Multi-Layer Optimization Strategies for Enhanced Performance in Digital Editions: A Study on Database Queries, Caches, Java EE and JSF

Marco Galster

Universität in Hagen, Deutschland

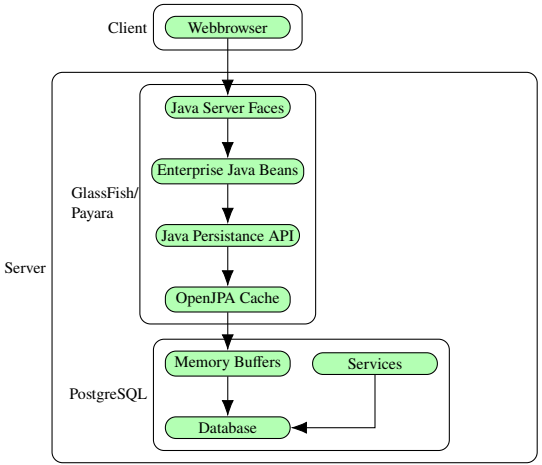
07. November 2024

Agenda

Problemstellung

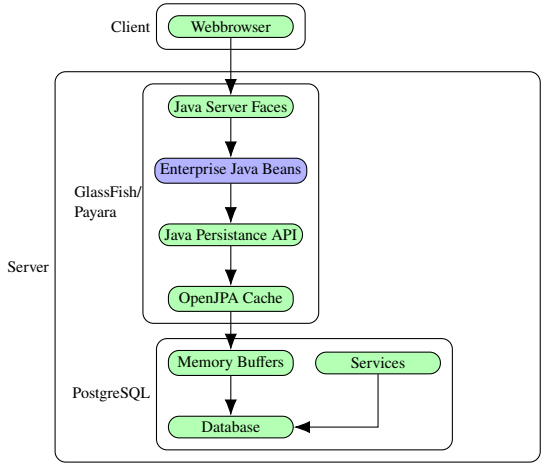
- Eine webbasierte Anwendung der Frank Wedekind Briefedition
- Performance Problem bei der Abfrage der Briefe sowie der Recherchen der Korrespondenzen
- entsprechend geringere Akzeptanz der Anwendung

Ablauf einer Web-Anfrage



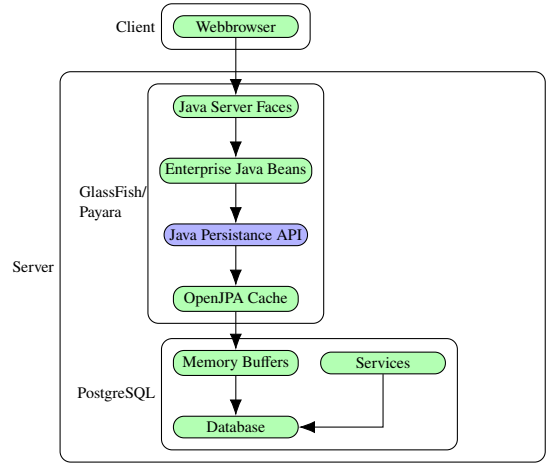
Anpassungen

- Caching EJB



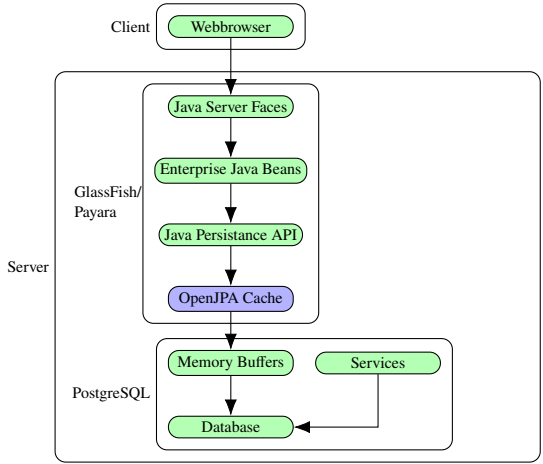
Anpassungen

- Abfragen in JPQL
- Abfragen in Criteria API

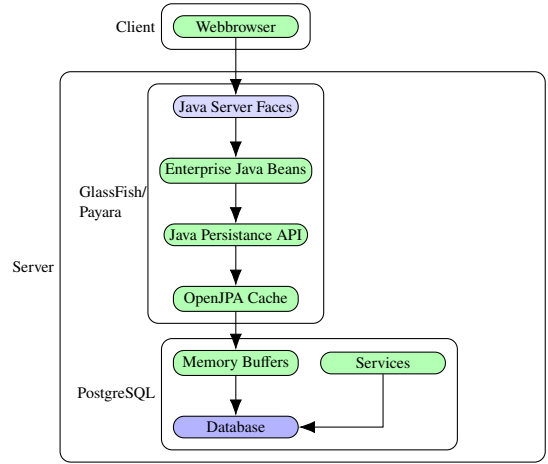


Anpassungen

- Caching in OpenJPA
- Cached Queries
- Caching mit Ehcache



Anpassungen



- Materialized Views
- Optimierung der Abfrage

Voraussetzungen

- Verwendung von Docker, zur Performance-Limitierung
- Eigene Container für die Datenbank und den Webserver
- Für die Untersuchung wird nur die Dokumentenliste beobachtet
- Verwendung von Scripts zur besseren Vergleichbarkeit und Wiederholgenauigkeit

Vorgehen

- Vor jeder Messung werden die Container neugestartet und die Startroutinen abgewartet
- Aufruf eines Bash-Script auf dem gleichen Rechner wie die Docker-Container um die Latenz des Netzwerkes auszuschließen
- Ermittlung des Speicherbedarfs vor und nach dem Webseitenaufrufen
- Messung der Aufruf der Index-Seite (ohne Datenbankaufrufe) und der Dokumentenliste, jeweils 10 mal
- Report über die SQL-Abfragen mit pgBadger erstellen

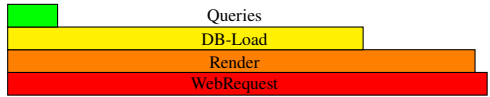
Erste Auffälligkeiten

- OutOfMemory-Ausnahme ausgelöst nach dem vierten Script Aufruf (~40 Webseitenaufrufe)
- Erhöhung des Java-Heapspeichers von 512MB auf 4096MB hat nur die Anzahl der Aufrufe bis zum Absturz verzögert

Vergleichsmessung - Ohne Cache

- Deaktivieren aller Caches
- Auffälliger Speicheranstieg, trotz deaktiviertem Cache
- Gleichmässige Anzahl an Datenbankabfragen
- Gleichmässige Laufzeit der Datenbankabfragen
- Laufzeitanteil der Datenbankabfragen unter 10%

#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
0	1931	-	-	1849	710
1	682	1223.0 / 30.3	54.8	666	399
2	389	1208.0 / 31.2	172.5	378	282
3	407	1208.0 / 33.5	110.0	398	307
4	359	1208.0 / 33.7	193.0	351	269
5	317	1208.0 / 32.9	107.0	309	235



Caching EJB

- Konfiguration über die Webseite des Payara-Servers
- habt keine Auswirkungen auf die Performance
- Der Cache wird nur von den Provider selbst und nicht den geladenen Datenbank-Objekten verwendet

Corollary

Nicht nutzbar für dieses Szenario

Abfragesprachen

- Ähnliche gemessene Zeiten
- Untersuchung im Debugger zeigt ähnlicher Abfrage-Syntax innerhalb von OpenJPA
- Prüfung der SQL-Abfragen zeigt, dass die identischen Befehle an die Datenbank übertragen werden
- Keine Unterschied im Speicherverbrauch feststellbar
- Gleiche Optimierung durch Hint `openjpa.FetchPlan.EagerFetchMode` (halbierte Laufzeit, geviertelte Datenbankaufrufe)
- Andere Hints hatte keine messbaren Auswirkungen

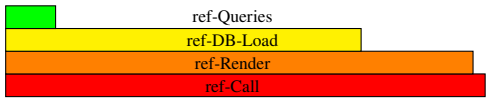
Corollary

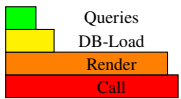
Daher kann die Art der Programmierung verwendet werden die für den Anwendungsfall die einfachere ist

Caching mit OpenJPA

- Einfaches aktivieren
- Direkte Unterstützung
- Konfiguration über die Anzahl der Objekte
- Messung mit 1000 und 10000 Objekten

#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	2347	-	-	2286	770
1	611	730.2 / 28.8	39.2	595	284
3	281	680.6 / 27.6	56.0	271	180
5	272	683.6 / 27.6	97.0	264	175
0	1904	-	-	2232	847
1	368	141.2 / 20.8	30.5	404	124
3	126	6.0 / 19.9	3.3	136	47
5	114	6.0 / 19.7	1.2	107	32

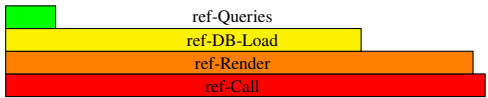


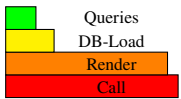


Caching mit OpenJPA

- Erwartete Reduzierung der Datenbankabfragen
- Laufzeit in der Datenbank halbiert sich nicht, trotz halbierter Abfragen
- Speicheranstieg wurde reduziert

#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	2347	-	-	2286	770
1	611	730.2 / 28.8	39.2	595	284
3	281	680.6 / 27.6	56.0	271	180
5	272	683.6 / 27.6	97.0	264	175
0	1904	-	-	2232	847
1	368	141.2 / 20.8	30.5	404	124
3	126	6.0 / 19.9	3.3	136	47
5	114	6.0 / 19.7	1.2	107	32





Cached Queries

- Einfaches aktivieren
- Einzelne Abfragen können ausgeschlossen werden
- Keine Auswirkung auf die Performance
- Wird nur beachtet wenn keine Argumente vorhanden sind

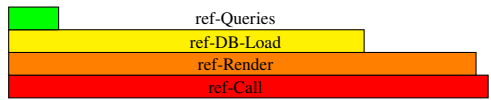
Corollary

Nur gut verwendbar, wenn keine Bedingungen vorhanden sind

Caching mit Ehcache

- Benötigt zusätzliche Pakete
- Aufwendigere Konfiguration, ohne Fehlerausgabe
- Zusätzliche Konfiguration bzw. Code zum aktivieren des Caches
- Weitere Konfiguration an den Klassen möglich

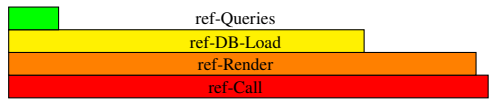
#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	2820	-	-	2809	1186
1	488	135.2 / 20.7	24.4	490	175
2	144	6.0 / 20.1	1.0	136	38
3	129	6.0 / 19.4	1.0	121	34
4	123	6.0 / 19.7	8.0	116	33
5	118	6.0 / 12.7	4.0	111	34

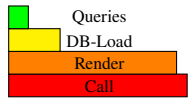


Caching mit Ehcache

- Erwartete Reduzierung der Datenbankabfragen
- Laufzeit der Datenbankabfragen halbiert sich nur trotz signifikant weniger Abfragen
- Starke Reduzierung der Laderoutine von Datenbank nach Java-Objekten
- Geringer Speicheranstieg trotz des Caches
- Sehr effizienter Cache, auch bei größeren Datenmengen

#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	2820	-	-	2809	1186
1	488	135.2 / 20.7	24.4	490	175
2	144	6.0 / 20.1	1.0	136	38
3	129	6.0 / 19.4	1.0	121	34
4	123	6.0 / 19.7	8.0	116	33
5	118	6.0 / 12.7	4.0	111	34

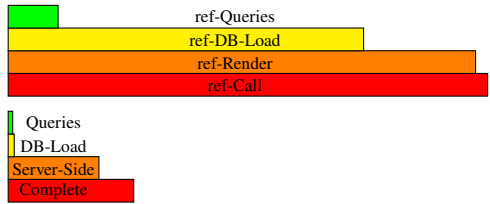




Abfragen über materialized views

- Materialized View und Webseite aus dem aktuellen Wedekind-Projekt übernommen [Dok]
- Zusätzliche Anpassung an der View um die Parameter in der Abfrage zu entfernen für Test mit QueryCache
- Verschiebung des JSON-Parsen in den Webclient

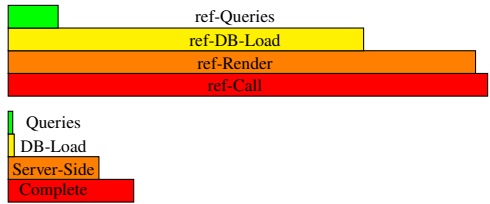
#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	859	-	-	803	334
1	348	16.8 / 2.5	36.4	331	174
2	194	9.0 / 2.4	2.6	185	99
3	161	9.0 / 2.4	3.0	152	77
4	145	9.0 / 2.4	-3.4	137	73
5	137	9.0 / 2.4	3.0	129	72
js	70+13	9.0 / 2.4	-	~60	4



Abfragen über materialized views

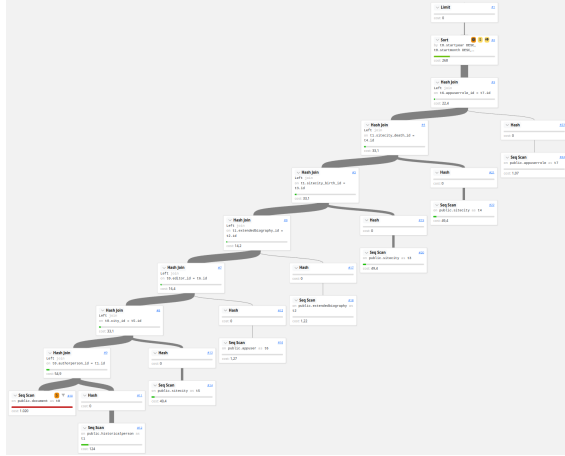
- Deutliche Reduzierung der Datenbankabfragen und -laufzeiten
- Unter-Abfragen werden als JSON-Objekte direkt hinterlegt
- Teuer beim erstellen, aber selten notwendig
- Geringe Schwankung der Aufrufzeiten
- Anteil der Datenbank nochmals reduziert
- Deutliche Reduzierung der DB-Load Laufzeit durch Verschiebung des JSON-Parsing

#	Call (ms)	Query (# / ms)	Memory Diff (MB)	Render (ms)	DB-load (ms)
ref-5	317	1208.0 / 32.9	107.0	309	235
0	859	-	-	803	334
1	348	16.8 / 2.5	36.4	331	174
2	194	9.0 / 2.4	2.6	185	99
3	161	9.0 / 2.4	3.0	152	77
4	145	9.0 / 2.4	-3.4	137	73
5	137	9.0 / 2.4	3.0	129	72
js	70+13	9.0 / 2.4	-	~60	4



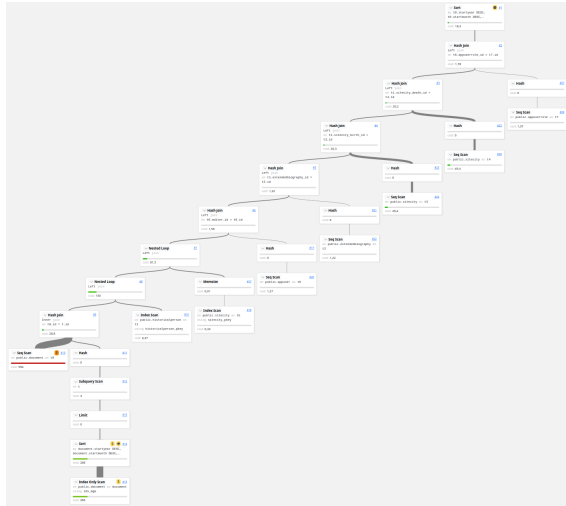
Optimierung der Abfrage

- Große Datenmenge vom ersten Befehlt bis fast zur Ausgabe
- Höchste Kosten im Seq Scan der Dokumententabelle
- Nur Seq Scan bei den verlinkten Tabellen
- Am Ende ist ein teurer Sort notwendig

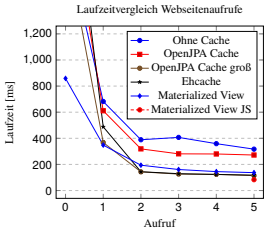


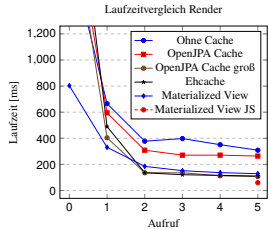
Optimierung der Abfrage

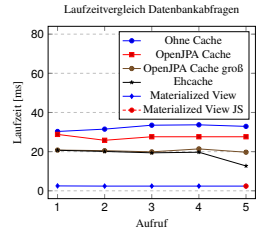
- Umstellung mit dem WITH-Statement
- Große Datenmenge verschwindet nach dem ersten Hash join
- Verwendung von Index Scan
- Kosten der Sortierung am Ende reduziert
- Reduktion der Laufzeit um den Faktor drei
- Nur möglich wenn die Bedingen ins WITH eingetragen werden können

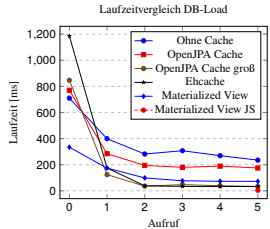


Vergleich

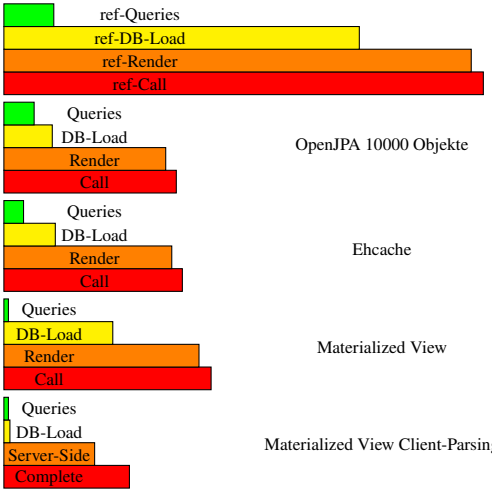

Laufzeitvergleich Webseitenaufrufe
Laufzeit [ms]
Aufruf
Ohne Cache
OpenJPA Cache
OpenJPA Cache groß
Ehcache
Materialized View
Materialized View JS


Laufzeitvergleich Render
Laufzeit [ms]
Aufruf
Ohne Cache
OpenJPA Cache
OpenJPA Cache groß
Ehcache
Materialized View
Materialized View JS


Laufzeitvergleich Datenbankabfragen
Laufzeit [ms]
Aufruf
Ohne Cache
OpenJPA Cache
OpenJPA Cache groß
Ehcache
Materialized View
Materialized View JS


Laufzeitvergleich DB-Load
Laufzeit [ms]
Aufruf
Ohne Cache
OpenJPA Cache
OpenJPA Cache groß
Ehcache
Materialized View
Materialized View JS

Vergleich der besten Laufzeiten



- Die Datenbankabfragen nehmen den kleinsten Teil der Laufzeit ein
- Beim Großteil der Optimierung wird die Zeit bei DB-Load verringert
- Die Differenz zwischen DB-Load und Render verändert sich gering
- Hoher Offset im DB-Load, außer bei Materialized View mit Client Parsing
- Materialized View geringste Datenbankabfragezeiten, aber nicht schneller als OpenJPA-Cache mit größerem Cache

Zusammenfassung

- Query-Cache und EJB-Cache nicht verwendbar
- Bei Verwendung von Cache auf Ehcache umstellen, OpenJPA-Cache ist ineffizienter
- Die Wahl der Abfragesprache hat keine Performance-Beeinflussung
- Verwendung der Materialized-View mit JSON-Parsing am Client hat die beste Performance
- Abfragen können optimiert werden, aber besitzen nur geringe Auswirkung auf den gesamten Aufruf
- Größtes Optimierungspotential im ORM vorhanden, bei der Bereitstellung der Entitäten (gut an der Umsetzung mit Materialized-View zu sehen)

Ausblick

- Weitestgehende optimale Umsetzung der Schichten im GlassFish-Server, bzw. wenige Optimierungsmöglichkeiten
- Verlagerung des Aufbaus der Darstellung an den Client um weniger Ressourcen am Server zu nutzen
- Speicherleck im OpenJPA durch neue Version behebbar
- Wechseln auf anderen ORM oder eigene Entwicklung
- Verwendung von Caches benötigen zusätzliche Ressourcen, daher nicht sinnvoll bei schwächeren Serversystemen

Referenzen

[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).

[Pos] 2023. URL: https://postgrespro.com/docs/postgresql/14/runtime-config-resource (besucht am 27. 12. 2023).

[Dok] *Dokumentenliste mit Native Query und Materialized View (b1f4c93d) · Commits · Wedekind / briefdb 2.0 · GitLab*. URL: https://code.dbis-pro1.fernuni-hagen.de/wedekind/briefdb-2.0/-/commit/b1f4c93d49ba31bf4da49845f47b5656e23aa76e (besucht am 21. 09. 2024).