Hibernate Second-Level Caching mit JBoss Cache und Infinispan

The Next Level

Wenn man JEE-Anwendungen schreibt, denkt man früher oder später über Performanceoptimierungen nach. Unnötige Roundtrips vom Applikationsserver zur Datenbank lassen sich durch gezieltes Second-Level Caching vermeiden. Im Cluster betrieben, stellt sich schnell die Frage, wie die Konsistenz der Caches auf den einzelnen Knoten sichergestellt werden kann. Mit Hibernate als JPA-Provider sollte man sorgsam überlegen, welche konkrete Cache-Implementierung am besten zu verwenden ist. Der vorliegende Artikel zeigt anhand der Beispiele JBoss Cache und Infinispan, wie sich verteiltes Second-Level Caching mit Hibernate in der Praxis realisieren lässt, und was beim Cachen von Entities und Queries zu beachten ist.

von Dirk Mascher

Angenommen erste Performancemessungen in einem beispielhaften JEE-Projekt haben aufgezeigt, dass bestimmte Datenbanktabellen sehr häufig abgefragt werden. Die Daten in den betreffenden Tabellen sind relativ stabil und werden nur selten verändert. In der Regel handelt es sich bei solchen Daten um Referenz- oder Stammdaten. Sie sind gut geeignet, vom Applikationsserver zwischengespeichert zu werden, womit die Anzahl der Roundtrips zur Datenbank verringert und die Last des Datenbankservers reduziert wird. Gehen wir weiter davon aus, dass wir von den Performancetestern in unserem Projekt genau diese Vorgabe erhalten haben. Die Lösung für das Problem liegt schnell auf der Hand: Wir aktivieren in Hibernate [1], dem JPA-Provider unserer Anwendung, „einfach" den Second-Level Cache, ahnen aber schon, dass es wohl nicht ganz so einfach wird. Das Deployment Setup unserer Anwendung sieht in der Produktion mehrere Knoten vor, und es muss sichergestellt werden, dass in den lokalen Caches auf den jeweiligen Knoten immer nur gültige Daten vorgehalten werden. Als zusätzliche Hürde müssen wir herausfinden, welcher konkrete Cache-Provider am besten einzuklinken bzw. zu konfigurieren ist.

Suche nach dem passenden Cache-Provider

Ein erster Blick in die offizielle Dokumentation von Hibernate [2] liefert als vermeintlich einzigen Cache-Provider, der sowohl transaktional als auch Clusterfähig ist und auch das Cachen von Query-Ergebnissen unterstützt, JBoss Cache [3]. Ansonsten wird zwar auf die unterschiedlichen Arten des Second-Level Caching eingegangen und auch die allgemeine, sprich providerunabhängige Konfiguration beschrieben, sobald man aber eine konkrete Cache-Implementierung in einem realen Projekt einklinken will, erweist sich die offizielle Dokumentation als wenig hilfreich. Unglücklicherweise gibt es sogar in der aktuellen (bezogen auf den Zeitpunkt der Erstellung dieses Artikels) und ansonsten eigentlich sehr guten Dokumentation von Hibernate (Version 3.6) noch folgenden Fehler, der den Einstieg in die Thematik unnötig erschwert: Das Kapitel „The Second Level Cache" dokumentiert, dass konkrete Cache-Provider das Interface *org.hibernate.cache.CacheProvider* implementieren müssen und über das Hibernate Property *hibernate.cache.provider_class* eingebunden werden. Das Interface ist jedoch seit Version 3.3 von Hibernate als *@deprecated* markiert und wurde durch *org.hibernate.cache.RegionFactory* ersetzt. Konkrete *RegionFactory*-Implementierungen werden über das neue Hibernate Property *hibernate.cache.region.factory_class* eingebunden. Die API-Änderung für die Einbindung von Cache-Providern trägt dem Umstand Rechnung, dass beispielsweise das Cachen von Entitäten andere Charakteristika aufweist, als das Cachen von Query-Ergebnissen. Insbesondere der interne *UpdateTimestamps*-Cache, über den sichergestellt wird, dass zwischengespeicherte Query-Ergebnisse (noch) aktuell sind, verlangt eine andere Semantik als dies für die anderen Cache-Regions notwendig ist (Kasten: „Hibernate Second-Level Caching in a Nutshell"). Somit sollte es möglich sein, Cache-Regions mit unterschiedlichen Semantiken auf entsprechend konfigurierte Cache-Instanzen abzubilden, um beispielsweise unnötig teures Pessimistisches Locking oder unnötig aufwändige clusterweite Replizierung beim internen Zugriff auf Objekte im Entity-Cache zu vermeiden. Genau dies war mit dem alten *CacheProvider*-API aber nicht möglich, sondern wird erst über das neue *RegionFactory*-API unterstützt. Auf der Suche nach weiterführenden Quellen ist man schnell versucht, einen Blick in die offizielle Dokumentation von JBoss Cache zu werfen [4]. Im zugehörigen Wiki

[5] findet man erste konkrete Hinweise. Als extrem wertvoll stellt sich der Verweis auf den JBoss-Applikationsserver heraus: JBoss integriert bis einschließlich Version 5.x JBoss Cache und bringt u. a. optimierte Konfigurationen für die Verwendung als Hibernate Second-Level Cache mit. Interessanterweise findet man in dem Wiki zu JBoss Cache aber auch einen prominent platzierten Hinweis, besser dessen offiziellen Nachfolger Infinispan [6] zu verwenden. Obwohl es sich dabei um eine Data-Grid-Plattform handelt (also um wesentlich mehr als „nur" einen verteilten Cache), stellt man nach kurzer zusätzlicher Recherche fest, dass Infinispan tatsächlich auch als vollwertiger Second-Level Cache für Hibernate eingesetzt werden kann. Infinispan weist dabei dieselben Merkmale wie JBoss Cache auf, nämlich Transaktionalität, Clusterfähigkeit und Unterstützung von Query Caching. Nochmalige Recherche ergibt, dass Infinispan auch innerhalb des

Hibernate Second-Level Caching in a Nutshell

Im Gegensatz zum First-Level Cache von Hibernate, der ja bekanntlich auf der Ebene der Hibernate *Session* (i. a. also auf der Ebene der umschließenden Transaktion) operiert, wird der Gültigkeitsbereich des Hibernate Second-Level Cache über die Hibernate *SessionFactory* definiert. Da es pro Anwendung i. d. R. genau eine *SessionFactory*-(bzw. eine JPA-*EntityManagerFactory*-)Instanz gibt, stehen Objekte, die im Second-Level Cache zwischengespeichert werden, also transaktionsübergreifend zur Verfügung. Im Hibernate Second-Level Cache können folgende Objekte gecacht werden:

- Entities
- Collections (Assoziationen)
- Query-Ergebnisse

Streng genommen ist der Query-Cache jedoch ein separater Cache, der nur in Kombination mit dem Second-Level Cache verwendet werden kann, da Query-Ergebnisse nicht in Form von Entities, sondern lediglich als Primärschlüssel bzw. IDs der referenzierten Objekte zwischengehalten werden. Beim Auflösen von Query-Ergebnissen über den Query-Cache finden intern also immer n+1-Cache-Abfragen statt: eine Abfrage in den Query-Cache, gefolgt von n Abfragen in den eigentlichen Second-Level Cache, um die referenzierten Entities zu holen.

Konfiguration

Der Hibernate Second-Level Cache wird explizit über das Hibernate Property *hibernate.cache.use_second_level_cache* in der *hibernate.cfg.xml* bzw. in der *persistence.xml* aktiviert, der Query-Cache über das Hibernate Property *hibernate.cache.use_query_cache*. Den eigentlichen Cacheprovider klinkt man über das Hibernate Property *hibernate.cache.region.factory_class* ein.

Aktivierung

Per Default werden nur die Entities gecacht, deren Klassen explizit mit der Annotation *@org.hibernate.annotations.Cache* oder *@javax.persistence.Cacheable* markiert sind. Für Collections, d. h. abgebildete 1:n- oder n:m-Assoziationen gilt im Wesentlichen das Gleiche, nur dass hier die Standard-JPA-Annotation nicht verwendet werden kann, da diese nur auf Typen anwendbar ist. Sollen Assoziationen gecacht werden, müssen die entsprechenden Getter in den zugehörigen Entity-Klassen also explizit mit der Annotation *@org.hibernate.annotations.Cache* markiert werden.

Sollen Query-Ergebnisse gecacht werden, muss das pro betreffendem Query mittels providerspezifischer JPA Query Hints explizit definiert werden. Im Falle von Hibernate ist hierbei der Hint *org.hibernate.cacheable* auf den Wert *true* zu setzen.

Cache-Regions

Hibernate definiert für das Second-Level und Query Caching sog. Cache-Regions. Diese ermöglichen die logische Trennung unterschiedlicher Daten im Cache und bilden den Aufhänger, um z. B. gezielt für eine Entity-Klasse konkrete Eviction Policies definieren zu können. Für Entities und Collections können Cache-Regions abweichend vom Default über die Annotation *@org.hibernate.annotations.Cache* festgelegt werden. Bei Queries werden Cache-Regions über den JPA Query Hint *org.hibernate.cacheRegion* definiert.

Der Query-Cache und der UpdateTimestamps-Cache

Eine Besonderheit beim Query Caching ist der interne *UpdateTimestamps*-Cache, über den die Gültigkeit gecachter Query-Ergebnisse sichergestellt wird. Beim Cachen von Queries werden nicht nur die eigentlichen Ergebnisse (in Form von Primärschlüsseln bzw. IDs) gespeichert, sondern zusätzlich pro Query auch der Zeitstempel seiner Ausführung. Ruft man später dieselbe Query mit identischen Parametern wieder auf, wird der Zeitstempel verwendet, um über den *UpdateTimestamps*-Cache herauszufinden, ob in den betreffenden Tabellen in der Zwischenzeit Daten hinzugefügt, gelöscht oder verändert wurden. Ist dies der Fall (unabhängig davon, ob die veränderten Daten die Ergebnismenge des Query tatsächlich beeinflusst hätten), wird die SQL-Abfrage nochmal ausgeführt. Die Ergebnisse werden dann zusammen mit dem Zeitstempel der neuen Ausführung erneut zwischengespeichert. Hibernate stellt intern sicher, dass der *UpdateTimestamps*-Cache bei jedem Insert, Update und Delete transparent und konsistent gepflegt wird. Im Cluster bedeutet das, dass lokale Veränderungen des *UpateTimestamps*-Cache auf einem Knoten transaktional in alle anderen Konten repliziert werden müssen.

Das n+1-Problem

Im besten Fall, d. h. wenn die Ergebnisse eines Hibernate Queries bereits vollständig gecacht sind, finden intern n+2 lokale Cache-Lookups statt. An die Datenbank wird dabei kein SQL Query geschickt. Im schlimmsten Fall, d. h. wenn der Query und die referenzierten Objekte noch nicht gecacht sind (oder das Ergebnis des Query nicht mehr gültig ist), erfolgen insgesamt n+1 Roundtrips zur Datenbank.

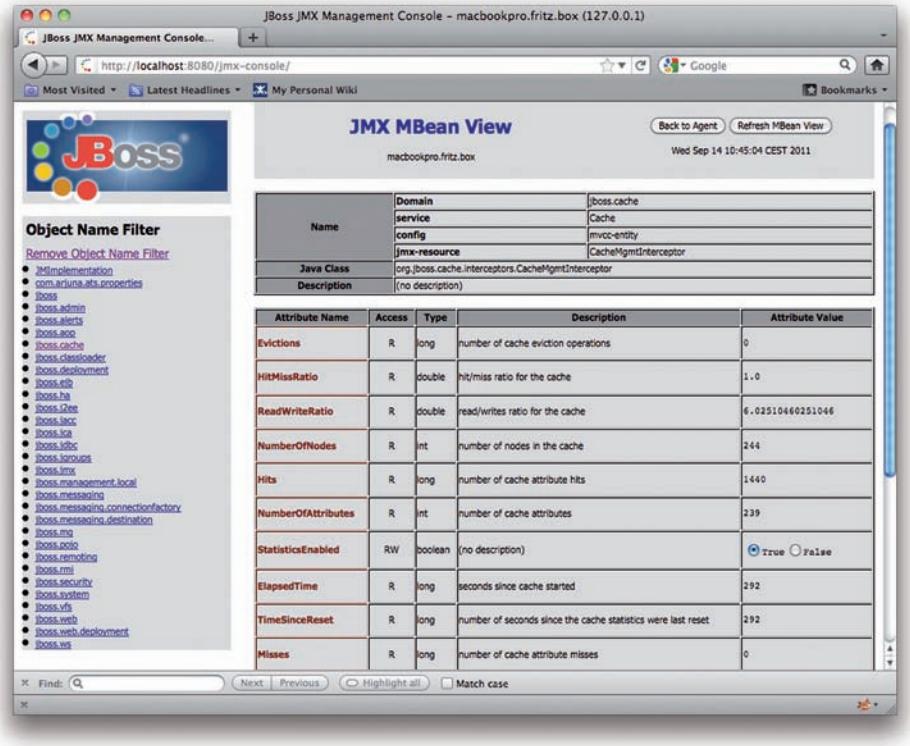

Abb. 1: Statistikzähler einer JBossCache-Entity-Cache-Instanz

JBoss-Applikationsservers ab Version 6.0 JBoss Cache verdrängt hat. Somit bieten sich zwei unterschiedliche Lösungen für die konkrete Realisierung eines verteilten Hibernate Second-Level Cache an: JBoss Cache und Infinispan.

JBoss Cache als Cacheprovider

Auch wenn der Name anderes vermuten lässt, kann JBoss Cache als Hibernate-Second-Level-Cache-Provider sehr gut außerhalb des Applikationsservers JBoss eingesetzt werden, z. B. in einer Standalone-Applikation oder innerhalb eines anderen JEE-Applikationsservers. Am einfachsten ist die Verwendung allerdings innerhalb des JBoss-Applikationsservers – sofern JBoss 4 oder JBoss 5 verwendet werden, denn ab JBoss 6 wurde JBoss Cache durch Infinispan ersetzt. Angenommen unsere Anwendung wird innerhalb von JBoss 5 deployt: In der *persistence.xml* unserer Beispielanwendung (Listing 1) aktivieren wir das Second-Level Caching von Hibernate inklusive dem Query-Cache. Wir verwenden dabei eine spezielle *RegionFactory*-Implementierung, die keine eigene JBoss-Cache-*CacheManager*-Instanz erzeugt, sondern auf eine bereits erzeugte *CacheManager*-Instanz zugreift. Diese wird im JNDI-Namensbaum unter dem Namen *java:CacheManager* beim Start der „all"-Konfiguration von JBoss 5 eingebunden und steht somit global für alle Anwendungen zur Verfügung. Konfiguriert wird der JNDI-Name über das Hibernate Property *hibernate.cache.region.jbc2.cachefactory*. Über *hibernate.cache.region.jbc2.cfg.entity*, *hibernate.cache.region.jbc2.cfg.query* und *hibernate.cache.region.jbc2.cfg.ts* werden separate und unterschiedlich konfigurierte JBoss-Cache-Instanzen für den Entity-Cache, den Query-Cache und den *UpdateTimestamps*-Cache (Kasten: „Hibernate Second-Level Caching in a Nutshell") definiert.

Die in Listing 1 referenzierten Cachekonfigurationen "*mvcc-entity*", "*local-query*", "*timestamps-cache*" sind in der Datei *cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml* unterhalb des *deploy*-Verzeichnisses der „all"-Konfiguration hinterlegt. Per Default, d. h. wenn nicht, wie in Listing 1, explizite Cachekonfigurationen definiert sind, werden der Entity-Cache und der Query-Cache in einer gemeinsamen Cacheinstanz, die über die Konfiguration mit dem Namen *optimistic-entity* definiert ist, zusammengefasst.

Die wesentlichen Merkmale dieser Konfiguration sind Synchronisierung der internen Zugriffe per Optimistischem Locking und Gewährleistung der Konsistenz im Cluster per Invalidierung. Ändert sich also der Inhalt des Caches auf einem Knoten, weil über diesen beispielsweise eine Entity verändert wird, werden die entsprechenden Caches auf den anderen Knoten im Cluster innerhalb derselben Transaktion invalidiert. Wird auf einem dieser Knoten später auf die geänderte Entity zugegriffen, wird sie neu von der Datenbank geladen und wieder im betreffenden lokalen Cache abgelegt. Auch wenn Invalidierung bereits besser skaliert als Replizierung, ist es in den meisten Fällen jedoch nicht notwendig, sowohl die Entity-Cache-Instanzen als auch die Query-Cache-Instanzen auf den einzelnen Knoten im Cluster per Invalidierung konsistent zu halten. Hibernate pflegt bei aktiviertem Query Caching auch immer den *UpdateTimestamps*-Cache, über den letztendlich die Konsistenz der Query-Caches gewährleistet wird – selbst wenn diese unabhängig voneinander verwaltet werden. Somit empfiehlt es sich i. d. R. den Query-Cache über die Konfiguration *local-query* zu starten.

Eine weitere Optimierung besteht darin, anstelle der Konfiguration *optimistic-entity* für die Entity-Cache-Instanz die Konfiguration *mvcc-entity* zu verwenden. Bei dieser werden interne Zugriffe auf Knoten innerhalb einer Cache-Instanz über das Multi-Version-Concurrency-Control-(MVCC-)Verfahren synchronisiert, das besser skaliert als Optimistisches Locking [7].

Die Konfiguration *timestamps-cache* entspricht im Übrigen der Default-Konfiguration für den *UpdateTimestamps*-Cache und wurde nur der Vollständigkeit halber mit aufgeführt. Die wesentlichen Merkmale dieser Konfiguration sind Pessimistisches Locking und asynchrone Replizierung.

Für den Fall, dass ein anderer Applikationsserver als JBoss zum Einsatz kommt, ist ein bisschen mehr zu tun. In diesem Fall stehen weder passende *RegionFactory*-Implementierungen im Klassenpfad des Applikationsservers zur Verfügung, noch eine JBoss-Cache-*CacheManager*-Instanz, die beim Start des Applikationsservers automatisch erzeugt wird. Das erste Problem lässt sich dadurch lösen, dass in dem Enterprise-Archiv der Anwendung das Modul *hibernate-jbosscache2.jar* aufgenommen wird. Für die Lösung des zweiten Problems ist man versucht, eine *RegionFactory*-Implementierung zu verwenden, die implizit eine *CacheManager*-Instanz zur Verfügung stellt. Anbieten würde sich hier beispielsweise *org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory*. Allerdings stehen in diesem Fall nur solche JBoss-Cache-Konfigurationen zur Verfügung, die in der Datei *jbc2-configs.xml* in *hibernate-jbosscache2.jar* hinterlegt sind. Leider enthält auch die zum Zeitpunkt der Erstellung dieses Artikels neuste Version (3.3.2.GA) von *hibernate-jbosscache2.jar* noch keine *mvcc-entity*-Konfiguration. Bleibt also nur die Möglichkeit, innerhalb der Anwendung selbst eine passende JBoss-Cache-*CacheManager*-Instanz zu erzeugen, sie mit einer um die fehlende *mvcc-entity*-Konfiguration ergänzten *jbc2-configs.xml*-Datei zu konfigurieren und anschließend in JNDI einzubinden. Dann kann Hibernate wieder über die bereits aus Listing 1 bekannte *RegionFactory*-Implementierung *org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory* per JNDI-Lookup auf genau diese *CacheManager*-Instanz zugreifen. Im Falle von Oracle WebLogic bietet sich hierfür eine *ApplicationLifecycleListener*-Implementierung an (Listing 2).

Hat man die Anwendung dann mit aktivem Hibernate Second-Level Cache deployt, möchte man verifizieren, ob der Cache auch korrekt konfiguriert

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
            http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
                                                    version="2.0">
  <persistence-unit name="caching-demo" transaction-type="JTA">
    <jta-data-source>java:/CachingDemoDS</jta-data-source>
      <class>de.mascher.caching.demo.domain.Country</class>
      <properties>
        <property name="jboss.entity.manager.jndi.name" value="java:/
                                          CachingDemoEntityManager"/>
        <property name="jboss.entity.manager.factory.jndi.name"
                      value="java:/CachingDemoEntityManagerFactory"/>
        <!-- Allgemeine Properties -->
        <property name="hibernate.dialect" value="org.hibernate.dialect.
                                                    MySQL5Dialect"/>
        <property name="hibernate.show_sql" value="true"/>
        <!-- Aktivierung 2nd-Level Caching -->
        <property name="hibernate.cache.use_second_level_cache"
                                                    value="true" />
        <property name="hibernate.cache.use_query_cache"
                                                    value="true" />
        <property name="hibernate.generate_statistics" value = "true" />
        <property name="hibernate.cache.use_structured_entries"
                                                    value="true" />
        <!-- Konfiguration JBoss Cache -->
        <property name="hibernate.cache.region.
                        factory_class" value="org.hibernate.cache.jbc2.
                          JndiMultiplexedJBossCacheRegionFactory" />
        <property name="hibernate.cache.region.jbc2.cachefactory"
                                          value="java:CacheManager" />
        <property name="hibernate.cache.region.jbc2.cfg.entity"
                                              value="mvcc-entity" />
        <property name="hibernate.cache.region.jbc2.cfg.query"
                                              value="local-query" />
        <property name="hibernate.cache.region.jbc2.cfg.ts"
                                          value="timestamps-cache" />
    </properties>
  </persistence-unit>
</persistence>
```

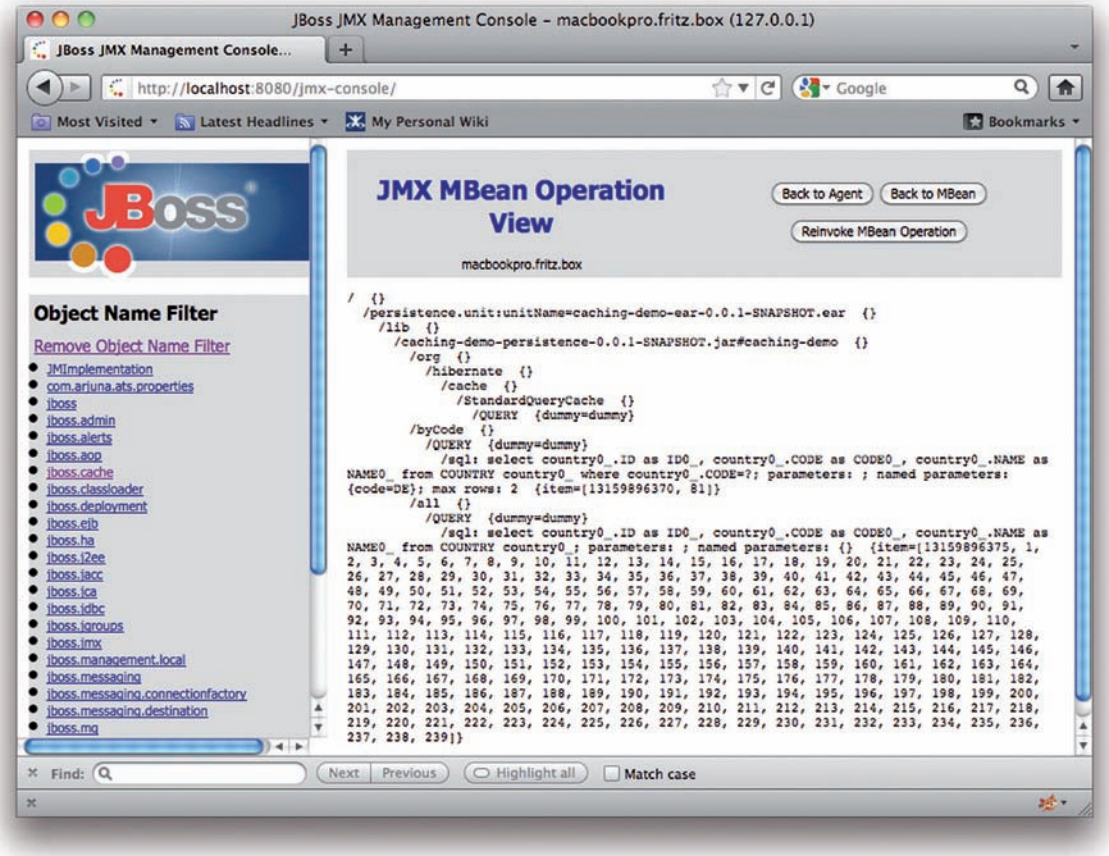

Abb. 2: Einblick in die interne Struktur einer JBossCache-Query-Cache-Instanz

ist bzw. greift. JBoss Cache registriert hierfür beim Erzeugen einer neuen Cacheinstanz automatisch eine Menge von MBeans, über die es möglich ist, per JMX von außen den Zustand der Instanz abzufragen. Die interessantesten MBeans hierbei sind die *CacheMgmtInterceptor* MBean und die *DataContainer* MBean. Die *CacheMgmtInterceptor* MBean stellt Statistiken in Form von MBean-Attributen zur Verfügung. Dafür muss das Hibernate Property *hibernate.generate_statistics* auf den Wert *true* gesetzt werden. Die *DataContainer* MBean erlaubt darüber hinaus einen direkten Einblick in die interne Struktur des Caches und die aktuell gecachten Daten. Werden, wie in Listing 1, unterschiedliche Regions für die einzelnen Caches konfiguriert und dadurch separate Cacheinstanzen erzeugt, stehen die erwähnten MBeans also getrennt für den Entity-Cache, den Query-Cache und auch den *UpdateTimestamps*-Cache zur Verfügung. Die JMX-Namen der MBeans reflektieren dabei u. a. die Namen der zugrunde liegenden JBoss-Cache-Konfigurationen, sodass die MBeans den entsprechenden Cacheinstanzen leicht zuzuordnen sind. **Abbildung 1** zeigt eine Momentaufnahme der Statistikattribute der *CacheMgmtInterceptor* MBean der Entity-Cache-Instanz für eine Beispielanwendung deployt in JBoss 5. Am interessantesten sind hier der *Hits*- und der *Misses*-Zähler sowie daraus abgeleitet der *HitMissRatio*-Wert. Je näher dieser bei 1.0 liegt (entspricht 100 %), desto besser greift der Cache.

Abbildung 2 zeigt die Ausgabe der Operation *printDetails* der *DataContainer* MBean der Query-Cache-Instanz für dieselbe Beispielanwendung. Hier ist schön zu sehen, dass die Query-Ergebnisse in dem Cache lediglich in Form von Primärschlüsseln hinterlegt werden. Außerdem sieht man, dass zu jedem Result Set eines Queries neben den eigentlichen IDs auch der Zeitstempel der tatsächlichen Ausführung des Queries als Long-Wert gespeichert wird. Mithilfe dieses Zeitstempels und des parallelen *UpdateTimestamps*-Cache kann Hibernate jederzeit auf jedem Knoten im Cluster feststellen, ob ein gecachtes Query-Ergebnis noch gültig ist, oder ob ein erneuter Roundtrip zur Datenbank notwendig ist.

Infinispan als Cacheprovider

Wenn die Anwendung, für die das Hibernate Second-Level Caching aktiviert werden soll, also in JBoss 6 deployt, bietet es sich an, als Cacheprovider Infinispan zu verwenden. Hierfür muss in der *persistence.xml* als *RegionFactory*-Implementierung die Klasse *org.hibernate.cache.infinispan.JndiInfinispanRegionFactory* konfiguriert werden (Listing 3). Als Parameter wird dabei der JNDI-Name der Infinispan-*CacheManager*-Instanz angegeben, die beim Start der JBoss-„all“-Konfiguration automatisch erzeugt und in den JNDI-Namensbaum unter dem Namen *java:CacheManager/entity* eingehängt wird.

Infinispan erzeugt per Default bereits getrennte Instanzen für den Entity-Cache und den Query-Cache.

Die einzelnen Entity-Cache-Instanzen werden dabei im Cluster per Invalidierung konsistent gehalten. Die Query-Cache-Instanzen werden dagegen unabhängig voneinander als jeweils lokale Caches gestartet, was aber letztendlich genau der Konfiguration entspricht, die in den meisten Fällen am besten skaliert, da die clusterweite Konsistenz der Query-Caches immer über den parallelen *UpdateTimestamps*-Cache gewährleistet ist. Infinispan implementiert als einzige Strategie für die Synchronisierung interner Zugriffe MVCC. Die Default-Konfiguration von Infinispan sollte also in den meisten Fällen bereits optimale Ergebnisse liefern.

Genaugenommen werden pro Knoten nicht nur separate Instanzen für Entity-Cache und Query-Cache gestartet, sondern sogar pro logischer Cache-Region, beispielsweise pro entsprechend annotierter Entity-Klasse oder auch pro Query, sofern für einzelne Queries per entsprechender JPA Query Hints separate Cache-Regions definiert sind. Dies hat einerseits den Vorteil, dass die Zugriffe auf Entities bzw. Query-Ergebnisse aus unterschiedlichen Regions besser skalieren. Andererseits lässt sich über die *Statistics* MBeans, die pro Cache-Instanz registriert werden, sehr schön überprüfen, wie gut das Cachen der jeweiligen Entities bzw. der jeweiligen Queries greift. Voraussetzung dafür ist jedoch, dass das Hibernate Property *hibernate.cache.infinispan.statistics*, wie in Listing 3, auf den Wert *true* gesetzt wird. **Abbildung 3** zeigt eine Momentaufnahme der Statistikattribute einer Entity-Cache-Instanz für eine Beispielanwendung, deployt in JBoss 6.

Hier fällt auf, dass die Java-Klasse, die die *Statistics* MBean von Infinispan implementiert, den einfachen Klassennamen *CacheMgmtInterceptor* trägt. Nicht nur an dieser Stelle wird deutlich, dass die Wurzeln von Infinispan in JBoss Cache liegen. Ein Äquivalent zu der *DataContainer* MBean von JBoss Cache, die es ja erlaubt den tatsächlichen Inhalt einer Cache-Instanz anzuschauen, sucht man in Infinispan allerdings vergebens.

Infinispan kann wie JBoss Cache auch außerhalb von JBoss in einer Standalone-Umgebung oder innerhalb eines anderen Applikationsservers als Hibernate-Second-Level-Cache-Provider verwendet werden, wobei sich die Verwendung hier im direkten Vergleich mit JBoss Cache

Listing 2

```
public class JBossCacheApplicationLifecycleListener extends
ApplicationLifecycleListener {

 private static final String JGROUPS_STACKS_CONFIG = "jgroups-stacks.
                                                          xml";
 private static final String JBOSS_CACHE_CONFIGS = "jbc2-configs.xml";
 private static final String CACHE_MANAGER_JNDI_NAME =
                                                  "java:CacheManager";

 private CacheManagerImpl cacheManager;

   @Override
   public void preStart(ApplicationLifecycleEvent event) throws
                                                  ApplicationException {
  try {
   startCacheManager();
   bindCacheManager();
  } catch (Exception e) {
   // ApplicationException werfen
  }
 }

 @Override
 public void postStop(ApplicationLifecycleEvent event) throws
                                                  ApplicationException {
  unbindCacheManager();
  stopCacheManager();
 }

 private void startCacheManager() throws Exception {
  ChannelFactory channelFactory = new JChannelFactory();
  channelFactory.setMultiplexerConfig(JGROUPS_STACKS_CONFIG);
     cacheManager = new CacheManagerImpl(JBOSS_CACHE_CONFIGS,
                                                  channelFactory);
     cacheManager.start();
 }

 private void bindCacheManager() throws NamingException {
  InitialContext context = new InitialContext();
  context.bind(CACHE_MANAGER_JNDI_NAME, cacheManager);
 }

 private void unbindCacheManager() {
  try {
   InitialContext context = new InitialContext();
   context.unbind(CACHE_MANAGER_JNDI_NAME);
  } catch (NamingException e) {
   // Log-Meldung ausgeben
  }
 }

 private void stopCacheManager() {
  try {
   cacheManager.stop();
   cacheManager = null;
  } catch (Exception e) {
   // Log-Meldung ausgeben
  }
 }
}
```

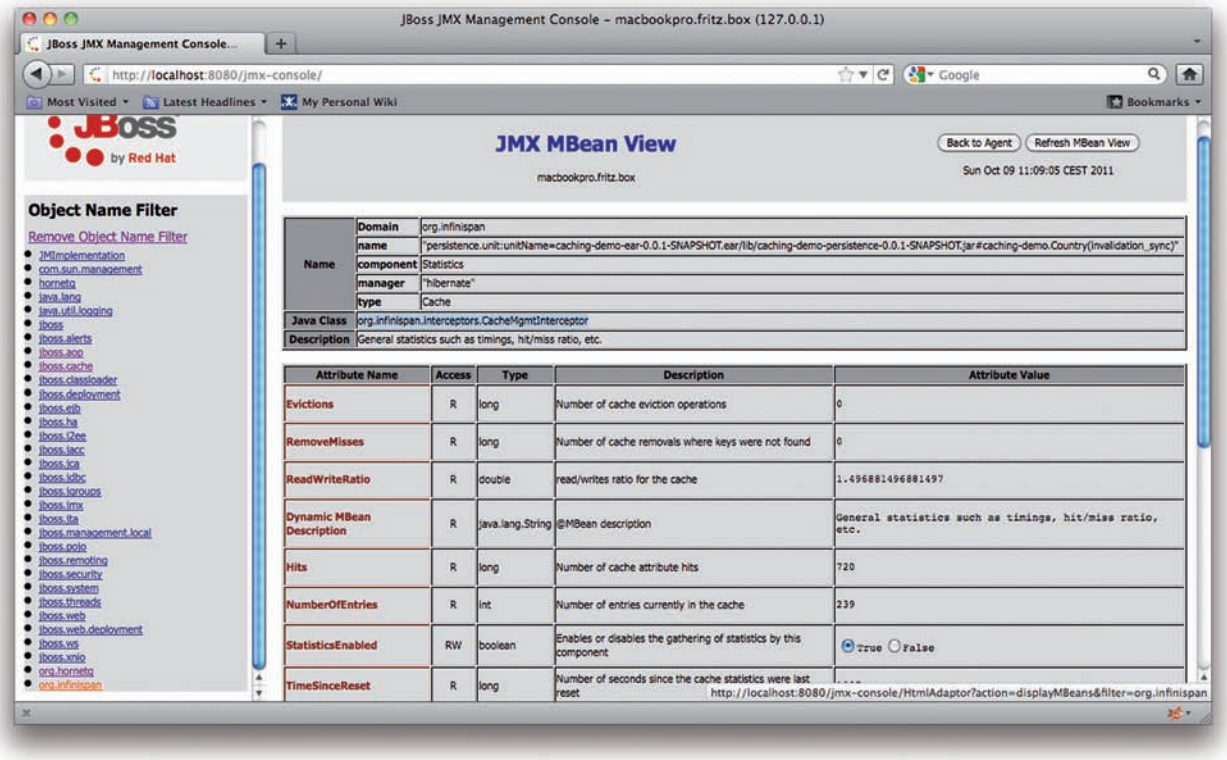

Abb. 3: Statistikzähler einer Infinispan-Entity-Cache-Instanz

wesentlich einfacher gestaltet. Die Anwendung muss neben Infinispan lediglich das Modul *hibernate-infinispan.jar* mitbringen, das die notwendigen *RegionFactory*-Implementierungen und entsprechende Cache-Konfigurationen enthält. Im Gegensatz zu JBoss Cache ist es bei Infinispan nicht notwendig, die *CacheManager*-Instanz über eine eigene Start-up-Klasse selber zu erzeugen bzw. zu konfigurieren und in den JNDI-Namensbaum einzuhängen. Stattdessen ist es ausreichend als *RegionFactory*-Implementierung die Klasse *org.hibernate.cache.infinispan.InfinispanRegionFactory* einzutragen, die automatisch eine passende *CacheManager*-Instanz erzeugt.

Fazit

Um für eine JEE-Anwendung, die im Cluster betrieben wird, das Second-Level Caching für Hibernate richtig zu konfigurieren, sind einige Hürden zu nehmen. Die angesprochenen Fehler in der Dokumentation von Hibernate und die Tatsache, dass Infinispan darin noch nicht einmal erwähnt wird, erschweren den Einstieg unnötig. Gerade wegen des zweiten Grunds wird man zuerst JBoss Cache in Betracht ziehen. Leider findet man jedoch zu dem speziellen Thema wenig detaillierte Quellen. Hier hilft der JBoss-Applikationsserver weiter, da er bis einschließlich Version 5 JBoss Cache integriert und geeignete Cache-Konfigurationen mitbringt. Die Verwendung außerhalb von JBoss gestaltet sich leider ein wenig kompliziert, was im Wesentlichen an den nicht optimalen Default-Konfigurationen liegt. Da kann Infinispan punkten, weil seine Default-Konfigurationen für die meisten Anwendungen bereits bestens geeignet sein sollten. Auch in puncto Dokumentation hat Infinispan die Nase vorn. Diese beinhaltet u. a. einen recht detaillierten Artikel, der konkret auf die Verwendung als Hibernate Second-Level Cache innerhalb und außerhalb des JBoss-Applikationsservers eingeht [8]. Allerdings ist Infinispan noch recht jung. Diese Tatsache wurde von den Entwicklern bzw. von Red Hat geschickt verschleiert, indem die Versionierung nicht bei 1.0, sondern bei 4.0 begonnen wurde. Auf der anderen Seite ist Infinispan ab JBoss 6.0 integraler Bestandteil des Applikationsservers und bildet die Basis für dessen Clusterimplementierung, sodass es ganz automatisch mehr und mehr in der Breite eingesetzt wird und dementsprechend schnell an Reife und Stabilität gewinnen wird.

Unabhängig davon, welcher konkrete Cacheprovider am Ende gewählt wird, sollte das Second-Level Caching in Hibernate allerdings immer nur dediziert für bestimmte Entities, Collections bzw. ausgewählte Queries aktiviert werden, wobei der Erfolg in jedem Fall durch Performancetests zu verifizieren ist.

Listing 3

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
            http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="caching-demo" transaction-type="JTA">
    <jta-data-source>java:/CachingDemoDS</jta-data-source>
      <class>de.mascher.caching.demo.domain.Country</class>
      <properties>
        <property name="jboss.entity.manager.jndi.name" value=
                                    "java:/CachingDemoEntityManager"/>
        <property name="jboss.entity.manager.factory.jndi.name"
                    value="java:/CachingDemoEntityManagerFactory"/>
        <!-- Allgemeine Properties -->
        <property name="hibernate.dialect" value="org.hibernate.dialect.
                                                MySQL5Dialect"/>
        <property name="hibernate.show_sql" value="true"/>
        <!-- Aktivierung 2nd-Level Caching -->
        <property name="hibernate.cache.use_second_level_cache"
                                                value="true" />
        <property name="hibernate.cache.use_query_cache" value=
                                                "true" />
        <property name="hibernate.generate_statistics" value = "true" />
        <property name="hibernate.cache.use_structured_entries"
                                                value="true" />
        <!-- Konfiguration Infinispan -->
        <property name="hibernate.cache.region.factory_class" value=
          "org.hibernate.cache.infinispan.JndiInfinispanRegionFactory" />
        <property name="hibernate.cache.infinispan.cachemanager"
                                    value="java:CacheManager/entity" />
        <property name="hibernate.cache.infinispan.statistics"
                                                value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Dirk Mascher arbeitet als unabhängiger Architekt und Entwickler im Java-Enterprise-Umfeld. Als Java-Entwickler der ersten Stunde konnte er in über zehn Jahren Projekterfahrung sein Wissen rund um alle Technologien im JEE-Umfeld ständig ausbauen. Parallel zu seiner Projekttätigkeit hat er mehrere Fachartikel veröffentlicht und eigene Seminare u. a. auch zu Hibernate entwickelt und gehalten.

Links & Literatur

[1] http://www.hibernate.org/

[2] http://www.hibernate.org/docs

[3] http://www.jboss.org/jbosscache

[4] http://community.jboss.org/wiki/JBossCacheOfficialDocumentation

[5] http://community.jboss.org/wiki/JBossCache

[6] http://www.jboss.org/infinispan

[7] http://community.jboss.org/wiki/JBossCacheMVCC

[8] https://docs.jboss.org/author/display/ISPN/Using+Infinispan+as+JPA-Hibernate+Second+Level+Cache+Provider

[9] http://www.jboss.org/infinispan/documentation