Frameworks

**Het gebruik van een ORM-framework zoals Hibernate is de normaalste zaak van de wereld. Met de opkomst van JPA zijn ORM's als Hibernate de facto standaard geworden in enterprise Java-applicaties. Dit is niet zonder reden: Met Hibernate kun je complexe datamodellen productief ontsluiten. We moeten ons wel afvragen of het meest betreden pad ook altijd het beste is.**

# Performance & Tuning met Hibernate

## Overwegen van alternatieven kan nooit kwaad

Eerdere artikelen in Java Magazine hebben laten zien dat de kracht van de database vaak onderschat of onderbenut wordt, zeker wanneer abstracties zoals een ORM worden gebruikt. Wanneer de performance van de applicatie dan te wensen over laat, krijgt de ORM al snel de zwarte piet toegespeeld.
Gavin King, de maker van Hibernate, verwoordt het in een interview [1] als volgt: *"The problem is sort of cultural [..] developers use Hibernate because they are uncomfortable with SQL and with RDBMSes. You should be very comfortable with SQL and JDBC before you start using Hibernate - Hibernate builds on JDBC, it doesn't replace it. It is otherwise very easy to build an application that performs badly because you simply have no idea what is happening underneath. That is the cost of extra abstraction [..] Save yourself effort, pay attention to the database at all stages of development."*

In dit artikel bekijken we een viertal situaties waarbij Hibernate out-of-the-box zeker niet performt zoals je zou willen. Uiteindelijk is hier vaak wel een oplossing voor te vinden, maar regelmatig brengt dit onverwachte complexiteit met zich mee. We gaan vooral kijken naar de wat onbekendere uithoeken van het framework. Concepten zoals lazy loading en dirty-checking veronderstellen we als bekend. Mocht dit niet zo zijn dan is er altijd de referentie handleiding van Hibernate [2].

### Performance tuning

Voordat we de voorbeelden gaan bekijken is het goed om te beseffen dat het lang niet altijd nodig is om de laatste druppel performance uit Hibernate te knijpen. De code en configuratie worden er namelijk meestal niet eenvoudiger op. Wanneer het wel nodig is om te optimaliseren, zorg dan voor een goed proces:

- Gebruik bij alle wijzingen geautomatiseerde tests.
- Meet de gevolgen van iedere aanpassing (zowel tijdsduur als geheugengebruik).
- Test op een representatieve omgeving.
- Ken je database, of beter nog: betrek een DBA bij het tunen.

Op deze manier bewaak je zowel de correctheid van de aanpassing als het effect op de performance. Niet zelden lijkt een aanpassing een goed idee, maar valt de performance in de praktijk tegen. Of kan de wijziging in Hibernate-gerelateerde code beter vervangen worden door een goedgekozen index in de database.

Wat betreft het monitoren van Hibernate zijn er verschillende opties. Uiteraard kun je een Java-profiler gebruiken, maar daarmee is niet altijd goed te onderscheiden of Hibernate zorgt voor vertraging, of de database. Gelukkig is het mogelijk om met Hibernate alle gegeneerde SQL-statements te laten loggen. De volgende properties zorgen dat de queries op het scherm verschijnen:

```
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.use_sql_comments=true
```

Het bekijken van de gegenereerde SQL is een must tijdens het optimaliseren van Hibernate-code. Doet

**Sander Mak**
is Java-ontwikkelaar bij Info Support.


het wat je verwacht? Vaak moeten we deze vraag ontkennend beantwoorden, waarna de zoektocht kan beginnen. Verder biedt Hibernate zelf diverse statistieken aan, die je kunt uitlezen via een JMX-client zoals JConsole of VisualVM (meegeleverd bij de JDK). De volgende property zorgt ervoor dat deze statistieken worden gegenereerd:

```
hibernate.generate_statistics=true
```

Met deze maatregelen kom je als developer goed beslagen ten ijs wanneer je de performance van een Hibernate-applicatie gaat meten en verbeteren.

## Lazy batch fetching

Dan nu de eerste voorbeeldsituatie: stel we hebben Child entity met een Parent veld dat niet altijd opgehaald hoeft te worden. Geen probleem: daar maken we een lazy many-to-one associatie van (zie Codevoorbeeld 1). Maar met het lazy maken van een associatie of collectie ben je er nog niet. Nu wordt het interessant wat voor leespatronen de applicatie vertoont op deze velden. Stel er is een situatie waarbij we voor meerdere Child entities toch geïnteresseerd zijn in de Parent. We itereren bijvoorbeeld over een lijst van 100 Child entities en roepen child.getParent() aan binnen deze loop. Dat zal leiden tot 100 losse queries om de Parents op te halen; verre van ideaal. Dit is op verschillende manieren te ondervangen: al bij het ophalen van de 100 Child entities kun je met een fetch join in de query zorgen dat ook alle Parents worden opgehaald. Een minder bekende mogelijkheid is het gebruik van de @BatchSize annotatie. In Codevoorbeeld 1 zien we hoe deze annotatie op de Parent entity is gezet.

```
@Entity
@BatchSize(size=100)
public class Parent { .. }

@Entity
public class Child {
  @ManyToOne(fetch = FetchType.LAZY)
  Parent parent;
}
```

Codevoorbeeld 1.

Wanneer we nu in de loop voor de eerste keer c*hild.getParent()* aanroepen, zal Hibernate kijken of er nog meer Child entities in de persistence context zitten die een ongeïnitialiseerde parent hebben. Deze worden dan in dezelfde query opgehaald, tot een maximum van de gedefinieerde size. In dit geval zullen de 100 Parents dus in één query opgehaald worden en zullen de volgende iteraties in de loop geen query meer genereren omdat alle Parents al opgehaald zijn. Het terugbrengen van 100 queries (en dus database roundtrips) naar één levert een enorme snelheidswinst op. De @BatchSize annotatie kan ook op lazy collecties gebruikt worden.

## Batch wijzigingen

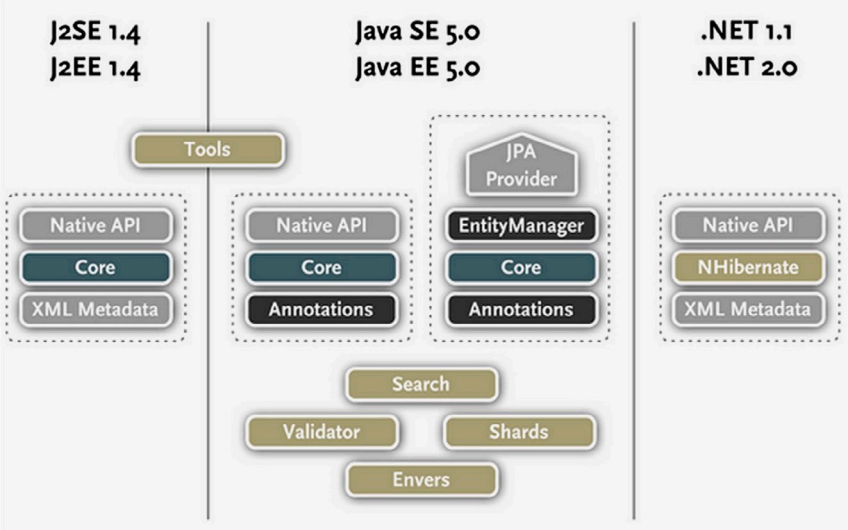
Niet alleen het ophalen van data kan sneller door te batchen: juist het wegschrijven of updaten van veel entities kan hier ook baat bij hebben. Het komt in de praktijk toch geregeld voor dat grote aantallen entities (honderden of meer) in een keer opgeslagen moeten worden. Bijvoorbeeld een Order entity waaraan honderden Item entities worden toegevoegd bij een grote bestelling. Hibernate biedt de mogelijkheid om de JDBC batch updates te gebruiken met de volgende property:

```
hibernate.jdbc.batch_size=50
```

Onder water zal Hibernate insert/update queries niet direct uitvoeren, maar ze opsparen tot de batch vol is. Dan worden ze in één keer over de lijn naar de database gestuurd. De winst die dit oplevert moet je wel afwegen tegen de extra tijd die je verliest wanneer het aantal queries onder de batch_size blijft (throughput versus latency). Die blijven dan onnodig lang in de batch wachten totdat Hibernate tijdens het committen kan concluderen dat de batch nooit vol zal komen en deze alsnog uitvoert. De waarde van batch_size is globaal, daarom wordt ook aangeraden geen extreme waardes te kiezen. Ergens tussen de 5 en 50 is vaak optimaal. In de praktijk blijkt dat het opslaan van een grote collectie tot 10x sneller kan gaan dan met individuele queries.

Helaas is het niet altijd zo simpel als het lijkt: wanneer bijvoorbeeld optimistic concurrency toegepast wordt op entities, zal JDBC batching automatisch worden uitgeschakeld. Een ander venijnig detail is dat JDBC batching alleen werkt wanneer de opeenvolging van queries die verstuurd wordt vanuit Hibernate gelijkvormig zijn. Dus bijvoorbeeld allemaal inserts op een bepaalde tabel, of allemaal

**Opslaan van een grote collectie kan tot tien keer sneller gaan dan met individuele queries.**




Voorheen faciliteerde Hibernate de opslag en consumptie van Java domein objecten via Object/Relational Mapping. Tegenwoordig bestaat Hibernate uit een collectie van gerelateerde projecten. Deze maken het de ontwikkelaar mogelijk om POJO-achtige domeinmodellen in hun applicaties te gebruiken op manieren, die ver voorbij Object/Relational Mapping liggen.

**Stateless sessions zijn paardenmiddelen in Hibernate.**

updates op een bepaalde tabel, waarbij de queries gelijk zijn modulo de parameters. Maar wanneer we een groot aantal entities inserten met een id die automatisch in de database wordt gegenereerd, zal Hibernate na iedere insert een select doen om de gegenereerde id op te halen. Dan wordt er dus geen batching toegepast, omdat insert en select queries alterneren. Zelfs wanneer de id's door de applicatie worden toegewezen, kunnen cascading saves op velden van de entities in de collectie voorkomen dat er een stroom van gelijkvormige queries gegenereerd wordt door Hibernate. In deze laatste situatie biedt Hibernate gelukkig wat properties om te sturen:

```
hibernate.order_inserts=true
hibernate.order_updates=true
```

Met deze properties probeert Hibernate de voorbijkomende update en insert queries te groeperen en sorteren, voordat ze aan de JDBC batch worden aangeboden. Hierdoor is de kans veel groter dat er daadwerkelijk gebatched kan worden. Het probleem met de gegenereerde id's kan hier overigens niet mee worden opgelost, aangezien Hibernate echt de id nodig heeft om op de entity in de persistence context te zetten.

## Stateless session

De persistence context (of session) van Hibernate functioneert als een first-level cache: alle entities blijven 'attached' voor de duur van de session. In de situatie waarbij entities vaker worden opgevraagd tijdens de session is dit ideaal, er is dan geen nieuwe database query nodig. Maar wat als we alleen geïnteresseerd zijn in het wegschrijven van veel data? Denk bijvoorbeeld aan een webservice call in een SOA-applicatie om een grote order op te slaan. We weten dan zeker dat in dezelfde call deze data niet meer wordt opgevraagd. In dit soort gevallen kun je gebruik maken van een StatelessSession. De StatelessSession houdt entities niet vast na het opslaan of updaten, waardoor het geheugengebruik constant is in plaats van lineair groeiend bij grote hoeveelheden data. Helaas mist de StatelessSession veel van de automatische management functies voor entities:

- Collecties worden genegeerd, entities in collectie velden moet je dus zelf expliciet persisteren.
- Cascade acties worden niet uitgevoerd.
- Dirty-checking wordt niet gedaan op bestaande entities.
- Het Hibernate event-model en interceptors worden genegeerd.

Het is dus nogal een paardenmiddel, maar uiteraard leveren deze beperkingen performance voordelen op. In een SOA-applicatie waar deze methode is toegepast leidde het tot een drie keer snellere opslag, met constant geheugengebruik
In Codevoorbeeld 2 is te zien hoe we een Order met meerdere Items kunnen opslaan met de StatelessSession. Eerst moeten alle Items expliciet worden opgeslagen (ongeacht de cascade instelling op die collectie) en daarna slaan we de Order zelf op. Het feit dat de methode insert heet op een StatelessSession, tegenover saveOrUpdate op een normale session, geeft al aan dat het hier om een heel dun schilletje bovenop JDBC gaat.

```
public void saveOrder(Order order) {
  StatelessSession stateless =
    session.getSessionFactory
     .getStatelessSession();

    for(Item item: order.getItems()) {
       stateless.insert(item);
    }
    stateless.insert(order);
}
```

Codevoorbeeld 2.

Let wel op wanneer je StatelessSession gebruikt in combinatie met het eerder beschreven JDBC batching. Dit kan leiden tot dataverlies! Gelukkig hebben we geautomatiseerde tests die dit meteen detecteren. Het is namelijk zo dat Hibernate de laatste batch, als die niet volledig tot de maximale jdbc_batchsize gevuld is, vergeet te flushen naar de database. Voor een workaround zie Hibernate bug HHH-4042 [3]. Helaas hebben de Hibernate developers deze bug als rejected gemarkeerd, omdat dit

'by design' zou zijn. Zo zie je maar, hoe verder je een abstractielaag als Hibernate probeert te masseren, hoe lastiger de problemen worden. Uiteraard kun je in dit geval ook terugvallen op puur JDBC, maar dit vraagt veel meer boilerplate code en brengt een tweedeling in de data-access architectuur van de applicatie. Dit kan weer gevolgen hebben voor transactionaliteit et cetera.

## Loop en vul collectie

Een laatste voorbeeld. Hibernate garandeert dat queries altijd tegen de meest recente data worden uitgevoerd. Dat klinkt mooi, maar het mechanisme om dit te bereiken kan ook onverwachte effecten hebben. Neem bijvoorbeeld Codevoorbeeld 3. Daarin zien we de (pseudo-)implementatie van een service call die Items aan een Order toevoegt. Order heeft een collectie van items, oftewel Order heeft een 1-op-n relatie met Item. Een binnenkomend Item wordt gevalideerd, waarbij lookup queries worden uitgevoerd, bijvoorbeeld om een prijs of autorisatie te verifiëren in andere tabellen met referentiedata. Vervolgens wordt het item aan de items collectie binnen Order toegevoegd.

```
public void addToOrder(List<Item> items) {
   Order order = getOrder();
   for(Item item: items) {
      // Performs lookup queries!
      validate(item);
      order.getItems().add(item);
   }
   session.saveOrUpdate(order);
}
```

Codevoorbeeld 3.

Normaal gesproken zou je verwachten dat het opslaan van de order buiten de loop leidt tot een stroom van inserts in de Item tabel wanneer *saveOrUpdate(order)* wordt aangeroepen. Dit kan door het eerder besproken JDBC batch mechanisme worden geoptimaliseerd. Helaas gebeurt er iets anders: er worden afwisselend selects en inserts uitgevoerd gedurende de loop, waarbij niet gebatched kan worden:

```
select price from Product where id = ?
select authorized from Authorizations where id = ?
insert into Item values (?, ?, ?)
select price from Product where id = ?
select authorized from Authorizations where id =
insert into Item values (?, ?, ?)
```

.. etc

De oorzaak is dat de flushmode default op AUTO staat. In dit geval besluit Hibernate bij de tweede iteratie, wanneer er weer een lookup query moet worden gedaan, dat er dirty data in de Hibernate sessie zit: het toegevoegde element uit de vorige iteratie. Hibernate detecteert dit doordat de collectie eigenlijk een proxy object is wat aan het framework signaleert dat er iets gewijzigd is. Deze data moet eerst weggeschreven worden om te garanderen dat de lookup queries de meest recente data zien. Helaas is Hibernate niet intelligent genoeg om te zien dat het om totaal verschillende tabellen gaat. Hoe kunnen we voorkomen dat dit gebeurt? In ieder geval door de flushmode op COMMIT of MANUAL te zetten, maar dit kan gevolgen hebben voor de rest van de applicatie, want soms is de dirty data inderdaad van invloed op de query. Codevoorbeeld 4 laat een workaround zien, die er voor zorgt dat de inserts wel gebatched kunnen worden zonder de flushmode te veranderen.

```
public void addToOrder(List<Item> items) {
   Order order = getOrder();
   for(Item item: items) {
      // Performs lookup queries!
      validate(item);
   }
   order.getItems().addAll(items);
   session.saveOrUpdate(order);
}
```

Codevoorbeeld 4.

Door de elementen pas buiten de loop toe te voegen aan de collectie, zal de collectie-proxy niet binnen de loop detecteren dat er veranderingen zijn. Hierdoor zullen eerst alle selects worden uitgevoerd en daarna alle inserts achter elkaar, die dan gebatched kunnen worden. Het omzetten van de onschuldig ogende add() binnen de loop naar een addAll() buiten de loop levert daardoor een enorme snelheidswinst op.

## To Hibernate or not to Hibernate

Alle voorbeeldcode in dit artikel is geïnspireerd op situaties uit productieapplicaties, maar het gaat niet zozeer om de voorbeeldcode. Er zijn nog talloze andere situaties waarin Hibernate niet direct performt zoals je zou willen, of juist wel. Het gaat meer om het besef dat kleine wijzigingen, zowel in code als in configuratie, een groot verschil kunnen maken in de performance van Hibernate. Maar het gaat ook zeer zeker om het besef dat eenvoudig ogende optimalisaties veel complexiteit met zich mee kunnen brengen.

Het is duidelijk dat de kosten van abstractie, zoals aangestipt door Gavin King in het begin van dit artikel, duidelijk naar voren komen wanneer je dieper in Hibernate duikt dan alleen op tutorialniveau. Het tunen van Hibernate is niet voor beginners.

Dat Hibernate en ORM's een standaardkeuze zijn is niet zonder reden. Maar wanneer een applicatie veel data van en naar de database stuurt, en hoge performance erg belangrijk is, dan kan het zeker geen kwaad om alternatieven te overwegen die dichter tegen JDBC en de database aan liggen. «

**Het tunen van Hibernate is niet voor beginners!**

**Referenties**


1. http://www.javaperformancetuning.com/news/interview041.shtml
2. http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/
3. http://opensource.atlassian.com/projects/hibernate/browse/HHH-4042