Bachelorarbeit

Titel der Bachelorarbeit:

Inkrementelle Datenbank-Sicherung

Englischer Titel der Bachelorarbeit:

Incremental Database Backup

Verfasser/in: (Vorname, Nachname)	Thomas Pascher
Matrikelnummer:	0603646
Studium:	BaWISO 2006
Beurteiler/in: (Titel, Vorname, Nachname)	Dipl.-Ing. Mag. Dr. Albert Weichselbraun

Hiermit versichere ich, dass

1.) ich die vorliegende Bachelorarbeit selbständig und ohne Verwendung unerlaubter Hilfsmittel verfasst habe. Alle Inhalte, die direkt oder indirekt aus fremden Quellen entnommen sind, sind durch entsprechende Quellenangaben gekennzeichnet.

2.) die vorliegende Arbeit bisher weder im In- noch im Ausland zur Beurteilung vorgelegt bzw. veröffentlicht worden ist.

3.) diese Arbeit mit der beurteilten bzw. in elektronischer Form eingereichten Bachelorarbeit übereinstimmt.

4.) (nur bei Gruppenarbeiten): die vorliegende Arbeit gemeinsam mit (Vorname, Nachname) entstanden ist. Die Teilleistungen der einzelnen Personen sind kenntlich gemacht, ebenso wie jene Passagen, die gemeinsam erarbeitet wurden.

_____ _____

Datum Unterschrift

Abstrakt

Heutzutage müssen Datenbanken immer größere Datenmengen verarbeiten können. Für diverse existierende Datenbanksysteme wurden verschiedenste Lösungen entwickelt, um die Sicherung der Daten effizient und sicher zu gestalten. In dieser Arbeit werden einige dieser Lösungen vorgestellt und miteinander verglichen. Anschließend wird eine eigene Lösung zur Sicherung von Datenbeständen der frei verfügbaren Datenbanksystem PostgreSQL entwickelt. Im vorliegenden Fall werden geänderte, gelöschte und neue Datensätze aus der Datenbank mittels eines Hilfsprogrammes ausgelesen und in eine schematisch gleiche Datenbank gespeichert. Aus dieser Datenbank sollen die Daten im Anschluss per Java-Programm ausgelesen und in Textdateien gespeichert werden. Bei einem Absturz der Originaldatenbank können die Daten mithilfe dieser Textdateien lückenlos wiederhergestellt werden.

Abstract

Nowadays, databases have to be able to deal with big amounts of data. People developed many solutions for the various existing database systems, which should make the backup of data efficient and save. This work describes some of these solutions and compares their advantages and disadvantages. The next step is the development of a backup solution for data of the open source database system PostgreSQL. This solution is able to detect changed, deleted and new records with a separate tool and saves them to another database, which has the same scheme as the original database. A Java programme should then read out the data from this database, write it to a text file and save it. If the original database breaks down, the text files could be used for restoring the database.

Keywords

Incremental, Backup, Replication, Database, PostgreSQL, Java, Skytools, Londiste

Inhaltsverzeichnis

1. Einführung und Motivation

In Zeiten wachsenden Informationsbedarfs gewinnen Datenbanken immer mehr an Bedeutung. Die Technik auf diesem Gebiet entwickelt sich rasend schnell, ebenso werden auch Datenbestände immer größer. Um diese riesigen Datenbestände effizient und schnell sichern zu können, bedarf es unter Umständen sehr viel Zeit (bis zu mehreren Tagen oder Wochen). Aus diesem Grund wurden diverse Verfahren und Techniken entwickelt, um die Dauer für solche Sicherungsvorgänge zu minimieren.

In dieser Arbeit werden verschiedene traditionelle Ansätze von Datensicherung untersucht und deren Vor- und Nachteile aufgezeigt. Ebenso wird eine eigene Lösung zur Sicherung von Datenbeständen entwickelt. Dabei dient PostgreSQL als Datenbankmanagementsystem. Diese Lösung und sämtliche im Zuge dieser Arbeit verwendete Software sind frei erhältlich (Open Source). Die Verwendung von frei verfügbarer Software spart einerseits Kosten, soll aber andererseits auch zeigen, dass man heutzutage mit kostenlosen Programmen ebenso effizient arbeiten kann, wie mit kommerzieller Software.

Im Folgenden wird auch der Ausdruck Replikation verwendet. Replikation bezeichnet die mehrfache Speicherung von Daten an typischerweise unterschiedlichen Standorten. [Kudraß 2007]

2. Problemstellung und Forschungsfrage

Ziel dieser Bachelorarbeit am Institut für Informationswirtschaft der Wirtschaftsuniversität Wien ist es, ein Datensicherungs-Programm für die inkrementelle Sicherung von Datenbeständen zu erstellen. Im konkreten Fall handelt es sich um ein inkrementelles Datensicherungs-Programm für den *Media Watch on Climate Change (www.ecoresearch.net/climate).* In diesem System werden verschiedenste Technologien miteinander vereint, beispielsweise die automatisierte Informationsgewinnung aus dem Internet mit natürlicher Sprachverarbeitung. Durch die Arbeit von automatisierten Programmen (sogenannten Webcrawlern) werden 150 angloamerikanische Nachrichtenseiten durchsucht. Die dabei generierten Informationen werden bezüglich ihrer Verwendbarkeit von einem Inhaltsfilter bearbeitet, sodass nur Texte mit Umweltkontext verwendet werden. Jede Woche werden auf diese Art etwa 300.000 Artikel und Dokumente gewonnen, die in einer PostgreSQL-Datenbank (siehe Kapitel 3.4.1) gespeichert werden. Man spricht dabei von mirroring (spiegeln), da die Artikel eins zu eins übernommen werden [Hubmann et al 2009].

Da eine Datensicherung nur dann sinnvoll ist, wenn man verschiedene physische Datenträger einsetzt, muss man auch die möglichen Übertragungswege berücksichtigen. Meist wird man dabei auf Netzwerkverbindungen oder sogar das Internet und die damit zur Verfügung stehenden Bandbreiten zurückgreifen müssen. Im Sommer 2009 betrug die gespeicherte Datenmenge in der behandelten Klimadatenbank etwa 1,5 Terabyte. Eine Vollsicherung dieser Daten benötigte bereits gut eine Woche. Dabei können Probleme auftreten, zum Beispiel wenn es während der Sicherung weitere Änderungen in der Datenbank gibt. Damit ist eine Vollsicherung nach dem Ende des Sicherungsvorganges bereits wieder eine veraltete Version und es können Inkonsistenzen auftreten. Es kann aber auch zu Netzwerkausfällen kommen, womit der Sicherungsvorgang unterbrochen wird und man diesen neu durchführen muss.

Es gibt verschiedene Ansätze, wie man diese Probleme in den Griff bekommen kann. Grundsätzlich sollte man in so einem Fall die Sicherungszeit verkürzen, dabei kommen inkrementelle Siche-

rungen ins Spiel. Inkrementell bedeutet in diesem Kontext, dass bei einem Sicherungsvorgang nicht die kompletten Daten gesichert werden müssen, sondern lediglich jene Datensätze, welche seit der letzten Sicherung hinzugefügt, verändert oder gelöscht wurden. Je nach Intervall, in dem man solche inkrementellen Sicherungen durchführt, verkürzt sich die Zeit für einen Sicherungsvorgang erheblich (im Vergleich zu einer Vollsicherung).

Die Entwicklung einer Lösung zur inkrementellen Datensicherung für große PostgreSQL-Datenbanken, wie jene des oben beschriebenen Systems, bildet daher die Problemstellung der vorliegenden Bachelorarbeit. Spezielle Berücksichtigung gilt dabei der Minimierung von Sicherungszeiten und Speicherplatzanforderungen, um die erwähnten Nachteile einer Vollsicherung sowie auch Nachteile anderer Replikations- und Sicherungsverfahren (siehe Kapitel 3.2) zu umgehen.

3. Datenbanken und Sicherungsmethoden im Überblick

Dieses Kapitel soll dem Leser einen allgemeinen Überblick über Datenbanken und deren Verwendungsmöglichkeiten geben. Danach wird speziell auf die verschiedenen Sicherungsstrategien in kommerziellen sowie in Open Source Datenbanken eingegangen.

3.1 Datenbanken und deren Einsatzgebiete

Datenbanken sind aus der heutigen Zeit nicht mehr wegzudenken und sind zentraler Bestandteil von fast jedem Software-System. Dabei muss an dieser Stelle eine wichtige Unterscheidung getroffen werden, zwischen einer Datenbank an sich und einem Datenbankmanagementsystem (kurz DBMS). Beides zusammen ergibt ein Datenbanksystem (kurz DBS). In der Theorie versteht man unter dem Begriff Datenbank einen logisch zusammenhängenden Datenbestand. Dieser Datenbestand wird von einem laufenden DBMS verwaltet und für Anwendungssysteme und Benutzer unsichtbar auf nichtflüchtigen Speichermedien abgelegt. Neben den eigentlichen Daten werden auch Metadaten gespeichert, dazu zählen Angaben über die vorhandenen Tabellen und deren Beziehungen (auch Datenschemata) sowie Angaben über die Benutzer der Daten und deren Zugriffsrechte. Diese sind ein wichtiger Bestandteil der Datensicherheit. [Kudraß 2007]

Datenbankmanagementsysteme stellen einen kritischen Teil der Infrastruktur vieler Unternehmen und Behörden dar. Besonders die Verfügbarkeit, Vollständigkeit und Richtigkeit der Daten spielen dabei für die Anwender von Datenbank-Lösungen eine wichtige Rolle. Sämtliche Banken verwalten beispielsweise Kontodaten ihrer Kunden mithilfe von Datenbanken [Kudraß 2007]. Unvorstellbar wäre ein Szenario, in dem die Datenbank einen falschen Kontostand oder jenen, einer anderen Person liefern würde. Ein weiteres Szenario wäre, dass ein Datenbanksystem für die Verwaltung riesiger Lagerhallen falsche Standorte von benötigten Produkten für die Kommissionierung liefert, sodass wertvolle Zeit bei diesem Arbeitsvorgang verloren gehen würde.

Um solchen und ähnlichen Unannehmlichkeiten vorzubeugen, wurden DBMS seit den Ursprüngen in den Sechzigerjahren des vergangenen Jahrhunderts ständig weiterentwickelt und perfektioniert. Mit zunehmender Wichtigkeit und wachsenden Datenbeständen bildete die Datensicherung einen wichtigen Bestandteil dieser Entwicklungen. Die Sicherungsstrategien müssen dafür sorgen, dass die Daten im Ernstfall schnell zum letztmöglichen Zeitpunkt wiederhergestellt werden können. Doch es muss auch auf die Leistungsaspekte Rücksicht genommen werden. Ziel ist es, dass Sicherungen bei laufender Datenbank und mit minimalen Ressourcen durchgeführt werden, sowie dass die Rücksicherungen in einem definierten Zeitrahmen möglich sind. Idealerweise sorgt der Administrator für eine lokale sowie eine entfernte Sicherung. Dies wird auch Disaster Recovery Planning (DRP) genannt [Abramson et al 2004].

Verschiedene Möglichkeiten zur Replikation der Datenbestände wurden entwickelt, diese werden im nächsten Abschnitt genauer erläutert. Ebenso werden nachfolgend einige DBMS beschrieben und deren jeweilige Sicherungsmöglichkeiten aufgezeigt.

3.2 Traditionelle Lösungsansätze zur Datensicherung

Nicht nur der Wunsch der Datenbank-Betreiber nach ständiger Verfügbarkeit, Vollständigkeit und Richtigkeit der Daten für ihre Kunden spielt im Datenbankalltag eine wichtige Rolle, auch gesetzliche Bestimmungen wurden nach und nach eingeführt, um die Sicherheit der Daten zu gewährleisten. Dabei wurden unterschiedliche Bestimmungen für diverse Branchen eingeführt (Banken, Versicherungen usw.). [Kudraß 2007] Dies soll jedoch hier nur am Rande erwähnt bleiben und würde in seiner kompletten Ausführung auch den Rahmen dieser Arbeit sprengen.

Der folgende Abschnitt behandelt die derzeit am häufigsten verwendeten Strategien der Datensicherung. Es wird das Verfahren beschrieben und auf die Vor- und Nachteile eines jeden Verfahrens eingegangen. Außerdem werden die unterschiedlichen Möglichkeiten anhand von ausgewählten Datenbanksystemen aus dem kommerziellen sowie dem Open Source Bereich erläutert.

Grundsätzlich kann man unterscheiden zwischen Online- und Offline-Backups. Bei einem Online-Backup werden die Benutzer nicht von der Datenbank getrennt und können während der Sicherung weiter an den Daten arbeiten, während beim Offline-Backup die Benutzer von der Datenbank getrennt sind, diese heruntergefahren wird und anschließend die Sicherung vorgenommen wird. Selbstverständlich muss man bei Online-Backups mehr Dinge berücksichtigen, um die Konsistenz der Daten zu gewährleisten. Bei den beschriebenen Sicherungslösungen in diesem Kapitel werden jedoch keine Unterschiede zwischen Online- und Offline-Modus angeführt, da es sich hier nur um eine allgemeine Beschreibung der Funktionsweisen handelt.

3.2.1 Datenbank-Dumping

Ein so genannter Datenbank-Dump wird erstellt, um eine Datenbank zu sichern. Die Daten aus der Datenbank werden exportiert und an einem anderen Speicherplatz physisch gespeichert. Die Form eines solchen Extrakts unterscheidet sich je nach Datenbanksystem. Das Ergebnis kann dabei eine komprimierte Datei sein, oder auch eine Liste von SQL-Befehlen.
Es handelt sich dabei um eine Folge von SQL-Befehlen, mit deren Hilfe man die Datenbank im Fall von Abstürzen oder Datenverlust wiederherstellen kann. Ein solcher Datenbank-Dump bestehend aus SQL-Befehlen kann ein komplettes Abbild der Datenbank mit allen gespeicherten Daten und Metadaten sein, oder aber auch nur die Metadaten enthalten, um beispielsweise Projektentwicklungen von einer Testdatenbank in eine produktive Datenbank zu überführen. [Kudraß 2007]

```
--
-- Name: mirror_document; Type: TABLE; Schema: public; Owner: weblyzard;
--
CREATE TABLE mirror_document (
    mirror_id integer NOT NULL,
    document_id bigint NOT NULL,
    counter bigint NOT NULL
);


--
-- Dumping data for table mirror_document
--
 INSERT INTO `mirror_document` VALUES (1, 184464407370, 12345678910);
 INSERT INTO `mirror_document` VALUES (2, 234567890123, 98765432100);
```

Abbildung 1: Auszug eines Datenbank-Dumps

Abbildung 1 zeigt einen Auszug aus dem Datenbank-Dump der behandelten Klimadatenbank „The Media Watch on Climate Change“. Die vorangestellten doppelten Bindestriche bezeichnen hier Kommentare, die der Lesbarkeit für den Menschen dienen sollen. Die Maschine überspringt diese Kommentare bei der Ausführung. CREATE TABLE lautet die Anweisung zur Erstellung einer Tabelle. In diesem Fall lautet der Name der Tabelle ‚mirror_document‘. Weiters werden die Inhalte dieser Tabelle definiert, insgesamt enthält sie drei Spalten (mirror_id, document_id, counter). Für jede dieser Spalten wird der Datentyp definiert, welche den Inhalt der Spalte definieren. So darf als mirror_id nur ein ganzzahliger Wert eingetragen werden (Integer). In diesem Fall werden neben der Definition der Tabelle auch zwei Datensätze eingetragen. Dies geschieht mittels des Befehls INSERT INTO. Die Werte, welche eingefügt werden sollen, werden durch VALUES in der folgenden Klammer beschrieben. Im ersten Fall wird die mirror_id ‚1‘ vergeben und je ein fiktiver Wert für document_id und counter zu Demonstrationszwecken.

Vorteile von Datenbank-Dumping:

Wenn man in regelmäßigen Abständen solche Datenbank-Dumps erstellt (zB täglich), so kann man bei Datenverlusten auf einzelne Vollsicherungen zurückgreifen und somit die Daten wiederherstellen. Außerdem kann man die Zeit zwischen dem letzten Dump und einem Absturz (inklusive dem daraus resultierenden Datenverlust) möglichst gering halten und mit vergleichsweise geringem Aufwand wiederherstellen. Das bedeutet auch, dass jeder Dump für sich unabhängig ist und für sich allein zur Wiederherstellung des Datenbestandes verwendet werden kann (**Versionsunabhängigkeit**). Ebenso eignen sich Datenbank-Dumps auch hervorragend, um bei einer Umstellung des Betriebssystems die Daten lückenlos auf das neue System zu übertragen (**Portabilität**), da ja in einem Dump die vollständigen Informationen und Daten vorliegen und SQL-Kommandos vom DBMS auf dem neuen Betriebssystem gleich interpretiert werden. Ein weiterer Vorteil ist die **Menschenlesbarkeit** des Dumps. Wie im Beispiel beschrieben, kann man mit wenigen SQL-Kenntnissen die Zeilen aus dem vorhergegangenen Beispiel ohne Probleme interpretieren.

Nachteile von Datenbank-Dumping:

Die benötigte Zeit ist jenes Kriterium, welches bei Dumps mit vollständigen Daten die größten Nachteile aufzeigt. Die Erstellung solcher Dumps kann je nach Datenmenge mehrere Tage oder sogar Wochen in Anspruch nehmen. In dieser Zeit werden höchstwahrscheinlich Änderungen an der Datenbank durchgeführt, die nicht mehr in den Dump aufgenommen werden können. Weiters wird die Leistung des Datenbanksystems während der Sicherung nachlassen, da das System einen Teil der Ressourcen für die Sicherung benötigt. Außerdem wird für die Speicherung von mehreren Dumps auch sehr viel Speicherplatz auf dem Sicherungsmedium benötigt. Man betrachte hier noch einmal die 1,5 Terabyte der Klimadatenbank. Unter der Annahme, wöchentliche Dumps zu erstellen, würde man in einem Monat 6,0 Terabyte an Daten produzieren, wobei ein Großteil der Daten in jedem einzelnen Dump gleich wäre.

Diese Nachteile treten lediglich bei Datenbank-Dumps mit vollständigen Daten auf, da ein Dump der Metadaten im Normalfall innerhalb kürzester Zeit erstellt ist.

3.2.2 Datenbank-Snapshots

Lang andauernde Datensicherungen im laufenden Betrieb bringen das Risiko inkonsistenter Sicherungsdaten mit sich. Das heißt konkret: wenn während der Datensicherung etwas verändert wird, kann nicht gewährleistet werden, dass diese Änderungen auf dem Sicherungsmedium berücksichtigt wurden.

Dieses Problem kann man unter anderem mit Snapshots (zu Deutsch Schnappschüsse) lösen. Hier wird eine neue Datenbank angelegt, mit den gleichen Strukturen wie die zu sichernde Datenbank. Diese Datenbank ist zu Beginn leer und beinhaltet alle Datensätze, die seit der Erstellung des Snapshots verändert wurden, jedoch in der ursprünglichen Version. Werden Daten in der Original-Datenbank verändert, so werden die alten Daten vor der Änderung in die Snapshot-Datenbank kopiert. Dies wird auch als ‚copy-on-write' bezeichnet, das heißt also, dass der alte Datensatz dann kopiert wird, wenn er verändert wird. Ziel dieses Verfahrens ist es, den Zustand zum Erstellungszeitpunkt des Snapshots sicherzustellen. Wenn ein Datensatz niemals verändert wird, so wird dieser auch niemals in die Snapshot-Datenbank kopiert werden. Will man die Daten zum Zeitpunkt des Snapshots wieder herstellen, so werden die geänderten Daten aus der Snapshot-Datenbank gelesen, während die unveränderten Daten aus der Original-Datenbank stammen. Will man auf frühere Zustände der Daten zurückspringen, so muss man schrittweise die erstellten Snapshots bis zum jeweils gewünschten Zustand integrieren. [dbresource01]

Diese Technik ist nicht zu verwechseln mit einem Filesystem-Snapshot. In den meisten Linux-Betriebssystemen gibt es die Möglichkeit, mit Hilfe des LVM (Logical Volume Manager) dynamisch veränderbare Partitionen zu bilden. Diese Partitionen werden Snapshot-Volumes (Volume bezeichnet einen festgelegten Bereich auf einem Speichermedium) genannt. Solche Snapshot-Volumes sind in diesem Zusammenhang nichts anderes, als die Kopie eines logischen Volumes. Logische Volumes sind unveränderbar und können daher keine Konsistenzprobleme bei der Da-

tensicherung verursachen. Ein Snapshot ist zwar nur eine Momentaufnahme eines logischen Volumes, speichert aber eigentlich nur dessen Veränderungen während der Datensicherung. Die Größe des Snapshot Volumes muss daher auch nicht der Größe des logischen Volumes entsprechen. Es reicht, wenn es groß genug ist, um die Datenänderungen während des Backups aufzunehmen. Das Snapshot Volume wird ganz normal in den Dateisystem-Baum eingebunden (im Nur-Lesen-Modus). Nach erfolgter Datensicherung kann das Snapshot Volume wieder aus dem Dateisystem-Baum entfernt und gelöscht werden. Weiters werden die Dateiänderungen während der Lebensdauer des Snapshot Volume auf das Original-Volume übertragen und der Normalbetrieb kann fortgesetzt werden [freistil-consulting01].

Vorteile von Datenbank-Snapshots:

Mit der regelmäßigen Erstellung von Snapshots werden immer neue Datenbanken mit der alten Version der veränderten Datensätze erstellt. Diese Datenbanken benötigen (bei regelmäßiger Erstellung) nicht viel Speicherplatz und in Summe spart der Benutzer mit diesem Verfahren eine Menge Speicherplatz ein, da die doppelte (=redundante) Speicherung der unveränderten Daten im Gegensatz zu Datenbank-Dumps vermieden wird. Neben dem Speicherplatz wird ebenso Zeit gespart, da die Übertragung auf die Snapshot-Datenbank simultan zur Änderung von Datensätzen erfolgt.

Im Unterschied zu Datenbank-Dumps wird immer in die aktuelle Datenbank geschrieben und die geänderten Datensätze simultan in die Snapshot-Datenbank kopiert. Da dies zu jedem Zeitpunkt einer Änderung passiert, kann es nicht vorkommen, dass Datensätze während einer langen Vollsicherung wie beim Datenbank-Dumping verändert werden (Konsistenz).

Nachteile von Datenbank-Snapshots:

Die Zeitvorteile bei der Erstellung werden durch Zeitnachteile bei der Wiederherstellung der Daten eingebüßt. Um Daten zu einem bestimmten Zeitpunkt wiederherzustellen, muss man

schrittweise alle Snapshot-Datenbanken integrieren, die zwischen dem aktuellen Datenbestand und dem Zustand zum gewünschten Zeitpunkt liegen.

3.2.3 Write Ahead Logging (WAL)

Write Ahead Logging (WAL) ist eine Standardtechnik für das Aufzeichnen von Transaktionen. Dieses Aufzeichnen nennt man auch Logging. Vor dem eigentlichen Ausführen von Transaktionen in der Datenbank werden diese in einer Logdatei protokolliert. Im Falle eines auftretenden Fehlers kann die Transaktion mithilfe der Logdatei wiederholt werden und auch wieder rückgängig gemacht werden. Wenn Änderungen der Datenseiten zunächst festgehalten werden, so nennt man das REDO oder Vorwärts-Wiederherstellung. Werden Änderungen bei nicht abgeschlossenen Transaktionen aus den Datenseiten entfernt, so nennt man das UNDO oder auch Rückwärts-Wiederherstellung. [neogrid01]

Wird beispielsweise ein Datensatz gelöscht, so passiert dies nicht unmittelbar in der Datenbank sondern wird zunächst in der Logdatei protokolliert. Damit gilt die Transaktion als abgeschlossen. Erst zu einem geeigneten Zeitpunkt wird der Inhalt der Logdatei in die Datenbank übertragen. Diese Zeitpunkte sind festgelegt und werden Checkpoints genannt. WAL-Logdateien besitzen eine bestimmte Größe, im Fall von PostgreSQL sind diese Dateien 16 Megabyte groß. Erreicht eine Datei zwölf Megabyte (das entspricht 75%), so wird eine neue Logdatei erstellt. Nach der Ausführung eines Checkpoints, wird in der Logdatei eine Markierung gesetzt, von dieser aus das System bei der nächsten Übertragung der Inhalte an der richtigen Stelle fortsetzen kann. Dabei kann es auch mehrere Checkpoints in einer Logdatei geben, es wird immer von der letzten Markierung fortgesetzt. Liegt der letzte Checkpoint bereits in einer neuen Logdatei, so wird die alte Logdatei entfernt, um den Speicherplatz freizugeben. [Hartwig 2001]

Vorteile von Write Ahead Logging:

Diese Art der Replikation hat eine Reihe von Vorteilen. Ein wichtiger Vorteil dieser Technik ist, dass die Anzahl der Schreibvorgänge auf die Festplatte erheblich verringert wird, da nur die Log-

datei am Ende jeder Transaktion auf die Festplatte zurückgeschrieben werden muss. Im Falle eines Absturzes bleibt die Datenbank mithilfe des Logs wiederherstellbar und konsistent. Durch die verringerten Schreibvorgänge liegt der **Geschwindigkeitsvorteil** auf der Hand, weil die Logdatei sequentiell geschrieben ist. Das macht eine Synchronisation effizienter als ein Rückschreiben der Daten. [postgresql 01]

Weiters wird die **Konsistenz** der Daten zu jeder Zeit gewährleistet, da durch die einzelne Eintragung von Änderungen in die Logdatei Informationsverluste ausgeschlossen werden. Ohne den WAL-Mechanismus würde eine einfache Änderung in der Datenbank mehrere Aktionen notwendig machen. Eine Datensatzänderung würde beispielsweise die Aktualisierung der Indextabelle zur Folge haben. Ein Systemabsturz zwischen der Änderung und der Indexaktualisierung hätte den Verlust der noch ausstehenden Aktionen zur Folge. Bei WAL wird nach einem Systemabsturz zunächst die Logdatei schrittweise (sequentiell) abgearbeitet. In diesem Fall wird das bereits erwähnte REDO durchgeführt. Alle Transaktionen, die vor dem Systemausfall noch nicht komplett geschrieben wurden, werden rückgängig gemacht (UNDO). Danach ist die Datenbank wieder im konsistenten Zustand. [Hartwig 2001]

Nachteile von Write Ahead Logging:

Den genannten Vorteilen steht entgegen, dass die WAL-Einträge für relativ lange Zeiträume gespeichert werden müssen. Im Falle eines ‚Transaktions-UNDO' müssen die Einträge so lange gespeichert werden, bis die längstmögliche Transaktion abgeschlossen ist. Die Einträge enthalten meist eine größere Anzahl an Schnappschüssen von Diskseiten und können daher eine beträchtliche Größe annehmen. Bei Anwendung des Prinzips auf große Datenmengen empfiehlt sich daher die Verwendung irgendeiner Art von Komprimierung. [postgresql02]

3.3 Lösungen in kommerziellen Datenbanken

In diesem Kapitel werden die verschiedenen Datensicherungsmöglichkeiten in den Datenbanksystemen der Firmen Oracle und IBM beschrieben, da diese auf dem Sektor eine bedeutende Rolle spielen.

3.3.1 Oracle Database

Die Oracle Corporation ist mit mehr als 70.000 Mitarbeiterin in 145 Ländern einer der größten Softwarehersteller der Welt. Die Firma hat ihren Hauptsitz in Silicon Valley (Kalifornien, USA). Das bekannteste und auch erfolgreichste Produkt ist Oracle Database, das Datenbankmanagementsystem. [oracle01]

In diesem Datenbanksystem gibt es eine umfangreiche Implementierung von Sicherungsoptionen. Das Gesamtkonzept nennt sich Oracle Backup und Recovery (Sicherung und Wiederherstellung). Drei grundsätzliche Unterscheidungen werden getroffen:

Image Backup und Recovery – Sicherungsoperationen werden mit der gesamten Datenbank und allen Daten durchgeführt. Alle Dateien werden gemeinsam gesichert, um sie auch gemeinsam für eine etwaige Wiederherstellung verwenden zu können. [Abramson et al 2004]

Logisches Backup und Recovery – Sicherungsoperationen werden nur mit einer bestimmten logischen Datenbankstruktur (Tabellen, Indices) durchgeführt. Somit kann die Datenbank in feineren Strukturen gesichert werden, als beim Image Backup. Die verwendeten Tools in Oracle Database heißen Data Pump Export und Data Pump Import. Achtung: Logische Replikationen können nicht für eine komplette Wiederherstellung verwendet werden, mit ihnen ist lediglich das Zurückschreiben der Daten möglich (Restore). [Abramson et al 2004]

Der Oracle Recovery Manager (RMAN) – Mit diesem Tool werden Sicherungen auf kontrollierte Weise durchgeführt. Es werden generell Image-Kopien und Backups unterschieden. Image-Kopien sind vollständige Kopien der verwendeten Datendateien, sie enthalten sämtliche Daten und Metadaten. RMAN unterstützt bei der Durchführung von Datenbank-Backups sowohl vollständige als auch inkrementelle Sicherungen. Die inkrementelle Methode erfordert ein Level 0-Backup, was nichts anderes ist als eine vollständige Sicherung. Der Unterschied besteht darin, dass es die Basis für inkrementelle Backups darstellt. Oracle unterscheidet dabei weiter in differenzielle inkrementelle Backups und kumulative inkrementelle Backups. Das kumulative inkrementelle Backup sichert nur die Blöcke, die seit dem letzten Backup auf den untergeordneten Levels geändert wurden. Im Unterschied dazu speichert das differenzielle inkrementelle Backup alle geänderten Blöcke seit dem letzten Backup auf diesem oder einem niedrigeren Level. [Abramson et al 2004]

Differenzielles inkrementelles Level 2-Backup – es werden nur die Daten gesichert, die seit dem letzten Backup auf Level 2, 1 oder 0 geändert wurden.

Kumulatives inkrementelles Level 2-Backup – sichert alle Daten seit dem letzten Backup auf Level 1 oder 0.

Abbildung 2: Beispiel eines Level-2-Backups

Differenzielle Backups sichern weniger Daten, während kumulative Backups weniger Daten zurückschreiben und somit die Geschwindigkeit des Restore erhöhen. Bis zur Version 10g wurden Datenänderungen in Oracle mit Hilfe von Table Scans ermittelt und brauchten somit mehr Zeit als ein vollständiger Dump. Ab Version 10g wurde eine aus Bitmaps bestehende Datei zur Überwachung der Datenänderungen verwendet. [Abramson et al 2004]

Die bereits beschriebene Lösung mit Hilfe des WAL-Prinzips (siehe Kapitel 3.2.3) gibt es auch in Oracle, die Logdateien heißen hier Redo Logs. Jedoch ist diese Lösung nicht im Recovery Manager integriert. Wird eine Transaktion durchgeführt, so schreibt das System alle damit zusammenhängenden Aktionen (Redo Records) in die Redo Logdatei, zusammen mit einer eindeutigen System Change Number (SCN) und einem Timestamp. Damit ist die Konsistenz bereits sichergestellt, auch wenn die Änderungen in der Logdatei bleiben und noch nicht auf das Medium selbst geschrieben wurden. Ebenso wird, wie bereits beschrieben, die Rückwärts-Wiederherstellung mit Undo-Segmenten durchgeführt, als Auslöser werden auch hier Checkpoints verwendet. [Abramson et al 2004]

3.3.2 IBM DB2

Ein weiterer großer Spieler der gesamten Computerbranche ist IBM. Im Datenbanksektor besitzt IBM's Datenbanklösung DB2 einen beachtlichen Marktanteil. Mit der neunten Version von DB2 brachte IBM erstmals ein Hybridprodukt auf den Markt, welches sowohl XML-Daten als auch relationale Daten speichern kann. [ibm01]

IBM bietet mehrere Produkte als Lösung für die Replikation der Daten, wie zB den IBM InfoSphere Replication Server oder den DB2 DataPropagator. An dieser Stelle soll jedoch auf den ‚DB2 Replication V8' näher eingegangen werden. Hierbei handelt es sich um ein Programm, mit dessen Hilfe der Benutzer oder Administrator der Datenbank die Arten der Datenbanksicherung definieren kann. Genauer gesagt verwendet der Benutzer das Interface namens Replication Center.

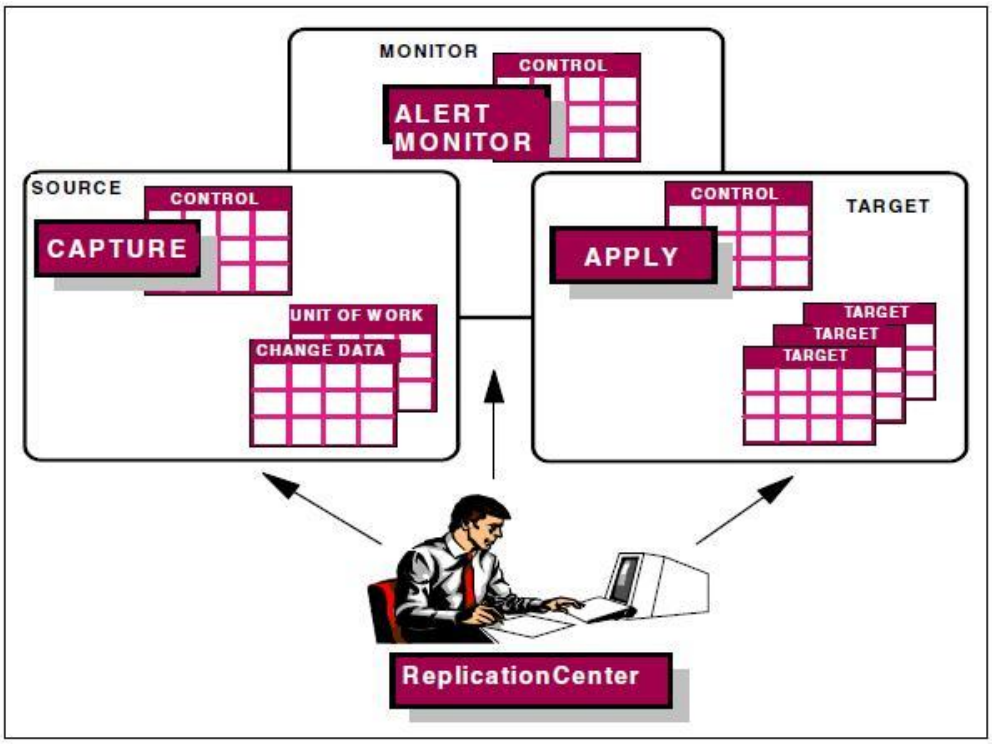

Abbildung 3: Verwendung des Replication Centers [ibm02]

Das Programm besteht im Wesentlichen aus drei Komponenten. Die **Capture-Komponente** sorgt dafür, dass Veränderungen in den Quelltabellen aufgegriffen und zwischengespeichert werden. Log Records werden geschrieben, die in weiterer Folge für die Replikation verwendet werden. Zu jeder Quelltabelle die gesichert werden soll, existiert eine Change-Data Tabelle, in der die geänderten Einträge gespeichert werden. Je nach Konfiguration geschieht die Übernahme der Log Records durch ein Commit-Ereignis oder durch das Ausführen einer vordefinierten Methode (stored procedure) [ibm02].

Die so gespeicherten Informationen werden anschließend von der **Apply-Komponente** entgegengenommen und in die Zieltabellen übertragen. Im Falle einer Speicherung auf einem entfernten Rechner läuft diese Komponente bereits auf einem anderen Rechner, der aber nicht Zwangsweise der Zielrechner sein muss. Es kann auch ein unabhängiger, dritter Rechner zwischengeschaltet werden, auf dem keine DB2-Datenbank läuft. Apply kann auf verschiedene Arten ausgeführt werden. So ist es möglich, einen fixen Zeitraum zu definieren, nachdem Apply ausgeführt wird, oder ein bestimmtes Ereignis sorgt für die Ausführung [ibm02].

Die dritte Komponente ist der **Alert-Monitor**. Dabei handelt es sich um ein Dienstprogramm, welches die beiden anderen Komponenten überwacht und nach Ausnahmesituationen sucht. Bei der Überschreitung von bestimmten Grenzwerten werden vorher definierte Aktionen durchgeführt, wie zum Beispiel der Versand von E-Mails oder SMS an verantwortliche Personen [ibm03]. Die Fehlermeldungen werden in einer eigenen Tabelle mitprotokolliert, um eine spätere Fehlersuche der alarmierten Personen möglich zu machen [ibm02].

3.4 Lösungen in Open Source Datenbanken

Im Folgenden werden die verschiedenen Lösungen in Open Source Datenbanken beschrieben. Open Source Software spielt in nahezu allen Bereichen der Informatik eine zunehmend größere Rolle. Auch im Bereich der Datenbanken hat freie Software ihren Einzug gehalten. Für diese Arbeit werden zwei frei verfügbare Datenbankmanagementsysteme näher betrachtet, PostgreSQL und MySQL.

3.4.1 PostgreSQL

„Das heutzutage als PostgreSQL bekannte objektrelationale Datenbankverwaltungssystem stammt vom POSTGRES-Paket der Universität von Kalifornien in Berkeley ab. Nach mehr als einem Jahrzehnt Entwicklungsarbeit ist es jetzt das fortschrittlichste Open Source Datenbanksystem.“ [postgresql 03]

Grundsätzlich gibt es zwei Ansätze. Die Replikation mit Hilfe von Datenbankmitteln und Unix Tools (copy, rsync) oder die Verwendung von speziellen Tools, wie beispielsweise pgpool, SQL-Relay, Slony-I oder Londiste [tu-chemnitz01].

Da zur Erstellung des Programmes für das inkrementelle Backup die Verwendung von Londiste angestrebt wird, soll dieses hier näher beschrieben werden. Londiste ist ein Produkt des Sky-

tools-Pakets (https://developer.skype.com/SkypeGarage/DbProjects/SkyTools), welches von und für Skype entwickelt wurde. Das Grundkonzept von Londiste ist eine Ticker-Datei, in der man festlegt, welche Tabellen für die Replikation benötigt werden. Dabei wird spezifiziert, auf welche Datenbank (Provider-Datenbank) zugegriffen werden muss und in welche Datenbank (Subscriber-Datenbank) die ausgelesenen Daten gespeichert werden. Dabei kommt das ‚Master to Multiple Slaves Konzept' zur Geltung. Das bedeutet, dass aus einer Provider-Datenbank in eine oder mehrere Subscriber-Datenbanken gespeichert werden kann. Sind alle Parameter konfiguriert, kann die Replikation gestartet werden. Dabei werden die geänderten Daten in die Subscriber-Datenbank geschrieben. Diese enthält also alle geänderten Datensätze. Dabei ist es vernünftig, in regelmäßigen Abständen neue Subscriber-Datenbanken zu erstellen. Kombiniert man diese Datenbank mit einem anfangs erstellten Datenbank-Dump, kann man schrittweise durch Integration der einzelnen Subscriber-Datenbanken verschiedene Datenstände wiederherstellen [postgresql04].

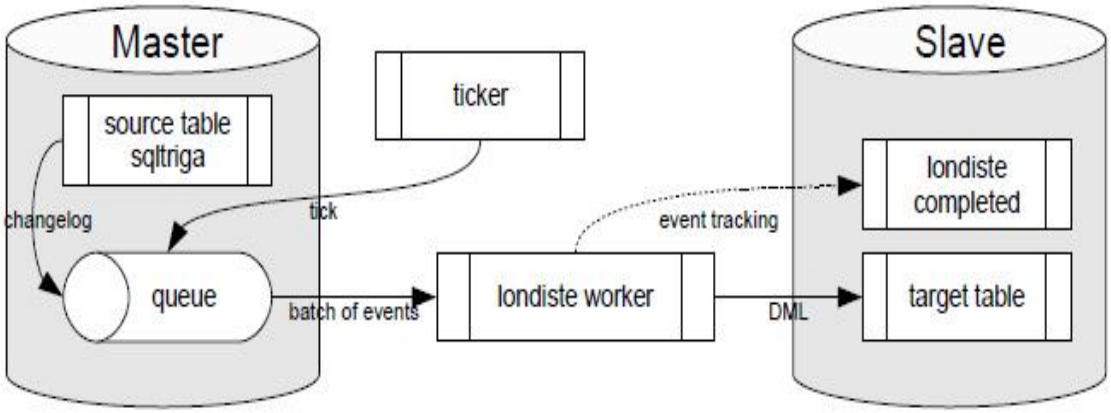

Abbildung 4: Master/Slave Darstellung von Londiste [postgresql05]

3.4.2 MySQL

Neben PostgreSQL ist vor allem MySQL aus der Datebankwelt der Open Source Entwicklungen nicht mehr wegzudenken. Ursprünglich wurde MySQL Server von der Firma MySQL AB in Schweden entwickelt. Im Jahr 2008 wurde MySQL AB von Sun Microsystems übernommen. Da Sun das

volle Copyright am gesamten Quellcode besitzt, gibt es auch kommerzielle Versionen für Unternehmen [mysql01].

MySQL unterstützt viele moderne Methoden der Replikation, Snapshots werden jedoch bis dato nicht unterstützt. Um mit Snapshots arbeiten zu können, werden externe Tools wie zB der Zamanda Recovery Manager benötigt. Das WAL-Prinzip wird in MySQL ‚Binary Logging' genannt, die grundsätzliche Funktion zur Erstellung von Dumps lautet mySQLDump [feyrer01].

Für kleine Datenbestände eignet sich der PhpMyAdmin perfekt. Es handelt sich dabei um ein grafisches Verwaltungstool, das bis ca. zwei Megabyte Datenbankgröße hervorragende Dienste bietet. Bei größeren Datenbanken versagen jedoch die Recovery-Funktionen. Dieses Problem löst man am besten mit dem Tool BigDump, welches in vernünftiger Geschwindigkeit auch mit großen Datenbeständen umgehen kann [phpbb01].

3.4.3 NoSQL-Datenbanken

NoSQL steht für *Not Only SQL* und beschreibt eine Art von Datenbanken, die einen nichtrelationalen Ansatz verfolgen, keine festgelegten Tabellenschemata benötigen und mit der Absicht arbeiten, Joins zu vermeiden [nosql01]. Diese Art von Datenbanken ist für große Datenmengen und eine hohe Änderungsrate dieser Daten besser geeignet als relationale Datenbanken, bietet jedoch als eine geringere Sicherheit der Datenkonsistenz. Das ergibt sich daraus, dass die Datenbestände in NoSQL-Datenbanken zumeist auf verschiedenen Maschinen redundant gehalten werden. Somit sind Daten grundsätzlich mehrmals vorhanden, jedoch kann jeder dieser redundanten Datenbestände für sich von Benutzern oder Programmen verändert werden. Für den Nutzer ist es unerheblich, an welchem Ort er sich befindet und mit welchem Datenbestand er verbunden ist. Solche Datenbanken werden auch von den großen sozialen Netzwerken wie Facebook und Twitter verwendet. Die große Herausforderung für die Entwickler ist es, die Änderungen der einzelnen Datenbestände allen anderen Datenbeständen mitzuteilen und diese zu synchronisieren. In diesem Zusammenhang gibt es den Begriff der *eventuellen Konsistenz.* Das bedeutet, dass man nach längeren Zeiten ohne Aktualisierung davon ausgehen kann, dass die

Datenbestände konsistent sind [uni-magdeburg01]. Auch bei NoSQL-Datenbanken gibt es verschiedene Ansätze bezüglich des Aufbaus der Datenbank, der Datenabfrage sowie der Backup-Strategien. Nachfolgend wird eine NoSQL-Datenbank in Bezug auf ihre Backup-Strategien beschrieben, um die generellen Unterschiede zu relationalen Datenbanken zu verdeutlichen.

Apache Cassandra wurde ursprünglich bei Facebook entwickelt, später jedoch freigegeben, im März 2009 von der Apache Software Foundation als Projekt aufgenommen [cassandra04] und ist daher frei verfügbar. Facebook und Twitter sind nur zwei Beispiele für die Verwendung von Cassandra [cassandra01]. Das System arbeitet mit verteilten Datenbeständen und bringt somit bereits einen wichtigen Aspekt der Sicherung mit sich, nämlich die Sicherheit vor Datenverlust. Zusätzlich müssen Mechanismen dafür sorgen, dass die Datenbestände konsistent gehalten werden. Gibt es nun eine Datenänderung auf einer Maschine, müssen die anderen Maschinen informiert werden. Dabei kann es auch vorkommen, dass nicht alle Maschinen laufen. Maschine A ist beispielsweise ausgefallen und kann die Daten einer bestimmten Änderung nicht empfangen. Mit Hilfe der *Hinted Handoff-Methode* [cassandra02] wird dieses Problem gelöst, indem einer laufenden Maschine B mitgeteilt wird, dass bestimmte Datenänderungen an Maschine A nicht durchgeführt werden konnten. Maschine B speichert diese Information und sorgt dafür, dass die Änderungen neuerlich an Maschine A übertragen werden, wenn diese wieder verfügbar ist.

3.5 Aktuelle Trends in der Literatur

Aktuell wird versucht, den Benutzern von Datenbanken bereits beim Anlegen der Datenbanken verschiedenste Optimierungsmaßnahmen anzubieten, welche die Leistung der Datenbank an sich erhöhen und die spätere Replikation vereinfachen sollen.

[Noyer et al 2007] beschreibt die Funktionsweise von **Tabellenpartitionierung** in PostgreSQL-Datenbanken. Es handelt sich dabei um ein Verfahren, mithilfe dessen man den Zugriff auf große Datenmengen effizient gestalten kann. Dabei wird durch die Partitionierung eine logische große Tabelle einer Datenbank in mehrere, kleinere Tabellen zerlegt. Dadurch kann die Datenbank

erhebliche Geschwindigkeitssteigerungen bei ausgeführten Abfragen erzielen. Dies ist allerdings nur möglich, wenn die Partitionierung optimal durchgeführt wurde und zwar von Personen, die detailierte Informationen über die Datenbank besitzen. So wäre es zum Beispiel vorteilhaft, selten benutzte Tabellenpartitionen auszulagern auf langsamere (und damit kostengünstigere) Speichermedien. Häufig verwendete Tabellenpartitionen würde man in Verbindung mit schnellen (und damit teureren) Speichermedien verwenden. Es gibt zwei verschiedene Arten von Partitionierung. Die Range-Partitionierung ordnet jeder Partition einen bestimmten Wertebereich zu und legt die Datensätze, je nach Referenzwert, in den entsprechenden Partitionen ab, während die List-Partitionierung explizite Werte anstelle von Wertbereichen zum Vergleich fordert. Als Ausgangspunkt wird eine Master-Tabelle benötigt, die üblicherweise keine Datensätze beinhaltet, sondern Informationen über die Kind-Tabellen [Noyer et al 2007]. Abbildung 5 zeigt eine Übersicht über Vor- und Nachteile der PostgreSQL-Partitionierung.

Vorteile	Nachteile
Partition Elimination* bei Ausführung	Literale müssen verwendet werden wie bspw. WHERE = 100
LIST, RANGE Partitioning, unbeschränkt viele Ebenen	Kein Dynamic Partition Pruning*
Subpartitioning: Unbegrenzte Partitionierungstiefen und Detailgrade	Keine automatische Datentypenkonvertierung
Partitioning-Syntax ist deklarativ. Transparent für Benutzer und Applikationen	Nur STABLE-Funktionen – die Konstanten zurückliefern - können verwendet werden
Die Child-Tabellen können separat gepflegt werden (Statistiken berechnen, Daten aufräumen, Bulk Deletes auf Partitionen)	Beträchtlicher Aufwand für das Erstellen notwendiger DDL-Statements (kann aber automatisiert werden)
Child-Tabellen/Partitionen können auf unterschiedlichen Tablespaces liegen	Kann unter Umständen die Performance von ORDER BY, GROUP BY negativ beinträchtigen

Abbildung 5: Vor- und Nachteile der PostgreSQL-Partitionierung [Kulev, Welker 2007]

In Bezug auf Replikationsvorgänge kann dieser Ansatz vor allem durch die erwähnten Geschwindigkeitsvorteile erhebliche Verbesserungen bringen, wenngleich die Einrichtung der Partitionen wiederum einen höheren Zeitaufwand erfordert.

Eine Variante, die vorrangig die Administration und weniger die Leistung einer Datenbank unterstützen soll, bieten sogenannte **Tablespaces**. Gerade in Verbindung mit Partitionierung können Tablespaces aber auch die Leistung der Datenbank positiv beeinflussen. Es handelt sich um Speichereinheiten innerhalb der Datenbank, denen mehrere Tabellen zugeordnet werden können. Dabei darf es keine Tabelle geben, die nicht irgendeinem Tablespace zugeordnet ist. [Albrecht

2001] Dadurch können die Daten auf unterschiedliche Speichermedien ausgelagert werden. Ebenso können die Zuordnungen von Tabellen zu Tablespaces schnell geändert werden, wodurch es bei Änderungen in der Nutzung bestimmter Tabellen möglich ist, diese rasch auf einen schnelleren (oder langsameren) Speicher umzulagern [Noyer et al 2007].

Eine weitere Möglichkeit, um große Datenbestände optimal verwalten zu können, bieten ‚**Materialized Views**' (deutsch: Materialisierte Sichten). Mit dieser Methode können Datenbankoperationen definiert werden, die nach Ausführung in eigenen Tabellen gespeichert werden, um sie später wiederverwenden zu können. Sie dienen dazu, Datenabfragen schneller ausführen zu können. Besonders eignet sich diese Methode für große Datenbestände, deren Änderungsrate relativ gering ist und die Daten über längere Zeiträume gleich bleiben. Als Anwendungsgebiete für materialisierte Datenbankoperationen gelten beispielsweise aufwändige Joins (= Verknüpfungen von Tabellen) und Aggregationen (= Zusammenfassung kleinerer Datengruppen zu größeren Einheiten). Es reicht jedoch nicht aus, solche Materialized Views zu definieren, man muss ebenso auf deren Wartung und spätere Verwendung Rücksicht nehmen. Als Nachteile dieser Methode kann man anführen, dass die Definition und Wartung von Materialized Views viel Zeit in Anspruch nimmt. Demgegenüber steht der klare Vorteil, dass Datenbank-Abfragen bei sinnvoller Definition der Materialized Views die Leistung der Datenbank erheblich beschleunigt werden können. [Albrecht, Fiedler 2006]

In der Literatur wird die Verwendung von Materialized Views oft auch als **Query Rewriting** bezeichnet. Genauer gesagt wird dadurch jedoch das Umschreiben einer Abfrage in alle möglichen Formen bezeichnet. Die Variante mit den geringsten Kosten wird schlussendlich ausgeführt. Kosten bedeutet in diesem Fall Ausführungszeit. Die Methode hat es zur Aufgabe, Materialized Views auf deren Verwendbarkeit zu überprüfen und weiters, ob die Verwendung dieser auch die Leistung der Datenbankabfragen steigert. Es könnte zum Beispiel sein, dass die einfache Verwendung von Indizes kostengünstiger wäre. [Halevy 2001]

4. Entwurf und Implementierung eines Datensicherungs-Programms für PostgreSQL

Um das Ziel dieser Bachelorarbeit zu erreichen, soll eine Lösung zur Sicherung von Datenbeständen gefunden werden. Da die Zieldatenbank eine Open Source Datenbank von PostgreSQL ist, wird auch die Implementierung mithilfe dieser Datenbanklösung erfolgen. Die weiteren Schritte und notwendigen Programme werden im folgenden Kapitel näher erläutert.

4.1 Anforderungsdefinition für ein Programm zur inkrementellen Datenbank-Sicherung

Für diese Arbeit wurde als Ziel definiert, eine Lösung zur inkrementellen Sicherung von PostgreSQL-Datenbanken bereitzustellen. Definitionsgemäß soll die Implementierung geänderte, gelöschte oder hinzugefügte Daten in der Datenbank erkennen und daraus zu jedem beliebigen Zeitpunkt textbasierte Dateien erstellen, in denen jeweils die Veränderungen zum Zeitpunkt der letzten Sicherung festgehalten werden. Somit kann in weiterer Folge bei einem Absturz der Datenbank (oder auch aus anderen Gründen) diese wieder hergestellt werden. Dabei ist es möglich, die Datenbank an jeden beliebigen Zeitpunkt in der Sicherungskette zurückführen. Der wichtigste Aspekt dabei ist die Datenkonsistenz, es sollen also unter keinen Umständen Daten verloren gehen. Weiters soll die Replikation den laufenden Betrieb der Datenbank nicht einschränken, also keine Geschwindigkeits- oder Qualitätsverluste verursachen. Es soll nur lesend auf die Datenbank zugegriffen werden. Dabei wird zu Beginn eine Datei mit dem Inhalt der Metadaten in einem beliebigen Verzeichnis abgelegt. Anschließend sorgt das Backup-Programm dafür, dass zum Ausführungszeitpunkt eine neue Sicherungsdatei im gleichen Verzeichnis abgelegt wird. Diese Sicherungsdatei enthält die inkrementell hinzugefügten Daten seit dem letzten Sicherungszeitpunkt. Eine spätere Wiederherstellung der Datenbank ist dann leicht realisiert, indem man die Dateien mit dem Sicherungsprogramm wiederherstellt. Dabei wird zunächst die Datei mit den Metadaten aus dem Verzeichnis eingelesen und anschließend die Sicherungsdateien bis

zum festgelegten Zeitpunkt. Die ausführliche Beschreibung dieser zwei Programme erfolgt im nächsten Abschnitt.

4.2 Programmierung und Implementierung

Unter diesem Punkt werden die einzelnen Schritte vom Entwurf bis zur fertigen Lösung beschrieben. Dabei werden auch die verwendeten Programme näher erläutert.

4.2.1 Diagramme und Beschreibung der verwendeten Programme

Im aktuellen Fall wurde eine Datenbank der Version 8.3.11 auf einem Debian-System eingerichtet. Dieses Debian-System läuft im Programm VirtualBox, einer virtuellen Maschine von Sun Microsystems. Das Mutterbetriebssystem, in dem diese virtuelle Maschine läuft, ist die neueste Generation von Microsoft, Windows 7. Der Grund für die Verwendung eines Open Source Linux Betriebssystems liegt im speziellen Fall darin, dass die nachfolgend beschriebene Replikationslösung Londiste für Windows nicht existierte. Das UML-Komponentendiagramm in Abbildung 6 zeigt, wie die einzelnen Programmteile zusammenarbeiten. Nachfolgend werden die einzelnen Komponenten näher beschrieben.

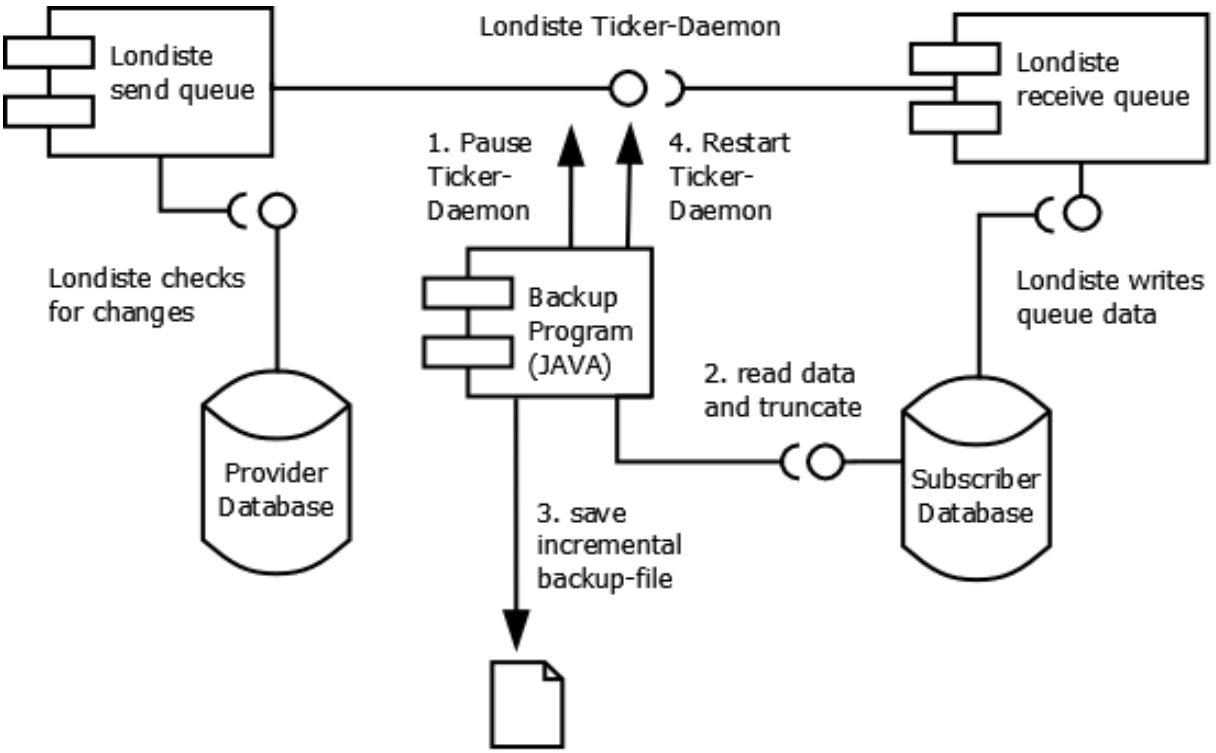

Abbildung 6: Funktionsweise der Backup-Lösung (Komponentendiagramm)

Auf der Suche nach geeigneten Replikationslösungen erwies sich das Programm Londiste am zuverlässigsten, um geänderte Einträge in der Datenbank zu identifizieren. Das Programm wurde bereits in Kapitel 3.4.1 grundsätzlich erläutert. Es entstammt dem Entwicklerteam der bekannten Telefonie-Software Skype und ist in einem Paket mit dem Namen Skytools erhältlich. Im konkreten Fall wurde die Version 2.1.9 verwendet. Londiste muss nach der Installation konfiguriert werden, damit das Programm weiß, aus welcher Datenbank gelesen werden muss (Provider) und in welche Datenbank die identifizierten Einträge dann geschrieben werden sollen (Subscriber). Unter der Annahme, dass der Provider ohnehin rund um die Uhr läuft, ist nach einmaliger erfolgreicher Konfiguration kein weiterer Schritt notwendig. Selbstverständlich ist es auch möglich, die Konfiguration zu ändern, wenn Tabellen hinzugefügt oder gelöscht werden. Sollte der Provider allerdings einmal ausfallen oder neu gestartet werden, so müssen die Londiste-Dienste jeweils auch neu gestartet werden, wenn man sie nicht an den Start der Datenbank koppelt.

Zu Testzwecken wurde eine sehr einfache Datenbank eingerichtet, welche lediglich drei Tabellen beinhaltet. Die Tabelle ‚students' besteht aus einer ID als Primärschlüssel sowie dem Vor- und

Zunamen von Studenten. In der Tabelle ‚books' wird ebenfalls eine ID als Primärschlüssel erfasst, sowie weiters der Titel des Buches und das Erscheinungsjahr. Um neben den Primärschlüsseln zumindest eine tabellenübergreifende Beschränkung (engl. Constraint) zu setzen, wurde eine dritte Tabelle ‚rent' angelegt. Diese erfasst die Entlehnungen der Studenten. Dazu benötigt es die Primärschlüssel aus den Tabellen students und books als Fremdschlüssel und ein Datumsattribut. Alle drei Attribute zusammen ergeben den Primärschlüssel für die Tabelle rent. Somit ist es möglich, dass ein Buch mehrmals vom gleichen Studenten ausgeborgt wird. Mit dem Datumsattribut wird sichergestellt, dass diese Entlehnungen nicht an ein und demselben Tag passieren bzw. dass man mehrmalige Entlehnungen des gleichen Buches durch einen Studenten eindeutig zuordnen kann. Abbildung 6 zeigt das Modell dieser Datenbank, welches mit dem frei verfügbaren Programm Dia erstellt wurde.

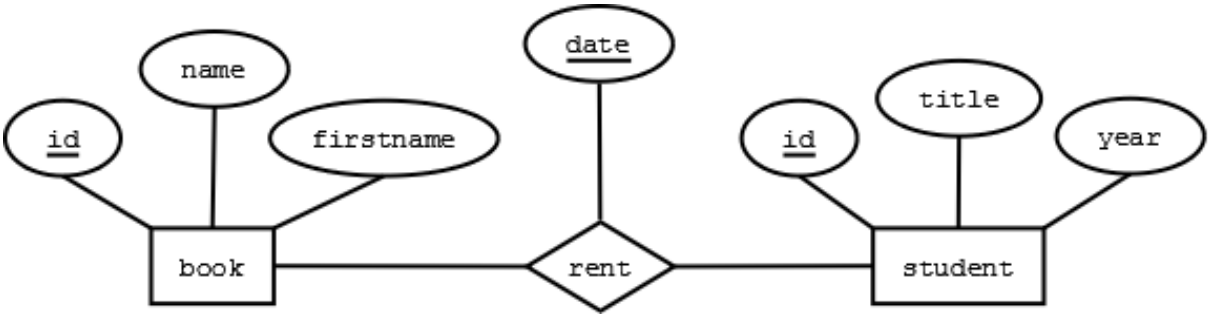

Abbildung 7: ER-Modell der Testdatenbank

Das relationale Schema der Testdatenbank sieht folgendermaßen aus:

book: <**id**, name, firstname | Σ> Σ = {"**id** ist PS"}

student: <**id**, title, year | Σ> Σ = {"**id** ist PS"}

rent: <**book_id**, **student_id**, **date** | Σ> Σ = {{"**book_id, student_id, date}** ist PS"}

Diese Datenbank wurde im Testbetrieb zwei Mal auf dem gleichen Datenbankserver, nämlich dem lokalen Rechner, angelegt. Einmal unter dem Namen ‚tprovider' (Test-Provider) und einmal unter dem Namen ‚tsubscriber' (Test-Subscriber). Nach erfolgreicher Installation und Konfiguration von Londiste wurden die neu erfassten sowie die geänderten und gelöschten Einträge ein-

wandfrei von tprovider auf tsubscriber übertragen. Weiters muss hierbei noch berücksichtigt werden, dass im Fall der Testdatenbank keine automatisch generierten Datenfelder vorhanden sind. Jedoch kann Londiste auch mit solchen Einträgen, wie zum Beispiel RANDOM (erzeugt einen Zufallswert) oder NOW() (speichert die aktuelle Systemzeit) umgehen. Aus der Provider-Datenbank wird nämlich der aktuelle Wert ausgelesen, nicht jedoch RANDOM oder NOW. Wäre dies der Fall, so würde ein Abgleich der beiden Datenbanken Unterschiede ergeben, da zufällig generierte Zahlen in zwei Datenbanken mit Sicherheit andere Werte liefern würden. Somit ist der Schritt des Auslesens der Daten inklusive deren Speicherung in eine andere Datenbank abgeschlossen.

Natürlich kann dieser Schritt auch auf zwei physisch getrennten Maschinen passieren. Dazu muss einfach die Konfiguration entsprechend geändert werden, um die Subscriber-Datenbank nicht auf dem gleichen Rechner zu suchen, sondern mithilfe der IP-Adresse und des entsprechenden Ports auf dem Zielrechner. Zudem ist es auch möglich, mehrere Subscriber zu konfigurieren, sodass die Sicherung der Daten idealerweise auf mehr als einem entfernten Rechner passieren kann. Jede Subscriber-Datenbank benötigt lediglich eine eigene Konfiguration. So kann es auch möglich sein, einen Teil der Daten auf einem Rechner zu sichern und einen anderen Teil der Daten auf einem weiteren Rechner, je nachdem wie die Beschränkungen innerhalb der Datenbank dies zulassen. Fazit: der Benutzer kann frei entscheiden, wie die Replikation der Daten ablaufen soll und mehr oder weniger sichere Konfigurationen entwickeln. Grenzen sind dabei nur die Speicher der Zielrechner bzw. bei sehr vielen Subscriber-Datenbanken und häufigen Änderungen der Daten wird die Leistung der Datenbank eingeschränkt oder auch die Netzwerkverbindung könnte überlastet werden.

Nach dem Auslesen der Daten und der Speicherung ist es jetzt das Ziel, die Daten aus der Subscriber-Datenbank auszulesen und in einer Textdatei zu speichern. Diese Textdateien sollen vom Benutzer zu jedem Zeitpunkt erstellt werden können. Dazu genügt ein Aufruf in der Kommandozeile (= Shell) mit den entsprechenden Parametern. Nach Erstellen einer solchen Textdatei mit dem kompletten Inhalt (= nur Daten) der Subscriber-Datenbank werden alle Daten gelöscht. Dabei muss darauf geachtet werden, dass nur jene Daten gelöscht werden, die auch bereits aus-

gelesen und gesichert wurden. Denn Londiste arbeitet im Hintergrund weiter und kopiert kontinuierlich alle geänderten Datensätze. Zu Beginn des kompletten Verfahrens sollte ein Dump mit den Metadaten der Provider-Datenbank erstellt werden, der in einem beliebigen Verzeichnis abgelegt wird. Wichtig: dieses Verzeichnis ist zu Testzwecken auf jenem Rechner, auf dem auch die Subscriber-Datenbank läuft. Durch Änderungen in der Konfiguration sollte man jedoch dafür sorgen, dass man verschiedene physische Maschinen einsetzt. Von der Subscriber-Datenbank erhält das Verzeichnis nach und nach die Textdateien mit den hinzugefügten Datensätzen. Bei einem Absturz der Datenbank kann somit aus dem Verzeichnis jeder beliebige Zeitpunkt der Datenbank rekonstruiert werden.

Zu erwähnen ist auch noch, dass die Subscriber-Datenbank keine Constraints enthalten darf, also beispielsweise keine Primärschlüssel. Der Grund dafür: Nach dem Auslesen der Daten aus der Subscriber-Datenbank werden diese gelöscht, Londiste schreibt kontinuierlich weiter in diese Datenbank und kann dabei Daten eintragen, die in der Provider-Datenbank alle Constraints erfüllen, aber in der Subscriber-Datenbank mit hoher Wahrscheinlichkeit Daten referenzieren würden, die dort nicht mehr vorhanden sind. Im erwähnten System *The Media Watch on Climate Change* handelt es sich um eine Datenbank, in der sich die Datensätze nach dem Eintragen nicht mehr ändern und es werden auch keine Datensätze gelöscht. Die Datenbank wird daher ständig um neue Datensätze erweitert. Dies ist ein wichtiger Punkt für die beschriebene Implementierung. Denn wenn das Programm den Inhalt aus der Subscriber-Datenbank ausliest und im Anschluss löscht, kann das Londiste-Programm auf Änderungen oder Löschungen bestimmter Datensätze nicht korrekt reagieren, da der entsprechende Datensatz in der Subscriber-Datenbank nicht mehr existiert.

4.2.2 Ablauf nach Systemstart

Nachdem das Betriebssystem hochgefahren ist, müssen verschiedene Schritte ausgeführt werden, um die inkrementelle Sicherung durchzuführen bzw. zu ermöglichen. Zuerst muss das PostgreSQL-Datenbanksystem gestartet werden. Wenn der Server läuft, muss Londiste entsprechend konfiguriert werden. Im Fall der Testdatenbank tprovider und dessen Replikationsdatenbank tsubscriber müssen folgende Shell-Befehle ausgeführt werden (siehe auch [postgresql04]). Zur einfacheren Handhabung wurden diese Befehle in einer Datei gespeichert, deren Ausführen die Befehle automatisch ausführt.

/usr/bin/pgqadm.py /home/tommy/cfg/tticker.ini install
/usr/bin/pgqadm.py /home/tommy/cfg/tticker.ini ticker -d

Wirkung:
Im Verzeichnis /usr/bin wurde das Skytools-Paket installiert. Nacheinander werden die beiden Befehle ausgeführt, dadurch wird Londiste in der Provider-Datenbank installiert und das System angewiesen, Ticks zu produzieren. Ein einzelner Tick bezeichnet hierbei eine einzelne Änderung eines Datensatzes, unabhängig davon, ob es sich um das Hinzufügen, das Löschen oder die Änderung handelt. Die Datei pgqadm.py wird ausgeführt und erhält als Parameter die Werte aus der Konfigurationsdatei tticker.ini (tticker = Kurzname für Test-Ticker), welche in diesem Fall im Verzeichnis /home/tommy/cfg abgelegt wurde. Nachfolgend wird der Inhalt dieser Datei erläutert.

```
job_name = my_test_ticker
db = dbname=tprovider
# how often to run maintenance [seconds]
maint_delay = 600
# how often to check for activity [seconds]
loop_delay = 0.1
logfile = /home/tommy/log/%(job_name)s.log
pidfile = /home/tommy/pid/%(job_name)s.pid
```

Abbildung 8: Inhalt der Ticker-Konfigurationsdatei (tticker.ini)

Es wird ein Name für diese Replikation vergeben (job_name) und der Name der Provider-Datenbank muss spezifiziert werden. Es folgen weitere Parameter, welche die Häufigkeit der Überprüfung auf geänderte Datensätze betreffen. Der job_name wird anschließend verwendet, um die log-Datei sowie die pid-Datei entsprechend zu benennen, ansonsten hat er keinerlei Bedeutung. In der log-Datei werden die Änderungen in der Provider-Datei mitgeschrieben, bei auftretenden Fehlern kann diese Datei dann nach Hinweisen zur Fehlersuche verwendet werden, da dort die Fehlermeldungen protokolliert werden. Die pid-File ist lediglich dazu da, eine spezifische Nummer für den Prozess zu speichern. Diese Nummer wird von anderen Daemon-Programmen (Programme, die im Hintergrund ablaufen) verwendet, wenn diese ebenfalls auf das laufende Skript zugreifen sollen.

/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini provider install
/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini subscriber install

Wirkung:
Jetzt wird londiste.py gestartet, das in beiden Datenbanken installiert wird. Auch dieses Skript greift auf eine Konfigurationsdatei (tp-to-ts.ini = Test-Provider-to-Test-Subscriber) zu. Der Inhalt dieser Datei:

```
[londiste]
job_name = test_transfer
provider_db = dbname=tprovider port=5432 host=127.0.0.1
subscriber_db = dbname=tsubscriber port=5432 host=127.0.0.1
# it will be used as sql ident so no dots/spaces
pgq_queue_name = londiste.replica
logfile = /var/log/londiste/%(job_name)s.log
pidfile = /var/run/londiste/%(job_name)s.pid
```

Abbildung 9: Inhalt der Londiste-Konfigurationsdatei (tp-to-ts.ini)

Auch hier wird ein job_name vergeben, der anschließend für log-Datei und pid-Datei verwendet wird. Diese log-Datei speichert Daten in Bezug auf die Verknüpfung der beiden Datenbanken. Hier würden Fehler protokolliert, wenn beispielsweise eine replizierte Tabelle nicht mehr in der Datenbank wäre, weil sie nicht mehr benötigt wird. In solchen Fällen müssen die Londiste-Skripts gestoppt werden, danach die Tabellen in beiden Datenbanksystemen angepasst werden und erst dann könnte Londiste wieder gestartet werden. Weiters werden hier Informationen zu den beiden Datenbanken gegeben. Die Port-Nummer (5432 = Standard-Port von PostgreSQL) sowie der Host (127.0.0.1 = localhost) auf dem die Datenbanksysteme laufen. In diesem Beispiel liegen beide Datenbanksysteme auf dem gleichen physischen Rechner. Bei verschiedenen physischen Maschinen müsste man hier die entsprechenden Host-Adressen festlegen. Weiters wird noch ein spezifischer Name für die pgq_queue (= Ticker-Warteschlange) festgelegt. Unter diesem Namen wird eine Tabelle in der Provider-Datenbank angelegt, welche bei hoher Belastung, also vielen Ticks innerhalb kürzester Zeit, die zu schreibenden Datensätze vor der Übertragung auf die Zieldatenbank zwischenspeichert. Die pid-Datei erfüllt den gleichen Zweck, wie eben erwähnt.

/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini replay –d

Wirkung:
Zuletzt wird mit dem Befehl replay –d die Subscriber-Datenbank angewiesen, die produzierten Ticks zu übernehmen und zu schreiben. Allerdings würden zu diesem Zeitpunkt keine Informationen geschrieben werden, da noch keine Tabellen definiert wurden, welche überhaupt in die Replikation einfließen sollen. Dabei gibt es die Möglichkeit, einzelne Tabellen hinzuzufügen. In unserem Testfall ist die Datenbank mit drei Tabellen noch relativ übersichtlich, bei größeren Datenbanken empfiehlt es sich jedoch, alle Tabellen mit dem folgenden Kommando hinzuzufügen:

/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini provider tables |xargs
/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini subscriber add

Hier wird wiederum das bekannte Skript (londiste.py) aufgerufen, das alle Tabellen aus der Provider-Datenbank ausliest und die Replikation auf die Subscriber-Datenbank veranlasst. Wenn dieser Befehl keinen Fehler aufwirft, sind die Datenbanken bereit zur Replikation und ab diesem Zeitpunkt werden alle Änderungen in der Provider-Datenbank auf die Subscriber-Datenbank übertragen.

Neben der Konfiguration von Londiste sollten noch weitere Schritte durchgeführt werden, um auch für das Wiederherstellen der Daten gerüstet zu sein. Diese Schritte werden ebenfalls mit der Londiste-Konfiguration ausgeführt. Bei einer Wiederherstellung der Datenbank müssen die Constraints vor dem Einfügen der Daten entfernt werden und danach wiederhergestellt werden. Dazu müssen diese Constraints jedoch vorher bekannt sein. Dazu wurde eine weitere Datenbank angelegt (*tdummy*), welche die Metadaten inklusive Constraints enthält, jedoch sonst keinerlei Verwendung hat. Es wurde deshalb eine eigene Datenbank angelegt, weil die Constraints der Provider-Datenbank durch die Integration von Londiste zu umfangreich wären. Dort sind nämlich Tabellen und Constraints vorhanden, die bei der Wiederherstellung nicht benötigt werden.

psql -f queries.sql -d tdummy -U tommy

Dieser Befehl wird nach der Londiste-Konfiguration ausgeführt und bewirkt, dass eine Datei namens queries.sql aus dem gleichen Verzeichnis eingelesen und an unsere Dummy-Datenbank (tdummy) mit dem Benutzernamen tommy übergeben wird. Der Inhalt dieser Datei sieht folgendermaßen aus:

```
\t
\o data/drop_constraints.sql
SELECT 'ALTER TABLE "'||nspname||'."'||relname||'" DROP CONSTRAINT
"'||conname||'";'
FROM pg_constraint
INNER JOIN pg_class ON conrelid=pg_class.oid
INNER JOIN pg_namespace ON pg_namespace.oid=pg_class.relnamespace
ORDER BY CASE WHEN contype='f' THEN 0 ELSE 1 END,contype,nspname,relname,conname;

\o data/create_constraints.sql
SELECT 'ALTER TABLE "'||nspname||'."'||relname||'" ADD CONSTRAINT "'||conname||'"
'||
pg_get_constraintdef(pg_constraint.oid)||';'
FROM pg_constraint
INNER JOIN pg_class ON conrelid=pg_class.oid
INNER JOIN pg_namespace ON pg_namespace.oid=pg_class.relnamespace
ORDER BY CASE WHEN contype='f' THEN 0 ELSE 1 END DESC,contype DESC,nspname
DESC,relname DESC,conname DESC;
\t
```

Abbildung 10: Inhalt der Datei queries.sql

Die Funktionsweise dieser Datei ist schnell erklärt. Der Befehl \t (= tuples only, also nur Datensätze) sorgt dafür, dass die Datenbank das Ergebnis einer SQL-Query (= Datenbankabfrage) ohne Metadaten ausgibt. Konkret werden die Spaltennamen sowie die Trennzeichen der Spalten nicht angezeigt. Danach wird mit dem Befehl \o (= output) definiert, dass die Ausgabe der folgenden SQL-Query in einer Datei gespeichert werden soll. Im ersten Fall wird im Unterverzeichnis data eine Datei drop-constraints.sql angelegt. Diese enthält dann genau jene Befehle, mit denen die Constraints gelöscht werden. Analog erfolgt dies auch für die zweite Datei, create-constraints.sql mit deren Hilfe die Constraints wiederhergestellt werden. Zu guter letzt mit dem Befehl \t die Ausgabe wieder umgestellt, sodass auch die Metadaten wieder angezeigt werden.

pg_dump -s -f data/00_schema.sql tdummy

Mit diesem Befehl wird schließlich noch ein einfacher Dump der Metadaten aus der Dummy-Datenbank erstellt, der ebenfalls im Verzeichnis data abgelegt wird. Mit der Option -s wird festgelegt, dass lediglich das Schema (also die Metadaten) ausgelesen und gespeichert werden sollen.

Nach diesen Schritten sind alle Vorbereitungen getroffen, die Sicherung durchzuführen. Die Provider-Datenbank wird mit neuen Datensätzen bestückt, daraufhin arbeitet Londiste und schreibt diese neuen Datensätze in die Subscriber-Datenbank. Aus dieser Subscriber-Datenbank müssen wir jetzt die Daten zu beliebigen Zeitpunkten auslesen. Dazu wurde ein Backup-Programm erstellt, dass im folgenden Abschnitt näher erläutert wird.

4.2.3 Das Backup-Programm

Die Erfordernisse wurden nun mithilfe der Programmiersprache Java implementiert. Java ist eine objektorientierte Programmiersprache und wartet durch die weite Verbreitung mit einer umfangreichen Klassenbibliothek auf. Viele Methoden können somit leicht in Programme implementiert werden ohne den Aufwand, diese selbst programmieren zu müssen. Als Unterstützung wurde NetBeans in der Version 6.8, eine sogenannte IDE (englische Abkürzung für Integrierte

Entwicklungsumgebung) verwendet. Ein derartiges Programm unterstützt den Entwickler zum einen durch kontinuierliche Überprüfung des Programmcodes. Bei Syntaxfehlern wird der Entwickler sofort gewarnt und bekommt Verbesserungsvorschläge präsentiert. Weiters hat der Entwickler die Möglichkeit, das Programm in der Laufzeitumgebung direkt auszuführen und direkt zu testen. Ebenso gibt es die Möglichkeit, ausführbare Dateien zu erzeugen, die von der Kommandozeile ausgeführt werden können. Solche Dateien nennt man Java-Archiv (Dateiendung ‚.jar'). Sie enthalten alle Informationen, die rund um den Programmcode gebraucht werden, um eine Aufgabe auszuführen. Beispielsweise muss im Fall der Verbindung zu einer PostgreSQL-Datenbank eine Treiberdatei integriert werden. Diese wird einmal in NetBeans den Bibliotheken zugeordnet und bei der automatischen Erstellung des Java-Archives eingebunden. Für das Backup-Programm wurde die aktuelle Version des Treibers verwendet, die ebenfalls als Java-Archiv (postgresql-8.4-701.jdbc4.jar) kostenlos im Internet zur Verfügung steht. Das Programm wurde so entwickelt, dass es von der Kommandozeile aufgerufen werden kann und es daher einfach ist, das Programm in Systemprozesse einbinden und automatisiert ausführen zu können. Dabei werden alle Parameter in ein Kommando gepackt und dem Programm übergeben.

Das Backup-Programm findet sich im Debian-System unter folgendem Pfad:

/home/tommy/NetBeansProjects/Backup/dist/

Nach dem Wechsel in dieses Verzeichnis kann das Backup mit folgendem Befehl durchgeführt werden (ohne Wechsel in das Verzeichnis kann natürlich auch mit dem absoluten Pfad zum Java-Archiv gearbeitet werden):

java -jar Backup.jar --host localhost --port 5432 --db tsubscriber --user tommy --pw "" --path "/home/tommy/Desktop/backup/"

Mit -jar wird der Java-Laufzeitumgebung die Java-Archiv Datei übergeben. Danach folgen die Parameter, die an das Java-Programm übergeben werden. Es folgt eine detaillierte Beschreibung zu diesen Parametern.

--host

Zuerst muss ein Server definiert werden, auf dem die Datenbank läuft. In diesem Fall handelt es sich um den lokalen Rechner, also localhost der auch mit der IP-Adresse 127.0.0.1 definiert werden könnte.

--port

Jeder Serverdienst läuft auf einem bestimmten Port, ohne Port kann man nicht auf den Dienst zugreifen. PostgreSQL läuft hier auf dem Standard-Port 5432, könnte aber auch für jeden anderen Port eingerichtet werden.

--db

Da ein Datenbankserver mehrere Datenbanken verwalten kann, muss auch die Datenbank übergeben werden, für die das Backup ausgeführt werden soll. In unserem Fall handelt es sich um die bekannt Subscriber-Datenbank tsubscriber.

--user

Nicht jeder Benutzer kann einfach Datenbanken öffnen. Bei der Einrichtung des Datenbankservers muss bereits definiert werden, welche Benutzer Zugriff auf den Datenbankserver haben sollen. In PostgreSQL gibt es immer einen Standard-Benutzer namens postgres und beliebig viele weitere Benutzer. Hier wurde ein Benutzer namens tommy eingerichtet. Dieser Benutzer verfügt über Administratorrechte und wird auch für die Sicherung verwendet.

--pw

Zu jedem Benutzernamen sollte auch ein Passwort vergeben werden. Zu Testzwecken wurde im aktuellen Fall auf ein Passwort verzichtet, daher ist dieser Parameter leer (””).

--path

Als letzter Parameter muss der Pfad übergeben werden, der die erzeugten Sicherungsdateien enthalten soll. Wir verwenden hier den gleichen Pfad, der bereits die Dateien zum Löschen und Erzeugen der Constraints sowie die Metadaten der Datenbank enthält.

Nach Übergabe der Parameter werden die Daten an die entsprechende Datenbank gesendet. Bei erfolgreicher Verbindung wird der im Hintergrund laufende Londiste-Dämon mit dem Kommando *„/usr/bin/londiste.py /home/tommy/cfg/tp-to-ts.ini replay -s“* gestoppt. Diesen Aufruf kennen wir von der Londiste-Konfiguration. Als Unterschied erkennen wir den Parameter -s, der für den Ticker-Dämon ‚stop‘ bedeutet, während -d für ‚Dämon‘ steht, womit der Dämon nach der Sicherung wieder gestartet werden kann. Alle Änderungen werden ab dem Zeitpunkt des Stoppens in einer Warteschlangen-Tabelle, einer sogenannten Queue, der Provider-Datenbank zwischengespeichert und nicht mehr an die Subscriber-Datenbank übertragen. Somit wird sichergestellt, dass die Datenbank während der Sicherung nicht geändert wird. Danach werden die Tabellen einzeln ausgelesen. Nacheinander werden die Tabellen durchgearbeitet und mit der SQL-Query „select * from TABELLE“ werden alle Daten aus der entsprechenden Tabelle ausgelesen. Da zu Beginn die Metadaten ausgelesen werden, ist es unerheblich, wie viele Tabellen eine Datenbank enthält und auch die Anzahl der Spalten ist nicht vordefiniert, es werden alle Tabellen und alle Spalten ausgelesen. Das Ergebnis wird in einer Variablen ‚fileText‘ gespeichert, die Schritt für Schritt erweitert wird. Als Syntax wurde zu Beginn das einfache INSERT-Kommando verwendet.

Dieses hat jedoch den Nachteil, dass Hochkommas nur durch Escaping verwendbar sind. Daher wurde schließlich die COPY-Syntax verwendet. Zu den Vorteilen dieser Syntax gehören deren bessere Performance und eben die Möglichkeit, Hochkommas ohne Escaping zu verwenden.

Beispiel:

```
INSERT into book (id, title, year) VALUES (1, 'Tommy\'s Book', '2004');
INSERT into book (id, title, year) VALUES (2, 'Tcl Tk Handbook', '2005');
```

Nach Tommy ist ein Hochkomma, welches zum Inhalt der Spalte gehört, von PostgreSQL jedoch als Ende des Spalteneintrages verarbeitet werden würde. Daher wird dieses Zeichen mit einem Backslash-Symbol explizit als Teil des Spaltenwertes deklariert.

Das gleiche Kommando in COPY-Syntax:
COPY book (id, title, year) FROM stdin;
1 Tommy's Book 2004
2 Tcl Tk Handbook 2005

COPY benötigt keine Hochkommas bei der Definition der Spaltenwerte, dagegen wird jedoch zwischen den Werten ein Tabulator verwendet, um den Beginn und das Ende einer Spalte zu definieren. Somit werden auch Hochkommas korrekt interpretiert.

Parallel zur Variable ‚fileText' wird eine Variable ‚sqlTruncate' mit der Syntax zur Löschung der Datensätze geschrieben. Diese ist rasch erstellt, da man hier lediglich die Tabellennamen benötigt. Wenn die Variable vollständig geschrieben ist, lautet sie für die verwendete Testdatenbank

Truncate book, rent, student;

und sorgt dafür, dass alle Daten in den Tabellen freigegeben werden. Somit werden sie für Datenbankabfragen nicht mehr berücksichtigt und damit steigt die Performance der Datenbank. Nach dem erfolgreichen Auslesen der Daten wird der Londiste-Dämon wieder fortgesetzt und überträgt alle zwischengespeicherten Daten aus der Queue sowie alle weiteren Änderungen, die ab diesem Zeitpunkt passieren, wieder an die Subscriber-Datenbank.

Da nun alle Daten gesammelt sind und die Datensätze der Subscriber-Datenbank entfernt wurden, wird der Inhalt von fileText in eine Datei geschrieben. Hier kommt der Pfad-Parameter ins Spiel, den wir über die Kommandozeile übergeben haben. Dieser wird erweitert auf das Unterverzeichnis data, in dem die Sicherungsdateien abgelegt werden sollen. Zuerst muss jedoch noch kontrolliert werden, welche Daten dort schon vorhanden sind. Dazu wird überprüft, ob es eine

Datei namens counter.txt gibt. Diese dient dazu, den Sicherungsdateien Nummern zu vergeben. Existiert die Datei nicht, so wird sie angelegt und mit dem Wert 1 beschrieben. Die Sicherungsdatei bekommt den Namen ‚1.sql', wird mit dem fileText beschrieben und abgespeichert. Ist counter.txt bereits vorhanden, so wird die aktuelle Zahl ausgelesen, um eins erhöht und der Sicherungsdatei als Name übergeben. Danach wird counter.txt mit der neuen Zahl als Inhalt wieder abgespeichert. Somit wächst das Verzeichnis mit jeder Sicherung um eine weitere Datei, mit aufsteigenden Dateinamen.

4.2.4 Das Wiederherstellungs-Programm

Bei Problemen mit der Datenbank oder einem Absturz des Servers ist es wichtig, die Daten rasch wiederherstellen zu können. Dazu wurde ebenfalls ein Programm geschrieben, welches diese Aufgabe rasch erledigen kann. Das Programm findet sich unter folgendem Pfad:

/home/tommy/NetBeansProjects/DataInsert/dist/

Die ausführbare Datei lautet DataInsert.jar. Mit folgendem Kommando kann diese Datei ausgeführt werden:

java -jar DataInsert.jar --host localhost --port 5432 --db trestore --user tommy --pw "" --path "/home/tommy/Desktop/backup/" --target 5

Die Parameter kennen wir bereits vom Backup-Programm. Sie sind wieder notwendig, um die Verbindung zur Datenbank herzustellen und das Verzeichnis zu definieren, in dem die Sicherungsdateien abgelegt sind. Als zusätzlichen Parameter benötigen wir aber noch folgenden:

--target
Die Zahl 5 steht in diesem Beispiel dafür, dass die Sicherung bis zur Datei 5.sql durchgeführt werden soll.

Nach dem Aufruf und der Verbindung zur Datenbank wird zuerst die Datei 00_schema.sql eingelesen, um die Metainformationen der Datenbank herzustellen. Das heißt es kann im Prinzip jede beliebige Datenbank neu erstellt werden, ohne jegliche Informationen über die Datenbank, die wiederhergestellt werden soll. Nach dem Einlesen der Schema-Datei werden die Constraint-Dateien benötigt, die mit der Londiste-Konfiguration erstellt wurden und ebenfalls im Verzeichnis data abgelegt wurden. Zuerst werden die Constraints durch das Einlesen der Datei drop-constraints.sql gelöscht, da die Einfüge-Reihenfolge der Tabellen nicht festgelegt werden kann. Somit kann es vorkommen, dass in einer Tabelle Daten eingefügt werden, deren Referenzen noch nicht existieren. In unserer Testdatenbank wäre dies der Fall, wenn ein Buch an einen Studenten verliehen wird, der jedoch in der Studenten-Datenbank noch nicht existiert. Dies kommt dadurch zustande, dass die Reihenfolge der Tabellen in der Sicherungsdatei alphabetisch erfolgt und die Tabelle rent vor der Tabelle student verarbeitet wird. Also werden die Constraints gelöscht um danach, beginnend mit 1.sql, bis zur Zieldatei 5.sql alle Dateien mit den gesicherten Datensätzen einzulesen. Abschließend müssen die Constraints mithilfe der Datei create-constraints.sql wiederhergestellt werden. Am Ende der Laufzeit ist der gleiche Datenstand vorhanden, wie zum Zeitpunkt des fünften Backups, inklusive aller Constraints.

4.3 Test

Die Testphase ist besonders bei Software-Projekten eine der wichtigsten Phasen überhaupt. Nichts ist nerviger als ein Programm, das nicht ordnungsgemäß funktioniert.

Da das Backup-Programm Datensätze aus der Provider-Datenbank in Textdateien speichert und das Wiederherstellungsprogramm diese Datensätze in eine Wiederherstellungs-Datenbank (trestore) einfügt, besteht der einfachste Test darin, die Anzahl der Datensätze in der Provider-Datenbank mit jener Anzahl in der Wiederherstellungs-Datenbank zu vergleichen. Dazu wurde ein weiteres kleines Progamm in Java geschrieben, welches die Provider-Datenbank mit Werten beschreibt. Dabei wird so vorgegangen, dass eine Variable x mit dem Wert 1 am Beginn einer Schleife existiert. Ebenso wird eine weitere Variable y festgelegt, die das Ende der Schleife bedeutet. In jedem Schleifendurchlauf wird jetzt je ein Datensatz für die Tabellen aus der Provider-

Datenbank erfasst. Dazu wird jedoch an jeden Eintrag die Variable x angehängt und diese nach dem Schleifendurchlauf um den Wert 1 erhöht. Das dient einerseits dazu, dass keine Primärschlüssel doppelt vorkommen können und andererseits dazu, die Einträge auf einen Blick identifizieren zu können. Es folgt ein Beispiel zur Verdeutlichung.

Im ersten Schleifendurchlauf werden folgende Werte geschrieben:
INSERT into book (id, title, year) values (1, 'Title_1', '1')
INSERT into student (id, name, firstname) values (1, 'pa_1', 'th_1')
INSERT into rent (book_id, student_id, date) values (1, 1, now())

Die Spalte year aus der Tabelle book wurde lediglich als Text definiert, daher wird kein Fehler aufgeworfen, wenn nur die Zahl 1 als Wert geschrieben wird. Die Spalte date in der Tabelle rent ist als Datum definiert und wird einfach mit dem aktuellen Datum beschrieben. Diese Werte steigen nun mit jedem Schleifendurchlauf. Erreicht die Variable x den Wert der Variable y, so wird die Schleife und das Programm beendet. Lautet die Variable y zum Beispiel 1.500, so müssen in jeder Tabelle am Ende des Programmes 1.500 Datensätze vorhanden sein.

Im letzten Schleifendurchlauf werden somit folgende SQL-Queries abgesetzt:
INSERT into book (id, title, year) values (1500, 'Title_1500', '1500')
INSERT into student (id, name, firstname) values (1500, 'pa_1500', 'th_1500')
INSERT into rent (book_id, student_id, date) values (1500, 1500, now())

Während dieses Programm arbeitet und die Provider-Datenbank mit Datensätzen füllt, überträgt Londiste im Hintergrund die Daten auf die Subscriber-Datenbank. Parallel dazu wird in unregelmäßigen Abständen das Backup-Programm ausgeführt, das Londiste vorübergehend stoppt, die Daten aus der Subscriber-Datenbank holt und die Sicherungsdatei abspeichert sowie die Datensätze löscht. Am Ende des Testprozesses stehen in der Provider-Datenbank 1.500 Datensätze in jeder der drei Tabellen zur Verfügung. Die Wiederherstellungs-Datenbank rekonstruieren wir nun mit Hilfe unseres Wiederherstellungs-Programms und kontrollieren dann, ob die gleiche

Anzahl an Datensätzen existiert. Dabei werden auch stichprobenartig Zustände getestet, die zwischen der ersten und der letzten Sicherungsdatei liegen.

Der beschriebene Testlauf wurde mehrmals mit unterschiedlichen Werten durchgeführt. Drei dieser Testläufe werden nachfolgend näher erläutert. Zu beachten ist dabei, dass diese Testläufe ebenfalls in der virtuellen Maschine Debian durchgeführt wurden und dieses System lediglich 768 MB Arbeitsspeicher zur Verfügung hatte. Als Prozessor des Mutterrechners stand ein Dual-Core Prozessor von Intel mit einer Leistung von 1,66 GHz je Kern zur Verfügung. Dies gilt es bei den nachfolgend angegebenen Zeiten zu berücksichtigen.

4.3.1 Erster Testlauf

Dieser Testlauf soll zeigen, ob bei relativ wenig Datensätzen und relativ großem Intervall zwischen dem Schreiben der einzelnen Datensätze die Daten verlustfrei gespeichert und wiederhergestellt werden können.

Vorgaben:
Die Schleife sollte erst nach einer Pause von einer Sekunde (= 1000 Millisekunden) wiederholt werden, die Zielvariable y wurde auf 1.500 gesetzt. Das bedeutet, dass die maximale Wartezeit ungefähr 1.500 Sekunden, also rund 25 Minuten beträgt. In der Ausführungszeit wurde manuell in unregelmäßigen Abständen das Backup ausgeführt und somit bis zum Ende der Laufzeit 19 Sicherungsdateien erstellt. Diese wurden anschließend an das Wiederherstellungsprogramm übergeben und die Datenbank damit rekonstruiert.

Ergebnis:
Nach erfolgreicher Rekonstruktion der Datenbank waren in allen drei Tabellen wie gewünscht 1.500 Datensätze vorhanden. Dies wurde mit der einfachen SQL-Query „SELECT count(*) from TABELLE“ manuell kontrolliert. Zusätzlich wurde stichprobenartig überprüft, ob auch dazwischenliegende Zustände konsistent wiederhergestellt werden konnten. Beim Einlesen bis zur Datei 8.sql (Zustand 8) ergab die Zählung der Zeilen je 347 in allen drei Tabellen, im Zustand 13

ergab die Zählung je 904 Datensätze und im Zustand 18 wurden je 1498 Datensätze gezählt. Die Ergebnisse sprechen also für die korrekte Funktionsweise des Backup-Programmes.

Man konnte feststellen, dass diese Aufgabe das Sicherungssystem vor keine großen Herausforderungen gestellt hat. Mit einer gesamten Ausführungszeit beim Einfügen der Daten von 25 Minuten und 51 Sekunden kann man beim Abzug der Wartezeit von 25 Minuten feststellen, dass die eigentliche Einfüge-Arbeit lediglich eine Minute der gesamten Zeit benötigte. Die Ausführung des Backup-Programms brachte keinerlei Wartezeiten und funktionierte innerhalb weniger Sekunden. Daher musste ein weiterer Testlauf gestartet werden, der das System und den Datenbankserver etwas mehr forderte.

4.3.2 Zweiter Testlauf

Nach dem erfreulichen Ergebnis des ersten Testlaufes wurden die Vorgaben für den zweiten Testlauf etwas verschärft. Es sollte getestet werden, ob bei kürzerem Intervall und mehr Datensätzen ein ebenso erfreuliches Ergebnis wie beim ersten Testlauf zustande kommen würde.

Vorgaben:
In diesem Testlauf wurde die Wartezeit zwischen den Schleifendurchläufen auf eine Zehntelsekunde (= 100 Millisekunden) reduziert und die Zielvariable y auf 3.500 erhöht. Die maximale Wartezeit beträgt somit etwa 350 Sekunden (ca. 6 Minuten) und die Anzahl der Datensätze pro Tabelle erhöht sich auf 3.500.

Ergebnis:
Beim Ausführen des Backup-Programms gab es wiederum kaum Wartezeiten. Insgesamt wurden durch die kürzere Laufzeit nur noch sieben Sicherungsdateien angelegt. Durch die geringe Anzahl konnte der Konsistenztest für alle Zustände durchgeführt werden. Dieser Konsistenztest brachte auch für alle Zustände die korrekten Ergebnisse.

Insgesamt dauerte der zweite Testlauf acht Minuten und 51 Sekunden, daraus konnte man schließen, dass neben der Wartezeit von ca. sechs Minuten etwa drei Minuten für die Verarbeitung der Daten aufgewendet werden mussten.

4.3.3 Dritter Testlauf

Nachdem auch der zweite Testlauf keine Probleme aufgezeigt hatte, sollte der dritte Testlauf eine noch härtere Belastungsprobe für das System darstellen. Das Intervall des Schleifendurchlaufes wurde weiter verkürzt und die Anzahl der Datensätze fast verdreifacht.

Vorgaben:
Die Wartezeit zwischen den einzelnen Schleifen wurde auf 10 Millisekunden verkürzt und die Zielvariable mit 9.999 festgelegt. Die Wartezeit beim Eintragen der Daten sollte also nur etwa 100 Sekunden betragen.

Ergebnis:

Die erste Sicherung wurde kurz nach dem Start des Einfüge-Programms gestartet. Zu diesem Zeitpunkt waren bereits 455 Datensätze geschrieben. Nach Fertigstellung der ersten Sicherung wurde unmittelbar daran die zweite Sicherung gestartet. Durch die Dauer der ersten Sicherung war das Einfüge-Programm bereits bei Datensatz 1.926 angekommen. Das bedeutet, dass für die zweite Sicherung 1.471 Datensätze ausgelesen werden mussten und dies wiederum mehr Zeit beanspruchte, als die erste Sicherung für 455 Datensätze. Die dritte Sicherung beinhaltete bereits die Datensätze 1.472 bis 7.568, musste also 5.642 Datensätze auslesen. Daraus kann man erkennen, dass jedes neuerliche Ausführen des Backup-Programms die Zeit für die Sicherung weiter erhöhte und bei noch mehr als 9.999 Datensätzen wohl Probleme aufwerfen könnte. Schließlich muss auch Londiste nach Neustart des Tickers zuerst die Daten aus der Queue übernehmen und benötigt dafür erhebliche Rechnerleistungen. Hier empfiehlt es sich, die Konfigurationsdatei (tp-to-ts.ini) mit folgender Zeile zu ergänzen:

pgq_lazy_fetch = 500

Das sorgt dafür, dass Londiste beim Übernehmen der Einträge aus der Queue nur jeweils 500 Datensätze berücksichtigt und erst nach erfolgreicher Abarbeitung werden die nächsten 500 Datensätze übernommen usw. In jedem Fall entsteht nach dem Neustart des Londiste-Dämons eine Art Wartezeit, in der noch keine neuerlichen Sicherungen ausgeführt werden sollten, da Londiste noch mit dem Schreiben der Queue-Einträge beschäftigt ist.

Die Ausführungszeit der Dateneinträge betrug fünf Minuten und 37 Sekunden. Nach Abzug der Wartezeit bleiben etwa viereinhalb Minuten, die das Programm damit beschäftigt war, die Daten in die Datenbank einzutragen. Die Sicherung der inkrementellen Backups dauerte mit etwa zehn Minuten fast doppelt so lange. Die dritte Sicherung war noch im Gang, als die Daten bereits fertig eingetragen waren. Nach der dritten Sicherung, als das Einfüge-Programm längst fertig war, musste jedoch noch eine Sicherung ausgeführt werden, da bis zu diesem Zeitpunkt nur die Einträge bis 7.568 berücksichtigt wurden.

In Summe entstanden also vier Sicherungsdateien. Auch in diesem Fall konnten alle Zustände auf deren Konsistenz überprüft werden und diese Prüfung war in allen Zuständen erfolgreich. Somit konnte die korrekte Zusammenarbeit zwischen dem Sicherungsprogramm und Londiste bestätigt werden.

5. Zusammenfassung und Ausblick

In der heutigen Zeit setzen viele Unternehmen in irgendeiner Form Datenbanksysteme ein. Diese Datenbanksysteme sind in einem stetigen Wachstum und können riesige Datenmengen beherbergen. Dadurch wurden die Anforderungen an Sicherungsprogramme im Laufe der Zeit immer höher. Diese Programme sollen zum einen ressourcenschonend arbeiten, also die Leistung des Computers nicht merklich einschränken und im Hintergrund ihre Arbeit verrichten, zum anderen sollen sie jedoch so zuverlässig sein, dass die gesicherten Daten mit den Originaldaten übereinstimmen und eine Wiederherstellung der Daten in möglichst kurzer Zeit funktioniert. Diese Anforderungen wurden von zahlreichen Entwicklern aufgegriffen und deren Umsetzung auf unterschiedlichste Arten versucht. Im Rahmen dieser Arbeit wurde ein solches Programm namens Londiste verwendet. Dieses Programm identifiziert Änderungen in einer Quelldatenbank und speichert diese synchron in eine Zieldatenbank. Parallel zu diesem Programm wurde, basierend auf der Programmiersprache Java, eine Lösung entwickelt, diese Daten aus der Zieldatenbank weiterzuverarbeiten. Konkret wurden die Daten aus der Zieldatenbank auf Befehl ausgelesen, das Ergebnis in eine Textdatei gespeichert und im Anschluss daran wurden die ausgelesenen Datensätze aus der Zieldatenbank entfernt. Damit wurden die Sicherungsdateien möglichst klein gehalten, ebenso wie die Zieldatenbank, da diese nach jeder Sicherung von den Daten befreit wurde. Zusätzlich wurde ein weiteres Programm entwickelt, das diese gesicherten Dateien im Zusammenhang mit den Metadaten der Datenbank zur Rekonstruktion der Datenbank verwendet. Diese Programme wurden mit einer kleinen Testdatenbank auf deren korrekte Funktion überprüft.

Beim Testen dieser Sicherungslösung wurde festgestellt, dass Londiste mit kleinen Datenbeständen sehr gut umgehen kann. Auch das Sicherungsprogramm erfüllte seinen Zweck. Jedoch stieß das ganze Projekt etwas an seine Grenzen, als die Datenbank in sehr kurzen Intervallen mit Datensätzen gefüllt werden sollte. Die Testläufe ergaben zufriedenstellende Ergebnisse in Bezug auf die eventuelle Konsistenz der Daten, da die zufällige Eintragung von Daten zu einem gewissen Zeitpunkt gestoppt wurde und somit das Sicherungsprogramm mit der letzten generierten Datei alle relevanten Daten sichern konnte. Für einen ständig laufenden Prozess würde die stei-

gende Laufzeit der Sicherungen wohl in der einen oder anderen Form Probleme aufwerfen, je nach Sicherungsintervallen und Datenmengen.

Diese Sicherungslösung basiert auf einer Mischung aus existierender Software sowie selbst geschriebener Software in einer anderen Programmiersprache. Wünschenswert wäre eine Lösung, welche in einer Programmiersprache alle Anforderungen aus einer Hand bieten kann.

A. Literaturverzeichnis

A.1 Literatur

[Abramson et al 2004]: Abramson, Ian und Abbey, Michael und Corey, Michael: Oracle Database 10g für Einsteiger, Seiten 171ff; Carl Hanser Verlag 2004.

[Albrecht 2001]: Albrecht, Jens: „Business Intelligence mit Oracle9i“, Datenbank-Spektrum, 1(1), September 2001, Seiten 36-43.

[Albrecht, Fiedler 2006]: Albrecht, Jens und Fiedler, Marc: „Datenbank-Tuning – einige Aspekte am Beispiel von Oracle 10g“, Datenbank-Spektrum, 6(16), Februar 2006, Seiten 26-33.

[Halevy 2001]: Halevy, Alon. „Answering queries using views: A survey“, VLDB Journal, 10(4), Jahr 2001, Seiten 270-294.

[Hartwig 2001]: Hartwig, Jens: „PostgreSQL – professionell und praxisnah“, Seiten 293-295; Addison-Wesley-Verlag 2001.

[Hubmann et al 2009]: Hubmann-Haidvogel, Alexander und Scharl, Arno und Weichselbraun, Albert: „Multiple Coordinated Views for Searching and Navigating Web Content Repositories“, Information Sciences, 179(12), Jahr 2009, Seiten 1813-1821.

[Kudraß 2007]: Kudraß, Thomas: „Taschenbuch Datenbanken”, Fachbuchverlag Leipzig, Carl Hanser Verlag München 2007.

[Kulev, Welker 2007]: Kulev, Milen und Welker, Peter: „Data Warehouses & Open-Source-Datenbanken“, Datenbank-Spektrum, 8/2007, Seiten 25-36.

[Noyer et al 2007]: Noyer,Ulf und Ruschinzik, Martin und Rataj, Jürgen: „Tabellenpartitionierung in PostgreSQL - Teile und herrsche“, iX – Magazin für professionelle Informationstechnik, 4/2007, Seiten 141-143.

A.2 Webseiten

[cassandra01]: The Apache Cassandra Project. http://cassandra.apache.org/ – letzter Zugriff: 13.10.2010.

[cassandra02]: HintedHandoff – Cassandra Wiki. http://wiki.apache.org/cassandra/HintedHandoff – letzter Zugriff: 13.10.2010.

[cassandra03]: ReadRepair – Cassandra Wiki. http://wiki.apache.org/cassandra/ReadRepair – letzter Zugriff: 13.10.2010.

[cassandra04]: Cassandra is an Apache top level project. http://www.mail-archive.com/cassandra-dev@incubator.apache.org/msg01518.html – letzter Zugriff: 16.10.2010.

[dbresource01]: dbresource.de. http://www.dbresource.de/Default.aspx?tabid=198 – letzter Zugriff: 29.7.2010.

[feyrer01]: Vergleich der Möglichkeiten von Backup und Recovery zwischen MySQL und PostgreSQL. http://www.feyrer.de/SA/vortraege/ss2008-gomez-slides.pdf - letzter Zugriff: 1.5.2010.

[freistil-consulting01]: Freistil-Consulting – Viel IT mit wenig Aufwand. http://www.freistil-consulting.de/lvm-snapshots-f-r-konsistente-datensicherung – letzter Zugriff: 18.3.2010.

[ibm01]: IBM DB2 9. http://www-01.ibm.com/software/de/data/db2ims/db2ud.html – letzter Zugriff: 29.6.2010.

[ibm02]: IBM Redbooks. http://www.redbooks.ibm.com/redbooks/pdfs/sg246828.pdf: A Practical Guide to DB2 UDB Data Replication V8, Kapitel 1, Seiten 9-18; – letzter Zugriff: 12.4.2010.

[ibm03]: DB2 Database für Linux, UNIX und Windows. http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004634.htm – letzter Zugriff: 12.6.2010.

[mysql01]: MySQL::Warum MySQL? http://www.mysql.de/why-mysql/ – letzter Zugriff: 16.10.2010.

[neogrid01]: Neogrid EDV Lexikon. http://www.neogrid.de/pc_lexikon.php?Feld=Write-Ahead-Logging&PHPSESSID=i86q0j3o6i1hres3cp3ap5a3h5 – letzter Zugriff: 29.7.2010.

[nosql01]: NoSQL Databases. http://nosql-databases.org/ – letzter Zugriff: 16.10.2010.

[oracle01]: Produktliste von Oracle.com. http://www.oracle.com/products/product_list.html – letzter Zugriff: 16.10.2010.

[phpbb01]: MySQL Backup Knowledge Base. http://www.phpbb.de/kb/mysqlbackup - letzter Zugriff: 20.7.2010.

[postgresql 01]: PostgreSQL: Das Offizielle Handbuch. http://www.postgresql.org/files/documentation/books/pghandbuch/html/wal.html – letzter Zugriff: 8.4.2010.

[postgresql02]: PostgreSQL: Das Offizielle Handbuch. http://www.postgresql.org/files/documentation/books/pghandbuch/html/wal-benefits-later.html - letzter Zugriff: 10.5.2010.

[postgresql 03]: PostgreSQL: Das Offizielle Handbuch. http://www.postgresql.org/files/documentation/books/pghandbuch/html/history.html – letzter Zugriff: 30.6.2010.

[postgresql04]: Londiste Tutorial, PosgreSQL Wiki. http://wiki.postgresql.org/wiki/Londiste_Tutorial - letzter Zugriff: 30.6.2010.

[postgresql05]: Skytools: PgQ, Queues and applications. http://www.pgcon.org/2009/schedule/attachments/91_pgq.pdf - letzter Zugriff: 14.7.2010.

[tu-chemnitz01]: Datenreplikation am PostgreSQL. http://archiv.tu-chemnitz.de/pub/2005/0053/data/html/node52.html – letzter Zugriff: 30.7.2010.

[uni-magdeburg01]: Softwaretechnik für verteilte Systeme – Vorlesungsskript. http://www-ivs.cs.uni-magdeburg.de/~dumke/STV/STVKons.html – letzter Zugriff: 13.10.2010.

B Abbildungen