FernUniversität in Hagen

– Fakultät für Mathematik und Informatik –

Lehrgebiet Datenbanken und Informationssysteme

Analyse und Optimierung der Datenbankabfragen 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

Marco Galster: *Analyse und Optimierung der Datenbankabfragen der Webseite des Wedekind Projektes*, © 01. Oktober 2023

ERKLÄRUNG

Ich erkläre, dass ich die Expose 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.

Hagen, 01. Oktober 2023

Marco Galster

INHALTSVERZEICHNIS

1 Expose 1
1.1 Ausgangslage 1
1.2 Ziel 2
1.3 Aktueller Forschungsstand 2
1.4 Vorgehen bei der Umsetzung 6
1.5 Vorläufige Gliederung der Abschlussarbeit 7

Literatur 8

ABBILDUNGSVERZEICHNIS

Abbildung 1.1 Ablauf einer Web-Anfrage 5

1 EXPOSE

1.1 AUSGANGSLAGE

Die Editions- und Forschungsstelle Frank Wedekind (EFFW) wurde 1987 in der Hochschule Darmstadt gegründet. Ihr Intention ist es, den lange vernachlässigten Autor der europäischen Moderne in die öffentliche Aufmerksamkeit zu bringen. Die Publikation der »Kritischen Studienausgabe der Werke Frank Wedekinds. Darmstädter Ausgabe« im Verlag Jürgen Häuser wurde 1994 direkt nach der Erschließung der Wedekind-Nachlässe in Aarau, Lenzburg und München begonnen und im Jahre 2013 abgeschlossen (8 Bände in 15 Teilbänden, jetzt in Wallstein Verlag). Die EFFW wurde im Sommer 2015 an die Johannes Gutenberg-Universität Mainz umgezogen.

Da Frank Wedekind heute zu einem der bahnbrechenden Autoren der literarischen Moderne zählt, aber bisher sehr wenig erforscht wurde, soll sich diese nun Ändern. Die nationalen und internationalen Korrespondenzen von und an Wedekind zeigen eine starke Vernetzung in der europäischen Avantgarde. Dies zeigt das die Wissenschaft um die Korrespondenzen von Wedekind eine immer größere Rolle spielen. Aktuell sind lediglich 710 der 3200 bekannten korrespondenzstücke veröffentlicht worden.

Um diese zu verändern entstand das Projekt »Edition der Korrespondenz Frank Wedekind als Online-Volltextdatenk« [Mar23], welches bei der EFFW angesiedelt ist und als Kooperationsprojekt an der Johannes Gutenberg-Universität Mainz, der Hochschule Darmstadt und der Fernuni Hagen umgesetzt wird und durch die Deutsch Forschungsgemeinschaft (Bonn) gefördert wird.

Hierbei werden sämtliche bislang bekannten Korrespondenz in die Online-Edition überführt. Diese Korrespondenz beinhaltet substantiell das literarhistorische und kulturgeschichtliche Wissen über die Kultur zwischen 1880 und 1918, indem das überlieferte Material zum einen transkribiert editiert und editionswissenschaftlich kommentiert wird. Und zusätzlich durch Kommentar die den historischen Kontexten inhaltlich erschließen.

Hierfür entstand das Pilotprojekt der Online-Volltextdatenbank für Briefe von und an Frank Wedekind, welches 2015 als Beta-Version freigeschalten wurde. Diese Projekt kann aktuell unter http://briefedition.wedekind.h-da.de eingesehen werden.

Die benutzerfreundliche Erfassung und Annotation der Briefe, ist eines der Hauptziele der konzeptionierten technischen Architektur. Die ist der Grund, warum die Präsentation-, Recherche- und Erstellungsebene vollständig webbasiert umgesetzt wurde. Die Briefe selbst, werden im etablierten TEI-Format gespeichert. Dies muss von den Editoren und Editorinnen nicht selbst eingegeben werden, sondern kann über einen entstanden WYSIWYG-Editor direkt eingegeben werden, welcher es bei der Speicherung in das

TEI-Format wandelt. Ebenfalls wurde hierbei auf eine modulare und unabhängige Architektur geachtet, wodurch die Komponenten im Nachgang auch von anderen Briefeditionen genutzt werden können.

1.2 ziel

Die aktuelle Umsetzung beinhaltet die bisher definierte Anforderungen komplett. Darunter fallen die Recherchemöglichkeiten, sowie auch die Eingabe und die Verarbeitung der Briefe. Ein größeres Problem hierbei ist die Performance der Oberfläche. Durch die langen Abfragedauern des Datenbestandes leidet die Akzeptanz der Anwendung.

Das Ziel der Arbeit ist es, die Abfragedauern zu verringern, wodurch die Performance der Oberfläche signifikant verbessert wird. Anhand von Performance-Messungen und einer Befragung der Benutzer und Entwickler, werden die größten Performance-Probleme ermittelt und bewertet. Anhand dieser Auswertungen ist dann das weitere vorgehen zu ermitteln.

Hierbei ist auch ein Vergleich mit anderen Technologien angedacht.

1.3 aktueller forschungsstand

Die Speicherveraltung des PostgreSQL-Server 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 sollte, mit dieser Einstellung wird das häufige schreiben des Buffers durch Änderung von Daten und Indexen auf die Festplatte reduziert. Die Einstellung *temp_buffers* die definiert wie groß der Speicher für temporäre Tabellen pro Verbindung maximal werden darf, sollte ebenfalls überprüft werden, da ein zu kleiner Wert bei großen temporären Tabellen zu einem signifikanten Leistungseinbruch führt, wenn die Tabellen nicht im Hauptspeichern sondern in einer Datei bearbeitet werden. Der *work_mem* definiert die Obergrenze des zur Verfügung gestellt Hauptspeichers pro Datenbankoperation wie effizientes Sortieren, Verknüpfen oder Filtern. Auch wird im Falle eines zu klein gewählten Speichers auf temporäre Dateien auf der Festplatte ausgewichen, was ebenfalls zu signifikanten Leistungseinbrüchen führt. Die *maintenance_work_mem* wird bei Verwaltungsoperation wie Änderung und Erzeugung von Datenbankobjekten als Obergrenze definiert. Aber auch für die Wartungsaufgaben `Vacuum`, die fragmentierte Tabellen aufräumt und somit die performance hebt.

Die Wartung des Datenbanksystems ist eine der wichtigen Aufgaben und sollte regelmässig durchgeführt werden, damit die Performance des Systems durch die Änderung des Datenbestandes nicht einbricht [EH13, S. 75]. Hierfür gibt es den `VACUUM`-Befehl, der 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 eingestellt 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 werden und auf Probleme hin untersucht werden. Hiermit kann sehr einfach die häufigsten bzw. langsamsten Anfragen ermittelt werden.

Für weitere Optimierungen müssen dann die Anfragen einzeln überprüft werden. Hierfür ist es sinnvoll die Ausführungspläne der Abfrage zu analysieren [EH13, S. 252]. Hierbei ist es wichtig die verschiedenen Plantypen und ihre Kosten zu kennen, sowie die angegeben Werte für die Plankosten zu verstehen [DNB21, S. 24–30]. Hinzu kommt noch, dass man den tatsächlich ausgeführten Plan mit dem ursprünglichen Plan vergleichen sollte [EH13, S. 254]. Eine er wichtigsten Aussage hierbei ist, ob die Zeilenschätzung akkurat war. Größere Abweichung weißen 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 eine *Kurze* oder eine *Lange* Abfrage ist. Im Falle von einer Kurzen Abfrage werden zuerst die Abfragekriterien geprüft. Wenn dies nicht hilft, werden die Indexe geprüft. Sollte dies auch keine Verbesserung bringen, wird die Abfrage nochmal genauer analysiert und so umgestellt, dass die restriktivste Einschränkung zuerst zutrifft. Bei Langen Abfragen sollte überprüft werden, ob es sinnvoll ist das Ergebnis in eine Tabelle zu speichern und bei Änderung zu aktualisieren. Wenn dies nicht möglich ist, sollten die nachfolgenden Schritte durchgeführt werden. Zuerst wird der restriktivste Join gesucht und geprüft ob dieser als ersten ausgeführt wird. Danach fügt man weitere Joins hinzu und prüft die Ausführungszeit und die Abfragepläne. Als nächstes wird geschaut, dass große Tabellen nicht mehrfach durchsucht werden. Bei Gruppierungen ist noch zu prüfen, ob diese früher durchgeführt werden können, ob die Abfragemenge zu verringern.

Des Weiteren können über das Modul `pg_stat_statements` Statistiken über die Aufrufe die an den Server gestellt wurden, ermittelt werden [Posb]. Hierbei kann ermittelt werden, welche der Anfragen am häufigsten gerufen werden und welche die längsten Laufzeiten besitzen.

Die *Java Persistence API* (JPA) wird als First-Level-Cache in Java-EE-Anwendung gehandhabt. Hierbei nehmen die Objekte einen von 4 Zuständen ein [MW12, S. 57]. Im *Transient* sind die Objekt erzeugt, aber noch noch in den Cache überführt worden. Wenn Sie in den Cache überführt werden, nehmen sie den Zustand *Verwaltet* ein. Für das löschen eines Objektes gibt es den Zustand *Gelöscht*, wodurch auch das Objekt aus der Datenbank entfernt wird. Als letzten Zustand gibt es noch *Losgelöst*, hierbei wird das Objekt aus dem Cache entfernt, aber nicht aus der Datenbank.

Eine Menge von Objekten wird als *Persistenzkontext*. Solange die Objekte dem Persistenzkontext zugeordnet sind, also den Zustand Verwaltet besitzen, werden diese auf Änderungen überwacht um diese am Abschluss mit

der Datenbank zu synchronisieren. In der Literatur nennt man das *Automatic Dirty Checking* [MW12, S. 61].

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]. Hiermit 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, die automatisch vor dem Aufruf erzeugt und danach wieder gelöscht werden. Dies hat 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] führt, muss hier darauf geachtet werden, nicht mehr benötigte Entities aus dem Persistenzkontext zu lösen.

Zusätzlich kann im JPA ebenfalls noch der *Second Level Cache* (L2-Cache) aktiviert werden. Dieser Cache 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]. Entgegen der Verwendung spricht, dass die Daten im Second Level Cache explizit über Änderungen informiert werden müssen, sonst werden beim nächsten Laden wieder die alten Werte geliefert. Ebenfalls benötigt so ein Cache einen höheren Bedarf an Arbeitsspeicher, in diesem dann die Daten parallel zur Datenbank bereitgestellt werden. Daher ist die Benutzung nur problemlos bei Entities, 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 Abfragen und die zurückgelieferten Ids der Entities gespeichert. Bei erneutem Aufruf dieser Abfrage werden die referenzierten Objekte aus dem Objekt-Cache zurückgegeben. Bei veränderten referenzierten Entities wird der Query-Cache nicht benutzt und die betroffenen Abfragen werden unverzüglich aus dem Query-Cache entfernt [MW12, S. 316].

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

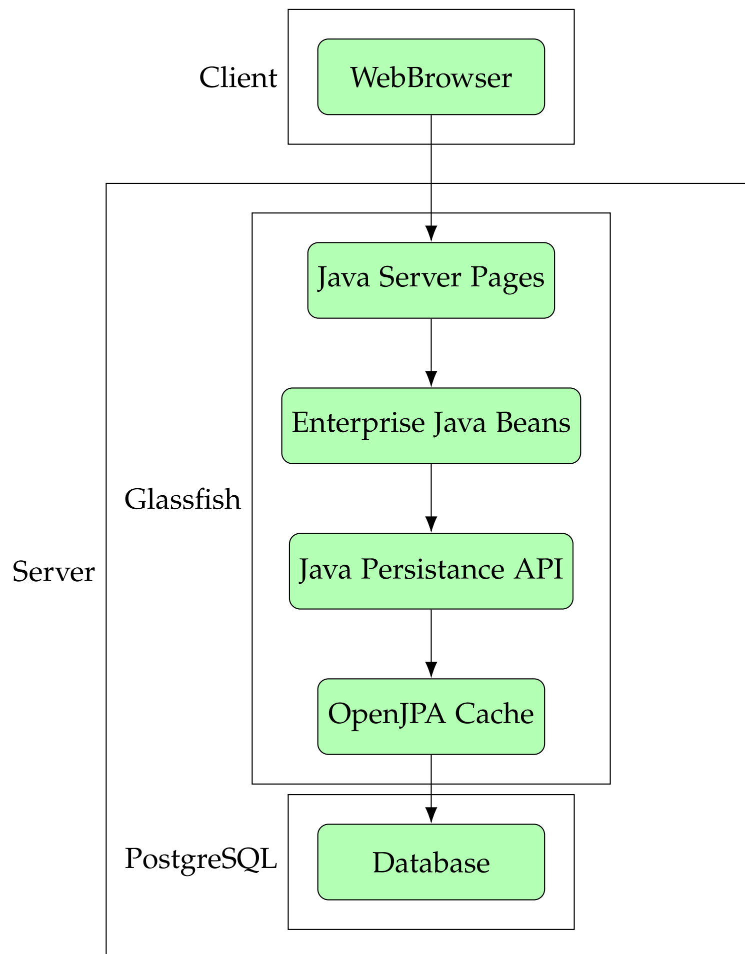

Abbildung 1.1: Ablauf einer Web-Anfrage

1.4 VORGEHEN BEI DER UMSETZUNG

Anhand der Umfrage der Bediener und Entwickler werden die größten Performance-Probleme in der Webseite ermittelt. Anhand dieser werden nun die dazugehörigen Quellcode identifiziert und analysiert. Hierbei müssen verschieden Blickwinkel betrachtet werden um die Performance zu optimieren.

Darunter fallen zum einen die Cache-Algorithmen der JDBC-Engine, sowie auch die Einstellungen am Datenbanksystem. Hierbei ist noch ein besonderes Augenmerk auf die vorhandene Serverkonstellation mit zu beachteten, da diese enormen Einfluss auf die Einstellungen bewirkt. Ebenso werden die Aufrufe im ganzen überprüft und untersucht um zu prüfen ob die Anfragen sich gegenseitig durch Transaktionen oder Locks sperren. Hierfür wird ebenfalls die interne Protokollierung der Aufruf aktiviert und dessen Ausgabe untersucht und analysiert.

Danach werden die Abfrage selbst untersucht und auf Optimierungen überprüft. Hierbei wird als erstes der Aufruf der Abfrage betrachtet. Dann wird die Abfrage selbst genauer untersucht. Dabei wird beachtet ob die Anfragen selbst viel Zeit für die Bearbeitung benötigen oder auf Ressourcen warten. Zum anderen wid geprüft ob durch gezielte Umstellung oder Einfügen von Zwischenergebnissen schnellere Abfragen möglich sind, Wie es in der Abhandlung Öptimizing Iceberg Queries with Complex Joins"[WRY17] gezeigt wird. Zum Schluss werden noch die Abfragekriterien und die vorhanden, beziehungsweise genutzten, Indizierungen überprüft.

Als letztes wird noch die Art des Aufrufers betrachtet. Hierbei wird die Art und Weise der Aufrufe genauer betrachtet. Ob zum Beispiel eine vorhandene Anfrage mehrfach verwendet wird und diese besser auf 2 ähnliche Abfrage aufgeteilt werden kann. Oder ob an den Stellen ein Paging eingebaut werden kann, um die übertragene Datenmengen zu reduzieren.

Zeitgleich wird der PostgreSQL sowie der Server selbst untersucht und die Einstellungen überprüft. Hierzu gehören die Größen der Speicher und die Wartungsaufgaben des Datenbanksystems. In diesem Zuge werden auch die Log-Dateien vom PostgreSQL, unter Zuhilfenahme von pgFouine, untersucht und auf Probleme und Unregelmässigkeiten geprüft.

1.5 VORLÄUFIGE GLIEDERUNG DER ABSCHLUSSARBEIT

1. Einleitung
2. Grundlagen
3. Konzept
4. Performance-Untersuchung
   - 4.1 Befragung der Benutzer und Entwickler
   - 4.2 Überprüfung des PostgreSQL und Servers
   - 4.3 Einbau und Aktivieren von Performancecounter
   - 4.4 Laufzeitanalyse starten
5. Optimierung
   - 5.1 Anpassung der Konfiguration
   - 5.2 Veränderung der Abfragen
   - 5.3 Veränderung der Webseite
6. Evaluierung
   - 6.1 Erneute Laufzeitanalyse starten
   - 6.2 Vergleich der Ergebnisse vor und nach der Optimierung
7. Zusammenfassung und Ausblick

LITERATUR

[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).

[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).

[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.

[Mar23] AK Prof. Dr. Ariane Martin. *Frank Wedekind | FB 05 - AK Prof. Dr. Ariane Martin*. 2023. URL: https://www.martin.germanistik.uni-mainz.de/forschung/frank-wedekind/ (besucht am 24. 09. 2023).

[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.