ENSSAT
LANNION

Projet

Bases de données

Création d'une base de données pour une agence de voyage

Colin Leverger
Andreas Ducluzeau

Informatique, Multimédia et Réseaux
Promotion 2017

CONTACT : FirstLetterOfSurname SixFirstLettersOfLastName w/o accents [at] enssat [dot] fr

Destinataire : Olivier SOUFFLET

28 juin 2015

Table des matières

Introduction

Dans le cadre du module Bases de Données dispensé à l'ENSSAT formation Informatique, Multimédia et Réseaux, nous avons été amenés à travailler sur la conception d'un système pour une agence de voyages fictive. Nous sommes partis du cahier des charges fourni et avons nous même spécifié et conçu le système from scratch. Il était explicitement demandé de "mettre de l'intelligence" au sein de ce système. Autrement dit, un bon nombre de vérifications diverses et variées doivent être effectuées dans la base elle-même avant certaines opérations. Avoir une base conçue comme telle permet de gagner du temps dans le développement de l'interface graphique : en effet, il suffit d'utiliser les fonctions fournies par la base de données pour toute la gestion. L'interface graphique peut donc être développée dans plusieurs langages, et disposera toujours des mêmes mécanismes (sans avoir à les retravailler pour chaque langage...).

Nous allons tout d'abord présenter rapidement les exigences du client ainsi que le cahier des charges. Après ce rappel, nous allons expliciter la démarche suivie dans la conception, en fournissant évidemment les schémas associés : MCD & MLD. Nous allons ensuite vous parler des différents mécanismes utilisés lors du développement (Triggers, procédures stockées, etc.). Nous allons finir par une conclusion, et parler des limites de la conception utilisée.

1. Présentation du sujet

Le client est ici une agence de voyages fictive. Concrètement, cette agence propose des circuits touristiques, composés d'une ou plusieurs journées de voyage. Les clients peuvent réserver ces circuits selon plusieurs critères : nombre de places disponibles, paiement de la réservation, etc.

Pour pouvoir simuler une utilisation de la base, nous avons instancié un panel de test ; il s'agit de créer plusieurs instances de chaque classe et de vérifier les interactions entre celles-ci. Afin d'avoir un panel de test cohérent, il était important de penser à valider chacun des cas prévus (réservation impossible car plus de place dans le circuit, réservation impossible car le client n'a pas payé les 10 % de réservation, etc.) Pour pouvoir utiliser et exploiter les données de manière simplifiée, il est aussi demandé de créer des vues et des procédures stockées pour effectuer différentes manipulations : affichage de circuits en fonction de paramètres donnés par le client, création d'utilisateurs...

2. Développement technique

2.1 Les relations & les modèles

Nous avons choisi pour ce projet de scinder notre modèle en *six* relations :

- Client_ : Un client peut faire une ou plusieurs réservations et est identifié par un ID. L'ID va s'auto incrémenter dans la base.
- Ville : Une ville est traversée lors d'un déplacement journalier et s'identifie avec son nom. Le nom d'une ville étant unique, c'est une clef primaire.
- Circuit : Chaque circuit possède plusieurs réservations et plusieurs déplacements journaliers. Ils sont identifiés par leur ID, qui s'auto incrémentent.
- Déplacement : Un déplacement journalier appartient à un circuit et possède une ville de départ et une ville d'arrivée.
- Reservation : Une réservation est effectuée par un client et concerne un circuit ; elle se valide via deux paiements. Chaque réservation a un ID. Le client doit tout d'abord effectuer une réservation, puis effectuer les deux paiements. Le premier paiement doit être égal à 10 % de la somme totale du circuit, le second à 90 %. Il est possible dans notre système pour le client d'effectuer un seul et unique paiement de 100 % de la somme. Il est aussi possible pour un client de réserver plusieurs places.
- Paiement : Le paiement se fait par un client et concerne une réservation. Il y a maximum deux paiements, qui se différencient par leur ID (AI). À noter : cette table est configurée "Delete on cascade", cela signifie que si une réservation est supprimée, tous les paiements qui sont associés a cette réservation le seront aussi. Cette notion sera utile dans le cas d'un nettoyage de base.

Les tables dont les tuples sont identifiés par des ID sont auto-incrémentées. C'est-à-dire que l'ID augmente à chaque nouveau tuple inséré dans une table.

Le modèle entité/association suivant (fig 2.1) décrit les différentes relations entre ces six tables :

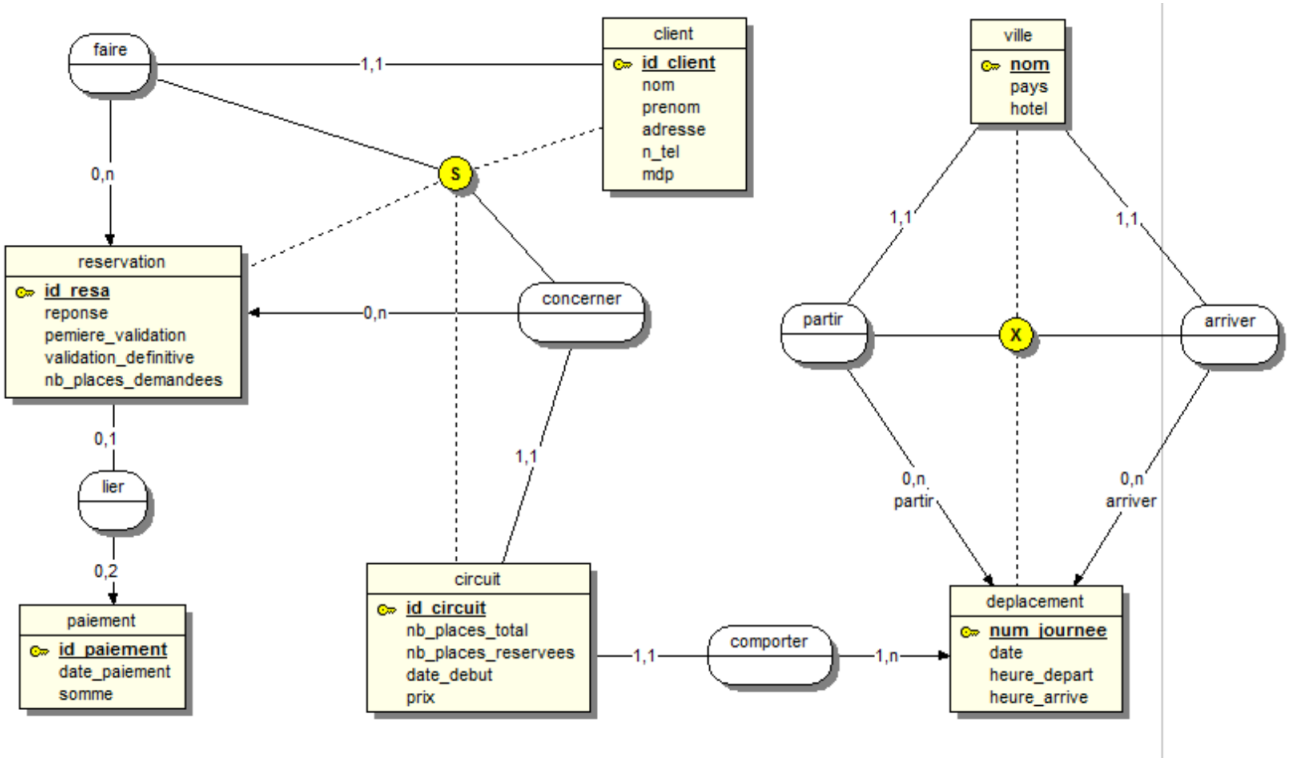

Figure 2.1 – Modele MCD

Les "Foreign Key" nécessaires au bon fonctionnement de la base en SQL ne sont pas présentes dans le modèle MCD, et c'est normal. C'est en effet le schéma relationnel qui les représente. (figure 2.2)

2.2 Les procédures stockées (création de données)

Il était nécessaire de créer des procédures permettant d'instancier des tuples pour chaque relation de notre schéma. Chacune de ces procédures permet de remplir une table avec un tuple. L'une de ces procédures utilise le système de "transaction" ; en effet, il était important de considérer que la base pouvait être sollicitée par plusieurs clients en même temps. Les transactions permettent donc de sécuriser et de locker une procédure, pour éviter que deux personnes l'utilisent en même temps. S'il n'y a pas de problème, la transaction va se solder par un "commit" et les modifications seront effectuées en base. A contrario, s'il y a un problème il y aura un "rollback" et les modifications ne seront pas effectuées. La procédure qui utilise la transaction est la procédure de réservation : si deux clients veulent réserver simultanément et qu'il ne reste plus qu'une place au circuit, le deuxième client ne doit pas pouvoir réserver...

Certaines procédures ne demandent pas à l'utilisateur tous les paramètres nécessaires à la création d'un tuple dans la table voulue. À la place, elles les initialisent. Par exemple pour la création d'une nouvelle réservation, les deux validations (première et finale) sont initialisées à "nul", car

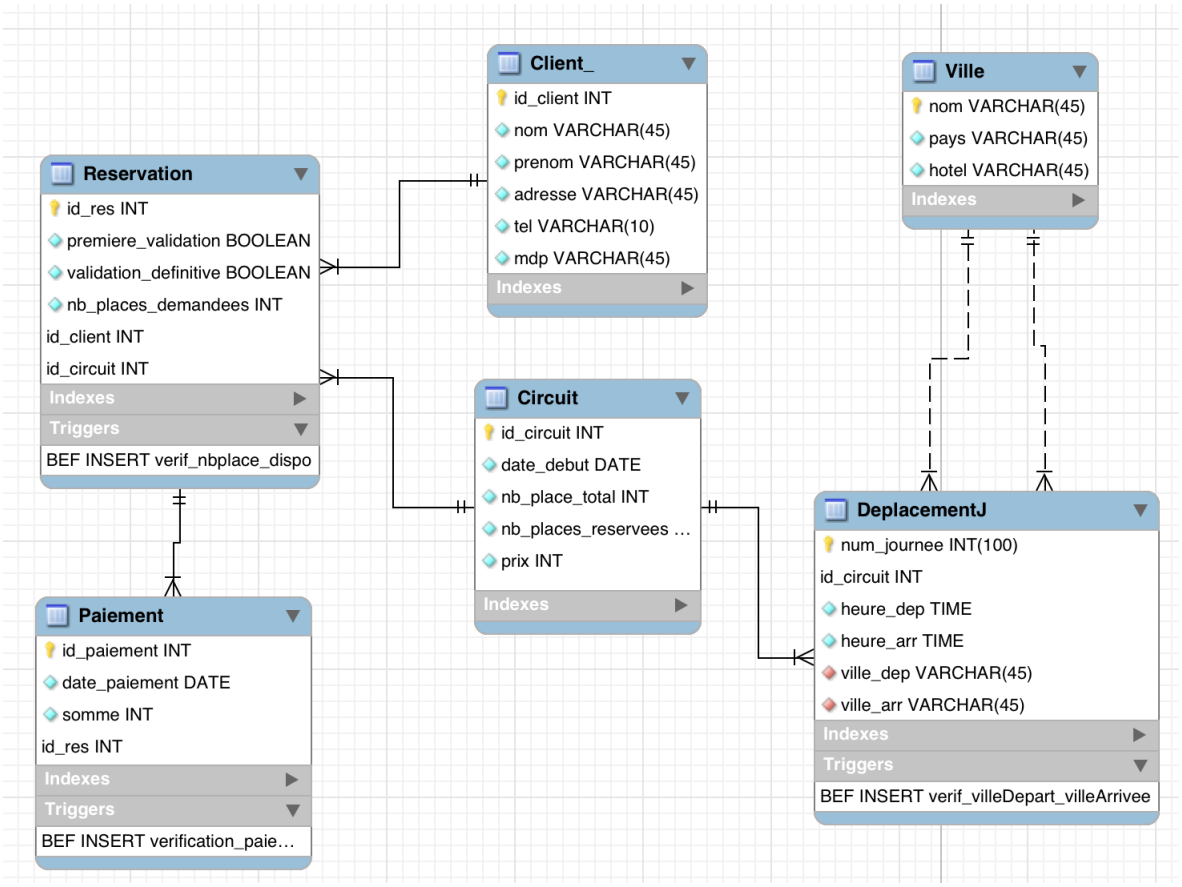

Figure 2.2 – Schema relationnel de notre base

aucun paiement n'a été effectué à cet instant. C'est également le cas lors d'un paiement, où la date est générée avec la fonction "now()", qui récupère la date actuelle. Pour nos tests, il est possible d'instancier un circuit déjà plein, mais c'est évidemment uniquement à des fins de débug (instancier un circuit plein n'a pas vraiment de sens dans la vraie vie.)

2.3 Les triggers

Dans notre modèle, certaines actions sont automatiques, comme la validation d'une réservation par exemple. Pour cela, nous avons utilisé des triggers. Ils sont au nombre de trois :

- Le premier trigger se déclenche juste avant la création d'un nouveau déplacement journalier. Il vérifie que la ville de départ et la ville d'arrivée sont différentes l'une de l'autre. Il contrôle également la cohérence du trajet, car la ville de départ d'un déplacement doit être la même que la ville d'arrivée du déplacement précédent. Dans le cas contraire, un message d'erreur explicite est redirigé vers l'utilisateur.
- Le deuxième trigger s'amorce quant à lui lors d'une nouvelle réservation. Lorsque le nombre de places vacantes n'est pas suffisant pour répondre à la demande du client, un message lui indique que le circuit est complet.

- Le troisième trigger se lance avant un paiement. Il vérifie tout d'abord si le paiement est effectué plus d'un mois avant le début du circuit. Ensuite, une vérification est effectuée pour savoir de quel type de paiement il s'agit, soit le première soit le deuxième.
  - Dans le premier cas, la réservation se voit validée une première fois lorsque le paiement est supérieur ou égale à 10 % du prix. C'est à ce moment-là que l'utilisateur voit sa/ses places réservées dans le circuit (nb_places_reservees est incrémenté de nb_places_demandees).
  - Lorsque c'est le deuxième versement, la validation devient définitive si le paiement est complet, donc s'il respecte 90 % du prix. Dans le cas contraire, des messages d'erreur sont renvoyés au client.

Nous avons fait le choix de tout traiter "BEFORE INSERT" dans le cas des triggers pour la table "Paiement" ; nous sommes partis du postulat que si un paiement ne satisfait pas les critères, il n'était pas la peine de mettre à jour certains champs (nombre de place décomptée dans la table "Circuit" par exemple.) Il aurait pu être intéressant de faire un trigger "BEFORE INSERT" qui vérifie que les paiements satisfont les exigences de la date et des 10 %/90 % et un trigger "AFTER INSERT" qui met à jour le nombre de places prises dans "Reservation".

2.4 Les vues

Afin de fournir quelques statistiques à l'utilisateur de la base, nous avons créé quatre vues :

- La première affiche différentes infos pour un circuit. On y retrouve le numéro du circuit, son prix, sa date de départ, son nombre d'étapes, le nombre de réservations confirmées, le nombre de réservations définitives, le nombre de places totales, le nombre de places confirmées et le nombre de places définitives.
- La deuxième vue permet de visualiser pour un circuit son numéro, son prix, sa date de départ, le nombre de villes traversées, le nombre de places totales, le nombre de places confirmées et le nombre de places définitives.
- La troisième vue présente pour chaque année le nombre de places offertes, le nombre de places réservées et le chiffre d'affaires réalisé.
- La quatrième et dernière vue affiche les statistiques d'un client. Avec son numéro, son nom, l'année et le chiffre d'affaires généré.

Pour les deux dernières vues, les chiffres d'affaires réalisé par un circuit et généré par un client sont faux. En effet, il y a multiplication du premier prix, trouvé par la fonction (100 ici), avec le nombre de places définitives. Il n'y a pas de relation entre les prix et leur circuit. Il faudrait donc pour chaque circuit multiplier son prix avec le nombre de places réservées, et faire la somme de tous ces résultats. Nous n'avons malheureusement pas réussi à corriger ce problème par manque d'expérience et de maitrise des requetes SQL.

2.5 Nettoyage de la base

La procédure de nettoyage de la base n'est pas à 100 % fonctionnelle dans notre système. Actuellement, elle vérifie la différence entre la date d'aujourd'hui et la date de toutes les réservations non définitives, et si celles-ci sont inférieures à 31 jours elle les supprime purement et simplement de la base "Reservation". La suppression des paiements liés suis avec la close "DELETE ON CASCADE" de la table "Paiement".

La fonctionnalité manquante pour avoir une fonction de nettoyage cohérente est la *remise en jeu des places des réservations supprimées*. Pour ce faire, nous aurions pu utiliser une boucle qui vérifie chaque réservation, et qui en fonction de son statut incrémente le nombre total de places du circuit concerné avec le nombre de places de la réservation (si la réservation est non définitive un mois avant le départ). La procédure de suppression de réservation aurait alors pu être effectuée à postériori. Nous n'avons pas trouvé de fonctions pouvant effectuer cette boucle et avons malheureusement terminé sur ce point.

2.6 Et les tests ?

Grâce à nos procédures, nous avons créé un panel de test afin de vérifier le bon fonctionnement de nos triggers et de nos vues. Ce panel vous permettra aussi de tester nos fonctions. Nous allons tout d'abord à remplir les tables Client_ et Ville, avec des exemples de clients/villes classiques (le fameux "Jean Dupont","1 rue des mimosas"...). Nous créons ensuite *cinq* circuits :

- Un premier commençant dans moins d'un mois (donc impossible de payer)
- Un second commençant exactement dans un mois (possibilité de s'inscrire, mais il faut se dépêcher !)
- Un troisième avec un nombre de places complet (donc impossible de s'inscrire)
- Et deux autres circuits libres, où il est possible de s'inscrire

Vient ensuite la création de déplacements pour nos circuits. Ici, il y a deux cas à vérifier. Il faut tout d'abord s'assurer qu'un déplacement avec la même ville de départ et d'arrivée ne peut pas être créé. Ensuite, la ville de départ et la ville d'arrivée de la journée précédente doivent être les mêmes. Nous avons donc deux déplacements qui ne seront pas créés (pour vérifier le trigger) et dix autres répartis entre les différents circuits.

Une fois ces déplacements créés, nous nous occupons des réservations. Une de ces réservations sera impossible par manque de place libre dans le circuit. Les autres se répartissent entre les différents clients et circuits.

La création des paiements vient en dernier. Ce sont eux qui utilisent le trigger le plus complet de notre modele. Il faut donc vérifier la date du paiement, qui doit être plus d'un mois avant le début du circuit. Le premier paiement doit également être égale ou supérieur à 10 % du prix du circuit, et le deuxième 90 %. Pour ce dernier, la réservation doit être validée définitivement.

Les vues sont toutes appelées à la fin du fichier de test commenté.

3. Conclusion

Ce projet nous a permis d'appréhender la conception d'une base de données de A jusqu'à Z, et nous a permis de nous familiariser avec la conception. Avoir une conception solide et bien pensée est un prérequis essentiel afin de facilité l'utilisation et l'exploitation les données facilement. Nous nous en sommes rendu compte au fil du développement. Nous avons en effet modifié notre schéma au milieu de notre développement, car trop d'attributs étaient redondants et inutiles. Ce refactoring nécessaire nous a permis de rendre notre système plus cohérent.

Il n'existe pas qu'une solution pour un problème donné ; notre solution nous a semblé bonne, mais plusieurs interprétations étaient possibles pour la phrase suivante : "Procédure de réservation qui permet à un utilisateur de réserver un circuit en versant un acompte de 10 %" . La réservation aurait donc pu être directement liée à un paiement, et ce dès sa création. Nous n'avons pas fait ce choix, et avons porté notre confiance sur la fonction de nettoyage de base demandée pour assurer la cohérence.