechnung der Attributnummern zwischen den zwei Schemata

Damit braucht man im Wesentlichen nur bei jeder Verwendung einer Attributnummer in diesem Modul das jeweilige Makro anzugeben, falls ein Wechsel von logischem zu physikalischem Schema oder andersherum notwendig ist. Da die Schnittstelle nach außen hin allerdings immer die Attributnummern des logischen Schemas verwendet, müssen darüber hinaus für einige Funktionen, die von dem Modul selbst aufgerufen werden und bei der demnach auch eine Attributnummer des physikalischen Schemas angegeben werden kann, eine zweite Version angelegt werden, die die physikalische Attributnummer statt der logischen erwartet. Bei dieser Version fällt dann lediglich der Aufruf des Makros weg, das die logische sonst in die physikalische überführen würde.

4.8.6 Offsetcache beim Zugriff auf Attribute

Im Abschnitt 3.6 wurde der Tupel Deskriptor erklärt. Insbesondere wurde dort auch daraufhingewiesen, dass die Offsets, an denen der Wert eines Attributs im Datenbereich des `HeapTuple`-Structs beginnt, unter gewissen Umständen in einen im Tupel Deskriptor befindlichen Cache geschrieben werden kann, damit diese beim nächsten Zugriff nicht erneut berechnet werden müssen.

Leider wird dieser Cache auf eine fragwürdige Art und Weise in PostgreSQL realisiert. Da für jedes Attribut ein Offset abgespeichert werden können muss, wurde dafür im Systemkatalog ein Attribut `attcacheoff` angelegt, das zur Laufzeit überschrieben wird, sobald das Offset das erste Mal berechnet wurde, anstatt dass die Variablen erst zur Laufzeit angelegt werden.

Diese Tatsache wirft bei den isolierten Tabellen eines neue Frage auf. Da auf die Einträge der Attribute des Basisschemas aus dem Systemkatalog von mehreren Tupel Deskriptoren verwiesen wird, greifen damit alle Deskriptoren auf das selbe Offset zu. Im Abschnitt 4.8.2 wurde jedoch schon dargelegt, warum aus Sicht eines Mandanten im physikalischen Schema unterschiedlich viele Attribute vor einem Attribut stehen können. Daher kann sich das Offset eines solchen Attributs von Mandant zu Mandant unterscheiden.

4.8.6.1 Neuer Offsetcache im Tupel Deskriptor

Offensichtlich muss der Offsetcache auf eine andere Art und Weise gespeichert werden. Da es keinen triftigen Grund gibt, den Cache im Eintrag des Attributs zu behalten, kann dieser stattdessen direkt im Deskriptor angelegt werden. Die Erweiterung ist in Ausschnitt 4.10 zu sehen. Es gibt lediglich ein neues Attribut `attcacheoff` im Deskriptor, dass auf ein Array mit einer der Anzahl der Attribute entsprechenden Größe zeigt. Im Gegensatz zu der Permutation `mtattnummapping` und der Inversen soll bei jedem Typ von Tabellen dieses Array verwendet werden. Das entsprechende Attribut im Systemkatalog wird damit überflüssig.

```
typedef struct tupleDesc {
   ...
  int2             *attcacheoff; /* offset cache */
   ...
}       *TupleDesc;
```

Ausschnitt 4.10: Weitere Attribute für den Relationen Deskriptor

Der somit erweiterte Tupel Deskriptor erlaubt es jedem Mandanten, der eigene Attribute besitzt, eigene Offsets im eigenen Deskriptor abzuspeichern. Andererseits kann ein Mandant der keinen eigenen Tupel Deskriptor benötigt, da er keine eigenen Attribute hat, weiterhin den Deskriptor des Basisschemas verwenden, da die Offsets in diesem Fall ohnehin gleich sein müssen.

4.8.6.2 Neuen Offsetcache verwenden

Ein Offset aus einem Cache wird immer dann gebraucht, wenn ein Zugriff auf ein Attribut im `HeapTuple`-Struct erfolgen soll. Wie schon zuvor erwähnt wurde, werden diese Zugriffe von einem einzelnen Modul ausgeführt, so dass auch hier im Großen und Ganzen nur dieses eine Modul angepasst werden muss.

Da der Deskriptor an allen Stellen des Moduls direkt verfügbar ist, kann jede Verwendung der Variabe `attcacheoff` eines im Tupel Deskriptor gespeicherten Eintrag eines Attributs leicht umgeschrieben werden, damit stattdessen die entsprechende Variable im Array des Tupel Deskriptors genommen wird. Zwei weitere Stellen im Quellcode, die neue Attributeeinträge in einem Deskriptor anlegen und dementsprechend das Offset korrekt initialisieren müssen, können dank vorhandem Deskriptor ebenfalls leicht angepasst werden.

4.8.7 Plancache

Für eine Anfrage an das Datenbanksystem, deren genaue Ausführung unbestimmt ist, muss das Datenbanksystem selbständig einen Plan erstellen, der vom Executor ausgeführt werden kann. Dieser Plan wird in einem Cache abgespeichert, der dementsprechend Plancache genannt wird. Darüber hinaus ist es über die SQL-Anweisungen Prepare und Execute möglich, den selben Plan für mehrere gleich strukturierte Anfragen zu verwenden.

Im Zusammenhang mit isolierten Tabellen bietet sich daher die Möglichkeit, einen Plan zu erstellen, den mehrere Mandanten verwenden können. Das hätte natürlich zum Einen den Vorteil, dass man den Speicherplatz spart, den mehrere Pläne benötigen würden. Zum Anderen ist der Aufbau eines Plans sehr kostspielig, da auf Basis der Statistiken versucht wird, unter allen möglichen Plänen den optimalen zu finden, dessen Ausführungszeit am geringsten eingeschätzt wird.

Die Umsetzung dieser Idee war nicht Teil der Aufgabenstellung dieser Arbeit und deshalb gab es kein Konzept dafür, da der Aufwand der anderen Punkte schon ausgereicht hat. Dennoch soll an dieser Stelle festgehalten werden, was man dabei beachten müsste, insbesondere welche Probleme sich dabei auftun.

Zum Beispiel muss dabei beachtet werden, dass die von einem Mandanten in seiner Tabelle abgespeicherten Werte eine vollkommen andere Verteilung wie die der anderen Mandanten aufweisen können. Daher wären die Statistiken, auf Basis derer ein Plan erstellt wird, andere und deshalb könnte ein anderer Plan viel besser für den Mandanten sein. Somit würde es notwendig werden, dass für einen Mandant trotzdem ein eigener Plan erstellt wird. Ob er einen benötigt, könnte dadurch entschieden werden, ob seine Statistiken zu stark abweichen oder besser noch, ob die ursprünglich berechneten Kosten des Plans zu stark von denen abweichen, die berechnet werden, falls seine Statistiken verwendet werden.

Außerdem stellt sich die Frage, wie mit mandanteneigenen Attributen umgegangen werden soll. Werden in der Anfrage Attribute eines Mandanten verwendet, ist natürlich ohnehin schon klar, dass sie sich nicht für eine gemeinsame Nutzung mittels Prepare und Execute eignet. Aber auch ohne dass ein mandanteneigenes Attribut explizit in der Anfrage vorkommt, kann der aus der Anfrage resultierende Plan trotzdem welche enthalten. Beispielsweise wird das nicht selten bei einer SELECT-Abfrage verwendete Asterisk noch vor dem Erstellen des Plans durch den Parser zur vollständigen Liste aller Attribute der Tabelle erweitert. Zwar dürfte diese Tatsache die Struktur des Plans kaum beeinflussen, da die meisten Abschätzungen der Kosten, die dadurch mitaufgenommenen mandanteneignen Attribute ignorieren würden, da sie meist nur durchgereicht werden müssen. Allerdings würde der Plan die vollständige Liste der Attribute enthalten und unterscheidet sich daher dennoch von denen anderer Mandanten.

4.8.8 Verwendung einer Tabelle als Typ

Da PostgreSQL ein objektrelationales Datenbanksystem ist, ergibt sich durch die mandanteneigenen Attribute eine Besonderheit. Denn als solches wird durch jede Tabelle eine Klasse definiert, die wiederum einen Datentyp darstellt. Ein derart definierter Datentyp kann für ein Attribut einer anderen Tabelle verwendet werden.

Bei isolierten Tabellen definiert sich dieser Datentyp für jeden Mandanten aus teilweise unterschiedlichen Attributen. Das bedeutet demnach, dass jeder Mandant auf die den Datentyp verwendende Tabelle wieder eine eigene Sicht darauf benötigten. Handelt es sich bei dieser Tabelle um eine isolierte, wäre das in Ordnung. Bei anderen Tabellen hingegen ist eine eigene Sicht nicht vorgesehen und daher führt eine Verwendung des Datentyps zwangsweise zu Problemen.

4.9 Hierarchisches Locking

Vor jedem Zugriff auf Daten einer Relation, muss die Relation geöffnet werden. Dabei muss die Relation für andere Backends gesperrt werden, damit diese keine Operationen auf dieser Relation ausführen können, die mit dem eigenen Zugriff auf die Daten in Konflikt stehen. Andererseits gibt es durchaus Operationen, wie beispielsweise das Lesen von Daten von mehreren Backends, die nicht in Konflikt zueinander stehen. Daher gibt es für jede Klasse von Operationen einen Modus, mit dem man die Relation vor der Ausführung der Operation sperren muss. Je nachdem, ob eine Sperre aktiv ist, die mit diesem Modus in Konflikt steht oder nicht, wird die Sperre gesetzt.

Zusätzlich zu den Sperren auf Relationen werden an einigen Stellen welche auf auf anderen Objekten benötigt, wie beispielsweise auf Tupeln oder Speicherseiten. Daher wird die Verwaltung der Sperren aller unterstützter Objekte von einem eigenständigen Modul übernommen, dem Lock Manager. Es merkt sich sowohl alle aktiven Sperren, damit Konflikte mit neuen entdeckt werden, als auch alle Sperren, auf die noch gewartet werden muss, da sie bisher auf Grund eines Konflikts noch nicht aktiv sind.

Eine Sperre auf einer Relation wird vom Lock Manager durch eine Referenz auf den Eintrag im Systemkatalog `pg_class` identifiert. Daher würde ein Mandant, der nur auf seine eigene Daten einer isolierten Tabelle zugreifen, automatisch alle Relationen der anderen Mandanten sperren. Bei mehreren tausend Mandanten würden die Sperren das System dadurch viel zu sehr ausbremsen, weshalb die Identifikation der Sperren zusätzlich die ID des Mandanten berücksichtigen muss.

Andererseits müssen einige vom Administrator ausgeführte Operationen auf den Daten aller Mandanten arbeiten können. Zum Beispiel müssen beim Anlegen eines gemeinsamen Index auf einer isolierten Tabelle die Tupel aller Mandanten vom Administrator gelesen werden. Durch die Verwendung der ID des Mandanten müsste daher zu Beginn der Operation jede Relation der Mandanten einzeln gesperrt werden. Um das zu vermeiden, muss ein neuer Mechanismus eingeführt werden, der im Grunde ein Hierarchisches Locking ermöglicht und der in den folgenden Abschnitten erläutert wird. Dessen Idee wird es sein, bei der Relation des Basisschemas zwischen Sperren von Mandanten und denen des Administrators zu unterscheiden. Dann wird für ein Relation eines Mandanten zusätzlich die Relation des Basisschemas gesperrt, allerdings so, dass diese Sperre nicht mit denen anderer Mandanten in Konflikt steht, sondern nur mit einer Sperre vom Administrator.

4.9.1 Neue Sperr-Modi

Die eingangs erwähnten Klassen von Anfragen werden so gewählt, dass alle zu einer Klasse zusammengefassten Anfragen genau mit der gleichen Menge von Anfragen, und nur mit dieser, in Konflikt stehen. Dadurch wird für jede Klasse genau ein Modus benötigt, in dem die Sperre erfolgt. Die somit minimale Anzahl an Modi beläuft sich bei PostgreSQL auf

acht Stück. Sie werden im Quelltext als Enum dargestellt, also C-typisch durch fortlaufende ab null beginnende Zahlen, anstelle derer im Quelltext ein Makro verwendet wird.

Wie schon im vorherigen Abschnitt erwähnt wurde, ist es die Idee, zwischen zwei Sperren zu unterscheiden. Eine neue Sperre auf der Relation des Basisschemas soll es mehreren Mandanten erlauben, auf ihren Daten zu arbeiten, gleichzeitig soll sie jedoch mit der normalen Sperre im Konflikt stehen, die ein Administrator setzt, falls er auf den Daten aller Mandanten arbeiten will. Offensichtlich müssen sich diese Sperren daher im Modus unterscheiden, weshalb neue Modi für Mandanten notwendig sind.

Eine Möglichkeit wäre es, genau einen neuen Modus für den Mandanten anzulegen. Falls nun ein Mandant eine Sperre für seine Relation setzt, die mit allen anderen bisher definierten Modi in Konflikt steht, der Mandant also demnach den exklusiven Zugriff wünscht, dann würde er diesen neuen Modus auch für die Relation des Basisschemas setzen. Damit der Administrator den exklusiven Zugriff des Mandanten respektiert, müsste der neue Modus daher mit allen vom Administrator verwendeten Modi, also den alten, in Konflikt stehen. Da der Mandant beim einfachen Lesen ebenfalls nur diesen einen neuen Modus verwenden kann, könnte damit ein Konflikt mit einem gleichzeitigen Leseversuch des Administrators auftreten, obwohl dagegen eigentlich gar nichts sprechen würde.

Da sich auf die zuvor genannte Art und Weise immer ein Negativbeispiel finden lässt, sofern versucht wird, die Modi der Mandanten ausgehend von den bisher definierten Modi zusammenzufassen, müssen alle bisher definierten Modi eine Version erhalten, die von den Mandanten genutzt werden können. Beispielsweise gibt es für den Lesezugriff somit zusätzlich zur Makrodefinition `#define AccessShareLock 1` eine `#define MTAccessShareLock 9`. Insbesondere ist der Abstand zwischen den Versionen immer exakt acht.

Der interessanteste Teil sowohl dieser neuen Versionen der Modi als auch der alten sind die Konflikte. Diese müssen im Quellcode repräsentiert werden. Offenbar handelt es sich bei dem Prädikat ïn Konflikt stehenüm eine symmetrische Relation im rein mathematischen Sinn. Eine Relation, die wiederum eine Menge ist, kann natürlich als Liste repräsentiert werden. Da jedoch schnell geprüft werden muss, ob ein Tupel enthalten ist, wird stattdessen eine Bit-Matrix genommen. Eine als Array von Bit-Vektoren realisierte Bit-Matrix ist in Ausschnitt 4.11 zu sehen. In ihr ist für jeden Konflikt zwischen zwei bisherigen Modi ein neuer zwischen dem bisherigen und der neuen Version angelegt worden. Der Symmetrie wegen steht ein Konflikt jeweils in den Zeilen der bisherigen Modi und einer in der der neuen Versionen.

```
1 static const LOCKMASK
2 LockConflicts[] = {
3 ...
4   /* AccessShareLock */
5   (1 << AccessExclusiveLock) |
6   (1 << MTAccessExclusiveLock),
7
```

```
8    /* RowExclusiveLock */
9    (1 << ShareLock)        | (1 << ShareRowExclusiveLock) |
10   (1 << ExclusiveLock)    | (1 << AccessExclusiveLock) |
11   (1 << MTShareLock)      | (1 << MTShareRowExclusiveLock) |
12   (1 << MTExclusiveLock) | (1 << MTAccessExclusiveLock),
13   ...
14   /* MTAccessShareLock */
15   (1 << AccessExclusiveLock),
16
17   /* MTRowExclusiveLock */
18   (1 << ShareLock)      | (1 << ShareRowExclusiveLock) |
19   (1 << ExclusiveLock) | (1 << AccessExclusiveLock),
20   ...
21 }
```

Ausschnitt 4.11: Definition der Konflikte zwischen den Sperr-Modi

4.9.2 Anpassungen an Schnittstelle des Lock Managers

Als eigenständiges Modul bietet der Lock Manager gegenüber anderen Modulen eine Schnittstelle an. Zum Beispiel gibt es für jedes Objekt, das gesperrt werden kann, eine Funktion, die die Sperre setzt, und eine, die die Sperre wieder entfernt. Diese Funktionen müssen jeweils um eine ID eines Mandanten als Parameter ergänzt werden, damit die mandanteneigenen Objekte ohne Einfluss auf andere Objekte gesperrt werden können.

Es bleibt die Frage, wo die Sperrung der Relation des Basisschemas vom Mandanten erfolgt und wie der Administrator sie sperrt. Gewöhnlich erfolgt die Sperrung einer Relation automatisch bei einem Aufruf der Funktion `relation_open`. Diese Funktion bestimmt laut dem Abschnitt 4.7.2.1 den Mandanten, für den die Relation geöffnet werden soll, und kann ihn dementsprechend an die Funktion des Lock Managers weitergeben.

Soll vom Administrator eine Operation auf allen Daten der Mandanten einer isolierten Tabelle ausgeführt werden, so wird ein großer Teil des Codes wie zuvor auch ausgeführt. Bisher gab es immer nur eine Relation und daher wird irgendwo im Code versucht, diese eine Relation zu öffnen. Da der Administrator die Operation ausführt, ist insbesondere kein Mandant gesetzt, und daher wird die Relation des Basisschemas geöffnet.

Betrachtet man beide Fälle, so läuft es auf eine einfache Fallunterscheidung hinaus. Wird eine Relation für einen Mandanten geöffnet, so muss die Funktion des Lock Managers zusätzlich die Relation des Basisschemas mit der entsprechenden neuen Version des angegebenen Modus sperren. Der neue Modus kann laut dem vorherigen Abschnitt ganz einfach berechnet werden, indem man zum angeforderten Modus acht addiert. Falls kein Mandant gesetzt wird, handelt es sich bei der zu lockenden Relation ohnehin schon um die des Basisschemas, die daher vollkommen korrekt mit einem der bisher definierten Modi gesperrt

wird. Ein vereinfachter Ausschnitt dieser Funktion ist in Ausschnitt 4.12 zu sehen. Bei den LockTags handelt es sich jeweils um den zusammengesetzten Schlüssel, über den die Relation identifiziert wird, und LockAcquire sperrt die Relation.

```
1  SetLocktagRelationOid(&tag, relid, tenantid);
2  if (tenantid!=InvalidTenantId) {
3     SetLocktagRelationOid(&tag2, relid, InvalidTenantId);
4     LockAcquire(&tag2, lockmode + 8);
5  }
6  res = LockAcquire(&tag, lockmode);
```

Ausschnitt 4.12: Sperre der Relation des Basisschemas im Falle eines Mandanten

4.10 Gemeinsamer Index

Eines der wichtigsten Instrumente, mit denen ein Administrator eine Datenbank möglichst effizient gestalten kann, ist die Möglichkeit, Indizes auf den Tabellen anzulegen. Der Performanzunterschied zwischen einer Anfrage auf einer Tabelle mit Index und einer ohne kann so gravierend sein, dass das Anlegen der Indizes nicht den Mandanten überlassen werden kann. Demnach muss der Administrator die Möglichkeit haben einen Index für alle Mandanten anzulegen.

Bei isolierten Tabellen stellt sich hier die Frage, wie so ein Index angelegt wird. Es könnte beispielsweise für jeden Mandanten ein vollständig eigener Index angelegt werden, der insbesondere eigene Einträge im Systemkatalog bekommt. Auf Grund der unnötig redundanten Informationen sieht das Konzept dagegen einen gemeinsamen Index vor, der nur eine Definition im Systemkatalog benötigt. Allerdings sieht PostgreSQL nicht vor, für eine Definition mehrere Instanzen eines Index zu erlauben. Daher muss es ähnlich wie bei den isolierten Tabellen eigene Relationen Deskriptoren geben, die auf eigene von der Speicherungsschicht verwaltete Dateien verweisen.

Darüber hinaus sieht das Konzept eine weitere Möglichkeit vor. Statt für jeden Mandant eine eigene Datei anzulegen, soll eine Datei von mehreren genutzt werden. Dazu wird es demnach notwendig, dass die darin enthaltenen Daten nach Mandanten getrennt werden können. Die Daten müssen daher beim Schreiben eine Kennzeichnung des Mandanten erhalten und beim Lesen muss nach dieser gefiltert werden können. Des Weiteren kann als Quelle der zu indizierenden Tupel beim initialen Aufbau eines über den Relationen Deskriptor angesprochenen Index nur eine Datei angegeben werden. Somit bedarf es weiterer Anpassungen, damit beim Aufbau einer Indexdatei die Tupel aus allen Dateien der Mandanten gelesen werden.

Die Abbildung 4.5 skizziert den Entwurf für den gemeinsamen Index. Sie ähnelt der Abbildung 4.2. In diesem Fall handelt es sich allerdings um einen gemeinsamen Index. Daher

können die Einträge im Systemkatalog `pg_mtclass` zu Mandantengruppen gehören. Bei den Einträgen in der Abbildung ist das bei beiden der Fall. Das kann man daran erkennen, dass sie jeweils auf eine Indexdatei verweisen, die nach Mandanten partitioniert wurde. Jede Partition indiziert die Daten eines Mandanten, die in einer mandanteneigenen Datei liegen. Der Zugriff auf die zwei Indexdateien wird wie bei der isolierten Tabelle dadurch erreicht, dass zwei Relationen Deskriptoren aufgebaut werden.

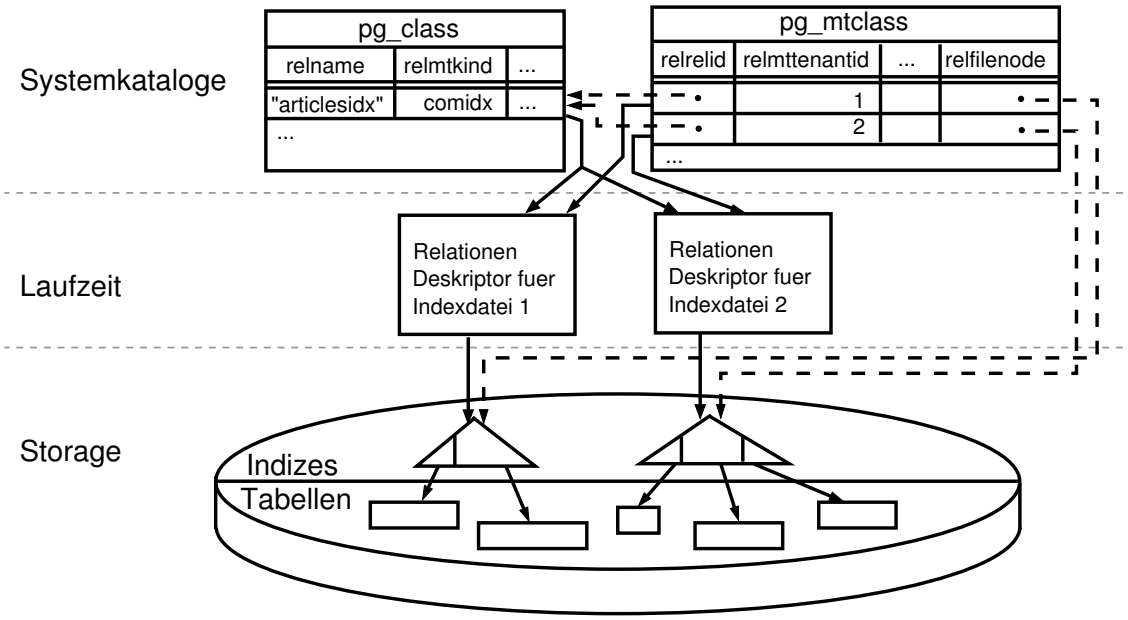

Abbildung 4.5: Entwurf für den gemeinsamen Index

4.10.1 Kennzeichnung in gemeinsamen Indexdateien

An dieser Stelle soll zunächst geklärt werden, wie die Kennzeichnung in einer gemeinsamen Indexdatei stattfinden soll, damit die darin gespeicherten Daten nach Mandanten getrennt werden können. Hierfür gibt es mindestens zwei verschiedene Ansätze. Zum Einen kann der Quellcode des Index verändert werden, damit dieser zwischen den Daten von Mandanten unterscheiden kann und demnach auch die Kennzeichnung selbst vornimmt. Die Variante könnte mandantenweise Operationen unterstützen. Zum Anderen kann der Quellcode des Index vollkommen unangetastet bleiben, womit die bisher gebotenen Schnittstellen genügen müssen. In diesem Entwurf wird der zweite Ansatz verfolgt, da somit nicht nur ein B-Baum als Index verwendet werden kann, sondern auch andere.

Bei dem gewählten Ansatz bleibt einem nur die Möglichkeit, über eine zusätzliche Spalte zu indizieren, die ID eines Mandanten enthält, das dadurch in das `IndexTuple`-Struct geschrieben werden kann und dadurch jeder Eintrag im Index gekennzeichnet wird. Zu diesem

Zweck wurde im Abschnitt 4.6.1 festgelegt, dass beim Anlegen einer isolierten Tabelle, eine zusätzliche Spalte `pg_tenantid` hinzugefügt wird. Somit existiert ein zusätzliches Attribut, auf das beim Anlegen des Index Bezug genommen werden kann, indem die zu indizierenden Attribute um das Attribut `pg_tenantid` ergänzt werden. Entsprechend dem Namen des Attributs wird zur Kennzeichnung direkt die jeweilige ID des Mandanten verwendet.

4.10.2 Das Struct IndexInfo

Für einen Index gibt es neben den Structs für den Relationen Deskriptor und den Tupel Deskriptor ein weiteres wichtiges Struct, das an viele Funktionen übergeben wird. Das Struct hat den Bezeichner `IndexInfo` und fasst die Eigenschaften eines Index zusammen. Eine Eigenschaft ist bisher beispielsweise, ob der Index auf dem Primärschlüssel angelegt wurde, oder ob er doppelte Einträge erlaubt.

Um das Verhalten von Funktionen für den gemeinsamen Index anzupassen, an die nur dieses Struct übergeben wird, liegt es nahe, dem Struct eine weitere Eigenschaft hinzuzufügen. Dazu wird das Attribut `ii_commonIndex` angelegt, das `true` oder `false` sein kann.

4.10.3 Anlegen eines gemeinsamen Index

Im Gegensatz zu den eingeführten Schlüsselwörtern zur Definition einer isolierten Tabelle gibt es für das Anlegen eines gemeinsamen Index keinen separaten Syntax. Stattdessen wird angenommen, dass einen Index, den der Administrator mit dem bisherigen Syntax auf einer isolierten Tabelle anlegt, immer ein gemeinsamer Index sein soll.

Das Anlegen erfolgt durch die schon erwähnte Funktion `DefineIndex`. Sie erkennt am Attribut `relmtkind` des zur Tabelle zugehörigen Eintrags des Systemkatalogs `pg_class`, dass es sich um eine isolierte Tabelle handelt, und somit, dass ein gemeinsamer Index angelegt werden soll, weshalb das Attribut `ii_commonIndex` auf `true` gesetzt wird. Des Weiteren wird der Liste der zu indizierenden Attribute das zuvor erwähnte Attribut `pg_tenantid` hinzugefügt. Zuletzt wird die im Abschnitt 4.5.3 beschriebene Funktion `createPGMtClassEntriesByMTTablespace` aufgerufen, damit die Indexdateien angelegt und die jeweilige Referenz auf die Datei in einem Eintrag im Systemkatalog `pg_mtclass` hinterlegt wird.

4.10.4 Auslesen von Daten aus gemeinsamen Indexdateien

Im Kapitel über PostgreSQL wurde unter Anderem kurz angedeutet, zu welchem Zweck ein Index dient. Er soll bei einem IndexScan genutzt werden, um zu einem Prädikat die externen Speicheradressen aller Tupel zu bestimmmen, die das Prädikat erfüllen. Da für isolierte Tabellen ein zusätzliches Attribut `pg_tenantid` angelegt wurde, kann dieses ebenfalls im Prädikat verwendet werden. Es genügt daher das Prädikat, um die Bedingung zu

ergänzen, dass das Attribut `pg_tenantid` gleich der ID des Mandanten sein muss, mit der die Relation geöffnet wurde.

Das Hinzufügen dieser Bedingung muss beim Initialisieren eines im Plan befindlichen Knotens, der den IndexScan repräsentiert, erledig werden. Dafür eignet sich die Funktion `ExecIndexBuildScanKeys`, die aus allen Bedingungen ein Array aus ScanKeys erstellen soll. Im Falle, dass es sich um eine Relation handelt, bei der das Attribut `rolmtkind` angibt, dass die Relation einen gemeinsamen Index darstellt, muss also lediglich das Array um eins größer angelegt werden und ein ScanKey für die neue Bedingung erstellt werden, der ebenfalls im Array abgelegt wird.

4.10.5 Einfügen von Daten in eine gemeinsame Indexdatei

Die Daten, die im Index gespeichert werden, müssen mittels der ID des Mandanten gekennzeichnet sein. Allerdings soll das Attribut `pg_tenantid` in einem Tupel der Tabelle nie gesetzt sein. Daher kann der bisherige Quellcode die ID nicht vom Tupel der Tabelle in das `IndexTuple`-Struct kopieren. Stattdessen muss der Wert explizit geschrieben werden.

Als Ausgangspunkt, um das Setzen der ID zu realisieren, bietet sich die bestehende Funktion `FormIndexDatum` an. Sie wird aufgerufen, falls aus einem Tupel der Tabelle ein `IndexTuple`-Struct erzeugt werden soll. Dazu bekommt sie einen weiteren Parameter, mit dem die zu setzende ID übergeben werden soll. Anhand des ebenfalls übergebenen `IndexInfo`-Structs wird festgestellt, ob es sich um einen gemeinsamen Index handelt. In dem Fall muss bei der üblichen Erzeugung aus den Werten des `HeapTuple`-Structs die erste Spalte, das Attribut `pg_tenantid`, zunächst leer gelassen werden, damit sie anschließend auf die übergebene ID gesetzt werden kann.

Damit wurde geklärt, wie und wo die Kennzeichnung stattfindet. Insbesondere wird vom bisherigen Quellcode bei einer Einfügeoperation eines Tupels in eine Tabelle automatisch eine Einfügeoperation in alle zur Tabelle gehörenden, aufgebauten Indizes gestartet, weshalb beim Einfügen von Tupel in Tabellen nichts weiter angepasst werden muss.

4.10.6 Aufbau einer Indexdatei für mehrere Mandanten

Im vorherigen Abschnitt wurde die Kennzeichnung beim Einfügen von Daten in einen Index und dadurch gleichzeitig das Einfügen von Tupeln in eine Tabelle abgehandelt. Es gibt jedoch eine weitere Operation, bei der Daten in einen Index eingefügt werden muss, der durch die Anpassungen an der Funktion `FormIndexDatum` nicht abgehandelt wurde. So muss beim Anlegen eines Index mittels dem Befehl `CREATE INDEX` der Index zunächst aus allen schon vorhandenen Tupeln in den Tabellen aufgebaut werden.

Der Aufbau der Indexstruktur soll größtenteils im Arbeitsspeicher erfolgen. Beim norma-

len Einfügen hingegen wird das zu speichernde `IndexTuple`-Struct direkt in einer externen Speicherseite abgelegt. Daher wird zunächst eine spezielle für den Aufbau eines Index zuständige Funktion aufgerufen, die mit der Initialisierung der eigenen Strukturen beginnt. Diese Funktion bekommt als Parameter einen Relationen Deskriptor eines Index übergeben und ist demnach lediglich für den Aufbau der zum Deskriptor gehörenden Indexdatei zuständig. Diese ruft danach eine Funktion namens `IndexBuildHeapScan` auf und übergibt ihr eine Callback-Funktion, an die alle von der Funktion geladenen Tupel der Tabelle durch einzelne Aufrufe übergeben werden sollen.

Bisher hat die Funktion `IndexBuildHeapScan` nur Tupel aus einer zuvor geöffneten Relation geladen. Es muss daher ein Weg gefunden werden, mit dem es leicht möglich ist, mehrere Relationen als Quelle zu verwenden. Ein Ansatz wäre beispielsweise, den Aufbau für jede Relation zu starten, allerdings unterstützt der wichtigste Indextyp B-Baum es nicht, die im Arbeitsspeicher aufgebaute Struktur in die bereits bestehende auf dem externen Speicher zu integrieren. Daher kann der Aufbau für jede Indexdatei, die leer sein muss, genau ein Mal gestartet werden.

Aus dem genannten Grund bleibt lediglich der zweite Ansatz, die Funktion `IndexBuildHeapScan` so abzuändern, dass sie die Tupel aus allen Relationen liest, für die die Indexdatei zuständig sein soll. Dazu muss zunächst festgestellt werden, welche Relationen dies sind. Daher wird die Variable `rd_mttenantid` des Relationen Deskriptors des Index verwendet, um den Eintrag des Mandanten bzw. der Mandantengruppe im Systemkatalog nachzuschlagen. Falls der Eintrag zu einem Mandant gehört, so wurde der Deskriptor nur für diesen einen Mandanten geöffnet, der demnach eine eigene Indexdatei hat und somit nichts weiters zu tun ist.

Erst wenn der Relationen Deskriptors des Index für eine Mandantengruppe geöffnet wurde, muss die zugehörige Indexdatei mit den Tupeln mehrerer Relationen befüllt werden. In diesem Fall werden alle Mandanten der Gruppe nachgeschlagen und jeweils durch den Systemkatalog `pg_mtclass` überprüft, dass für den Index und die ID des Mandanten kein Eintrag vorhanden ist und der Mandant demnach keine eigene Indexdatei besitzt. Für jedes Gruppenmitglied ohne eigene Indexdatei kann dadurch die Relation des Mandanten geöffnet werden, indem zuerst der aktuell gesetzte Mandant mittels einem Aufruf von `SetCurrentTenantId` mit der ID des Mandanten als Parameter geändert wird, damit anschließend die Funktion `relation_open` die isolierte Tabelle als diesen Mandant öffnet.

4.10.7 Aufbau mehrerer Indexdateien je gemeinsamer Index

Wie schon im vorherigen Abschnitt erwähnt wurde, gibt es eine Funktion für den Indexaufbau, die genau eine über den Deskriptor beschriebene Indexdatei aufbaut. Um den kompletten Index zu erzeugen, muss daher der Aufbau für jede Indexdatei gestartet werden.

Die Funktion, die den Aufbau startet, trägt den Bezeichner `index_build`. Sie ist darüber

hinaus noch dafür verantwortlich, die beim Lesen der Tupel der Tabelle angefallenen Statistiken abzuspeichern, wie beispielsweise die Anzahl der Tupel und die Anzahl der benötigten Seiten auf dem externen Speicher.

Die Funktion `index_build` muss zunächst alle Deskriptoren aufbauen können. Dazu hätte man die Möglichkeit, alle Mandanten und Gruppen zu durchlaufen und jeweils einen Deskriptor zu öffnen. Da man jedoch bei einem Mandanten ohne eigene Indexdatei den Deskriptor der Gruppe bekommt, würde man unter Umständen denselben Deskriptor mehrmals öffnen. Stattdessen muss über den Systemkatalog `pg_mtclass` iteriert werden. Im Gegensatz zu dem Fall einer isolierten Tabelle, bei der ein Mandant eine eigene Datei hat, obwohl kein Eintrag in diesem Systemkatalog vorhanden ist, wurden beim gemeinsamen Index alle Einträge durch die Funktion `createPGMtClassEntriesByMTTablespace` angelegt. Daher spiegelt der Systemkatalog alle Indexdateien wieder und damit alle verschiedenen Deskriptoren, die geöffnet werden können.

4.11 Gemeinsame Tabellen

Der bisherige Teil des Entwurfs befasste sich im Wesentlichen mit den Änderungen, die durch die hinzugekommenen isolierten Tabellen notwendig sind. Darüber hinaus sieht das Konzept noch eine weitere Tabelle vor. Eine, die mit den Schlüsselwörtern `SHARED BETWEEN TENANTS` angelegt wird, und deren Tupel von allen Mandanten gelesen werden können sollen, jedoch nur der Administrator neue hinzufügen können soll.

So eine Tabelle entspricht im Wesentlichen einer normalen Tabelle. Diese unterstützen ebenfalls eine Einschränkung der Rechte. Die Rechte werden üblicherweise für jeden Benutzer in einem Attribut `relacl` des Eintrags der Tabelle im Systemkatalog `pg_class` hinterlegt. Da das Konzept jedoch keine Verbindung zwischen Mandanten und Benutzern vorsieht, kann dieses Attribut nicht benutzt werden, um die Einschränkung vorzunehmen, weshalb im Folgenden ein paar Änderungen am Quellcode beschrieben werden müssen.

4.11.1 Anlegen einer gemeinsamen Tabelle

Wie schon im vorherigen Abschnitt erklärt wurde, kann eine gemeinsame Tabelle nicht ganz durch bestehende Mittel realisiert werden. Es sind deshalb ein paar kleine Anpassungen am Verhalten notwendig, falls es sich um eine solche Tabelle handelt. Daher muss ein eigener Wert für das Attribut `relmtkind` angelegt werden, auf den das Attribut durch die Funktion `DefineRelation` gesetzt wird, um diese Tabellen von anderen zu unterscheiden.

4.11.2 Einschränkung der Rechte

Da das Attribut `relacl` nicht direkt verwendet werden kann, um die Einschränkung zu erzielen, müssen beim Auslesen der Rechte des aktuellen Benutzers, alle Schreibrechte entfernt werden, falls die Relation von einem Mandant geöffnet wurde. Glücklicherweise erfolgt das Auslesen des Attributs ausschließlich durch eine Funktion namens `pg_class_aclmask`, so dass nur diese eine Funktion angepasst werden muss.

Die Funktion `pg_class_aclmask` bekommt den Deskriptor übergeben und damit auch den Wert des Attributs `rolmtkind`. Falls es sich demnach um eine gemeinsame Tabelle handelt und der Deskriptor einem Mandanten gehört, brauchen die Bits der zurückzugebenden Bitmaske, die die Schreibrechte des Benutzers auf die Tabelle repräsentieren, lediglich vorher entfernt werden.

5 Evaluation

Basierend auf dem Entwurf wurde zum Ende des Jahres 2009 eine über Git vom Repository heruntergeladene Version von PostgreSQL modifiziert, die auf einem Stand zwischen der öffentlichen Version 8.4 und 9.0 war. Der dadurch entstandene Prototyp wird im Folgenden auf seine Funktionstüchtigkeit hin getestet, um damit den sogenannten proof-of-concept abzuliefern.

5.1 Anlegen eines Mandanten und einer Mandantengruppe

Als Erstes soll im folgenden Ausschnitt 5.1 dokumentiert werden, wie ein Mandant und eine Gruppe angelegt werden kann. Dazu werden die zwei vom Konzept vorgesehenen Befehle ausgeführt. Nach dem Anlegen können sie über ein `SELECT` ausgegeben werden, wobei diejenigen mit einer ID gleich null ausgeblendet werden, da die normalen Rollen null verwenden. Man sieht, dass die fortlaufende Vergabe der ID und die Unterscheidung zwischen Mandant und Gruppen funktioniert.

```
1  postgres=# CREATE TENANTGROUP tenantgroup1;
2  CREATE TENANTGROUP
3  postgres=# CREATE TENANTGROUP tenantgroup2;
4  CREATE TENANTGROUP
5  postgres=# CREATE TENANT tenant1 IN TENANTGROUP tenantgroup1;
6  CREATE TENANT
7  postgres=# CREATE TENANT tenant2 IN TENANTGROUP tenantgroup2;
8  CREATE TENANT
9  postgres=# SELECT rolname,rolmtkind,rolmttenantid FROM
       pg_authid WHERE rolmttenantid!=0;
10    rolname    | rolmtkind | rolmttenantid
11 --------------|-----------|---------------
12  tenantgroup1 | g         |             1
13  tenantgroup2 | g         |             2
14  tenant1      | t         |             3
15  tenant2      | t         |             4
16
17 (2 rows)
```

Ausschnitt 5.1: Anlegen eines Mandanten und einer Mandantengruppe

5.2 Setzen des Mandanten je Datenbankverbindung

Eine wichtige Funktion eines Datenbanksystems ist der Mehrbenutzerbetrieb. Davon abgesehen, müssen auf Grund der Latenzen des externen Speichers mehrere Anfragen parallel eingeliefert und abgearbeitet werden können, damit das System voll ausgelastet werden kann. Daher muss das Setzen eines Mandanten unabhängig, also insbesondere parallel, je Backend bzw. Verbindung funktionieren. Daher wurde in Ausschnitt 5.2 und 5.3 parallel jeweils der Mandant gesetzt und ausgegeben.

```
1 postgres=# SET TENANT tenant1;
2 SET
3 postgres=# SHOW TENANT;
4  tenant
5 ----------
6  tenant1
```

Ausschnitt 5.2: Mandant setzen bei erster Datenverbindung

```
1 postgres=# SET TENANT tenant2;
2 SET
3 postgres=# SHOW TENANT;
4  tenant
5 ----------
6  tenant2
```

Ausschnitt 5.3: Mandant setzen bei zweiter Datenverbindung

5.3 Anlegen einer isolierten Tabelle mit Tablespaces je Manantengruppe

Der Administrator soll eine isolierte Tabelle anlegen können und dabei einen Tablespace je Mandantengruppe angeben können. Das ist im Ausschnitt 5.4 dokumentiert. Die erste Zeile legt zusätzlich zum Standardtablespace namens `pg_default` einen neuen namens `fastspace` an. Dieser kann dann beim Anlegen der isolierten Tabelle allen Mandanten einer zuvor angelegten Gruppe `tenantgroup2` zugewiesen werden. Das nachfolgende `SELECT` listet alle neuen Tabellentypen auf, da die normalen durch ein n markierten Tabellen in der `WHERE`-Klausel ausgeschlossen wurden. Man sieht also, dass die isolierte Tabelle durch das b unterschieden wird, und dass zusätzlich zu den zwei angegebenen Spalten die Spalte `pg_tenantid` angelegt wurde. Die Funktionstüchtigkeit der Tablespace-Zuweisung kann erst im nächsten Abschnitt gezeigt werden, da die Dateien der Mandanten zu diesem Zeitpunkt noch nicht erstellt wurden. In diesem Abschnitt wird der ebenfalls gelistete Verweis auf eine Datei mittels dem Attribut `relfilenode` wichtig sein.

```
postgres=# CREATE TABLESPACE fastspace LOCATION '/misc/
   postgresql/fastspace';
CREATE TABLESPACE
postgres=# CREATE TABLE articles (id INT, name VARCHAR(20))
   SEGREGATED BETWEEN TENANTS TABLESPACE pg_default FOR (
   tenantgroup1), fastspace FOR (tenantgroup2);
CREATE TABLE
postgres=# SELECT relname,relfilenode, relnatts,relmtkind FROM
    pg_class WHERE relmtkind!='n';
 relname  | relfilenode | relnatts | relmtkind
----------------------------------------------
 articles |       16389 |        3 | b
(1 row)
```

Ausschnitt 5.4: Anlegen einer isolierten Tabelle

5.4 Trennung der Daten bei einer isolierten Tabelle

Aufbauend auf dem vorherigen Fall, muss es nun für zwei Mandanten möglich sein, der Tabelle Tupel hinzuzufügen und auszulesen, ohne dass Tupel anderer Mandanten sichtbar sind. Dieser Fall ist in 5.5 zu sehen. In dem oberen Abschnitt wird jeweils zu einem der Mandanten gewechselt, einige Tupel werden eingefügt und diese nachfolgend ausgegeben. Man sieht beim zweiten, dass keine Tupel vom ersten Mandanten ausgegeben werden.

Der nachfolgende Teil demonstriert, dass das verzögerte Anlegen und das Aufsplitten zu mehreren mandanteneigenen Dateien funktioniert. Bei letzterem kann man insbesondere beobachten, wie die IDs der Mandanten als Suffix in den jeweiligen Dateinamen einfließen, wenn man die Nummern mit denen im Ausschnitt 5.1 vergleicht.

Der Teil zeigt darüber hinaus, dass die Zuweisung der Tablespaces korrekt umgesetzt wurde, da die Datei des Mandanten `tenant2` im Verzeichnis des Tablespace `fastspace` angelegt wurde.

```
postgres=# SET TENANT tenant1;
SET
postgres=# INSERT INTO articles VALUES (1, 'RAG␣20GB␣MP3-
   Player'), (2, 'Q-View␣Monitor');
INSERT 0 2
postgres=# SELECT * FROM articles;
 id |        name
----------------------------
  1 | RAG 20GB MP3-Player
  2 | Q-View Monitor
```

```
10 (2 rows)
11
12 postgres=# UNSET TENANT;
13 SET
14 postgres=# SET TENANT tenant2;
15 SET
16 postgres=# INSERT INTO articles VALUES (1, 'Yellow␣bed-linen')
       ;
17 INSERT 0 1
18 postgres=# SELECT * FROM articles;
19  id |        name
20 ----------------------------
21   1 | Yellow bed-linen
22 (1 row)
23
24 postgres=# UNSET TENANT;
25 SET
26 postgres=# \q
27 user@pc:/misc/postgresql$ find -name "*16389*"
28 ./data/base/11566/16389_0
29 ./data/base/11566/16389_3
30 ./fastspace/11566/16389_4
```

Ausschnitt 5.5: Veranschaulichung einer isolierten Tabelle

5.5 Mandanteneigenes Attribut

Bisher wurde erst gezeigt, dass die Trennung einer isolierten Tabelle funktioniert. Eine weiterer wichtiger Punkt des Konzepts war, dass der Mandant Schemaanpassungen vornehmen kann. Hierfür wurde als Beispiel das Hinzufügen eines Attributs als Mandant im Entwurf aufgenommen, das im folgenden Ausschnitt 5.6 getestet wurde.

Der obere Teil zeigt das Anlegen und das anschließende Abfragen der Tabelle als Mandant. Das `SELECT` zeigt alle Attribute der Tabelle, wobei die sogenannten Systemattribute mit den Nummern kleiner eins ausgeblendet wurden. Man sieht, dass die Zugehörigkeit des neuen Attributs über die Spalte `attmttenantid` dokumentiert wird, dass als verwendete Nummer die nächsthöhere verwendet wird, und dass das im Abschnitt 4.8.5.1 beschriebene Attribut `attmtlatestbaseatt` auf die Nummer des Attributs des Basisschemas gesetzt ist, das zum Zeitpunkt des Anlegens die höchste Nummer hatte.

```
1 postgres=# SET TENANT tenant1;
2 SET
```

```
3 postgres=# ALTER TABLE articles ADD COLUMN category INT
     DEFAULT 5;
4 ALTER TABLE
5 postgres=# SELECT * FROM articles;
6  id |         name          | category
7 ------------------------------------
8   1 | RAG 20GB MP3-Player |        5
9   2 | Q-View Monitor      |        5
10 (2 rows)
11
12 postgres=# UNSET TENANT;
13 SET
14 postgres=# SELECT attname, attnum,attmttenantid,
     attmtlatestbaseatt FROM pg_attribute, pg_class WHERE
     attrelid=oid AND relname='articles' AND attnum>0;
15    attname    | attnum | attmttenantid | attmtlatestbaseatt
16 ------------------------------------------------------------
17  pg_tenantid |      1 |             0 |                  0
18  id          |      2 |             0 |                  0
19  name        |      3 |             0 |                  0
20  category    |      4 |             3 |                  3
21 (4 rows)
```

Ausschnitt 5.6: Als Mandant Attribut zu einer isolierten Tabelle hinzufügen

5.6 Gemeinsamer Index mit mehreren Indexdateien

Ein Administrator soll die Möglichkeit haben, für alle Mandanten einen gemeinsamen Index auf einer isolierten Tabelle anzulegen. Dieser Fall ist in 5.7 zu sehen. Insbesondere wurde dort die im Abschnitt 4.5.3 erwähnte Tatsache benutzt, dass pro Tablespace-Zuweisung eine eigene Indexdatei angelegt wird, um zu demonstrieren, dass ein Mandant eine eigene Indexdatei besitzen kann.

Das `SELECT` zeigt alle Einträge in `pg_mtclass` der Mandanten samt Name der Relation. Man sieht anhand dem Attribut `relfilenode`, dass für jeden Mandant eine eigene Indexdatei angelegt wurde und dabei die Statistiken `reltuples` und `relpages` aktualisiert wurden. Darüber hinaus sieht man, dass beim Anlegen der Indexdateien die Statistiken für die mit Tupeln der Tabelle gefüllten Dateien ebenfalls abgespeichert wurden, wobei bei dem Mandant `tenant1` zusätzlich das Attribut `relfilenode` gesetzt ist, da er durch die für das `ALTER TABLE` im vorherigen Abschnitt notwendige Reorganisation eine neue Datei zugewiesen bekommen hat.

```
1 postgres=# CREATE INDEX commonidx ON articles (id) TABLESPACE
     pg_default FOR (tenant1), pg_default FOR (tenant2);
2 CREATE INDEX
3 postgres=# SELECT c.relname, mtc.relmttenantid, mtc.
     relfilenode, mtc.reltuples, mtc.relpages FROM pg_authid a,
     pg_mtclass mtc, pg_class c WHERE c.oid=mtc.relrelid AND mtc
     .relmttenantid=a.rolmttenantid AND a.rolmtkind='t';
4   relname  | relmtten.. | relfilenode | reltuples | relpages
5 -----------------------------------------------------------
6  articles  |          3 |       16393 |         2 |        1
7  commonidx |          3 |       16398 |         2 |        2
8  articles  |          4 |             |         1 |        1
9  commonidx |          4 |       16397 |         1 |        2
10 (4 rows)
```

Ausschnitt 5.7: Gemeinsamer Index anlegen

5.7 Gemeinsamer Index verwenden

Es bleibt zu zeigen, dass der im vorherigen Abschnitt angelegte gemeinsame Index verwendet werden kann. Dieser sollte jeweils eine Indexdatei je Mandant besitzen. Des Weiteren muss geprüft werden, dass eine Indexdatei einer Mandantengruppe ebenso verwendet wurde. Für beide Fälle wurde der obere Teil des Ausschnitts 5.8 ausgeführt und ergab bei beiden das korrekte Ergebnis.

Allerdings wurde im Abschnitt 3.7 schon erklärt, dass ein Datenbanksystem mehrere Möglichkeiten hat, die Tupel zu finden, die die `WHERE`-Klausel erfüllen. Man muss sich also davon überzeugen, dass wirklich der Index verwendet wurde. Dazu wurde im darauf folgenden Teil des Ausschnitts der Plan des `SELECT`s abgebildet. Obwohl der Plan schon stark vereinfacht wurde, ist er immer noch zu lange, um ihn komplett zu erklären. Es reicht jedoch vollkommen aus, zu sehen, dass in der Zeile dreizehn steht, dass ein IndexScan verwendet wird.

```
1 postgres=# SET TENANT tenant1;
2 SET
3 postgres=# SELECT * FROM articles WHERE id=1;
4  id |        name         | category
5 ------------------------------------
6   1 | RAG 20GB MP3-Player |        5
7 (1 row)
8 postgres=# \q
9 user@pc:/misc/postgresql$ cat pg_log/logfile
10 2010-05-24 00:15:51 CEST DETAIL:     {PLANNEDSTMT
```

```
11          :commandType 1
12          :planTree
13             {INDEXSCAN
14             :startup_cost 0.00
15             :total_cost 93960.66
16             :plan_rows 5368709
17             :targetlist (
18                {TARGETENTRY
19                :expr
20                   {VAR
21                   :varno 1
22                   :varattno 1
23                   :vartype 23
24                   }
25                :resno 1
26                :resname id
27                :resorigtbl 16389
28                :resorigcol 1
29                }
30                {TARGETENTRY
31                :expr
32                   {VAR
33                   :varno 1
34                   :varattno 2
35                   :vartype 1043
36                   }
37                :resno 2
38                :resname name
39                :resorigtbl 16389
40                :resorigcol 2
41                }
42                {TARGETENTRY
43                :expr
44                   {VAR
45                   :varno 1
46                   :varattno 4
47                   :vartype 23
48                   }
49                :resno 3
50                :resname category
51                :resorigtbl 16389
52                :resorigcol 4
53                }
```

```
54                )
55                :qual <>
56                :lefttree <>
57                :righttree <>
58                :initPlan <>
59                :extParam (b)
60                :allParam (b)
61                :scanrelid 1
62                :indexid 16396
63                :indexqual (
64                   {OPEXPR
65                   :opno 96
66                   :opfuncid 65
67                   :opresulttype 16
68                   :args (
69                      {VAR
70                      :varno 1
71                      :varattno 1
72                      :vartype 23
73                      }
74                      {CONST
75                      :consttype 23
76                      :constlen 4
77                      :constisnull false
78                      :constvalue 4 [ 1 0 0 0 0 0 0 0 ]
79                      }
80                   )
81                   }
82                )
83                }
84             :rtable (
85                {RTE
86                :eref
87                   {ALIAS
88                   :aliasname articles
89                   :colnames ("id" "name" "<>" "category")
90                   }
91                :rtekind 0
92                :relid 16389
93                :inFromCl true
94                :selectedCols (b 9 10 12)
95                :modifiedCols (b)
96                }
```

```
97              )
98              :relationOids (o 16389)
99              }
```

Ausschnitt 5.8: Test eines gemeinsamen Index

6 Zusammenfassung

In diesem letzten Kapitel wird die Arbeit zusammengefasst. Es nimmt dabei direkt Bezug zu den in der Einleitung formulierten Anforderungen, um festzustellen, welche erfüllt wurden, und bei welchen es noch etwas zu tun gibt, damit sie erfüllt sind.

Im Konzept und im darauf basierenden, im Abschnitt 4.3 erläuterten Entwurf wurde der Mandant als Objekt im Datenbanksystem eingeführt. In der Einleitung wurde erklärt, warum das Datenmanagement für mehrere Mandanten vom Datenbanksystem übernommen werden soll, und daher auch, dass die Verwaltung der Mandanten als Datenbankobjekte im Datenbanksystem erfolgen muss.

Mithilfe dieser Objekte wurde die Isolierung von Daten einer Tabelle im Abschnitt 4.11 erreicht, die von allen Mandanten genutzt wird. Dadurch können beispielsweise die Artikel mehrerer Onlineshops unterschiedlicher Mandanten isoliert werden. So eine Tabelle wurde als isolierte Tabelle bezeichnet. Dagegen hat ein Mandant bisher noch keine Möglichkeit, selbst eine Tabelle anzulegen, die kein anderer Mandant sieht.

Mit der Definition einer isolierten Tabelle für alle Mandanten wird die Möglichkeit der Konsolidierungseffizienz ausgenutzt. Gleichzeitig enthält der Entwurf alles nötige, um einer isolierte Tabellen ein mandantenspezifisches Attribut hinzuzufügen, was im Abschnitt 4.8 nachzulesen ist. Dadurch hat das System einen Vorteil gegenüber dem Ansatz, ein Shared Schema zu verwenden.

Die genannten Punkte gelten außerdem für den im Abschnitt 4.10 beschriebenen gemeinsamen Index auf isolierten Tabellen. Es wird ebenfalls nur eine Definition für alle Mandanten benötigt und die Isolierung wurde erreicht. Es gibt allerdings schon Ideen den gemeinsamen Index weiter zu verbessern. Im Moment wird bei jedem indizierten Tupel gekennzeichnet, zu welchem Mandant es gehört. Es gibt Bemühungen die indizierten Tupel zu sortieren und nur zu kennzeichnen, dass alle folgenden Tupel zu einem Mandanten gehören, um Speicherplatz zu sparen.

Eine Möglichkeit der Konsolidierungseffizienz wurde noch nicht ausgenutzt. Durch die gemeinsame Definition der isolierten Tabelle könnten die zur Ausführung angereicherten Informationen für alle Mandanten genutzt werden, die die gleiche Anfrage an das Datenbanksystem übermitteln. Es hat sich im Laufe der Arbeit herausgestellt, dass dabei einige Probleme auftreten, die im Entwurf im Abschnitt 4.8.7 dokumentiert sind, die es zu lösen gilt.

Bezüglich der Wartbarkeit ist zu erwähnen, dass der Entwurf zwar alles notwendige vorsieht, damit ein Administrator ein Attribut hinzufügen kann, der Prototyp dies allerdings im Moment noch nicht unterstützt. Bei einer Schemaänderung durch den Administrator muss wegen des Aufbaus von PostgreSQL die Tabelle für alle Mandanten gesperrt werden. Es stellt sich die Frage, ob dieser Engpass nicht beseitigt werden könnte. Dazu wären allerdings weitere tiefgreifende Änderungen in PostgreSQL notwendig.

Zusätzlich zu den schon genannten Punkten, sind weiterführende Arbeiten nötig, um die Skalierbarkeit, die dynamische Buchung von Ressourcen und die Datensicherheit zu untersuchen. Zu der Skalierbarkeit kann jedoch erwähnt werden, dass das eingangs genannte Problem der häufigen Kontextwechsel bei dem entworfenen System nicht auftreten kann, da sich durch den im Abschnitt 2.1.8 beschriebenen Befehl mehrere Mandanten eine Datenbankverbindung und daher laut dem Abschnitt 3.3 einen Serverprozess teilen.

Literaturverzeichnis

[1] Gartner: *Gartner Says 25 Percent of New Business Software Will Be Delivered As Software As A Service by 2011.* Oktober 2006. – URL `http://www.gartner.com/it/page.jsp?id=496886`. – Zugriffsdatum: 25 Mai 2010

[2] Jacobs, D. ; Aulbach, S.: *Ruminations on Multi-Tenant Databases, in BTW.* 2007

[3] Kerkhoff, Hendrik: *Multi-Tenant-Datenbanken für SaaS.* Januar 2010

[4] Lane, Tom: *A Tour of PostgreSQLs Internals.* Oktober 2000

[5] Momjian, Bruce: *PostgreSQL Internals Through Pictures.* Dezember 2001

[6] salesforce.com: *Salesforce.* Mai 2010. – URL `http://www.salesforce.com`. – Zugriffsdatum: 25 Mai 2010

[7] The PostgreSQL Global Development Group: *PostgreSQL - About.* Mai 2010. – URL `http://www.postgresql.org/about/`. – Zugriffsdatum: 25 Mai 2010

[8] Wikipedia-Benutzer: *MySQL.* Mai 2010. – URL `http://de.wikipedia.org/wiki/MySQL`. – Zugriffsdatum: 25 Mai 2010

[9] Wikipedia-Benutzer: *PostgreSQL.* Mai 2010. – URL `http://de.wikipedia.org/wiki/PostgreSQL`. – Zugriffsdatum: 25 Mai 2010

Erklärung

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

(Benjamin Schiller)