WIRTSCHAFTSUNIVERSITÄT WIEN
BAKKALAUREATSARBEIT

Titel der Bakkalaureatsarbeit:
Optimierung von Datenbankzugriffen und Volltext Indexierung von umfangreichen Datenbankapplikationen

Englischer Titel der Bakkalaureatsarbeit:
Optimizing Database Access and Full Text Indexing for Large Scale Database Applications

VerfasserIn:	Michael S. Feurstein
Matrikel-Nr.:	0450743
Studienrichtung:	Wirtschaftsinformatik
Kurs:	0996 IT-Praktikum mit Bakkalaureatsarbeit
Textsprache	Deutsch
BetreuerIn:	Dr. Dipl.-Ing. Mag., Albert Weichselbraun

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

dass ich die Ausarbeitung zu dem obigen Thema 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 Betreuer beurteilten Arbeit übereinstimmt.

_____ _____

Datum Unterschrift

Abstrakt

Aktuelle Forschungsprojekte des Instituts für Informationswirtschaft wie zum Beispiel AVALON, IDIOM, RAVEN sowie der Election 2008 Monitor benötigen exakte Sampledaten aus zahlreichen datenintensiven Quellen. Dabei spielen Daten wie Bilder, Mikroformate und Kommentare eine entscheidende Rolle im Hinblick auf die Performance der zugrunde liegenden Datenbank. Ziel dieser Arbeit ist es Datenbank Testcases zu entwerfen und im Weiteren zu analysieren wie Veränderungen am Design der Datenbank, ihrer Konfiguration und den Zugriffsmethoden sich auf die Performance der Datenbank auswirken.

Abstract

Many current research projects, including AVALON, IDIOM, RAVEN, and the Election 2008 Monitor depend on accurate sampling of relevant media sites. Richer media data like pictures, microformats, and annotations put a serious strain on the database's performance. The goal of this work is to design database test cases and to evaluate how changes to the database's design, configuration, and database access mechanisms influence the database's performance.

Key Words
Database tuning, Performance, Benchmark, PostgreSQL

Inhaltsverzeichnis

1 Einleitung

Diese Bakkalaureatsarbeit beschäftigt sich mit dem Problem von Engpässen datenintensiver Abfragen auf Datenbanken. Das Institut für Informationswirtschaft an der Wirtschaftsuniversität Wien ist an mehreren Projekten wie zum Beispiel AVALON, IDIOM und RAVEN beteiligt. Forschungsinhalt dieser Projekte ist im Allgemeinen die Suche, Sammlung und Analyse von Daten aus dem Internet mit einem Schwerpunkt auf die Bereiche Semantic Web, Knowledge Management und Web Analyse. Durch den datenintensiven Charakter dieser Forschungsprojekte ist die Datenbank stets ein Punkt an dem alle Daten zusammentreffen und somit Leistungsengpässe bei komplexen Abfragen auftreten. Die Optimierung der zugrunde liegenden Datenbank und ihrer Anfragen ist daher von enormer Wichtigkeit.

Aus diesem Grund benötigt das Institut für Informationswirtschaft ein Tool zur Analyse von bestimmten Use Cases. Diese sollen repräsentativ zu aktuellen Abfragen und darüber hinaus erweiterbar sein. Ziel ist es anhand des Tools Leistungsengpässe zu identifizieren, um Rückschlüsse ableiten zu können wie sich beispielsweise Veränderungen an den Einstellungen der Datenbank auf die Leistung auswirken. Als Datenbank wird die relationale Open-Source Datenbank PostgreSQL verwendet mit einem speziellen Fokus auf das Volltextsuchmodul *"tsearch2"*.

Die Arbeit ist in drei Teile gegliedert. Der erste Teil (Kapitel 2) gibt einen Überblick über Vorgehensweisen und Prinzipien zur Messung von Leistung von Datenbanken. Im zweiten Abschnitt (Kapitel 3) sollen Einstellungsmöglichkeiten und Optionen zur Optimierung von einer PostgreSQL Datenbank erörtert werden. Der letzte Teil (Kapitel 4) dokumentiert die Implementation und Verwendung des erstellten Tools.

2 Datenbank Performance Messung

Dieses Kapitel beschäftigt sich mit den Grundlagen der Messung der Leistung von Abfragen auf einer Datenbank. Es soll ein fundierter Überblick über entwickelte Vorgehensweisen und Prinzipien im Bereich der Leistungsmessung gegeben werden. Grundlage für diesen Abschnitt der Arbeit stellt das Kapitel *"Database Performance Measurement"* [O'N97] verfasst von Patrick O'Neil aus dem *"CRC [1] Handbook for Computer Science and Engineering"* dar.

2.1 Grundlagen

Wenn es darum geht, Leistung in Zahlen auszudrücken, will man repräsentative Ergebnisse erzielen. Genauso wie man Elektrizität in kilowatt misst oder die Leistung von Motoren in Pferdestärken angibt, benötigt man bei der Messung der Leistung einer Datenbank im Hinblick auf Transaktionen aussagekräftige Werte, um entscheidende Schlussfolgerungen treffen zu können. Solche Leistungsmessungen werden im Allgemeinen mit dem Begriff Benchmarks in Verbindung gebracht [O'N97]. Ein Benchmark ist ein Test, der die Performance eines Systems während der Ausführung von genau definierten Arbeitsschritten misst. Benchmarks sollen wiederholbare, vorhersagbare und messbare Ergebnisse liefern, welche dazu verwendet werden können, um verschiedene Aspekte im Bereich von Informationssystemen direkt zu vergleichen. Diese Eigenschaft kann auf Datenbanken übertragen werden und bietet somit die Möglichkeit, gezielt Vergleiche zwischen

[1] Chemical Rubber Company, CRC Press Taylor and Francis Group

verschiedenen Konfigurationen aufzustellen. Solche Vergleiche können sich auf Konfigurationen in der Datenbank beziehen oder auf Eigenschaften von bestimmten Use Cases, die auf mögliche Engpässe abzielen. Unter Datenbank Benchmarks versteht man somit eine Kombination aus der Simulation von Datenbank Verbindungen und der Ausführung von Datenbank Transaktionen [ABF+06].

Bei Benchmarks ist im Allgemeinen zwischen zwei Hauptkategorien zu unterscheiden. Es gibt den anwendungsspezifischen Benchmark (*"custom benchmark "*) und generischen Benchmark (*"generic benchmark"*) sowie den hybriden Benchmark (*"hybrid benchmark"*), welcher auf einem generischen Benchmark basiert und durch anwendungsspezifische Eigenschaften erweitert wird [O'N97]. Der anwendungsspezifische Benchmark zielt darauf ab, spezielle Anforderungen und Szenarien einer bestimmten Applikation auf unterschiedlichen Plattformen zu testen. Umso spezialisierter die mögliche Simulation desto besser kann der Benutzer Entscheidungen im Bezug auf diese eine Applikation treffen. Der generische Benchmark definiert generelle Szenarien, die es dem Benutzer erlauben, allgemeine Entscheidungen zu treffen. Ein solcher Test ist meist nur für eine Applikationsdomäne anwendbar und beinhaltet typische Szenarien, die für eine solche Domäne gültig sind. Ein hybrider Benchmark ist definiert durch die Rahmenbedingungen eines generischen Benchmarks mit spezifischen Erweiterungen, welche die Grundlage des allgemeinen Use Cases erweitern und an die jeweilige Situation anpassen. Aufbauend auf einem generischen Testszenario können somit spezielle Anforderungen sowie Umgebungen berücksichtigt werden [O'N97]. Bei Datenbank Performance Messungen ist des Weiteren zu unterscheiden zwischen *"single-user"* und *"multi-user"* Simulationen. Je nach Anforderungen muss zwischen beiden differenziert werden.

Als Ergebnis eines Benchmarks muss eine möglichst aussagekräftige und universell verständliche Variable, beziehungsweise Variablen gewählt werden. Je nach Anforderung werden Kosten mit einbezogen oder nicht. Dies hängt davon ab, ob der Benchmark darauf abzielt Systeme zu vergleichen und den Benutzer bei der Kaufentscheidung zu unterstützen, oder der Benutzer das

bestehende System optimiert und zwischen Konfigurationseinstellungen verglichen wird. Allgemein ist hier anzumerken, dass der Trend darin besteht, Benchmarks im Bezug auf die Kaufentscheidung zu entwickeln. Neben Werten können beispielsweise Eigenschaften als Kriterium herangezogen werden, um von vornherein bestimmte Systeme auszuschließen. So wird bei aktuellen OLTP[2]-Benchmarks die Vorraussetzung überprüft, ob Transaktionen die ACID[3]-Eigenschaften berücksichtigen. Als Variablen zur Messung der Leistung wird meistens der im DebitCredit Benchmark verwendete Ansatz [ea85] herangezogen, welcher zwei Variablen definiert. Erstens die Höchstleistung in Form von *transactions per second (tps)* und zweitens der Wert des Preis-Leistungs Verhältnisses in Form von *dollars per transactions per second ($/tps)* [O'N97].

Weiters gilt es zu unterscheiden zwischen Skalar- und Vektorwert als Endprodukt eines Benchmarks. Der oben erwähnte Ansatz im DebitCredit Benchmark mit zwei Variablen *transactions per second (tps)* und *dollars per transactions per second ($/tps)* ist ein Beispiel für Skalarwerte als Endergebnis. Vektorwerte werden zum Beispiel in Form von Laufzeit (*elapsed time*) und CPU Auslastung in Kombination präsentiert. Eine Eigenschaft die der Vektor als Ergebnis mit sich bringt, ist die Option der Trennbarkeit. Man kann somit ein Benchmark Ergebnis in mehrere Bereiche unterteilen und wenn nötig, sich nur auf einen spezifischen Wert konzentrieren. Dies führt wiederum dazu, dass auf der Entscheidungsebene gewisse Nachteile entstehen. Ein fiktiver Leistungsvergleich zwischen System A und B führt beispielsweise dazu, dass im Vergleich System A im Bereich "CPU" besser abschneidet jedoch im Bereich der "tps" schlechter und System B genau umgekehrt. Je nachdem auf welchen Bereich man sich als Entscheidungsträger fokussiert, fällt der Entschluss auf ein jeweils unterschiedliches System [O'N97].

Schlussendlich müssen der Inhalt eines Benchmarks klar definiert und die

[2] Online-Transaction-Processing

[3] Atomicity, Consistency, Isolation und Durability (Atomizität, Konsistenz, Isolation und Dauerhaftigkeit) [EN05]

Szenarien gut strukturiert sein. Man kann zwei Arten von Szenarien gegenüber stellen. Einerseits Datenbanktransaktionen, die nur eine bestimmte Art von Abfragen behandeln; andererseits Transaktionen, die aus unterschiedlichen Abfragen bestehen. Je nach Anforderung und Zielsetzung des Benchmarks muss unterschieden werden, um jeweils das optimale Ergebnis zu erzielen. So kann es sein, dass bestimmte Engpässe nur durch Lasttests spezifischer Abfragen erzielt werden, oder ein realistisches Szenario simuliert werden soll, was wiederum für unterschiedliche Abfragen spricht [O'N97].

Die Komponenten eines Datenbank Benchmarks setzen sich laut den Verfassern des Buches *"Database benchmarking"* [ABF+06] aus verschiedenen Ebenen zusammen. Spezifikation und Kontroll Logik stellen den elementaren Grundstein eines Tests dar. Daraufhin folgt die Implementation, basierend auf den zuvor klar definierten Komponenten. Die Spezifikation ist zu unterteilen in:

- Detailinformation
- Design Ziele
- Datenpunkte
- Testplan

Die Detailinformation soll den Benchmark in seinen Details beschreiben und etwaige Prozesse definieren die für den spezifischen Test von Bedeutung sind. Die Design Ziele sind in vier Unterpunkten definiert und dienen als Kriterien für einen erfolgreichen Benchmark:

- Skalierbarkeit
- Simplizität
- Repräsentativität
- Genauigkeit

Skalierbarkeit steht für die Option, möglichst flexibel eine Vielzahl an Benutzern zu simulieren was wiederum auf die Größe der zugrunde liegenden

Datenbank Auswirkungen hat. Ein Benchmark sollte von Grund auf einfach zu verstehen und umzusetzen sein. Dies ist vor allem dann wichtig, wenn Endbenutzer im Spiel sind, die auf die Funktionalität des Benchmarking Tools angewiesen sind und keine Zeit aufwenden können, Quellcode Änderungen vorzunehmen. Die Repräsentativität ist ein weiterer wichtiger Punkt bei der Entwicklung eines Benchmarks. Testdaten und Testtabellen sollten realistisch modelliert werden und die Realität möglichst detailgetreu abbilden. Als letzter Punkt ist die Genauigkeit zu nennen, welche darauf abzielt den Test an sich an gegebene Situationen anzupassen. So kann es sein, dass während bestimmter Zeitfenster Engpässe erwartet werden könnten; werden diese nicht berücksichtigt oder miteinbezogen so verliert der Benchmark an Genauigkeit und Relevanz.

Bevor die eigentliche Implementation startet, müssen Datenpunkte definiert werden, die klar aufzeigen, was gemessen werden soll und auf was gezielt geachtet wird. So stellt ein Beispiel als Datenpunkt die *transactions per second* bei einer Benutzerlast von 200 dar. An solch einem Punkt können beispielsweise Engpässe erwartet, oder realistische Vergleiche angestellt werden. Der letzte Punkt der Spezifikation ist der Testplan welcher die realistische Struktur des Benchmarkvorgangs sicherstellt.

Die Kontroll Logik als zweiter Grundstein des Benchmarks stellt den Teil abseits der Implementierung und statistischen Sammlung dar. Hier sollen Wiederholbarkeit und exakte Statistiken gegeben sein. Unter der Wiederholbarkeit versteht man, dass der Benchmark Test Ergebnisse aus wiederholten Simulationen liefern soll. Kann ein Test nicht repräsentativ wiederholt werden, ist dieser bedeutungslos. Exakte Statistiken gewährleisten einen Anhaltspunkt für Benutzer wann Engpässe erreicht sind und wie sie zu identifizieren sind. Auch hier sind als Beispiel *transactions per second* zu nennen, wodurch Engpässe identifiziert werden können [ABF+06].

2.2 Forschungsentwicklungen

Patrick O'Neil erwähnt in seinem Kapitel *"Database Performance Measurement"* als ältesten Benchmark den Wisconsin Benchmark [O'N97]. Dieser Benchmark, welcher an der Universität von Wisconsin entwickelt wurde, war eine *"single-user"* Simulation, die fünf verschiedene Arten von Abfragen behandelte:

- Selektion mit verschiedenen Selektivitäts Faktoren
- Projektion mit unterschiedlichen Anteilen an redundanten Attributen
- Single und Multiple Joins
- Aggregierte Funktionen
- Update Statements: append, delete und modify

Dabei wurde der Benchmark so implementiert, dass jede der Abfragen entweder einen *"primary index"* oder einen *"secondary index"* verwenden konnte. Als Variable zur Performance Messung wurde die Laufzeit (*elapsed time*) gewählt. Diese wurde mithilfe von Systemprogrammen auf den Testsystemen gemessen (Kommando *"time"* auf einer Unix Umgebung und *"date"* auf einer VMS[4] Umgebung) [BDT83].

Neben dem Wisconsin Benchmark entwickelten führende Fachleute den IBM basierten TP1 Benchmark, welcher darauf abzielte die Leistung von ATM[5] Transaktionen im Batch-Modus zu messen [ABF+06]. Darüber hinaus wurde als Weiterentwicklung des Wisconsin Benchmarks der DebitCredit Benchmark entwickelt. Dieser wurde im von Jim Gray publizierten Artikel *"A Measure of Transaction Processing Power"* [ea85] näher definiert. Dabei handelt es sich um eine Performance Messung im Bereich der OLTP-Systeme [O'N97]. Der Benchmark wurde aufgrund der Nachfrage nach einem Leistungstest für ein Bankensystem, welches im Jahre 1973 online gehen wollte, entwickelt. Inhalt des Tests sind dabei eigentlich drei Benchmarks: der Debit-

[4] Virtual Memory System, ein Betriebssystem des ehemaligen Computerherstellers Digital Equipment Corporation. Heute bekannt als OpenVMS [vms06].

[5] Automated Teller Machine - Bankautomat

Credit an sich, ein Sort und ein Scan Benchmark. Der Inhalt des Scan Benchmark ist eine kleine Cobol Transaktion die im Batchmodus 1000 Einträge abfragt und aktualisiert; der Sort Benchmark eine Batch-operation die eine Million Einträge sortiert. Der DebitCredit Benchmark basiert auf der Struktur eines großen Bankensystems mit 1.000 Filialen, 10.000 Automaten und 10.000.000 Konten. Zusätzlich dazu wurde die Datenbank mit einer Statistik Tabelle definiert, welche 90 Tage an Gebrauchsstatistik beinhaltete. Um die Leistung zu messen, wurde eine DebitCredit Transaktion definiert, welche den Benchmark als solches darstellt und in Abbildung 2.1 zu sehen ist.

```
BEGIN-TRANSACTION
    READ MESSAGE FROM TERMINAL (100 bytes)
        REWRITE ACCOUNT (random)
        WRITE HISTORY (sequential)
        REWRITE TELLER (random)
        REWRITE BRANCH (random)
    WRITE MESSAGE TO TERMINAL (200 bytes)
COMMIT-TRANSACTION
```

Abbildung 2.1: DebitCredit Transaktion [ea85]

Basierend auf dieser Transaktion wird der Begriff *"transactions per second" (tps)* wie folgt definiert: TPS is der *"Höchstwert der Durchsatzrate von DebitCredit Transaktionen bei einer Reaktionszeit von einer Sekunde für 95% aller Transaktionen" (vgl. Jim Gray [Gra87])*. Dieser Wert wird mit dem Preisleistungsverhältnis in Verbindung gebracht und liefert in Kombination mit den Scan und Sort Benchmarks, laut Aussage der Verfasser einen optimalen Vergleich zwischen Datenbanksystemen [ea85].

In dem Bestreben, Datenbank Benchmarks zu standardisieren, gründete Omri Serlin im August 1988 mit acht weiteren Firmen ein Konsortium genannt *"Transaction Processing Performance Council"*, kurz "TPC" [Sha98]. Bald darauf wurden die beiden Benchmarks TPC-A und TPC-B veröffentlicht. TPC-A basierte auf dem von Gray entwickelten DebitCredit Benchmark und TPC-B auf IBM's TP1 Benchmark. Es wurde daraufhin der TPC-

C Benchmark entwickelt, welcher TPC-A und TPC-B im Jahre 1992 ablösen sollte und somit einen neuen Standard im Bezug auf OLTP-Benchmarks setzte. Weiters wurde der TPC-D entwickelt, welcher sich auf die Applikationsdomäne der Entscheidungsunterstützungssysteme[6] fokussierte. TPC-D wird im Laufe der Zeit durch TPC-H, auf den noch näher eingegangen wird, ersetzt. Zuletzt wurde der TPC-E im Jahre 2007 entwickelt, welcher den aktuellsten Benchmark im Bezug auf OLTP-Systeme darstellt. Dieser simuliert ein Maklerunternehmen, löst den TPC-C jedoch nicht als OLTP-Benchmark ab sondern hat ergänzenden Charakter. Im Folgenden wird auf die drei aktuellsten Benchmarks TPC-C, TPC-H und TPC-E genauer eingegangen, um Vorgehensweisen und Prinzipien ersichtlich zu machen [ABF+06].

Der TPC-C Benchmark soll einen Großhandellieferanten simulieren. Dabei gibt es fiktive Benutzer, die fünf verschiedene Transaktionen ausführen können. Diese beinhalten Funktionen wie zum Beispiel die Eingabe von Bestellungen und Zahlungen oder die Überprüfung des Bestands im Lager sowie die konstante Berichterstattung über den Status von Bestellungen. Die Transaktionen bestehen zum größten Teil aus schreibgeschützten sowie update-intensiven Statements. Die Messung der Performance erfolgt durch die Angabe von *"transactions per minute" (tpmC)* und gibt somit an wieviel Bestellungen pro Minute ausgeführt wurden [ABF+06].

Der TPC-H Benchmark ist ein Entscheidungsunterstützungssystembenchmark, welcher so aufgebaut ist, dass bestimmte Abfragen zum Großteil im Hintergrund und ohne Benutzerinput erfolgen. TPC-H gehört zu den Data-Warehouse Umgebungen was schlussfolgern lässt, dass die Größe der Datenbank immens und damit einhergehend die Dauer länger ist. Als Minimum ist der TPC-H Benchmark mit 10.000 Datensätzen von Lieferanten bestückt und beinhaltet 10.000.000 Tupel, welche auf eine Summe von 1GB kommen. Es gibt einen Skalenfaktor, der es erlaubt die Datenbank auf eine Größe von 100TB zu definieren. Die Messung der Performance erfolgt in Form von *"Composite Query-per-Hour" (QphH@Size)* und wird in Zusammenhang mit der Grösse der Datenbank dargestellt. Letzten Endes gibt es zwei

[6] DSS - Decision Support System

Varianten von Tests, die der TPC-H Benchmark zur Verfügung stellt. Diese sind der *"Power Test"*, welcher einen einzigen Benutzer simuliert sowie der *"Throughput Test"*, welcher mehrere Benutzer simuliert. Wichtig is hierbei, dass Vergeiche zwischen Systemen nur dann Sinn machen wenn die Größe der Datenbank gleich bleibt [ABF+06].

Der TPC-E Benchmark ist der aktuellste unter den zur Verfügung gestellten Benchmarks von TPC. Er soll ein Maklerunternehmen simulieren, welches in drei Aufgabengebiete unterteilt ist. Darunter fallen die Verwaltung der Kundenkonten, die Ausführung von Kundenaufträgen und die Interaktion zwischen Kunde und Finanzmarkt. Die Anzahl der Kunden kann frei definiert werden, um verschiedene Arbeitslasten zu generieren. Die zugrunde liegende Datenbank ist in drei Tabellen für Kunden, Makler und den Markt an sich unterteilt, sowie in eine weitere Tabelle welche allgemeine Daten beinhaltet wie besipielsweise Postleitzahlen und sonstige statische Daten. Die Transaktionen beinhalten schreibgeschützte sowie update-lastige Statements, welche auf den drei Haupttabellen ausgeführt werden. Die Messung der Performance erfolgt in Form von *"transactions per second" (tpsE)* [Tra07].

2.3 Schlussfolgerungen

Im Bereich der Datenbank Performance Messung gibt es je nach Applikationsdomäne unterschiedliche Vorgehensweisen. Grundlegende Motivation für die Entwicklung von Benchmarks ist die Möglichkeit des Vergleichs von Datenbanksystemen untereinander. Dabei ist ein wesentlicher Charakter von aktuellen Benchmarks, dass sie darauf abzielen dem Benutzer beim Entscheidungsprozess für eine Datenbank weiterzuhelfen. Dieses Szenario ist in unserem Fall nicht gegeben. Benchmarking-Tools bieten auch die Grundlage für die Optimierung von bestehenden Datenbanken. Wichtig ist dabei, dass bei der Auswahl von Variablen für diese Tools preisliche Komponenten, wie es bei den oben beschriebenen Benchmarks der Fall war, nicht Inhalt der

Leistungsmessung und Statistik sein sollen. Dadurch dass eine bestehende Datenbank optimiert wird, gilt es, die grundlegenden Prinzipien der aktuellsten Benchmarks zu übernehmen, um somit aussagekräftige Leistungsmessung zu gewährleisten, die auch möglicherweise für andere Bereiche vergleichend herangezogen werden kann. Für den weiteren Verlauf dieser Arbeit wird somit die in vielen Benchmarks verwendete Kombination aus Transaktionen pro Sekunde und Preis nur als Transaktionen pro Sekunde kurz "tps" übernommen. Weiters wird zusätzlich, wie zum Beispiel im Debit-Credit Benchmark von Gray ein Höchstwert für den Durchsatz von Transaktionen als Wert einfließen, sowie der durchschnittliche Wert aller Transaktionen im Bezug auf die verstrichene Zeit. Darüber hinaus soll die Skalierbarkeit wie es beispielsweise beim TPC-H Benchmark der Fall ist durch die Anzahl der Tabellen und die Anzahl der darin enthaltenen Tupel definiert werden. Bezogen auf die Simulation der Verbindungen und somit der möglichen Benutzer soll wie auch beim TPC-H Benchmark die Möglichkeit gegeben werden zwischen zwei Simulationsvarianten, nämlich dem Einbenutzer- und Mehrbenutzerbetrieb, zu wählen. Je nach Festlegung des Mehrbenutzerbetriebs wird hier ein zusätzlicher Skalierungsfaktor ins Spiel kommen was Auswirkungen auf die Generierung von Daten hat. Schlussendlich gilt es zu unterscheiden, in welche Kategorie von Benchmark das Tool fällt. Dadurch dass sehr spezielle Abfragen in Kombination mit allgemeinen Transaktionen implementiert werden wird das Programm zur Leistungsmessung in die Kategorie der hybriden Benchmarks fallen.

3 PostgreSQL Tuning

Folgendes Kapitel behandelt allgemeine Database Tuning Methoden anhand des Buches *"Database Tuning" [SB03]* und geht im weiteren Verlauf auf spezifische Tuning Möglichkeiten für eine PostgreSQL Datenbank ein.

3.1 Prinzipien des Tunings

Die Optimierung einer Datenbank beinhaltet eine Vielfalt an Aspekten die berücksichtigt werden müssen. Diese reichen von den grundlegenden Hardware Optionen über Datenbank Parameter bis hin zu Transaktions- und Datenbankdesign [SKS01]. Die Problematik besteht darin, dass ein breites und zugleich tiefgehendes Wissen in Verbindung mit den potentiellen Wechselwirkungen zwischen den ausschlaggebenden Ebenen und Einstellungen gegeben sein muss. In dem Buch *"Database Tuning" [SB03]* verfasst von Dennis Shasha und Philippe Bonnet definieren die Autoren fünf grundlegende Prinzipien im Bezug auf Datenbank Tuning[1]. Anhand folgender Prinzipien soll dem Benutzer eine Möglichkeit geboten werden, der Komplexität entgegen wirken zu können.

- Think globally; fix locally.
- Partitioning breaks bottlenecks.
- Start-up costs are high; running costs are low.

[1] Aufgrund der informellen Schreibweise der Autoren und der Annahme dass die englische Sprache kein Hindernis sondern mitunter auch den Ursprung vieler technischer Begriffe darstellt, wird in diesem Fall darauf verzichtet direkt zu übersetzen. Es soll die Aussagekraft der fünf Prinzipien durch die englische Sprache beibehalten werden.

- Render unto server what is due unto server.
- Be prepared for trade-offs.

Das erste Prinzip, *"Think globally; fix locally"*, zielt auf eine global orientierte Denkweise und lokal orientierte Problemlösung ab. Prinzipiell soll der Überblick dazu beitragen, das Problem aus mehreren Sichten zu beleuchten, um dann eine genaue Identifikation und effiziente Lösung zu gewährleisten. Zwei typische Praxisfallen sollen dieses Prinzip näher erläutern. Als erstes Beispiel ist die typische Herangehensweise an den Prozess der Optimierung einer Datenbank zu nennen. Der primäre Fokus im ersten Schritt besteht meist darin, auf die Hardware Aktivitäten zu achten. Dies könnten zum Beispiel eine hohe Prozessor Auslastung oder hohe Input-Output Aktivität sein. Eine Schlussfolgerung die man daraus ziehen könnte, würde man nur lokal orientiert an die Problemlösung herangehen, wäre die Anschaffung besserer, oder Erweiterung vorhandener Hardware, beispielsweise durch mehr Festplattenspeicher. Eine global orientierte Sichtweise im Bezug auf dieses Beispiel bringt andere Ebenen in die Problemlösung mit ein. So besteht bei hoher Input-Output Aktivität die Möglichkeit, dass eine Abfrage eine ganze Tabelle scannen muss anstatt einen Index zu verwenden. Es kann auch sein, dass Log-dateien auf einer Festplatte gespeichert werden die im Allgemeinen eine hohe Zugriffsrate aufweist. Betrachtet man das Problem hoher Input-Output Aktivität unter diesem Aspekt wäre die Verwendung eines Index oder die Auslagerung von Log-dateien auf andere Festplatten als Lösung gegenüber der Hardware Anschaffung zu bevorzugen.

Als zweites Beispiel für eine typische Praxisfalle ist der Zusammenhang zwischen gemessener Zeit und Abfrage zu nennen. Benötigt eine Abfrage viel Zeit wird die Reduzierung dieser Zeit zum Ziel gesetzt. Dennoch muss auch hier der größere Zusammenhang in Betracht gezogen werden. So ist es wenig hilfreich, eine Abfrage zu optimieren, die nur ein Prozent der gesamten Laufzeit einnimmt. Die Beschleunigung würde sich fast gar nicht bzw. wenn dann nur leicht-bemerkbar machen und das eigentliche Problem nicht lösen. Betrachtet man die anderen 99% und lokalisiert die signifikante Abfrage, so besteht dabei mehr Potenzial eine Performance Steigerung zu erzielen. Es

gilt somit die kritischen Abfragen im globalen Zusammenhang zu erkennen und diese gezielt zu optimieren [SB03].

Das zweite Prinzip, *"Partitioning Breaks Bottlenecks"*, zielt auf die Bedeutung von Partitionierung im Bezug auf Engpässe ab. *"Ein langsames System ist in den meisten Fällen langsam, nicht aus dem Grund weil es voll ausgelastet ist, sondern weil ein Punkt die allgemeine Performance einschränkt"* (vgl. Dennis Shasha und Philippe Bonnet [SB03]). Diese Einschränkung wird als Engpass bezeichnet. Dennis Shasha und Philippe Bonnet definieren als Strategie zur Vorgehensweise bei Engpässen die Stelle im System zuerst zu beschleunigen, falls dies nicht ausreichend ist, sollte versucht werden, durch Partitionierung eine Optimierung zu erreichen. Allgemein ist unter Partitionierung die Unterteilung in disjunkte, sich nicht kreuzende Teile zu verstehen. Das Prinzip dieser Vorgehensweise basiert darauf, dass verschiedene Arten von Partitionierungen unterschiedliche Lösungen mit sich bringen. Grundsätzlich gibt es die physikalische, logische und zeitliche Partitionierung. Unter der ersten Variante ist die physische Unterteilung von Daten zu verstehen. Dabei geht es darum durch Auslagerung von Daten eine Entlastung eines bestimmten Bereiches zu erreichen. Durch die logische Partitionierung soll durch eine Unterteilung von internen Strukturen und Verweisen eine Optimierung erzielt werden. Darunter fallen beispielsweise Konkurrenzsituationen in denen mehrere Threads auf eine Liste mit Verweisen auf freien Datenbankpuffer zugreifen. Will man eine solche Situation logisch partitionieren, bestünde eine Möglichkeit darin, mehrere Listen mit Verweisen zu erzeugen. Der Engpass der dadurch entsteht, dass mehrere Threads auf eine Liste zugreifen wird somit entlastet, dadurch dass Threads nun die Auswahl zwischen mehreren Listen haben, was schlussendlich die Konkurrenzsituation deutlich entlasten kann. Letztendlich kann durch eine zeitliche Partitionierung auch eine Optimierung erzielt werden. Zeitlich bedeutet in diesem Zusammenhang Serialisierung. So können lange zeitintensive Transaktionen die synchron mit kurzen zeitsparenden Transaktionen auf dieselben Daten zugreifen, durch eine zeitliche Unterteilung beschleunigt werden [SB03].

Das dritte Prinzip, *"Start-Up Costs Are High; Running Costs Are Low"*, zielt auf die Problematik ab, dass der Aufbau einer Verbindung, das Initialisieren einer Lese- oder Schreiboperation sowie das Senden von Daten über ein Netzwerk viel Zeit in Anspruch nehmen kann. Es gilt dabei das Prinzip, sich dessen bewusst zu sein und bei jeglichen Abläufen zu versuchen, so wenig Initialisierungen wie möglich zu veranlassen. Als Beispiel wäre hier eine Standardimplementierung in Java, C++ oder Perl zu nennen, bei der einfache Abfragen auf eine Datenbank gemacht werden. Die Reduzierung auf ein Minimum an Transaktionen zwischen Datenbank und Programm ist hierbei anzustreben, was auf das Design des Programms Auswirkungen hat. So ist es beispielsweise besser, eine einzige SELECT Abfrage abzuschicken, und das resultierende Ergebnis in einer Schleife voll auszuschöpfen, als in einer Schleife mehrere SELECT Abfragen zu tätigen. Ziel ist es somit im Allgemeinen eine Umsetzung mit möglichst wenigen Initialisierungsprozessen, wie zum Beispiel ein Verbindungsaufbau, zu implementieren.

Das vierte Prinzip, *"Render unto Server What Is Due unto Server"*, basiert auf einer Designfrage, die sich mit der Verteilung von Prozessen zwischen Datenbanksystem (Server) und Applikation (Client) beschäftigt. Je nachdem wo Prozesse ausgeführt werden, kommt es zu Verzögerungen oder nicht. Es gibt dabei drei Faktoren, die es erleichtern die Verteilung zu optimieren. Erstens muss berücksichtigt werden, wie viel Ressourcen zur Verfügung stehen; zweitens spielt es eine Rolle, wo sich bestimmte Daten zwischen Datenbanksystem und Applikation befinden; drittens sollte in Betracht gezogen werden ob die Applikation mit dem Endbenutzer kommuniziert und somit Ausgaben über den Bildschirm getätigt werden müssen. Grundsätzlich gilt es rechenintensive Operationen, die nicht mit Transaktionen direkt in Verbindung stehen, auf den Client auszulagern um den Server zu entlasten. Weiters kann es zum Beispiel sein, dass bestimmte Änderungen in der Datenbank beim Client rückgemeldet werden sollen. Dies kann clientseitig implementiert werden indem die Applikation in konstanten Abständen abfragt ob sich etwas in der Datenbank geändert hat oder indem serverseitig ein Trigger eingebaut wird der die Applikation nur dann benachrichtigt wenn sich etwas

geändert hat. Hier ist die serverseitige Implementation performancemässig von Vorteil, da durch die konstante Abfrage der Applikation ein weitaus höherer Overhead entstehen würde als durch den Trigger. Letztendlich spielt es auch eine Rolle ob die Datenbank während einer Transaktion mit dem Bilschirm kommuniziert. Ist dies der Fall so kann es sein, dass durch die Interaktion mit dem Bildschirm, sei es durch eine Benutzereingabe oder Rückmeldung, Locks innerhalb der Transaktion gesetzt werden und möglicherweise andere Transaktionen dadurch nicht auf diese Daten zugreifen können. Es gilt daher in diesem Zusammenhang den Teil der Transaktion der mit dem Bildschirm in Verbindung steht auszulagern, und zwar durch eine Aufteilung der Transaktion in drei Teile. Der erste Teil bereitet all die Daten auf die zur Interaktion mit dem Bildschirm nötig sind, der zweite Teil kommuniziert mit dem Client außerhalb der Transaktion und der dritte Teil verarbeitet die Kommunikation mit einer finalen Transaktion. Durch eine solche Aufteilung wird verhindert, dass durch lange Kommunikationszeiten mit dem Bildschirm und gleichzeitig noch aktiven Locks von Transaktionen, Blockierungen unter mehreren Transaktionen entstehen [SB03].

Das fünfte Prinzip, *"Be Prepared for Trade-Offs"*, weist auf das Zusammenspiel mehrerer Komponenten hin und die damit einhergehenden Kompromisse die zu treffen sind. Will man einen Bereich beschleunigen, in dem man zum Beispiel zusätzlichen Arbeitsspeicher hinzufügt, um die Puffergröße zu erhöhen, muss man damit rechnen dass Kosten entstehen, die bei großen Anschaffungen nicht gering ausfallen werden. Man darf nicht vergessen, dass bestimmte Optimierungsmöglichkeiten höhere Kosten mit sich tragen als standardmäßige. Eine Abwägung des Verhältnisses zwischen Kosten und dem einhergehenden Nutzen ist somit in jedem Fall zu empfehlen. Neben dem Kostenfaktor sind andere Wechselwirkungen, die durch das Tunen einer Datenbank entstehen, zu beachten. So ist das Hinzufügen eines Index für eine kritische Abfrage im Hinblick auf die zeitliche Optimierung sicherlich von Vorteil, dennoch muss man sich bewusst sein, dass ein zusätzlicher Index mehr Platz im Arbeits- und Festplattenspeicher beansprucht. Im Allgemeinen müssen der Kostenfaktor sowie die Wechselwirkungen von Optimie-

rungsmöglichkeiten Bestandteil des Entscheidungsprozesses sein [SB03].

Das Tunen einer Datenbank wird wie schon oben erwähnt von verschiedenen Wechselwirkungen beeinflusst. Dabei bilden die fünf Prinzipien des Tunings den Grundstein für eine optimale Herangehensweise zur Optimierung einer Datenbank. Basierend auf diesen Punkten ist es dem Tuner möglich auf verschiedenen Ebenen Einfluss auf die Performance der Datenbank zu nehmen. Es soll im weiteren Verlauf auf diese Ebenen und ihren Zusammenhang zu einer PostgreSQL Datenbank eingegangen werden.

3.2 Hardware Tuning

Die Hardware stellt den Grundbaustein für die Performance einer Datenbank dar. Drei grundlegende Komponenten, bestehend aus Festplattenspeicher, Arbeitsspeicher und Prozessor tragen dazu bei, dass ein Datenbanksystem eine gewisse Last bewältigen kann oder nicht. Die schlichte Maximierung der vorhandenen Komponenten ist erstens aus Kostengründen zu vernachlässigen und zweitens kein Garant dafür, dass auch wirklich eine optimale Konfiguration erreicht wird. Es kann jedoch empfohlen werden sich bei der Optimierung von vorhandener Hardware auf die interne Speicherstruktur zu fokussieren.

3.2.1 Optimierung der Hardwarekonfiguration

Die Speicherstruktur der Hardwarekonfiguration wird definiert durch mehrere Festplattenspeicher sowie Arbeitsspeicher und ihre Einstellungen bezüglch der Verwaltung von Daten. Dabei stellt sich im Bezug auf Festplattenspeicher grundsätzlich die Frage welche RAID[2] Konfiguration gewählt werden soll. Die meist verwendeten RAID Konfigurationen im Bereich von Datenbanksystemen sind *RAID 0, RAID 1, RAID 5, RAID 6,*

[2]Redundant Array of Inexpensive Disks

RAID 10, RAID 50 und *RAID 60*. Im Folgenden sollen Ihre Eigenschaften im Bezug auf Optimierungsmöglichkeiten näher erläutert werden. *"Bei RAID 0 werden die Daten in Blöcke aufgeteilt, die parallel auf die vorhandenen Plattenlaufwerke geschrieben werden (engl.: disk striping) [HN05]."* Es besteht somit keine redundante Information, was zur Folge hat, dass bei einem Defekt alle Daten verloren gehen. Bei mehreren Festplatten die über *RAID 0* zusammen geschlossen sind bedeutet eine Lese- oder Schreiboperation den parallelen, physikalischen Zugriff auf alle vorhandenen Platten [SB03]. *"Bei RAID 1 wird mit gespiegelten Platten (engl.: disk mirroring, shadowing) gearbeitet, das heißt, die gleichen Daten werden gleichzeitig auf unterschiedliche Laufwerke geschrieben [HN05]."* Dies bedeutet, dass die Information redundant gespeichert wird und bei dem Defekt einer Platte die Daten auf der gespiegelten Platte noch immer vorhanden sind. Im Bezug auf Lese- und Schreiboperationen kommt es in manchen Situationen auf die Vorgehensweise des Controllers an. Generell wird bei einer Schreiboperation auf beiden physikalischen Platten geschrieben. Bei einer Leseoperation kann es sein, dass entweder auf beiden physikalischen Platten gelesen wird oder auf der Platte die weniger Aktivität aufweist [SB03]. *"Bei RAID 5 werden Daten und zur Rekonstruktion nötige Prüfsummen quer über alle Laufwerke geschrieben [HN05]."* Bei dieser Konstellation wird durch Prüfsummen eine Toleranz gegenüber Defekten gewährleistet. Aufgrund dieser Eigenschaft besteht ein gewisser Overhead bei Schreiboperationen. So muss bei einer solchen je zweimal gelesen und zweimal geschrieben werden, um erfolgreich Daten abzuspeichern [SB03]. Weiters besteht Redundanz in dem Sinne, dass bei einem Ausfall einer Festplatte Daten mittels Prüfsummen wiederhergestellt werden. Dies führt zu einer gewissen Redundanz an Informationen, da einerseits die Daten an sich und andererseits die Prüfsummen gespeichert werden müssen [Adv08]. RAID 6 ist die erweiterte Form von RAID 5. Dies bedeutet, dass eine zusätzliche Prüfsumme mittels eines weiteren Algorithmus generiert wird. Die resultierende zweite Prüfsumme wird unabhängig von der ersten auf einer unterschiedlichen Festplatte gespeichert. Der Vorteil der dadurch entsteht ist, dass eine extrem hohe Resistenz gegen Datenverluste gegeben ist und zusätzlich dazu auch mehrere Festplat-

ten zugleich ausfallen können. Trotzdem muss dabei bedacht werden, dass N+2 Festplatten verwendet werden müssen, was wiederum sehr kostspielig sein kann. Als erste zu erwähnende Kombination von RAID Systemen ist RAID 10 oder auch RAID 1+0 zu nennen. Es handelt sich dabei um eine Kombination aus RAID 0 und RAID 1 Eigenschaften. Grundsätzlich werden die Daten in Blöcke aufgeteilt und über die Hälfte der vorhandenen Festplatten verteilt gespeichert. Zusätzlich dazu wird jede Platte gespiegelt. Es wird somit von der Geschwindigkeit durch RAID 0 und der Toleranz gegen Defekte unter RAID 1 gleichzeitig profitiert. Dadurch dass jedoch zweimal so viele Platten verwendet werden müssen wie bei RAID 0, ist diese Option eine kostspielige, und rentiert sich daher nur bei zahlreich vorhandenen Festplatten [SB03]. Eine weitere Option stellt RAID 50 dar. Diese Konstellation ist charakterisiert durch ihre Kombination aus RAID 0 und RAID 5. Es werden parallel zwei RAID 5 Systeme in einem RAID 0 System zusammengeschlossen. Dies hat den Vorteil, dass eine hohe Datentransferrate aufgrund der RAID 5 Aufstellung erreicht werden kann. Weiters trägt die RAID 0 Eigenschaft dazu bei, dass relativ hohe Input/Output Raten für kleine Anfragen erreicht werden können. Dennoch ist hier der Kostenfaktor nicht ausser Acht zu lassen, da unter anderem ein Minimum an sechs Festplatten benötigt wird, um eine RAID 50 Installation umzusetzen. Darüber hinaus ist zu bedenken, dass das System unbrauchbar wird falls zwei Festplatten in einem RAID 5 Segment ihre Funktion aufgeben [Adv08]. Somit bleibt als letzte Variante noch RAID 60 zu erwähnen. Diese Konstellation geht nach dem selben Prinzip wie RAID 50 vor, nur unter der Verwendung von RAID 6 Segmenten in einer RAID 0 Verbindung. Ein solcher Verbund bringt einerseits die bereits genannten Vorteile von RAID 6 und RAID 0 mit sich. Darüber hinaus kann eine Steigerung an Ausfallstoleranz erreicht werden, dadurch dass je zwei Platten pro RAID 6 Verbund ausfallen können, sprich vier Platten insgesamt. Andererseits muss hier in Betracht gezogen werden, dass durch eine doppelte Steigerung auch eine doppelte Menge an Prüfsummen zweifach generiert werden muss, was im Vergleich zu RAID 50 zu geringen Performanceschwächen führt [wik08b].

Nun stellt sich die Frage, welche Konfiguration für welche Umgebung am Besten geeignet ist. Laut Aussage der Autoren Dennis Shasha und Philippe Bonnet kann zwischen Logdateien, temporären Dateien und Index Dateien, beziehungsweise den zu speichernden Daten an sich, unterschieden werden. Für Logdateien wird empfohlen auf RAID 1 zu setzen. Hier soll die Toleranz gegenüber Defekten, welche durch die Spiegelung gegeben ist, ausgenutzt werden. Bei temporären Daten reicht es aus auf RAID 0 zu setzen. Normalerweise kann es in Kauf genommen werden, falls es zu einem Defekt einer Platte kommt, dass Daten verloren gehen. Schlussendlich wird bei Index Dateien und den zu speichernden Daten an sich empfohlen auf RAID 5 zu setzen. Insbesondere bei Systemen die sich auf schnelle Leseoperationen verlassen müssen, und die Geschwindigkeit von Schreiboperationen vernachlässigen können, ist eine solche Konfiguration laut den Autoren des Buches *"Database Tuning"* zu empfehlen. Bei der Wahl des RAID Controllers gehen die Meinungen auseinander. Es kann zwischen Software und Hardware RAID unterschieden werden. Grundsätzlich sollte dabei beachtet werden, dass bei einer Hardware RAID Lösung die Gefahr besteht durch einen Defekt des RAID Controllers die dahinterliegedenden Daten zu verlieren [SB03].

Ein weiterer Teil der Speicherstruktur ist der Arbeitsspeicher, welcher im Vergleich zu Festplattenspeicher um einiges teurer ist. Generell kann man sagen, dass durch die Auslagerung von häufig verwendeten Daten auf den Arbeitsspeicher eine Reduzierung der Input-Output Aktivitäten von Festplatten erreicht werden kann. Es stellt sich somit die Frage, bei gegebenem Budget eine optimale Kombination aus Platten- und Arbeitsspeicher zu erzielen, um die Transaktionen pro Sekunde (*"tps"*) zu optimieren [SKS01]. Jim Gray und Franco Putzolo definieren eine fünf Minuten Regel, die auf einer Break-Even-Analyse basiert, und die Entscheidung zwischen Arbeitsspeicherauslagerung und Festplattenspeicherung erleichtern soll [GP87]. 1986 veröffentlicht und 1997 von Jim Gray und Goetz Graefe neu überarbeitet, kann die fünf Minuten Regel als Ausgangspunkt für ein optimales Verhältnis zwischen Arbeitsspeicher und Festplattenspeicher angesehen werden

[GG97]. Grundsätzlich besagt die fünf Minuten Regel, dass Datenblöcke, die häufiger als einmal in fünf Minuten zufällig angesprochen werden im Arbeitsspeicher gecached werden sollen. Es rentiert sich laut Aussage von Gray in einer solchen Situation in Arbeitsspeicher zu investieren, um die benötigte Kapazität zu erhalten. Dabei haben sich über den Verlauf der Zeit die Faktoren wie Speichereinheiten und Kosten in einem gleich bleibenden Verhältnis geändert und gewährleisten somit die Gültigkeit der fünf Minuten Regel [SKS01]. Im Bezug auf sequentiellen Zugriff können mehr Datenblöcke pro Sekunde angesprochen werden, was zu einer "ein-Minuten Regel" führt [GG97]. Diese besagt, im selben Stil wie die fünf Minuten Regel, dass Datenblöcke die häufiger als einmal pro Minute sequentiell angesprochen werden, im Arbeitsspeicher gecached werden sollen [SKS01]. Es ist an dieser Stelle anzumerken, dass es sich bei den eben vorgestellten Regeln um Faustregeln handelt, die jeweils im Bezug auf die vorhandene Umgebung und den damit einhergehenden Zielsetzungen angewendet werden sollten um Optimierungsmöglichkeiten bewerten zu können. Um somit die Performance zu steigern, muss im Vorhinein bekannt sein, wie Daten angesprochen werden und in welcher Häufigkeit dies vorkommt. Auch hier gilt, wie in Punkt 3.1 bereits erwähnt, die globale Betrachtungsweise des Problems als grundlegendes Prinzip des Tunings.

3.2.2 Fünf Hardware Prinzipien für PostgreSQL

Josh Berkus, ein Mitglied der PostgreSQL Gruppe seit dem Jahre 2002 [too08], definiert fünf Prinzipien im Bezug auf Hardwareoptimierung, die dem Benutzer eine grundlegende Herangehensweise an die Problemstellung bieten sollen. Das erste Prinzip geht dabei auf das Verhältnis zwischen Plattenspeicher, Arbeitsspeicher und Prozessorleistung beziehungsweise -anzahl ein, welches absteigend definiert wird. Berkus empfiehlt als erste Option die Investition in hochleistungsfähige Plattensysteme, gefolgt von der Empfehlung erst im Anschluss daran in mehr Arbeitsspeicher zu investieren. Die genannte Reihenfolge wird auf die für PostgreSQL, sowie

auch für andere ACID kompatible relationale Datenbanksysteme, charakteristisch hohe Input/Output Aktivität zurückgeführt. Somit lässt sich aus Berkus' erstem Prinzip schlussfolgern, dass die zwei wichtigsten Komponenten für eine mögliche Optimierung das zugrunde liegende Plattensystem und der zur Verfügung stehende Arbeitsspeicher sind. Das zweite Prinzip geht auf das Verhältnis zwischen der Anzahl an Platten und der damit verbundenen Leistung der Datenbank ein. Der grundlegende Gedanke geht hin zu einer Maximierung von Platten, auch bei verhältnismässig kleinen Umgebungen. Wichtig ist dabei der Zusammenhang zwischen Anzahl und Performance. Je mehr Festplatten, desto schneller die Performance. Diese Schlussfolgerung zieht jedoch auch zwei Voraussetzungen mit sich. Erstens muss das Betriebssystem parallelen Schreib- und Lesezugriff auf Festplatten erlauben und zweitens müssen die Festplatten als Verbund in einer RAID Kombination implementiert und verwendet werden. Im Bezug auf die vorhandene Hardwarearchitektur der Datenbank geht Berkus in seinem dritten Prinzip auf die Kombination zwischen Datenbank und Logsystem ein. Laut seiner Aussage kann bei einer schreibintensiven Datenbank, eine Auslagerung des Logsystems auf eine eigene, seperat entkoppelte Festplatte, eine Leistungssteigerung von bis zu 12 Prozent herbeiführen. Insbesondere bei kleinen Umgebungen, wie beispielsweise einer Datenbank bestehend aus nur zwei Festplatten, kann eine solche Auslagerung zu mehr Performance führen. Als viertes Prinzip nennt Berkus die RAID Konfiguration von Festplatten. Spezifisch für PostgreSQL empfiehlt Josh Berkus die Verwendung von RAID 10 und RAID 1 im Bezug auf eine Plattenanzahl von zwei, vier oder sechs Stück; erst ab einer Anzahl von mehr als sechs Festplatten wäre RAID 5 zu empfehlen. Seine Argumentation begründet er dadurch, dass man unter RAID 5 bei einer Plattenanzahl kleiner als sechs nur 50% der Abfragegeschwindigkeit erhält, wie im Vergleich zu normalen Konfigurationen. Darüber hinaus äußert sich Josh Berkus kritisch gegenüber billigen RAID Controllern und meint, dass es oftmals sicherer wäre eine Software Lösung zu implementieren, als auf mitgelieferte Komponenten zu setzen. Es sei hier angemerkt, dass die Meinungen grundsätzlich auseinander gehen im Bezug auf Software oder Hardware RAID.

Schlussfolgernd zum vierten Prinzip bezogen auf RAID Konfiguration können folgende Annahmen getroffen werden. Je grösser die Anzahl der Platten, desto eher ist die Verwendung von RAID 5 zu empfehlen. Ansonsten sollte man auf RAID 10 oder RAID 1 setzen, wobei zu berücksichtigen ist, dass RAID 10 doppelt soviele Platten benötigt wie RAID 0. Ist die Toleranz von Defekten nicht unbedingt benötigt so kann es sinnvoll sein zum Beispiel Logdateien auf eine RAID 0 Umgebung auszulagern. Als letztes Prinzip verweist Berkus auf die Wichtigkeit von Wechselwirkungen zwischen Applikationen, Datenbanken und Server Architektur. Grundsätzlich ist es nicht empfehlenswert mehrere relationale Datenbanksysteme auf einem Server laufen zu lassen. Dadurch dass beide Systeme über diesselben Ressourcen verfügen müssen kommt es zu Konkurrenzsituationen was wiederum dazu fürt, dass keines der beiden Datenbanksysteme performant läuft. Für PostgreSQL empfiehlt es sich auf Kombinationen zu setzen die CPU und RAM lastig ausgelegt sind [ber05].

3.3 Parameter Tuning

Jedes Datenbanksystem kann mittels Parameter fein abgestimmt werden. Dabei hängt es vom ausgelieferten System ab welche Parameter wie eingestellt werden können [SKS01]. Für das relationale Datenbanksystem PostgreSQL können grundsätzlich an die 200 Werte in der Konfigurationsdatei[3] gesetzt werden [ber08]. Aufgrund dieser Fülle an Möglichkeiten wird im folgenden Abschnitt nur auf einige Parameter eingegangen. Dabei fällt die Auswahl hauptsächlich auf jene, die eine mögliche Performancesteigerung herbeiführen können. Ausgewählt wurden dabei diejenigen die laut offizieller PostgreSQL Wiki[4] am ausführlichsten behandelt wurden [wik08c]. Diese lauten wie folgt:

- max_connections

[3] postgresql.conf
[4] http://wiki.postgresql.org/

- shared_buffers
- effective_cache_size
- work_mem
- maintenance_work_mem
- random_page_cost
- log_directory und pg_xlog

Der erste Parameter *max_connections* definiert die maximale Anzahl an simultanen Verbindungen für einen PostgreSQL Server. Grundsätzlich muss dabei der Zusammenhang zu den Speicherparametern beachtet werden. Wird ein zu hoher Wert gewählt, so hat dies direkte Auswirkungen auf den verteilten Speicher *shared_buffer*, was auf den client- und serverseitigen Overhead pro Verbindungsaufbau zurückzuführen ist. Bei Umgebungen mit zahlreichen Verbindungen ist ausserdem auf Connection Pooling zu setzen um dem vorhin genannten Overhead entgegenzuwirken. Die Bereitstellung vieler Verbindungen bringt zugleich mehr Speicherbedarf mit sich [ber05].

Bezogen auf die Parameter bezüglich der Speicherverwaltung gibt es mehrere Punkte an denen gedreht werden kann. Wie schon oben erwähnt steht der *shared_buffer* in engem Zusammenhang mit den maximal zur Verfügung gestellten simultanen Verbindungen. Wichtig ist dabei zu beachten, dass es ich bei den *shared_buffers* nicht um den gesamten Speicher, der PostgreSQL zur Verfügung steht handelt, sondern um den Teil der für aktive Operationen zuständig ist, sprich für das Cachen von Daten. Die Größe sollte ein Bruchteil des gesamten Arbeitsspeichers sein, da davon auszugehen ist, dass der Datenbank Server auch vom Plattencache Gebrauch macht. Bei Systemen mit einem Gigabyte an RAM oder mehr ist es zu empfehlen ein Viertel des gesamten Arbeitsspeicher für den *shared_buffers* Parameter zu verwenden. Bei weniger Arbeitsspeicher liegt der Anteil bei etwa 15 Prozent [wik08d]. Zu beachchten ist daher, dass ein zu hoher Wert für den *shared_buffers* Parameter zu weniger Performance führen kann. Dies liegt unter anderem daran, dass durch die damit verbundene Grösse des Caches mehr Zeit für das Scannen des Inhalts benötigt wird [ber05].

Der nächste Parameter im Bezug auf Speicherverwaltung ist die Größe des effektiven Caches, genannt *effective_cache_size*. Als Standardwert wird dabei meist auf die Hälfte des zur Verfügung stehenden Speichers gesetzt. Betrachtet man den Zusammenhang zum System so kann ein durchaus effektiverer Wert gewählt werden. Dieser sollte nach folgendem Prinzip abgeschätzt werden. Die effektive Größe des Caches ist jene Größe des Arbeitsspeichers, die übrig bleibt, wenn man den Verbrauch durch Betriebssystem, PostgreSQL Server und andere Applikationen, falls vorhanden, abzieht [wik08d]. Weiters besteht ein Zusammenhang zum Verhalten des Query Planners von PostgreSQL. Laut Josh Berkus sagt die *effective_cache_size* aus wie groß das grösste, zu erwartende Datenbank Objekt, welches gecached werden kann, sein könnte [ber05]. Dies hat zur Folge, dass bei einem zu niedrigen Wert, bestimmte Indizes falsch oder gar nicht verwendet werden [wik08d].

Ein weiterer wichtiger Parameter nennt sich *work_mem*. Grob zusammengefasst geht es dabei um die maximal mögliche Verwendung von Speicher für jeweilige Sorts, Aggregates oder ähnliche Operationen. Ziel ist es somit eine obere Grenze zu setzen um nicht einen Engpass an verfügbarem Speicher zu produzieren. Wichtig ist auch hier wiederrum der Zusammenhang zum erstgenannten Parameter *max_connections*; für jede Operation die beispielsweise einen Sort beansprucht, wird die festgelegte Menge an *work_mem* reserviert. Dies hat zur Folge, dass bei einem *work_mem*-Wert von 50 Megabyte und 30 Benutzern mit auszuführenden Transaktionen, es im schlechtesten Fall zu 50 mal 30, sprich 1500 Megabyte an benötigtem Speicher kommt. Daher muss dieser Parameter unbedingt im Zusammenhang mit den maximal simultan möglichen Verbindungen abgestimmt werden, um nicht unnötige Speicherengpässe zu generieren. Werden in der Datenbankumgebung somit hauptsächlich komplexe Sortier- oder Aggregatoperationen getätigt, sollte eindeutig in genügend Speicher investiert werden um den Parameter *work_mem* maximieren zu können [wik08d].

Der letzte Parameter, der im Zuge dieses Abschnittes einen direkten Bezug zur Speicherverwaltung hat, ist der *maintenance_work_mem* Parameter.

Dieser gibt an wieviel RAM für VACUUM, ANALYZE und Indexerstellungen in Anspruch genommen wird. Je grösser also die Tabellen sind, desto höher sollte man diesen Wert setzen um die Effizienz von Wartungsoperationen, im speziellen VACUUM zu optimieren. Josh Berkus empfiehlt beispielsweise 50 bis 75 Prozent der Grösse der grössten Tabelle für diesen Wert einzukalkulieren. Eine weitere Option ist, während dem Nachtbetrieb diesen Wert zu maximieren, und Wartungsarbeiten somit auf ein zugriffsarmes Zeitfenster zu verschieben. Dennoch lässt sich dies nicht für alle Server arrangieren aufgrund von permanentem Dauerbetrieb. Somit empfiehlt es sich hier im Verhältnis zur Grösse der Datenbank im Bereich zwischen 32 und 256 Megabyte zu bleiben [ber05].

Ein Parameter der im Bezug auf Abfrageoptimierung zusammen mit der *effective_cache_size* eine wichtige Rolle spielt, heisst *random_page_cost*. Dabei geht es um die Einschätzung der zeitlichen Inanspruchnahme für einen nicht sequentiellen Speicherzugriff. Im Grunde sollte dieser Parameter bei Standardumgebungen nicht verändert werden, trotzdem sollte man sich bei hochspezialisierten Datenbankumgebungen Gedanken über diesen Parameter machen [ber08]. Grundsätzlich kann folgende Faustregel angenommen werden. Werden sequentielle Scans im Gegensatz zu Index Scans bevorzugt, sollte man den Wert verringern. Werden jedoch langsame Indizes verwendet, in Situationen in denen der Query Planner dies eigentlich nicht tun sollte, empfiehlt es sich den *random_page_cost*-Wert zu erhöhen [ber05]. Allgemein ist in diesem Bereich mit dem Ziel zu optimieren, dass möglichst oft ein Index Scan in Anspruch genommen wird. Ist daher eine leistungsfähige Infrastruktur in Form von optimalen RAID Verbunden oder Ähnlichem vorhanden, können die Kosten für einen nicht sequentiellen Speicherzugriff gesenkt werden, um somit die Nutzung von Indizes zu maximieren [wik08d].

Als letzter Parameter für die Optimierung von Datenbanksystemen, insbesondere von industriellem Umfang, ist das Verzeichnis für Log Dateien (*log_directory*), sowie der oben erwähnte Transaktionslog (*pg_xlog*) zu nennen. Die Auslagerung dieser auf externe physikalische Plattenspeicher kann zu deutlichen Performancesteigerungen führen, welche in Punkt 3.2.2 bereits

näher erläutert wurden.

3.4 Design Tuning

Unter Design Tuning ist all das zu verstehen, was sich nicht in die Hardware- oder Parameterebene einstufen lässt. Wie auch bei den anderen Ebenen gibt es zahlreiche Möglichkeiten und Ansichten zur Optimierung, dennoch ist diese Ebene am weitläufigsten. Es soll somit im folgenden Abschnitt mit unter eine Grundlage für allgemeine Vorgehensweisen und Praktiken geschaffen werden, die eine Performancesteigerung insbesondere im Transaktionsbreich von PostgreSQL herbeiführen können. Die Auswahl ist ausserdem durch die Anforderungen des Instituts für Informationswirtschaft dahingehend beeinflusst, dass eine Optimierung der Volltextsuche[5] sowie von zeitintensiven, insbesondere JOIN-intensiven, Transaktionen benötigt wird. Somit soll grundsätzlich die Option der Denormalisation und die Optimierung von Indizen behandelt werden. Diese Aufteilung von Schema Tuning und Index Tuning wurde auch von Dennis Shasha und Philippe Bonnet in ihrem Buch *"Database Tuning" [SB03]* aufgegriffen, was als grundlegende Orientierung dient.

3.4.1 Schema Tuning

Anhand eines sehr einfachen Lieferanten-Bestellung Schemas lassen sich grundlegende Optionen und Szenarien für eine Performanceoptimierung darstellen. Das in Abbildung 3.1 dargestellte Schema soll unter anderem aufzeigen, wie bei Designentscheidungen das fünfte Prinzip aus Punkt 3.1 zur Geltung kommt: *"Be prepared for trade-offs"*.

Geht man davon aus, dass es 100.000 Bestellungen gibt und 2000 Lieferanten; für die Lieferanten_ID ein acht-byte Integer und für die Lieferantenaddresse 50 Byte in Anspruch genommen werden können, so kann man

[5] tsearch2 - ein seit PostgreSQL 8.3 integriertes Volltext-Suchmodul [Pos08]

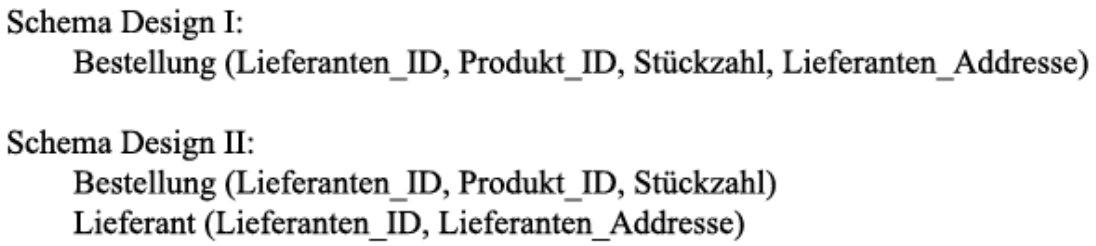

Abbildung 3.1: Lieferanten-Bestellung Schema [SB03]

beide Schemen anhand von drei klaren Punkten bewerten. Der erste befasst sich mit dem Speicherproblem. Das zweite Schema verwendet redundante Lieferanten_ID's dadurch dass in beiden Tabellen die ID abgespeichert werden muss um die Verknüpfung zu erhalten. Dies hat zur Folge dass 2000 x 8 = 16.000 Byte an zusätzlichem Speicher benötigt wird. Trotzdem schafft es das zweite Schema Platz zu sparen. Es müssen aufgrund des Schema Designs nur 2000 Lieferanten Addressen gespeichert werden und nicht wie im ersten Schema 100.000; was bedeutet, dass somit 98.000 x 50 = 4.950.000 Bytes weniger an Addresseinformationen verschwendet werden müssen. Im Endeffekt können in einem solchen Szenario etwa fünf Megabyte eingespart werden. Erhöht man die Tupelanzahl der Tabellen um das tausendfache so kommt es schon zu erheblicheren fünf Gigabyte [SB03].

Als zweiten Punkt im Vergleich der beiden Schemen kann man die Informationshaltung heranziehen. Wenn man davon ausgeht, dass zum Beispiel nach einer erfolgreichen Lieferung die Bestellung, aus was für Gründen auch immer, gelöscht werden soll, so würde man im ersten Schema alle Addressdaten verlieren. Dieses klassische Beispiel einer Lösch-Anomalie spricht eindeutig für das zweite Schema [EN05].

Als dritten Punkt, bezüglich des Vergleichs beider Schemen, kommen wir zum Aspekt der Performance. Werden in bestimmten Abfragen beispielsweise Daten abgefragt die auf beide Tabellen verteilt sind so ist das erste Schema eindeutig vorzuziehen. Der Grund dafür liegt in der Vermeidung der Kosten eines JOINs. Dennoch ergibt sich auch in dieser Hinsicht ein

weiterer Vorteil für das zweite Schema, nämlich im Falle einer INSERT-intensiven Datenbank. Müssen immer wieder neue Bestellungen aufgenommen werden so kommt es im Falle des ersten Schemas zu einem Overhead an zu schreibender Information. Es muss für jeden Eintrag die Addresse mit eingetragen werden. Diese offensichtliche Redundanz kann zu erheblichen Datenmengen führen, die mithilfe des zweiten Schemas umgangen werden können. Man sieht also, obwohl beide Schemen das selbe abbilden, verfügen sie dennoch über unterschiedliche Eigenschaften bezüglich der Performance, welche erhebliche Auswirkungen auf mögliche Abläufe in der Datenbank haben [SB03].

Was desweiteren durch die Gegenüberstellung beider Schemen dargestellt wird, ist die Unterscheidung zwischen Normalisierung und Denormalisierung. Durch Denormalisierung kann an Performance gewonnen werden, jedoch auf Kosten redundanter Daten, was INSERT und UPDATE Transaktionen wiederum verlangsamt. In Situationen in denen zahlreiche Abfragen über mehrere Tabellen getätigt werden, muss man sich somit überlegen inwieweit es möglich ist hohe INSERT und UPDATE Kosten in Kauf zu nehmen, um schnellere Abfragen zu erhalten. Je nach Zielsetzung fällt die Entscheidung auf Normalisierung oder Denormalisierung [SB03].

3.4.2 Index Tuning

Ein weiterer Aspekt, im Bezug auf Performanceoptimierung für Datenbanktransaktionen, ist das Setzen von Indizes. Ein Index stellt eine organisierte Datenstruktur dar die es spezifischen Abfragen ermöglicht effizienter bestimmte Werte zu lokalisieren. Es gibt Szenarien bei denen es sich lohnt einen Index zu setzen und Szenarien bei denen es eher kontraproduktiv sein kann. Diese Situationen gilt es mittels genauer Überlegungen, bezüglich der Sinnhaftigkeit des jeweiligen Indizes selbst, zu eliminieren. Ein solche Sinnhaftigkeit geht insbesondere dann verloren, wenn ein Index auf ein bestimmtes Attribut gesetzt wird jedoch dieses eine Attribut in keiner Abfrage enthalten ist. In solch einem trivial klingenden Szenario führt ein Index zu

mehr Overhead als Nutzen. Dieser Overhead kommt dadurch zu Stande, dass bei INSERT, UPDATE und DELETE Abfragen der Index überprüft werden muss, inwieweit die Manipulation der Daten eine Neuberechnung oder Ähnliches zur Folge hat [SB03]. Allgemein bieten die meisten gängigen Datenbanksysteme mindestens zwei Datenstrukturen an; die B-Tree und Hash Struktur. PostgreSQL bietet darüber hinaus speziell für Volltextsuche angepasste Indizes an, genannt GiST und GIN [Pos08]. Im Folgenden Abschnitt soll eine kurze Evaluierung dieser vier Indextypen erfolgen um schlussfolgernd Empfehlungen zur Optimierung zu geben.

B-Tree Index

Ein B-Tree, zu Deutsch auch B-Baum genannt, ist nicht gleichzusetzen mit einem binären Baum. Ein B-Baum hat im Gegensatz zu einem Binärbaum mehrere sortierte Verweise auf Kindknoten. Ein Binärbaum hingegen höchstens nur zwei [wik08a]. Eine weit verbreitete Variante unter führenden Datenbankherstellern ist die Verwendung von B+Trees. Diese unterscheiden sich von normalen B-Trees dadurch, dass die inneren Knoten Schlüssel beinhalten und nur die Blattknoten Verweise auf Daten [SB03]. Inwieweit PostgreSQL Verwendung von der B+Tree Datenstruktur macht, ist aus der offiziellen Dokumentation nicht ersichtlich, daher soll im weiteren Verlauf auf die vorher angesprochenen B-Trees eingegangen werden [Pos08].

Laut Dennis Shasha und Philippe Bonnet sind B-Trees die beste Option in der Wahl der Datenstruktur. Dies hat zwei Gründe. Erstens ist eine Reorganisation des Baumes durch Datenmanipulation seltener der Fall, und zweitens ist diese Datenstruktur für mehrere Abfragen sehr gut geeignet. So bieten sich B-Trees für Abfragen über einen bestimmten Wertebereich sowie für Abfragen auf Maximum- oder Minimumwerte bestens an [SB03].

Hash Index

"Die Datenstruktur eines Hash Index basiert auf einer Hash-Funktion, mit deren Hilfe aus einem Schlüssel die Adresse des zugehörigen Datensatzes errechnet wird [HN05]." Der grundlegende Unterschied zu einem B-Tree liegt hierbei im Vergleich der Bedeutung des Wortes *"Nähe"*. Zwei Werte liegen in einem B-Tree nah bei einander, wenn sie in ihrer sortierten Reihenfolge nah bei einander liegen; beispielsweise 50 und 51 oder Smith und Sneed. Bei Hash-strukturen liegen zwei Werte nur dann nah bei einander, wenn sie identisch sind. In allen anderen Fällen wird durch die Verwendung der Hash-Funktion eine gewisse Zufallsverteilung der Daten herbeigeführt, dadurch dass ein Schlüssel mit seiner Hash-Funktion die Lokalisierung der Daten erlaubt kommt, es zu keinen Zusammenhängen zu umliegenden Daten. Der Zusammenhang wird durch die Hash-Funktion und den Schlüssel gegeben [SB03].

Ein Hash Index ist gut geeignet für Abfragen die gezielte Daten anfordern. Bei einer klassischen Abfrage die beispielsweise den Angestellten mit der Sozialversicherungsnummer 2304-1985 sucht, ist eine Hash Struktur eindeutig zu bevorzugen. Außerdem eignen sie sich auch gut für Abfragen die mehr als nur einen Wert aufgrund eines Vergleichs zurückliefern; zum Beispiel alle Angestellten in der Entwicklungsabteilung.

GiST und GIN Index

GiST steht für *"Generalized Search-Tree"* und GIN für *"Generalized Inverted Index"*. Beide Indizes werden spezifisch für die Volltextsuche eingesetzt und bieten im Vergleich zueinander vier grundlegende Merkmale im Bezug auf Performance. Vorerst soll jedoch kurz auf die Eigenschaften der jeweiligen Strukturen eingegangen werden. Der GiST Index ist ein Index, der mit einem gewissen Verlustfaktor behaftet ist. Dies bedeutet, dass der Index bei einer Volltextsuchanfrage möglicherweise falsche Treffer zurück

liefert, was jedoch von PostgreSQL durch eine zusätzliche Überprüfung behandelt und somit eliminiert wird. Durch diese Eigenschaft entstehen vorerst klare Nachteile für die Wahl eines GiST Indexes, dadurch dass die doppelte Überprüfung eine klare Leistungseinbusse mit sich bringen kann. GIN Indizes hingegen sind vorerst nicht verlustbehaftet, weisen jedoch, im Bezug auf die Anzahl der eindeutigen Wörter im Text, einen logarithmischen Zusammenhang zur Performance auf. Werden bei Volltextsuchanfragen zusätzliche Gewichtungen mit einbezogen, so kommt es auch bei GIN Indizen zu verlustbehafteten Ergebnissen, die nochmals überprüft werden müssen [Pos08]. Beachtet man diese Eigenschaften lassen sich laut dem *"offiziellen PostgreSQL Handbuch"* [Pos08] vier performancebezogene Aussagen treffen:

- Ein GIN Index ist in etwa dreimal so schnell wie ein GiST Index
- Ein GIN Index benötigt dreimal länger zum erstellen als ein GiST Index
- Ein GIN Index benötigt zehnmal länger zum aktualisieren als ein GiST Index
- Ein GIN Index ist zwei bis dreimal so groß wie ein GiST Index

Was aus dieser Aufstellung klar ersichtlich wird ist, dass ein GIN Index eindeutig schneller als ein GiST Index ist, jedoch im Bezug auf dynamische Daten, eher schlecht abschneidet was auf die hohen Zeitansprüche bei Initialisierungen und Aktualisierungen zurückzuführen ist. Somit kann man einen GIN Index bei statischen Datenbeständen aufgrund der Schnelligkeit empfehlen. Im Grunde kommt es hier zu einem Kompromiss zwischen Suchgeschwindigkeit, im Bezug auf statische Umgebungen und Aktualisierbarkeit, im Bezug auf dynamische Umgebungen. Je nach Anforderungen muss eine Entscheidung getroffen werden, die auch eindeutige Nachteile mit sich bringt.

3.5 Schlussfolgerungen

Wie schon am Anfang dieses Kapitels unter Punkt 3.1 erwähnt, gibt es zahlreiche Aspekte, die sich grob über drei Ebenen erstrecken: Hardware, Parameter und Design. Jeder dieser Punkte bietet eine Fülle an Informationen und Möglichkeiten der Optimierung, bezogen auf die Performance einer Datenbank. Ein Versuch, alle Optionen zu evaluieren, würde den Rahmen dieser Arbeit eindeutig sprengen, daher konnte nur auf gezielte Punkte eingegangen werden. Es muss hier angemerkt werden, dass es somit noch um einiges mehr Optionen geben würde. Dennoch bieten die hier vorgestellten Punkte einen Spielraum, um vorhandenen Engpässen in einem ersten Schritt entgegen zu wirken. Grundsätzlich bieten die fünf Prinzipien des Tunings eine logische Grundlage zur Herangehensweise an Datenbankprobleme. Im nächsten Schritt sollte ganz klar beachtet werden, dass es in jeder Situation zu unterschiedlichen Lösungswegen kommen kann. Sowohl auf Hardware-, Parameter- als auch Designebene kann durch eine erste Analyse des Problems und der Umgebung meist mehr geleistet werden als durch eine Anwendung starrer Regeln im Sinne eines so genannten *"Tuningkatalogs"*. Dennis Shasha und Philippe Bonnet vergleichen die Tätigkeit der Optimierung einer Datenbank mit der eines Therapeuten der das Problem hinter dem Problem zu lösen versucht: *"[...] this means sitting with the designer and talking the whole thing through, a little like a therapist trying to solve the problem behind the problem [SB03]."* Somit kann schlussfolgernd zu diesem Kapitel keine allgemeine Empfehlung abgegeben werden. Im Bezug auf die Implementation im folgenden Kapitel sind jedoch klare Rahmenbedingungen geschaffen, um systematisch eine erfolgreiche Optimierung von groß angelegten Datenbanken, ihren Transaktionen und den spezifischen Volltextindexierungen vorzunehmen.

4 Implementierung

Inhalt dieses Kapitels ist die Dokumentation und Erläuterung der Implementierung von *"pgperformer"*, einem Tool, zur Durchführung von gezielten Testcases für postgreSQL Datenbanken. Im ersten Abschnitt wird eine fundierte Projektübersicht gegeben; darauf folgend soll auf wichtige Elemente des Programms näher eingegangen werden, um schlussfolgernd projektspezifische Optimierungsvorschläge zu erläutern. Ziel soll somit neben einer klaren Dokumentation der Software, die Schaffung einer Basis für eine problemlose Weiterentwicklung des Tools sein.

4.1 Projekt

Das Institut für Informationswirtschaft ist an mehreren Projekten wie zum Beispiel AVALON, IDIOM oder RAVEN beteiligt. Forschungsinhalt dieser Projekte ist im Allgemeinen die Suche, Sammlung und Analyse von Daten aus dem Internet. Als zentraler Punkt der Infrastruktur dieser Projekte, ist die zugrunde liegende Datenbank zu nennen. Bei datenintensiven Projekten, wie es jene des Instituts für Informationswirtschaft sind, kommt es somit zwangsläufig zu Engpässen. Ein wichtiges Ziel des Instituts ist es daher, diesen Engpässen entgegenzuwirken. Dies ist insbesondere dann von enormer Wichtigkeit, wenn es darum geht, dem Endbenutzer möglichst schnell und effizient Daten zu präsentieren.

Aus diesen Umständen heraus ergibt sich einerseits, die Nachfrage nach möglichen Optionen zur Optimierung von Datenbanken und andererseits,

die Nachfrage nach einem Tool zur Analyse und Auswertung von bestimmten Szenarien, die Engpässe darstellen könnten. Diese Arbeit stellt somit einen ersten gezielten Schritt hin zur Analyse von Optimierungsmöglichkeiten von Datenbanken und Volltext Indexierungen für groß angelegte Datenbankumgebungen dar. Im Spezifischen wird dabei der Fokus auf die Open Source Datenbank postgreSQL, im Zusammenhang mit dem dabei integrierten Modul Tsearch, gelegt. Als Entwicklungsumgebung wurde eine reine Javaumgebung, in Form von Eclipse, gewählt.

Das Projekt ist in zwei unterschiedliche Bereiche zu unterteilen. Einerseits die theoretische Aufarbeitung des Themengebiets wie es in Punkt 2 und 3 der Fall ist; andererseits die Implementierung eines Tools zur Durchführung und Analyse von angepassten Testcases. Das Programm soll dabei Möglichkeiten bieten, um einfache Benchmarks, aber auch manuell angepasste Szenarien, laufen zu lassen, um in späterer Folge Analysedaten zu erhalten. Diese Analysedaten sollen in Form von graphischen als auch textuellen Daten verfügbar gemacht werden. Desweiteren soll das seit postgreSQL 8.3 integrierte Modul Tsearch, welches aktuell auf Institutsebene verwendet wird, getestet werden können. Dabei liegt der Schwerpunkt auf dem Vergleich zwischen einer Tsearch-Abfrage inklusive Gewichtung und derselben ohne Gewichtung. Im Zuge des Projekts sollen darüber hinaus Optimierungsvorschläge für spezielle, am Institut verwendete, JOIN-Abfragen präsentiert werden.

Zielsetzung ist es somit ein Programm zur Verfügung zu stellen, mithilfe dessen (I) Zusammenhänge zwischen Datenbank und Einstellungen, wie sie in Punkt 3 dargestellt wurden, und (II) die Performance von unterschiedlichen SQL Abfragen registriert werden können.

4.2 Programmbeschreibung

Folgender Abschnitt soll eine detaillierte Übersicht über die kompletten Funktionalitäten, wie sie in Punkt 4.1 grob angeführt wurden, bieten.

4.2.1 Systemübersicht

Anhand von Abbildung 4.1 werden die zur Verfügung gestellten Funktionalitäten in Verbindung mit Use Cases dargestellt.

Abbildung 4.1: Use Cases

Als Endbenutzer hat man zwei Möglichkeiten einen Test durchzuführen. Unter *Generic Benchmarks* befinden sich all jene Szenarien, die allgemein anwendbar sind und den Prinzipien, wie sie unter Punkt 2 behandelt wurden, entsprechen. Darunter fallen typische INSERT, SELECT und UPDATE Testcases, welche jeweils unabhängig voneinander durchgeführt werden müssen. Der VIEW Testcase ist durch einen Vergleich zwischen einer herkömmlichen SELECT Abfrage, und der identischen Abfrage, implementiert als VIEW, charakterisiert. Wichtig ist dabei anzumerken, dass die INSERT, SELECT und UPDATE Testcases einerseits als normale Transaktionen ausgeführt werden können, aber auch in Form von *Prepared Statements* und *Stored Procedures* implementiert sind. Grundsätzlich bieten somit die genannten Testcases die Möglichkeit eines Vergleichs zwischen verschiedenen Datenbankeinstellungen, aber auch den Vergleich zwischen unterschiedlichen Datenbanken. Es könnte somit ein Vergleich zwischen einer postgreSQL Datenbank und einer mySQL Datenbank, unter definierten Einstellungen anhand der normalen Testcases, exklusive aller postgreSQL-spezifischen Erweiterungen, durchgeführt werden.

Als zweite Möglichkeit, kann der Endbenutzer einen *Custom Benchmark* auswählen. Darunter fallen einerseits der Tsearch Testcase sowie der Custom Testcase. Was beim Tsearch Testcase zu beachten ist, ist dass eine zusätzliche Wörterliste mit zu verwendenden Suchbegriffen zu definieren ist. Mittels eines Custom Testcases kann der Endbenutzer seine eigene Datenbank und seine eigenen SQL Abfragen mit dem Programm verbinden, um Analysedaten zu erhalten.

Weiters stehen dem Benutzer Funktionalitäten zur Verfügung, um die gesammelten Daten zu verwalten. Dies bedeutet einerseits das Exportieren von Analysedaten sowie das Löschen von temporären Dateien, welche während einem Testcase erstellt wurden.

Um sich ein besseres Bild der Implementierung machen zu können soll vorerst ein grober Überblick über die Systemarchitektur gegeben werden. Dies wird in Abbildung 4.2 anhand von Schichten dargestellt. Grundsätzlich ist zwischen drei Ebenen zu unterscheiden: dem *"Performance View Layer"*,

dem *"Threadable Layer"* und dem *"Statistics / Utilities Control Layer"*. Der *"Performance View Layer"* stellt die Schnittstelle zum Endbenutzer dar und behandelt alle graphischen Ein- und Ausgabefunktionen. Der *"Threadable Layer"* ist durch den *"Database Control Layer"* und den *"Testcase Control Layer"* definiert. Durch diese Darstellung soll auf die Vorgehensweise der Implementierung eines Testcases hingewiesen werden, worauf später noch im Detail eingegangen wird. Was vorerst von Bedeutung ist, ist die Konstellation des *"Threadable Layers"*, welcher den Grundbaustein für alle Testcases darstellt.

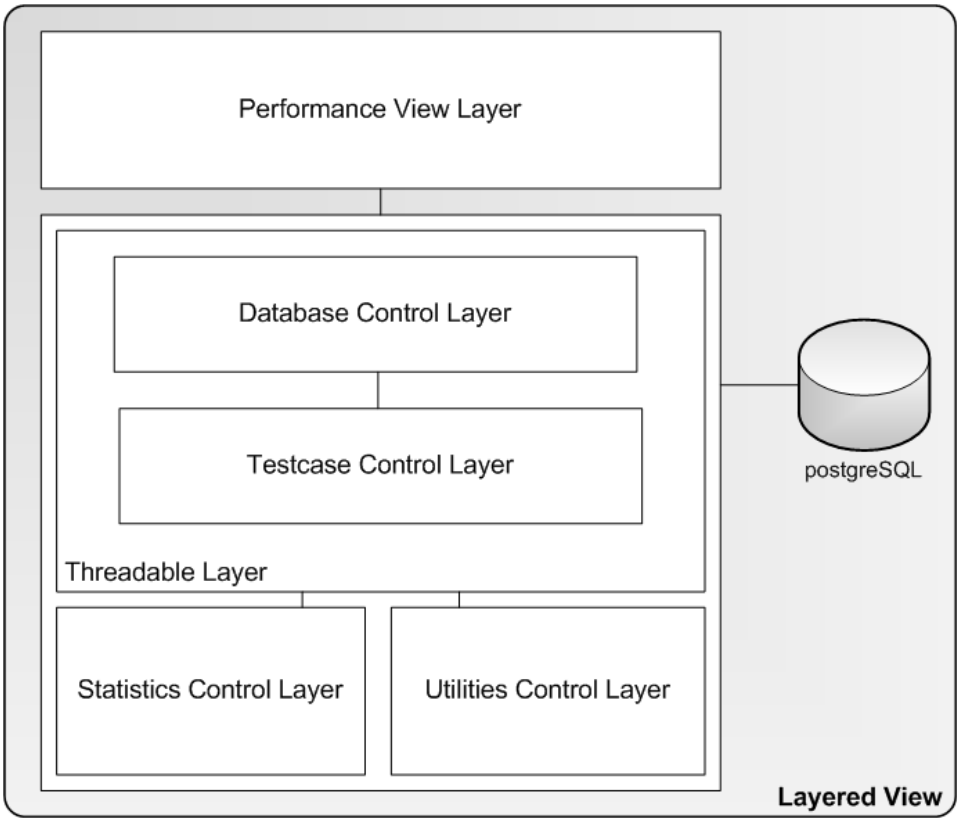

Abbildung 4.2: Systemarchitektur

Der *"Threadable Layer"* ist in weiterer Folge mit dem *"Statistics / Utilities Control Layer"* verbunden und wird dazu verwendet um nicht-persistente als auch persistente Daten, in Verbindung mit einer Statistik Datenbank, zu

verwalten. Grundfunktionalität dieser Ebene ist somit die Sammlung und Verwaltung von Statistikdaten, sowie die Bereitstellung von Werkzeugen zur Messung von SQL Abfragen.

4.2.2 Datenbank

Um die programmiertechnische Implementierung zu verstehen, soll nun kurz auf die zugrunde liegenden Datenbankmodelle der verschiedenen Testcase Kategorien eingegangen werden. Prinzipiell gibt es drei verschiedene Strukturen die zu beachten sind. Erstens, das Schema der einfachen generischen Benchmarks; zweitens, die Testtabelle bezüglich der Tsearch Testcases und drittens, das externe Testschema, welches spezifisch für eine JOIN-Abfrageoptimierung angewendet wird und dabei alle relevanten Tabellen beinhaltet.

Generisches Datenbankmodell

In Abbildung 4.3 ist die zugrunde liegende Datenbank für generische Benchmarks dargestellt. Ziel bei dem Entwurf dieser Datenbank war es ein möglichst leicht verständliches und skalierbares Modell zu entwerfen, um zielgerichtete Lasttests durchführen zu können.

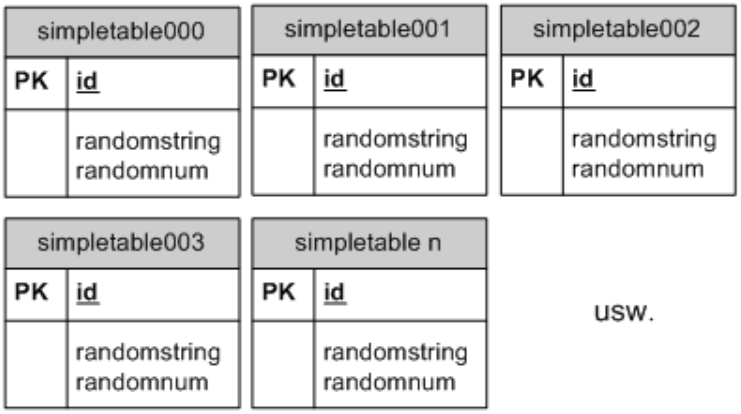

Abbildung 4.3: Generisches Datenbankmodell

Die Struktur, wie sie oben abgebildet ist, basiert auf einer Tabelle mit drei Spalten. Die erste Spalte, genannt *"id"*, ist durch einen Integer-Wert definiert, welcher als fortlaufende Sequenz geführt wird und als Primärschlüssel deklariert ist. Die Spalten *"randomstring"* und *"randomnum"* sind vom Typ Character und Double Precision. Basierend auf diesem Schema können beliebig viele Tabellen mit beliebig vielen Tupeln erstellt werden. Es kann somit eine gewünschte Datenbankgröße erreicht werden, um anhand dieser bestimmte generische Benchmarks durchzuführen.

Tsearch Datenbankmodell

Die zugrunde liegende Datenbank für die Tsearch Testcases ist durch eine einzige Tabelle, wie sie in Abbildung 4.4 zu sehen ist, definiert. Diese beinhaltet alle Informationen um Tsearch Abfragen auszuführen.

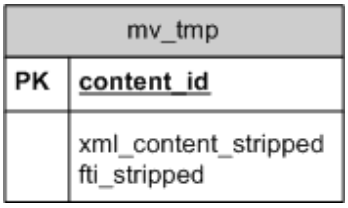

Abbildung 4.4: Tsearch Datenbankmodell

Es ist an dieser Stelle anzumerken, dass die dargestellte Tabelle Teil einer viel größeren Datenbankumgebung ist. Für Testzwecke wurde diese extrahiert, um es zu ermöglichen gezielte Tsearch Abfragen zu tätigen, auf die im weiteren Verlauf noch näher eingegangen wird. Inhaltlich relevant sind dabei die beiden Spalten *"xml_content_stripped"*, vom Typ Text und *"fti_stripped"*, vom Typ Tsvector. Ein Tsvector ist eine sortierte Liste eindeutiger Lexeme, welche im vorhandenen Text (in diesem Fall *"xml_content_stripped"*) vorkommen; dabei wird jene Liste automatisch beim Setzen des GiST Index generiert [Pos08].

Externes Datenbankmodell

Abbildung 4.5 zeigt das zugrunde liegende Schema der externen Datenbank, welches an die Institutsumgebung angelehnt ist. Die Namen der Tabellen sind dabei auf einzelne Buchstaben reduziert.

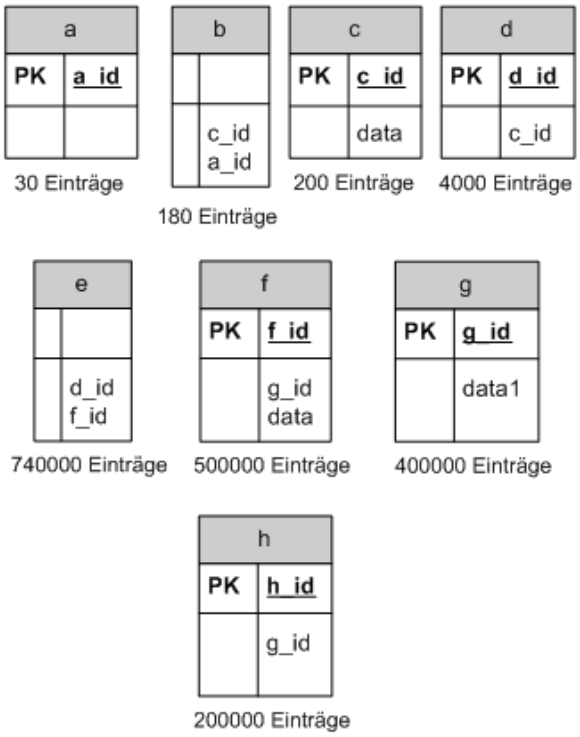

Abbildung 4.5: Externes Datenbankmodell

Die Struktur besteht aus insgesamt acht Tabellen. Um die Komplexität zu reduzieren, wurden nur jene Spalten übernommen, die relevant für die JOIN-Abfrage sind. Wie aus der Abbildung zu sehen ist, bestehen die Tabellen somit nur aus Spalten mit einer jeweiligen *"id"* sowie teilweise einer zusätzlichen Spalte *"data"*. Die *"id"*-Spalten sind dabei entweder als Primärschlüssel oder Fremdschlüssel deklariert. Zusätzlich zu der Struktur sind die einzelnen Tabellen mit einer gewissen Menge an Daten befüllt und repräsentieren dabei die Institutsdatenbank in einem Verhältnis von einem Hundertstel. Auf die Struktur und Optimierung der Abfragen wird im Verlauf

dieses Kapitels noch näher eingegangen.

4.2.3 Programmierung

Bis jetzt wurden allgemeine Darstellungsmethoden verwendet, um einen groben Überblick über die verwendeten Datenbankstrukturen und das System zu geben. Abbildung 4.6 geht einen Schritt weiter und stellt alle involvierten Klassen und ihre Beziehungen zueinander in grundlegender Form dar. In weiterer Folge sollen elementare Teile des Programmcodes anhand einer detaillierten Systemübersicht näher erläutert werden. Dabei werden folgende Aspekte beleuchtet:

- Generieren von Testdaten
- Initialisierung des *Threadable Layers*
- Inhalt der run() Methode der DataBase Klasse
- Durchführung eines Testcases
- Ausgabe mittels eines Graphen
- Ermittlung der *transactions per second*

Generieren von Testdaten

Um einen Testcase durchführen zu können, müssen Testdaten vorhanden sein. Diese Testdaten werden bei einem generischen Benchmark automatisch generiert und bei allen anderen Szenarien durch zusätzliche Dateien über den FileLoader eingespielt. Ausgangspunkt der Initiierung ist die DataBase Klasse, welche mittels eines SetupWorker Threads aufgerufen wird, um die Erstellung der relevanten Zufallsdaten vorzunehmen. Anhand eines Codebeispiels für die Generierung eines *SELECT Testcases* soll dies veranschaulicht werden. Die Wahl fällt hierbei auf ein verhältnismässig einfaches Beispiel, um die prinzipielle Vorgehensweise besser zu veranschaulichen. Wie in Abbildung 4.6 zu sehen ist, sind mehrere Klassen involviert.

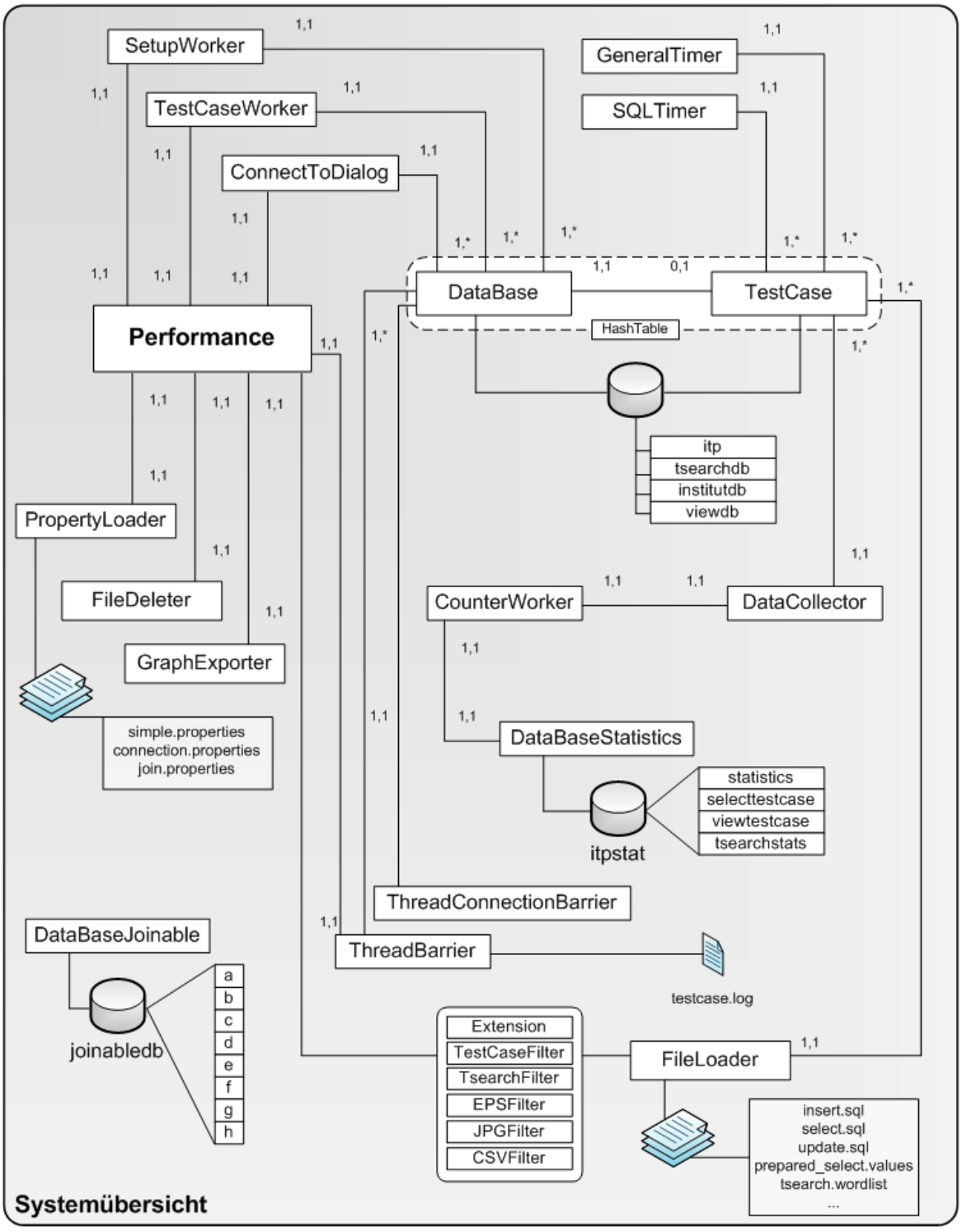

Abbildung 4.6: Systemübersicht mit allen Klassen

Ausgangspunkt ist die Performance Klasse, welche durch den Input eines Benutzers den SetupWorker Thread startet. Dieser hat alle benötigten Parameter übergeben bekommen, um in weiterer Folge die DataBase Klasse aufzurufen, welche durch einen *switch* die zugehörige Methode aufruft. Im Endeffelt wird in diesem Fall die Methode für den Setup eines einfachen SELECT Testcases aufgerufen, welche wie folgt aufgebaut ist:

```
1   public void setupSelectTestcase(JTextArea setup, JProgressBar progressbar)
        {
2       try {
3           progress = 0;
4           rows = getRowCountSTRING(tables, prefix);
5           BufferedWriter out = new BufferedWriter(new FileWriter("select.sql",
                true));
6           System.out.println("Initializing randomValues");
7           for (int x = 0; x<select; x++) {
8               randomTable = rg.getRandomTableNR(tables);
9               first = String.valueOf((Integer.valueOf(randomTable).intValue()*
                    Integer.valueOf(rows).intValue()));
10              out.write("SELECT randomstring FROM "+prefix+"table"+randomTable
                    +" WHERE ID = "+rg.getRandomID(first, rows)+";\n");
11              progress++;
12              progressbar.setValue(progress);
13          }
14          setup.setText("");
15          setup.append("SELECT randomstring FROM "+prefix+"table"+randomTable+
                " WHERE ID = "+rg.getRandomID(first, rows)+";");
16          System.err.println("Setup complete!");
17          out.close();
18      } catch (Exception e) {
19      }
20  }
```

Es wird vorerst die Anzahl der Tupel der ersten Tabelle[1] in der Datenbank abgefragt (Zeile 4). Basierend auf dieser Zahl, sowie der berechneten ersten ID einer jeweiligen Zufallstabelle (Zeile 8) wird eine gültige Zufalls-ID berechnet (Zeile 10), die für diese zuvor ausgewählte Tabelle zutrifft und somit auch ein gültiges Ergebnis zurück liefern wird. Diese Schritte werden nun

[1] Aufgrund der Tabellenstruktur, wie sie in Punkt 4.2.2 dargestellt wurde, haben alle Tabellen eine identische Anzahl an Tupel.

so oft wiederholt, wie sie in der *select*-Variable, welche aus der Property Datei[2] gelesen wurde, definiert sind. Als Endergebnis ist nun eine fertig befüllte Datei genannt "select.sql" vorhanden, welche für den eigentlichen Testcase verwendet werden kann.

Initialisierung des Threadable Layers

Der *Threadable Layer* wird durch folgende for-Schleife initialisiert:

```
1 for (int i=1; i<=connectionsINT; i++) {
2   DataBase multiuserDB = new DataBase(i, testcase);
3   Thread multiuserThread = new Thread(multiuserDB);
4   multiuserThread.start();
5   threads.put(String.valueOf(i), multiuserDB);
6 }
```

Je nach Testcase wird ein unterschiedlicher Konstruktor für die *DataBase* Klasse verwendet. Dabei wird zusätzlich in den Fällen eines Tsearch oder VIEW Testcases der spezifische Datenbankname mit übergeben. Wie aus diesem Beispiel ersichtlich ist, kann die *DataBase* Klasse einerseits als normale Instanz behandelt werden, um wie oben angeführt, Zufallsdaten vorzubereiten, andererseits kann sie als Thread genutzt werden, um eine beliebige Mehrbenutzerumgebung zu simulieren. Wichtig ist dabei zu beachten, dass die *DataBase* Klasse jeweils eine zugehörige *TestCase* Klasse mit der eigenen Datenbankverbindung instantiert. Dies geschieht durch die Aktivierung des Threads in Zeile fünf, wodurch die *run()*-Methode gestartet wird, dessen Inhalt im nächsten Codebeispiel betrachet werden soll.

Inhalt der run() Methode der DataBase Klasse

```
1   public void run() {
2     try {
3       if (!connect()){
4         System.err.println("Thread: "+threadnumber+" NOT connected");
```

[2] simple.properties

```
5       } else {
6         System.out.println("Thread: "+threadnumber+" connected");
7       }
8       if (testcase == 4 || testcase == 5) {
9         initializePreparedStatements();
10      }
11      //wait for others to connect and generate optional data for testcases.
12      Performance.threadconnectionbarrier.waitForOthers();
13
14        //thread ends up here when its started
15        while (paused==true) {
16            Thread.sleep(1);
17        }
18      test(testcase);
19      //wait for other threads to finish
20      Performance.threadbarrier.waitForOthers();
21      } catch (Exception e) {
22        e.printStackTrace();
23      }
24  }
```

Was anhand dieses Ausschnitts verdeutlicht werden soll ist, dass der Thread in seinem Ausgangszustand, wie er im *Threadable Layer* aktiviert worden ist, nach erfolgreichem Verbindungsaufbau mittels eines *Barrier Wait*-Mechanismus' [bar08] dem Endbenutzer eine Rückmeldung bietet, ab dem alle Threads verbunden und bereit sind; sprich sich in der *while*-Schleife befinden. Erst durch eine vom Endbenutzer gesetzte Eingabe durch das Drücken des *"Run"*-Buttons wird der eigentliche Testcase gestartet. Dieser soll im nächsten Abschnitt genauer beleuchtet werden.

Durchführung eines Testcases

```
1   public void selectTestcase() {
2     try {
3       strings1 = null;
4       System.out.println("Starting Benchmark");
5       while (!(strings1 = FileLoader.getLine(1))[0].equals("endofFile")) {
6           db.executeStatement(strings1[0]);
7           DataCollector.increment();
```

```
8           DataCollector.incrementTransactions();
9         }
10      db.finalCommit();
11      System.out.println("finished");
12    } catch (Exception e) {
13      System.out.println(strings1[0]);
14      e.printStackTrace();
15    }
16  }
```

Für die Durchführung eines Testcases werden zusätzlich zur Testcase Klasse zwei statische Klassen *"FileLoader"* und *"DataCollector"* verwendet. Die FileLoader Klasse stellt Methoden zur Verfügung die Datei auszulesen. In diesem Fall wird jeweils eine Zeile, welche in Form eines Array's zurückgeliefert wird, ausgelesen. Durch die Methode *getLine(int lines)* der FileLoader Klasse können je nach Bedarf unterschiedlich viele Zeilen in einem Gang ausgelesen werden. Dies ist dann von Vorteil wenn die generierte Datei Zufallsdaten getrennt durch neue Zeilen speichert.

```
1   public static synchronized String[] getLine(int lines) throws IOException {
2     strings = new String[lines];
3     if ((strings[0] = in.readLine()) != null) {
4       for (int i = 1; i<lines; i++) {
5         strings[i] = in.readLine();
6       }
7     } else {
8       strings[0] = "endofFile";
9     }
10    return strings;
11  }
```

In weiterer Folge wird die SQL-Abfrage ausgeführt, wobei die DataCollector Klasse einerseits die gesamte Anzahl an Transaktionen mitzählt und gleichzeitig einen Zähler mitlaufen lässt, der pro Sekunde auf null gesetzt wird. Durch diesen Mechanismus werden einerseits die *"transactions per second"* und die Gesamtanzahl der Transaktionen festgehalten. Zum Schluss wird bei Bedarf ein COMMIT gesetzt, wobei die *finalCommit()* Methode überprüft ob die autocommit-Variable auf true oder false gesetzt wurde.

Ausgabe mittels eines Graphen

Bei der Darstellung der gesammelten Daten mittels eines Graphen, wird auf die Bibliothek JChart2D [Wes08], entwickelt von Achim Westermann, zurückgegriffen. Diese stellt einfache Methoden zur Verfügung„ um in wenigen Schritten einen Graphen mit dynamischen Daten zeichnen zu können. Das Hinzufügen von einem Punkt bestehend aus einer x und y Koordinate geschieht dabei mit dem einfachen Aufruf folgender Methode:

```
1  public static void addValue(double x, double y){
2    TracePoint2D point = new TracePoint2D(x, y);
3    trace.addPoint(point);
4  }
```

Ermittlung der transactions per second

Die Kennzahl *transactions per second (tps)* wurde schon in Punkt 2 ausführlich erklärt. Folgender Codeausschnitt soll verdeutlichen wie dies mittels des Threads *"CounterWorker"* implementiert wird.

```
1  while (paused!=true) {
2    try {
3      Performance.addValue((double)DataCollector.incrementTime(), (double)
           DataCollector.getTransactions());
4      DataBaseStatistics.insertStatSet(DataCollector.getTransactions());
5      DataCollector.resetTransactions();
6      Thread.sleep(1000);
7    } catch (Exception e) {
8      e.printStackTrace();
9    }
10 }
```

Es sei hier angemerkt, dass dies ein, zwecks Lesbarkeit, geringfügig geänderter Ausschnitt des Originals ist. Zusätzlich zu dieser while-Schleife, werden je nach Testcase spezifische Werte übergeben. Das Prinzip kann jedoch anhand dieses Beispiels erläutert werden. Es handelt sich hierbei um einen

Thread der mittels *resumeIt()* aktiviert wird, um schliesslich, die zuvor vorgestellte Methode, *addValue(double x, double y)* aufzurufen, um danach diese Daten auch in die Statistik Datenbank einzufügen. Der Sekundenintervall wird durch den *sleep(1000)* Aufruf gewährleistet.

4.2.4 Verwendung

Folgendes Unterkapitel dient als Anleitung zur Verwendung des Programms. Es soll anhand eines Beispieldurchlaufs, mittels Screenshots und eingehenden Erläuterungen der Use Case, genannt Tsearch Testcase beschrieben werden. Dabei wird die Option gewählt einen Vergleich zwischen einer gewerteten und nicht gewerteten Tsearch Abfrage durchzuführen. Beginnend mit den Einstellungen für die Property Dateien, wird jeder Schritt bis hin zum Endergebnis, einem Graph mit einer zugehörigen Log-Datei, dokumentiert.

Property Dateien

Bevor das Programm gestartet werden kann muss sichergestellt werden, dass alle Werte in den Property Dateien *"connection.properties"* und *"simple.properties"* richtig gesetzt sind. Beide Dateien bieten die Möglichkeit Einstellungen für den Verbindungsaufbau und die Testcaseausführung zu setzen. Der Inhalt ist in den Tabellen 4.1 und 4.2 mit den jeweiligen Erklärungen aufgelistet.

Wichtig für das Ausführen eines Tsearch Testcases ist es, die richtigen Einstellungen anzusprechen. Diese sind bezogen auf die *"connection.properties"* Datei immer von derselben Struktur. Man benötigt eine funktionierende Serveradresse, einen Datenbanknamen, den Benutzernamen sowie das Passwort zum erfolgreichen Verbindungsaufbau. Charakteristisch für die in Tabelle 4.1 gezeigten Variablen ist, dass fünf verschiedene Datenbanknamen vorhanden sind, die je nach Szenario angesprochen werden. Dabei stellt die Statistikdatenbank jedoch lediglich einen Zwischenspeicher dar und wird

für jeden Testcase mitverwendet um, wie im Falle eines Tsearch Testcases, Suchbegriffe und Laufzeiten mitzuloggen. Wichtig ist auch hier anzumerken, dass die VIEW Datenbank von der Struktur her identisch zur generischen Datenbank aufgebaut ist, mit dem Unterschied zusätzlich gesetzter Views.

Tabelle 4.1: connection.properties

Variable	Beispielwert	Beschreibung
dbserver	localhost	Datenbank Server
dbname	itp	Name der generischen Datenbank
statdbname	itpstat	Name der Statistikdatenbank
joindbname	joindb	Name der externen Datenbank
tsearchdbname	institutdb	Name der Tsearch Datenbank
viewdbname	itpview	Name der View Datenbank
dbuser	postgres	PostgreSQL Datenbankbenutzername
dbpass	passwort	Passwort für den zugehörigen Benutzer

Die *"simple.properties"* Datei wie sie in Tabelle 4.2 dargestellt ist, beinhaltet um einiges mehr an Variablen, die gesetzt werden können. Einige davon sprechen für sich selbst und bedürfen keiner weiteren Erklärung. Dennoch soll zu zwei spezifischen Werten eine erweiterte Erläuterung zur Verfügung gestellt werden. So definiert die Variable *inserts* mit Tupel pro Tabelle, die Anzahl der Tupel für jede generische Tabelle bevor der Testcase gestartet wird. Mittels dieser Variable kann daher im Zusammenspiel mit dem *tables* Wert die initiale Größe der Testdatenbank gesteuert werden. Weiters gibt es noch die, von der Namensgebung her, irreführenden Variablen genannt *insert*, *select* und *update*. Wie auch schon in der Tabelle beschrieben, definieren diese die Iterationen für die jeweiligen Testcases. Mit Iterationen sind hierbei die gesamten Iterationen pro Testcase gemeint und nicht die Iterationen pro Verbindung. Falls also eine bestimmte Iteration pro Verbindung erreicht werden soll, muss dies hochgerechnet werden. Dies würde beispielsweise bedeuten, dass bei einer gewünschten Anzahl von 1000 Iterationen pro Verbindung und einer vorgegebenen Anzahl von zehn simulierten

Verbindungen, die Iterationsvariable auf 10000 gesetzt werden muss. Insbesonders wichtig für den hier angeführten Beispieldurchgang sind in diesem Fall die Variablen *tsearchtable* und *tsearchiterations*, welche mit den Werten *mv_tmp* und *drei* gesetzt sind.

Tabelle 4.2: simple.properties

Variable	Beispielwert	Beschreibung
tables	10	Tabellenanzahl in generischer Datenbank
connections	1	Definiert Mehrbenutzerumgebung
autocommit	true	Setzt die autocommit Variable
prefix	simple	Prefix für generische Tabellen
inserts	100	Tupel/Tabelle in generischer Datenbank
stringlengh	50	Anzahl der Zeichen für den Zufallsstring
numlength	30	Anzahl der Zeichen für die Zufallszahl
insert	1000000	Iterationen für alle INSERT Testcases
select	100000	Iterationen für alle SELECT Testcases
update	200000	Iterationen für alle UPDATE Testcases
workingdir	/home/user/	Arbeitsverzeichnis
tsearchtable	mv_tmp	Name der Tsearch Tabelle
tsearchiterations	3	Iterationen für den Tsearch Testcase
viewtablename	viewtable	Name der Tabelle auf der ein View erzeugt wird
viewrows	100000	Tupel/Tabelle in der View Datenbank
viewiterations	10000	Iterationen für den VIEW Testcase
viewtable	1	Anzahl der Tabellen mit einem View
externaliterations	1	Iterationen für einen manuellen Testcase

Testcase Vorbereitung

Um einen Testcase erfolgreich durchführen zu können, müssen im Programm bestimmte Aktionen gesetzt werden, um dies zu gewährleisten. Grundsätzlich sollte nach jedem Start des Programms alles zurückgestellt werden, um falsche Analysedaten zu vermeiden. Dies bedeutet, dass temporäre Dateien gelöscht werden und ein Reset der Datenbank durchgeführt werden muss. Ausgangspunkt in einer solchen Situation ist die leere Startoberfläche, wie sie in Abbildung 4.7 zu sehen ist, die den Benutzer darauf hinweist, dass kein Testcase ausgewählt wurde.

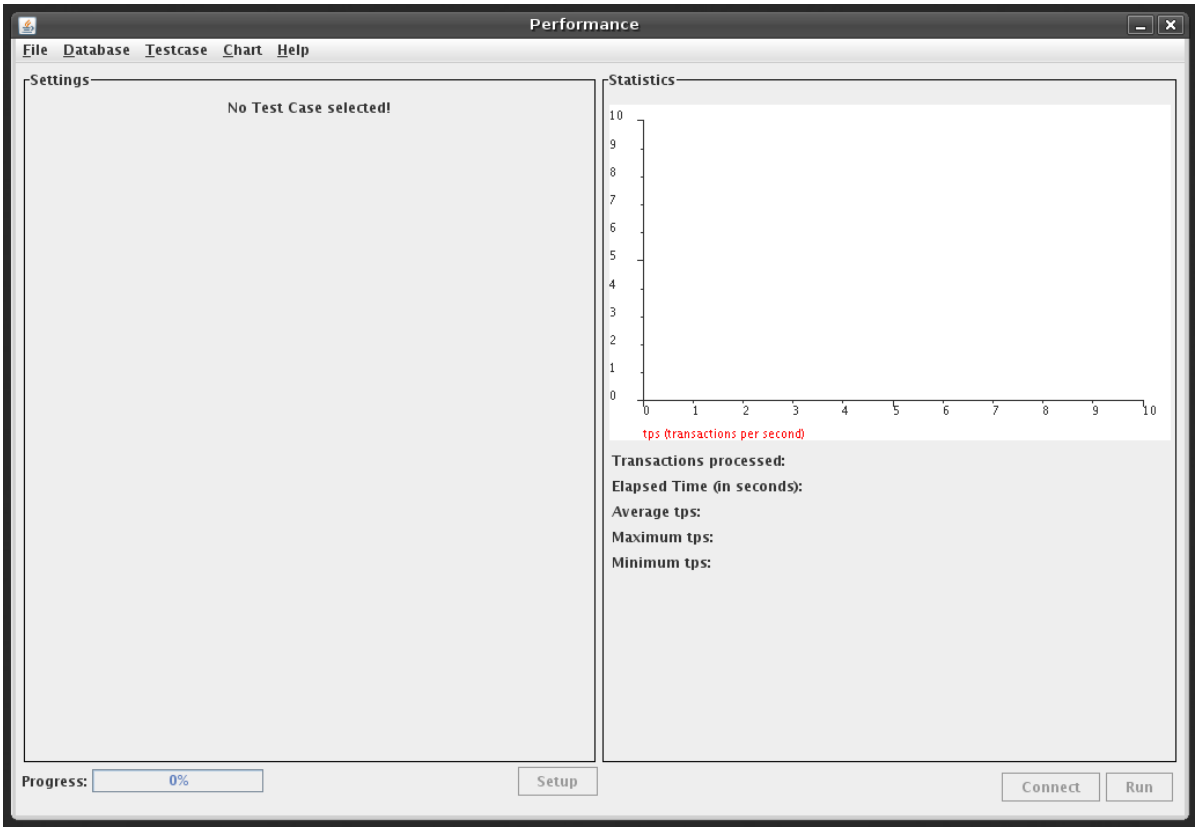

Abbildung 4.7: Startoberfläche

Unter dem Menüpunkt *File* und *Database* können die zuvor genannten Aktionen, genannt *Delete temporary files* und *Reset DB*, getätigt werden. Nachdem nun die Umgebung eingestellt ist, kann über das Menü *Testcase* ein

Testcase ausgewählt werden. Dabei fällt die Kategorisierung so aus, wie sie auch im Use Case Diagramm (4.1) strukturiert ist. Um den vergleichenden Tsearch Testcase auszuwählen, muss der Untermenüpunkt mit dem Namen *Comparison* selektiert werden. Es öffnet sich daraufhin eine Ansicht die dem Endbenutzer einen generellen Überblick bietet, zu sehen in Abbildung 4.8.

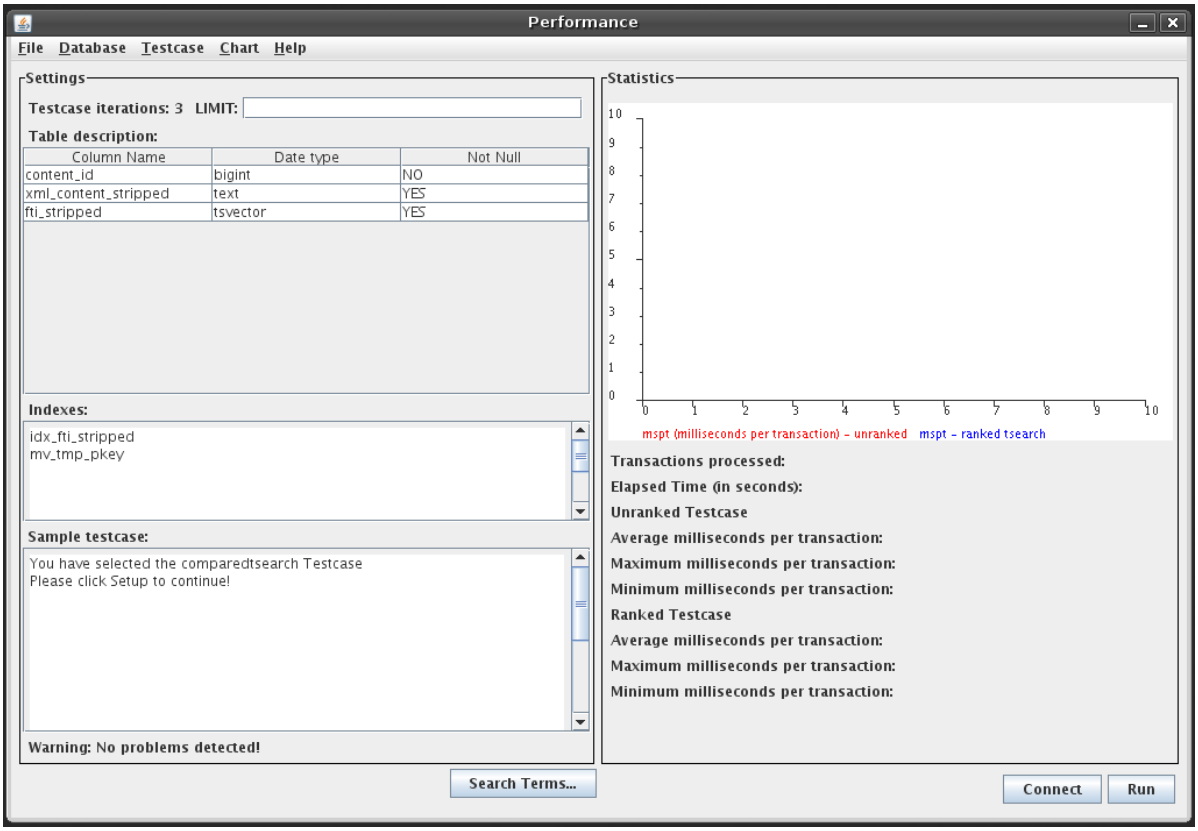

Abbildung 4.8: Testcaseoberfläche für den Tsearch Vergleich

Auf der linken Seite werden alle Einstellungen dargestellt. Man sieht ganz oben die in der *simple.properties* Datei gesetzten Iterationen für das Szenario sowie ein Textfeld mittels dem ein Limit der zurückzulieferenden Werte pro Transaktionen eingegeben werden kann. Darunter sind Eigenschaften der Tabelle, wie sie in der Property Datei definiert wurden, und die gesetzten Indizes abgebildet. In der unteren Hälfte befindet sich ein Textfeld, welches dazu dient eine Beispieltransaktion, zwecks Überprüfung der vorhandenen Daten, darzustellen.

Als nächstes muss der Endbenutzer die Tsearch Wörterliste laden. Dies wird durch den Button *Search Terms...* gehandhabt. Nach erfolgreichem Laden der Datei sollte eine Beispieltransaktion angezeigt werden. Nun kann der Benutzer auf *Connect* und daraufhin *Run* drücken. Man kann jetzt den Verlauf der Testcases anhand des dynamisch mitlaufenden Graphen verfolgen. In diesem Fall werden Millisekunden pro Transaktion gemessen und vergleichend auf dem Graphen dargestellt. Sobald der Testcase zu Ende ist wird eine Zusammenfassung unter dem Graphen abgebildet, wie es in Abbildung 4.9 zu sehen. Diese ist unter anderem auch in der Datei *testcase.log* zu finden.

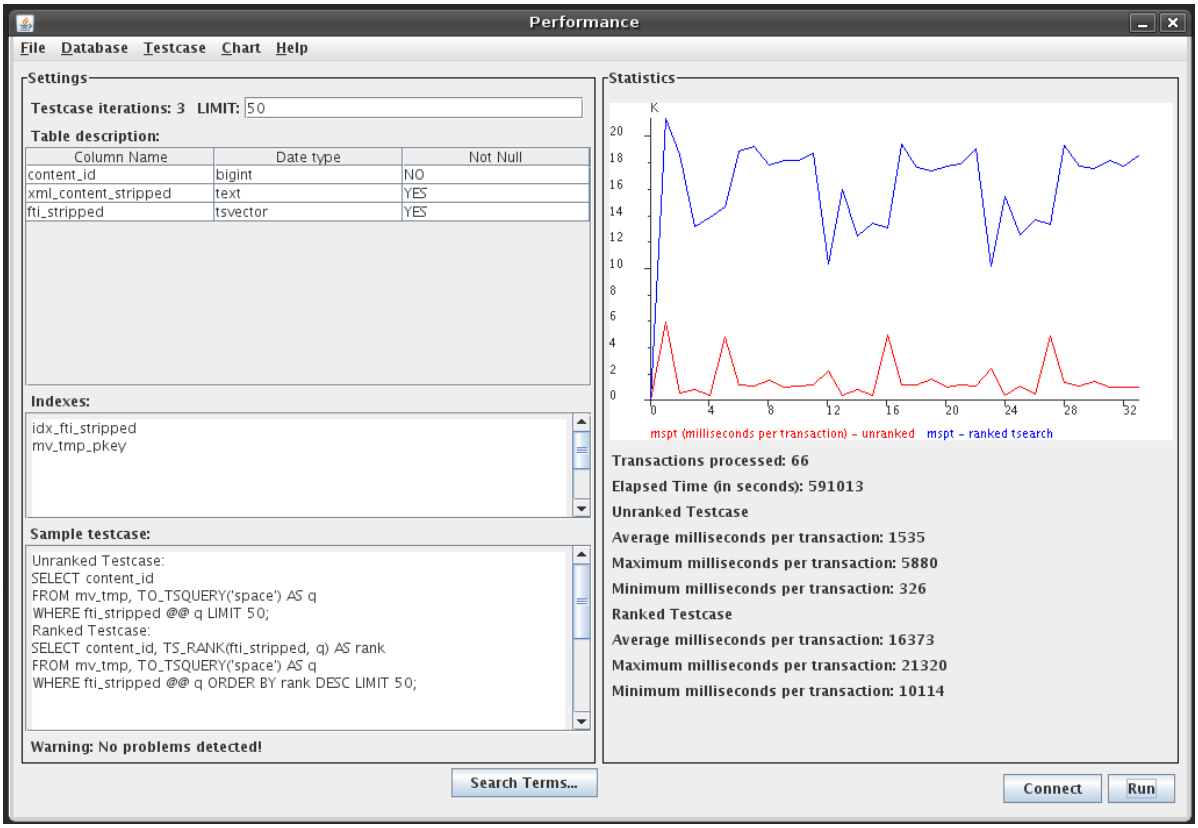

Abbildung 4.9: Endergebnis des Tsearch Testcase

Nun steht es dem Benutzer offen welche Option er wählt, um die graphischen Analysedaten zu exportieren. Es stehen drei Möglichkeiten zur Auswahl:

Das Exportieren als EPS[3]-Datei, als CSV[4]-Datei oder als JPG Bilddatei. Zugriff zu den genannten Exportfunktionen hat der Endbenutzer über den Menüpunkt *Chart*. Es muss lediglich ein Name gewählt werden, um dann im gewünschten Format abzuspeichern. Basierend auf der vorgestellten Implementierung können nun Parameter verstellt und Testcases verglichen werden.

4.2.5 Abfrageoptimierung

Der letzte Teil dieses Abschnitts ist der externen Datenbankumgebung, wie sie in Abbildung 4.5 zu sehen ist, gewidmet. Dabei geht es um einen Optimierungsvorschlag für bestimmte JOIN-Abfragen, welche im Rahmen von laufenden Projekten des Instituts für Informationswirtschaft verwendet werden. Das zugrunde liegende Problem ist die Geschwindigkeit der Abfragen. Ziel ist es somit, die JOIN's soweit zu optimieren, dass eine spürbare Leistungssteigerung gegeben ist. Grundlage dafür stellt der in Punkt 2.2 erwähnte Benchmark, genannt Wisconsin Benchmark dar. Zur Messung der Performance wurde in dem genannten Benchmark die Laufzeit als Variable gewählt. Es soll dies übernommen werden, inklusive der Vorgehensweise der Zeitmessung mittels vorhandener Systemprogramme. Hierbei soll auf die interne Zeitmessung von postgreSQL zurückgegriffen werden. Es folgt nun der dokumentierte Prozess der Optimierung ausgehend von den gegebenen JOIN-Abfragen.

Folgende Abfragen sollen optimiert werden:

```
1 SELECT h.h_id, a_id
2 FROM g, h, f, e, d, c, b
3 WHERE g.g_id = h.g_id
4 AND f.g_id = g.g_id
5 AND e.f_id = f.f_id
6 AND d.d_id = e.d_id
7 AND c.c_id = d.c_id
8 AND b.c_id = c.c_id
```

[3]Encapsulated Postscript
[4]Comma Seperated Values

```
9  AND g.g_id = xxx
10 AND d.d_id = yyy;
11
12 SELECT g_id
13 FROM d JOIN e USING(d_id) JOIN f USING(f_id) JOIN g USING(g_id)
14 WHERE d.d_id = xxx;
15
16 SELECT g_id
17 FROM d JOIN e USING(d_id) JOIN f USING(f_id) JOIN g USING(g_id) JOIN h USING(
       g_id)
18 WHERE d.d_id = xxx and h_id=yyy;
```

Vorweg muss hier angemerkt werden, dass laut offizieller postgreSQL Dokumentation eine Optimierung einer JOIN-Abfrage, wie sie im Folgenden vorgenommen wird, nur im Bezug auf die erste JOIN-Abfrage über sieben Tabellen Sinn macht. Dennoch kann das angewandte Prinzip auch auf den kleineren JOIN's angewandt werden, jedoch wird die Leistungssteigerung dabei nicht in dem Ausmaße zu spüren sein, wie im Vergleich zur ersten Abfrage.

Grundlegendes Problem dieser Abfragen ist, dass dem internen *Query Planner* von postgreSQL viel Spielraum gegeben wird. Somit entsteht ein gewisser Overhead durch die Planung der Abfragen. Kommt es zu einer schlechten Planung, so wird die Abfrage auch dementsprechend langsam ausgeführt. Ursachen dafür liegen meist darin, dass ein zu komplizierter Weg generiert wird oder falsche Join Mechanismen gewählt werden.

Eine Methode die, wie es in Abbildung 4.10 zu sehen ist, eine eindeutig spürbare Leistungssteigerung erbracht hat, war das gezielte Zusammenführen von Tabellen. Es wurde dabei nach dem Prinzip vorgegangen die grössten Tabellen zuerst zu verbinden und darauf folgend die jeweils kleineren. Dies kann mittels expliziter JOIN Klauseln erreicht werden, was bewirkt, dass dem *Query Planner* bestimmte Grenzen gesetzt sind und gleichzeitig Entscheidungen abgenommen werden. Bei einer Abfrage wie sie hier zum Beispiel zu sehen ist, kann der *Query Planner* frei entscheiden welche Tabelle wann mit welcher verbunden wird, was optimierungstechnisch nicht von Vorteil ist [Pos08].

```
1 SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

Es kann daher folgende Optimierung vorgeschlagen werden:

```
1 SELECT h.h_id, a_id
2 FROM (b JOIN (c JOIN (d JOIN (h JOIN (g JOIN (e JOIN f USING(f_id)) USING(g_id)
      ) USING (g_id)) USING(d_id)) USING(c_id)) USING(c_id))
3 WHERE g.g_id = xxx AND d.d_id = yyy;
4
5 SELECT g_id
6 FROM (d JOIN (g JOIN (e JOIN f USING(f_id)) USING(g_id)) USING(d_id))
7 WHERE d.d_id = xxx;
```

Durch die dargestellte Verschachtelung kann bei der ersten JOIN-Abfrage eine Leistungssteigerung von etwa 52 Prozent erreicht werden. Wie schon oben erläutert, kann diese Steigerung bei den restlichen *"kleineren"* JOIN's nicht dokumentiert werden, was schlussfolgernd die offizielle postgreSQL Dokumentation in ihrem Hinweis bestätigt. Bei dem Vorgehen der Messung wurde dafür gesorgt, dass die Datenbank sich in einem aktiven Zustand befand. Ausserdem wurde durch mehrere Durchgänge mit unterschiedlichen Werten ein möglicher Zufall ausgeschlossen.

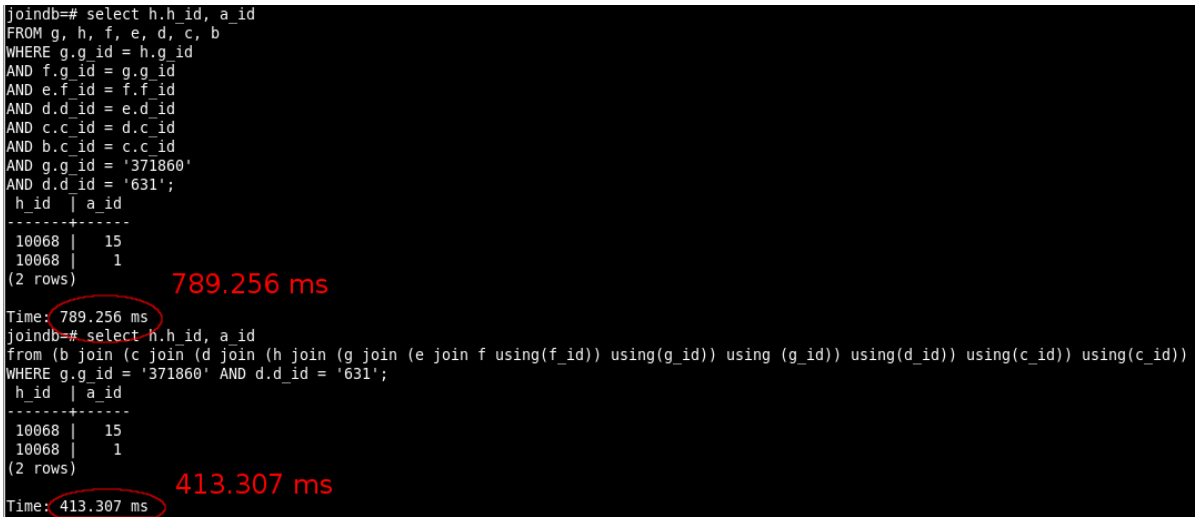

Abbildung 4.10: Optimierung einer JOIN-Abfrage

Wie in Abbildung 4.10 zu sehen ist, kann alleine die Reihenfolge der JOIN Operationen zu einer erheblichen Leistungssteigerung führen. Dieser erste Schritt dient somit als Grundlage zur weiteren Optimierung von Anfragen. Zusammen mit dem entwickelten Tool zur Messung der Performance einer Datenbank sind theoretische und praktische Grundlagen definiert, um vertiefende Schritte zu setzen.

4.3 Entwicklungsperspektiven

Abschliessend stellt sich die Frage welche Weiterentwicklungsmöglichkeiten gegeben sind. Das implementierte Programm ist in einem lauffähigen Status. Dennoch erfordert es ein gewisses Grundwissen, um verwertbare Daten anhand des Tools zu generieren. Ziel dieses Kapitels war es unter anderem dieses Grundwissen zu vermitteln. Durch die Fertigstellung des Tools ergeben sich neue Perspektiven der Weiterentwicklung. Dabei können insbesondere auf der Usability Ebene zwei Richtungen eingeschlagen werden. Einerseits die Entwicklung hin zu einer *Rich Client Platform*[5] unter Eclipse, andererseits die zusätzliche Unterstützung für die Kommandozeile, was die Möglichkeit bietet bestimmte Szenarien durch einfache Aufrufe zu starten. Dies hätte unter anderem den Vorteil, dass zum Beispiel in einer vorhandenen Unixumgebung, Szenarien automatisiert über einen längeren Zeitraum mittels Kommandoaufruf gestartet werden können.

Weiters können Entwicklungen im funktionalen Bereich zu einer Vertiefung der vorhandenen Grundlagen führen. Die Verfeinerung und Einbettung der externen Datenbankstruktur wäre ein Schritt, um den Funktionsumfang zu bereichern. Darüber hinaus wäre zu bedenken inwieweit es Sinn machen würde, entweder die vorhandenen generischen Benchmarks zu vertiefen, oder sich gezielt auf die Weiterentwicklung von spezifischen Testcases zu konzentrieren. Nachdem postgreSQL das Modul Tsearch in die aktuellste Version

[5] RCP - Ein Framework von Eclipse zur Wiederverwendung von Komponenten für die Erstellung eigener Software. [VBS08]

als Standard mitaufgenommmen hat, kann davon ausgegangen werden, dass diese Funktionalität auch in Zukunft von Bedeutung sein wird. Dadurch dass es im spezifischen zu Tsearch, nach aktuellem Stand, wenig detaillierte Untersuchungen gibt, könnte man hier als Vorreiter insbesondere im Bereich der Tsearch-Optimierung Pionierarbeit leisten.

Sicher ist, dass zahlreiche Optionen der Weiterentwicklung bestehen. Es gilt daher diese untereinander abzuwägen, um eine effektive mögliche Fortführung des Projekts zu gewährleisten.

5 Zusammenfassung und Ausblick

Motivation dieser Arbeit war (I) der Bedarf an einem fundierten Überblick über die Grundlagen der Performance Messung und Datenbankoptimierung im Bereich von umfangreichen Datenbankapplikationen, sowie (II) die Entwicklung eines Tools zur Messung der Leistung von Datenbankabfragen. Trotz der teilweise relativ früh datierten Grundlagen aus Kapitel 2, haben sich diese bis heute als wertvoll erwiesen und bieten somit eine gute Referenz im Bezug auf Vorgehensweisen der Performance Messung. In diesem Zusammenhang stellt das *"Transaction Processing Performance Council"* einen aktuellen Referenzpunkt dar im Bezug auf Standardisierung von Benchmarks.

Durch den Fokus auf eine postgreSQL Datenbank konnten spezifische und allgemeine Aussagen bezüglich einer erfolgreichen Optimierung getätigt werden. Von enormer Wichtigkeit sind hierbei die fünf Grundprinzipien des Tunings aus Kapitel 3, welche als Grundlage für jegliche Herangehensweisen einer Datenbankoptimierung dienen sollen. Im Spezifischen kann durch die Verwendung von postgreSQL auf ein umfangreiches Angebot an communitybasierten Empfehlungen für die Optimierung von Parametern zurückgegriffen werden. Dennoch fehlt es hier an postgreSQL orientierter Literatur, wie es zum Beispiel bei Oracle der Fall ist, um gezielte wiederverwendbare Lösungsvorschläge zu präsentieren. Somit stellt diese Arbeit auch einen ersten Versuch dar, vorhandene Empfehlungen zu bündeln.

Das implementierte Tool zur Messung der Leistung von Datenbankabfragen stellt ein Framework dar, um spezifisch für postgreSQL designierte Testcases durchzuführen. Obwohl generische Benchmarks implementiert wurden

spricht das Programm eher spezialisierte Endbenutzer im Bezug auf Tsearch Abfragen und *custom Benchmarks* an. In diesen Bereichen ist auch das größte Potential gegeben, da im Grunde generische Benchmarks schon in standardisiertem Ausmaß vom *"Transaction Processing Performance Council"* behandelt werden.

Es lässt sich aus den oben zusammengefassten Punkten eine grundlegende Tendenz hin zur Spezialisierung von spezifischen Bereichen in Datenbankumgebungen und ihren Funktionalitäten ableiten. Zusammenfassend bietet diese Arbeit mit dem implementierten Tool eine fundierte Grundlage zur möglichen Weiterentwicklung und Spezialisierung von Testcases im Bereich von groß angelegten postgreSQL Datenbanken und ihren spezifischen Funktionalitäten.

Literaturverzeichnis

[ABF+06] Mike Ault, Donald K. Burleson, Claudia Fernandez, Kevin Klein, and Dr Bert Scalzo. *Database Benchmarking, Practical Methods for Oracle and SQL Server*. Buch 3. Rampant TechPress, 2006.

[Adv08] Advanced Computer & Network Corporation. *Get to know RAID*, 2008. http://www.acnc.com/, zuletzt besucht am 18.07.2008.

[bar08] Java threads, 2008. http://www.cs.usfca.edu/ parrt/course/601/lectures/threads.html, zuletzt besucht am 14.07.2008.

[BDT83] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Florence*, 1983.

[ber05] Postgresql 8.0 performance checklist, Januar 2005. http://www.powerpostgresql.com/PerfList, zuletzt besucht am 14.06.2008.

[ber08] Annotated gucs version 8.3, Mai 2008. http://www.pgcon.org/2008/schedule/events/104.en.html, zuletzt besucht am 06.07.2008.

[DB03] Shridhar Daithankar and Josh Berkus. Tuning postgresql for performance, Juli 2003. http://www.varlena.com/varlena/GeneralBits/Tidbits/perf.html, zuletzt besucht am 07.07.2008.

[ea85] Anon. et al. A measure of transaction processing power. *Datamation*, page 112, 1 1985.

[EN05] Ramez Elmasri and Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. 3. Auflage. Pearson Studium, 2005.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.

[GP87] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *SIGMOD Rec.*, 16(3):395–398, 1987.

[Gra87] Jim Gray. A view of database system performance measures. In *SIGMETRICS '87: Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 3–4, New York, NY, USA, 1987. ACM.

[HN05] H.R. Hansen and G. Neumann. *Wirtschaftsinformatik 2, Informationstechnik*. Lucius & Lucius, UTB 2670, 9. Auflage, April 2005.

[O'N97] Patrick E. O'Neil. Database performance measurement. In Tucker [Tuc97], pages 1078–1092.

[Pos08] PostgreSQL Global Development Group. *PostgreSQL 8.3.3 Documentation*, 2008. http://www.postgresql.org/docs/8.3/static/, zuletzt besucht am 09.07.2008.

[SB03] Dennis Shasha and Philippe Bonnet. *Database tuning principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers, 2003.

[Sha98] Kim Shanley. History and overview of the tpc, 1998. http://tpc.org/information/about/history.asp, zuletzt besucht am 07.06.2008.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, international edition, Juli 2001.

[Smi07] Gregory Smith. 5-minute introduction to postgresql performance, April 2007.

http://www.westnet.com/ gsmith/content/postgresql/pg-5minute.htm, zuletzt besucht am 07.07.2008.

[too08] Josh berkus, 2008. http://www.ittoolbox.com/profiles/josh_berkus, zuletzt besucht am 14.06.2008.

[Tra07] Transaction Processing Performance Council. *Detailed TPC-E description*, 2007. http://www.tpc.org/tpce/spec/TPCEDetailed.doc, zuletzt besucht am 07.06.2008.

[Tuc97] Allen B. Tucker, editor. *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[VBS08] Lars Vogel, Gowthaman Basuvaraj, and Remy Chi Jian Suen. Rich client platform, Juni 2008. http://wiki.eclipse.org/index.php/Rich_Client_Platform, zuletzt besucht am 18.07.2008.

[vms06] The openvms frequently asked questions, 2006. http://eisner.encompasserve.org/vms/vmsfaq.html, zuletzt besucht am 07.06.2008.

[Wes08] Achim Westermann. Jchart2d, precise visualization of data, Juni 2008. http://jchart2d.sourceforge.net/, zuletzt besucht am 16.07.2008.

[wik08a] B-baum, Juni 2008. http://de.wikipedia.org/wiki/B-Baum, zuletzt besucht am 10.07.2008.

[wik08b] Nested raid levels, Juni 2008. http://en.wikipedia.org/wiki/Nested_RAID_levels, zuletzt besucht am 18.07.2008.

[wik08c] Performance optimization - postgresql wiki, Juli 2008. http://wiki.postgresql.org/wiki/Performance_Optimization, zuletzt besucht am 06.07.2008.

[wik08d] Tuning your postgresql server, Juli 2008. http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server, zuletzt besucht am 07.07.2008.

Abbildungsverzeichnis

Tabellenverzeichnis