Hibernate - Performance Tuning

Performance-Optimierung durch Caching in Hibernate

von Marc van den Bogaard

codecentric

1 EINFÜHRUNG 2

2 PROBLEMSTELLUNG 2

3 GRUNDLAGEN DES CACHING 2

4 CACHING IN HIBERNATE 2

4.1 CACHE-SCOPES 3

4.1.1 Transaction scope cache 3

4.1.2 Process scope cache 3

4.1.3 Cluster scope cache 3

4.2 CACHE-PROVIDER 4

4.3 FIRST-LEVEL-CACHE 4

4.4 SECOND-LEVEL-CACHE 4

4.4.1 Hibernate Konfiguration 4

4.4.2 Cache-Provider Konfiguration 7

4.4.3 Cache-Handling 8

4.5 QUERY-CACHE 9

5 MONITORING 9

6 OPTIMIERUNG 10

6.1 CACHE-CANDIDATES 10

6.2 CONCURRENCY-STRATEGIES 10

7 ZUSAMMENFASSUNG 10

8 ANLAGEN 10

8.1 LITERATURVERZEICHNIS 10

1 Einführung

Dieser Artikel gibt eine Einführung in die Caching-Architektur von Hibernate, und beschreibt die Funktionsweise der verschiedenen Caches und deren Konfiguration. Im weiteren Verlauf werden konkrete Optimierungsmöglichkeiten aufgezeigt, durch die das Performance-Potenzial des Hibernate-Cache ausgenutzt werden können.

2 Problemstellung

Hibernate ist einer der bekanntesten O/R Persistenz-Services für Java. Ein Grund dafür ist unter Anderem die steile Lernkurve, durch die sehr schnell Erfolge verbucht werden können und die Anwendung erste Objekte über Hibernate persistiert.
Mit steigender Komplexität der Anwendung, steigt aber auch schnell der Bedarf nach einer anwendungsindividuellen Konfiguration bzgl. Transaktionshandling und Caching.

In der Standard-Konfiguration ist der Cache von Hibernate bereits aktiviert. Die Potenziale des Caches werden allerdings erst durch eine, auf die Anwendung zugeschnittene Konfiguration optimal genutzt. Im Gegenteil dazu kann ein falsch konfigurierter Cache dazu führen, dass Daten im Cache nicht mehr synchron mit der Datenbank sind und Probleme in der Anwendung auftreten. Weiterhin sind nicht alle Anwendungen für einen Cache ausgelegt und es muss entschieden werden unter welchen Voraussetzungen der Cache aktiviert bzw. deaktiviert werden sollte.

3 Grundlagen des Caching

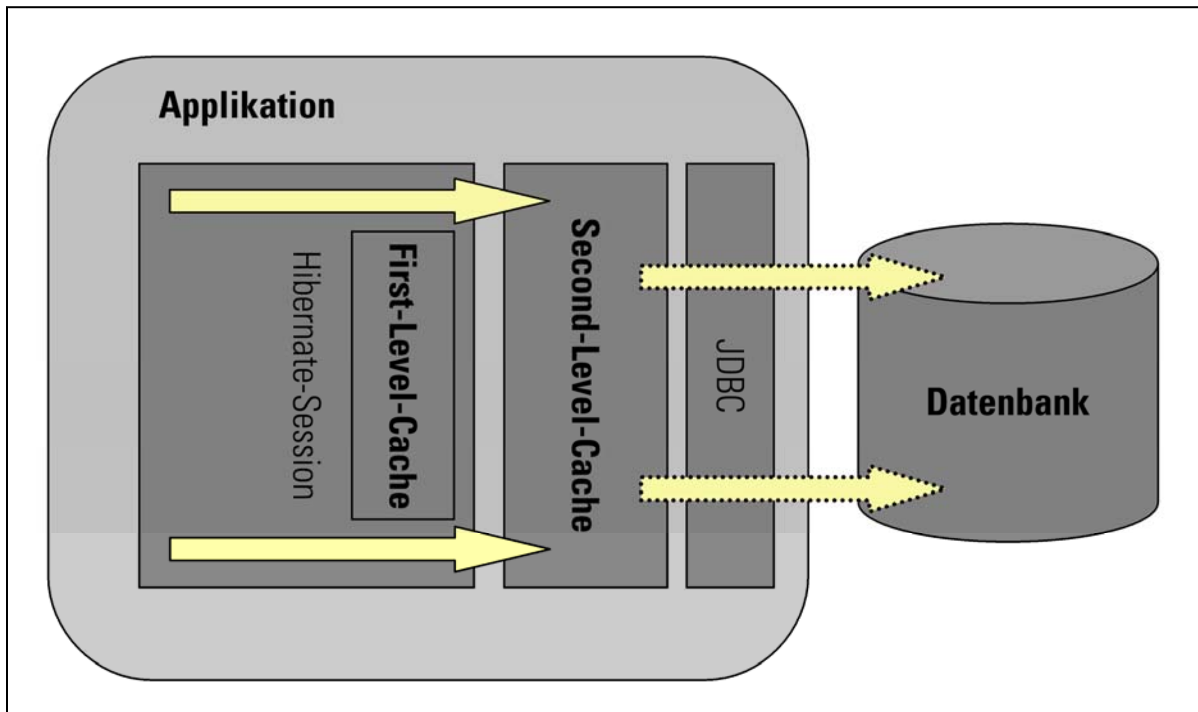
Caching bedeutet die Zwischenspeicherung von häufig abgefragten Daten in einem schnellen Medium, wie der lokalen Festplatte oder dem Arbeitsspeicher, um zeitaufwendige, langsame Zugriffe auf ein anderes Medium zu verhindern. Als Beispiele seien hier das Speichern von Webseiten auf dem lokalen Rechner (Browser-Cache) oder das Speichern von Datenbank-Daten im Arbeitsspeicher genannt. Im letzteren Fall können „teure" Zugriffe auf die Datenbank vermieden werden, indem zuvor ausgelesene Daten aus dem Arbeitsspeicher anstatt der Datenbank geholt werden. Dies bringt einen großen Geschwindigkeitsvorteil, da andernfalls die Daten über das DMBS von der Festplatte gelesen werden müssten.

Können Daten aus dem Cache gelesen werden spricht man von einem so genanten „Cache-Hit". Ein „Cache-Miss" liegt vor, wenn die angefragten Daten nicht im Cache gefunden werden konnten. In so einem Fall muss zuerst auf das Quell-Medium zugegriffen und die Daten anschließend im Cache gespeichert werden.

4 Caching in Hibernate

Hibernate arbeitet mit einer zweischichtigen Cache-Architektur, welche aus dem First- und dem Second-Level-Cache besteht. Der First-Level-Cache ist in der Grundform immer aktiv und bezieht sich auf eine „Unit-of-Work", d.h. meist auf einen Service-Aufruf, da der First-Level-Cache an die aktuelle Transaktion gebunden ist. Nach dem Service-Aufruf ist der Cache wieder geleert. Der First-Level-Cache kann demnach mehrfach existieren.
Der Second-Level-Cache hingegen muss für den Einsatz noch konfiguriert werden. Hibernate bringt hier selbst nur eine für Testzwecke gedachte Implementierung für einen Second-Level-Cache mit. Hier sollten die Schnittstellen für Third-Party Bibliotheken, genannt Cache-Provider (EhCacheProvider,

JBossCacheProvider, etc.), genutzt werden. Darauf aufbauend existiert ein Query-Cache der das Ergebnis einer Abfrage im Cache speichern kann.
Die unterschiedlichen Caches und Ihre Funktionsweisen werden im weiteren Verlauf des Artikels erläutert.



erläutert. „Cache-Scopes" geben in Hibernate die Gültigkeit und Laufzeit des Caches an. Unterschieden werden hierbei drei Arten von „Scopes":

1. Transaction scope cache
2. Process scope cache
3. Cluster scope cache.

4.1.1 Transaction scope cache

Beim „Transaction scope cache" ist der Cache an die aktuelle Transaktion gebunden. Jede Transaktion hat somit ihren eigenen Cache und kann von hier aus nicht von anderen Prozessen erreicht werden. Ist die Transaktion beendet wird auch der Cache geleert. Der First-Level-Cache ist ein Beispiel für einen „Transaction scope cache". Dieser ist, wie schon beschrieben, durch die Hibernate-Session an die aktuelle Transaktion gebunden. Ein „Transaction scope cache" kann mehrfach in einer JVM existieren.

4.1.2 Process scope cache

Der „Process scope cache" kann von mehreren Prozessen / Transaktion gleichzeitig erreicht werden und existiert für den gesamten Prozess nur einmal in einer JVM. Der Second-Level-Cache ist ein Beispiel für diesen Cache-Scope.

4.1.3 Cluster scope cache

Beim „Cluster scope cache" kann der Cache auch über JVM-Grenzen hinaus geshared werden. Hier werden spezielle Anforderungen an die Umgebung gestellt, da der Cache im gesamten Cluster synchron gehalten werden muss und die Netzwerk-Kommunikation eine entscheidende Rolle spielt.

4.2 Cache-Provider

Cache-Provider stellen die unterschiedlichen Third-Party Implementierung für einen Hibernate Second-Level-Cache dar. Hibernate selbst stellt einen auf einer Hashtable basierenden CacheProvider zur Verfügung, welcher aber nicht für einen produktiven Einsatz ausgelegt ist, da dieser über keine Einstellungsmöglichkeit wie „Time-to-Live" oder unterschiedliche Behandlung für Entities verfügt.

4.3 First-Level-Cache

Der First-Level-Cache oder auch Persistenz-Context genannt, ist die erste Caching-Schicht in Hibernate. Innerhalb einer Transaktion oder „Unit-of-work" wird der Cache verwendet und anschließend wieder geleert. Dieser Cache ist für alle Hibernate-Sessions obligatorisch. Eine Ausnahme bildet hier die Stateless-Session, die über keinen First-Level-Cache verfügt.

Der First-Level-Cache ist ein Zwischenspeicher für Objekte die von einem Query innerhalb einer Transaktion geladen wurden. Befinden wir uns innerhalb einer Transaktion werden alle an den Objekten getätigten Änderungen zuerst im First-Level-Cache gehalten und durch Hibernate „so spät wie möglich" an die Datenbank geschickt. Wird innerhalb dieser Session versucht Daten auszulesen, wird zuerst im First-Level-Cache geschaut ob diese bereits geladen wurden. Andernfalls wird ein Query auf die Datenbank abgesetzt. Primär versucht Hibernate hier die Anfragen an die Datenbank zu minimieren und/oder die Queries zu optimieren, indem bestimmte nicht notwendigen Abfragen weggelassen oder die Queries selbst von Hibernate optimiert werden z.B. Optimierung von Update-Statements.

4.4 Second-Level-Cache

Der Second-Level-Cache ist der eigentliche Cache von Hibernate, welcher auch die größten Performance-Potenziale mit sich bringt. Hierbei steigt allerdings auch die Komplexität der Einstellungen und damit die Auswirkungen auf die Applikation, d.h. hier können auch Probleme durch einen eingeschalteten Second-Level-Cache entstehen.

4.4.1 Hibernate Konfiguration

Der Second-Level-Cache verfügt über eine Reihe von Einstellungsmöglichkeiten, die zum einen in Hibernate selbst, zum Anderen in den Konfigurationseinstellungen der Third-Party Cache-Provider selbst eingestellt werden müssen. Der Second-Level-Cache ist in der Standardkonfiguration aktiviert. Über den Parameter **hibernate.cache.use_second_level_cache** kann dieses eingestellt werden. Für eine erfolgreiche Funktionsweise muss der Second-Level-Cache allerdings noch für die einzelnen Entities separat konfiguriert werden, damit diese gecached werden können. In der Standardkonfiguration für die Entities sind die Einstellungen deaktiviert.

Weiterhin muss auch ein entsprechender Cache-Provider ausgewählt werden. Andernfalls wird in der Standardkonfiguration der **Hashtable-Cache-Provider** verwendet, welcher nicht für den Einsatz in einer produktiven Umgebung ausgelegt ist und sonst auch keine Einstellungsmöglichkeiten mit sich bringt. In der Hibernate-Configuration wird der Cache-Provider über den Parameter **hibernate.cache.provider_class** angegeben. Hier können alle Cache-Provider angegeben werden, die das Interface **org.hibernate.cache.CacheProvider** implementieren. Auch eigene Implementierungen sind hier möglich. Der Cache-Provider muss als voll-qualifizierter Klassenname angegeben werden.

Die folgende Tabelle gibt eine Übersicht über die gängigen Open-Source- und kommerziellen Cache-Provider und deren Support für die verschiedenen Einsatzbereiche:

Cache	Provider-Klasse	Typ	Cluster-Safe	Query-Cache	Open-Source
Hashtable	org.hibernate.cache.HashtableCacheProvider	Speicher	nein	ja	ja
EHCache	org.hibernate.cache.EhCacheProvider	Speicher, Festplatte	nein	ja	ja
OSCache	org.hibernate.cache.OSCacheProvider	Speicher, Festplatte	nein	ja	ja
SwarmCache	org.hibernate.cache.SwarmCacheProvider	Cluster	ja	nein	ja
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	Cluster	ja	nein	ja
Tangosol Coherence	com.tangosol.coherence.hibernate.CoherenceCacheProvider	Speicher, Festplatte, Cluster	ja	ja	nein

Um den Second-Level-Cache für Entities nutzen zu können, muss dieser für die einzelnen Klassen und Collections noch konfiguriert werden.
Dies geschieht über das **Cache-Tag** in den entsprechenden Hibernate-Mapping-Files. Hier wird der Cache innerhalb der Entity-Definition konfiguriert und für die Collection innerhalb der Collection-Definition.
Das **Cache-Tag** verfügt über folgende Attribute:

Parameter	Beschreibung
Usage	transactional|read-write|nonstrict-read-write|read-only
Region	RegionName
Include	all|non-lazy

Die **Usage** gibt die Concurreny-Strategy und damit die Transaction-Isolation an. Da von mehreren Threads auf den Cache zugegriffen werden kann, ist das Transaktionshandling an dieser Stelle von entscheidender Bedeutung. Die einzelnen Strategien sind nach Isolationsgrad geordnet. Ein hoher Isolationsgrad bedingt mehr Transaktionssicherheit, führt aber auch zu Performance-Einbußen, da Locks länger auf den Tabellen gehalten werden müssen. Welches Isolationslevel gewählt wird, muss im Einzelfall für die Klasse entschieden werden.
Hier die Erläuterungen zu den einzelnen Strategien:

- **Transactional:** Vollkommene Transaktionssicherheit mit „repeatable-read"-Isolation. Sollte für Klassen eingesetzt werden, bei der inkonsistente Daten im Cache zu Problemen führt.
- **Read-Write:** Sichert die Synchronität zwischen Cache und Datenbank bis zu einem „read committed"-Isolationlevel. Sollte eingesetzt werden, wenn die Daten nur selten geändert werden.
- **Nonstrict-Read-Write:** Gibt keine Garantie das die Daten im Cache synchron mit der Datenbank sind. Sollte nur eingesetzt werden, wenn die Daten über Wochen / Monate nicht geändert werden. Um die Sicherheit zu erhöhen kann hier eine „Expire-Time", d.h. eine Zeit nach der die Daten automatisch ungültig und aus dem Cache entfernt werden, eingestellt werden. Die „Expire-Time" wird im Cache-Provider selbst eingestellt.

- **Read-Only:** Sollte nur für statische Daten eingesetzt werden. Auch hier kann über eine „Expire-Time", die Sicherheit erhöht werden.

Nicht alle Cache-Provider unterstützen jede Concurrency-Strategy. Im Folgenden eine Auflistung der bekannten Cache-Provider und deren Support für die unterschiedlichen Strategien. Eine eigene Concurrency-Strategy kann über das Interface **org.hibernate.cache.CacheConcurrencyStrategy** implementiert werden.

Concurrency Strategy	**Read-Only**	**Nonstrict-Read-Write**	**Read-Write**	**Transactional**
Hashtable	ja	ja	ja	nein
EHCache	ja	ja	ja	nein
OSCache	ja	ja	ja	nein
SwarmCache	ja	ja	nein	nein
JBoss Cache	ja	nein	nein	ja
Tangosol Coherence	ja	ja	ja	ja

Die gecachten Objekte werden im Second-Level-Cache in so genannten Cache-Regions gehalten. Cache-Regions gibt es für Klassen, Collections, Queries und zusätzlich für Tabellen-Timestamps (mehr dazu im Kapitel über Query-Caching). Beim Caching von Ojekten wird standardmäßig der voll-qualifizierte Klassenname als Regionname verwendet. Bei Collections entspricht der Regionname dem Klassennamen + dem Namen der Property innerhalb der Klasse. Über den Region-Parameter kann der Name individuell eingestellt werden. Soll für eine ganze SessionFactory ein anderes Prefix für den Regionname verwendet werden, kann dieser über den Parameter **hibernate.cache.region_prefix** in der Hibernate-Konfiguration eingestellt werden. Dies ist insbesondere dann wichtig, wenn mit mehreren SessionFactories und einem geclusterten Cache gearbeitet wird, da hier Cache-Regions mit gleichem Namen Probleme verursachen können.

Die Einstellung des Second-Level-Cache für die einzelne Klasse findet im entsprechenden Mapping-File statt. Es ist zu beachten, dass die Einstellungen sowohl für die Klasse selbst, als auch für die zugehörigen Collections gemacht werden müssen (falls die Collection gecached werden soll).

Hier eine Beispiel-Konfiguration:

```
<class name="de.codecentric.persistenz.modell.Benutzer" table="benutzer">
        <!-- Verwende für die Klasse Benutzer eine Read-Write-Strategy -->
        <cache usage="read-write"/>

        <!-- Liste der zugehörigen Gruppen -->
        <set name="gruppen">

                <!-- Verwende für die Collections „Gruppen" eine Read-Write-Strategy -->
                <cache usage="read-write"/>
        </set>
</class>
```

4.4.2 Cache-Provider Konfiguration

Zusätzlich zu den Konfigurationsmöglichkeiten in Hibernate bieten die verschiedenen Cache-Provider selbst Einstellungsmöglichkeiten für das Verhalten des Cache. Hierzu gehören u.A. Einstellungen für die „Expire-Time" für Objekte im Cache, wie viele Objekte maximal im Cache gehalten werden und ab welcher Größe diese zusätzlich auf die Festplatte geschrieben werden.

Im Weiteren soll hier die Konfiguration des Easy-Hibernate-Caches (EhCache) erläutert werden.
Der EhCache ist einer der bekanntesten Vertreter für JVM-Level Cache-Provider und wird mit Hibernate direkt ausgeliefert. Zudem ist der EhCache eine vollständige Implementierung des JSR107 JCache API.

Die Konfiguration des EhCache erfolgt in einer **ehcache.xml** die mit ins Hauptverzeichnis des Classpath gelegt werden kann. Hibernate findet diese hier automatisch. Kann diese Datei nicht gefunden werden, wird eine Standardkonfiguration des EhCaches verwendet, die im JAR mit enthalten ist (**ehcache-failsafe.xml**).

Hier eine Beispiel-Konfiguration für den EhCache:

```
<ehcache>
	<diskStore path="java.io.tmpdir"/>
	<defaultCache
		maxElementsInMemory="3000"
		eternal="true"
		overflowToDisk="true"/>
	<cache name="de.codecentric.persistenz.modell.Benutzer "
		diskPersistent="true"
		maxElementsInMemory="200"
		eternal="false"
		timeToIdleSeconds="200"
		timeToLiveSeconds="400"
	/>
</ehcache>
```

Die Konfiguration kann für alle Klassen (defaultCache) oder klassen-individuell (cache name="de.codecentric…") vorgenommen werden. Hier die Erläuterungen der wichtigsten Parameter:
diskStore: Pfadangabe wo die Objekte auf der Festplatte gespeichert werden sollen. In diesem Beispiel wird das Temp-Verzeichnis der JVM genommen.
maxElementsInMemory: Maximale Anzahl der Objekte im Speicher, bevor diese auf die Festplatte (wenn eingestellt) geschrieben werden.
eternal: Ist dieser Parameter auf true, werden die „Expire-Times" ignoriert und die Objekte verbleiben „ewig" im Speicher.
overflowToDisk: Einstellung ob die Objekte nach Überschreitung des „maxElementsInMemory"-Parameter auf die Festplatte geschrieben werden sollen.
timeToIdleSeconds: Die maximale Zeit seit dem letzten Zugriff auf dieses Objekt, bevor dieses aus dem Cache entfernt wird.
timeToLiveSeconds: Die maximale Zeit zwischen der Erstellung und Löschung eines Objektes aus dem Cache.

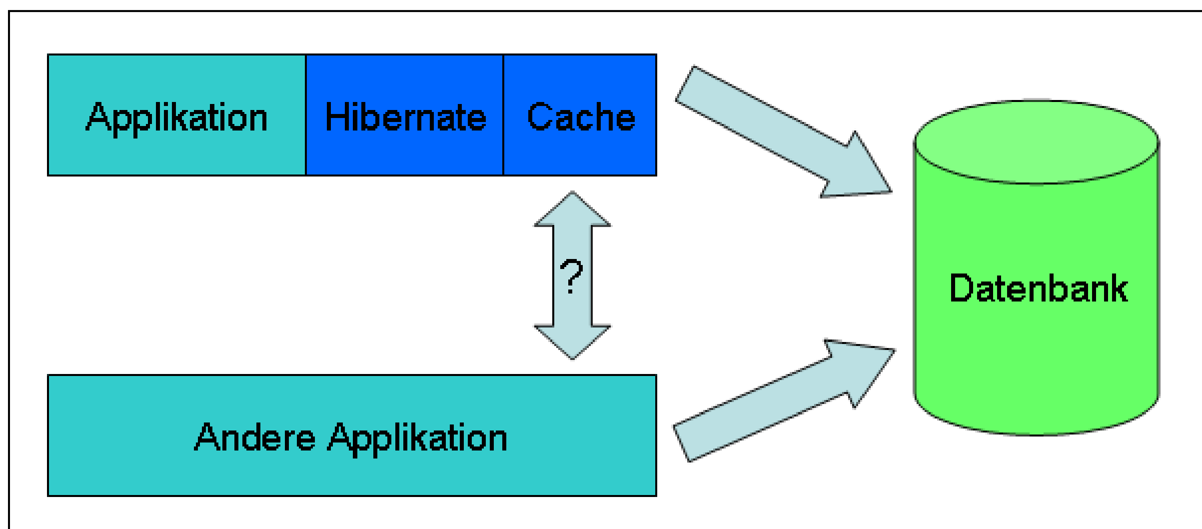
4.4.3 Cache-Handling

Bei eingeschaltetem Second-Level-Cache findet bei jeder Abfrage eine Prüfung des Caches statt. Der Second-Level-Cache wird allerdings nur benutzt, wenn beim Auslesen der Entity direkt über die ID gegangen wird, z.b. mit get(Long id). Ein Aufruf über HQL "... WHERE id = ?" würde den Second-Level-Cache nicht beanspruchen. Hier würde der Query-Cache aktiv werden. Gibt es einen „Cache-Miss", werden die Daten aus der Datenbank ausgelesen und anschließend im Cache gespeichert. Die Speicherung findet allerdings nicht als JDBC-Resultset statt, da dadurch weiterhin ein Cursor auf die Datenbank gehalten würde, sondern in einer anderen Form. Die Daten werden ähnlich der Serialisierung in „disassembled" oder mit den Worten von Hibernate „dehydrated"-Form gespeichert. Es werden keine Objekt-Instanzen gespeichert, sondern nur die Objekt-ID's mit zugehörigen Werten.
Hier ein Beispiel für die Speicherform der Objekte für ein Benutzer-Objekt:

```
{
324 => [max, mustermann, 29]
12   => [peter, mueller, 18]
154 => [arnold, peters, 54]
}
```

Durch den Parameter **hibernate.cache.use_structured_entries** können die Objekte lesbarer gespeichert werden, da dabei zur Speicherung eine Hash-Map verwendet wird. Dies ist insbesondere dann wichtig, wenn bei der Optimierung des Caches Statistiken erstellt werden (dazu mehr im Kapitel „Monitoring").

Der Second-Level-Cache kann Objekte nur entgegennehmen und Cachen, wenn dieser über die für den Cache konfigurierte SessionFactory getätigt wird. An dieser Stelle ist er sozusagen „one-way". Werden Daten „an Hibernate vorbei" in die Datenbank geschrieben, bleiben die veralteten Wert im Cache. Damit ist der Stand im Cache unbrauchbar und Probleme mit inkonsistenten Ständen sind vorprogrammiert.



SessionFactory bietet für diesen Mechanismus die Methode **evict()**, welche in verschiedenen Überladungen zur Verfügung steht und den Cache je nach Aufruf leeren kann:

```
evict(Class persistentClass)
evictCollection(String roleName, Serializable id)
evictQueries()
```

4.5 Query-Cache

Der Query-Cache von Hibernate ermöglicht es die Ergebnisse von Queries zu cachen. Insbesondere das Cachen von Collections als Ergebnis einer Abfrage kommt hier zur Geltung.
Der Query-Cache ist in der Standardkonfiguration deaktiviert und muss zum einen in der Hibernate-Konfiguration über den Parameter **hibernate.cache.use_query_cache** aktiviert werden, zum Anderen muss jedes Query für den Query-Cache einzeln eingestellt werden. Dies geschieht über die Methode **setCachable(boolean cachable)**.

Hier ein Beispiel wie der Query-Cache für eine Abfrage eingeschaltet werden kann:

```
Criteria criteria = session.createCriteria(Benutzer.class);
criteria.add(Restrictions.eq("vorname", vorname));

criteria.setCacheable(true);
```

Wird ein Query wie oben gezeigt für den Query-Cache eingestellt, werden die Ergebnisse im Cache gespeichert. Hierzu werden zwei neue Cache-Regions angelegt. Zum einen eine StandardCacheRegion und zum anderen eine UpdateTimestampsRegion. In der StandardCacheRegion werden nicht die „assembleten" Objekte gespeichert, sondern nur eine Liste von IDs. Die einzelnen Objekte liegen in der für die Klasse konfigurierten Cache-Region im Second-Level-Cache. D.h. ohne die Aktivierung des Second-Level-Cache ist auch kein Caching der Queries möglich. Die UpdateTimestampsRegion speichert einen Timestamp zu jeder Tabelle, die für den Cache ausgelesen wurde und hält somit den Query-Cache synchron. Wird eine Tabelle nun durch ein Insert-,Update- oder Delete-Statement aktualisiert, wird der Query-Cache für diese Abfrage invalidiert.

5 Monitoring

Hibernate bietet die Möglichkeit statistische Daten über die getätigten Abfragen und den Zugriff auf den Cache zu sammeln. Diese Daten können zur Optimierung und Kontrolle des Cache-Zugriffs verwendet werden.
Weiterhin bieten die unterschiedlichen Cache-Provider Möglichkeiten zur Analyse des Caches.

Über den Parameter **hibernate.generate_statistics** kann das Sammeln von Informationen eingeschaltet werden. Die gesammelten Informationen können dann über:

```
Statistics statistics = SessionFactory.getStatistics();
statistics.getSecondLevelCacheHitCount();
statistics.getSecondLevelCacheMissCount();
```

Eine weitere Möglichkeit statistische Daten über Zugriffe auf den Cache und die getätigten Abfragen zu erhalten ist der Zugriff auf den **org.hibernate.jmx.StatisticsService** von Hibernate. Dieser Service ist als MBean implementiert und kann auf dem JMX-Server registriert werden. Von da an hat man Zugriff auf Methoden wie getQueryCacheHitCount(), getQueryCacheMissCount(), getQueryCachePutCount(), etc.

6 Optimierung

6.1 Cache-Candidates

Ist der Cache in der Grundform konfiguriert und wurde ein Cache-Provider ausgewählt, gilt es die so genannten „Cache-Candidates" zu finden. „Cache-Candidates" sind potenzielle Klassen für den Cache. Nicht jede Klasse eignet sich dafür gecached zu werden. Nicht geeignet sind z.B. Klassen die sensible Daten wie Finanzdaten oder Preise für Produkte in E-Shops oder Online-Banking-Applikation enthalten. Weiterhin muss auch darauf geachtet werden, wie das Verhältnis zwischen den Lese/Schreib-Operation ist. Ist dieses Verhältnis relativ ausgeglichen, muss hier entschieden werden, ob der Aufwand den Cache zu pflegen, nicht den Performance-Zuwachs durch das Caching beeinflusst, da jede Schreib-Operation den Cache für die betroffenen Objekte invalidiert. Mit Hilfe der Statistiken die über Hibernate geliefert werden, kann hier geprüft werden, ob sich ein Caching der betroffenen Daten lohnt.

6.2 Concurrency-Strategies

Je nach gewählter Concurrency-Strategy steigt die Performance und sinkt die Aktualität / Transaktionssicherheit der Daten. Auf der anderen Seite sinkt die Performance mit steigender Aktualität / Transaktionssicherheit der Daten. Grundsätzlich bietet eine Read-Write Concurrency-Strategy ein ausgeglichenes Maß an Performance und Sicherheit, bei der zum einen die Transaktionssicherheit bis zu einem „read-committed"-Isolationlevel gewährleistet ist, zum Anderen die Pflege des Caches nicht merklich die Performance beeinflusst.

7 Zusammenfassung

Durch den Einsatz eines Second-Level-Cache ist es möglich die Performance in Hibernate entscheidend zu verbessern und die Potenziale von Hibernate voll auszuschöpfen. Hibernate verfügt hier über eine komplexe Caching-Architektur die jedoch individuell konfiguriert werden muss. Ob das Caching für bestimmte Klassen aktiviert wird und wie die entsprechenden Einstellungen vorgenommen werden, muss im Einzelfall und je nach Anwendungsfall entschieden werden. Für eine weitere Optimierung empfiehlt es sich, die von Hibernate generierten Statistiken über die Zugriffe auf den Cache zur Hilfe zu nehmen. Hier bietet es sich auch an den Statistik-Service für das Produktiv-System zu konfigurieren, um das Cache-Verhalten unter Live-Bedingungen zu beobachten.

8 Anlagen

8.1 Literaturverzeichnis

Buch: Java-Persistenz with Hibernate, von Christian Bauer und Gavin King
www.hibernate.org
www.springframework.org
http://ehcache.sourceforge.net
http://www.opensymphony.com/oscache
http://swarmcache.sourceforge.net
http://www.jboss.com/products/jbosscache
http://tangosol.com