WIRTSCHAFTSUNIVERSITÄT WIEN

MASTERARBEIT

Titel der Masterarbeit:
Optimierung der Datenretention für große Datenmengen in PostgreSQL

Verfasser:	Daniel Sturm, BSc (WU)
Matrikel-Nr.:	0250478
Studienrichtung:	Wirtschaftsinformatik
Beurteiler:	Univ. Prof. Dipl.-Ing. Mag. Dr. Wolfgang Panny
Betreuender Assistent:	Dipl.-Ing. Mag. Dr. Albert Weichselbraun

Ich versichere:

dass ich die Masterarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

dass ich dieses Masterthema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Datum　　　　　　　　　　　　　　　　　　　Unterschrift

Abstract

Technologien, welche die Handhabung großer Datenmengen erleichtern und die Performance sehr großer Datenbanken steigern können, gewinnen mit steigender Verfügbarkeit von Daten sowohl für Unternehmen als auch für Forschungseinrichtungen immer mehr an Bedeutung. Diese Arbeit bietet einen Überblick über aktuelle Technologien zur Handhabung großer Datenmengen und stellt in weiterer Folge Softwaretools vor, welche Performanceprobleme in PostgreSQL Datenbanken aufzeigen können und die Wahl geeigneter Maßnahmen zur Performanceverbesserung unterstützen. Der Fokus liegt bei dieser Arbeit auf dem Datenbankmanagementsystem PostgreSQL.

Einen wichtigen Teil der Arbeit bildet die Entwicklung eines Softwaretools, welches das Partitionieren von Tabellen in PostgreSQL Datenbanken erheblich vereinfacht. Eine wichtige Funktion dieses Tools ist die Möglichkeit, bislang nicht-partitionierte Tabellen mit einer nur geringen Beeinflussung des normalen Datenbankbetriebs zu partitionieren.

Key Words
Partitionierung, PostgreSQL, sehr große Datenbanken

Inhaltsverzeichnis

1 Einleitung **8**
1.1 Problemstellung 9
1.2 Zielsetzung und Forschungsfrage 10

2 Datenbanktechnologien zur Speicherung großer Datenmengen **12**
2.1 Indizes 14
2.2 Materialized Views 17
2.3 Partitionierung 20
2.3.1 Vertikale Partitionierung 21
2.3.2 Horizontale Partitionierung 22
2.4 Tablespaces 28

3 Softwarelösungen für den Umgang mit großen Datenmengen **30**
3.1 PL/Proxy 30
3.2 RRDtool - Round Robin Database 31
3.3 Tabellenpartitionierung bei IBM DB2 9 und Oracle Database 11g 33
3.3.1 IBM DB2 9 33
3.3.2 Oracle Database 11g 35

4 Softwaretools zur Analyse der Nutzung und Performance von PostgreSQL Datenbanken **38**
4.1 Practical Query Analysis (PQA) 39
4.2 Enterprise Practical Query Analyzer (epqa) 41

4.3 pgFouine 42
4.4 Pgstatspack 42
4.5 pg_statsinfo 43

5 Implementierung eines Softwaretools für das Partitionieren von PostgreSQL Datenbanken 44
5.1 Verwendete Technologien 44
5.1.1 Java 44
5.1.2 PostgreSQL 45
5.1.3 PL/pgSQL 45
5.2 Anforderungen 46
5.3 Implementierung des Softwaretools 51
5.3.1 Überblick 51
5.3.2 Erstellen einer partitionierten Tabelle 54
5.3.3 Eine nicht-partitionierte Tabelle partitionieren . . . 58
5.3.4 Partitionen zu einer partitionierten Tabelle hinzufügen 63
5.3.5 Partitionen zusammenführen 64
5.3.6 Partitionen teilen 66
5.3.7 Partitionen aus- und wieder einhängen 67
5.3.8 Primärschlüssel, Fremdschlüssel und UNIQUE-Constraints von partitionierten Tabellen überprüfen . . . 68

6 Softwaretests 70
6.1 Funktionstest 70
6.2 Performancetest 74

7 Zusammenfassung und Ausblick 77

Literaturverzeichnis 79

Abbildungsverzeichnis

2.1 Beispiel einer vertikal partitionierten Tabelle 21

5.1 Use-Case-Diagramm für das Softwaretool *pgEasyPart* 50
5.2 Klassendiagramm des Softwaretools *pgEasyPart* 52
5.3 Aktivitätsdiagramm für das Erstellen einer partitionierten Tabelle 56
5.4 Aktivitätsdiagramm für das Partitionieren einer nicht-partitionierten Tabelle 61
5.5 Zusammenführen zweier Partitionen mit *pgEasyPart* 65
5.6 Teilen einer Partition mit *pgEasyPart* 67

Tabellenverzeichnis

6.1 Vergleich der Laufzeiten beim Partitionieren einer nicht-partitionierten Tabelle 75
6.2 Vergleich der `INSERT`- und `SELECT`-Performance während des Partitionierens der Tabelle *content_ascii* 76

Listings

2.1 Beispiel für horizontale Partitionierung in PostgreSQL - Teil I 26
2.2 Beispiel für horizontale Partitionierung in PostgreSQL - Teil II 27
4.1 Beispiel eines PQA Berichtes zur Analyse von Datenbanken 40
5.1 PL/pgSQL-Funktionen zum Verschieben der Daten in die Partitionen 59
5.2 *Query Plan* einer partitionierten Tabelle 62
6.1 Datenbankschema der Testdaten des Forschungsprojektes *IDI-OM Media Watch on Climate Change* 72

1 Einleitung

Die vorliegende Masterarbeit befasst sich mit der Haltung großer Datenmengen in Datenbanken. Im Speziellen liegt das Augenmerk auf Datenbanken mit großen Tabellen, wobei es sich dabei vorwiegend um Daten handelt, die kaum Veränderungen erfahren.

Die Arbeit soll zunächst in Kapitel 2 einen möglichst allgemeinen Überblick der wichtigsten Datenbanktechnologien, die bei der Verwaltung großer Datenmengen zum Einsatz kommen, bieten. Der Fokus wird in diesem Kapitel nicht nur auf dem Datenbankmanagementsystem (DBMS) PostgreSQL liegen, wenngleich auch hier teilweise schon darauf eingegangen wird, wie diese Technologien in PostgreSQL umgesetzt werden. Es soll vielmehr ein Überblick über sämtliche Technologien geboten werden, auch solche, die in PostgreSQL nicht (nativ) unterstützt werden. In den Kapiteln 4, 5 und 6 wird der Fokus der Arbeit ausschließlich auf dem Datenbanksystem PostgreSQL liegen.

Kapitel 3 und 4 betrachten jeweils vorhandene Softwarelösungen. In Kapitel 3 werden Softwarelösungen für PostgreSQL und andere Produkte analysiert, die für den Umgang mit großen Datenmengen entwickelt wurden. Es soll erörtert werden, welche nützlichen Funktionen diese zur Handhabung großer Datenmengen bereitstellen. Kapitel 4 bietet einen Überblick über Softwaretools, mit denen sich Statistiken über die Nutzung von PostgreSQL Datenbanken sammeln und auswerten lassen beziehungsweise die den Nutzer bei der Auswertung unterstützen. Diese Tools können dazu verwendet werden, geeignete Schlüsselkriterien für die Partitionierung zu identifizieren.

Anschließend wird in Kapitel 5 die Implementierung eines Softwaretools zur Partitionierung einer PostgreSQL Datenbank parallel zum laufendem Betrieb dokumentiert. Am Ende der Arbeit werden in Kapitel 6 die Ergebnisse der Funktions- und Performancetests präsentiert. Dabei soll auch die Performance der Datenbank vor und während der Umstellung verglichen werden.

Die folgenden beiden Unterkapitel werden die Problemstellung und die Zielsetzung der Arbeit inklusive der Forschungsfrage detailliert beschreiben.

1.1 Problemstellung

Vorratsspeicherung von Daten hat die Eigenschaft, dass kontinuierlich neue Daten erfasst werden. Die Datenbanken wachsen daher stetig an, was zu einem ständig steigenden Speicherbedarf und zu sehr großen Tabellen führt. Dadurch wird die Handhabung der Datenmengen immer mehr zu einem Problem.

Die Folge dieser wachsenden Datenmengen ist eine schlechtere Performance bei Datenbankabfragen, weil das DBMS aufgrund der steigenden Anzahl von Tupeln in einer Tabelle eine immer größere Anzahl von Zeilen pro Anfrage durchsuchen muss. Außerdem treten mit steigender Datenmenge auch Speicherprobleme auf. Der benötigte Speicherplatz steigt, gleichzeitig werden auch schnellere, teurere Speichermedien benötigt, um die Performanceprobleme in den Griff zu bekommen.

Da auf manche Daten öfter und auf andere wiederum weniger oft zugegriffen wird (oft werden zum Beispiel aktuellere Daten häufiger benötigt als ältere Daten), bietet es sich an, diese in getrennten Tabellen zu speichern. Dadurch kann die Performance des DBMS gesteigert werden und weniger häufig verwendete Daten könnten auf langsamere, aber kostengünstigere Datenträger verschoben werden. Um dies zu ermöglichen, sind ein intelligentes Design der Datenbank und Strategien notwendig, um die Daten nach bestimmten Kriterien auf verschiedene Speichermedien zu verteilen.

Diese Arbeit soll das Forschungsprojekt *IDIOM Media Watch on Climate Change*[1] am Institut für Informationswirtschaft der Wirtschaftsuniversität Wien unterstützen. In diesem Projekt werden Daten von Online-Medien, Blogs und Firmenhomepages gesammelt und ausgewertet. Wöchentlich werden dabei um die 500.000 Artikel in einer PostgreSQL Datenbank gespeichert. Diese Masterarbeit soll einen Beitrag leisten, um die vorhandenen Speicherressourcen optimal einzusetzen, damit eine solche Menge an Daten erfasst werden kann und gleichzeitig ein leistungsfähiges System gewährleistet wird. Die Arbeit wird sich bei der Umsetzung an den Anforderungen dieses Projektes orientieren. Der Fokus der verwendeten Datenbanktechnologien liegt daher auch auf dem open-source objektrelationalen Datenbankmanagementsystem PostgreSQL.

Es ist bereits hinlänglich in vorhandener Literatur beschrieben, wie man die oben genannten Anforderungen an Datenbanken durch entsprechendes Datenbankdesign beim Entwurf der Datenbank (zum Beispiel durch Partitionierung von großen Tabellen) bewältigen kann. Diese Masterarbeit soll sich damit auseinandersetzen, Strategien zu entwickeln, um diese Konzepte auf einer bereits bestehenden Datenbank umzusetzen.

1.2 Zielsetzung und Forschungsfrage

Das Ziel dieser Masterarbeit ist, eine oder mehrere Strategien zu implementieren, um eine bestehende PostgreSQL Datenbank intelligent über mehrere Speicherressourcen verwalten zu können.

Die Datenbank soll dabei im Zuge der Umstellung möglichst verfügbar bleiben. Eine vollständige Replikation der Datenbank, bei der diese vorübergehend nicht verfügbar ist, wäre nicht möglich, da einige Tabellen so groß sind, dass die Umstellung der Datenbank mehrere Tage oder sogar Wochen in Anspruch nehmen würde, und diese somit für diesen Zeitraum auch nicht verfügbar wäre.

[1] http://www.ecoresearch.net/climate/

Nach der Umstellung der Datenbank soll das Datenbankdesign in einer solchen Form sein, dass einzelne Tabellen durch einen nur geringen Aufwand auf andere Speichereinheiten verschoben werden können. Zum Beispiel sollte es dann möglich sein, ältere Daten, die nicht mehr so oft abgefragt werden, einfach von einem schnellen SAS-Speicher auf einen langsameren, aber kostengünstigeren SATA-Speicher zu verschieben. Um dies zu bewerkstelligen, müssen große Tabellen auf mehrere kleine, besser handhabbare Tabellen aufgeteilt (partitioniert) werden. Die Änderungen an der Datenbank sollen sich bei der Kommunikation mit dem Client nicht auswirken, sodass Programme, die mit der Datenbank kommunizieren, nicht modifiziert werden müssen.

Gleichzeitig sollten die Modifizierungen an der Datenbank auch die Performance der Datenbank positiv beeinflussen. Somit soll sowohl eine bessere Skalierbarkeit als auch eine bessere Performance erreicht werden.

Die Masterarbeit soll daher Erkenntnisse zu folgenden Fragestellungen liefern:

1. Welche Möglichkeiten bietet das Datenbankmanagementsystem PostgreSQL, um Daten aus einer Datenbank mit sehr großen Tabellen intelligent (über mehrere Speicherressourcen) verwalten zu können?
2. Wie sehen mögliche Strategien aus, um Tabellen in einer bestehenden PostgreSQL Datenbank zu partitionieren, ohne dabei die Verfügbarkeit der Datenbank einzuschränken? Wie werden solche Strategien implementiert?
3. Welche Auswirkungen hat dies auf die Performance der Datenbank während der Partitionierung?

2 Datenbanktechnologien zur Speicherung großer Datenmengen

In diesem Kapitel werden Datenbanktechnologien beschrieben, die bei der Speicherung großer Datenmengen eingesetzt werden und die theoretische Grundlage für die spätere Implementierung schaffen. Bei den beschriebenen Technologien steht in erster Linie eine Verbesserung der Performance im Vordergrund. Einige können darüber hinaus auch die Administration großer Datenmengen erleichtern.

Grundsätzlich lässt sich eine Datenbank auf verschiedenen Ebenen optimieren. Die größten Performancegewinne lassen sich bereits in der Planungsphase beim Design der Datenbank erzielen. In dieser Phase sind auch die Kosten für die Optimierung am geringsten. Unter anderem können Datenbanken durch Indizierung oder Partitionierung auf die jeweiligen Anwendungen abgestimmt werden. Weiters lässt sich auch durch Optimieren des Betriebssystems und der Datenbankzugriffe (zum Beispiel die Formulierung von `SELECT`-Statements) die Performance einer Datenbank steigern. Allgemein lässt sich festhalten, dass sich durch eine Reduktion der Schreib- und Lesezugriffe auf die Datenbank die größten Performancesteigerungen erzielen lassen [Bartl2003].

In den folgenden Unterkapiteln werden die Datenbanktechnologien *Indizierung*, *Materialized Views*, *vertikale Partitionierung*, *horizontale Partitionierung* und *Tablespaces* näher beschrieben und deren Vor- und Nachteile erläutert. Es handelt sich dabei um Technologien, die das physische Design der Datenbank bestimmen. Das logische Design der Datenbank wird

dadurch in der Regel nicht verändert [Agrawal2004b]. Die genannten Technologien werden vor allem im Bereich von Datawarehouse-Anwendungen häufig verwendet und eignen sich gut, um die Verwaltung großer Datenmengen beziehungsweise komplexe Anfragen über große Datenmengen zu optimieren [Kulev2007].

All diesen Technologien ist gemein, dass der Anfragenoptimierer (Query Optimizer) der jeweiligen Datenbank diese möglichst gut unterstützen sollte. Der Anfragenoptimierer muss beispielsweise entscheiden, ob ein Materialized View bei einer Datenbankanfrage logisch verwendet werden kann und ob die Verwendung auch zu einer Performancesteigerung führt [Halevy2001]. Die Entscheidungen des Anfragenoptimierers lassen sich durch verschiedene Tools veranschaulichen [Bartl2003]. In PostgreSQL wird dafür der Befehl `EXPLAIN` verwendet [PostgreSQL2009a].

Ein wesentlicher Unterschied zwischen den genannten Technologien liegt bei der Verwendung von zusätzlichem Speicherplatz. Während bei Materialized Views und Indizierung Daten teilweise repliziert werden, wodurch ein zusätzlicher Speicherbedarf anfällt, ist dies bei Partitionierung nicht der Fall [Ailamaki2004]. In den entsprechenden Unterkapiteln wird darauf noch näher eingegangen.

Neben den schon erwähnten, im Anschluss beschriebenen, Technologien gibt es noch weitere Möglichkeiten, die Performance einer Datenbank zu steigern. Zu den wichtigsten zählen:

- **Denormalisierung:**
  Bei Denormalisierung werden Tabellen zusammengefasst, Spalten teilweise redundant gespeichert und zusätzliche Werte gespeichert, die normalerweise aus den vorhandenen Daten berechnet werden können. Dies führt zu einer Verminderung von Joins, Fremdschlüsseln, benötigten Indizes und aufwendigen Berechnungen [Bartl2003].
- **Reorganisation:**
  Durch Reorganisation können Degenerierungen in den Speicherstrukturen beseitigt werden. Diese Degenerierungen können durch Einfüge-,

Änderungs- und Löschoperationen entstehen. Dabei kann es zur Fragmentierung von Daten, zur Auslagerung von einzelnen Tupeln an einen anderen Speicherort oder zu einer Nichtberücksichtigung von Sortierreihenfolgen bei sortierten Datensätzen kommen [Dorendorf2003]. Reorganisation führt oft zu einer deutlich besseren Performance bei Datenbankzugriffen, und es kann dadurch auch Speicherplatz wiedergewonnen werden [Hrle2004, Dorendorf2003]. Reorganisation von Datenbanken zählt zu den typischen Wartungsarbeiten [Hrle2004]. Allerdings handelt es sich dabei um einen sehr aufwendigen Prozess, der während der Laufzeit einen starken negativen Einfluss auf die Systemleistung haben kann [Dorendorf2003].

- **Parallel Database Systems:**
  Die Nutzung paralleler Datenbanksysteme ist eine weitere Möglichkeit zur Performancesteigerung bei Datenbanken und kann auch in Verbindung mit anderen Technologien, wie Materialized Views, Indizes und Partitionierung eingesetzt werden. Manche Architekturen (beispielsweise shared-disk Systeme) verlangen allerdings nach einer speziell für parallele Systeme entwickelten Hardware. Shared-nothing Systeme bieten dazu eine Alternative. Bei solchen Systemen können herkömmliche low-cost Computer und herkömmliche lokale Netzwerke eingesetzt werden [Furtado2006]. Diese Technologie verlangt allerdings nach spezieller Hardware und Datenbank Architektur und kann daher nicht ohne weiteres in herkömmlichen Datenbanksystemen umgesetzt werden. Daher wird in den folgenden Unterkapiteln nicht näher darauf eingegangen.

2.1 Indizes

Die Performance von Datenbankoperationen kann durch die Verwendung von Indizes deutlich verbessert werden [Bartl2003]. In der Literatur wird besonders die Verwendung der zwei Indextypen B-tree-Indizes und Bitmap-Indizes diskutiert. Welcher dieser beiden Indextypen verwendet werden soll

hängt in erster Linie von der Datenverteilung ab [Bartl2003]. Es sollte bei der Auswahl des passenden Indextyps darauf geachtet werden, wie selektiv eine Anfrage ist und wie viele Wertausprägungen des zu indizierenden Attributes existieren [Albrecht2006].

B-tree-Indizes eignen sich besonders, wenn die Selektivität hoch ist. Dies ist insbesondere bei Schlüsselattributen der Fall. Die Nachteile dieser Indizes sind, dass sie sich schlecht dazu eignen, bei Datenbankabfragen zwei Indizes miteinander zu verknüpfen und dass bei geringer Selektivität oft ein Full-Table-Scan bessere Performance aufweist [Albrecht2006]. Nach Albrecht [Albrecht2006] ist ein Full-Table-Scan bereits schon dann vorzuziehen, wenn bei einer Abfrage, bei der über ein einziges Attribut selektiert wird, über 5-10% der Datensätze ausgewählt werden.

Bitmap-Indizes können, im Gegensatz zu B-tree-Indizes, gut miteinander verknüpft werden und weisen auch bei Datenbankabfragen mit einer geringen Selektivität eine bessere Performance, im Vergleich zu einem Full-Table-Scan, auf [Albrecht2006, Bartl2003]. Bitmap-Indizes sind allerdings für Systeme mit vielen einzelnen Transaktionen und für Attribute mit vielen Wertausprägungen ungeeignet [Albrecht2006]. Sie eignen sich besonders bei OLAP (Online-Analytical-Processing) Anwendungen zur Abfrage großer Datenmengen [Kulev2007].

Bei durchschnittlichen Anwendungen verbrauchen Bitmap-Indizes wesentlich weniger zusätzlichen Speicherplatz als B-tree-Indizes [Albrecht2006]. Bartl et al. [Bartl2003] konnten durch die Verwendung eines Bitmap-Indizes zur Indizierung eines schwach selektiven Attributes den zusätzlichen Speicheraufwand für die Indizierung von ursprünglich 150 MB mit B-tree-Indizes auf 2 MB reduzieren. Kulev und Welker [Kulev2007] konnten nachweisen, dass Bitmap-Indizierung bei Attributen mit schwach selektiver Werteausprägung (drei mögliche Werte) nicht nur eine geringere Indexgröße (45 MB für Bitmap-Indizes und 2780 MB für BTree+-Indizes bei 100 Millionen Datensätzen), sondern auch eine geringere Indexaufbauzeit (504 Sekunden bei Bitmap-Indizierung und 2401 Sekunden bei BTree+-Indizierung) aufweist.

Neben den bereits erwähnten Indextypen existieren auch noch weitere spezielle Indextypen, die vor allem zur Performancesteigerung bei großen Datenmengen und komplexen Abfragen von Bedeutung sind. Dazu zählen partielle Indizes und Funktionsindizes. Mit partiellen Indizes lässt sich ein durch eine festgelegte Bedingung ausgewählter Teil einer Tabelle indizieren. Funktionsindizes dienen hingegen zum Indizieren von festgelegten Operationen, wobei es bei Funktionsindizes auch möglich ist, eine Funktion über mehrere Tabellen zu indizieren [Kulev2007].

Zusätzlich zu dieser Auswahl an verschiedenen Indextypen bieten viele Datenbanken auch die Möglichkeit, einen sogenannten *Clustered Index* beziehungsweise *Indexorganized Table* anzulegen[2]. Dabei werden alle Datensätze einer Tabelle anhand eines bestimmten Kriteriums (des jeweiligen Index) geordnet [Dorendorf2003]. Durch diese Sortierung der Daten soll erreicht werden, dass Tupel mit ähnlichen Kriterien physisch nahe beieinander liegend gespeichert werden [Markl1999]. Da Datenbanken in der Regel so organisiert sind, dass I/O-Operationen jeweils seitenweise erfolgen, wobei eine Seite (Page) mehrere Objekte (z.B. Tupel) beinhaltet, kann durch die physische Sortierung der Datensätze die Anzahl der Datenbankzugriffe bei einer Abfrage verringert werden. Dies ist insbesondere dann der Fall, wenn alle relevanten Datensätze auf einer Seite liegen. In diesem Fall muss nur diese eine Seite gelesen werden [Marek1992]. Beim Einfügen neuer Datensätze kommt es hingegen zu einem größeren Aufwand als bei unsortierten Tabellen, da die Datensätze an einer bestimmten Stelle eingefügt werden müssen, um die Sortierreihenfolge aufrecht zu erhalten. Das Einfügen mehrerer tausend Datensätze kann dadurch mehrere Minuten in Anspruch nehmen und gleichzeitig den Zugriff auf die jeweilige Tabelle während dieser Operation einschränken [Graefe2003]. In manchen Fällen wird daher beim Einfügen neuer Datensätze auf die Clusterung verzichtet, wodurch die Sortierreihenfolge verloren geht [Dorendorf2003].

[2] Herkömmliche Indizierung wird in diesem Zusammenhang manchmal auch als *non-clustered Index* bezeichnet.

Die Realisierung dieser Speicherform ist von Datenbank zu Datenbank unterschiedlich. Die Datensätze können dabei in den Blattknoten des Index oder getrennt vom Index in einer eigenen Tabelle gespeichert werden, wobei der erste Fall, die Speicherung der Daten im Indexbaum, als *Indexorganized Table* bezeichnet wird [Dorendorf2003]. Dadurch, dass eine Tabelle nur auf eine festgelegte Weise sortiert werden kann, ergibt sich auch die Einschränkung, dass je Tabelle nur ein *Clustered Index* angelegt werden kann [Agrawal2004a].

PostgreSQL unterstützt sowohl B-tree-Indizierung als auch die genannten speziellen Indextypen [Kulev2007, PostgreSQL2009a]. Bitmap-Indizes (on disc) werden derzeit von PostgreSQL noch nicht unterstützt. Alternativ existiert mit Bizgres[3] eine spezielle Version von PostgreSQL, welche diesen Indextyp unterstützt [Kulev2007]. Weiters kann in PostgreSQL mit dem Befehl `CLUSTER` auch ein *Clustered Index* angelegt werden. Die Sortierung erfolgt anhand eines zuvor angelegten, im Befehl definierten, Index. Allerdings erfolgt die Clusterung immer nur einmal. Das heißt, dass nach dem Einfügen neuer Datensätze diese erst nach neuerlichem Ausführen des `CLUSTER`-Befehls sortiert werden. Während der Clusterung ist die entsprechende Tabelle für andere Datenbankoperationen gesperrt [PostgreSQL2009a].

2.2 Materialized Views

Mit einem Materialized View (deutsch: Materialisierte Sicht) lassen sich definierte Datenbankoperationen vorausberechnen. Die Ergebnisse dieser Operationen werden zur späteren Nutzung in speziellen Tabellen gespeichert [Albrecht2001]. Nachdem ein Materialized View erzeugt wurde, kann dieser dazu benutzt werden, Datenbankabfragen zu beschleunigen [Albrecht2006].

Materialized Views eignen sich besonders für Anwendungen, in welchen Daten kaum verändert werden und die gleichzeitig große Datenmengen verarbeiten müssen. Bei den Datenbankoperationen, die materialisiert (physisch

[3] http://pgfoundry.org/projects/bizgres

gespeichert) werden, handelt es sich meist um aufwendige Joins und Aggregationen [Albrecht2006].

Die Nutzung von Materialized Views lässt sich durch folgende Problembereiche charakterisieren [Goldstein2001]:

- Definition/Design der Materialized Views
- Wartung der Materialized Views
- Nutzung der Materialized Views zur Beschleunigung von Abfragen

Beim Design der Materialized Views sollte zunächst genau analysiert werden, welche Abfragen häufig an die Datenbank gestellt werden und wie kostenintensiv diese Abfragen sind. Häufige und sehr kostenintensive Abfragen bieten sich für Materialized Views an [Albrecht2006]. Es ist auch möglich, nur einen Teil der Berechnungen, welche für die Beantwortung einer Datenbankabfrage notwendig sind, in Materialized Views zu speichern [Halevy2001]. Dabei sollte darauf geachtet werden, das Datenvolumen des Materialized Views gering zu halten, diesen jedoch gleichzeitig möglichst allgemein zu formulieren, um damit unterschiedliche Datenbankabfragen zu unterstützen [Albrecht2006]. Weiters sollte bei der Formulierung darauf geachtet werden, dass beispielsweise nach einer Gruppierung von Werten bestimmte Operationen, wie Summierung oder die Berechnung von Durchschnittswerten, unter Umständen nicht mehr möglich sind und der Materialized View für solche Datenbankoperationen nicht verwendet werden kann [Halevy2001].

Unter der Wartung von Materialized Views wird die Aktualisierung der Daten des Views verstanden, um Änderungen an der Basistabelle zu übernehmen [Goldstein2001]. Es kann zwischen *Complete-Refresh* und *Fast-Refresh* unterschieden werden [Albrecht2006]. Beim Complete-Refresh wird der komplette Materialized View neu berechnet, während beim Fast-Refresh nur die geänderten Werte übernommen werden [Albrecht2001]. Neben der Art der Aktualisierung ist auch der Zeitpunkt ein wichtiger Faktor. Grundsätzlich lässt sich hier unterscheiden, ob eine Aktualisierung sofort nach ei-

ner Änderung oder zu festgelegten (periodischen) Zeitpunkten erfolgt [Albrecht2001].

Die Nutzung von Materialized Views zur Beschleunigung von Datenbankabfragen wird in der Literatur meist unter dem Begriff *Query Rewriting* diskutiert. Unter *Query Rewriting* versteht man, dass ein `SELECT`-Statement in alle möglichen Formen umformuliert und danach jene Formulierung mit den geringsten Kosten ausgeführt wird [Goldstein2001]. Für die Nutzung von Materialized Views bedeutet das, dass eine Datenbankabfrage intern automatisch so umgeschrieben wird, dass sie, wenn dies möglich ist und die Performance der Datenbankabfrage dadurch erhöht wird, vorhandene Materialized Views nutzt. Der Benutzer muss dabei nicht explizit angeben, dass ein Materialized View genutzt werden soll [Albrecht2001]. Weiters kann es auch möglich sein mehr als einen Materialized View zur Beantwortung einer Datenbankabfrage zu verwenden, wenn dadurch jeweils Teilausdrücke der Abfrage berechnet werden können [Chaudhuri1995].

Eine wichtige Aufgabe des *Query Rewriting* ist nicht nur Materialized Views zu identifizieren die logisch verwendet werden können, sondern auch darauf zu achten, ob die Verwendung dieser zu einer besseren Performance führt. Beispielsweise wäre es möglich, dass Indizes existieren, durch welche eine kostengünstigere Abfrage an der/den Basistabelle(n) möglich ist [Halevy2001]. Schon beim View Design sollte beachtet werden, dass die Zeit, die für das *Query Rewriting* benötigt wird, mit der Anzahl der Materialized Views steigen kann [Goldstein2001].

Materialized Views werden in PostgreSQL nicht unterstützt. Eine mögliche Alternative bietet das PG-Snapshot-Projekt[4], welches Materialized Views in PostgreSQL, wenngleich nur eingeschränkt, ermöglicht. Diese Variante weist aber eine schlechtere Performance als native Implementierungen auf [Kulev2007].

[4] http://pgfoundry.org/projects/snapshot

2.3 Partitionierung

Bei der Partitionierung von Datenbanken werden große Tabellen in mehrere kleinere Teiltabellen unterteilt. Dadurch wird nur das physische Design der Tabelle verändert, aus logischer Sicht verhält sich die Tabelle wie vor der Partitionierung [Albrecht2001]. Abfragen sowie das Einfügen, Ändern oder Löschen von Tupeln, erfolgen daher genauso wie vor der Partitionierung der Tabelle[5] [Noyer2007].

Es kann grundsätzlich zwischen zwei verschiedenen Arten, der horizontalen- und der vertikalen Partitionierung, unterschieden werden. Zusätzlich gibt es auch noch mehrere Möglichkeiten, eine Tabelle horizontal zu partitionieren. Bei der horizontalen Partitionierung wird eine Tabelle in mehrere Teiltabellen, bestehend aus Zeilen der Basistabelle, unterteilt. Hingegen wird bei der vertikalen Partitionierung die Basistabelle in mehrere Teiltabellen aus ein oder mehr Spalten aufgeteilt [Agrawal2004a].

Durch beide Arten der Partitionierung kann die Performance von Datenbanken gesteigert werden, weil durch Partitionierung bei manchen Datenbankoperationen auf weniger Daten zugegriffen werden muss beziehungsweise weniger Daten verarbeitet werden müssen [Agrawal2004a]. Im Gegensatz zu Indizes und Materialized Views hat Partitionierung den Vorteil, dass keine Daten repliziert werden und dadurch kein zusätzlicher Speicher- und Update-Aufwand entsteht [Ailamaki2004]. Trotz aller Gemeinsamkeiten handelt es sich bei horizontaler und vertikaler Partitionierung um zwei grundsätzlich verschiedene Techniken, die teilweise unterschiedliche Vorteile mit sich bringen und auch jeweils auf unterschiedliche Art unterstützt und umgesetzt werden.

In den folgenden beiden Unterkapiteln 2.3.1 und 2.3.2 werden diese zwei Arten der Partitionierung detailliert beschrieben.

[5] Bei vertikaler Partitionierung gilt dies allerdings nicht in jedem Fall, da die meisten DBMS keine native Unterstützung für vertikale Partitionierung bieten. Um vertikale Partitionierung dennoch umzusetzen, muss man das logische Design der Datenbank verändern (siehe Kapitel 2.3.1).

2.3.1 Vertikale Partitionierung

Vertikale Partitionierung bezeichnet die Aufteilung einer Tabelle in mehrere Teiltabellen, wobei jede Spalte der Basistabelle jeweils einer der Teiltabellen zugeordnet wird [Agrawal2004a]. Abbildung 2.1 veranschaulicht vertikale Partitionierung anhand eines Beispiels.

Basistabelle:

Student

ID	Vorname	Nachname	Matrikel_Nr	Geb_Datum	Wohnort
1	Daniel	Sturm	0250478	12.05.1981	Pressbaum
2	Max	Müller	0123456	04.01.1980	Wien
3	Moritz	Meier	0234567	05.09.1975	Linz

Vertikal partitionierte Tabellen:

Student_P1

ID	Vorname	Nachname	Matrikel_Nr
1	Daniel	Sturm	0250478
2	Max	Müller	0123456
3	Moritz	Meier	0234567

Student_P2

ID	Geb_Datum	Wohnort
1	12.05.1981	Pressbaum
2	04.01.1980	Wien
3	05.09.1975	Linz

Abbildung 2.1: Beispiel einer vertikal partitionierten Tabelle

Der Vorteil bei dieser Form der Datenspeicherung liegt darin, dass bei einigen Datenbankabfragen auf eine geringere Datenmenge zugegriffen werden muss. Bei einer Abfrage, die nicht alle Spalten der Tabelle benötigt, muss unter Umständen nur eine Teiltabelle gelesen werden [Agrawal2004a]. Wird in dem Beispiel aus Abbildung 2.1, bei vertikaler Partitionierung, nur der Name und die Matrikelnummer der Studenten benötigt, so muss nur die Tabelle *Student_P1* und damit deutlich weniger Daten gelesen werden. Der Nachteil vertikaler Partitionierung liegt freilich darin, dass bei Anfragen, bei welchen alle Attribute einer Zeile ausgewählt werden, die Teiltabellen wieder miteinander verknüpft werden müssen. Diese Verknüpfung stellt einen zusätzlichen Aufwand dar und führt zu einer schwächeren Performance der Datenbank. Des weiteren erfolgt auch das Einfügen neuer Datensätze lang-

samer [Abadi2007].

Gängige DBMS besitzen meist keine native Unterstützung für vertikale Partitionierung. Diese kann jedoch durch Veränderung des logischen Schemas der Datenbank umgesetzt werden. Auch im Beispiel aus Abbildung 2.1 wurde diese Methode angewandt. Die *ID* wird dabei in jeder der Teiltabellen gespeichert, um zusammengehörende Einträge wieder miteinander verknüpfen zu können [Abadi2007].

Ein Spezialfall der vertikalen Partitionierung ist, wenn für jedes Attribut der Basistabelle eine eigene Teiltabelle angelegt wird. Die Anzahl der Teiltabellen entspricht dann der Anzahl der Spalten (Attribute) der Basistabelle. Jede dieser Teiltabellen besitzt zwei Spalten, die *ID* und den Wert des Attributes. Diese Spezialform der vertikalen Partitionierung hat die zusätzliche Eigenschaft, dass mehrere Werte für ein Attribut existieren können. Dazu muss in der jeweiligen Tabelle des Attributes nur eine zusätzliche Zeile mit der selben *ID* eingefügt werden. Außerdem müssen 'leere' Felder nicht erfasst werden. In der entsprechenden Tabelle des Attributes wird in diesem Fall keine Zeile eingefügt [Abadi2007].

Im Zusammenhang mit vertikaler Partitionierung werden oft spaltenorientierte DBMS genannt. Im Gegensatz zu zeilenorientierten Systemen müssen diese nicht jeden Eintrag einer Tabelle (jede Zeile und jede Spalte) lesen, sondern nur jene Spalten, die für die jeweiligen Anfragen benötigt werden, da spaltenorientierte DBMS die Daten nach Spalten organisiert auf den Speichermedien ablegen [Zdonik2008]. Dadurch erreicht man eine ähnliche Speicherstruktur wie bei dem oben erwähnten Spezialfall der vertikalen Partitionierung mit dem Vorteil, dass solche DBMS für diese Art der Datenorganisation optimiert sind [Abadi2007].

2.3.2 Horizontale Partitionierung

Wie bereits beschrieben, wird auch bei horizontaler Partitionierung eine Tabelle in mehrere Teiltabellen unterteilt. Die Unterteilung erfolgt dabei nach

Zeilen. Jede Zeile wird anhand eines bestimmten Attributes jeweils einer Partition zugeordnet [Agrawal2004a, Albrecht2006]. Im Gegensatz zur vertikalen Partitionierung wird horizontale Partitionierung von vielen DBMS unterstützt [Noyer2007].

Neben Performancesteigerungen eignet sich horizontale Partitionierung sehr gut, um Datenbanken mit großen Tabellen leichter handhabbar zu machen [Albrecht2006]. Operationen, wie Backups und die Wiederherstellung von Backups, werden dadurch vereinfacht, vor allem dann, wenn auch Indizes partitionslokal angelegt werden [Agrawal2004a]. Überdies sollten Indizes (vor allem Bitmap-Indizes) möglichst immer partitionslokal angelegt werden [Albrecht2006]. Ein weiterer Vorteil ist, dass Daten relativ einfach archiviert oder gelöscht werden können, indem die entsprechende Partition offline gesetzt oder gelöscht wird [Albrecht2001, Bartl2003]. Dadurch wird allerdings die Auswahl eines geeigneten Schlüsselkriteriums für die Partitionierung zusätzlich erschwert, weil dabei nicht nur die Performance berücksichtigt werden sollte, sondern auch die erleichterte Administration der Datenbank [Agrawal2004a].

Die Performancesteigerung erfolgt bei horizontaler Partitionierung dadurch, dass bei manchen Datenbankabfragen auf weniger Daten zugegriffen werden muss. Daten mit ähnlichen Kriterien werden auf den Speichermedien nahe beieinander liegend gespeichert [Agrawal2004a, Ailamaki2004]. Das DBMS formuliert `SELECT`-Statements automatisch so um, dass nur auf jene Partitionen zugegriffen wird, die zur Beantwortung der Anfrage nötig sind. Dieser Vorgang wird *Partition Pruning* genannt. Der Nutzer kann die Datenbankoperationen dadurch wie gewohnt formulieren [Albrecht2006].

Damit Datenbankoperationen auch wirklich effizient durchgeführt werden können, ist die Auswahl des am besten geeigneten Schlüsselkriteriums für die Partitionierung wichtig. Oft eignen sich zeitliche Werte dazu gut, weil dadurch auch der Forderung nach einer erleichterten Administration (ältere Daten können beispielsweise leicht archiviert werden) nachgekommen wird [Albrecht2006].

Die Wahl eines geeigneten Schlüsselkriteriums für die Partitionierung kann von Anwendung zu Anwendung verschieden sein. Google partitioniert beispielsweise bei der Erfassung von Webseiten nach der URL dieser Seite. Damit zusammenhängende Seiten nahe beieinander liegend gespeichert werden, wird die umgekehrte URL verwendet, also beispielsweise at.ac.wu.ai[6]. Zusätzlich ist, in dem eigens entwickelten Datenbankmanagementsystem bei Google, auch eine Form der vertikalen Partitionierung möglich, damit beispielsweise bei Datenbankanfragen, die auf die Metadaten einer Seite zugreifen, nicht auch immer der gesamte Inhalt der Seite gelesen werden muss. Eine Partition wird dabei nicht größer als 100-200 MB. Wächst eine Partition über diese Grenze hinaus, so wird diese automatisch gesplittet [Chang2006].

Bei horizontaler Partitionierung wird zwischen drei verschiedenen Arten, Range-, List-, und Hash-Partitionierung, unterschieden. Zusätzlich sind auch Kombinationen aus diesen möglich.

Range-Partitionierung teilt eine Tabelle in verschiedene Wertebereiche auf. Eine typische Unterteilung wäre eine nach zeitlichen Wertebereichen, wie Monate, Quartale oder Jahre [Albrecht2006]. Bei einer Partitionierung nach Jahren wird beispielsweise für jedes Jahr eine Tabelle angelegt, in der ausschließlich alle Daten eines bestimmten Jahres gespeichert werden. Enthält die `WHERE`-Klausel einer Abfrage eine Einschränkung auf ein bestimmtes Jahr oder auf einen Bereich in einem bestimmten Jahr (z. B. Mai, 2009), dann muss nur auf die Tabelle dieses Jahres zugegriffen werden.

Ein ähnliches Prinzip wird bei der List-Partitionierung angewandt. Hier wird allerdings eine Tabelle nicht nach Wertebereichen, sondern nach diskreten Werten unterteilt, wodurch auch eine Partitionierung nach Regionen, Filialen oder ähnlichen Attributslisten möglich wird [Albrecht2001, Albrecht2006]. Wiederum muss bei Datenbankabfragen, die beispielsweise auf eine bestimmte Region beschränkt sind, nur eine Partition gelesen werden, sofern die Tabelle nach Regionen partitioniert wurde.

[6] http://ai.wu.ac.at/ - Die Website des Instituts für Informationswirtschaft an der Wirtschaftsuniversität Wien

Im Gegensatz zu Range- und List-Partitionierung wird bei Hash-Partitionierung die Performancesteigerung nicht durch Bereichseinschränkungen erzielt. Hash-Partitionierung hat zum Ziel, die Daten möglichst gleichmäßig auf eine festgelegte Anzahl von Partitionen zu verteilen [Albrecht2006]. Jeder Tupel einer Tabelle wird bei dieser Partitionierungsmethode mittels einer Hash-Funktion (über eine oder mehrere Spalten) genau einer Partition zugeordnet [Agrawal2004a]. Bei Hash-Partitionierung wird auf eine Performancesteigerung durch parallele Zugriffe gesetzt. Die gleichmäßige Verteilung der Daten auf mehrere Partitionen soll die Datenbank für parallele Operationen optimieren. Oft wird diese Parallelisierung und Datenverteilung auf die Ebene des Speichermedien-Systems verlagert, wodurch der Hash-Partitionierung eine geringere Bedeutung zukommt [Albrecht2006].

Zusätzliche Möglichkeiten der Partitionierung bieten sich durch die Verwendung von Subpartitionen, soweit diese Form der Partitionierung vom DBMS unterstützt wird. Dabei wird eine Partition einer Tabelle nochmals in Subpartitionen unterteilt. Diese Form der Partitionierung wird auch als Composite-Partitioning (beziehungsweise je nach verwendeter Partitionierungsmethode als Range-List-Partitionierung, Range-Hash-Partitionierung usw.) bezeichnet [Albrecht2006]. Eine Besonderheit von PostgreSQL ist, dass Subpartitionierung nicht symmetrisch erfolgen muss. Das bedeutet, dass einzelne Partitionen mehr, weniger oder gar keine Subpartitionen enthalten können [Kulev2007]. Überdies weist PostgreSQL noch weitere Besonderheiten bei der Partitionierung auf, welche in den nächsten Absätzen beschrieben werden.

Horizontale Partitionierung ist in PostgreSQL etwas umständlicher als in manchen anderen DBMS. In Oracle-Datenbanken etwa wird eine partitionierte Tabelle beim Anlegen durch

```
CREATE TABLE name_der_tabelle (...)
    PARTITION BY LIST (spaltenname) (
        PARTITION name_der_partition1 VALUE (attributswert1)
        PARTITION name_der_partition2 VALUE (attributswert2)
    );
```

erstellt [Albrecht2001, Noyer2007]. PostgreSQL hingegen kennt das Schlüsselwort `PARTITION` nicht, partitionierte Tabellen werden durch *Vererbung*, *Trigger* und *'Constraint Exclusion'* erzeugt [Kulev2007].

Zum Erzeugen einer partitionierten Tabelle in PostgreSQL sind fünf Schritte notwendig. Zunächst wird eine Master-Tabelle angelegt. In der Master-Tabelle werden nur die Spalten definiert. Daten sind in der Master-Tabelle keine enthalten. Im nächsten Schritt werden die Kind-Tabellen erzeugt. Diese erben die Spalten und gegebenenfalls '*Check-Constraints*' der Master-Tabelle. Jede dieser Kind-Tabellen wird mit einem '*Check-Constraint*' versehen, welches den Wertebereich für die jeweilige Partition definiert. Dabei muss darauf geachtet werden, dass sich diese Wertebereiche nicht überschneiden. Danach kann jede Partition gegebenenfalls noch indiziert werden [PostgreSQL2009a]. Listing 2.1 zeigt diese ersten drei Schritte anhand eines kurzen Beispiels.

Listing 2.1: Beispiel für horizontale Partitionierung in PostgreSQL - Teil I

```
-- 1. Schritt: anlegen der Master-Tabelle
CREATE TABLE student (
    Mat_No     int,
    first_name text,
    last_name  text,
    birthdate  date
);

-- 2. Schritt: anlegen der Kind-Tabellen
CREATE TABLE student_partition_1 (
    CHECK (birthdate < DATE '1980-01-01')
) INHERITS (student);
CREATE TABLE student_partition_2 (
    CHECK (birthdate >= DATE '1980-01-01')
) INHERITS (student);

-- 3. Schritt: anlegen der Indizes (optional)
CREATE INDEX student_partition_1_birthdate
    ON student_partition_1 (Mat_No);
CREATE INDEX student_partition_2_birthdate
    ON student_partition_2 (Mat_No);
```

Nachdem alle benötigten Tabellen angelegt wurden, müssen die Daten beim Einfügen noch in die richtige Partition geleitet werden. Dazu wird, wie in Listing 2.2 gezeigt wird, eine Trigger-Funktion angelegt, welche diese Aufgabe übernimmt. Ein Trigger ruft bei jedem `INSERT`-Statement auf die Master-Tabelle die Trigger-Funktion auf, welche die Daten in die Partitionen umleitet. Werden zu einem späteren Zeitpunkt zusätzliche Partitionen angelegt, so muss die Trigger-Funktion auch entsprechend erweitert werden. Statt einer Trigger-Funktion können auch Regeln (`RULE`s) verwendet werden. Diese haben aber einige Nachteile gegenüber Trigger-Funktionen, weshalb sich Trigger-Funktionen besser zur Partitionierung eignen [PostgreSQL2009a].

Listing 2.2: Beispiel für horizontale Partitionierung in PostgreSQL - Teil II

```
-- 4. Schritt: anlegen einer Trigger-Funktion
CREATE OR REPLACE FUNCTION student_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.birthdate < DATE '1980-01-01' ) THEN
        INSERT INTO student_partition_1 VALUES (NEW.*);
    ELSIF ( NEW.birthdate >= DATE '1980-01-01' ) THEN
        INSERT INTO student_partition_2 VALUES (NEW.*);
    ELSE RAISE EXCEPTION 'Error in student_insert_trigger function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

-- 5. Schritt: anlegen eines Triggers welcher die Trigger-Funktion aufruft
CREATE TRIGGER insert_student_trigger
    BEFORE INSERT ON student
    FOR EACH ROW EXECUTE PROCEDURE student_insert_trigger();
```

Wie bereits erwähnt, wird bei Partitionierung in PostgreSQL eine spezielle Technik zum Optimieren einer Anfrage, die sogenannte '*Constraint Exclusion*', verwendet. Diese Technik ermöglicht es, dass bei Abfragen nur die benötigten Partitionen durchsucht werden. Die Auswahl der Partitionen erfolgt dabei über die *Constraints* einer Tabelle [PostgreSQL2009a]. Um '*Constraint Exclusion*' in PostgreSQL zu aktivieren, muss die Varia-

ble `constraint_exlusion` in der Datei 'postgresql.conf' auf den Wert '`on`' gesetzt werden. Dies hat allerdings zur Folge, dass *Constraints* bei jeder Abfrage berücksichtigt werden müssen. Ab PostgreSQL Version 8.4 gibt es daher auch die Möglichkeit, die Variable `constraint_exlusion` auf den Wert '`partition`' zu setzen. Dadurch wird '*Constraint Exclusion*' nur bei partitionierten Tabellen verwendet [PostgreSQL2009a, PostgreSQL2009b].

Bei Partitionierung in PostgreSQL sind noch einige Problembereiche vorhanden, so muss beispielsweise für Updates, die den Wert der Spalte verändern, nach welcher partitioniert wurde, wiederum eine zusätzliche Trigger-Funktion geschrieben werden [PostgreSQL2009a]. Weiters werden bei Vererbung in PostgreSQL Primärschlüssel und Fremdschlüssel an die Kind-Tabellen nicht weiter vererbt. Diese müssen für jede Kind-Tabelle angelegt werden [PostgreSQL2009a, Noyer2007].

2.4 Tablespaces

Im Gegensatz zu den bisher vorgestellten Technologien werden Tablespaces nicht unbedingt zur Performancesteigerung von Datenbanken verwendet, sondern primär um die Handhabbarkeit, die Administration, von Datenbanken zu vereinfachen. Vor allem in Verbindung mit partitionierten Tabellen können Tablespaces, wie wir in diesem Kapitel noch sehen werden, insbesonders bei Systemen mit beschränktem Speicherplatz auf schnellen Speichermedien auch die Performance der Datenbank mit beeinflussen.

Tablespaces sind Speichereinheiten einer Datenbank, die jeweils mehrere Tabellen beinhalten können. Eine Tabelle muss immer einem bestimmten Tablespace zugeordnet sein. Dies gilt für jede Form von Tabellen, also auch für Indizes oder Partitionen [Albrecht2001]. Der Vorteil von Tablespaces liegt einerseits darin, dass damit die Daten auf verschiedene Speichermedien verteilt werden können. Dies können manche DBMS ausnutzen, um Engpässe bei Schreib- und Lesevorgängen zu verhindern [Bartl2003]. Andererseits erleichtern Tablespaces manche Aufgaben beim Managen großer

Datenbanken. So können Tablespaces beispielsweise schnell und relativ unkompliziert in andere Datenbanken verschoben oder einzelne Tabellen in andere Tablespaces, und damit auf andere Speichereinheiten, transferiert werden [Noyer2007].

Insbesondere bei partitionierten Tabellen lässt sich durch den Einsatz von Tablespaces in manchen Fällen die Performance zusätzlich beeinflussen. Bei Partitionierung werden bekanntlich sehr große Tabellen auf mehrere kleine Tabellen aufgeteilt. Wenn nun bekannt ist, dass auf einige Daten öfter zugegriffen wird als auf andere, so können diese auf teure, aber schnellere Speichermedien verlagert werden. In Systemen mit einer beschränkten Anzahl schneller Speichermedien kann dies einen signifikanten Einfluss auf die Performance haben [Noyer2007].

In PostgreSQL lässt sich ein Tablespace durch den Befehl

```
CREATE TABLESPACE tablespace_name
    LOCATION '/pfad/zu/einem/leeren/ordner';
```

anlegen. Voraussetzung ist, dass es sich dabei um einen existierenden, leeren Ordner handelt. Eine weitere Voraussetzung ist, dass das zugrunde liegende Betriebssystem symbolische Verknüpfungen unterstützt. Ist dies nicht der Fall, so können keine Tablespaces angelegt werden [PostgreSQL2009a].

Tabellen können mit

```
CREATE TABLE tabellen_name TABLESPACE tablespace_name;
```

einem bestimmten Tablespace zugewiesen werden. Indizes dieser Tabelle werden nicht automatisch im selben Tablespace gespeichert. Diese müssen, wenn dies benötigt wird, explizit einem Tablespace zugewiesen werden [PostgreSQL2009a].

3 Softwarelösungen für den Umgang mit großen Datenmengen

In den folgenden Unterkapiteln werden Softwarelösungen vorgestellt, die interessante Konzepte für den Umgang mit großen Datenmengen bereithalten. Die Softwarelösungen werden dabei keiner tief gehenden Analyse unterzogen. Es sollen vielmehr die grundlegenden Funktionen skizziert werden, um einen Überblick darüber zu bekommen, welche Lösungsansätze in den verschiedenen Produkten für den Umgang mit großen Datenmengen vorhanden sind. Das Kapitel soll dabei auch zur Vorbereitung von Lösungsansätzen für die spätere Implementierung dienen.

3.1 PL/Proxy

PL/Proxy [PlProxy2009a, PlProxy2009b] wurde von Skype[7] entwickelt, um die Skalierbarkeit deren PostgreSQL Datenbank zu gewährleisten. Die PostgreSQL Erweiterung wurde als eine in die Datenbank eingebettete prozedurale Programmiersprache implementiert und unter der BSD-Lizenz als open-source Projekt veröffentlicht [PlProxy2009c].

Die grundsätzliche Funktion von PL/Proxy ist, Remote-Funktionsaufrufe auf anderen Datenbanken durchzuführen. Dazu muss die Funktion nur den Namen der Remote-Datenbank oder eines Datenbank-Clusters beinhalten. Danach wird automatisch jene Funktion mit der selben Signatur wie die

[7]http://www.skype.com; https://developer.skype.com/SkypeGarage

PL/Proxy-Funktion auf der Remote-Datenbank aufgerufen. Somit kann beispielsweise durch die Funktion

```
CREATE FUNCTION get_email(name text) RETURNS text AS $$
    CONNECT 'dbname=mydb';
$$ LANGUAGE plproxy;
```

die gleichnamige Funktion *get_email(name text)* auf der Datenbank *'mydb'* aufgerufen werden [PlProxy2009a].

Für die fortlaufende Erfassung und Speicherung großer Datenmengen ist PL/Proxy deshalb interessant, weil damit eine horizontale Partitionierung der Daten auf mehrere Datenbanken durchgeführt werden kann. Dazu wird ein Datenbank-Cluster, bestehend aus zwei oder mehr Datenbanken, definiert. Anhand einer Funktion (beispielsweise einer hash-Funktion) wird die Partition (Datenbank) ermittelt, und der Funktionsaufruf erfolgt dann nur auf dieser Datenbank. Wird zum Beispiel bei der Funktion

```
CREATE FUNCTION get_email(name text) RETURNS text AS $$
    CLUSTER 'mycluster';
    RUN ON hashtext(name);
$$ LANGUAGE plproxy;
```

beim Aufruf von *hashtext(name)* der Wert '2' zurückgeliefert, so wird die entsprechende Remote-Funktion nur auf der zweiten Datenbank im Datenbank-Cluster ausgeführt. Ebenso muss natürlich beim Einfügen von neuen Daten vorgegangen werden [PlProxy2009a].

Neben Funktionsaufrufen können auch normale `SELECT`-Statements auf entfernten Datenbanken ausgeführt werden. Hierzu muss allerdings wiederum für jedes `SELECT`-Statement eine eigene PL/Proxy-Funktion erstellt werden [PlProxy2009a].

3.2 RRDtool - Round Robin Database

Das *Round Robin Database Tool* (RRDtool) wurde zur Speicherung und Analyse von Zeitreihendaten (wie beispielsweise Netzwerk- oder Systemaus-

lastung oder auch Temperaturmessungen uvm.) entwickelt [RRDtool2009a]. Was dieses Tool im Zusammenhang mit dieser Arbeit interessant macht, ist die Art und Weise, wie dieses die benötigten Daten speichert. Wenngleich in RRDtool nur numerische Daten gespeichert werden können [RRDtool2009a], bietet es dennoch einen interessanten Ansatz zur Lösung des Speicherplatzproblems bei der Archivierung von Daten aus Datenströmen und den sich dabei ergebenden großen Datenmengen.

RRDtool verwendet zum Speichern der Daten, wie der Name schon sagt, eine sogenannte *Round Robin Database* (RRD). In einer RRD werden Daten für eine bestimmte Zeit gespeichert. Dabei bleibt der verwendete Speicherplatz immer konstant. Dies wird realisiert, indem für eine bestimmte Anzahl von Datenpunkten Speicherplatz geschaffen wird. Ist der gesamte Speicherplatz belegt, so werden alte Daten überschrieben. Wird also eine Datenbank für 100 Einträge angelegt, würde der 101. Eintrag den ersten Eintrag überschreiben [RRDtool2009b].

Um ältere Daten trotzdem verfügbar zu halten, kann eine RRD mehrere *Round Robin Archive* (RRA) beinhalten. In jedem dieser RRA werden die Daten auf unterschiedliche Art konsolidiert. Die Konsolidierung erfolgt anhand eines vorherbestimmten Intervalls und einer Konsolidierungsfunktion (z.B. Durchschnittswert, Minimum, Maximum, ...). Somit können beispielsweise Durchschnittswerte für jeden Tag, jedes Monat und jedes Jahr gespeichert werden. Ältere Werte liegen nach einer gewissen Zeit nicht mehr auf Tagesbasis vor, können aber in aggregierter Form (z.B Durchschnittswert pro Monat oder Jahr) noch in Analysen einfließen [RRDtool2009b].

Durch diese Art des Datenbankmanagements kann der Speicherplatz konstant gehalten werden, und durch Verwendung mehrerer RRA sind diese trotzdem für lange Zeit verfügbar. Je länger die Daten zurückliegen, desto stärker werden sie allerdings verdichtet. Durch den gleich bleibenden Speicherverbrauch erfordert diese Form der Datenbank kaum Wartungsarbeiten [RRDtool2009b].

3.3 Tabellenpartitionierung bei IBM DB2 9 und Oracle Database 11g

In diesem Kapitel soll beschrieben werden, wie Tabellenpartitionierung bei den beiden kommerziellen Datenbanken, DB2 9 von IBM und Oracle Database 11g, unterstützt wird. Dabei sollen die Funktionen und SQL-Statements aus Sicht des Benutzers (Datenbankadministrator, Datenbankdesigner, ...) vorgestellt werden. An dieser Stelle soll daher auch keine detaillierte und in keinster Weise vollständige Beschreibung erfolgen, wie Tabellenpartitionierung in diesen Datenbankmanagementsystemen umgesetzt wird. Vielmehr soll hervorgehoben werden, wie und in welchen Bereichen der Umgang mit Partitionierung, im Vergleich zu PostgreSQL, für den Benutzer vereinfacht wird.

3.3.1 IBM DB2 9

In DB2 9 Datenbanken wird zwischen Datenbankpartitionierung und Tabellenpartitionierung unterschieden. Datenbankpartitionierung bezeichnet die gleichmäßige Aufteilung von Daten auf verschiedene Datenbanken, wohingegen Tabellenpartitionierung eine Tabelle horizontal auf mehrere Teiltabellen aufteilt. Diese zwei Arten der Partitionierung können auch gemeinsam eingesetzt werden. Im Folgenden befasst sich dieses Unterkapitel nur mit der Tabellenpartitionierung in DB2 9 [Ahuja2006].

Grundsätzlich wird in DB2 9 nur Range-Partitionierung unterstützt. List-Partitionierung wird nicht direkt unterstützt. Allerdings ist es möglich, generierte Spalten für die Partitionierung zu nutzen. Indem man für jede mögliche Attributsausprägung in einer zusätzlichen Spalte einen numerischen Wert erzeugt, lässt sich eine List-Partitionierung simulieren [Chen2007].

Mit DB2 9 können Partitionen sowohl automatisch als auch manuell angelegt werden. Automatisches Anlegen von Partitionen wird durch die `EVERY`-Klausel im '`PARTITION BY`'-Statement gekennzeichnet. Beim automatischen

Anlegen werden mehrere Partitionen mit gleich großen Wertebereichen des Schlüsselattributes erzeugt. Die `EVERY`-Klausel gibt das Intervall der erzeugten Partitionen an. Durch den Befehl

```
CREATE TABLE foo (date_column DATE)
    PARTITION BY RANGE(date_column)
        (STARTING ('1/1/2000') ENDING ('12/31/2010') EVERY 6 MONTH);
```

werden für den Zeitraum zwischen 1.1.2000 und 31.12.2010 jeweils für jedes Jahr zwei Partitionen erzeugt. Es würden somit 20 Partitionen angelegt werden für die Wertebereiche 1.1.2000 bis exklusive 1.7.2000, 1.7.2000 bis exklusive 1.1.2001, ..., 1.7.2010 bis inklusive 31.12.2010 [Chen2007].

Beim manuellen Anlegen von Partitionen wird der Wertebereich jeder Partition einzeln durch die `STARTING`- und `ENDING`-Klausel angegeben. Wird keine `STARTING`- oder keine `ENDING`-Klausel angegeben, so wird der fehlende Wert durch die `ENDING`-Klausel der vorhergehenden beziehungsweise die `STARTING`-Klausel der nachfolgenden Partition bestimmt. Durch die Ausdrücke `MINVALUE` und `MAXVALUE` können Partitionen mit nach unten beziehungsweise oben offenen Bereichen erzeugt werden. Zusätzlich kann durch das Anfügen einer `IN`-Klausel jede Partition in einem bestimmten Tablespace angelegt werden. Durch das Statement

```
CREATE TABLE foo (a int)
    PARTITION BY RANGE(a)
        (PART partition_1 STARTING MINVALUE IN tablespace_1,
         PART partition_2 STARTING FROM 1 ENDING AT 10 IN tablespace_2,
         PART partition_3 ENDING AT 50 IN tablespace_3,
         PART partition_4 ENDING AT 55 IN tablespace_4,
         PART partition_5 ENDING MAXVALUE IN tablespace_5);
```

wird beispielsweise eine Tabelle mit manuell angelegten Partitionen erzeugt. Zusätzlich zeigt dieses Beispiel die Verwendung der `IN`-Klausel und der Argumente `MINVALUE` und `MAXVALUE` [Chen2007].

Mit dem SQL-Statement '`ALTER TABLE ...`' können zusätzliche Partitionen erzeugt und einzelne Partitionen ausgehängt beziehungsweise eingehängt werden. '`ALTER TABLE ... ADD PARTITION ...`' erzeugt eine neue

Partition innerhalb einer bereits partitionierten Tabelle. Eine nützliche Funktion ist, dass Partitionen durch einen einfachen Befehl aus- oder eingehängt werden können. So ist es möglich, einzelne Partitionen in eine eigene Tabelle zu überführen und damit das Archivieren alter Daten zu erleichtern. Das Aus- und Einhängen erfolgt durch das SQL-Statement '`ALTER TABLE ... DETACH PARTITION ... INTO ...`', beziehungsweise '`ALTER TABLE ... ATTACH PARTITION ... FROM ...`' [Chen2007].

3.3.2 Oracle Database 11g

Oracle Database 11g erlaubt eine Vielzahl unterschiedlicher Partitionierungsmethoden. Es werden Range-, List-, Hash- und Composite-Partitionierung unterstützt [Morales2009]. Im Folgenden werden die Funktionen und die Handhabung der Range- und List-Partitionierung betrachtet.

Bei Range-Partitionierung in Oracle Database 11g wird immer nur die Obergrenze der Partition definiert. Das bedeutet, dass die Partition, welche die niedrigsten Werte des Schlüsselattributes enthält, immer nach unten offen ist (d.h. keine untere Grenze besitzt). Die Obergrenze wird durch die '`VALUES LESS THAN`'-Klausel bestimmt. Wie bei DB2 kann durch den Parameter `MAXVALUE` eine Partition ohne Obergrenze definiert werden, und eine jede Partition kann mit dem `TABLESPACE`-Statement direkt in einem bestimmten Tablespace angelegt werden. Durch Anfügen des Parameters '`ENABLE ROW MOVEMENT`' lässt sich festlegen, dass, wenn sich das Schlüsselattribut eines Tupels verändert, das entsprechende Tupel in eine andere Partition verschoben wird [Morales2009].

Eine Besonderheit bei Oracle Database 11g ist die Intervall-Partitionierung. Diese stellt eine Erweiterung zur Range-Partitionierung dar. Dazu muss zunächst mindestens eine Range-Partition definiert werden. Danach werden, abhängig von dem in der `INTERVAL`-Klausel festgelegten Intervall, automatisch neue Partitionen erstellt, wenn der Wert des Schlüsselattributes außerhalb der Bereiche der bisher existierenden Partitionen liegt. Diese Vorgehensweise erleichtert die Wartung von Range-Partitionen erheblich, da das

Erstellen neuer Partitionen völlig automatisiert abläuft. Das folgende Beispiel zeigt das Anlegen einer Range-Partition mit Intervall-Partitionierung [Morales2009]:

```
CREATE TABLE foo (time_column DATE)
    PARTITION BY RANGE(time_column)
    INTERVAL(NUMTOYMINTERVAL(1,'MONTH'))
        (PARTITION p1 VALUES LESS THAN
            (TO_DATE('1-1-2009', 'DD-MM-YYYY')) TABLESPACE tbs_1,
         PARTITION p2 VALUES LESS THAN
            (TO_DATE('1-1-2010', 'DD-MM-YYYY')) TABLESPACE tbs_2)
ENABLE ROW MOVEMENT;
```

Für List-Partitionierung wird analog zur Range-Partitionierung die Klausel '`PARTITION BY LIST`' verwendet. Die zulässigen Werte des Schlüsselattributes für jede Partition werden durch die `VALUES`-Klausel bestimmt. Eine Partition, die mit dem Wert `DEFAULT` versehen ist, kann alle Werte erfassen, die in keine der anderen Partitionen passen. Das SQL-Statement

```
CREATE TABLE foo (v varchar2(2))
    PARTITION BY LIST(v)
        (PARTITION p1 VALUES ('AB', 'BC'),
         PARTITION p2 VALUES ('CD'),
         PARTITION p3 VALUES (DEFAULT));
```

erstellt eine Tabelle mit drei Partitionen, wobei Partition *p3* alle Werte aufnimmt, die nicht in die Partitionen *p1* oder *p2* passen, also beispielsweise die Werte 'DE' oder 'EF' [Morales2009].

Oracle Database 11g stellt zusätzlich mehrere Funktionen für die Wartung von partitionierten Tabellen bereit. So kann eine Partition mit den Statements '`ALTER TABLE ... DROP PARTITION`' und '`ALTER TABLE ... ADD PARTITION`' leicht gelöscht beziehungsweise eine neue Partition hinzugefügt werden. Ähnlich wie bei DB2 ist es auch möglich, mit der Anweisung '`ALTER TABLE ... EXCHANGE PARTITION`' einzelne Partitionen auszuhängen, das heißt die Partition in eine nicht-partitionierte Tabelle zu konvertieren beziehungsweise auch umgekehrt wieder einzuhängen. Einen Vorteil beim Umgang mit großen Datenmengen bringen außerdem auch die Statements '`ALTER TABLE ... SPLIT PARTITION`' und '`ALTER TABLE`

... MERGE PARTITION', womit sich Partitionen teilen oder zwei Partitionen vereinen lassen. Dadurch, dass sich Partitionen vereinen lassen, können beispielsweise aktuelle Daten auf sehr kleinen Partitionen mit hoher Performance gehalten werden (z.B. auf Tagesbasis) und ältere Daten in größere Partitionen integriert werden (z.B. zuerst auf Monatsbasis und später auf Jahresbasis). Durch Teilen von Partitionen wird es wiederum möglich, sehr große Partitionen mit einer aufgrund deren Größe schwachen Performance in zwei kleinere zu unterteilen, welche wieder eine bessere Performance aufweisen können [Morales2009].

4 Softwaretools zur Analyse der Nutzung und Performance von PostgreSQL Datenbanken

Bevor man mit der Optimierung einer bestehenden Datenbank beginnt, ist es wichtig, die Nutzung und Performance der Datenbank zu analysieren. Erst nach einer solchen Analyse lässt sich beurteilen, welche Maßnahmen sich am besten dazu eignen, die Performance einer Datenbank zu steigern. Die Notwendigkeit einer eingehenden Analyse lässt sich am Beispiel einer partitionierten Tabelle veranschaulichen. Wenn eine Tabelle, welche Daten über Studenten beinhaltet, beispielsweise nach deren Geburtsdatum partitioniert wurde, in den `WHERE`-Klauseln der Datenbankabfragen auf diese Tabelle die Auswahl aber meist nur auf Studienrichtungen oder den Zeitpunkt der Inskription eingeschränkt wird, so hätte die Partitionierung dieser Tabelle nicht den erwarteten Effekt einer spürbaren Performancesteigerung. Daher ist es wichtig, Statistiken darüber zu erstellen, wie Nutzer und Anwendungen auf die jeweilige Datenbank zugreifen.

In diesem Kapitel werden daher Softwaretools vorgestellt, die den Datenbankadministrator bei der Analyse der Nutzung und Performance von Datenbanken unterstützen sollen. Die Auswahl der vorgestellten Tools erfolgte nach den Kriterien, dass diese zur Analyse von PostgreSQL Datenbanken geeignet sind und unter einer open-source Lizenz veröffentlicht werden.

Die vorgestellten Softwaretools lassen sich in zwei Kategorien einteilen. *Practical Query Analysis*, *Enterprise Practical Query Analyzer* und *pg-*

Fouine sollen beim Auswerten von PostgreSQL Log-Files[8] unterstützen. In diesen Log-Files kann jedes SQL-Statement inklusive der Dauer und auftretenden Fehlermeldungen erfasst werden[9]. Mit *pgstatspack* und *pg_statsinfo* können interne Statistiken aus PostgreSQL Datenbanken, wie die Nutzung einzelner Tabellen und Indizes, erfasst werden.

4.1 Practical Query Analysis (PQA)

Practical Query Analysis (PQA) [PQA2009a, PQA2009b] ist ein Softwaretool, das Datenbankadministratoren und Softwareentwickler bei der Analyse von PostgreSQL-Log-Dateien unterstützen soll. Das Tool wurde in der Programmiersprache *Ruby* geschrieben und kann sowohl Text- als auch HTML-Berichte erstellen.

Der Funktionsumfang von PQA ist im Vergleich mit ähnlichen Tools etwas eingeschränkt und auf das Wesentliche reduziert. Ein PQA-Bericht beinhaltet eine allgemeine Statistik, welche unter anderem Informationen über die Gesamtanzahl und Gesamtdauer der Datenbankoperationen bietet. Anschließend werden in den Berichten die für eine Analyse der Datenbank wertvolleren Informationen geliefert. Neben den SQL-Statements, die insgesamt die meiste Zeit (Summe aller Ausführungszeiten eines bestimmten SQL-Statements im Beobachtungszeitraum) in Anspruch genommen haben, werden noch die langsamsten und die häufigsten Datenbankoperationen aufgelistet. Die Anzahl der in diesen Listen angezeigten SQL-Statements kann je nach Bedarf festgelegt werden. Listing 4.1 zeigt ein Beispiel eines solchen PQA-Berichtes[10].

[8]Das Schreiben von Log-Einträgen kann in der PostgreSQL-Konfigurationsdatei aktiviert und danach wieder deaktiviert werden [PostgreSQL2009a].

[9]PostgreSQL erlaubt es, genau festzulegen, welche Einträge das Log-File enthalten soll, so können beispielsweise auch nur solche SQL-Statements erfasst werden, deren Ausführungszeit eine bestimmte Anzahl von Millisekunden überschreitet [PostgreSQL2009a].

[10]Der Bericht wurde aus einem Log-File erstellt, welches im Paket pqa-1.6 als Beispiel enthalten ist.

Listing 4.1: Beispiel eines PQA Berichtes zur Analyse von Datenbanken

```
$ ./pqa.rb --file /home/daniel/sampleLog/pglog_sample.log --normalize --top 3
######## Overall statistics
702 queries (70 unique), longest ran in 0.370524 seconds), parsed in 0.336212 seconds
######## Queries by type
SELECTs: 650 (93%)
INSERTs: 51  (7%)
UPDATEs: 1 (0%)
######## Queries that took up the most time
1.450 seconds: SELECT sum(commits) AS commits,sum(adds) AS adds
        FROM stats_cvs_group WHERE group_id={ }
0.545 seconds: INSERT INTO
        activity_log  (day,hour,group_id,browser,ver,platform,time,page,type)
        VALUES ({ },{ },{ },{ },{ },{ },{ },{ },{ });
0.375 seconds: SELECT count(*) AS count FROM users WHERE status={ }
        and user_id != { }
######## Slowest queries
0.371 seconds: SELECT { } FROM ONLY "public"."country_code" x
        WHERE "ccode" = $1 FOR UPDATE OF x
0.195 seconds: INSERT
        INTO activity_log (day,hour,group_id,browser,ver,platform,time,page,type)
        VALUES ({ },{ },{ },{ },{ },{ },{ },{ },{ });
0.154 seconds: INSERT
        INTO activity_log (day,hour,group_id,browser,ver,platform,time,page,type)
        VALUES ({ },{ },{ },{ },{ },{ },{ },{ },{ });
######## Most frequent queries
61 times: SELECT total FROM forum_group_list_vw WHERE group_forum_id={ }
46 times: INSERT
        INTO activity_log (day,hour,group_id,browser,ver,platform,time,page,type)
        VALUES ({ },{ },{ },{ },{ },{ },{ },{ },{ });
46 times: SELECT plugin_id, plugin_name FROM plugins
```

Eine wichtige Funktion für die Analyse der Log-Files ist die Normalisierung von Datenbankoperationen. Normalisierung bedeutet in diesem Zusammenhang, dass Werte unter Anführungszeichen für die Analyse entfernt werden. Dadurch werden beispielsweise die Datenbankabfragen

```
SELECT * FROM student WHERE Martrikelnummer = '0123456';
```

und

```
SELECT * FROM student WHERE Martrikelnummer = '1234567';
```

als die selbe Datenbankabfrage gewertet. In die Analyse würden beide `SELECT`-Statements anschließend als

SELECT * FROM student WHERE Martrikelnummer = '{}';

einfließen. Durch diese Maßnahme werden ähnliche SQL-Statements aggregiert, wodurch die Analyse vereinfacht werden kann. Normalisierung wird auch im Beispiel in Listing 4.1 durch den Parameter `normalize` verwendet.

Anmerkung:

Das Tool scheint Probleme beim Arbeiten mit neueren Log-Files zu haben (siehe dazu auch die Foreneinträge auf der Projekt-Homepage). Damit PQA mit Log-Files einer PostgreSQL Datenbank der Version 8.3 korrekt arbeiten kann, muss (dies ist in der offiziellen Dokumentation nicht vermerkt) in der PostgreSQL Konfigurationsdatei der Parameter `log_line_prefix = '%t [%p] '` gesetzt werden.

4.2 Enterprise Practical Query Analyzer (epqa)

Das Softwaretool *enterprise practical query analyzer (epqa)* [EPQA2009a, EPQA2009b] ist eine Alternative zu PQA. Das Tool ist nahezu ident mit dem im vorherigen Kapitel vorgestellten Tool PQA. Im Gegensatz zu diesem ist epqa in der Programmiersprache *Perl* geschrieben und erzeugt ausschließlich HTML-Berichte. Des weiteren normalisiert epqa immer jedes SQL-Statement. Anders als bei PQA ist es nicht möglich, zwischen normalisierten und nicht-normalisierten Analysen zu wählen.

Zusätzlich ist es mit epqa auch möglich, komprimierte Dateien zu analysieren. Außerdem werden auch Fehlermeldungen und die am schnellsten ausgeführten Datenbankoperationen in einem epqa-Bericht erfasst.

4.3 pgFouine

PgFouine [pgFouine2009a, pgFouine2009b] bietet von allen hier vorgestellten Log-Analyzer-Tools den größten Leistungsumfang, gemessen an den verfügbaren Funktionen (Analysemöglichkeiten). Das Softwaretool ist in der Programmiersprache *php* geschrieben und kann wie PQA sowohl Text- als auch HTML-Berichte erstellen. In gekennzeichneten Bereichen des Berichtes erfolgt die Auswertung standardmäßig anhand von normalisierten SQL-Statements. In HTML-Berichten können zudem in diesen Bereichen jeweils Beispiele der nicht-normalisierten SQL-Statements angezeigt werden. Dadurch, dass pgFouine auch vom `stdin` lesen kann, ist es auch möglich, komprimierte Log-Files beim Entpacken direkt an das Tool weiterzuleiten. Dies soll den Umgang mit komprimierten Log-Files vereinfachen.

Neben den Statistiken über SQL-Statements mit der längsten Gesamtdauer (Summe aller Aufrufe), SQL-Statements mit der längsten Laufzeit und den am häufigsten ausgeführten SQL-Statements bietet pgFouine auch die Möglichkeit, Berichte über Fehlermeldungen und Statistiken über die stündliche Anzahl von Datenbankoperationen und deren durchschnittliche Ausführungszeiten zu erstellen. Letztere können auch anhand von Grafiken dargestellt werden. Zusätzlich zu diesen Auswertungsmöglichkeiten sind auch weitere Funktionen vorhanden, die vor allem dabei unterstützen sollen, die Berichte individuell anzufertigen.

4.4 Pgstatspack

Anders als bei den zuvor vorgestellten Log-Analyzer-Tools sammelt *pgstatspack* [pgstatspack2009] die benötigten Daten nicht aus den Log-Files, sondern aus den internen Statistiken der jeweiligen PostgreSQL Datenbank. Dazu werden (manuell oder automatisch mittels 'cron jobs') in einem geeigneten Intervall Snapshots der jeweiligen Datenbank(en) erstellt. Diese Snapshots werden wiederum in einer Datenbank für die spätere Analyse

gespeichert. Zur Auswertung der Daten steht ein Report-Modul zur Verfügung, welches vordefinierte Berichte erstellt.

Bei Statistiken, die mit pgstatspack erstellt werden, liegt der Fokus, im Gegensatz zu den Log-Analyzer-Tools, nicht auf der Auswertung einzelner `SELECT`-Statements. Die Statistiken zeigen bei diesem Tool hauptsächlich die Zugriffe auf einzelne Datenbankobjekte. Ausgewertet wird unter anderem die Anzahl der Zugriffe auf die Tabellen und Indizes der Datenbank und die Anzahl der gelesenen Tupel beziehungsweise Indexeinträge pro Tabelle und Index. Außerdem gibt das Tool Auskunft über die Gesamtzahl der Transaktionen pro Sekunde und über die gesetzten Parameter einer Datenbank. Darüber hinaus werden jeweils Statistiken über den prozentualen Anteil von 'cache reads' und physischen Zugriffen geführt.

4.5 pg_statsinfo

Das Softwaretool *pg_statsinfo* [pgstatsinfo2009a, pgstatsinfo2009b] ist vergleichbar mit dem im vorherigen Kapitel vorgestellten Tool pgstatspack. Auch dieses Tool lädt Snapshots aus einer oder mehreren Datenbank(en) in eine 'Snapshot-Management Datenbank'. Aus diesen Snapshots können wiederum Berichte generiert werden. Für einige Statistiken wird zusätzlich das PostgreSQL-Log-File analysiert.

Das Tool pg_statsinfo bietet, neben einem Überblick über die Datenbankaktivität, Informationen über den Ressourcenverbrauch der einzelnen Tablespaces über die gesetzten Parameter und über die Tabellenzugriffe. Darüber hinaus bietet dieses Tool weitere Informationen über die Fragmentierung von Tabellen und über die Ordnung der Datensätze bei der Nutzung eines *Clustered Index* (siehe Kapitel 2.1). Neben diesen Statistiken zu den Tabellen einer Datenbank kann pg_statsinfo auch eine Liste der SQL-Statements und Funktionen mit der längsten Ausführungszeit erstellen[11].

[11] Diese Funktion (die Auflistung der SQL-Statements und Funktionen) ist erst ab Version 1.1.0 für PostgreSQL 8.4 Datenbanken verfügbar.

5 Implementierung eines Softwaretools für das Partitionieren von PostgreSQL Datenbanken

In den folgenden Unterkapiteln wird das implementierte Softwaretool beschrieben. Zunächst wird kurz auf die verwendeten Technologien eingegangen und die zu Beginn des Projektes definierten Anforderungen dokumentiert. Anschließend folgt eine Beschreibung der Funktionen und es wird beschrieben, wie die Implementierung dieser Funktionen erfolgte. Dabei wird weniger auf den Source Code des Softwaretools eingegangen, sondern vielmehr auf den allgemeinen Programmablauf und die einzelnen Problemlösungsstrategien. Die Dokumentation des Source Codes kann dem Quellcode selbst beziehungsweise den *javadocs* entnommen werden.

Das Softwaretool wurde als Java Kommandozeilenprogramm unter dem Projektnamen *pgEasyPart* entwickelt.

5.1 Verwendete Technologien

5.1.1 Java

Java[12] ist eine objektorientierte Programmiersprache. Einer der Vorteile von Java ist, dass Java Programme architektur-neutral sind. Dies wird dadurch

[12]http://java.sun.com/

erreicht, dass der Quellcode beim Compilieren zunächst in Bytecode übersetzt wird. Zum Ausführen des Programms wird die *Java Virtual Machine* benötigt, welche den compilierten Quellcode (Bytecode) interpretiert [Java2010a]. Die Kommunikation mit Datenbanken wird durch die JDBC (Java Database Connectivity) API ermöglicht [Java2010b].

Das gesamte Softwaretool *pgEasyPart* ist in der Programmiersprache Java geschrieben. Für die Entwicklung wurde der *Java Development Kit 'jdk 6 update 16'* von Sun verwendet. Zum Ausführen von *pgEasyPart* wird die JDBC API für PostgreSQL Datenbanken benötigt.

5.1.2 PostgreSQL

PostgreSQL[13] (manchmal auch nur als 'Postgres' bezeichnet) ist ein, ursprünglich vom *Computer Science Department* an der *University of California at Berkeley* entwickeltes, objekt-rationales, ACID konformes, open-source Datenbankmanagementsystem. PostgreSQL basiert auf der Datenbankabfragesprache SQL und unterstützt den ANSI-SQL:2008 Standard zu großen Teilen. PostgreSQL ist unter der sehr liberalen *BSD Lizenz* veröffentlicht, welche auch jegliche Art der kommerziellen Nutzung ermöglicht [PostgreSQL2009a, PostgreSQL2009c].

An dieser Stelle soll nochmals ausdrücklich erwähnt werden, dass das Softwaretool *pgEasyPart* ausschließlich für die Verwendung mit PostgreSQL Datenbanken entwickelt wurde. Bei der Entwicklung und den Funktionstests wurden die Versionen 8.3 und 8.4 eingesetzt.

5.1.3 PL/pgSQL

PostgreSQL bietet die Möglichkeit, benutzerdefinierte Funktionen mit sogenannten '*procedural languages*' (prozedurale Programmiersprachen) zu erstellen. Diese Funktionen können wie eingebaute Funktionen von einem

[13] http://www.postgresql.org/

Client aufgerufen werden und als Ergebnis, wie von SQL-Statements gewohnt, einen oder mehrere Datensätze zurückliefern. Alternativ können diese Funktionen auch als *void* deklariert sein und somit kein Ergebnis ausgeben. *Procedual languages* können dazu verwendet werden, Berechnungen durchzuführen oder mehrere SQL-Statements zu gruppieren. Eine Funktion wird vom Client wie ein SQL-Statement aufgerufen. So kann beispielsweise die Funktion `myFunction()` mit dem Statement `SELECT myFunction();` aufgerufen werden. Auch die Übergabe von Parametern beim Aufruf einer Funktion ist möglich [PostgreSQL2009a].

Für PostgreSQL stehen mehrere *procedual languages* zur Verfügung, unter anderen Pl/Perl, PL/Python und auch PL/pgSQL. Um benutzt werden zu können, müssen diese in der jeweiligen Datenbank installiert werden. Da PL/pgSQL im Kern von PostgreSQL enthalten ist, genügt hierzu das einmalige Ausführen des Statements `CREATE LANGUAGE plpgsql;` [PostgreSQL2009a].

PgEasyPart verwendet PL/pgSQL beim Partitionieren und zum Verschieben der Daten in die Partitionen. Die Funktionen werden alle automatisch vom Softwaretool erzeugt. Der Benutzer braucht daher grundsätzlich keine Kenntnis über den Umgang mit PL/pgSQL Funktionen. Voraussetzung für das einwandfreie Funktionieren ist aber, dass PL/pgSQL in der jeweiligen Datenbank installiert ist.

5.2 Anforderungen

Das Partitionieren von Datenbanken kann (wie bereits in Kapitel 2.3 beschrieben) die Handhabung großer Datenmengen erheblich verbessern. Das (horizontale) Partitionieren von Tabellen ist in PostgreSQL zwar prinzipiell möglich, die Umsetzung ist in der Praxis allerdings relativ aufwendig, da hierfür beim erstmaligen Partitionieren und beim Hinzufügen neuer Partitionen jedes Mal Trigger und Funktionen angelegt werden müssen. Auch das Teilen, Zusammenführen oder Löschen von Partitionen muss manuell

erfolgen und ist mit einem großen Aufwand verbunden. PostgreSQL stellt für all diese Aufgaben bisher noch keine Funktionen zur Verfügung. Das Ziel bei der Entwicklung von *pgEasyPart* war es, das Partitionieren und die Wartung von partitionierten Tabellen zu vereinfachen und weitgehendst zu automatisieren. Des weiteren sollte es möglich sein, mit Hilfe des Softwaretools eine bestehende Tabelle, welche bereits Daten enthält, zu partitionieren, ohne dabei den Zugriff auf die Datenbank beziehungsweise auf die zu partitionierende Tabelle zu beschränken.

Aufbauend auf dieser Zielsetzung wurden zu Beginn des Projektes folgende Anforderungen definiert:

Funktionale Anforderungen:

1. Erstellen partitionierter Tabellen für Range- und List-Partitionierung
    - Die erforderlichen Funktionen und Trigger werden automatisch erstellt.
    - Tablespaces können bereits beim Erstellen der Partitionen festgelegt werden.
    - Es wird automatisch geprüft, ob sich die Partition-Constraints der Partitionen überschneiden.
2. Hinzufügen von weiteren Partitionen zu bereits partitionierten Tabellen
    - Das Hinzufügen neuer Partitionen erfolgt analog zum Erstellen partitionierter Tabellen.
    - Wiederum wird automatisch geprüft, ob sich die Partition-Constraints der Partitionen überschneiden.
3. Automatisches Anlegen neuer Partitionen
    - Das automatische Hinzufügen von Partitionen wird nicht direkt im Softwaretool implementiert. Es soll aber ermöglicht werden, neue Partitionen mittels eines Cron Jobs hinzuzufügen. Somit hat der Datenbankadministrator die Möglichkeit, schon im Vorhinein gezielt festzulegen, wann Partitionen erstellt werden. Um dies zu

ermöglichen, kann eine Partition mittels Kommandozeilenbefehl, ohne Interaktion mit dem Benutzer, hinzugefügt werden.

4. Partitionieren von nicht-partitionierten Tabellen
   - Eine nicht-partitionierte Tabelle kann in eine partitionierte Tabelle umgewandelt werden, wobei die Tabelle während der Partitionierung verfügbar bleibt.
5. Partitionen teilen
   - Eine Partition kann in zwei Partitionen geteilt werden.
6. Partitionen zusammenführen
   - Zwei Partitionen können zu einer zusammengeführt werden.
7. Partitionen aus- und einhängen
   - Eine Partition kann aus einer partitionierten Tabelle herausgelöst werden und existiert als 'normale' Tabelle weiter.
   - Eine zuvor ausgehängte Tabelle kann einer partitionierten Tabelle wieder als Partition hinzugefügt werden.
8. Die (referentielle) Integrität der Daten bleibt erhalten, soweit dies möglich ist. In Fällen, wo dies nicht möglich ist, kann der Benutzer die (referentielle) Integrität mit dem Softwaretool automatisch überprüfen und wird auf Verstöße hingewiesen.
   - Fremdschlüssel, Primärschlüssel und UNIQUE-Constraints der Eltern-Tabelle werden in PostgreSQL nicht automatisch an die Kind-Tabellen vererbt. Das Softwaretool übernimmt diese Aufgabe und kann die Schlüssel für jede Partition automatisch anlegen.
   - Primärschlüssel sind immer nur für jede Kind-Tabelle beziehungsweise für die Eltern-Tabelle eindeutig. Das heißt, dass der Wert zwar innerhalb einer Partition nur einmalig vorkommt, aber in einer anderen Partition trotzdem aufscheinen kann. Bei Abfragen über mehrere Partitionen kann ein Wert des Primärschlüssels daher öfter vorkommen (wenn dieser nicht Teil des Schlüsselattributes ist). Das Softwaretool stellt daher eine Funktion bereit, die

es dem Datenbankadministrator ermöglicht, die Integrität der Primärschlüssel einer partitionierten Tabelle regelmäßig zu überprüfen.

- Fremdschlüssel, welche auf die partitionierte Tabelle zeigen, stellen ebenfalls ein großes Problem bei partitionierten Tabellen in PostgreSQL dar, weil diese immer nur auf die Eltern-Tabelle verweisen und nicht auf die Kind-Tabellen. Bei der Tabellenpartitionierung enthält die Eltern-Tabelle aber keine Daten, daher kann die Fremdschlüssel-Bedingung niemals erfüllt werden. Das Softwaretool kann alle Fremdschlüssel, die auf die Eltern-Tabelle zeigen, automatisch entfernen und in einer Fremdschlüssel-Tabelle speichern. Danach kann der Datenbankadministrator mittels einer weiteren Funktion regelmäßig überprüfen, ob die referentielle Integrität der gespeicherten Fremdschlüssel verletzt wurde.

Nicht-funktionale Anforderungen:

1. Die einfache Bedienbarkeit des Tools wird durch eine SQL-nahe Syntax gewährleistet.

Mögliche zusätzliche Feature, welche vorerst nicht unterstützt werden:

1. Unterstützung für Sub-Partitionierung.
2. Unterstützung für Partitionierungsschlüssel, welche sich aus zwei oder mehr Schlüsselwerten zusammensetzen.

Systemanforderungen:

1. Betriebssystem: Linux
2. Datenbank: PostgreSQL 8.3 oder 8.4
3. Java Umgebung: JRE 6

Abbildung 5.1 stellt die Anforderungen nochmals kompakt anhand eines Use-Case-Diagramms dar.

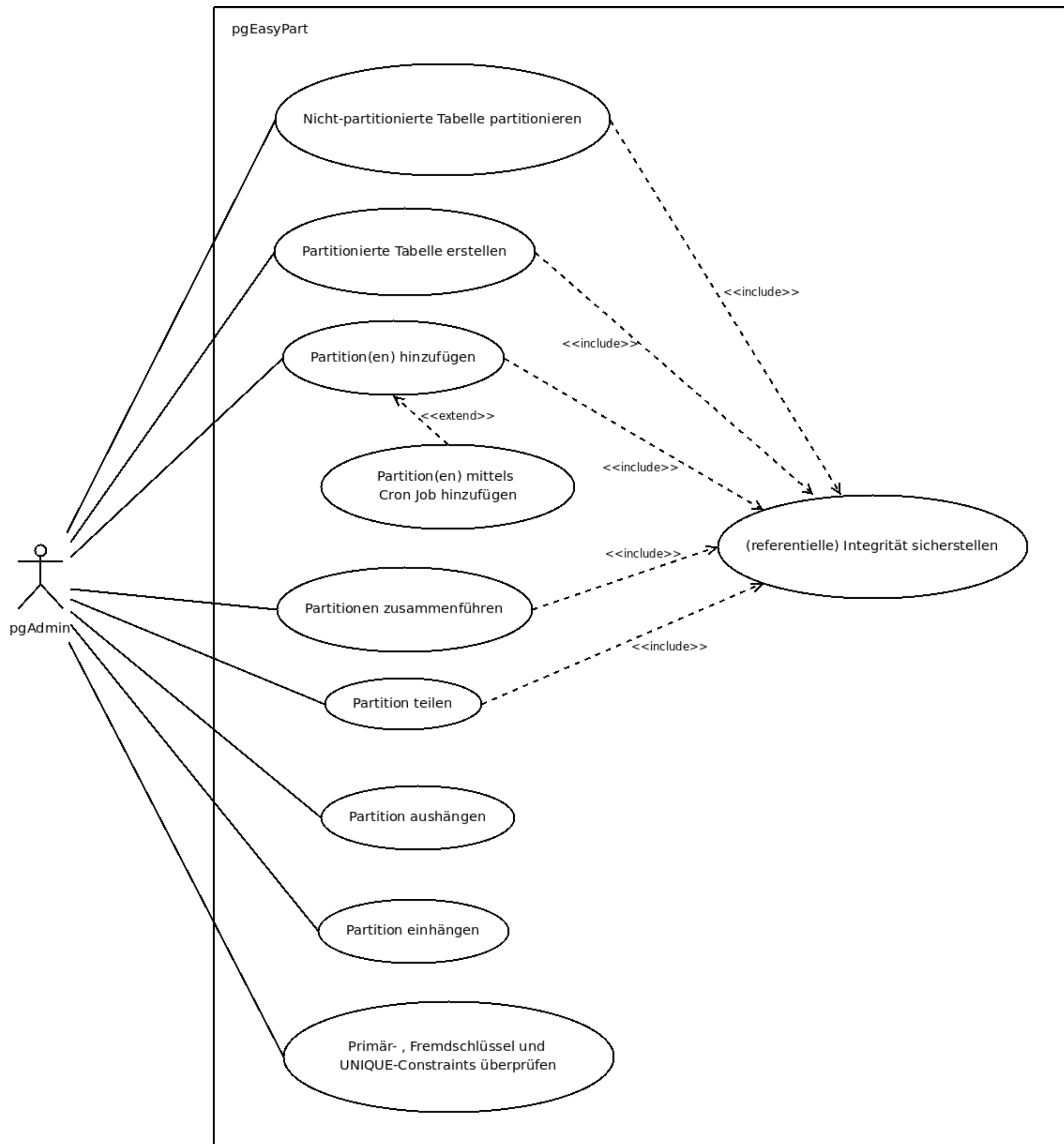

Abbildung 5.1: Use-Case-Diagramm für das Softwaretool *pgEasyPart*

5.3 Implementierung des Softwaretools

In diesem Kapitel wird das Softwaretool *pgEasyPart* detailliert beschrieben. Zunächst erfolgt in Kapitel 5.3.1 ein Überblick über das gesamte Softwaretool. In diesem Unterkapitel wird die Architektur und die Nutzung des Programms dargestellt. In den anschließenden Kapiteln wird auf die Umsetzung der einzelnen Anwendungsfälle genauer eingegangen[14].

5.3.1 Überblick

Das Softwaretool *pgEasyPart* ist ein reines, in der Programmiersprache Java entwickeltes, Konsolenprogramm. Für die Nutzung des Tools ist keine Installation erforderlich. Einzige Anforderungen sind die Verfügbarkeit einer *Java Runtime Environment (JRE)* und des PostgreSQL-JDBC-Treibers für die Datenbankanbindung. Das Programm kann von jedem PC (der die Systemvoraussetzungen erfüllt) aus genutzt werden, von welchem eine Verbindung zu der Datenbank aufgebaut werden kann. Zur Nutzung muss der Benutzer in der Konsole in den *pgEasyPart*-Ordner navigieren und in diesem Ordner das Programm mit dem Befehl `java -jar pgEasyPart.jar` starten.

Das Tool besteht aus drei ausführbaren JAR-Archiven. Das Archiv *pgEasyPart.jar* enthält das eigentliche Hauptprogramm. Die beiden anderen Archive, *pgEPCheck.jar* und *pgEPAddPartition.jar*, sind Hilfsprogramme, welche das Hinzufügen von Partitionen und das Überprüfen der Primär- und Fremdschlüssel mittels eines einzigen Kommandozeilenbefehls, ohne Interaktion mit dem Benutzer, ermöglichen. Auf den Zweck und die Nutzung dieser Hilfsprogramme wird in den nachfolgenden Unterkapiteln noch näher eingegangen.

Das Klassendiagramm in Abbildung 5.2 zeigt alle Pakete und die beinhalteten Klassen des Softwaretools. Die zwei Pakete *pgEasyPartCheckTool*

[14] Weitere Details zur Nutzung, welche den Rahmen dieser Arbeit sprengen würden und einer Befehlsreferenz, können den entsprechenden Dateien im Programmarchiv entnommen werden.

und *pgEasyPartAddPartitionTool* beinhalten die eben erwähnten Hilfsprogramme. Beide benutzen Klassen aus dem Paket *pgEasyPart*. Das Paket *pgEasyPart* beinhaltet den eigentlichen Kern des Softwaretools, das Paket *pgEasyPart.lib*. In diesem Paket findet der Großteil des Programmablaufs statt.

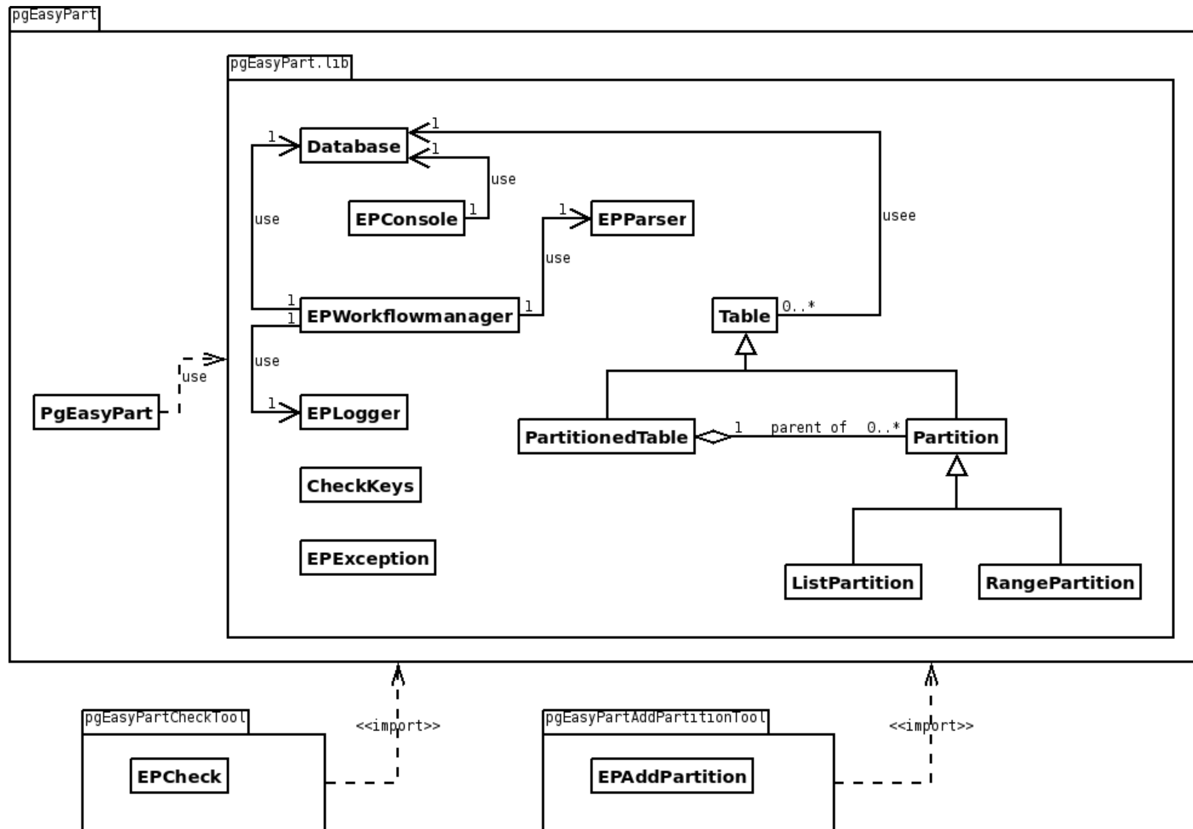

Abbildung 5.2: Klassendiagramm des Softwaretools *pgEasyPart*

Beim Programmstart wird zunächst die `main()` Methode der Klasse *pgEasyPart* aufgerufen. Im nächsten Schritt wird die Konfigurationsdatei eingelesen. Über die *pgEasyPart*-Konsole (Klasse: *EPConsole*) werden die Datenbankinformationen und die Partitionierungs-Statements eingelesen. Die Datenbankinformationen können auch in einer Text-Datei gespeichert werden. Die Partitionierungs-Statements sind SQL-ähnliche Statements. Eine bestehende nicht-partitionierte Tabelle kann beispielsweise mit dem Statement

```
ALTER TABLE s.foo PARTITION BY LIST(column_name) (
    foo_partition_01 (VALUES (AT DE) TABLESPACE mytablespace),
    foo_partition_02 (VALUES (FR US))
);
```

partitioniert werden. Diese Statements werden in der Arbeit in weiterer Folge zum leichteren Verständnis als 'SQL-Statements' bezeichnet, auch wenn dies nicht völlig korrekt ist, da *pgEasyPart*-Statements nicht SQL-Standard konform sind. Ein ';' in einem Statement wird wie in SQL immer als Ende des Statements interpretiert.

Durch das Parsen des SQL-Statements wird anschließend der entsprechende Programmablauf ermittelt. Alle Abläufe sind in der Klasse *EPWorkflow* implementiert. Diese Klasse steuert den gesamten verbleibenden Programmablauf.

Die Klassen *Table*, *PartitionedTable* und *Partition* repräsentieren die gleich lautenden Datenbankobjekte, wobei die Klasse *PartitionedTable* für die Eltern-Tabelle steht. Partitionen können entweder vom Typ (Klasse) *ListPartition* oder *RangePartition* sein, je nachdem, ob eine List- oder Range-Partitionierung gewählt wird[15]. Alle Datenbankmanipulationen (außer das Entfernen der Fremdschlüssel, dies erfolgt über die Klasse *CheckKeys*) finden über diese Klassen statt. Zum Überprüfen der Fremdschlüssel, Primärschlüssel und UNIQUE-Constraints stellt die Klasse *CheckKeys* Methoden bereit. Die Datenbankverbindungsinformationen werden während der Laufzeit in der Klasse *Database* gespeichert.

Die Klasse *EPLogger* dient zum Loggen spezieller Ereignisse. *PgEasyPart* loggt alle Datenbankmanipulationen und auftretende Ausnahmen. Alle Log-Einträge einer Session werden am Programmende ausgegeben und zusätzlich auch in eine Log-Datei geschrieben. Ausnahmen werden durch die Klasse *EPException* abgefangen. Wurden beim Auftreten des Fehlers bereits Tabellen oder Funktionen erstellt, so kann die Klasse *EPException* diese (nach Nachfrage beim Benutzer) wieder entfernen. Diese Option wird nur dann

[15]PgEasyPart unterstützt nur 'reine' List- oder Range-Partitionierung. Andere Partitionierungsformen, auch Composite beziehungsweise Subpartitionierung, werden nicht unterstützt.

angeboten, wenn keine Daten in die entsprechenden Datenbankobjekte geschrieben wurden.

Die Klasse *EPParser* wurde bereits beim Bestimmen der Workflows erwähnt. Mit den Methoden dieser Klasse werden alle benötigten Informationen aus den SQL-Statements extrahiert.

In diesem Kapitel sollte dem Leser nur ein Überblick über die gesamte Architektur und den Programmablauf geboten werden. In den folgenden Unterkapiteln wird die Implementierung der einzelnen Anwendungsfälle genauer beschrieben. Dabei handelt es sich prinzipiell um die verschiedenen Workflows, welche in der Klasse *EPWorkflowmanager* implementiert sind. Nochmals erwähnt werden sollte, dass sich die Klasse *EPWorkflowmanager* nur um die Steuerung kümmert und die Programmlogik daher teilweise auch in verschiedenen anderen Klassen zu finden ist.

5.3.2 Erstellen einer partitionierten Tabelle

Das Erstellen einer partitionierten Tabelle erfolgt mit einem Statement der Form

```
CREATE TABLE schema.foo (
    a integer,
    b date,
    PRIMARY KEY (a)
) PARTITION BY RANGE(b) (
    foo_partition_01 (STARTING MINIMUM ENDING 2008-12-31),
    foo_partition_02 (STARTING 2009-01-01 ENDING 2009-12-31),
    foo_partition_03 (STARTING 2010-01-01 ENDING 2010-12-31)
);
```

für Range-Partitionen beziehungsweise einem Statement der Form

```
CREATE TABLE schema.foo (
    a integer,
    b char(2),
    PRIMARY KEY (a)
) PARTITION BY LIST(b) (
```

```
        foo_partition_01 (VALUES (AT DE)),
        foo_partition_02 (VALUES (FR US))
);
```

für List-Partitionen.

Beim Erstellen einer partitionierten Tabelle wird zunächst eine 'normale' Tabelle in der Datenbank angelegt. Dazu wird das '`CREATE TABLE`'-Statement aus der gesamten Anweisung extrahiert. *PgEasyPart* interpretiert hierzu alle Anweisungen, welche sich vor der '`PARTITION BY`'-Klausel befinden, als '`CREATE TABLE`'-Statement und sendet dieses unverändert an die Datenbank. Im '`CREATE TABLE`'-Abschnitt des SQL-Statements kann der Benutzer daher, wie bereits gewohnt, eine Tabelle in der PostgreSQL-Datenbank anlegen.

Alle Daten, welche für den weiteren Programmablauf benötigt werden, erhält das Programm durch Parsen des SQL-Statements. Abbildung 5.3 zeigt alle Ablaufschritte anhand eines Aktivitätsdiagramms.

In den weiteren Schritten werden die Partitionen (durch Tabellenvererbung) und die erforderlichen Trigger und Funktionen in der Datenbank angelegt. Optional kann das Softwaretool auch so konfiguriert werden, dass automatisch Update-Trigger und die erforderlichen Funktionen angelegt werden. Diese ermöglichen einen Datensatz, dessen Partitionierungsschlüssel sich ändert, sodass dieser das Partitionierungskriterium für die jeweilige Partition nicht mehr erfüllt, automatisch in eine andere Partition zu verschieben.

Vor dem Anlegen der Partitionen in der Datenbank überprüft das Softwaretool, ob sich die Bedingungen (Partition-Constraints) der Partitionen überschneiden. Ist dies der Fall, so erfolgt eine Rückmeldung an den Benutzer. Dieser kann dann entscheiden, den Vorgang abzubrechen. Die Überprüfung ist bei Range-Partitionierung auf die Datentypen *integer*, *float*, *date* und *timestamp* beschränkt. Bei List-Partitionierung gibt es diese Beschränkung nicht, da hier jeweils nur die *String*-Repräsentation der Werte verglichen wird.

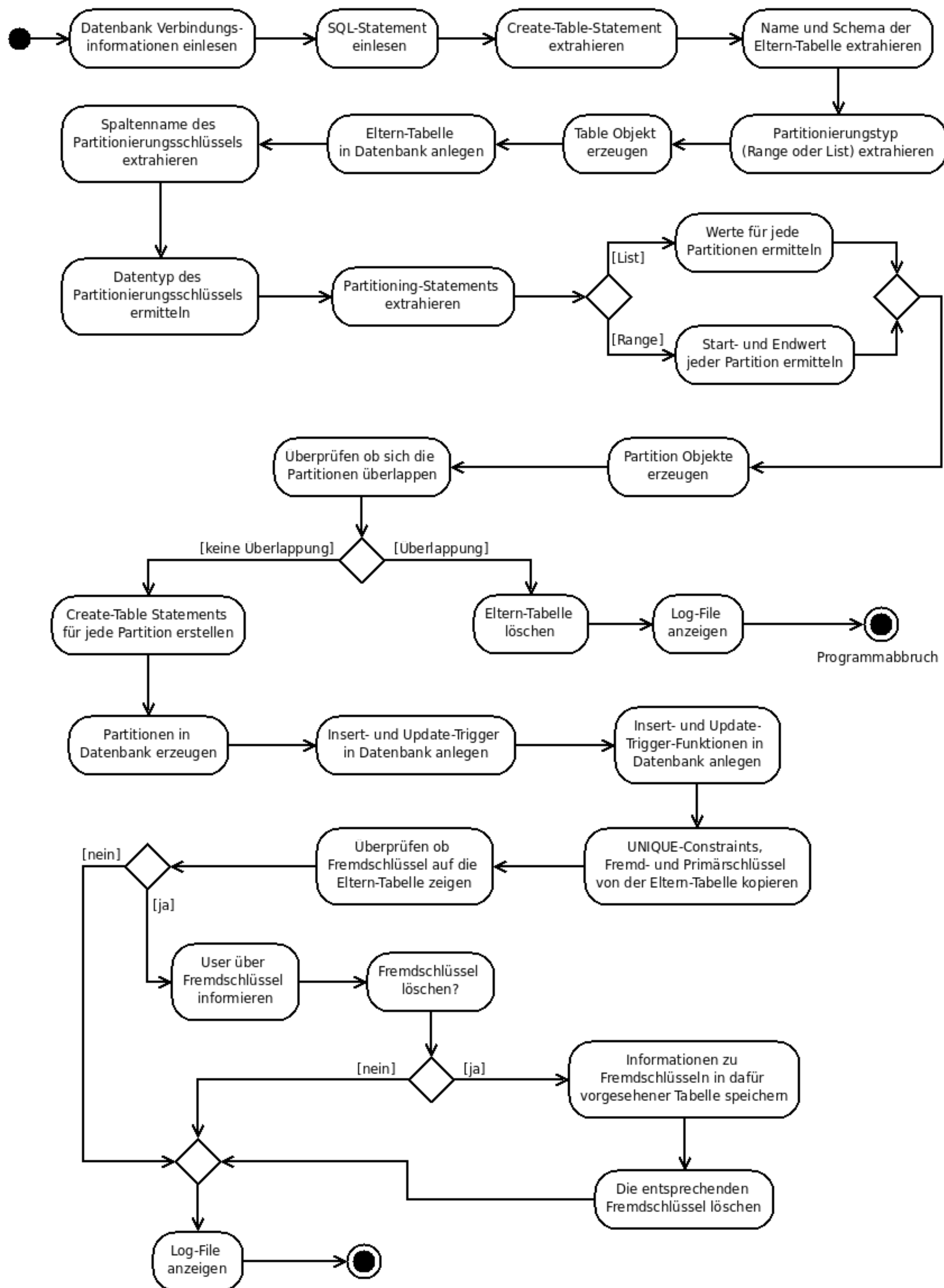

Abbildung 5.3: Aktivitätsdiagramm für das Erstellen einer partitionierten Tabelle

Nach dem Anlegen der Partitionen in der Datenbank kann der Benutzer *pgEasyPart* anweisen, alle Primärschlüssel, Fremdschlüssel und UNIQUE-Constraints auf die Kind-Tabellen zu übertragen. Um dies automatisch durchzuführen, bedient sich das Softwaretool der internen Systemtabellen von PostgreSQL. Alle Informationen über Constraints (dazu gehören auch Primärschlüssel, Fremdschlüssel und UNIQUE-Constraints) werden beispielsweise in der Systemtabelle *pg_ constraint* gespeichert. Die Systemtabellen werden von *pgEasyPart* immer dann benutzt, wenn Informationen über bereits existierende Datenbankobjekte benötigt werden.

Fremdschlüssel, welche auf eine partitionierte Tabelle zeigen, stellen in PostgreSQL Datenbanken, wie bereits in dieser Arbeit beschrieben, einen Spezialfall dar. Das Softwaretool bietet zur Lösung dieses Problems eine Funktion an, die zwar nicht die referentielle Integrität sicherstellt, aber zumindest Verletzungen der referentiellen Integrität erkennen kann. Um dies zu ermöglichen, können mit *pgEasyPart* automatisch alle Fremdschlüssel, welche auf die Eltern-Tabelle zeigen, erkannt, in eine spezielle Fremdschlüssel-Tabelle geschrieben und danach gelöscht werden. In der Fremdschlüssel-Tabelle werden jeweils der Tabellenname, das Schema und die Fremdschlüssel-Attribute beider referenzierender Tabellen erfasst. Später kann dann regelmäßig überprüft werden, ob diese Bedingungen durch Tabelleneinträge verletzt wurden (siehe auch Kapitel 5.3.8). Auch wenn die Überprüfung der Fremdschlüssel beim Anlegen einer neuen partitionierten Tabelle obligatorisch erfolgt, wird diese zunächst keine Auswirkungen haben, da im Normalfall keine Fremdschlüssel auf eine gerade eben erstellte Tabelle zeigen können. Daher kann dieser Ablaufschritt mit dem Statement

```
CHECK FOREIGN KEYS REFERENCING partitioned_table_tablename;
```

zu einem späteren Zeitpunkt wiederholt werden.

5.3.3 Eine nicht-partitionierte Tabelle partitionieren

Der Ablauf beim Partitionieren einer bestehenden, nicht-partitionierten Tabelle ist zu Beginn dem im vorhergehenden Unterkapitel beschriebenen Ablauf zum Erstellen einer neuen partitionierten Tabelle sehr ähnlich, mit dem Unterschied, dass in diesem Fall die Eltern-Tabelle nicht angelegt werden muss, da diese bereits existiert und auch schon Daten beinhalten kann. Die 'CREATE TABLE'-Klausel entfällt daher im SQL-Statement. Ein SQL-Statement zum Partitionieren einer nicht-partitionierten Tabelle hat (bei Range-Partitionierung) folgendes Aussehen:

```
ALTER TABLE schema.foo PARTITION BY RANGE(column_name) (
    foo_partition_1 (STARTING MINIMUM ENDING 400000),
    foo_partition_2 (STARTING 400001 ENDING 800000),
    foo_partition_3 (STARTING 800001 ENDING MAXIMUM)
);
```

Wiederum ist auch List-Partitionierung möglich.

Nachdem die Partitionen angelegt wurden, dies erfolgt genauso wie bereits in Kapitel 5.3.2 beschrieben (siehe auch Abbildung 5.4, welche das Aktivitätsdiagramm für das Partitionieren nicht-partitionierter Tabellen darstellt), müssen die Daten von der Eltern-Tabelle in die dafür vorgesehenen Partitionen verschoben werden. Zum Verschieben der Daten werden zwei PL/pgSQL-Funktionen angelegt. Mit der ersten Funktion wird zunächst eine festgelegte Anzahl von Zeilen selektiert. Danach wird für jede dieser Zeilen die zweite Funktion aufgerufen. Diese Funktion kopiert die Daten der Zeilen in die dafür vorgesehene Partition und löscht sie anschließend aus der Eltern-Tabelle. Während der Laufzeit der zweiten Funktion wird die Tabelle für alle Datenbankzugriffe gesperrt. Listing 5.1 zeigt ein Beispiel dieser beiden Funktionen.

Listing 5.1: PL/pgSQL-Funktionen zum Verschieben der Daten in die Partitionen

```
-- Erste Funktion:
-- Selektiert eine festgelegte Anzahl von Zeilen
-- und ruft fuer jede dieser Zeilen die zweite Funktion auf

CREATE OR REPLACE FUNCTION pgep_select_rows_schema_foo()
RETURNS SETOF integer AS $$
DECLARE
  partbl_data schema.foo%ROWTYPE;
BEGIN
  FOR partbl_data IN SELECT * FROM ONLY schema.foo LIMIT 10 LOOP
    PERFORM pgep_move_row(partbl_data);
  END LOOP;

  SELECT * INTO partbl_data FROM ONLY schema.foo LIMIT 1;
  IF NOT FOUND THEN
    RETURN NEXT 1;
  ELSE
    RETURN NEXT -1;
  END IF;

  RETURN;
END;
$$ LANGUAGE plpgsql;



-- Zweite Funktion:
-- Verschiebt jeweils eine Zeile

CREATE OR REPLACE FUNCTION pgep_move_row_schema_foo(schema.foo)
RETURNS void AS $$
DECLARE
  r schema.foo%ROWTYPE;
BEGIN
  LOCK TABLE schema.foo, schema.foo_partition_1, schema.foo_partition_2,
    schema.foo_partition_3;

  SELECT * INTO r FROM ONLY schema.foo WHERE foo_id = $1.foo_id;
  IF FOUND THEN
    -- INSERT INTO partition;
    IF ( r.foo_id >= '800001' )
      THEN INSERT INTO schema.foo_partition_3 VALUES (r.*);
    ELSEIF ( r.foo_id >= '400001' AND r.foo_id <= '800000' )
```

```
      THEN INSERT INTO schema.foo_partition_2 VALUES (r.*);
    ELSEIF ( r.foo_id <= '400000' )
      THEN INSERT INTO schema.foo_partition_1 VALUES (r.*);
    ELSE
      RAISE EXCEPTION 'Value out of range.
        Cannot insert parent-table row into any partition !';
    END IF;

    -- DELETE FROM parent_table
    DELETE FROM ONLY schema.foo WHERE foo_id = r.foo_id;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Die Anzahl der Zeilen, welche bei jedem Aufruf der ersten Funktion selektiert werden, kann der Benutzer in der Konfigurationsdatei festlegen. Die Anzahl sollte nicht zu hoch sein, da das Selektieren der Daten sonst sehr lange dauern kann und die Performance der Datenbank darunter leidet.

Bevor mit dem Verschieben der Daten begonnen wird, prüft das Softwaretool, ob jede Zeile der Eltern-Tabelle in eine der Partitionen verschoben werden kann. Außerdem wird zum Verschieben der Daten ein Primärschlüssel benötigt. Besitzt die Tabelle keinen Primärschlüssel, wird mit einem Fehler abgebrochen. Um die Daten zu verschieben, ruft *pgEasyPart* die erste Funktion so oft auf, bis diese als Ergebnis den Wert 1 zurückliefert. Der Wert 1 zeigt an, dass sich keine Daten mehr in der Eltern-Tabelle befinden. Tritt während des Verschiebens der Daten ein Fehler auf (zum Beispiel wenn die Datenbank nicht mehr erreichbar ist), so kann der Vorgang mit dem Befehl

```
CONTINUE PARTITIONING ON TABLE schema.foo;
```

zu einem späteren Zeitpunkt fortgesetzt werden.

Während des gesamten Partitionierungsvorganges kann auf die Tabelle normal zugegriffen werden. Sowohl lesende als auch schreibende Datenbankoperationen können auf die Tabelle angewendet werden. Bei `DELETE`- und

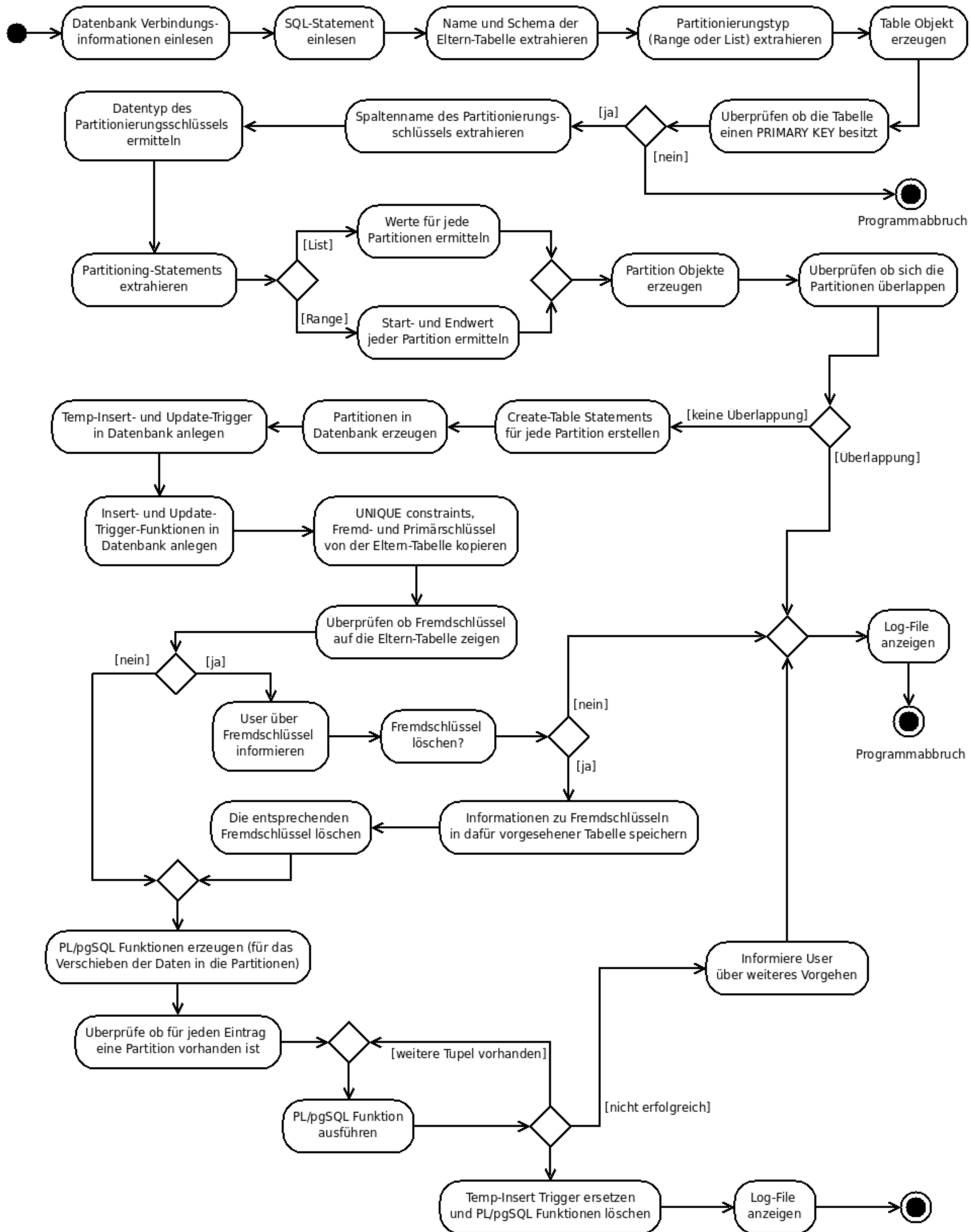

Abbildung 5.4: Aktivitätsdiagramm für das Partitionieren einer nicht-partitionierten Tabelle

UPDATE-Statements macht PostgreSQL keinen Unterschied, ob die betroffenen Daten aus der Eltern-Tabelle oder den Kind-Tabellen (Partitionen) stammen (solange nicht explizit das Schlüsselwort FROM ONLY angegeben wurde). Neue Datensätze werden in die jeweils passende Partition geschrieben, nachdem geprüft wurde, dass kein Datensatz mit dem gleichen Primärschlüssel in der Eltern-Tabelle existiert. Auch SELECT-Statements liefern zu jedem Zeitpunkt alle betroffenen Datensätze korrekt zurück, da bei Datenbankabfragen immer zuerst die Partitionen und danach auch die Eltern-Tabelle durchsucht werden. Auch wenn eine Datenbankabfrage auf eine oder wenige Partitionen beschränkt ist, wird am Ende immer auch die Eltern-Tabelle nach Datensätzen durchsucht. Listing 5.2 zeigt dies anhand eines 'EXPLAIN SELECT'-Statements.

Listing 5.2: *Query Plan* einer partitionierten Tabelle

```
myDB=# EXPLAIN SELECT * FROM foo WHERE a BETWEEN 1 AND 100;
                              QUERY PLAN
------------------------------------------------------------
 Result  (cost=0.00..84.20 rows=22 width=8)
   -> Append (cost=0.00..84.20 rows=22 width=8)
         -> Seq Scan on foo  (cost=0.00..42.10 rows=11 width=8)
               Filter: ((a >= 1) AND (a <= 100))
         -> Seq Scan on foo_partition_1 foo (cost=0.00..42.10 rows=11 width=8)
               Filter: ((a >= 1) AND (a <= 100))
(6 Zeilen)
```

Bei dem Statement in Listing 5.2 durchsucht PostgreSQL zunächst die Partition *foo_partition_1* (alle anderen Partitionen werden aufgrund der WHERE-Klausel ignoriert), und anschließend wird auch noch die Eltern-Tabelle (*foo*) durchsucht. Dies zeigt, dass bei SELECT-Statements immer sowohl die Daten aus der Eltern-Tabelle, die noch nicht in die Partitionen verschoben wurden, als auch die Daten aus den Partitionen berücksichtigt werden.

Die Verfügbarkeit der Datenbank und im Speziellen der zu partitionierenden Tabelle ist daher während des Partitionierens gewährleistet. Es kann allerdings zu spürbaren Performanceverlusten während dieses Vorgangs kommen.

5.3.4 Partitionen zu einer partitionierten Tabelle hinzufügen

In vielen Fällen wird es notwendig sein, regelmäßig neue Partitionen hinzuzufügen. Dies ist beispielsweise oft der Fall, wenn nach einem Datum (Jahr, Quartal, Monat etc.) partitioniert wurde. In einem solchen Fall muss der Benutzer nicht alle Partitionen beim Erstellen der partitionierten Tabelle anlegen, sondern kann Partitionen später zu dem Zeitpunkt, an dem sie benötigt werden, hinzufügen (zum Beispiel am Beginn jeden Jahres).

Das Hinzufügen von Partitionen erfolgt in *pgEasyPart* (bei Range-Partitionierung) mit einem Statement der Form

```
ALTER TABLE schema.foo ADD PARTITION (
    foo_partition_4 (STARTING 2011-01-01 ENDING 2011-12-31),
    foo_partition_5 (STARTING 2012-01-01 ENDING 2012-12-31)
);
```

Um Partitionen hinzuzufügen, muss *pgEasyPart* zunächst die partitionierte Tabelle laden. Dazu werden alle Kind-Tabellen, die Partitionen, ermittelt. Danach wird für jede Partition das Partition-Constraint aus den PostgreSQL Systemtabellen ausgelesen. Das Partition-Constraint wird dabei durch den Namen des Constraints identifiziert, welcher in der Konfigurationsdatei definiert wurde. Eine mögliche Form eines Partition-Constraints wäre zum Beispiel:

```
CHECK (partition_key >= 200001::bigint AND partition_key <= 400000::bigint)
```

Aus diesen Constraints kann anschließend der Partitionierungsschlüssel und die Ober- und Untergrenze beziehungsweise bei List-Partitionierung alle zulässigen Werte für die Partition durch Parsen des Constraints ermittelt werden.

Nach dem Laden der partitionierten Tabelle können die neuen Partitionen hinzugefügt werden. Durch das Laden der vorhandenen Partitionen ist es auch möglich, alle Partitionen wiederum auf sich eventuell überschneidende Partition-Constraints zu prüfen. Außerdem ist das Laden notwendig, da die

Partitionierungs-Funktion der Eltern-Tabelle, welche das Einfügen neuer Daten in die Partitionen regelt, neu geschrieben werden muss.

Eine weitere Möglichkeit zum Hinzufügen neuer Partitionen bietet das Hilfsprogramm `pgEPAddPartition.jar`. Dieses ermöglicht es, eine Partition mittels eines einzigen Kommandozeilenbefehls hinzuzufügen. Durch Setzen des Flag `-noUserInteraction` wird auch jede Interaktion mit dem Benutzer unterdrückt. Dieses Hilfsprogramm ermöglicht somit das Hinzufügen neuer Partitionen mittels Skripten und Cron Jobs. Dies kann das regelmäßige Hinzufügen neuer Partitionen erleichtern.

5.3.5 Partitionen zusammenführen

Eine weitere Funktion, welche das Softwaretool zum Managen der Partitionen bietet, ist die Möglichkeit, zwei Partitionen zusammenzuführen. Es können sowohl Range- als auch List-Partitionen zusammengeführt werden. Bei List-Partitionen wird eine neue Partition erstellt, welche die Schlüsselwerte beider zusammengeführter Partitionen enthält. Bei Range-Partitionen wird eine neue Partition mit einer Untergrenze gleich der Untergrenze der Partition mit den niedrigeren Werten und einer Obergrenze gleich der Obergrenze der Partition mit den höheren Werten erstellt. Liegt zwischen diesen beiden Partitionen ein Wertebereich, der bisher keiner Partition zugeordnet war, so ist dieser daher folglich in der neuen Partition auch enthalten.

Das Zusammenführen zweier Partitionen erfolgt in *pgEasyPart* durch Ausführen des SQL-Statements

```
ALTER TABLE schema.foo
    MERGE LOWER BOUND PARTITION foo_partition_2
    AND UPPER BOUND PARTITION foo_partition_3
    INTO PARTITION merged_partition;
```

Abbildung 5.5 zeigt in einer schematischen Darstellung die Vorgehensweise beim Zusammenführen zweier Partitionen. Die Methode ähnelt stark jener beim Partitionieren einer nicht-partitionierten Tabelle. Im Beispiel aus Abbildung 5.5 sollen die beiden Partitionen B und C vereinigt werden. Dazu

legt das Tool eine neue Partition an (Partition E), welche später die Daten der Partitionen B und C enthalten soll. Die Partitionen B und C werden als Sub-Partitionen der neu erstellten Partition E eingehängt und sind folglich nicht mehr direkte Nachfahren der partitionierten Tabelle.

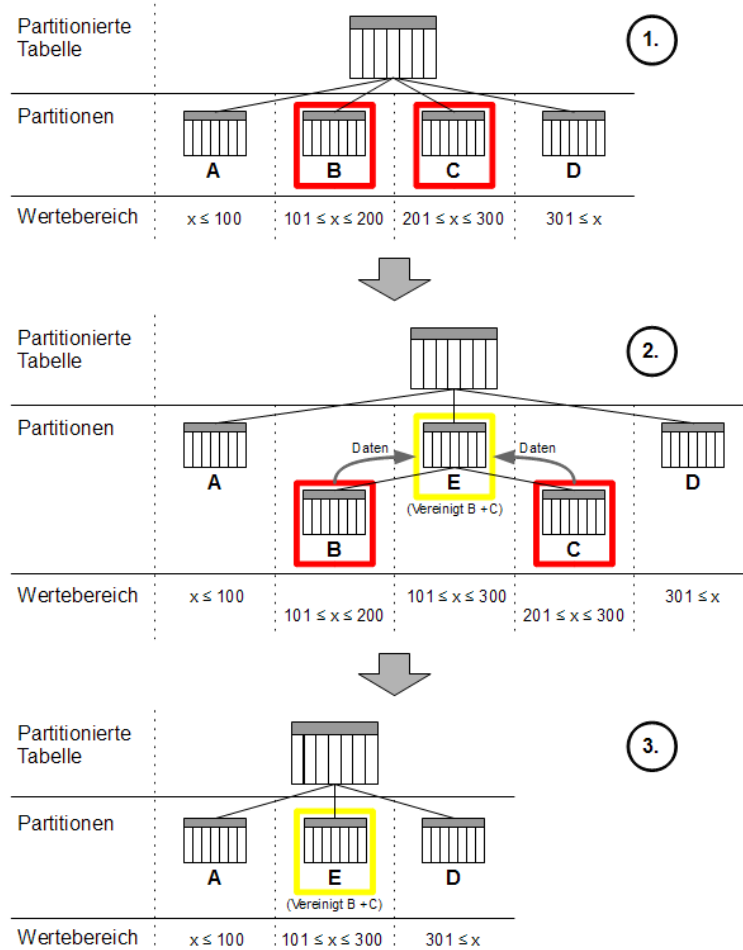

Abbildung 5.5: Zusammenführen zweier Partitionen mit *pgEasyPart*

Anschließend werden die Daten aus den Partitionen B und C in die Partition E verschoben. Das Verschieben der Daten erfolgt auf die selbe Weise wie beim Partitionieren einer nicht-partitionierten Tabelle, mit dem Unterschied, dass in diesem Fall die Daten von den Kind-Tabellen in die Eltern-Tabelle verschoben werden. Neue Daten werden schon während des Verschie-

bens der Daten immer nur in die neue Partition E geschrieben. Nachdem alle Daten verschoben wurden, werden die Partitionen B und C gelöscht. Danach ist der Vorgang abgeschlossen, und es existiert nur mehr die neue Partition E in der Datenbank, die nun die Daten der nicht mehr existierenden Tabellen B und C vereinigt.

5.3.6 Partitionen teilen

Wie beim Zusammenführen von Partitionen stellt auch das Teilen von Partitionen eine wichtige Funktion zum Managen von partitionierten Tabellen dar. Partitionen können mit dem Statement

```
ALTER TABLE schema.foo
    SPLIT PARTITION foo_partition_2 INTO (
        foo_p2a (STARTING 400001 ENDING 600000),
        foo_p2b (STARTING 600001 ENDING 800000)
);
```

geteilt werden.

Wie in Abbildung 5.6 ersichtlich, erfolgt das Teilen einer Partition methodisch abermals ähnlich wie das Partitionieren einer nicht-partitionierten Tabelle und damit auch ähnlich dem Zusammenführen zweier Partitionen.

Im Beispiel aus Abbildung 5.6 soll Partition B in zwei Partitionen (Partitionen D und E) geteilt werden. Dazu werden in diesem Fall zwei neue Partitionen erstellt und als Sub-Partition in der Partition B eingehängt. Die Partitionen D und E sind zunächst noch keine direkten Nachfahren der partitionierten Tabelle.

Anschließend werden die Daten abermals, wie beim Partitionieren einer nicht-partitionierten Tabelle, von der Eltern-Tabelle (Tabelle B) in die Kind-Tabellen (Tabellen D und E) verschoben. Neue Daten werden nicht mehr in die Partition B geschrieben, sondern wandern direkt in eine der Sub-Partitionen. Nachdem alle Daten verschoben wurden, hängt das Softwaretool diese als 'normale' Partitionen der partitionierten Tabelle ein und löscht die Partition B, welche nun keine Daten mehr enthält.

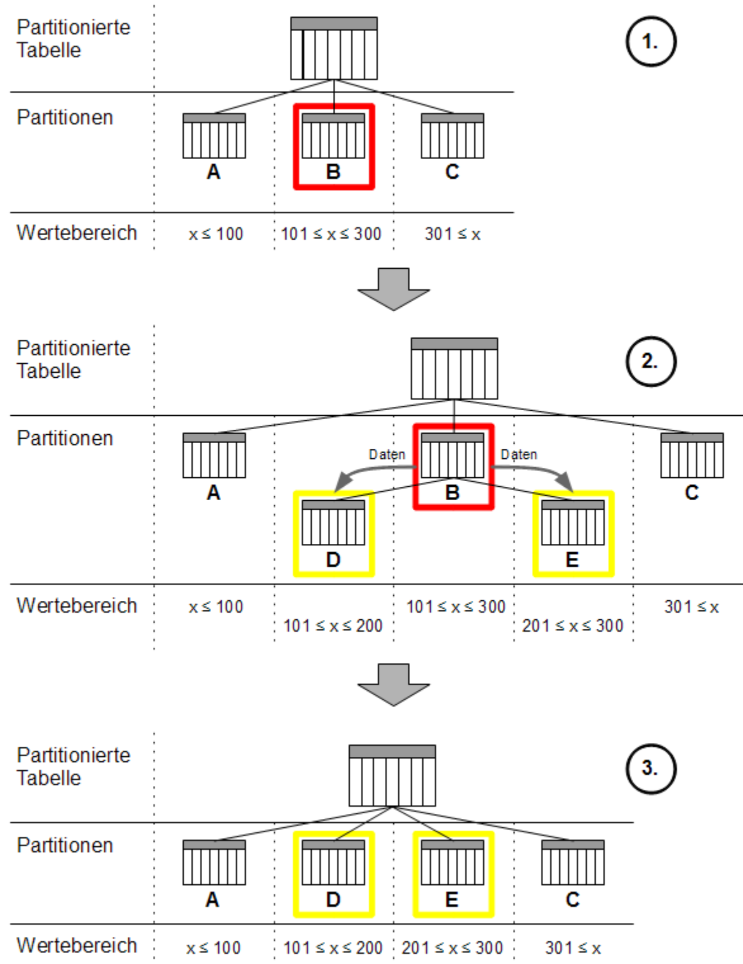

Abbildung 5.6: Teilen einer Partition mit *pgEasyPart*

5.3.7 Partitionen aus- und wieder einhängen

Partitionen können weiters mit dem Statement

ALTER TABLE schema.foo DETACH PARTITION foo_partition_2;

ausgehängt und anschließend mit dem Statement

ALTER TABLE schema.foo ATTACH PARTITION foo_partition_2;

wieder eingehängt werden. Beim Aushängen wird die Vererbungsbeziehung zwischen der partitionierten Tabelle (Eltern-Tabelle) und der Partition ge-

löst. Die Partition existiert folglich als 'normale' Tabelle in der Datenbank weiter. Beim Einhängen erfolgt dieser Vorgang wieder in die umgekehrte Richtung. Durch das Aushängen ist es beispielsweise möglich, alte Partitionen, die nicht mehr benötigt werden, zu archivieren.

Um ausgehängte Partitionen später wieder einhängen zu können, darf das Partition-Constraint der Tabelle nicht verändert oder entfernt werden. Das Softwaretool extrahiert alle Informationen aus diesem Constraint, die für das Einhängen benötigt werden.

5.3.8 Primärschlüssel, Fremdschlüssel und UNIQUE-Constraints von partitionierten Tabellen überprüfen

Wie bereits mehrfach in dieser Arbeit erwähnt, stellen Primärschlüssel, Fremdschlüssel und UNIQUE-Constraints bei Tabellenvererbung und damit auch bei Tabellenpartitionierung in PostgreSQL Datenbanken eine Besonderheit dar. Fremdschlüssel, welche auf die partitionierte Tabelle zeigen, werden, wie in Kapitel 5.3.2 beschrieben, in einer Fremdschlüssel-Tabelle gespeichert und anschließend entfernt. Primärschlüssel und UNIQUE-Constraints können zwar auf die Partitionen übertragen werden, gelten aber nur innerhalb einer Partition und nicht für die gesamte partitionierte Tabelle.

Auch für partitionierte Tabellen, die mit *pgEasyPart* angelegt wurden, gelten all diese Einschränkungen. Die Konsistenz der Datenbank muss daher auf Anwendungsebene sichergestellt werden. *PgEasyPart* stellt allerdings eine Funktion bereit, um die Konsistenz der Datenbank zu überprüfen. Mit dem Statement

```
CHECK KEY INTEGRITY FOR TABLE schema.foo;
```

können alle Fremdschlüssel, Primärschlüssel und UNIQUE-Constraints überprüft werden.

Bei den Primärschlüsseln und UNIQUE-Constraints prüft das Tool, ob diese in der gesamten partitionierten Tabelle nur einmal vorkommen. Die referen-

tielle Integrität wird unter Zuhilfenahme der Fremdschlüssel-Tabelle überprüft. Fehler können somit auch mit diesem Softwaretool nicht verhindert werden. Durch eine regelmäßige Überprüfung können Fehler aber schnell erkannt und (manuell) behoben werden.

Mit dem Hilfsprogramm *pgEPCheck.jar* bietet das Softwaretool auch die Möglichkeit, eine partitionierte Tabelle mittels eines einzigen Kommandozeilenbefehls zu überprüfen. Der Benutzer muss dabei nicht das Hauptprogramm starten. Dieses Hilfsprogramm ermöglicht es, die (referentielle) Integrität der Daten regelmäßig mittels Cron Jobs zu überprüfen. Der Benutzer erhält dabei nicht jedes Mal, wenn das Programm ausgeführt wurde, eine Benachrichtigung, sondern nur dann, wenn ein Fehler entdeckt wurde.

6 Softwaretests

Gegen Ende der Implementierung des Softwaretools folgten abschließende Funktions- und Performancetests. In diesem Kapitel werden der Ablauf und die Ergebnisse dieser Softwaretests dokumentiert. Für den Benutzer des Tools werden vor allem die Performancetests von Interesse sein, da diese einen Anhaltspunkt bieten, wie lange das Partitionieren einer nicht-partitionierten Tabelle in etwa dauert und wie hoch die zu erwartenden Performanceeinbußen während des Partitionierens sind.

6.1 Funktionstest

Bereits während der Entwicklung wurden die Funktionen aller Methoden des Softwaretools getestet. Am Ende der Entwicklung folgte abschließend noch ein ausführlicher Funktionstest. Dazu wurden die folgenden Tabellen angelegt:

```
CREATE TABLE s.bar_1 (
    a integer NOT NULL,
    b integer NOT NULL,
    c text,
    UNIQUE (a, b)
);

CREATE TABLE s.foo (
    a integer,
    b integer,
    c char(2),
    d double precision,
    e date,
```

```
    f timestamp,
    g integer,
    h integer,
    UNIQUE (g, h),
    PRIMARY KEY (a, b),
    FOREIGN KEY (g, h) REFERENCES s.bar_1 (a, b)
);

CREATE TABLE s.bar_2 (
    a integer NOT NULL,
    b integer NOT NULL,
    c text,
    PRIMARY KEY (a, b),
    FOREIGN KEY (a, b) REFERENCES s.foo (b, a)
);
```

Zusätzlich wurden diese Tabellen mit einigen Testdaten befüllt. Die Tabelle *s.foo* wurde bei den Funktionstests jeweils partitioniert. Die beiden anderen Tabellen dienten nur zum Testen der Fremdschlüssel. Die Tabellen und Daten wurden im Verlauf der Testfälle teilweise abgeändert, um das Softwaretool unter möglichst vielen unterschiedlichen Bedingungen und Szenarien zu testen.

Getestet wurden sowohl List- als auch Range-Partitionierung. Es wurde weiters darauf geachtet, dass alle Statements, die das Tool unterstützt, durch die Funktionstests abgedeckt werden. Auch Ausnahmen und die korrekte Ausnahmenbehandlung wurden überprüft. Alle Funktionstests konnten erfolgreich abgeschlossen werden.

Zusätzlich wurden einige Funktionen mit Testdaten des Forschungsprojektes *IDIOM Media Watch on Climate Change*[16] getestet. Zur Verfügung standen dabei zwei Tabellen mit je 10.000 und eine Tabelle mit 169.304 Datensätzen. Das Datenbankschema dieser Testdatenbank kann Listing 6.1 entnommen werden. Abermals wurden sowohl List- als auch Range-Partitionierung getestet. Auch diese Funktionstests verliefen alle erfolgreich.

[16] http://www.ecoresearch.net/climate/

Listing 6.1: Datenbankschema der Testdaten des Forschungsprojektes *IDIOM Media Watch on Climate Change*

```
CREATE TABLE public.content (
    content_length integer,
    content bytea,
    encoding character varying(50) DEFAULT 'latin-1'::character varying NOT NULL,
    content_id bigint NOT NULL,
    md5sum character(32) DEFAULT 0,
    content_type character varying(64)
);

CREATE TABLE public.content_ascii (
    content text,
    pos_tags text,
    md5sum character(32) DEFAULT '0'::bpchar,
    content_id bigint NOT NULL,
    sem_orient double precision,
    language_id character(2),
    content_stripped text,
    pos_tags_stripped text,
    content_stripped_regexp text
);

CREATE SEQUENCE test_content_ascii_content_id_seq
    INCREMENT BY 1
    NO MAXVALUE
    NO MINVALUE
    CACHE 1;

CREATE SEQUENCE test_content_html_content_id_seq
    INCREMENT BY 1
    NO MAXVALUE
    NO MINVALUE
    CACHE 1;

CREATE TABLE public.meta_document (
    document_id bigint NOT NULL,
    url character varying(1024),
    title character varying(1024),
    valid_from timestamp without time zone,
    deprecated_on timestamp without time zone,
    hierarchy_level smallint DEFAULT 0,
    last_seen timestamp without time zone,
```

```
    content_id bigint,
    lastmodified timestamp with time zone
);

CREATE SEQUENCE test_meta_document_document_id_seq
    INCREMENT BY 1
    NO MAXVALUE
    NO MINVALUE
    CACHE 1;

ALTER TABLE content   ALTER COLUMN content_id
  SET DEFAULT nextval('test_content_html_content_id_seq'::regclass);
ALTER TABLE content_ascii ALTER COLUMN content_id
  SET DEFAULT nextval('test_content_ascii_content_id_seq'::regclass);
ALTER TABLE meta_document ALTER COLUMN document_id
  SET DEFAULT nextval('test_meta_document_document_id_seq'::regclass);

ALTER TABLE ONLY content
  ADD CONSTRAINT test_content_html_pkey PRIMARY KEY (content_id);
ALTER TABLE ONLY meta_document
  ADD CONSTRAINT test_meta_document_pkey PRIMARY KEY (document_id);
ALTER TABLE ONLY content_ascii
  ADD CONSTRAINT test_pkey_content_id PRIMARY KEY (content_id);

CREATE INDEX test_idx_content_md5sum
  ON content USING btree (md5sum);
CREATE INDEX test_idx_meta_document_content_id
  ON meta_document USING btree (content_id);
CREATE INDEX test_idx_meta_document_url
  ON meta_document USING btree (url);
CREATE INDEX test_idx_meta_document_valid_from
  ON meta_document USING btree (valid_from);

ALTER TABLE ONLY meta_document
  ADD CONSTRAINT test_fkey_cont FOREIGN KEY (content_id)
  REFERENCES content(content_id);
```

6.2 Performancetest

Neben den Funktionstests wurden auch Performancetests durchgeführt. Für die Performancetests wurden abermals die Testdaten des Forschungsprojektes *IDOM Media Watch on Climate Change* (Listing 6.1 zeigt das Datenbankschema) verwendet. Die Tests beschränkten sich dabei ausschließlich auf das Partitionieren einer nicht-partitionierten Tabelle. Die gemessene Laufzeit ist nicht die gesamte Programmlaufzeit, sondern rein jene Zeit, die für das Verschieben der Daten benötigt wurde.

Die Tests wurden auf einem herkömmlichen Desktop-PC (AMD Turion™ 64 X2 Prozessor, 2048 MB RAM) unter dem Betriebssystem Ubuntu 9.04 durchgeführt. Als Datenbankmanagementsystem kam PostgreSQL Version 8.3.9 zum Einsatz. Während der Tests wurde das Betriebssystem und die Datenbank durch keine weiteren Programme oder Zugriffe belastet.

Für die Performancetests wurden die drei Tabellen *content*, *content_ascii* und *meta_document* partitioniert. Die Tabellen *content* und *content_ascii* beinhalteten je 10.000 Datensätze und hatten eine Gesamtgröße von 35 MB beziehungsweise 98 MB. In der Tabelle *meta_document* waren insgesamt 169.304 Tupel beziehungsweise 170 MB Daten gespeichert.

Die Ergebnisse des Performancetests können aus Tabelle 6.1 abgelesen werden. Die Spalte *PLpgSQLmoveLimit* zeigt an, welcher Wert für den gleich lautenden Parameter in der Konfigurationsdatei gesetzt wurde. Dieser Wert gibt an, wie viele Zeilen pro Funktionsaufruf (die Funktion, welche die Daten von der Eltern-Tabelle in die Partitionen verschiebt; siehe Kapitel 5.3.3) verschoben werden. Wird dieser Parameter beispielsweise auf den Wert 10 gesetzt, so werden bei einem Funktionsaufruf zehn Tupel verschoben. Danach wird geprüft, ob weitere Daten vorhanden sind und gegebenenfalls die Funktion nochmals aufgerufen. Bei 100 Tabelleneinträgen sind somit zehn Funktionsaufrufe notwendig. Wie sich aus Tabelle 6.1 erkennen lässt, hat dieser Parameter einen starken Einfluss auf die Laufzeit des Softwaretools. Es sollte hier vor allem kein zu geringer Wert gewählt werden.

Tabelle 6.1: Vergleich der Laufzeiten beim Partitionieren einer nicht-partitionierten Tabelle

Tabelle	Tupel	Tabellengröße	*PLpgSQLmoveLimit*	Laufzeit	Lfz / 1.000 Tupel	Lfz / MB
content	10.000	35 MB	3	116 sec	11,6 sec	3,31 sec
			10	52 sec	5,2 sec	1,49 sec
			50	25 sec	2,5 sec	0,71 sec
			100	22 sec	2,2 sec	0,63 sec
content_ascii	10.000	98 MB	3	125 sec	12,5 sec	1,28 sec
			10	58 sec	5,8 sec	0,59 sec
			50	37 sec	3,7 sec	0,38 sec
			100	39 sec	3,9 sec	0,40 sec
meta_document	169.304	170 MB	3	985 sec	5,8 sec	5,79 sec
			10	370 sec	2,2 sec	2,18 sec
			50	152 sec	0,9 sec	0,89 sec
			100	122 sec	0,7 sec	0,72 sec

In einer weiteren Testreihe wurde überprüft wie sich die Performance von `INSERT`- und `SELECT`-Statements während des Partitionierens der Datenbank verhält. Um dies zu überprüfen, wurden mehrmals die gleichen Statements ausgeführt. Wiederum ist zwischen den Tests der Parameter *PLpgSQLmoveLimit* in der Konfigurationsdatei des Softwaretools verändert worden. Außerdem wurden die gleichen Statements auch in der nicht-partitionierten Datenbank getestet, um einen Vergleichswert zu erhalten. Tabelle 6.2 zeigt die Ergebnisse dieser Performancetests.

Zunächst hat sich bei diesen Tests gezeigt, dass das Partitionieren einen deutlichen Einfluss auf die Performance der `INSERT`- und auch `SELECT`-Statements hat. Es ist während des Partitionierens mit einer deutlich längeren (bei diesen Tests mehr als doppelt so langen) Laufzeit dieser Statements zu rechnen. Weiters lässt sich erkennen, dass durch die Wahl eines höheren Wertes des Parameters *PLpgSQLmoveLimit* vor allem die Ausführungszeit des `INSERT`-Statements ebenfalls erhöht wird. Bei den Ausführungszeiten des `SELECT`-Statements war bei den Tests auffällig, dass mit höheren Werten des Parameters *PLpgSQLmoveLimit* die Ausführungszeiten stärker variiert haben.

Tabelle 6.2: Vergleich der `INSERT`- und `SELECT`-Performance während des Partitionierens der Tabelle *content_ascii*

		PLpgSQLmoveLimit			
	nicht-partitioniert	3	10	50	100
`INSERT`-Statement	0,567 ms	1,242 ms	1,382 ms	1,688 ms	1,837 ms
`SELECT`-Statement	1,490 ms	2,613 ms	2,730 ms	3,675 ms	2,583 ms

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde zunächst in Kapitel 2 untersucht, welche Datenbanktechnologien zur Handhabung großer Datenmengen derzeit existieren und in welcher Form diese von PostgreSQL unterstützt werden. Zu diesen Technologien zählen Indizes, Materialized Views, vertikale und horizontale Partitionierung.

Bei den Indizes gibt es eine Vielzahl unterschiedlicher Indextypen, welche größtenteils auch von PostgreSQL unterstützt werden. Das Anlegen von Materialized Views ist hingegen derzeit nur über ein externes Tool möglich, das jedoch nicht an die Funktionen und Performance anderer (kommerzieller) Datenbanken herankommt. Horizontale Partitionierung wird zwar von PostgreSQL unterstützt, ist aber zurzeit nur mit einem relativ hohen Aufwand realisierbar.

Für vertikale Partitionierung sind gängige DBMS meist nicht optimiert. In diesem Zusammenhang werden oft auch spaltenorientierte DBMS erwähnt. Solche Systeme werden oft als Ansatzpunkt für mögliche zukünftige Innovationen genannt. Aktuellere Fragestellungen werden sich jedoch nicht nur mit innovativen Technologien in diesem Bereich beschäftigen, sondern vor allem damit, wie die verfügbaren Technologien optimal in der jeweiligen Datenbank zur Anwendung kommen. Der optimale Einsatz und Mix dieser Technologien stellt derzeit oft noch eine sehr große Herausforderung dar.

Neben diesen Datenbanktechnologien bietet PL/Proxy einen interessanten Ansatz für den Umgang mit großen Datenbanken. Mit diesem Tool lassen sich relativ einfach PostgreSQL Datenbank-Cluster erstellen. Mit PL/Proxy werden die Daten nicht auf mehrere Tabellen innerhalb einer Datenbank

aufgeteilt, sondern auf mehrere Datenbanken verteilt. Interessant ist auch das Konzept hinter dem Round Robin Database Tool. Bei einer Round Robin Database bleibt der verwendete Speicherplatz durch Konsolidierung älterer Daten immer konstant.

In Kapitel 4 wurden Softwaretools zur Analyse der Nutzung von PostgreSQL Datenbanken zusammengefasst. Diese Tools sollen den Benutzer dabei unterstützen, geeignete Maßnahmen zur Performancesteigerung der Datenbank zu setzen. Wie sich gezeigt hat, gibt es für PostgreSQL Datenbanken eine Vielzahl solcher Softwaretools.

Weiters wurde in dieser Arbeit die Entwicklung eines Softwaretools zur Partitionierung von PostgreSQL Datenbanken beschrieben. Mit Hilfe dieses Tools können Tabellen mit einfachen SQL-ähnlichen Statements partitioniert werden. Außerdem kann der Benutzer nicht-partitionierte Tabellen mit Hilfe des Tools partitionieren und Partitionen teilen oder zusammenführen.

Das Tool bietet bereits viele wichtige Funktionen, die zum Partitionieren und Managen von Partitionen benötigt werden. Für zukünftige Versionen bieten sich auch noch weitere Funktionen an, die in das Softwaretool integriert werden könnten. So wäre beispielsweise, speziell bei Partitionierung nach einem Datumsattribut, eine Funktion, die neue Partitionen automatisch anlegt, sehr hilfreich. Weiters können Daten, die außerhalb der Partitionen liegen, derzeit nicht erfasst werden. Für zukünftige Entwicklungen könnte man auch hier nach einer geeigneten Problemlösung suchen. Auch die Datenbank PostgreSQL entwickelt sich im Bereich der Partitionierung zur Zeit stetig weiter. Diese Entwicklungen sollten in zukünftigen Versionen des Tools berücksichtigt werden.

Literaturverzeichnis

[Abadi2007] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[Agrawal2004a] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004.

[Agrawal2004b] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Narasayya Vivek, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *In Proceedings of the VLDB 2004 Conference*, pages 1110–1121, 2004.

[Ahuja2006] Rav Ahuja. Introducing DB2 9, part 2: Table partitioning in DB2 9, Mai 2006. http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja2/, Abruf am 13.10.2009.

[Ailamaki2004] Anastassia Ailamaki and Stratos Papadomanolakis. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 383. IEEE Computer Society, 2004.

[Albrecht2001] Jens Albrecht. Business Intelligence mit Oracle9i. *Datenbank-Spektrum*, 1(1):36–43, 2001.

[Albrecht2006] Jens Albrecht and Marc Fiedler. Datenbank-Tuning - einige Aspekte am Beispiel von Oracle 10g. *Datenbank-Spektrum*, 6(16):26–33, 2006.

[Bartl2003] Peter Bartl, Dieter Jungmann, Frank Hermel, and Mario Müller. Eine Hochleistungsdatenbank für den liberalisierten Energiemarkt. *Datenbank-Spektrum*, 3(7):38–47, 2003.

[Chang2006] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, pages 205–218, 2006.

[Chaudhuri1995] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. *International Conference on Data Engineering*, page 190, 1995.

[Chen2007] Whei-Jen Chen, Alain Fisher, Aman Lalla, Andrew D. McLauchlan, and Doug Agnew. *Database partitioning, table partitioning, and MDC for DB2 9*. IBM Corp., Riverton, NJ, USA, 2007.

[Dorendorf2003] Stefan Dorendorf. Reorganisationsbedarfsanalysen bei relationalen Datenbankmanagementsystemen unter Beachtung der Workload. *Datenbank-Spektrum*, 3(5):51–61, 2003.

[EPQA2009a] Enterprise Postgres Query Analyser. http://epqa.sourceforge.net/, Abruf am 30.09.2009.

[EPQA2009b] Enterprise Postgres Query Analyser (EPQA). http://sourceforge.net/projects/epqa/, Abruf am 30.09.2009.

[Furtado2006] Pedro Furtado. Node partitioned data warehouses: Experi-

mental evidence and improvements. *Journal of Database Management*, 17(2):43–61, 2006.

[Goldstein2001] Jonathan Goldstein and Per-Ake Larson. Optimizing queries using materialized views: a practical, scalable solution. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 331–342, New York, NY, USA, 2001. ACM Press.

[Graefe2003] Goetz Graefe. Partitioned b-trees - a user's guide. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz*, pages 668–671. GI, 2003.

[Halevy2001] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[Hrle2004] Namik Hrle and Andreas Müller. Real-Time Statistics: Baustein für ein selbst verwaltendes Datenbanksystem DB2 für z/OS und OS/390. *Datenbank-Spektrum*, 4(8):41–45, 2004.

[Java2010a] Inc. Sun Microsystems. About the java technology. http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html, Abruf am 18.01.2010.

[Java2010b] Inc. Sun Microsystems. Java SE technologies - database. http://java.sun.com/javase/technologies/database/index.jsp, Abruf am 18.01.2010.

[Kulev2007] Milen Kulev and Peter Welker. Data Warehouses & Open-Source-Datenbanken. *Datenbank-Spektrum*, 7(22):25–36, 2007.

[Marek1992] Robert Marek and Erhard Rahm. Performance evaluation of parallel transaction processing in shared nothing database systems. In *PARLE '92: Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 295–310, London, UK, 1992. Springer-Verlag.

[Markl1999] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *IDEAS '99: Proceedings of the 1999 International Sym-*

posium on Database Engineering & Applications, pages 165–177, Washington, DC, USA, 1999. IEEE Computer Society.

[Morales2009] Tony Morales and Paul Lane. *Oracle Database VLDB and Partitioning Guide 11g Release 2 (11.2)*. Oracle, August 2009. http://download.oracle.com/docs/cd/E11882_01/server.112/e10837.pdf, Abruf am 13.10.2009.

[Noyer2007] Ulf Noyer, Martin Ruschinzik, and Jürgen Rataj. Teile und herrsche - tabellenpartitionierung in postgresql. *iX - Magazin für professionelle Informationstechnik*, April 2007, 2007.

[PQA2009a] Practical Query Analysis. http://pqa.projects.postgresql.org, Abruf am 24.09.2009.

[PQA2009b] PgFoundry: Practical Query Analyzer. http://pgfoundry.org/projects/pqa, Abruf am 24.09.2009.

[PlProxy2009a] Skype. Pl/proxy. https://developer.skype.com/SkypeGarage/DbProjects/PlProxy, Abruf am 13.10.2009.

[PlProxy2009b] Pgfoundry: Pl/proxy. http://pgfoundry.org/projects/plproxy/, Abruf am 13.10.2009.

[PlProxy2009c] Skype. Postgresql at skype. https://developer.skype.com/SkypeGarage/DbProjects/SkypePostgresqlWhitepaper, Abruf am 13.10.2009.

[PostgreSQL2009a] The PostgreSQL Global Development Group. *PostgreSQL 8.3.6 Documentation*. http://www.postgresql.org/files/documentation/pdf/8.3/postgresql-8.3-A4.pdf, Abruf am 02.08.2009.

[PostgreSQL2009b] The PostgreSQL Global Development Group. *PostgreSQL 8.4.0 Documentation*. http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4.0-A4.pdf, Abruf am 02.08.2009.

[PostgreSQL2009c] The PostgreSQL Global Development Group. *PostgreSQL: About*. http://www.postgresql.org/about/, Abruf am 02.08.2009.

[RRDtool2009a] Rrdtool - rrdtutorial. http://oss.oetiker.ch/rrdtool/tut/rrdtutorial.en.html, Abruf am 13.10.2009.

[RRDtool2009b] Rrdtool - rrdtool. http://oss.oetiker.ch/rrdtool/doc/rrdtool.en.html, Abruf am 13.10.2009.

[Zdonik2008] Stan Zdonik. Revolutionize database performance. *DM Review*, 18(1):20, 2008.

[pgFouine2009a] pgfouine - a postgresql log analyzer. http://pgfouine.projects.postgresql.org/, Abruf am 30.09.2009.

[pgFouine2009b] Pgfoundry: pgfouine. http://pgfoundry.org/projects/pgfouine/, Abruf am 30.09.2009.

[pgstatsinfo2009a] pgstatsinfo: Project home page. http://pgstatsinfo.projects.postgresql.org/, Abruf am 30.09.2009.

[pgstatsinfo2009b] Pgfoundry: pgstatsinfo. http://pgfoundry.org/projects/pgstatsinfo/, Abruf am 30.09.2009.

[pgstatspack2009] Pgfoundry: pgstatspack. http://pgfoundry.org/projects/pgstatspack/, Abruf am 30.09.2009.