Projet de Base de Donnée

DAVID Aurélien, RAMOS Adrien

Projet : Développement d'un système de réservation

SOMMAIRE

Introduction

En cette troisième année de l'EFREI, le cours de Base de données est venu en complément de l'enseignement en programmation Java. Cela a impliqué de nouvelles attentes: changer sa façon de concevoir un programme informatique, celui-ci passe d'abord par une phase d'analyse de la représentation des données.

Pour ce projet: il s'agit de réaliser un gestionnaire de réservation, ce qui est assez conséquent. Il nous a donc fallu étudier longuement la façon dont nous allions aborder le gestionnaire, le concevoir et enfin l'optimiser.

Schéma conceptuel

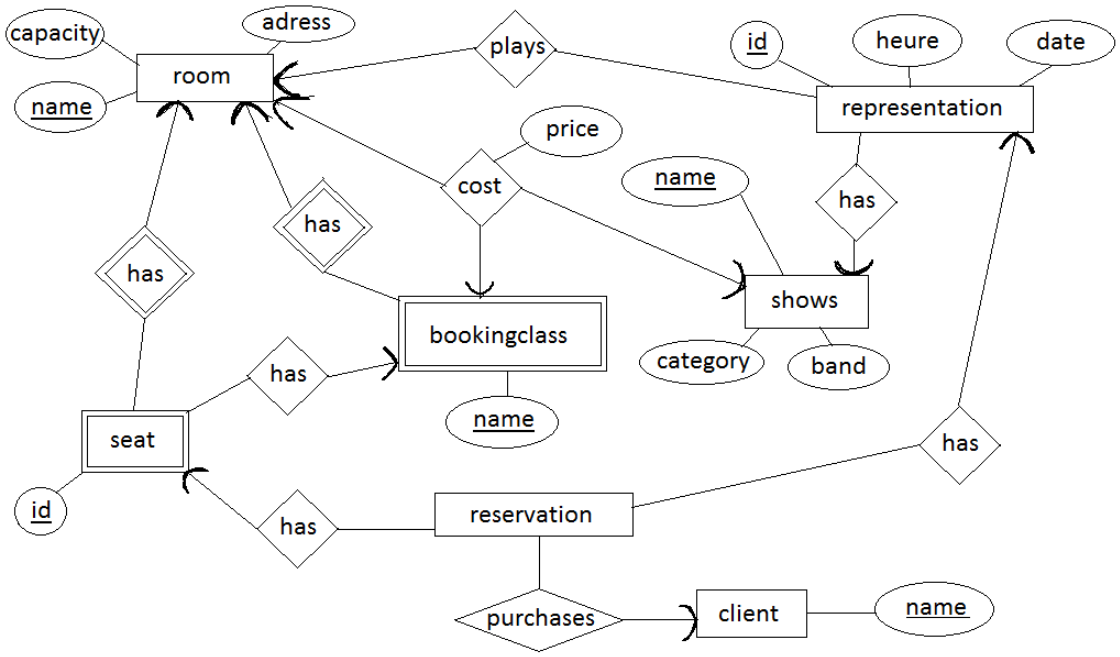

Schéma relationnel

ROOMS(**RNAME**, ADRESS, CAPACITY)

SHOWS(**SNAME**, LENGTH, CATEGORY, BAND)

CLIENTS(**CNAME**)

REPRESENTATION(**RID**, #SNAME, #RNAME, STARTDATE, STARTTIME)

RID, représente l'identifiant d'une représentation, nous avons préféré cela à l'utilisation d'une clé primaire de type SNAME,RNAME,STARTDATE pour plusieurs raisons:

- Plusieurs représentations peuvent être effectuées à la même date. (Exemple une le matin, une le soir)
- Optimisation en effet une comparaison lors des jointures avec un simple entier est plus rapide que de comparer les 3 champs, or c'est une table qui est utilisée dans la quasi totalité des requêtes !

Ce champ n'a pas à être rempli par l'administrateur puisqu'il ne signifie pas grand chose, il est donc auto incrémenté.

BOOKINGCLASSES**(#RNAME,BNAME**)

SEATS**(#RNAME, SID**, #BNAME)

Nous considérons que pour une salle de 100 places les places numérotées de 1 à 20 peuvent avoir le libellé "1ère classe" et les places numérotées de 21 à 100 le libellé "2nd classe".

Pour basculer sur des places numérotées en fonction du libellé (0 à 20 pour la "1ère classe" et 0 à 80 pour la "2nd classe", il suffit d'ajouter BNAME à la clé primaire.

COST**(#RNAME, #SNAME, #BNAME**, PRICE)

RESERVATIONS(**#SID, #RNAME, #RID**, #CNAME)

Le script **creabase.sql** comprend la création des tables avec les types associés mais également des commentaires informant de l'utilité du champ ainsi que lorsque cela nous a paru pertinent, un exemple.

Contraintes d'intégrités.

Le script **contraintes.sql** contient l'ensemble des contraintes d'intégrités du projet, pour plus de clarté celui-ci a été divisé en plusieurs parties en respectant la convention de nommage suivante:

- Clé primaires: **NomDeLaTable_PK**
- Clé étrangère: **NomDeLaTable_FK_NomDeLaClé**
- Contraintes de type check: **NomDeLaTable_CHK_NomDeAttribut**
- Contraintes de type trigger: **NomDeLaTable_TRG_NomDeAttribut ou Evenement**

Contraintes de validation obligatoire

Ce sont toutes les données qui doivent absolument avoir une valeur, on y retrouve l'ensemble des clés primaires et étrangère mais également les données suivantes:

- L'adresse d'une salle, à priori lors de l'ajout d'une salle son adresse est forcément connue.
- La troupe d'un spectacle est également obligatoire lors de la création d'un spectacle.
- Lors de l'ajout d'une représentation d'un spectacle la date et l'heure sont également nécessaire.
- Evidemment lors de l'ajout du cout d'un spectacle dans une salle et une classe tarifaire donnée, le cout est obligatoire.

D'autres données qui peuvent paraitre obligatoire seront définies avec des valeurs par défaut.

Contraintes de valeur par défaut

- En imaginant qu'une salle puisse être en construction ou rénovation, lors de son ajout la capacité a pour valeur pas défaut 0, laissant ainsi à l'administrateur la possibilité de la mettre à jour lorsque les travaux seront terminés.

Contraintes de domaine

Les contraintes telles que "*la classe de tarif d'une place doit être une de celles proposées par la salle correspondante*" de même que "*une salle ne peut pas vendre de réservation pour un spectacle qu'elle ne joue pas*" sont déjà satisfaites grâce à notre schéma.

Cependant nous avons ajouté d'autres contraintes afin d'assurer la fiabilité des données:

Contraintes d'attributs:

- Le prix d'une représentation est un réel avec 2 décimales.

Contraintes de type Check:

- Le prix d'une représentation est forcément positif ou nul.
- La capacité d'une salle est positive ou nulle.
- La durée d'un spectacle est forcément positive.
- L'adresse d'une salle contient forcément un nombre (*en commentaire car la version utilisée d'Oracle ne gère pas les expressions régulières*).

Contraintes de type déclencheur (Trigger):

- Le nombre de sièges d'une salle est forcément égal ou inférieur à sa capacité et par conséquent, le nombre de places réservées ne peut excéder la capacité d'une salle.
- Une séance ne peut pas proposer un prix pour une classe tarifaire qui n'est pas disponible pour une salle donnée.
- Un spectacle ne peut être joué simultanément (même date, même heure) dans deux salles.
- Une salle ne peut pas jouer simultanément deux spectacles (même date, même heure).

Contraintes d'intégrité référentielle

Contraintes de type déclencheur (Trigger):

- Lorsqu'un spectacle est supprimé, toutes les représentations postérieures à la date de suppression sont effacées.
- Lors de la suppression d'une représentation, toutes les réservations sont supprimées.

Implémentation des stratégies de consultations

Nous utilisons une transaction de type READ_COMMITED afin que les utilisateurs lisent uniquement les données qui ont été validées.

Stratégie instable

Lorsqu'un utilisateur consulte les places celles-ci sont indiquées comme libre même si un autre utilisateur les consultent ou est en train de commencer à réserver. Lorsque le client effectue la réservation, les places tentent d'être réservées une à une dans la base de données. Si tout se passe bien les données sont validées par un COMMIT. En revanche si on tente de réserver une place qui a déjà été réservé par un autre utilisateur (et donc commité) alors la réservation échoue et on effectue un ROLLBACK pour annuler les éventuelles réservations qui n'ont pas été validées. Le client doit alors lister à nouveau les places disponibles pour effectuer une réservation.

Stratégie stable

Lorsqu'un utilisateur consulte les places disponibles alors toutes les places libres lui sont temporairement réservées. On effectue un commit pour communiquer aux autres utilisateurs les réservations, ceux-ci ne voient donc aucune place de disponible.

Lorsque l'utilisateur a choisi ses places, il ne nous reste plus qu'à supprimer les réservations des places qui ont été effectuées en trop et à valider la transaction afin de permettre aux autres utilisateurs de réserver à leur tour.

La méthode listerPlace nécessitait d'envoyer toutes les places et donc de toutes les réserver, en implémentation réelle on s'imagine bien qu'il serait plus intéressant de limiter le nombre de réservations, par exemple à 10 afin de réserver seulement 10 places par utilisateur, permettant à plusieurs utilisateurs de réserver simultanément.

Mesure de performances

Nous avons réalisé le programme de test en utilisation plusieurs Threads dans le fichier Main.java.

Pour effectuer nos tests nous avons peuplé la base de 5 spectacles, chacun ayant entre 1 et 2 représentations.

Différentes salles ont été créées:

- Une grande salle de 20 places.
- Deux petites salles de 8 places.
- Une salle moyenne de 14 places.

Les classes tarifaires n'influent pas le tests, les places sont choisies aléatoirement.

Nous utilisons 3 variables: le temps moyen avant que la réservation soit complètement terminée (le client se déconnecte), le nombre d'échec dans le sens où un client réserve mais voit sa réservation annulée et le nombre de fois qu'il ne voit aucune place de disponibles. Un client qui voit 5 fois de suite aucune place de disponible abandonne la réservation.

8 clients se connectent simultanément à la base de données, les temps d'attente de ces clients entre les différentes étapes sont ceux proposés par le sujet.

Nous lançons 3 simulations consécutives (aucune réservation préalable dans la base, puis la base est de plus en plus complète) au delà des 3 simulations les résultats ne sont plus intéressants car les clients abandonnent souvent: leurs choix de spectacles étant complets.

Résultats méthode stable

Pour la méthode stable, le nombre d'échec est toujours égal à 0.

1. 46 secondes d'attente moyen, 0.5 erreur "aucune place de disponibles" / client.
2. Le temps d'attente est généralement légèrement supérieur mais le nombre d'erreur a doublé: 1 erreur / client.
3. Pour la troisième simulation consécutive le temps d'attente moyen est de 65 secondes et le nombre d'erreurs équivalent.

Au total 44 places sur 48 ont été réservé, 2 clients ont abandonné (la seule réservation était complète) et 0 échec de réservation.

Résultats méthode instable

Les résultats sont plus délicats à analyser car très aléatoires en fonction des choix (il arrive d'avoir 0% d'échecs)

La tendance globale est la suivante:

1. 40 secondes d'attente moyen, 0.125 échecs.
2. 47 secondes d'attente moyen, 0.25 échecs.
3. 61 secondes d'attente moyen, 0.625 échecs

C'est logique plus nombre de place est restreint plus le risque de conflit est élevé.

Interprétation

Le critère de sélection est la satisfaction du client, pour les deux stratégies on observe un temps moyen relativement équivalent bien qu'il soit régulier pour la méthode stable et très variable pour la méthode instable. Le critère qui est essentiel est alors l'échec et le nombre d'erreurs.

On imagine qu'un client qui réserve sera très insatisfait si on lui propose des places et que finalement il ne peut pas réserver celles-ci. En revanche un client sera un peu moins insatisfait de lire "aucune places disponibles pour le moment, veuillez actualiser...", bien que celui-ci puisse abandonner alors que des places pourraient être encore disponibles.

C'est pourquoi suite à ces tests nous allons privilégier la méthode stable.

Conclusion

Pour projet de Base de données, nous avons eu l'occasion de découvrir réellement la complexité de ce domaine, bien plus que nos précédents projets qui se contentaient de persister simplement des données.

Il s'agit aussi de s'assurer que les données soient conformes via des contraintes, ce qui s'avère utile lorsque l'on travail avec plusieurs programmeurs mais également sur le long terme pour assurer le fonctionnement du logiciel.

De plus, cela nous a permis de prendre du recul sur notre utilisation des ORM, Doctrine par exemple, puisque nous utilisions du "Lazy Loading", sans le savoir. Désormais, nous seront capable d'améliorer nos performances.

La partie gestion de plusieurs utilisateurs s'est avérée très intéressante et bien plus complexe qu'à première vue, ce projet nous a permis une première approche mais il va nous falloir approfondir encore, puisque nous sommes intéressé de près au fonctionnement des jeux en ligne massivement multi-joueurs.

Annexe

Sources

- http://www.swila.be/files/bd/oracle-trigger.pdf
- http://sqlpro.developpez.com/cours/sqlaz/fondements/
- Base de Données, de Georges Gardarin.