Methodology

Veel Java-applicaties, die een database als opslagmedium gebruiken, worden gebouwd op de Java Persistence API (JPA). Veel van deze applicaties bevatten zowel dynamische als statische data. Statische data worden ook wel referentiedata genoemd. Maar hoe verloopt de caching? We beperken ons in dit artikel tot het cachen van referentiedata.

Het cachen van referentiedata

Met gebruik van JPA via Hibernate en Ehcache

Anton Gerdessen
is consultant bij
Transfer Solutions.

Referentiedata zijn data die veel gelezen, maar zelden worden bewerkt. Dynamische data zijn data die veelvuldig gelezen en bewerkt worden.
Referentiedata bestaan in vele vormen. Bij een webwinkel kunnen dit bijvoorbeeld de te verkopen producten zijn. Bij een route-applicatie kunnen dit de wegenkaarten zijn. Een ander voorbeeld zijn de klantgegevens bij een callcenter.
Referentiedata hetzelfde behandelen als dynamische data betekent dat dezelfde referentiedata veelvuldig opgehaald worden uit de database, terwijl er niets is veranderd aan deze data. Referentiedata opnieuw ophalen is een nutteloze actie en dit kan de performance van een applicatie enorm drukken.

Referentiedata zijn een perfecte kandidaat om gecacht te worden. Door referentiedata te cachen, kunnen we de performance van de applicatie verbeteren zonder diep in te hoeven gaan op de traditionele cachingproblemen.
We willen inzichtelijk maken hoe we referentiedata kunnen cachen in JPA versie 1.0. Verder geven we aan de hand van voorbeeldcode een handvat om zelf te experimenteren met caching. We gebruiken Hibernate als de motor, de EntityManager onder JPA. Verder gebruiken we voor het cachen van de referentiedata Ehcache. We stellen Ehcache in als de second level cache voor Hibernate.

Referentiedata

We scherpen de definitie van referentiedata verder aan. We definiëren referentiedata als data, die voldoen aan de volgende criteria:

- Data die zelden wijzigen. Waarbij we zelden afhankelijk zijn van het type applicatie, voor batchapplicaties ligt dit anders dan voor een webwinkel..
- Data die door meerdere applicaties gedeeld kan worden.

We beperken ons tot het cachen van referentiedata, omdat de problemen die optreden een subset zijn van de problemen bij het cachen van dynamische data. De hele set problemen valt niet te behandelen in één artikel. De winst die te behalen is bij het cachen van dynamische data is variabel, terwijl de winst bij referentiedata constant en beter te meten is. Ook is de potentiële winst bij referentiedata groter dan bij dynamische data.

Referentiedata in de database

De vraag die kan worden gesteld is waarom referentiedata überhaupt in de database staan? Waarom laden we deze data niet bij het starten van de applicatie in het interne geheugen?
We nemen als voorbeeld de referentiedata in de vorm van producten voor een webshop. In theorie is het mogelijk de link te leggen van order naar product met producten in het interne geheugen en orders in de database. Maar op deze manier kunnen we de integriteit van de data in de database niet garanderen.
Verder moeten andere applicaties die dezelfde referentiedata gebruiken, bijvoorbeeld een applicatie die trends in verkopen analyseert, hetzelfde referentiemodel van producten in het interne geheugen opbouwen. Als er verschillen ontstaan in deze onderlinge modellen van producten, kan het zijn

dat een klant product x besteld, terwijl de rapportage product y aangeeft.
Samenvattend kunnen we zeggen dat referentiedata opgeslagen wordt in de database, omdat:

- we de integriteit van de data willen bewaken;
- het delen van data tussen applicaties wordt vergemakkelijkt.

Om deze redenen wordt referentiedata meestal opgeslagen in dezelfde database als de dynamische data.

Caching-problemen

Cachen wordt vaak genoemd als oplossing voor performanceproblemen. Programmeurs die in aanraking zijn gekomen met caching, beamen echter dat dit zeker niet altijd het geval is. Indien caching slecht ingezet wordt, kunnen er meer problemen ontstaan dan opgelost worden.
Deze caching-problemen treden op indien het consistent houden van de cache meer performance kost, dan de gecachte resultaten opleveren. Dit komt neer op hetzelfde principe als het aanmaken van teveel indexen op een databasetabel. Het bijwerken van de indexen kost meer tijd dan de besparing door de data op te halen via deze indexen oplevert.
Het belangrijkste caching-concept waar we over moet nadenken als er willen gaan cachen is:

- Eviction-strategy. Wat moet worden weggegooid als de cache vol is en wat niet? Voorbeelden zijn: minst gebruikt, als eerste opgeslagen.

Twee concepten die uit de eviction-strategy volgen zijn:

- Stale-data. De data in de cache is niet meer consistent met de data in de database. Dit kan omstaan doordat de data gewijzigd wordt, zonder dat de cache dit 'ziet'. Voorbeelden zijn: niet-exclusieve toegang tot de database, verschillende manieren van data manipulatie binnen dezelfde applicatie.
- Cache hits en misses. Cache hits zijn het aantal keer dat data uit de cache komt en niet uit de database. Cache misses zijn het omgekeerde. De verhouding tussen de cache hits en cache misses geeft aan hoe effectief de cache is. Een verkeerde Eviction-strategy, lijdt tot veel cache misses en dus een niet effectieve cache.

Als we deze concepten toepassen op referentiedata, dan kunnen we concluderen dan we een Eviction-strategy willen, waarbij nooit iets uit de cache wordt gegooid. De cache moet groot genoeg zijn om alle referentiedata te bevatten. Stale-data hebben we ook niet, omdat referentiedata niet wijzigen. Mocht de referentiedata toch wijzigen, dan gooien we de cache handmatig leeg. Ook gaan we uit van 100% cache hits op de referentiedata, nadat alle referentiedata eenmaal geraadpleegd zijn.

Java Persistence API en caching

JPA gebruikt maar één cache in versie 1.0. Dit is een transactie-scope cache. Dit wordt ook wel de level 1 cache genoemd. Alle data die JPA raadpleegt in dezelfde transactie wordt in deze cache bewaard. Oftewel, als ik object x opzoek en vervolgens object x nogmaals opzoek binnen dezelfde transactie, dan is het tweede resultaat geen databasehit, maar een gecachet resultaat. De level 1 cache heeft een Eviction-strategy, waarbij alles gecachet blijft, binnen deze transactie tenzij de programmeur expliciet aangeeft dat het niet moet gebeuren.
JPA versie 2.0 introduceert het concept van een second level cache. Een second level cache is een cache over individuele transacties heen, oftewel een shared-cache. Een transactie kan bijvoorbeeld een druk op en knop zijn met de verwerking achter deze knop. De volgende druk op een knop is een nieuwe transactie. Deze vorm van transactieverwerking is het typerende gedrag van een thin-client. Een applicatie die niet constant een databaseconnectie open heeft. Bijna alle Java-applicaties vertonen dit gedrag.

Het second level cache concept zit al langer in Hibernate. De second level cache is in te delen in de volgende onderdelen:

- Entiteiten-cache
- Collectie-/ relatiecache
- Query-cache

Met het woord entiteiten bedoelen we in de rest van dit artikel JPA-entiteiten. Indien we Hibernate als motor in JPA 1.0 gebruiken, kunnen we ook de second level cache van Hibernate gebruiken. Behalve Hibernate biedt ook EclipseLink een second level cache implementatie.

Entiteiten-cache: Als we spreken over entiteiten-cache, spreken we over het gedeelte van de cache voor de attributen van een entiteit, niet de relaties. Deze attributen komen overeen met alle velden van een rij in de database, die geen onderdeel zijn van een foreign key.
Collectie-/ relatiecache: In de literatuur kom je meestal de term collectie-cache tegen. Dit is wat verwarrend omdat de term collectie impliceert dat er een collectietype gebruikt gaat worden. Maar voor 1 op N of 1 op 1 relaties geldt dit niet, er wordt niet altijd een collectietype gebruikt. We kunnen collectie-/ relatiecache definiëren als het gedeelte van de cache voor relaties tussen verschillende objecten. Deze relaties worden meestal in de database gerepresenteerd als een foreign key.
Query-cache: Het laatste type is query-cache. Een query-cache cachet de resultaten van een query en de eventuele parameters met de timestamp van uitvoer. De parameters worden volledig opgeslagen als objecten en het resultaat wordt opgesla-

Als caching slecht wordt ingezet, dan ontstaan er juist meer problemen dan worden opgelost.

Query-caching is een vorm van caching die erg selectief moet worden toegepast.

gen als de key van de gevonden object(en). Deze resultaten zijn alleen geldig zolang de data van de onderliggende tabel niet wijzigt. Hiervoor dient de timestamp. Als er een update op een tabel plaatsvindt, wordt deze timestamp gebruikt om te controleren of de gecachte resultaten nog geldig zijn of moetweggegooid moeten worden. Hieruit kan worden afgeleid dat een query-cache op een tabel met veel updates niet efficiënt is.
Query-caching is een vorm van caching die erg selectief moet worden toegepast. De kans op een negatieve impact op de performance is groot.

Hibernate

We gebruiken Hibernate, de meest gebruikte JPA implementatie, als motor onder JPA 1.0. Omdat JPA versie 1.0 nog geen second level cache heeft, moeten we specifieke Hibernate-extensies gebruiken om de manier van cachen aan te geven. Dit doen we door Hibernate-specifieke annotaties te gebruiken. In het geval van entiteiten-cache kunnen we aangeven dat een entiteit gecachet moet worden via de volgende annotatie:

```
@Entity
@Cache(usage= =CacheConcurrencyStrategy.<STRATEGY>)
```

Hibernate biedt de volgende cache-strategieën aan:

- NONE. Deze entiteit moet niet gecachet worden.
- READ_ONLY. Deze entiteit moet gecachet worden. Deze entiteit wordt nooit gewijzigd via JPA.
- NONSTRICT_READ_WRITE. Deze entiteit moet gecachet worden. Deze entiteit wordt bijna nooit gewijzigd en concurrent updates zijn bijna uitgesloten.
- READ_WRITE. Deze entiteit moet gecachet worden en kan gewijzigd worden.
- TRANSACTIONAL. Deze entiteit moet gecachet worden. Deze vorm van cachen is de zwaarste vorm en is transactie-safe in tegenstelling tot alle andere vormen.

Collectie-/ relatiecachen kan via dezelfde annotatie op de collectie:

```
@OneToMany(mappedBy="exam", fetch = FetchType.LAZY)
@Cache(usage = CacheConcurrencyStrategy.<STRATEGY>)
private Set<Question> questions;
```

De eerder genoemde cache-strategieën gelden ook voor collectie-/ relatiecaching.
Query-cache werkt via het opgeven van een query-hint. In het geval van named query's kan dit als volgt:

```
@NamedQuery(name = "Exam.findExamsByName", query = ""
          + "SELECT    eam "
          + "FROM        Exam eam "
          + "WHERE    eam.name like :name "
          , hints = {
@QueryHint(name="org.hibernate.cacheable",value="true")})
```

In het geval van andere query's kan het als volgt:

```
Query query = entityManager.createQuery("SELECT eam
FROM Exam eam");
query.setHint("org.hibernate.cacheable", "true");
```

Ehcache

Ehcache is een open-source cache-oplossing. Ehcache kan worden ingezet om verschillende resources te cachen zoals:

- SOAP/RESTful web services;
- Servlets;
- Object/relational mappers, zoals Hibernate.

Alleen de laatste optie is voor ons van belang. Ehcache en Hibernate integreren goed. Dit vloeit voort uit het feit dat de oprichter van Hibernate, Gavin King, ook meewerkt aan Ehcache.
Ehcache is opgekocht door Terracotta. Terracotta stelt Ehcache beschikbaar in twee varianten:

- Enterprise, betaalde variant
- Open-source, gratis variant

De enterprise variant biedt bovenop de open-source variant commerciële support, de codebase is gelijk.

Omgeving en testopzet

Om het cachen van referentiedata te demonstreren hebben we een kleine applicatie gemaakt. De case van deze applicatie is het uitvoeren van examens voor verschillende leveranciers. Deze examens worden extern aangeleverd. Ieder examen heeft een set van mogelijke vragen waar er per uitvoer van een examen een aantal van moeten worden beantwoord.
In deze case kunnen we de examens, die extern aangeleverd worden, erkennen als referentiedata. We richten ons alleen op de examens en de onderliggende entiteiten.
Belangrijk is om enige vorm van betrouwbare resultaten te krijgen als de database op een dedicated machine moet staan.
Ook kan deze machine het beste niet dezelfde machine zijn als de machine die de testen draait. Als beide machines hetzelfde zijn, is er bijna geen netwerk-latency die er in een productieomgeving wel is. Eigenlijk komt het erop neer dat de omgeving zo dicht mogelijk bij de productieomgeving moet zijn, zoals bij iedere performance gerelateerde test.
De applicatie bestaat uit drie entiteiten:

- Exam;
- Question;
- Answer.

Hierbij bestaat Exam uit een naam en een set van Questions. Questions bestaan uit een inhoud (de vraagtekst) en een set van Answers. Answers bestaan uit een inhoud (het mogelijke antwoord) en een indicator of dit het correcte antwoord is.

Omdat we onder verschillende datasets de mogelijke voordelen van cachen willen testen, gebruiken we DBunit om snel van dataset te kunnen wisselen. We hebben de testopzet zo simpel mogelijk gehouden en gebruiken JPA in een Java SE omgeving. De code van deze applicatie is beschikbaar in een zip-file (zie referenties).
De zip-file bevat het volgende:

- Database directory. Hierin staat het Data Definition Language(DDL) script om de database aan te maken. Let op: dit script gaat uit van een Oracle database. Er worden sequences gebruikt;
- Lib directory. Hierin staan alle gebruikte bibliotheken, alleen in de variant met alle bibliotheken;
- Src directory. Hierin staat alle source-code.

De klasse TestCache bevat de functionaliteit om zelf te experimenteren met Ehcache.
De configuratie van Hibernate is te vinden in de src/main/resources directory.
De twee bestanden die de cache-configuratie bevatten zijn: hibernate.cfg.xml en ehcache.xml.
In de file hibernate.cfg.xml staat de volgende configuratie:

```
<property name="hibernate.cache.provider_class">
                              net.sf.ehcache.hibernate
.SingletonEhCacheProvider
</property>
<property name="net.sf.ehcache.
configurationResourceName">
         /META-INF/ehcache.xml
</property>
<property name="hibernate.cache.use_query_cache">
         true
</property>
<property name="hibernate.cache.use_second_level_cache">
         true
</property>
```

Deze configuratie geeft aan dat we Ehcache gebruiken, waar de configuratiefile voor Ehcache staat, dat query-cache aanstaat en dat we de second level cache gebruiken.
In de file ehcache.xml staat onder andere de volgende configuratie:

```
 <defaultCache   maxElementsInMemory="10000"
 eternal="true"     timeToIdleSeconds="120"
 timeToLiveSeconds="120"        overflowToDisk="true"
 diskPersistent="false"
         diskExpiryThreadIntervalSeconds="120"
         memoryStoreEvictionPolicy="LRU"/>
<cache   name="eh_cache_case.entities.Exam"
         maxElementsInMemory="0"eternal="true"
         overflowToDisk="false"/>
 <cache    name="eh_cache_case.entities.Exam.questions"
         maxElementsInMemory="0"eternal="true"
         overflowToDisk="false"/>
```

Deze configuratie geeft het default cache-gedrag aan. Verder geven we Exam een eigen cache-lokatie aan waarin alle Exam objecten terecht komen. Ook geven we voor de collectie van Exam naar Question aan dat deze een aparte cache-locatie heeft. We geven voor deze beide cache-locaties op dat de inhoud eternal is, oftewel nooit verloopt. Verder geven we aan dat het maximum van elementen 0 is, oftewel oneindig. We willen immers dat alle referentiedata gecachet beschikbaar is. Ook geven we op dat er geen overflow naar disk mogelijk is. We gebruiken alleen het interne geheugen.

Zelf de tests uitvoeren

Om zelf de tests uit te kunnen voeren zijn drie stappen nodig:

- Voer het DDL script uit op een Oracle-database om de tabellen aan te maken;
- Vervang de connectiegegevens in de Hibernate.cfg.xml file met die van je eigen database;
- Vervang de connectiegegevens in de constanten bovenaan de DbUnitHelper klasse met die van je eigen database.

Entiteiten cachen

Als voorbeeld gaan we 5000 Exams opslaan in de database. Vervolgens gaan we deze Exams ophalen via JPA. Dan sluiten we de sessie af, zodat de level 1 cache wordt geschoondt. Vervolgens halen we nogmaals de Exams op.
We kunnen dit demonstreren door op de TestCache klasse de methode testEntityCacheSimple uit te voeren. Als we in de Hibernate.cfg.xml, de second level cache uitschakelen kunnen we de verschillen zien.
De uitvoer zonder caching ziet er ongeveer zo uit:

```
First  pass took = 6 seconds, and  78 milliseconds.
statements prepared=5002
statements closed=5002
second level cache puts=0
second level cache hits=0
second level cache misses=0

Starting new session, level 1 cache cleared.

Second  pass took = 6 seconds, and  31 milliseconds.
statements prepared=10003
statements closed=1000
second level cache puts=0
second level cache hits=0
second level cache misses=0
```

Met caching zo:

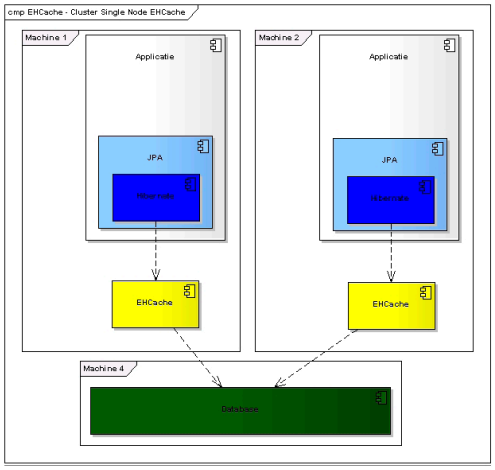
```
First  pass took = 7 seconds, and  109 milliseconds.
statements prepared=5002
statements closed=5002
second level cache puts=5001
second level cache hits=0
second level cache misses=5001

Starting new session, level 1 cache cleared.

Second  pass took = 0 seconds, and  688 milliseconds.
statements prepared=5002
statements closed=5002
second level cache puts=5001
second level cache hits=5001,
second level cache misses=5001
```

Te zien is dat er zonder caching toch verschil zit in de eerste en tweede uitvoer. Dit is te verklaren. De database cachet zelf ook. De database ziet dezelfde query's voorbijkomen en voert optimalisatie uit. De netwerkaanroep naar de database is echter nog steeds aanwezig.

Zonder caching zit er een groot verschil tussen de eerste en tweede uitvoer.



Cachen met single nodes van EHCache. Het cachen kan ook worden geclusterd met een versie van EHCache op een aparte machine.

Cachen kost tijd, maar bij referentiedata kunnen we ons die veroorloven.

Collecties/relaties cachen

Als voorbeeld gaan we 20 Exams inladen in de database. Onder deze examens zit een groot aantal Questions en Answers. Vervolgens gaan we alle Exams met onderliggende relaties naar Question en Anwser ophalen via JPA. Dan sluiten we de sessie af, zodat de level 1 cache wordt geschoond. Vervolgens herhalen we het ophalen van alle relaties.
We kunnen dit demonstreren door op de TestCache klasse de methode testEntityCollectionCache uit te voeren. Als we in de Hibernate.cfg.xml, de second level cache uitschakelen kunnen we de verschillen zien.
De uitvoer zonder caching ziet er ongeveer zo uit:

```
First  pass took = 0 seconds, and  546 milliseconds
statements prepared=279
statements closed=279
second level cache puts=0
second level cache hits=0
second level cache misses=0

Starting new session, level 1 cache cleared.

Second  pass took = 0 seconds, and  438 milliseconds.
statements prepared=558
statements closed=558
second level cache puts=0
second level cache hits=0
second level cache misses=0
```

Met caching zo:

```
First  pass took = 1 seconds, and  125 milliseconds.
statements prepared=279
statements closed=279
second level cache puts=1115
second level cache hits=0
second level cache misses=279

Starting new session, level 1 cache cleared.

Second  pass took = 0 seconds, and  203 milliseconds.
statements prepared=279
statements closed=279
second level cache puts=1115
second level cache hits=1115
second level cache misses=279
```

Wat te zien is, is dat de uitvoer met cachen de eerste keer meer tijd in beslag neemt dan de uitvoer zonder caching. Dit was ook zo bij het cachen van entiteiten, maar het is hier beter zichtbaar. Dit komt doordat cachen zelf ook tijd kost. Dit is een belangrijk punt. Bij referentiedata kunnen we ons deze vertraging veroorloven. De vertraging is immers eenmalig. We verwijderen nooit iets uit de cache. Bij dynamische data is dit echter anders en kan de waarde van caching alleen worden achterhaald door het te meten. Ook moet in deze opzet de cache zelf gestart worden, dit gebeurd pas na de eerste aanroep naar JPA.
Ook is te zien dat de tweede uitvoer met caching een tweevoud sneller is dat de uitvoer zonder caching. De winst de we behalen is ook permanent, iedere keer dat we de entiteiten of collecties gebruiken, hebben we weer deze winst.

Query's cachen

Als voorbeeld gaan we 20 Exams inladen in de database. We gaan de Exams opzoeken via de naam om zo de query-cache te demonstreren
Dan sluiten we de sessie af, zodat de level 1 cache geschoond wordt. Vervolgens zoeken we dezelfde 20 Exams op.
We kunnen dit demonstreren door op de TestCache klasse de methode testQueryCache uit te voeren. Als we in de Hibernate.cfg.xm, de query-cache EN de second level cache uitschakelen kunnen we de verschillen zien. Let op: voor de meeste realistische test moeten we ook de second level cache uitschakelen zodat het opslaan in de second level cache geen invloed heeft op de tijden.
De uitvoer zonder caching ziet er ongeveer zo uit:

```
First  pass took = 0 seconds, and  79 milliseconds.
querys executed to database=20
query cache puts=0
query cache hits=0
query cache misses=0
max query time=16

Starting new session, level 1 cache cleared.

Second  pass took = 0 seconds, and  47 milliseconds.
querys executed to database=40
query cache puts=0
query cache hits=0
query cache misses=0
max query time=16
```

Met caching zo:

```
First  pass took = 0 seconds, and  109 milliseconds
querys executed to database=20
query cache puts=20
query cache hits=0
query cache misses=20
max query time=16

Starting new session, level 1 cache cleared.

Second  pass took = 0 seconds, and  16 milliseconds
querys executed to database=20
query cache puts=20
query cache hits=20
query cache misses=20
max query time=16
```

De uitvoer met cachen neemt de eerste keer meer tijd in beslag dan zonder caching. Cachen kost ook tijd. Verder zien we dat zonder caching het aan-

tal statements dat wordt uitgevoerd steeds met 20 omhoog gaat. Bij caching is dit niet zo en is na de eerste uitvoer het aantal uitgevoerde statements constant; namelijk 20. We halen alle resultaten uit de query-cache.

Referentiedata cachen

Cachen blijft een ingewikkeld onderwerp. Voor het cachen van referentiedata kunnen we na deze tests enige handvatten geven:

- Gebruik entiteiten en collectie/relatie-cache voor referentiedata.
- Maak losse cache-gebieden per entiteit en per collectie/relatie.
- Zorg dat de cache-gebieden groot genoeg zijn om alle objecten te bevatten.
- Gebruik de query-cache alleen voor query's op referentiedata.

Het laatste punt vereist enig uitleg. Omdat we alle entiteiten en collecties cachen, is het niet nodig deze via query's op te halen. Haal alleen de 'rand' op en gebruik de al gecachete collecties. Als we bijvoorbeeld alle examens met vragen en antwoorden willen ophalen, halen we alleen de examens op met een query. Alle vragen en antwoorden onder de examens zitten al in de cache en halen we niet op via de query. Als we dit wel doen, kost de query meer tijd, maar krijgen we hier niets voor terug. In het geval van onze voorbeeldapplicatie halen we bijvoorbeeld een examen op via een query, terwijl we de onderliggende vragen en antwoorden via de collecties op het examen benaderen die al gecachet zijn en geen extra query's veroorzaken.
Cachen is eigenlijk alleen goed toe te passen door het te meten, maar deze vier punten gelden in ieder geval voor het cachen van referentiedata. Indien verder gekeken wordt naar caching, moeten we vooral op de query-cache letten. Deze kan indien verkeerd gebruikt nadelige performance opleveren. Na iedere update op een tabel wordt de gehele query-cache geïnvalideerd voor die tabel. Bij veel updates zijn er zo goed als geen cache-hits en kost het cachen alleen performance.

Mogelijke verbeteringen

In dit onderdeel benoemen we kort een aantal verbeterpunten en kritiekpunten.
In de voorbeelden gebruiken we READ_ONLY als cache-strategie. Dit legt beperkingen op het gebruik van de entiteit. Zo moeten de entiteit en alle collecties in 1 transactie aangemaakt worden en mogen relaties niet bidirectioneel zijn. Een veilige optie is het gebruik van NONSTRICT_READ_WRITE in plaats van READ_ONLY. Als de referentiedata via een andere applicatie worden ingeladen of als SQL statements ingeladen worden, speelt dit probleem niet. De referentiedata is dan voor JPA ook echt read only.
Ehcache gebruikt de @Id annotatie bij het bepalen van de identiteit van een element. Er wordt verder geen gebruik gemaakt van equals en hashCode voor caching. Dit neemt niet weg dat iedere entiteit equals en hashCode moet overschrijven om goed te werken binnen JPA.
Ehcache biedt buiten de statistieken van Hibernate, die gebruikt worden in de voorbeeldcode, nog twee manieren om statistieken op te vragen: Java Management Extensions (JMX) en Ehcache monitor.
Ehcache biedt eigen JMX beans, die handmatig geregistreerd moeten worden. De Ehcache-monitor werkt met zogenaamde probes. Deze probes worden meegeleverd met de applicatie. De monitor maakt een verbinding met deze probes en verzamelt zo statistieken.
Om deze statistieken uit te lezen moet wel statistics="true" worden toegevoegd aan de cache declaraties in de ehcache.xml file. De voorbeeldcode gebruikt een cast die niet altijd werkt in Java EE omgevingen, het is veiliger om het volgende te gebruiken:

```
org.hibernate.Session session = (org.hibernate.Session)
 entityManager.getDelegate();
```

Hibernate biedt twee extra annotaties die kunnen helpen bij caching. Omdat we echter JPA gebruiken, hebben we deze nog niet eerder benoemd. Deze twee annotaties zijn:

- @Immutable. Dit geeft aan dat een entiteit alleen gelezen wordt en nooit aangepast mag worden. Deze vult de READ_ONLY cache-strategie aan.
- @NaturalId. Dit geeft aan dat deze kolom(men) een natural key van deze entiteit zijn, wat betekent dat ze niet muteerbaar zijn. Hibernate kan op deze manier de query-cache optimaliseren.

Conclusie

We hebben in dit artikel inzichtelijk gemaakt hoe we referentiedata met JPA kunnen cachen. We hebben gekeken naar de problemen die ontstaan bij het gebruik van de second level cache. Verder hebben we aan de hand van voorbeeldcode een beginpunt gegeven om te experimenteren met JPA, Hibernate en Ehcache.
Ook hebben we een aantal vuistregels gegeven voor het cachen van referentiedata:

- Gebruik entiteit en collectie-/ relatie cache.
- Maak losse cache-gebieden per entiteit en collectie.
- Zorg dat deze cache-gebieden groot genoeg zijn om alle elementen te bevatten.
- Gebruik query-cache selectief.

In ieder geval blijft de regel bij cachen, zoals bij alle performance-gerelateerde onderwerpen binnen de IT, meten is weten. Hierbij kunnen de statistieken van Ehcache helpen. «

Refrenties

- Voorbeeldcode zonder bibliotheken http://gerdessen.com/anton/EHCacheCaseNoLibs.zip
- Voorbeeldcode met alle bibliotheken http://gerdessen.com/anton/EHCacheCaseFull.zip
- Ehcache - http://ehcache.org/
- Alex Miller - Hibernate query cache considered harmful? - http://tech.puredanger.com/2009/07/10/hibernate-query-cache/
- Darren L. Oldag - Hibernate Wars: The Query Cache Strikes Back - http://darren.oldag.net/2009/05/hibernate-wars-query-cache-strikes-back.html
- Darren L. Oldag - Hibernate Query Cache: A Dirty Little Secret - http://darren.oldag.net/2008/11/hibernate-query-cache-dirty-little_04.html
- R.J. Lorimer - Hibernate: Truly Understanding the Second-Level and Query Caches - http://www.javalobby.org/java/forums/t48846.html
- Stale data - Wikipedia - http://en.wikibooks.org/wiki/Java_Persistence/Caching#Stale_Data
- Cameron Purdy - Distributed caching lessons - http://www.infoq.com/presentations/distributed-caching-lessons
- Wikipedia - Reference data - http://en.wikipedia.org/wiki/Reference_Data
- Wikipedia - foreign key - http://en.wikipedia.org/wiki/Foreign_key
- http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/