**Optimistic lock működése:** Ennek használata akkor javasolt, amikor tudjuk, hogy ritkán lesz konfliktus. Ilyenkor az egyik tranzakció sikeresen lefut, a másik elszáll, rollbackelődik, exception is dobódik. Ez a megoldás nyilván nem lockolja fizikailag az adatbázisban a táblát, csak mielőtt perzisztálna, előtte megnézi, hogy az adatbázisban a verzió szám maradt-e az amit ő felolvasott, vagy megváltoztatták, és akkor rollbackeli a tranzakciót.

Lehetséges példa:

1. Adam's transaction reads data X
2. Barbara's transaction reads data X
3. Adam's transaction modifies data X, and changes it to XA
4. Adam's transaction **writes data XA**
5. Barbara's transaction modifies data X and changes it to XB
6. Barbara's transaction tries to write data XB, but **receives and error**
7. Barbara needs to read data XA (or start a completely new transaction)
8. Barbara's transaction modifies data XA and changes it to XAB
9. Barbara's transaction **writes data XAB**

Látható, hogy Barbara tranzakciója rá van kényszerítve, hogy újraolvassa Adam módosításait.

**Pessimistic lock működése:** Amikor egy tranzakciónak egy entitást kell módosítania, ami párhuzamosan más tranzakciók álltal is módosítható, tranzakciók lockolhatják az entitást. Az összes tranzakció visszatartva van, amíg aki először lockolta be nem fejezi amit akart.

Egy pessimistic lock példa:

1. Adam’s transaction reads data X
2. Adam’s transaction locks X
3. Barbara’s transaction wants to read data X, but waits as X is already locked
4. Adam’s transaction modifies data X, and changes it to XA
5. Adam’s transaction **writes data XA**
6. Barbara’s transaction reads data XA
7. Barbara’s transaction modifies data XA and changes it to XAB
8. Barbara’s transaction **writes data XAB**

Ebben az esetben Barbara tranzakciója még olvasni sem tudja az adatot addig, amíg Adam be nem fejez a tranzakcióját.

**Optimistic lock típusok:**

* LockType.OPTMISTIC: Ez felel meg a régebbi verziókban a (READ) megoldásnak. Ez a verzió az entitás változtatásakor növeli annak verzió számát. Ha csak olvassuk az entitást, akkor nem növel verziószámot.

**public** Product getProductOptimisticJustRead(**int** id){

Product product = manager.getManager().find(Product.**class**,id);

manager.getManager().lock(product, LockModeType.***OPTIMISTIC***);

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

**return** product;

}

A fenti metódust meghívva nem fogja a verziót növelni, hisz csak olvassuk az entitást.

A következő metódusban már változtatjuk az állapotát az entitásnak, így a tranzakció végén, amikor perzisztál, a verzió számot is növeli eggyel.

**public** Product getProductOptimisticReadAndWrite(**int** id){

Product product = manager.getManager().find(Product.**class**, id);

manager.getManager().lock(product, LockModeType.***OPTIMISTIC***);

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

product.setPrice(**new** Random().nextInt(1000));

**return** product;

}

* LockType. ***OPTIMISTIC\_FORCE\_INCREMENT:*** Ebben az esetben a tranzakció végén eggyel minden esetben megnöveli a version értékét.

**public** Product getProductOptimisticForceIncrementJustRead(**int** id){

Product product = manager.getManager().find(Product.**class**,id)

manager.getManager().lock(product, LockModeType.***OPTIMISTIC\_FORCE\_INCREMENT***);

System.***out***.println(product.getVersion());

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

**return** product;

}

A fenti esetben látható, hogy csak olvassuk az entitást, viszont a tranzakció legvégén így is növeli a verziót eggyel.

Kiad egy selectet, majd öt másodperccel később egy update lekérdezéssel növeli a verzió értékét.

**public** Product getProductOptimisticForceIncrementReadAndWrite(**int** id){

Product product = manager.getManager().find(Product.**class**, id);

manager.getManager().lock(product, LockModeType.***OPTIMISTIC\_FORCE\_INCREMENT***);

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

product.setPrice(**new** Random().nextInt(1000));

**return** product;

}

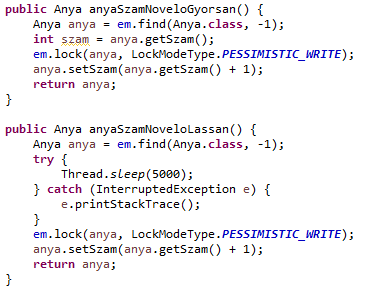
A fenti megoldás kettővel növeli a verzió értékét. Mindegyik növelés az 5 másodperces várakozás után történik. Először lefut a find select lekérdezése. Aztán update fut le a product-re(nyilván itt minden adattagot updatel, a verziót is beleértve).

Majd egy sima verzió növelő update lekérdezés is lefut.

**Pessimistic lock:**Ez rögtön lelockolja adatbázis szinten az entitást, ellenben az optimista megoldással, ami commitig vár, és reméli, hogy az entitás nem változott a findolás óta,(az entitás nem változott az adatbázisban a legutóbbi feolvasás óta). A pessimistic lock az szinkorn, garantlja, hogy a lockolt entitás nem fog addig módosulni más tranzakció álltal, amíg az aktuális tranzakció be nem fejeződik. Sajnos az alkalmazások skálázhatósságát rontja, mivel párhuzamosságot nem enged. Ha sűrűn keletkezhet konkurens hozzáférés egy entitáshoz, akkor pessimistic lock használata érdemes, hisz optimistic loknál a sok újrapróbálkozás ront a teljesítményen.

**Pessimistic\_Write lockType:** Pessimistic lock esetén ez az amit nagy általánosságban használni szoktunk.

Lássunk egy példát. Van egy Anya osztályunk. Ezen szeretnénk lockolást végrehajtani.

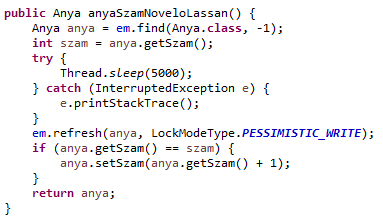
****

Lássuk mi történik, az anyaSzamNoveloLassan() metódusban: Látható, hogy az elején felfindoljuk az anya entitást, majd 5 másodpervet várunk(tekintsük úgy hogy 5 másodperces más művelet történik.). Majd lockoljuk az entitást, és ezt követően módosítunk rajta. Látható, hogy a lockolást közvetlenül a módosítás előtt hívjuk meg, hogy minnél kevesebb ideig legyen lockolva a rekord.

Lássuk, hogy mit csinál az anyaSzamNovelogyorsan metódus: Ez annyival másabb mint az előző, hogy nincs 5 másodperces várakozás, ez rögtön módosít az entitáson.

Lássunk egy olyan szituációt, amikor ebből probléma lehet. Lefut az anyaSzamNoveloLassan() metódus, ami gyorsan felfindolja az entitást, majd várakozik 5 másodpercig, miközben ő várakozik, az anyaSzamNovelogyorsan() metódus teljesen lefut, és növeli a Szam field értékét eggyel(legyen ilyenkor 3). Letelt az 5 másodperc, és lockol a lassú metódus majd növel. Ezzel az a probléma, hogy a gyors metódus változtatásait teljesen felülíírja, azaz az ignorálódik, hisz a lassú az a felfindolt értékével dolgozik. Ez az eset akkor következhet be, ha az Anya entitásnak nincs version adattagja. Ugyanis ha van @Version adattagja, akkor akkor em.lock(Pessimistic\_Lock) híváskor is lefut az optimistic lock ellenőrzés, és bizony Optimistic lock exceptionnel elszáll a második esetben, hisz Pessimistic\_write módosításkor is növelődik a version értéke. Tehát, ha van version, akkor a hosszú metódus elszáll,hisz a findolása után már módosítás történt.

Megoldás lehet a locking refresh() metódus használata.

****

Az eset ugyan az mint legutóbb, a lassu metódus végrehajtása közben a gyors módosítást végez az adatbázisban. Mit is csinál az átalakított metódus: Lekéri a felfindolt entitás szam adattagját, ugyanis számít rá, hogy esetleg konkurensen változtathatják. Eltelik öt másodperc, amikor gyanítja, hogy megváltoztatták a szam adattag értékét. Használja a lockoló refresh() metódust, amivel szinkronizálja a felfindolt entitást az adatbázissal. Ha azt látja, hogy módosítottak a szam adattagon, akkor nemcsinál semmit, ha pedig az érték ugyan az mint findoláskor, akkor módosít. A lockoló refresh() azért király, mert egyszerre szinkronizál az adatbázissal, és egyúttal lockolja is az adott entitást, hogy mi használjuk most.

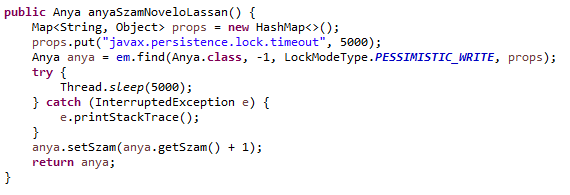
Azért akarjuk a pessimistic lockot használni, mert nem commitkor szeretnénk meglepődni, és a tranzakció végén kezelni az ütközésből következő kivételeket.

Egyik megoldás lehetne még, hogy find után rögtön pessimistic lockolunk, ekkor viszont hosszabb és felesleges időkre is lockolva van az entitás. Nem beszélve arról, hogy a skálázhatósságon tovább ront.

**Pessimistic\_Read LockType:** repetable read kiküszöböésére jó. Amikor egy entitást pessimistic\_read lockkal zárolunk, és módosítunk az entitáson, akkor a lock felfog upgradelődni pessimistic\_write lockra. Azonban ez az upgradelődés nem történik meg, amíg az entitás nem flushölődik, tehát ha hiba van(konkurens módosítás történt), akkor csak a tranzakció commitnál fog előjönni, ez viszont ekvivalens az optimistic lockkal, hisz ott van az, hogy leghamarabb commitnál jöhet elő a hiba. Ezt a típusú lockolást egyébként nem mindegyik adatbázismotor támogatja.

**Pessimistic\_forced\_increment:** Ez növeli a versiont, akár volt változtatva, akár nem.

**Pessimistic lock timeout megoldás:**

****A fenti példában azt láthatjuk, hogyan állíthatjuk be, hogy mennyi ideig várjon egy tranzakció arra, hogy ő is lockolhassa az entitást. Ez a megoldás H2 adatbázissal nem működik, és előfordulhat, hogy más adatbázisokban vagy nem vagy máshogy oldották meg a háttérben, ezért ez egy instabil pont.