**Introducere**

Cu destul timp în urmă, pe când eram oarecum încă la început ca programator Java, unul din colegii mei avea ambiția de a face doar OOP, fără a coda deloc SQL. La acea vreme utilizam Hibernate ORM 3 (care abia apăruse); JPA încă nu exista; a face OOP cu Hibernate la acel moment însemna utilizarea de Hibernate Criteria pentru a genera query-uri pentru baza de date-Oracle Database Server pe acel proiect. Colegul meu refuza în mod sistematic orice însemna SQL, mergând pe ideea că Hibernate ORM era îndeajuns de inteligent încât să optimizeze singur orice fel de query până când ... a generat Hibernate o comandă SQL de o dimensiune atât de mare încât a refuzat Oracle Database Server să o execute (era un SELECT ... WHERE ID IN (....)-unde lista de ID-uri era enormă).

În cei 18 ani ai mei ca programator, am lucrat la numeroase proiecte, în diferite etape. Am făcut și am văzut numeroase greșeli. Am văzut mulți developers inițial extrem de entuziasmați de JPA (ORM-uri în general) pentru ca, ulterior, să devină extrem de sceptici (“Hibernate este extrem de incet” sau “JPA se mișcă foarte greu”). Scopul acestui articol este de a descrie cea mai des întâlnită problemă specifică JPA, luând JPA 2.1 ca punct de referință și particularizând pe diverse implementări JPA (Hibernate și EclipseLink).

Exemplele utilizate sunt accesibile via Git la:  
  
<https://github.com/catam1976/JPASamples/tree/master/jpa-issues>  
(detalii de build sunt oferite pe readme-ul de pe GitHub).  
  
**Lipsa de logging a comenzilor SQL generate de ORM**

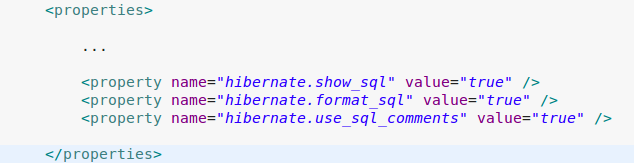
Oricât ar pare de ciudat, majoritatea programatorilor Java ignoră complet monitorizarea comenzilor SQL generate de JPA. Acum aproape 2 ani am făcut review la o aplicație web codată în Java 7 cu SpringFramework, JSF 2 și JPA2 (cu Hibernate ca JPA provider), Maven 3, rulând pe Tomcat 7. Aplicația făcea managementul membrilor unei organizații. Pe lângă access complet la surse, am primit și un hint de genul următor: “In development, aplicația noastră se mișcă foarte bine; în producție, cu baza de date reală - aproape 1000 de membri ai organizației noastre - aplicația se mișcă extrem de încet”. Un scenariu clasic, de fapt, pe care l-am reîntâlnit de multe ori. După ce am reușit să îmi configurez local aplicația de analizat, primul lucru care am vrut să-l verific a fost să văd câte comenzi SQL sunt executate pentru fiecare ecran (funcționalitate) în parte. Spre supriza mea, pentru 10 linii de date afișate în browser se generau și se executau enorm de multe comenzi SQL: în jur de 3000 (da, trei mii!) de comenzi SQL! Devenise clar motivul principal pentru care acea aplicație nu performa: o greșită utilizare a ORM-ului (JPA-Hibernate în acel caz). De ce nu sesizaseră și programatorii care codaseră acea aplicație? Lipsă de experiență? Posibil, dar siguranță pentru ca nu utilizaseră un feature extrem de simplu oferit de JPA providers: logarea de comenzi SQL generate - un feature ignorat de majoritatea programatorilor, care, combinat cu o bază de date de development populată cu date extrem de puține, a dus la performanțe extrem de scăzute.

**De ce este nevoie de acest logging ?**

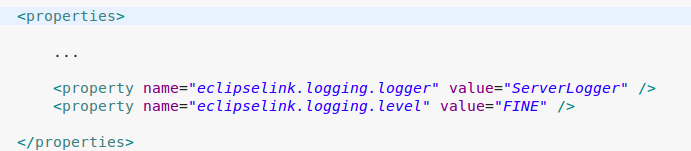
Răspunsul este extrem de simplu: logarea comenzilor SQL generate de JPA reprezintă cea mai simplă formă de monitorizare / diagnosticare a unei aplicații Java care folosește JPA pentru access baze de date relaționale: este de bun simț ca 10 linii de date pe ecran să nu ducă la execuția a mii de comenzi SQL; 2-3 comenzi SQL ar trebui să fie îndeajuns, în general-pentru acele 10 linii de date afișate pe browser.

**Cum se face SQL logging pentru JPA providers ?**

Deși JPA 2.1 a venit cu multe elemente noi, nu există nimic standard pe partea de logare comenzi SQL. Pentru Hibernate, în persistence.xml trebuie să apară:

  
  
Evident, valorile pentru acele proprietăți specifice Hibernare ar putea fi filtrate cu Maven-printr-un profil de development, unde se loghează toate comenzile SQL și un profil de production, unde nu se loghează acele comenzi-dar aceasta e în afară subiectului curent.

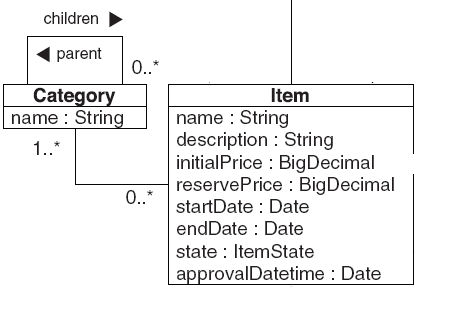
Pentru EclipseLink, setările echivalente sunt:

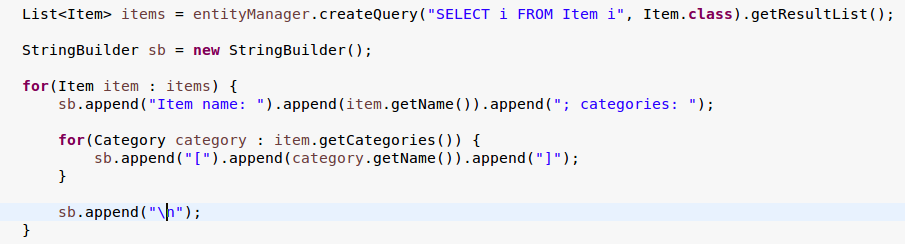


În exemplul de pe GitHub, se pot găsi două fișiere persistence.xml-unul pentru Hibernate (/tools/orm/hibernate/persistence.xml) și altul pentru EclipseLink (/tools/orm/hibernate/eclipseLink/persistence.xml)

Acum, că avem o modalitate de monitorizare a comenzilor SQL, putem trece la cea mai des întâlnită problemă de performantă specifică JPA: problemă N + 1.   
  
**Problema N + 1 - prima versiune**

Acum 5 ani, pe când făceam un review la o aplicație la o firmă din Brașov, am găsit 4-5 linii de cod Java care generau sute de comenzi SQL. Pare ciudat, nu ? Mai jos este o diagrama UML care prezintă un model de date pe care voi demonstra această problemă N + 1.



În diagrama de mai sus, cele două entități (Item și Category) sunt în relație many-to-many. Pentru fiecare item, dorim să listăm numele item-ului împreună cu numele categoriilor la care este atașat. Mai jos este un cod Java oarecum similar celui care l-am văzut generând sute de comenzi SQL în 3-4 linii de cod:   
  


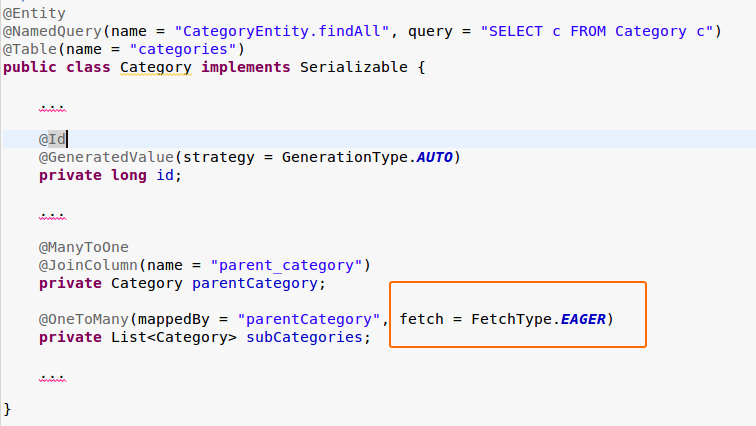
Pentru a se execută acest cod, trebuie executată metodă Boostrap.main parametrizată cu doNPlus1FirstCase ca prim parametru. Boostrap este the main entry point în exemplul de pe GitHub.

După cum se vede în log, comenzile SQL generate sunt foarte multe. Una pentru a obține itemii (1) și restul pentru a obține categorii (N)-de aici și numele N+1 : 1 comandă SQL executată inițial duce la măcar alte N comenzi SQL (N fiind numărul de inregistrări returnate de prima comandă). Cu alte cuvinte, cu cât mai multe date în bază de date, cu atât mai multe comenzi SQL generate și executate. De exemplu, dacă ar fi sute de mii de items puși pe mii de categorii, performantă acelui cod de mai sus este extrem de scăzută.

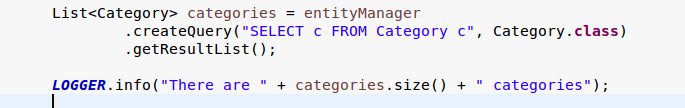
Common sense spune că un singur SQL ar trebui să fie îndeajuns (un join între cele 2 tabele). Ideal, folosind JPA, nu SQL direct (deși query-uri native sunt uneori soluții mai bune decât JPA-QL; în plus, aceste query-uri native sunt ușor de integrat în JPA-cu @NativeQuery și @SqlResultSetMapping). Mai jos am descris modalitătile mele de detecție plus soluțiile de fixare-pentru problemă N + 1.

**Problema N + 1 - a doua versiune**

O altă versiune al aceluiași gen de problemă vine nu din loop-uri Java de tipul celor de mă sus, ci dintr-o greșită modelare a relației părinte-copil dintre o categorie și lista ei de subcategorii (pe exemplul de pe GÎT, se poate vedea în Category.java), fiind cu încărcare EAGER a unei relații OneToMany



Dorința programatorului a fost ca, incarcand categoriile (sau doar una din ele), să încarce dintr-un foc și categoriile copil. Doar că .. a generat un anume tip de problemă N + 1, una dintr-o singură linie de cod Java, generând potențial atâtea comenzi SQL câte categorii sunt definite în bază de date. Problemă vine codul marcat cu roșu mai sus.

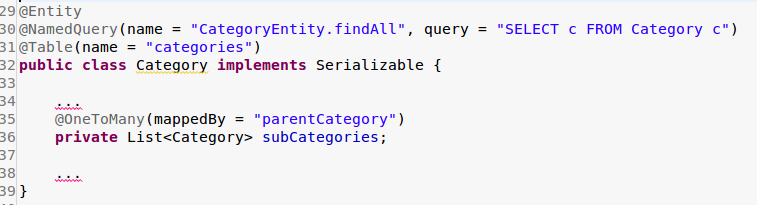
  
  
Pentru a se execută acest cod, trebuie executată metodă Boostrap.main parametrizată cu doNPlus1SecondCase ca prim parametru.

După cum se vede în logs, s-a generat o comandă SQL pentru a obține toate categorille; apoi, pentru fiecare categorie din primul rezultat, se execută câte un SELECT-SQL pentru a obține lista de copii. Common sense spune care dacă s-ar fi mers pe relația inversă (pentru o categorie, să se obțină părintele direct), un singur SELECT-SQL ar fi fost îndeajuns.

**Cum se detectează problema N + 1**   
Al doilea caz e ușor de detectat: un eager fetch pe relațiile OneToMany sau ManyToMany. În general, prefer mereu LAZY loading pe relațiile ManyToOne, OneToMany (by default, EAGER), OneToOne, ManyToMany (by default, LAZY).

Primul caz e puțin mai dificil de detectat doar din cod (de obicei, apare la loop-uri), dar logarea comenzilor SQL plus execuția pas cu pas al codului poate duce ușor la identficarea loop-ului care generează multitudinea de comenzi SQL.   
  
**Cum se rezolvă problema N + 1**

În cazul celei de-a doua probleme, rezolvarea constă în eliminarea încărcării EAGER pe subCategories; și de codat în Java gruparea categoriilor copil-părinte (eventual, cu Java Stream API din Java 8). O data eliminat acel fetch de tip EAGER, numărul de SELECT-SQL ar trebui să se reduce la una singură.

Pentru prima problemă: mai întâi trebuie eliminat acel EAGER menționat mai sus (a se modifică manual în codul sursă în entitatea Category:   
   


Apoi, se poate utiliza un join fetch (desigur, items vor fi “dublați” deci duplicatele trebui eliminate):



Pentru a se execută acest cod, trebuie executată metodă Boostrap.main parametrizată cu doNPlus1FirstCaseSolution va prim parametru.

Utilizând Hibernate sau EclipseLink că JPA provider, codul de mai sus generează doar o singură comandă SQL. Indiferent câți Items sau Categories sunt definite în bază de date.

O altă soluție (specifică JPA 2.1) constă în utilizarea de entity graphs cu nouă adnotare @NamedEntityGraph (practic, poate defini un graf între entități JPA permițând EAGER fetch).  
  
**Încheiere**ORM-urile sunt tool-uri extrem de puternice - cu condiția să fie utilizate corect, monitorizând mereu comenzile SQL generate și executate. Fiecare provider de JPA are mecanisme specifice de generare a comenzilor SQL.

Indiferent de tipul de relație între două entităti, mereu am preferat să le modelez că fiind cu incărcare LAZY, introducând o incărcare EAGER fie din JPA-QL, fie din JPA Criteria.