Formation Optimisation PostgreSql, Hibernates

Ihab ABADI / UTOPIOS

SOMMAIRE – Partie 1

1. Rappel d'une utilisation efficaces des indexs.
2. Optimisation et bon pratique Index.
3. Identification des problèmes sur les requêtes.
4. Optimisation des requêtes et bonne pratiques (Réduction nombre requêtes, requêtes
5. parallèles...)
6. Utilisation des shared_buffers.
7. Partitionnement et méthodes de partitionnement.
8. Index et clés de partitionnement.
9. Opérations DDL, opération de maintenance sur les partitions.
10. Scalabilité et réplication.

SOMMAIRE – Partie 2

1. Utilisation des caches d'Hibernate
    1. Le cache de session.
    2. Le cache de second niveau.
    3. Le cache mapping.
    4. Les stratégies de cache.
    5. Avantages et inconvénients des différentes implémentations.
    6. Le cache de requête.
2. Problématiques liées à la concurrence d'accès.
3. Optimisation des associations.
    1. Présentation des principes et techniques.
    2. Classes techniques de type Stub ou Skeleton.
    3. Pattern proxy.
    4. Cas des associations bi-directionnelles.
    5. Gestion de l'attribut inverse.
    6. Associations polymorphes.
4. Problèmes liés à l’héritage.
5. Suivi des performances.

Rappel d'une utilisation efficaces des indexs.

- Uniquement destinés à l'optimisation
- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire récupérer directement l'enregistrement de l'index
- Un index facilite :
  - certains tris
  - certains agrégats
- Utilisé pour les contraintes (PK, unicité)
- Les index sont à la charge du développeur.

Rappel d'une utilisation efficaces des indexs.

- PostgreSQL propose différentes formes d'index :
  - index classique sur une seule colonne d'une table.
  - index composite sur plusieurs colonnes d'une table.
  - index partiel, en restreignant les données indexées avec une clause WHERE.
  - index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table.
  - index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table.

Rappel d’une utilisation efficaces des indexs.

- L’index n’est pas gratuit.
- Ralentit les écritures.
- Maintenance.
- Place disque.

Rappel d'une utilisation efficaces des indexs.

- Un index ne résout pas tout
- Importance de la conception du schéma de données
- Importance de l'écriture de requêtes SQL correctes
- Types d'index dans PostgreSQL
  - Défaut : B-tree classique (balancé)
  - UNIQUE (préférer la contrainte)
  - Multicolonne, fonctionnel, partiel, couvrant
  - Index spécialisés : hash, GiST, GIN, BRIN…

Optimisation et bon pratique Index

- On indexe pour une requête, ou idéalement une collection de requêtes.
- On n’indexe pas « une table ».
- Identifier les requêtes nécessitant un index.
- Créer les index permettant de répondre à ces requêtes.
- Valider le fonctionnement.

Optimisation et bon pratique Index

- Indexation des colonnes faisant référence à une autre.
  - Optimisation des performances des DML
  - Optimisation performances des jointures
- Optimiseur peut décider de ne pas utiliser les indexs dans les cas suivant:
  - L'optimiseur pense qu'il n'est pas rentable
  - La requête n'est pas compatible
  - Pas le bon type.
  - Utilisation de fonctions.
  - Pas les bons opérateurs.
  - Index redondants.

Optimisation et bon pratique Index

- Nous avons plusieurs possibilité pour la création d'index.
  - Index partiels
  - Index fonctionnels
  - Index couvrants
  - …

TP

- En utilisant la base de données magasin.
- Index simple
  - Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.
  - Afficher le plan de la requête.
  - Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan.
  - Pour optimiser ces requêtes, on peut créer un index permettant de répondre à ces requêtes.
  - Afficher le plan et comparer.
  - Écrire la requête listant les commandes pour client_id = 3. Afficher son plan.
  - Créer un index pour accélérer cette requête.
  - Afficher de nouveau son plan.

TP

- En utilisant la base de données magasin.
- Sélectivité
  - Écrivez une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').
  - Afficher le plan d'exécution.
  - Ajoutez un index sur la colonne type_client, et rejouez les requêtes précédentes.
  - Affichez le plan d'exécution et comparer.

TP

- En utilisant la base de données magasin.
- Sélectivité
  - Écrivez une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').
  - Afficher le plan d'exécution.
  - Ajoutez un index sur la colonne type_client, et rejouez les requêtes précédentes.
  - Affichez le plan d'exécution et comparer.

TP

- En utilisant la base de données magasin.
- Index partiels
  - Nous souhaitons mettre en place un système d'alerte pour assurer un suivi de qualité sur les lots en fonction de leur état:
    - En dépôt depuis plus de 12 h, mais non expédié.
    - Expédié depuis plus de 3 jours, mais non réceptionné.
  - Écrire les requêtes correspondant à ce besoin fonctionnel.
  - Créer l'index pour optimiser les requêtes.
  - Affichez les plans d'exécution.

TP

- En utilisant la base de données magasin.
- Index non utilisés
  - Un développeur cherche à récupérer les commandes dont le numéro d'expédition est 190774 avec un cast
  - Écrire la requête.
  - Afficher le plan de la requête.
  - Créer un index pour améliorer son exécution.
  - L'index est-il utilisé ? Quel est le problème ?
  - Écrivez une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.
  - Créez un index pour améliorer l'exécution de cette requête.
  - Pourquoi celui-ci n'est-il pas utilisé ?
  - Faites le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

Identification des problèmes sur les requêtes

- L'identification des problèmes se base sur une utilisation optimale des outils de surveillances.
  - Quelles sont les requêtes lentes ?
  - Quelles sont les requêtes les plus fréquentes ?
  - Quelles requêtes génèrent des fichiers temporaires ?
  - Quelles sont les requêtes bloquées ?
  - Progression d'une requête
  - …

Identification des problèmes sur les requêtes

- Utilisation de log_min_duration_statements pour tracer les requêtes.
- Utilisation log_temp_files.
- Utilisation Vue pg_stat_statements.
  - Les requêtes sont normalisées.
  - Indique les requêtes exécutées, avec durée d'exécution, utilisation du cache, etc.
  - Durée d'exécution :
    - total_exec_time
    - min_exec_time/max_exec_time
  - stddev_exec_time
    - mean_exec_time
  - Avant la version 13, les colonnes n'avaient pas _exec dans leur nom.
  - Nombre de lignes retournées.
  - …

Identification des problèmes sur les requêtes

- L'identification des requêtes bloquées.
  - Vue pg_stat_activity
    - colonnes wait_event et wait_event_type
  - Vue pg_locks
    - colonne granted
  - Fonction pg_blocking_pids

Identification des problèmes sur les requêtes

- L'identification des problèmes sur les requêtes passe par la surveillance écriture.
- La quantité de données écrites.
- Utilisation des traces checkpoints (log_checkpoints).
- Affichage des informations à chaque checkpoint.
  - mode de déclenchement
  - volume de données écrits
  - durée du checkpoint
- Vue pg_stat_bgwriter
  - Activité des écritures dans les fichiers de données
  - Visualisation du volume d'allocations et d'écritures

Optimisation des requêtes et bonne pratiques

- Utilisation d'outils tiers pour l'analyse les requêtes.
- Utilisation pgBadger pour l'analyse.
- Utilisation d'outils tiers pour l'optimisation des requêtes.
- Utilisation de PoWA.

Utilisation des shared_buffers

- shared_buffers permet de configurer la taille du cache disque de PostgreSQL.
- Chaque requêtes de lecture ou de mise à jour, par exemple, les lignes sont mise en cache dans un premier temps.
- Ce cache est en mémoire partagée, et donc commun à tous les processus PostgreSQL.
- Il faut lui donner une grande taille, tout en conservant t la majorité de la mémoire pour le cache disque du système.
- Pour dimensionner shared_buffers sur un serveur dédié à PostgreSQL, la norme est d'utiliser 25% de la mémoire disponible et ne pas dépasser 40%

Partitionnement et méthodes de partitionnement

- Faciliter la maintenance de gros volumes.
- Performances
  - parcourir complet sur de plus petites tables.
  - purger par partitions entières.

Partitionnement et méthodes de partitionnement

- Partitionnement applicatif
  - Intégralement géré au niveau applicatif.
  - Complexité pour le développeur
  - Intégrité des liens entre les données.

Partitionnement et méthodes de partitionnement

Partitionnement par héritage

- Table principale :
  - table mère définie normalement
- Tables filles :
  - CREATE TABLE primates(debout boolean) INHERITS (mammiferes) ;
  - héritent des propriétés de la table mère
  - mais pas des contraintes, index et droits
- Insertion applicative ou par trigger.

Partitionnement et méthodes de partitionnement

Partitionnement déclaratif

- Mise en place et administration simplifiées car intégrées au moteur
- Gestion automatique des lectures et écritures
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles
  - partition par défaut

Index et clés de partitionnement

Partitionnement par liste

- Liste de valeurs par partition
- Clé de partitionnement forcément mono-colonne
- Créer une table partitionnée :
  CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
- Ajouter une partition :
  CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);

Index et clés de partitionnement

Partitionnement par intervalle

- Intervalle de valeurs par partition
- Clé de partitionnement mono- ou multi-colonnes
- Deux mots clés
  - MINVALUE pour la valeur minimale
  - MAXVALUE pour la valeur maximale
- Créer une table partitionnée :
- CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
- Ajouter une partition :
- CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);

Index et clés de partitionnement

Partitionnement par hachage

- Hachage de valeurs par partition
  - indiquer un modulo et un reste
- Clé de partitionnement mono- ou multi-colonnes
- Créer une table partitionnée :
- CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);
- Ajouter les partitions :

```
CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2);
```

Index et clés de partitionnement

Clé de partitionnement multi-colonnes

- Clé sur plusieurs colonnes acceptée
  - si partitionnement par intervalle ou hash, pas pour liste
- Créer une table partitionnée avec une clé multi-colonnes :
  CREATE TABLE t1(c1 integer, c2 text, c3 date)
  PARTITION BY RANGE (c1, c3);
- Ajouter une partition :
  CREATE TABLE t1_a PARTITION OF t1
  FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11');

Index et clés de partitionnement

Partition par défaut

- Pour le partitionnement par liste et par intervalle
- Toutes les données n’allant pas dans les partitions définies iront dans la partition par défaut
- CREATE TABLE t2_autres PARTITION OF t2 DEFAULT

Index et clés de partitionnement

Indexation

- Propagation automatique.
- Index supplémentaires par partition possibles.
- Clés étrangères entre tables partitionnées.

Opérations DDL, opération de maintenance sur les partitions

Attacher une partition

- Vérification du respect de la contrainte
  - parcours complet de la table
  - potentiellement lent
- Si la clé de partitionnement concerne des éléments déjà dans la partition par défaut
  - erreur à l'ajout de la nouvelle partition
  - commencer par détacher la partition par défaut
  - ajouter la nouvelle partition
  - déplacer les données de l'ancienne partition par défaut
  - rattacher la partition par défaut

Opérations DDL, opération de maintenance sur les partitions

Détacher une partition

- ALTER TABLE … DETACH PARTITION …
- Simple et rapide
- Mais nécessite un verrou exclusif

- Supprimer une partition
  - Un simple DROP TABLE

Opérations DDL, opération de maintenance sur les partitions

Fonctions de gestion

- pg_partition_root() : récupérer la racine d'une partition
- pg_partition_ancestors() : parents d'une partition
- pg_partition_tree() : liste entière des partitions

TP

- En utilisant la base de données cave.
- Nous allons partitionner la table stock sur l'année.
- Renommez stock en stock_old.
  Créer une table partitionnée stock vide, sans index pour le moment.
- Créer les partitions de stock, avec la contrainte d'année : stock_2001 à stock_2005.
- Insérer tous les enregistrements venant de l'ancienne table stock.
- Vérifier la présence d'enregistrements dans stock_2001
- Vérifier qu'il n'y en a aucun dans stock.
- Vérifier qu'une requête sur stock sur 2002 ne parcourt qu'une seule partition.

TP

- Remettre en place les index présents dans la table stock originale.
- Quel est le plan pour la récupération du stock des bouteilles du vin_id 1725, année 2003 ?
- Essayer de changer l'année de ce même enregistrement de stock (la même que la précédente). Pourquoi cela échoue-t-il ?
- Supprimer les enregistrements de 2004 pour vin_id = 1725. Retenter la mise à jour.
- Pour vider complètement le stock de 2001, supprimer la partition stock_2001.
- Tenter d'ajouter au stock une centaine de bouteilles de 2006. Pourquoi cela échoue-t-il ?
- Créer une partition par défaut pour recevoir de tels enregistrements.

TP

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?
- Pour créer la partition sur 2006, au sein d'une seule transaction :
  - détacher la partition par défaut ;
  - y déplacer les enregistrements mentionnés ;
  - ré-attacher la partition par défaut.

Scalabilité et réplication

- Réplication interne
  - réplication physique
  - réplication logique
- Quelques logiciels externes de réplication

Scalabilité et réplication

Réplication interne physique

- Réplication
  - asymétrique
  - asynchrone (défaut) ou synchrone (et selon les transactions)
- Secondaires
  - non disponibles (*Warm Standby*)
  - disponibles en lecture seule (*Hot Standby*)
  - cascade
  - retard programmé

Scalabilité et réplication

Réplication interne logique

- Réplique les changements
  - sur une seule base de données
  - d'un ensemble de tables défini
- Principe Éditeur/Abonnés
- Création d'une publication sur un serveur
- Souscription d'un autre serveur à cette publication
- Limitations :
  - DDL, Large objects, séquences
  - peu adaptée pour un *failover*

Scalabilité et réplication

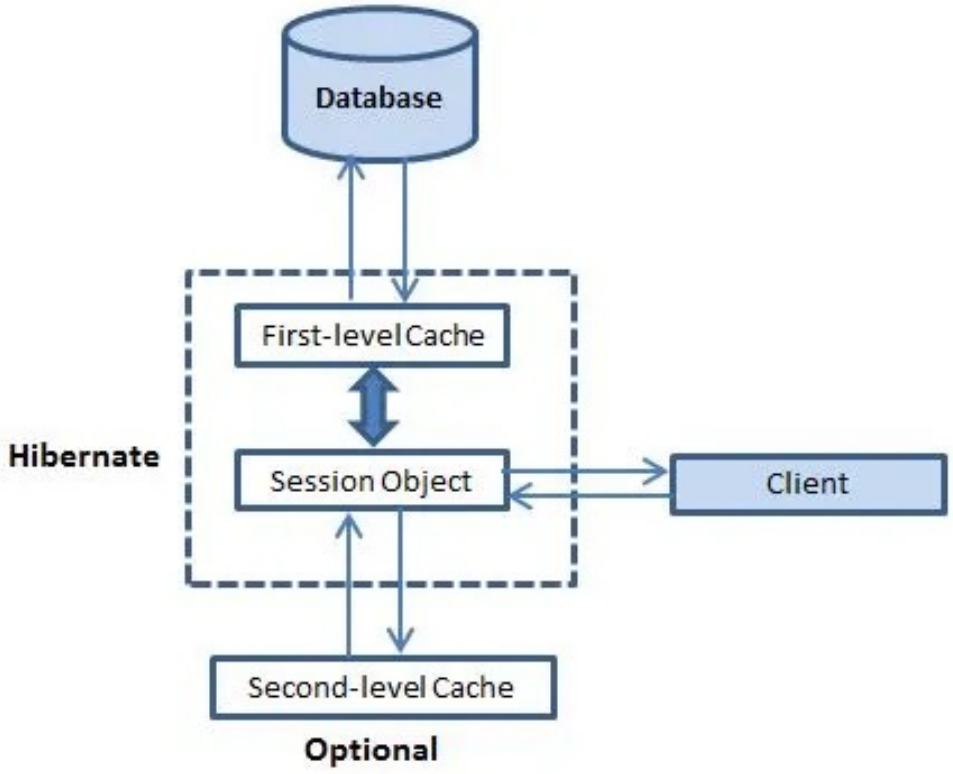
Réplication externe

- Outils les plus connus :
  - Pgpool
  - Slony, Bucardo
  - pgLogical

Utilisation des caches d'Hibernate

- La mise en cache concerne l'optimisation des performances de l'application.
- Elle se situe entre votre application et la base de données.
- Elle diminue autant que possible le nombre d'accès à la base de données afin d'améliorer les performances des applications critiques.

Utilisation des caches d'Hibernate



Cache de session

- Le cache de session et est un cache par défaut par lequel toutes les requêtes doivent passer.
- L'objet Session conserve un objet avant de le valider dans la base de données.
- Hibernate essaie de retarder la mise à jour aussi longtemps que possible pour réduire le nombre d'instructions SQL de mise à jour émises.
- A la fermeture da session, tous les objets mis en cache sont soit, perdus, soit, conservés ou mis à jour dans la base de données.

Cache de second niveau

- le cache de second niveau est une scope de SessionFactory.
- Il est partagé par toutes les sessions créées avec la même fabrique de sessions.
- Si une instance est déjà présente dans le cache de premier niveau, elle est renvoyée à partir de là
- Si une instance est introuvable dans le cache de premier niveau et que l'état de l'instance correspondant est mis en cache dans le cache de second niveau, les données y sont extraites et une instance est assemblée et renvoyée.
- Sinon, les données nécessaires sont chargées à partir de la base de données et une instance est assemblée et renvoyée.

Cache de second niveau

- Une fois l'instance stockée dans le contexte de persistance (cache de premier niveau), elle y est renvoyée dans tous les appels ultérieurs au sein de la même session jusqu'à ce que la session soit fermée ou que l'instance soit manuellement expulsée du contexte de persistance. De plus, l'état de l'instance chargée est stocké dans le cache second niveau s'il n'y était pas déjà.

Cache de second niveau

- La mise en cache de second niveau d'Hibernate est conçue pour ignorer le fournisseur de cache réel utilisé.
- Hibernate a seulement besoin d'être fourni avec une implémentation de l'interface org.hibernate.cache.spi.RegionFactory qui encapsule tous les détails spécifiques aux fournisseurs de cache réels.

Cache de second niveau

- Activation du cache de second niveau
  - hibernate.cache.use_second_level_cache=true
    hibernate.cache.region.factory_class=factory_cache_provider

Cache de second niveau

- Rendre une entité cachable.
- Afin de rendre une entité éligible pour la mise en cache de second niveau, nous l'annotons avec l'annotation @org.hibernate.annotations.Cache spécifique à Hibernate et spécifions une stratégie de concurrence de cache.
- Pour chaque classe d'entité, Hibernate utilisera une région de cache séparée pour stocker l'état des instances de cette classe. Le nom de la région est le nom complet de la classe.

Les stratégies de cache.

- En fonction des cas d'utilisation, nous pouvons choisir l'une des stratégies de cache suivantes :
- READ_ONLY
  - utilisé uniquement pour les entités qui ne changent jamais (une exception est levée si une tentative de mise à jour d'une telle entité est effectuée). C'est très simple et performant. Très approprié pour certaines données de référence statiques qui ne changent pas
- NONSTRICT_READ_WRITE
  - le cache est mis à jour après qu'une transaction qui a modifié les données affectées a été validée. Ainsi, une cohérence forte n'est pas garantie et il existe une petite fenêtre de temps pendant laquelle des données obsolètes peuvent être obtenues à partir du cache. Ce type de stratégie convient aux cas d'utilisation qui peuvent tolérer une éventuelle cohérence

Les stratégies de cache.

- READ_WRITE :
  - cette stratégie garantit une cohérence forte qu'elle atteint en utilisant des verrous "soft"
    - lorsqu'une entité mise en cache est mise à jour, un verrou logiciel est également stocké dans le cache pour cette entité, qui est libéré après la validation de la transaction. Toutes les transactions simultanées qui accèdent aux entrées à verrouillage logiciel récupèrent les données correspondantes directement à partir de la base de données
- TRANSACTIONNEL :
  - les modifications du cache sont effectuées dans les transactions XA distribuées. Une modification dans une entité mise en cache est soit validée, soit annulée dans la base de données et le cache dans la même transaction XA

Gestion du caches

- Si les règles d'expiration et d'éviction ne sont pas définies, le cache peut croître indéfiniment et finir par consommer toute la mémoire disponible.
- Dans la plupart des cas, Hibernate laisse de telles tâches de gestion de cache aux fournisseurs de cache, car elles sont en effet spécifiques à chaque implémentation de cache.

Cache de collections

- Les collections ne sont pas mises en cache par défaut et nous devons les marquer explicitement comme pouvant être mises en cache.

Cache de requêtes

- Les résultats des requêtes HQL peuvent également être mis en cache. Ceci est utile si vous exécutez fréquemment une requête sur des entités qui changent rarement.
- Pour activer le cache des requêtes, définissez la valeur de la propriété hibernate.cache.use_query_cache sur true.
- Pour chaque requête, vous devez indiquer explicitement que la requête peut être mise en cache (via un indicateur de requête org.hibernate.cacheable).

Cache de requêtes

- Comme c'est le cas avec les collections, seuls les identifiants des entités renvoyés à la suite d'une requête pouvant être mise en cache sont mis en cache, il est donc fortement recommandé d'activer le cache de second niveau pour ces entités.
- Il existe une entrée de cache pour chaque combinaison de valeurs de paramètres de requête (variables de liaison) pour chaque requête, donc les requêtes pour lesquelles vous attendez de nombreuses combinaisons différentes de valeurs de paramètres ne sont pas de bons candidats pour la mise en cache.
- Les requêtes qui impliquent des classes d'entités pour lesquelles il y a des changements fréquents dans la base de données ne sont pas non plus de bons candidats pour la mise en cache, car elles seront invalidées chaque fois qu'il y a un changement lié à l'une des classes d'entités participant à la requête, que les instances modifiées soient ou non mis en cache dans le cadre du résultat de la requête ou non.

Cache de requêtes

- Par défaut, tous les résultats du cache de requête sont stockés dans la région org.hibernate.cache.internal.StandardQueryCache. Comme pour la mise en cache d'entité/collection, vous pouvez personnaliser les paramètres de cache pour cette région afin de définir des politiques d'éviction et d'expiration en fonction de vos besoins. Pour chaque requête, vous pouvez également spécifier un nom de région personnalisé afin de fournir différents paramètres pour différentes requêtes.
- Pour toutes les tables interrogées dans le cadre de requêtes pouvant être mises en cache, Hibernate conserve les horodatages de la dernière mise à jour dans une région distincte nommée org.hibernate.cache.spi.UpdateTimestampsCache. Connaître cette région est très important si vous utilisez la mise en cache des requêtes, car Hibernate l'utilise pour vérifier que les résultats des requêtes mises en cache ne sont pas obsolètes. Les entrées de ce cache ne doivent pas être supprimées/expirées tant qu'il existe des résultats de requête mis en cache pour les tables correspondantes dans les régions de résultats de requête. Il est préférable de désactiver l'éviction et l'expiration automatiques pour cette région de cache, car elle ne consomme pas beaucoup de mémoire de toute façon

Problématiques liées à la concurrence d'accès

- Hibernate s'appuie sur le niveau d'isolation fourni par le SGBD, et ne tente pas de ré-implanter des protocoles de concurrence.
- Le mode par défaut dans la plupart des systèmes est **read committed** ou **repeatable read**.
- Ce niveau d'isolation (par défaut) n'ore pas toutes les garanties
- il est indipensable de se poser sérieusement la question des risques liés à la concurrence d'accès.
- Dans une application transactionnelle, le mode **read uncommitted** ne devrait même pas être envisagé.
  Il reste donc comme possibilités :
  - d'accepter le mode par défaut, après évaluation des risques;
  - ou de passer en mode **serializable**, en acceptant les conséquences (performances moindres, risques de **deadlock**);
  - ou, enfin, d'introduire une dose de gestion "manuelle" de la concurrence en eectuant des verrouillages préventifs (dits, parfois, "verrouillage pessimiste").

Problématiques liées à la concurrence d'accès

- La configuration Hibernate permet de spécifier le mode d’isolation choisi pour une application:
- hibernate.connection.isolation = <val>
- où val est un des codes suivants:
  - 1 pour read uncommitted
  - 2 pour read committed
  - 3 pour repeatable read
  - 4 pour serializable

Problématiques liées à la concurrence d'accès

- La configuration Hibernate permet de spécifier le mode d'isolation choisi pour une application:
- hibernate.connection.isolation = <val>
- où val est un des codes suivants:
  - 1 pour read uncommitted
  - 2 pour read committed
  - 3 pour repeatable read
  - 4 pour serializable
- On peut donc spécifier un niveau d'isolation, qui s'applique à **toutes** les sessions,
- même celles qui ne présentent pas de risque transactionnel.
- Dans ces conditions, choisir le niveau maximal (**serializable**) systématiquement représente une pénalité certaine.
- Il semble préférable d'utiliser le niveau d'isolation par défaut du SGBD, et de changer le mode d'isolation à **serializable** ponctuellement pour les transactions sensibles.

Problématiques liées à la concurrence d'accès

- Il ne semble malheureusement pas possible, avec Hibernate, d’aecter simplement le niveau d’isolation pour une session.
- On en est donc réduit à passer par l’objet **Connexion** de JDBC
- **Le mode serializable, malgré ses inconvénients (apparition d’interblocages) est le plus sûr pour garantir l’apparition d’incohérences dans la base**, dont la cause est très dicile à identifier.
- Une autre solution consiste à verrouiller explicitement, au moment de leur mise en cache, les objets que l’on va modifier.

Problématiques liées à la concurrence d'accès

- Hibernate se contente de transmettre les requêtes de verrouillage au SGBD aucun verrou en mémoire n'est posé
- (cela n'a pas de sens car chaque application ayant son propre cache, un verrou n'aurait aucun eet concurrentiel).
- La principale utilité des verrous est de gérer le cas de la **lecture** d'une ligne/objet, suivie de **l'écriture** de cette ligne.
- Par défaut, la lecture pose un verrou **partagé** qui n'empêche pas d'autres transactions de lire à leur tour.

Problématiques liées à la concurrence d'accès

- Le principe du verrouillage explicite est donc d'eectuer une lecture qui **anticipe** l'écriture qui va suivre.
- C'est l'eet de la clause **for update**.
- Le **for update** déclare que les lignes sélectionnées vont être modifiées ensuite.
  Pour éviter qu'une autre transaction n'accède à la ligne entre la lecture et l'écriture,
- le système va alors poser un **verrou exclusif**. Le risque de mise à jour perdue disparaît.
- Avec Hibernate, l'équivalent du **for update** est la pose d'un verrou au moment de la lecture.

Problématiques liées à la concurrence d'accès

- L’énumération LockMode implique l’envoi d’une requête au SGBD avec une demande de verrouillage. Les valeurs possibles sont :
- LockMode.NONE. Pas d’accès à la base pour verrouiller (mode par défaut). LockMode.READ. Lecture pour vérifier que l’objet en cache est synchronisé avec la base.
- LockMode.UPGRADE. Pose d’un verrou exclusif sur la ligne.
- LockMode.WRITE. Utilisé en interne.
- Gérer la concurrence dans le code d’une application est une opération lourde et peu fiable.
- Se limiter au principe simple suivant : quand on lit une ligne que l’on va modifier ensuite, on place un verrou avec LockMode.UPGRADE.
- Pour tous les autres cas, n’utilisez pas les autres modes et laissez le SGBD appliquer son protocole de concurrence.

Optimisation des associations

- Par défaut, Hibernate utilise le chargement différé par select pour les collections et le chargement différé par proxy pour les associations vers un seul objet. Ces valeurs par défaut sont valables pour la plupart des associations dans la plupart des applications.
- L'accès à une association définie comme "différé", hors du contexte d'une session Hibernate ouverte, entraîne une exception.

Optimisation des associations

- Hibernate ne supporte pas le chargement différé pour des objets détachés. La solution à ce problème est de déplacer le code qui lit à partir de la collection avant le "commit" de la transaction.
- Une autre alternative est d'utiliser une collection ou une association non "différée" en spécifiant lazy="false" dans le mappage de l'association.
- Cependant il est prévu que le chargement différé soit utilisé pour quasiment toutes les collections ou associations.
- Si vous définissez trop d'associations non "différées" dans votre modèle objet, Hibernate va finir par devoir charger toute la base de données en mémoire à chaque transaction.

Pattern proxy

- Le chargement différé des collections est implémenté par Hibernate qui utilise ses propres implémentations pour des collections persistantes.
- L'entité qui est pointée par l'association doit être masquée derrière un proxy.
- Hibernate implémente l'initialisation différée des proxies sur des objets persistants via une mise à jour à chaud du bytecode
- Par défaut, Hibernate génère des proxies (au démarrage) pour toutes les classes persistantes et les utilise pour activer le chargement différé des associations many-to-one et one-to-one.