



FernUniversität in Hagen

– Fakultät für Mathematik und Informatik –
Lehrgebiet Datenbanken und Informationssysteme

Analyse und Optimierung der Webseite des Wedekind Projektes

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt von

Marco Galster

Matrikelnummer: 8335710

Referentin : Prof. Dr. Uta Störl

Betreuer : Sebastian Bruchhaus, Tobias Holstein

ERKLÄRUNG

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe.

Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht.

Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Höchstadt, 01. Januar 2024

Marco Galster

ABSTRACT

A short summary of the contents in English of about one page. The following points should be addressed in particular:

- **Motivation:** Why did this work come about? Why is the topic of the work interesting (for the general public)? The motivation should be abstracted as far as possible from the specific tasks that may be given by a company.
- **Content:** What is the content of this thesis? What exactly is covered in the thesis? The methodology and working method should be briefly discussed here.
- **Results:** What are the results of this work? A brief overview of the most important results as a teaser to read the complete thesis.

BTW: A great guide by Kent Beck how to write good abstracts can be found here:

<https://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache von ca. einer Seite Länge. Dabei sollte vor allem auf die folgenden Punkte eingegangen werden:

- Motivation: Wieso ist diese Arbeit entstanden? Warum ist das Thema der Arbeit (für die Allgemeinheit) interessant? Dabei sollte die Motivation von der konkreten Aufgabenstellung, z.B. durch eine Firma, weitestgehend abstrahiert werden.
- Inhalt: Was ist Inhalt der Arbeit? Was genau wird in der Arbeit behandelt? Hier sollte kurz auf Methodik und Arbeitsweise eingegangen werden.
- Ergebnisse: Was sind die Ergebnisse der Arbeit? Ein kurzer Überblick über die wichtigsten Ergebnisse als Teaser, um die Arbeit vollständig zu lesen.

Eine großartige Anleitung von Kent Beck, wie man gute Abstracts schreibt, finden Sie hier:

<https://plg.uwaterloo.ca/~migod/research/beck00PSLA.html>

INHALTSVERZEICHNIS

1	Einleitung	1
1.1	Motivation	1
1.2	Ausgangslage	1
1.3	Ziel der Arbeit	1
1.4	Gliederung	2
2	Grundlagen	3
2.1	Glassfish - Enterprise Java Beans	3
2.2	Glassfish - Java Persistence API	4
2.3	Glassfish - OpenJPA Cache	5
2.4	PostgreSQL - Memory Buffers	5
2.5	PostgreSQL - Services	6
2.6	PostgreSQL - Abfragen	6
3	Konzept	8
3.1	Aufbau der Umfrage	8
3.2	Allgemeine Betrachtung des Systems	8
3.3	Das Vorgehen der Optimierung	8
3.4	Aktueller Aufbau der Software	8
3.5	Vergleich mit anderen Technologien	8
4	Performance-Untersuchung	9
4.1	Auswertung der Umfrage	9
4.2	Einbau und Aktivieren von Performance-Messung	9
4.3	Statistiken im PostgreSQL auswerten	9
4.4	Überprüfung des PostgreSQL und Servers	9
5	Optimierung	10
5.1	Ermittlung der Performance-Probleme	10
5.2	Analyse der Abfrage	10
5.3	Optimierungen der Abfragen	10
5.4	Anpassung der Konfiguration	10
6	Evaluierung	11
6.1	Befragung der Benutzer und Entwickler	11
6.2	Erneute Laufzeitanalyse starten	11
6.3	Statistiken im PostgreSQL auswerten	11
6.4	Vergleich der Ergebnisse vor und nach der Optimierung	11
7	Zusammenfassung und Ausblick	12
	Literatur	13

ABBILDUNGSVERZEICHNIS

Abbildung 2.1	Ablauf einer Web-Anfrage	4
---------------	------------------------------------	---

LISTINGS

Listing 5.1	ein sql beispiel	10
-------------	----------------------------	----

ABKÜRZUNGSVERZEICHNIS

EINLEITUNG

Die Akzeptanz und damit die Verwendung einer Software hängt von verschiedenen Kriterien ab. Hierbei ist neben der Stabilität und der Fehlerfreiheit die Performance beziehungsweise die Reaktionszeit der Software ein sehr wichtiges Kriterium. Hierfür muss sichergestellt werden, dass die Anwendung immer in kurzer Zeit reagiert oder entsprechende Anzeigen dargestellt um eine längere Bearbeitung anzuzeigen.

1.1 MOTIVATION

Die Frank-Wedekind-Bibliothek soll als Grundlage weiterer Forschungen dienen. Durch die weitere Forschung soll die literarhistorische und kulturgeschichtliche Wissen über die Kultur zwischen 1880 und 1918 erweitern. Um dieses Vorhaben umsetzen zu können muss die Anwendung dem Benutzer gerecht werden, damit diese verwendet wird und somit den Wissenstand entsprechend erweitert werden kann.

Um das Ziel der Akzeptanz zu erreichen und das sich die Bediener rein auf ihre Arbeit konzentrieren können, muss mit der Anwendung flüssig interagiert werden können. Entsprechend müssen die Wartezeiten auf ein minimum reduziert werden. Des Weiteren muss die Stabilität der Anwendung gesteigert werden.

1.2 AUSGANGSLAGE

TODO: hier die Grundlagen aus dem Expose? Braucht man die Motivation dann noch?

1.3 ZIEL DER ARBEIT

Die aktuelle Umsetzung beinhaltet die bisher definierten Anforderungen vollständig, darunter fallen die Recherchemöglichkeiten, sowie auch die Eingabe und die Verarbeitung der Briefe. Ein größeres Problem hierbei ist die Performance der Oberfläche. Auf Grund der langen Abfragedauer des Datenbestandes leidet die Akzeptanz der Anwendung.

Das Ziel der Arbeit ist es, die Abfragedauer zu verringern, wodurch die Performance der Oberfläche signifikant verbessert wird.

Hierbei ist auch ein Vergleich mit anderen Technologien angedacht.

1.4 GLIEDERUNG

Zu Beginn der Arbeit werden im Kapitel 2 die Struktur und der grundsätzliche Aufbau der Anwendung erklärt. Hierbei wird aufgezeigt an welchen Stellen immer wieder zu Unstimmigkeiten kommen kann und wie diese zu überprüfen sind.

Nachfolgend wird im Kapitel 3 die Konzepte vorgestellt mit welchen man die Probleme ermitteln wird. Hierbei ist zusätzlich ein Blick auf andere Frameworks sinnvoll, um dort aus den bekannten Anomalien zu lernen und deren Lösungsansatz zu überprüfen ob diese angewandt werden kann.

Bei den Performance-Untersuchung in Kapitel 4 werden nun die Konzepte angewandt, um die Problemstellen zu identifizieren. Diese werden dann bewertet, unter den Gesichtspunkten ob eine Optimierung an dieser Stelle sinnvoll ist, oder ob der Arbeitsaufwand dafür zu enorm ist.

Nach der Entscheidung der Reihenfolge der zu bearbeitenden Punkte, wird im Kapitel 5 je nach Problemart ein gesondertes Vorgehen der Optimierung durchgeführt, um diese zu beheben oder mindestens in einen akzeptablen Rahmen zu verbessern. Diese Optimierungen werden dann in der Software entsprechend dem Weg angepasst.

Nach der Optimierung kommt nun die Evaluierung im Kapitel 6, um zu überprüfen ob die Anpassungen die gewünschte Verbesserung in der Performance gebracht haben.

Zum Abschluss im Kapitel 7 wird explizit die Anpassungen dargestellt, die zu einer merklichen Verbesserung geführt haben und wie diese entsprechend umgesetzt werden müssen. Zusätzlich wird beschrieben wie ein weiteres Vorgehen durchgeführt werden kann.

Da die Anwendung als Webseite umgesetzt ist, ist der zugehörige Client für den Benutzer ein Webbrowser. Dies bedeutet, dass jeder Wechsel einer Seite oder eine Suchanfrage als Web-Request an den Server geschickt wird. Solch ein Web-Request geht durch mehrere Schichten des Server-System bis die Antwort an den Client zurückgesendet wird, wie in 2.1 dargestellt.

Angefangen bei der Anfrage die über den Webbrowser an den Server gestellt wird und vom *Glassfish*-Server empfangen wird. In diesem wird anhand des definierten Routing entschieden, an welche *Java Server Page* die Anfrage weitergeleitet und verarbeitet wird. In dieser wird die Darstellung der Webseite geladen und die Anfragen für den darzustellenden Datenbestand abgeschickt.

Die Datenanfragen werden über die *Enterprise Java Beans* an die *Java Persistence API* weitergeleitet. Hier wird nun geprüft, ob die Daten aus dem *OpenJPA Cache* direkt ermittelt werden können, oder ob die Abfrage an das unterlagerte Datenbankmanagementsystem *PostgreSQL* weitergeleitet werden muss. Die ermittelten Daten vom DBMS werden bei Bedarf im *OpenJPA Cache* aktualisiert.

Das *PostgreSQL* besteht aus mehreren Teilen die ineinander greifen um die Anfragen zu bearbeiten. Dabei sind die *Memory Buffers* notwendig um den Zugriff auf die Festplatte zu reduzieren, um die Bearbeitungszeit zu verringern. Um Anfragen die den Zugriff auf die Festplatte benötigen effizienter zu gestalten, bereiten die *Services* die Datenstrukturen auf.

2.1 GLASSFISH - ENTERPRISE JAVA BEANS

In den Java-EE-Anwendungen wird der *Persistenzkontext* für die Anfragen vom *Application-Server* bereitgestellt. Hierfür werden *Application-Server* wie *GlassFish* genutzt, um die Verwendung eines Pools von Datenbankverbindungen zu definieren [MW12, S. 68]. Dadurch kann die Anzahl der Verbindung geringer gehalten werden als die Anzahl der Benutzer die an der Anwendung arbeiten. Zusätzlich werden die Transaktionen über *Stateful Session-Bean (SFSB)* gehandhabt, welche automatisch vor dem Aufruf erzeugt und danach wieder gelöscht werden. Dies birgt allerdings den Nachteil, dass der *Persistenzkontext* sehr groß werden kann, wenn viele Entities in den *Persistenzkontext* geladen werden. Da dies häufig zu Speicher- und damit Performanz-Problemen [MW12, S. 79] führen kann, muss hier darauf geachtet werden, nicht mehr benötigte Entities aus dem *Persistenzkontext* zu lösen.

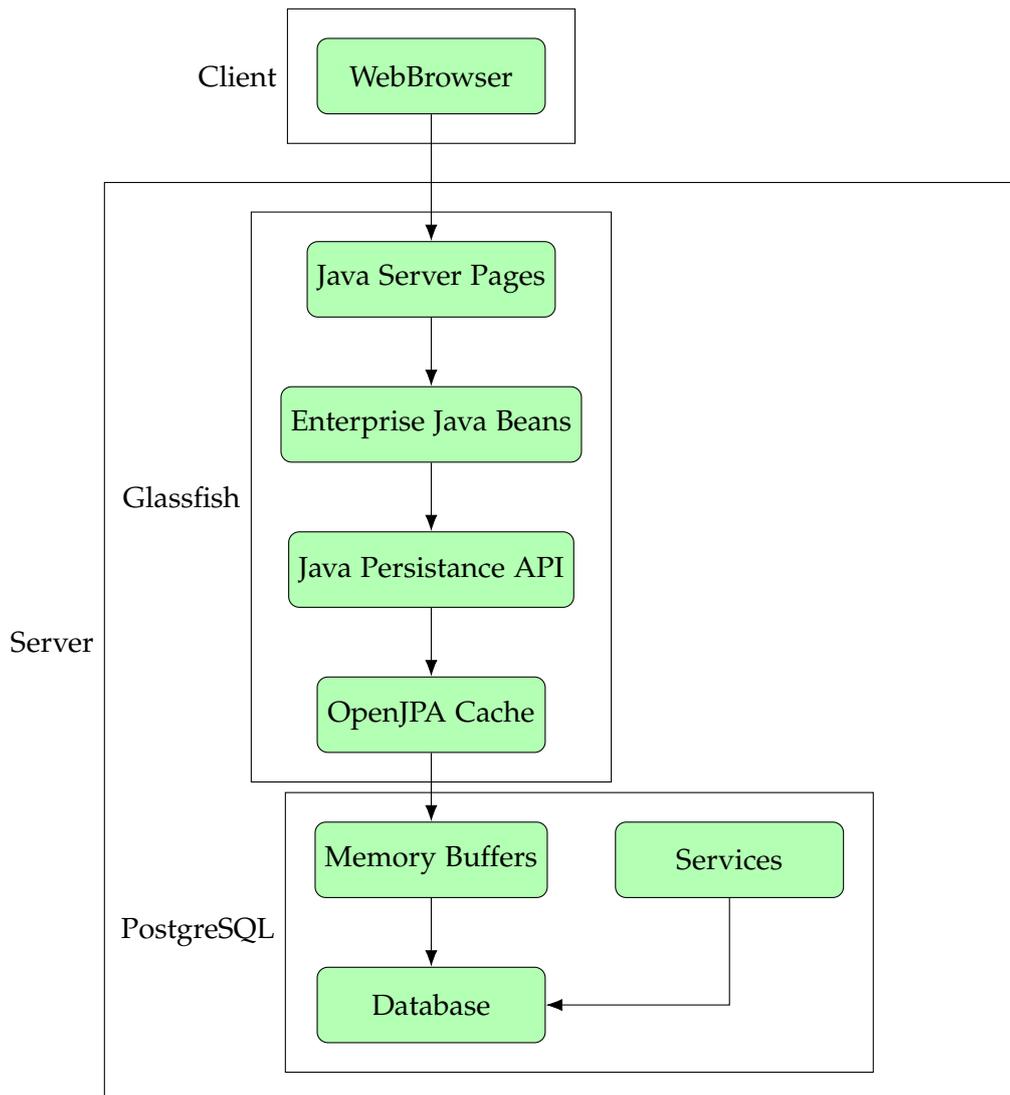


Abbildung 2.1: Ablauf einer Web-Anfrage

2.2 GLASSFISH - JAVA PERSISTENCE API

Die *Java Persistence API (JPA)* wird als First-Level-Cache in Java-EE-Anwendung verwendet, hier nehmen die Objekte einen von vier Zuständen ein [MW12, S. 57]. Im Zustand *Transient* sind die Objekte erzeugt, aber noch nicht in den Cache überführt worden. Wenn diese in den Cache überführt worden sind, nehmen sie den Zustand *Verwaltet* ein. Ist das Objekt aus dem Cache und der Datenbank entfernt worden, nimmt es den Zustand *Gelöscht* an. *Losgelöst* ist der letzte Zustand, bei dem das Objekt aus dem Cache entfernt worden ist, aber nicht aus der Datenbank.

Eine Menge von Objekten wird als *Persistenzkontext* bezeichnet. Solange die Objekte dem *Persistenzkontext* zugeordnet sind, also den Zustand *Verwaltet* besitzen, werden diese auf Änderungen überwacht, um sie am Abschluss mit der Datenbank zu synchronisieren. In der Literatur wird hierzu der Begriff *Automatic Dirty Checking* verwendet [MW12, S. 61].

2.3 GLASSFISH - OPENJPA CACHE

Zusätzlich kann im *JPA* ebenfalls noch der *Second Level Cache* (L2-Cache) aktiviert werden. Dieser steht jedem *Persistenzkontext* zur Verfügung und kann dadurch die Anzahl der Datenbankzugriffe deutlich reduzieren, was bei langsamen Datenbank-Anbindungen zu hohen Performance-Gewinnen führen kann [MW12, S. 171]. Gegen die Verwendung spricht, dass die Daten im *Second Level Cache* explizit über Änderungen informiert werden müssen, welche sonst beim nächsten Aufruf veraltete Werte liefern. Ebenfalls benötigt so ein Cache einen höheren Bedarf an Arbeitsspeicher, in dem die Daten parallel zur Datenbank bereitgestellt werden, daher ist die Benutzung nur problemlos bei Entities möglich, auf die meist lesend zugegriffen wird.

In der OpenJPA-Erweiterung für den L2-Cache, wird in *Objekt-Cache* (in OpenJPA als *DataCache* bezeichnet) und *Query-Cache* unterschieden. Über die Funktionen `find()` und `refresh()` oder einer Query werden die geladenen Entities in den Cache gebracht. Davon ausgenommen sind *Large Result Sets* (Abfragen die nicht alle Daten auf einmal laden), *Extent-Technologien* und *Queries*, die einzelne Attribute von Entities zurückliefern, aber nicht das Entity selbst. Hierbei kann genau gesteuert werden, welche Entity in den Cache abgelegt wird und welche nicht. Ebenfalls kann auf Klassenbasis der zugehörige Cache definiert werden, um eine bessere Last-Verteilung beim Zugriff zu ermöglichen [MW12, S. 314].

Im *Query-Cache* werden die Abfragen bzw. die Eigenschaften einer Abfrage und die zurückgelieferten Ids der Entities gespeichert. Bei einem erneuten Aufruf dieser Abfrage werden die referenzierten Objekte aus dem *Objekt-Cache* zurückgegeben. Bei veränderten referenzierten Entities wird der *Query-Cache* nicht genutzt und die betroffenen Abfragen werden unverzüglich aus dem *Query-Cache* entfernt [MW12, S. 316].

Um zu prüfen, ob die Einstellungen sinnvoll gesetzt sind, kann in OpenJPA eine Cache-Statistik abgefragt werden. Mit dieser kann die Anzahl der Lese- und Schreibzugriffe im Cache überprüft werden, entsprechend dieser Auswertung sollten die Einstellungen an den Entities angepasst werden [Ibm].

2.4 POSTGRESQL - MEMORY BUFFERS

Die Speicherverwaltung des PostgreSQL-Servers muss für Produktivsysteme angepasst werden [EH13, S. 34–38]. Hierunter fallen die *shared_buffers* die bei ca. 10 bis 25 Prozent des verfügbaren Arbeitsspeichers liegen sollten. Mit dieser Einstellung wird das häufige Schreiben des Buffers durch Änderungen von Daten und Indexen auf die Festplatte reduziert.

Die Einstellung *temp_buffers* definiert wie groß der Speicher für temporäre Tabellen pro Verbindung maximal werden darf und sollte ebenfalls überprüft werden. Ein zu kleiner Wert bei großen temporären Tabellen führt zu einem signifikanten Leistungseinbruch, wenn die Tabellen nicht im Hauptspeicher, sondern in einer Datei ausgelagert werden.

Der *work_mem* definiert die Obergrenze des zur Verfügung gestellt Hauptspeichers pro Datenbankoperation wie effizientes Sortieren, Verknüpfen oder Filtern. Ebenso wird im Falle eines zu klein gewählten Speichers auf temporäre Dateien auf der Festplatte ausgewichen, was signifikanten Leistungseinbrüchen zur Folge haben kann.

Die *maintenance_work_mem* wird bei Verwaltungsoperationen wie Änderungen und Erzeugungen von Datenbankobjekten als Obergrenze definiert. Die Wartungsaufgabe *VACUUM*, welche die fragmentierten Tabellen aufräumt und somit die Performance hebt, beachtet die Obergrenze ebenfalls.

2.5 POSTGRESQL - SERVICES

Die Wartung des Datenbanksystems ist eine der wichtigsten Aufgaben und sollte regelmäßig durchgeführt werden, damit die Performance des Systems durch die Änderungen des Datenbestands nicht einbricht [EH13, S. 75]. Hierfür gibt es den *VACUUM*-Befehl, welcher entweder per Hand oder automatisch durch das Datenbanksystem ausgeführt werden soll. Für die automatische Ausführung kann der maximal verwendete Speicher über die Einstellung *autovacuum_work_mem* gesondert definiert werden [Posa]. Neben dem Aufräumen durch *VACUUM*, sollten auch die Planerstatistiken mit *ANALYZE* [EH13, S. 83] aktuell gehalten werden, damit die Anfragen durch den Planer richtig optimiert werden können. Für beide Wartungsaufgaben gibt es den Autovacuum-Dienst, dieser sollte aktiv und richtig konfiguriert sein.

Mit dem Tool *pgFouine* [EH13, S. 155] können die Logs des PostgreSQL Server analysiert und auf Probleme hin untersucht werden. Hiermit können sehr einfach die häufigsten bzw. langsamsten Anfragen ermittelt werden.

2.6 POSTGRESQL - ABFRAGEN

Für weitere Optimierungen werden anschließend die Anfragen einzeln überprüft. Hierfür ist es sinnvoll die Ausführungspläne der Abfrage zu analysieren [EH13, S. 252], die verschiedenen Plantypen und ihre Kosten zu kennen, sowie die angegebenen Werte für die Plankosten zu verstehen [DNB21, S. 24–30]. Besonderes Augenmerk gilt dem Vergleichen des tatsächlich ausgeführten mit dem ursprünglichen Plan [EH13, S. 254]. Eine der wichtigsten Kennzeichen hierbei ist, ob die Zeilenschätzung akkurat war, größere Abweichungen weisen häufig auf veraltete Statistiken hin.

Um die Abfragen selbst zu optimieren, gibt es ein Vorgehen über mehrere Schritte [DNB21, S. 304–308]. Zuerst wird Unterschieden, ob es sich um eine *Kurze* oder eine *Lange* Abfrage handelt. Im Falle einer *Kurzen* Abfrage, werden zuerst die Abfragekriterien überprüft. Sollte dies zu keiner Verbesserung führen, werden die Indexe geprüft. Ist dies ebenso erfolglos, wird die Abfrage nochmals genauer analysiert und so umgestellt, dass die restriktivste Einschränkung zuerst zutrifft. Bei einer *Langen* Abfrage soll überprüft werden, ob es sinnvoll ist, das Ergebnis in einer Tabelle zu speichern und bei Änderungen zu aktualisieren. Wenn dies nicht möglich ist, sollten die

folgenden Schritte durchgeführt werden. Zuerst wird der restriktivste Join gesucht und überprüft, ob dieser als Erstes ausgeführt wird. Anschließend fügt man weitere Joins hinzu und prüft die Ausführungszeit und die Abfragepläne. Als Nächstes wird sich vergewissert, ob große Tabellen nicht mehrfach durchsucht worden sind. Bei Gruppierungen ist noch zu prüfen, ob diese früher durchgeführt werden können, um die Abfragemenge zu verringern.

Bei *Langen* Abfragen ist die Abhandlung »Optimizing Iceberg Queries with Complex Joins« [WRY17] ein zusätzlicher Ratgeber, um die Performance zu steigern.

Des Weiteren können über das Modul `pg_stat_statements` Statistiken der Aufrufe die an den Server gestellt wurden, ermittelt werden [Posb]. Hierbei können die am häufigsten Aufgerufenen und die Anfragen mit der längsten Ausführungszeit ermittelt werden.

KONZEPT

3.1 AUFBAU DER UMFRAGE

3.2 ALLGEMEINE BETRACHTUNG DES SYSTEMS

3.3 DAS VORGEHEN DER OPTIMIERUNG

3.4 AKTUELLER AUFBAU DER SOFTWARE

3.5 VERGLEICH MIT ANDEREN TECHNOLOGIEN

PERFORMANCE-UNTERSUCHUNG

- 4.1 AUSWERTUNG DER UMFRAGE
- 4.2 EINBAU UND AKTIVIEREN VON PERFORMANCE-MESSUNG
- 4.3 STATISTIKEN IM POSTRGRESQL AUSWERTEN
- 4.4 ÜBERPRÜFUNG DES POSTGRESQL UND SERVERS

OPTIMIERUNG

5.1 ERMITTLUNG DER PERFORMANCE-PROBLEME

5.2 ANALYSE DER ABFRAGE

5.3 OPTIMIERUNGEN DER ABFRAGEN

5.4 ANPASSUNG DER KONFIGURATION

und hier ein sql-beispiel Listing 5.1

Listing 5.1: ein sql beispiel

```
select *  
from tblCPDataX  
where szName = N'EDA01'
```

EVALUIERUNG

- 6.1 BEFRAGUNG DER BENUTZER UND ENTWICKLER
- 6.2 ERNEUTE LAUFZEITANALYSE STARTEN
- 6.3 STATISTIKEN IM POSTGRESQL AUSWERTEN
- 6.4 VERGLEICH DER ERGEBNISSE VOR UND NACH DER OPTIMIERUNG

ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

- [Ibm] 2023. URL: <https://www.ibm.com/docs/de/was/8.5.5?topic=applications-configuring-openjpa-caching-improve-performance> (besucht am 24.09.2023).
- [Posa] 2023. URL: <https://postgrespro.com/docs/postgresql/14/runtime-config-resource> (besucht am 27.12.2023).
- [Posb] 2023. URL: <https://www.postgresql.org/docs/8.4/pgstatstatements.html> (besucht am 27.12.2023).
- [DNB21] Henrietta Dombrovskaya, Boris Novikov und Anna Bailliekova. *PostgreSQL Query Optimization - The Ultimate Guide to Building Efficient Queries*. Berkeley, CA: Apress, 2021. ISBN: 978-1-4842-6885-8. DOI: 10.1007/978-1-4842-6885-8. eprint: <https://link.springer.com/book/10.1007/978-1-4842-6885-8>. URL: <https://link.springer.com/book/10.1007/978-1-4842-6885-8>.
- [EH13] Peter Eisentraut und Bernd Helmle. *PostgreSQL-Administration* -. Köln: O'Reilly Germany, 2013. ISBN: 978-3-868-99362-2.
- [MW12] Bernd Müller und Harald Wehr. *Java Persistence API 2*. München: Carl Hanser Verlag GmbH & Co. KG, 2012. DOI: 10.3139/9783446431294. eprint: <https://www.hanser-elibrary.com/doi/pdf/10.3139/9783446431294>. URL: <https://www.hanser-elibrary.com/doi/abs/10.3139/9783446431294>.
- [WRY17] Brett Walenz, Sudeepa Roy und Jun Yang. "Optimizing Iceberg Queries with Complex Joins". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 1243–1258. ISBN: 9781450341974. DOI: 10.1145/3035918.3064053. URL: <https://doi.org/10.1145/3035918.3064053>.