JOINS

Joins är av förklarliga skäl en nödvändighet i större applikationer för att minimera sin databas storlek. Dock förlorar man tid i sina frågor, och riktigt komplicerade frågor kan ta flera sekunder vilket inte alltid uppskattas av en användare.

För att minimera låsningstiden finns det ett par hjälpmedel i hur man styr upp en fråga.

För att ens kunna ha joins mellan tabellerna, skall dem "mappas".

Det finns fyra olika mappningstyper:

One-To-One
One-To-Many
Many-To-One
Many-To-Many

One To One är när det finns endast en möjlighet till länkning mellan två tabellers rader. Till exempel kan ett bibliotek bara låna ut en bok till en person åt gången.

Den "ägande" sidan har annotationen @One-To-One() samt annotationen @JoinColumn() medans barn tabellen använder sig av @One-To-One(mappedBy).

@One-To-Many innebär att förälder klassen sköter all kommunikation med barnklassen. Detta är en *Unidirectional* länkning. Nackdelen är att man skapar en JoinTable utan att veta om detta. Istället för två tabeller har man tre.

Om man använder sig av @JoinColumn och orphanRemoval kommer man runt denna nackdel.

Annars kan man välja en *BiDirectional* länkning istället. Till skillnad mot UniDirection som endast kan gå en väg, kan man fråga från båda håll. Fördelen med detta är att man behöver inte göra dubbla frågor, säkerställa att ens EntityManager inte stängs innan man hunnit fråga klart.

BiDirectional är normalt sätt bättre att använda, men den har en stor nackdel. Säkerheten blir något mindre eftersom man kan komma åt förälder tabellen i dåligt skrivna frågor och råka visa för mycket till en användare.

Alltsom oftast behöver man till sist en @Many-To-Many koppling.

Fördelarna med ManyToMany är många, men de har extrema oönskade fallgropar med.

Första fallgropen är att man använder sig av en List när man hämtar in data. List kan som bekant ha dubletter, och när man tar bort data från en List, tar den bort från alla associerade tabeller innan den (eventuellt) sätter in ny data.

Använder man Set istället, håller man sig enbart till den tabell man utför operationen på.

När man lägger till eller tar bort data, behöver man ha hjälp metoder i klasserna som är länkade. AddEntity, removeEntity är konventionen, men inget krav. När man nu gör en persist eller remove så görs det på båda tabeller vilket underlättar för JPA och inte gör så man hamnar i en loop där datan kan upprepas X antal gånger.

Den absolut största fallgropen är hur man hämtar datan från databasen.

FetchType.EAGER vilket är att man hämtar alla länkade tabeller när föräldrer tabellen tillfrågas. Problemet med detta är att man vid flera lager med länkningar kan tvinga JPA att göra flera frågor som inte behövs. Ibland är detta oundvikligt, men ska undvikas I största möjliga mån.

FetchType.LAZY är givetvis raka motsatsen, hämta länkade datan först vid frågan. ManyToMany är by default i Lazy och bör så förbli. Det finns i extrema fall där en ManyToMany måste vara EAGER, men då förlorar man prestanda.

I Lazy byggda relationer måste man då låta Query:n använda sig av "fetching" medans frågan ställs. Likt vanlig SQL så byggs SELECT ut med JOIN FETCH. Man kan också använda sig av: *@NamedEntityGraph* eller *@EntityGraph*.

Då skriver man in *attributeNodes* i sin Entity klass tillsammans med *subgraph* och lägger till en *orm.xml* där man registrerar sina Graphs.

I transaktionslagret använder sig EntityManagern av .getEntityGraph och sen .setHint() i frågan.

Den sista fallgropen man kan stöta på är om man inte kaskadar sig igenom tabellerna rätt. Det finns .ALL, .PERSIST, .MERGE, .REMOVE, .REFRESH, .DETACH.

I en M-T-M skall man aldrig använda .REMOVE. I bästa fal får man bara lite längre query tid, men I regel tar man bort mer data än man tänkte.

En .ALL innehåller samtliga kaskadtyper så använd rätt typ så inga oönskade deletes springer igenom databasen.

.PERSIST låter förälder klassen persistera till sin barn klass i samma transaktion.

.MERGE tar ut ett objekt från databasen och kopierar in den nya datan på objektet som hämtats ut.

.DETACH tar bort objektet från *Persistence Context*. Inget händer på databasen så länge man inte gjort en *.flush()*.

.REFRESH om man väljer göra en omladdning via entitymanagern på ett objekt, så görs det på alla länkade tabellrader.