Optimisations

Monitoring

Traitements batch

Cache de second niveau

Intercepteurs et évènements

Recommandations

Traces

Hibernate peut tracer :
- L'instruction SQL exécutée
- Quelles valeurs de paramètre de liaison il a utilisées,
- Combien d'enregistrements la requête a renvoyée
- Combien de temps a duré chaque exécution

Activer :
org.hibernate.SQL à DEBUG.
Mais il est souvent préférable d'utiliser des outils comme *datasource-proxy* ou *p6spy*

Statistiques

Par défaut, les statistiques ne sont pas collectées car cela ajoute un traitement et une surcharge mémoire.

Pour activer les statistiques:

hibernate.generate_statistics = true

Pour afficher les statistiques sur les journaux

- *org.hibernate.stat = DEBUG*
- Ou via l'API dans un intercepteur par exemple

Exemple trace

14:37:30,715 INFO StatisticalLoggingSessionEventListener:258 – Session Metrics {
48986 nanoseconds spent acquiring 1 JDBC connections;
23326 nanoseconds spent releasing 1 JDBC connections;
259859 nanoseconds spent preparing 1 JDBC statements;
1092619 nanoseconds spent executing 1 JDBC statements;
0 nanoseconds spent executing 0 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
22383767 nanoseconds spent executing 1 flushes (flushing a total of 1 entities and 1 collections);
72779 nanoseconds spent executing 1 partial-flushes (flushing a total of 0 entities and 0 collections)
}

Métriques

Tous les métriques disponibles sont décrits dans l'API ***Statistics***

Elle fournit :

- Statistiques agrégées sur les requêtes, les entités, les collections
- Statistiques SessionFactory: Nombre d'entités et de collections
- Statistiques de session: nombre d'ouverture, de fermeture, de rinçage
- Statistiques JDBC: Nombre de Prepared Statement
- Statistiques de transaction: nombre de transactions (réussies)
- Statistiques des identifiants naturels
- Cache de 2e niveau (Hit et Miss)

Optimisations

Monitoring
Traitements batch
Cache de second niveau
Intercepteurs et évènements
Recommandations

Batch JDBC

JDBC prend en charge le traitement par lots:

- Les drivers envoient les opérations de mise à jour par lots ; ce qui économise les appels réseau vers la base de données.

La propriété ***hibernate.jdbc.batch_size*** fixe le maximum d'opérations dans un lot

Traitements par paquets

Lorsque l'on désire exécuter de nombreuses mises à jour dans une même méthode, nous sommes confrontés à des problèmes de taille mémoire.

Le code suivant risque de provoquer un *OutOfMemoryException :*

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

Solution

Pour éviter les OutOfMemory, employer les méthodes ***flush ()*** et ***clear ()*** régulièrement pour contrôler la taille du cache de premier niveau.

De plus, n'utilisez pas de cache de 2e niveau et utilisez la méthode ***scroll ()*** pour profiter de curseurs côté serveur BD adapté au gros volumes de données

Exemple insertion

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);

    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

Exemple *scroll()*

```
int batchSize = 25;
Session session = entityManager.unwrap( Session.class );
scrollableResults = session
                    .createQuery( "select p from Person p" )
                    .setCacheMode( CacheMode.IGNORE )
                    .scroll( ScrollMode.FORWARD_ONLY );
int count = 0;
while ( scrollableResults.next() ) {
  Person Person = (Person) scrollableResults.get( 0 );
  processPerson(Person);
  if ( ++count % batchSize == 0 ) {
    //flush a batch of updates and release memory:
    entityManager.flush();
    entityManager.clear();
  }
}
txn.commit();
```

Optimisations

Monitoring
Traitements batch
Cache de second niveau
Intercepteurs et évènements
Recommandations

Cache de second niveau

La session fournit un cache de la base durant la transaction pour une thread donné

Hibernate permet de configurer un cache de second niveau partagé par toutes les threads et à durée de vie plus longue

Les objets cachés peuvent être :

- Des entités
- Des collections (association *Many*)
- Les résultats d'une requête

Configuration

La configuration consiste à :

- Spécifier l'implémentation du fournisseur de cache
- Définir ce qui doit être mis en cache:
  - Au niveau global, (pour toutes les entités, …)
  - Au niveau de l’entité, de l’association ou de la requête
- Définir les régions de cache qui peuvent avoir différentes stratégies d'expulsion / de gestion

Implémentation

Hibernate est indépendant de l'implémentation du cache via l'interface ***RegionFactory***.

- La configuration consiste à préciser la classe d'implémentation

```
# Exemple EHCache
hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegiOnFactory
# Activer le cache nécessaire pour certaines versions d'Hibernate
hibernate.cache.use_second_level_cache=true
```

Hibernate supporte plusieurs implémentations :

- Les implémentations legacy comme *EHCache* et *InfiniSpan*
- Les implémentations standard *JCache* via le jar additionnel ***hibernate-jcache.jar***

Cache d’entité

Afin qu'une entité placée dans le cache de second niveau, il suffit de l'annoter via ***@Cache*** (ou tiliser ***<cache>*** dans le fichier de mapping)

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
----
```

- Au premier chargement, Hibernate stocke l'entité dans le cache
  (sa représentation base de données, pas l'objet complet).
- S'il a besoin de charger à nouveau l'entité, il utilisera le cache
- S'il a connaissance de mises à jour, cela invalidera la région associé à l'entité

Attributs de *@Cache*

@Cache a 3 attributs:

- ***usage*** (obligatoire): contrôler l'accès concurrent au cache en positionnant un niveau d'isolation en R/W.
- ***region*** (facultatif): la région de cache associée à l'entité. (A chaque entité cachée correspond une région qui a le nom de la classe par défaut)
- ***include*** (facultatif): ***all*** (par défaut) ou ***non-lazy*** (les propriétés lazy ne sont pas mises en cache)

Stratégie de concurrence

L'attribut *usage* peut prendre 4 valeurs :

- **READ_ONLY** : A utiliser si seules des opérations de lecture sont effectuées
- **READ_WRITE** : Lecture et écriture
- **NONSTRICT_READ_WRITE** : Les opérations d'écriture sont rares et il y a peu de chances que 2 transactions écrivent des données simultanément
- **TRANSACTIONAL** : Seulement dans un environnement JTA

Cache des collections

A la différence des associations *ToOne* qui sont cachées (les clés étrangères), les associations *ToMany* doivent être explicitement cachées avec des annotations *@Cache*

```
@OneToMany(cascade=CascadeType.ALL,
  fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage =
  CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

Cache de requête

Les résultats des requêtes peuvent être également cachées (même si les cas d'utilisation sont assez rare)

Le cache requête est désactivé par défaut, il faut donc configurer hibernate pour l'activer :

hibernate.cache.use_query_cache true

Ensuite une requête peut être cachée :

- programmatiquement :
  `org.hibernate.Query.setCacheable(true)`
- Une requête nommée peut également être annotée :

```
@NamedQuery(name = "products.getProducts",
        query = "SELECT product FROM Product product",
        hints = {
@QueryHint(name="org.hibernate.cacheable", value = "true"),
@QueryHint(name="org.hibernate.cacheMode", value="NORMAL")
})
```

Interactions avec le cache de 2nd niveau

L'attribut ***CacheMode*** permet de contrôler comment une session interagit avec le cache de second niveau. Plusieurs valeurs sont possibles :

- ***CacheMode.NORMAL*** : La session lit et écrit dans le cache de second niveau. **Valeur par défaut**
- ***CacheMode.GET*** : La session lit à partir du cache de second niveau, n'y écrit pas et l'invalide lors de la mise à jour des données
- ***CacheMode.PUT*** : La session ne lit pas dans le 2nd cache mais y écrit lorsqu'il charge des données de la base
- ***CacheMode.IGNORE*** : N'interagit pas avec le cache sauf pour invalider des données.

Gestion du cache

Si les politiques d'expiration et d'éviction ne sont pas définies, le cache peut croître indéfiniment et finir par consommer toute la mémoire disponible.

Hibernate laisse ces tâches de gestion aux fournisseurs de cache, car elles sont spécifiques à chaque implémentation.

Par exemple pour *EHCache* :

```
<ehcache>
    <cache name="org.formation.model.Foo"
  maxElementsInMemory="1000" />
</ehcache>
```

API de gestion du cache

L'API de *SessionFactory* permet de supprimer une entité ou une collection du cache :

```
// Une entité particulière
sessionFactory.evict(Cat.class, catId);
// Toutes les entités de type Cat
sessionFactory.evict(Cat.class);
// Une collection particulière
sessionFactory.evictCollection("Cat.kittens", catId);
// Toutes les collections kittens des entités Cat
sessionFactory.evictCollection("Cat.kittens");
```

Optimisations

Monitoring
Traitements batch
Cache de second niveau
Intercepteurs et évènements
Recommandations

Intercepteurs et évènements
Introduction

- Hibernate permet à l'application de réagir en fonction de certains évènements session.
  - L'interface *Interceptor* permet d'implémenter des méthodes de callback lors de l'ajout, modification ou suppression des entités.
  - Des listeners d'évènements session peuvent être définis dans le fichier de configuration ou via des annotations.

Intercepteurs
Exemple

```
public class AuditInterceptor extends EmptyInterceptor {
    private int updates, creates ;
    public boolean onFlushDirty(Object entity, Serializable id, Object[] newState,
  Object[] oldState,String[] propertyNames,Type[] types) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) { newState[i] =
  new Date() ; return true ;  }
     }
  public boolean onSave(Object entity, Serializable id, Object[] state, String[]
  propertyNames,Type[] types) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) { state[i] = new
  Date() ; return true ; }
            }
  }
  public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) { System.out.println("Creations: " + creates + ",
  Updates: " + updates) ;  }
        updates=0; creates=0 ;
     }
```

Intercepteurs

Les intercepteurs peuvent être positionnés :

- Sur la *session* :

```
Session session =
sf.openSession( new AuditInterceptor() );
```

- Sur la *session-factory* (valable pour toutes les sessions ouvertes) :

```
new Configuration().
setInterceptor( new AuditInterceptor() );
```

Configuration via session factory

```
SessionFactory sessionFactory =
   new MetadataSources( new
   StandardServiceRegistryBuilder().build() )
.addAnnotatedClass( Customer.class )
.getMetadataBuilder()
.build()
.getSessionFactoryBuilder()
.applyInterceptor( new LoggingInterceptor() )
.build();
```

Événements Hibernate
org.hibernate.event

Pour réagir à des événements particuliers dans la couche de persistance, on peut également utiliser les listeners d'évènements. C'est une alternative aux intercepteurs

Toutes les méthodes de l'interface *Session* sont associés à des évènements

Lorsqu'une méthode est appelée, Hibernate diffuse l'événement à tous les listeners enregistrés

- Hibernate positionne des listeners par défaut
- Un listener est considéré comme un singleton et ne doit pas stocker des informations d'états dans ses variables d'instance
- Les listeners s'enregistrent
  - Par configuration (XML ou annotation)
  - Programmatiquement, via l'objet *Configuration*

Exemple

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

Exemple
Classe Listener

```
public class MyLoadListener implements LoadEventListener {

    // Unique méthode de l'interface LoadEventListener

    public void onLoad(LoadEvent event, LoadEventListener.LoadType
 loadType)

            throws HibernateException {

        if ( !MySecurity.isAuthorized( event.getEntityClassName(),
 event.getEntityId() ) ) {

            throw MySecurityException("Unauthorized access");

        }

    }

}
```

Exemple
Enregistrement du listener

```
EntityManagerFactory entityManagerFactory = entityManagerFactory();
SessionFactoryImplementor sessionFactory =
   entityManagerFactory.unwrap( SessionFactoryImplementor.class );
sessionFactory
.getServiceRegistry()
.getService( EventListenerRegistry.class )
.prependListeners( EventType.LOAD, new SecuredLoadEntityListener() );


Customer customer = entityManager.find( Customer.class, customerId );
```

JPA Méthodes de callback

JPA permet également de définir des méthodes de call-back dans une classe entité ou dans une classe externe indiquée via l'annotation *@EntityListener*

Annotations possibles:

@PrePersist : Avant l'appel à persist

@PostPersist : Après l'insertion en base

@PreUpdate : Avant l'appel à merge

@PostUpdate : Après la mise à jour de l'entité

@PreRemove : Avant l'appel à remove

@PostRemove : Après la suppression en base

@PostLoad : Après un chargement de l'entité

Exemple

```
@Entity
public class Client implements
  java.io.Serializable{
.
.
.
@PostPersist
public void insertion() { ... }
@PostLoad
public void chargement() { ... }
}
```

Exemple EntityListener

```
public class Logging {


  @PostPersist
  public void insertion(Object entite) {
    …
  }
  @PostLoad
  public void chargement(Object entite)
    …
  }
}
-----------
@Entity
@EntityListeners(Logging.class)
public class Personne implements java.io.Serializable {
```

EntityListeners par défaut

Des entity listeners par défaut peuvent être déclarés dans le fichier de mapping *orm.xml*

→ Ils interviennent pour tout bean entité de l'unité de persistance

```
<entity-mappings>
...
  <entity-listeners>
    <entity-listener class="com.sample.Logging">
      <post-persist method-name="insertion"/>
      <post-load method-name="chargement"/>
    </entity-listener>
    <entity-listener class="com.sample.Perform"/>
  </entity-listeners>
...
</entity-mappings>
```

Optimisations

Traitements batch
Cache de second niveau
Intercepteurs et évènements
Monitoring
Recommandations

Gestion du schéma

Même si Hibernate fournit l'option update pour la propriété de configuration hibernate.hbm2ddl.auto, cette fonctionnalité ne convient pas à un environnement de production

Utiliser un outil de migration de schéma automatisée comme ***Flyway*** ou ***Liquibase***

Pêle-mêle

- Identifiant: SEQUENCE est le meilleur choix
- Les bidirectionnels sont généralement meilleurs que les uni-directionnels
- L'association *@ManyToOne* et *@OneToOne* côté enfant sont mieux représenter avec une clé étrangère.
- Il es préférable de mapper l'association *@OneToOne* à l'aide de *@MapsId* afin que la clé primaire soit partagée entre l'enfant et le parent
- Pour les collections unidirectionnelles, les Set sont le meilleur choix car ils génèrent les instructions SQL les plus efficaces.
- L'annotation *@ManyToMany* est rarement un bon choix car elle traite les 2 côtés comme des associations unidirectionnelles. On peut généralement le remplacer avec une classe d'association et 2 *ManyToOne*

Cache et SQL

Bien que le cache de deuxième niveau puisse réduire les temps de réponse des transactions, il existe d'autres options pour atteindre le même objectif :

- réglage du cache de base de données sous-jacente afin que les données puissent tenir en mémoire et que le trafic disque soit réduit.
- optimisation des instructions de base de données via le batching JDBC, la mise en cache des instructions, l'indexation.
- la réplication de la base de données pour augmenter le débit des transactions en lecture seule