PROJET FUN PARK

Projet de Base de Données
03 Février 2023

COUVERT Nicolas
TORRES Andy
HÛ Quentin
HERLEM Quentin

Introduction

Pour ce projet nous sommes partis sur l'idée de créer une base de données pour un parc d'attraction. Celle-ci contiendra les informations sur les employés du parc, les attractions présentes, les restrictions de taille et d'âge ainsi que les tarifs pour chaque attraction en fonction du type de client (On considère ici qu'il faille payer pour chaque attraction).

Pour cela nous avons tout d'abord imaginé les différentes dépendances fonctionnelles à un tel système. Pour générer des tuples chacun a pris des dépendances fonctionnelles différentes et imaginé des tuples qui permettent de les illustrer.

Puis nous avons fait en sorte qu'elles soient peu optimisées et mal agencées afin d'exploiter au maximum les différents algorithmes de normalisation. On a utilisé 2 algorithmes qui sont : l'algorithme de Bernstein et l'algorithme de Décomposition, étant un groupe de 4 nous avons fait chaque algorithme 2 fois afin de comparer les résultats.

Déroulement du processus & Distribution des taches :

Nous avons créé la table collectivement puis, pour les 2 algorithmes de normalisations nous avons choisi de répartir le travail de la manière suivante :

Deux personnes ont réalisé chacun de leur côté l'algorithme de Bernstein et deux personnes pour l'algorithme de décomposition.

Par la suite, 2 personnes ont créé les différentes tables sur Oracle. Pendant ce temps, deux autres personnes ont crée les insertions de tuples pour pouvoir remplir les tables de notre base.

Nous avons ensuite réfléchi au type de triggers et fonctions qui pourraient être intéressante pour le projet.

Nous nous sommes donc ensuite réparti la création de celles-ci. Enfin, 2 autres personnes ont créé des views et les autres ont réfléchi et créer des rôles intéressant et utile pour notre base.

Modèle Relationnel de la base de données

Table d'origine :

(idEmploye, nomEmploye, prenomEmploye, idAttraction, nomAttraction, ouvertureAttraction, fermetureAttraction, idClient, catClient, ageClient, tailleClient, prixAttraction, jourSemaine, ageMin, tailleMin, dateDerniereMaintenance, idEmployeDerniereMaintenance)

Exemple de tuples :

idEmploye	nomEmploye	ageMin	idEmployeDerniereMaintenance
E1	Ferrer	6	E4
E2	Duchet	15	E1
E3	Marchand	13	E4
E3	Marchand	13	E4
E4	Leboison	5	E4
E5	Ferrer	6	E3
E5	Ferrer	6	E3
E6	Hugo	3	E5
E6	Hugo	8	E5
E7	Hugo	3	E5
E8	Dupont	12	E8
E8	Dupont	12	E8
E1	Ferrer	12	E8

nomAttraction	ouvertureAttraction	idClient	catClient	prixAttraction
DuckLand	9h30	8	Jeune	4,00 €
WesternTour	9h00	4	Adulte	6,00 €
DuckLand	9h30	4	Adulte	5,00 €
DuckLand	9h30	4	Adulte	4,00 €
Freezomachine	8h15	7	Enfant	4,00 €
Safari	12h45	6	Senior	5,00 €
Safari	12h45	7	Enfant	5,00 €
Luge	8h	5	Senior	3,00 €
Canard	7h	3	Jeune	1,00 €
Luge	8h	2	Jeune	2,50 €
Super8	9h00	9	Jeune	6,00 €
Super8	9h00	10	Senior	8,00 €
Super8	9h30	9	Jeune	6,00 €

jourSemaine	prenomEmploye	tailleMin	dateDerniereMaintenace
Jeudi	Michel	130	20/01/23
Lundi	Fabrice	160	24/01/23
Mardi	Victor	160	20/01/23
Mercredi	Victor	160	20/01/23
Vendredi	Christine	120	23/01/23
Dimanche	Michel	120	18/12/22
Dimanche	Michel	120	18/12/22
Lundi	Paul	100	24/01/23
Mardi	Paul	0	21/01/23
Jeudi	Jade	100	24/01/23
Mardi	Johnny	140	22/01/23
Mardi	Johnny	140	22/01/23
Vendredi	Michel	140	22/01/23

idAttraction	fermetureAtraction	ageClient	tailleClient
A45	18h	20	178
A78	20h	45	181
A45	19h	45	181
A45	19h	45	181
A30	12h30	8	133
A35	19h30	61	165
A35	19h30	8	133
A8	17h30	68	160
A7	9h	15	183
A8	17h30	17	170
A3	18h	20	167
A3	18h	70	160
A3	18h30	20	167

Dépendances fonctionnelles de la table :

1. idEmpl → nomEmpl, prenomEmpl
2. idEmpl → prenomEmpl
3. idAttraction → dateDerniereMaintenance
4. idAttraction → idEmployeDerniereMaintenance
5. idAttraction, jourSemaine → ouvertureAttraction, fermetureAttraction, idAttraction, dateDerniereMaintenance
6. idAttraction → nomAttraction, ageMin, tailleMin
7. idAttraction, jourSemaine → idEmpl
8. idEmpl, jourSemaine → idAttraction
9. idAttraction, catClient, jourSemaine → prixAttraction
10. idClient → ageClient, tailleClient, catClient

Calcul de la clé

LEFT	MID	RIGHT
jourSemaine idClient	idEmpl idAttraction catClient	nomEmpl prenomEmpl ouvertureAttraction fermetureAttraction nomAttraction ageMin tailleMin prixAttraction ageClient tailleClient

Fermeture de la partie de gauche :
{jourSemaine, idClient}+
= jourSemaine, idClient, ageClient, tailleClient, catClient

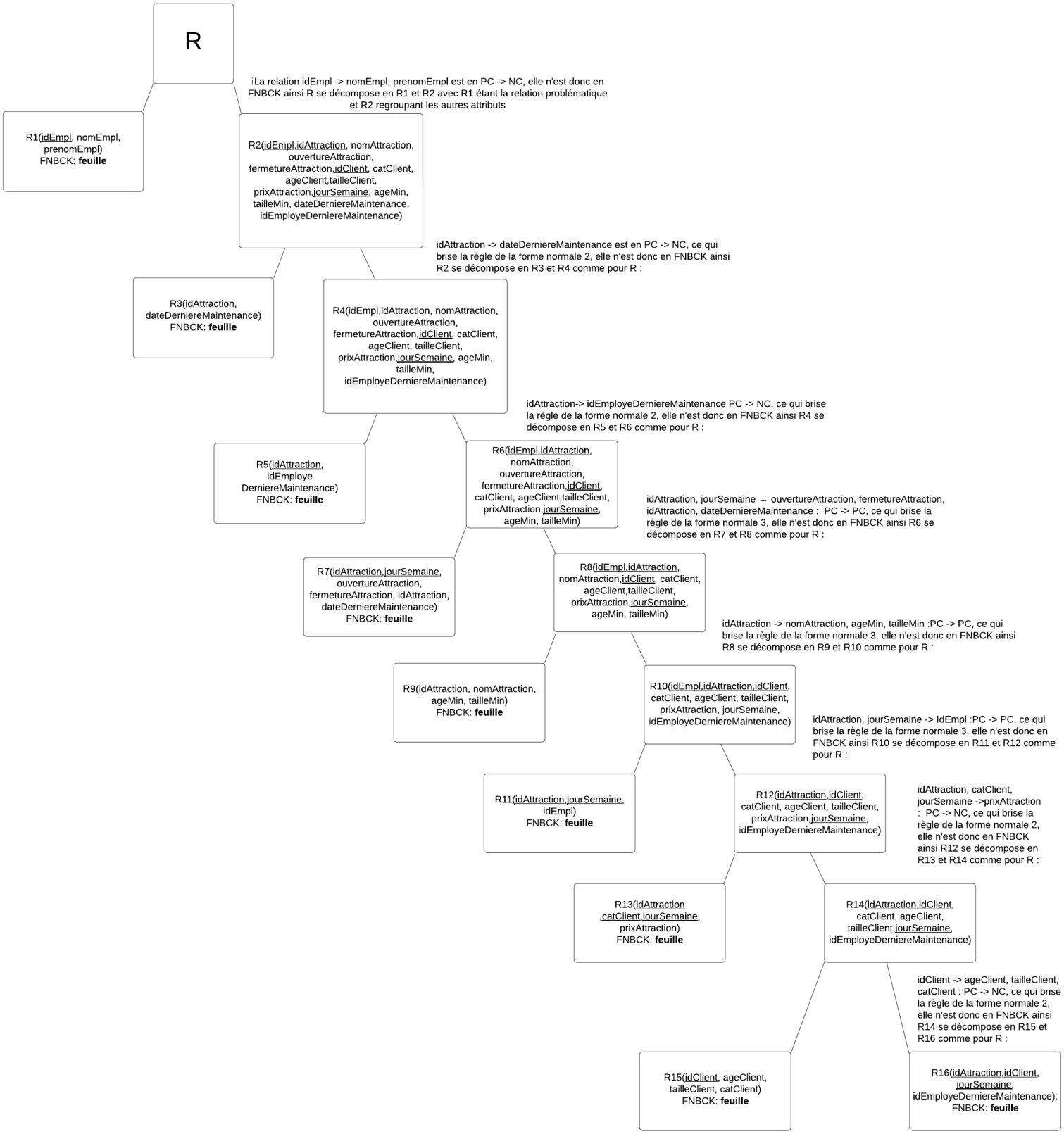
Fermeture de la partie de gauche plus un élément de la partie du milieu
{jourSemaine, idClient, idEmpl}+
= jourSemaine, idClient, idEmpl, ageClient, tailleClient, catClient, idAttraction, prixAttraction, nomAttraction, ageMin, tailleMin, ouvertureAttraction, fermetureAttraction, dateDerniereMaintenance, prenomEmpl, nomEmpl

{jourSemaine, idClient, idAttraction}+
= jourSemaine, idClient, idAttraction, catClient, prixAttraction, tailleClient, ageClient, idEmpl

{jourSemaine, idClient, catClient}+
= jourSemaine, idClient, catClient, ageClient, tailleClient

Donc la clé est
{jourSemaine, idClient, idEmpl}

Normalisation en FNBCK par décomposition



On obtient donc, à partir des feuilles, les relations suivantes:

Ra(idEmpl, nomEmpl, prenomEmpl) df{1}

Rb(idAttraction, dateDerniereMaintenance) df{3}

Rc(idAttraction,idEmployeDerniereMaintenance) df{4}

Rd(idAttraction, jourSemaine,ouvertureAttraction, fermetureAttraction, idAttraction, dateDerniereMaintenance) df{5}

Re(idAttraction, nomAttraction, ageMin, tailleMin) df{6}

Rf(idAttraction, jourSemaine, idEmpl) df{7}

Rg(idAttraction, catClient, jourSemaine, prixAttraction) df{9}

Rh(idClient, ageClient, tailleClient, catClient) df{10}

Ri(idAttraction, idClient, jourSemaine, idEmployeDerniereMaintenance)

Algorithme de Bernstein :

Rappel du fonctionnement de l'algorithme :

Cet algorithme se décompose en 2 étapes

1. On calcule la couverture minimale des dépendances fonctionnelles, c'est-à-dire décomposer les dépendances qui ne sont pas élémentaires, supprimer les attributs superflus du coté gauche de la dépendance et éliminer celles qui sont redondantes. On obtient un nouveau jeu de dépendances fonctionnelles, si celles-ci sont en 3FN on s'arrête.
2. On réunit ensuite les dépendances fonctionnelles qui ont la même partie de droite. Puis on créer une Relation R_i pour chaque DF_i avec comme clé de chaque relation la partie gauche de la DF.
3. On vérifie que des relations ne sont pas incluses dans d'autres, si 'est le cas on les fusionne. Et enfin on s'assure qu'au moins une relation contient l'ensemble des clés de la table relation originale. Si ce n'est pas le cas on créer une relation en plus qui les contient.

Etape 1

Pour chaque Dépendance Fonctionnelle, on calcule la fermeture du côté gauche sans prendre en compte la DF en question. Si des éléments de droite sont présents dans cette fermeture ainsi calculée alors on peut supprimer cet élément de la DF. S'il n'y a plus d'éléments à droite alors la dépendance fonctionnelle est supprimée.

(1)

nomEmpl $\notin$ {idEmpl}+

prenomEmpl $\in$ {idEmpl}+ donc on le supprime de la règle (1)

(2)

prenomEmpl$\notin$ {idEmpl}+

(3)
dateDerniereMaintenance ∉ {idAttraction}+
(4)
idEmployeDerniereMaintenance ∉ {idAttraction,dateDerniereMaintenance}+
(5)
ouvertureAttraction ∉ {idAttraction, jourSemaine}+
fermetureAttraction ∉ {idAttraction, jourSemaine}+
idAttraction est présent dans la partie gauche de la règle donc on le supprime de la partie droite
dateDerniereMaintenance ∈ {idAttraction, jourSemaine}+ donc on le supprime de la règle (5)
(6)
nomAttraction ∉ {idAttraction}+
idEmpl ∉ {idAttraction}+
ageMin ∉ {idAttraction}+
tailleMin ∉ {idAttraction}+
(7)
idEmpl ∈ {idAttractio, jourSemaine}+ donc on le supprime de la règle (on supprime donc la règle (7))
(8)
prixAttraction ∉ {idAttraction, catClient, jourSemaine}+
(10)
ageClient ∉ {idClient}+
tailleClient ∉ {idClient}+
catClient ∉ {idClient}+

On obtient donc les règles suivantes :

(1) idEmpl → nomEmpl
(2) idEmpl → prenomEmpl
(3) idAttraction → dateDerniereMaintenance
(4) idAttraction → idEmployeDerniereMaintenance
(5) idAttraction, jourSemaine → ouvertureAttraction, fermetureAttraction
(6) idAttraction → nomAttraction, ageMin, tailleMin
(7) idAttraction, jourSemaine → idEmpl
(8) idEmpl, jourSemaine → idAttraction
(9) idAttraction, catClient, jourSemaine → prixAttraction
(10)idClient → ageClient, tailleClient, catClient

Etape 2 :

On réunit les règles qui ont la même partie de droite :

(1) idEmpl → nomEmpl, prenomEmpl
(2) idAttraction → dateDerniereMaintenance, idEmployeDerniereMaintenance, nomAttraction, ageMin, tailleMin
(3) idAttraction, jourSemaine → ouvertureAttraction, fermetureAttraction, idEmpl
(4) idEmpl, jourSemaine → idAttraction
(5) idAttraction, catClient, jourSemaine → prixAttraction
(6) idClient → ageClient, tailleClient, catClient

On créer les relations correspondantes

<R1(idEmpl, nomEmpl, prenomEmpl),{(1)}>
<R2(idAttraction, dateDerniereMaintenance, idEmployeDerniereMaintenance, nomAttraction, ageMin, tailleMin),{(2)}>
<R3(idAttraction, jourSemaine, ouvertureAttraction, fermetureAttraction),{(3)}>
<R4(idEmpl, jourSemaine, idAttraction),{(4)}>
<R5(idAttraction, catClient, jourSemaine, prixAttraction),{(5)}>
<R6(idClient, ageClient, tailleClient, catClient),{(6)}>

Etape 3 :

R5 est inclus dans R4 donc on peut écrire

<R1(idEmpl, nomEmpl, prenomEmpl),{(1)}>
<R2(idAttraction, dateDerniereMaintenance, idEmplDerniereMaintenance, nomAttraction, ageMin, tailleMin),{(2)}>
<R3(idAttraction, jourSemaine, idEmpl, ouvertureAttraction, fermetureAttraction),{(3),(4)}>
<R4(idAttraction, catClient, jourSemaine, prixAttraction),{(5)}>
<R5(idClient, ageClient, tailleClient, catClient),{(6)}>

On rajoute R6 qui contient la clé soit

<R6(idClient, jourSemaine, idEmpl),{}>

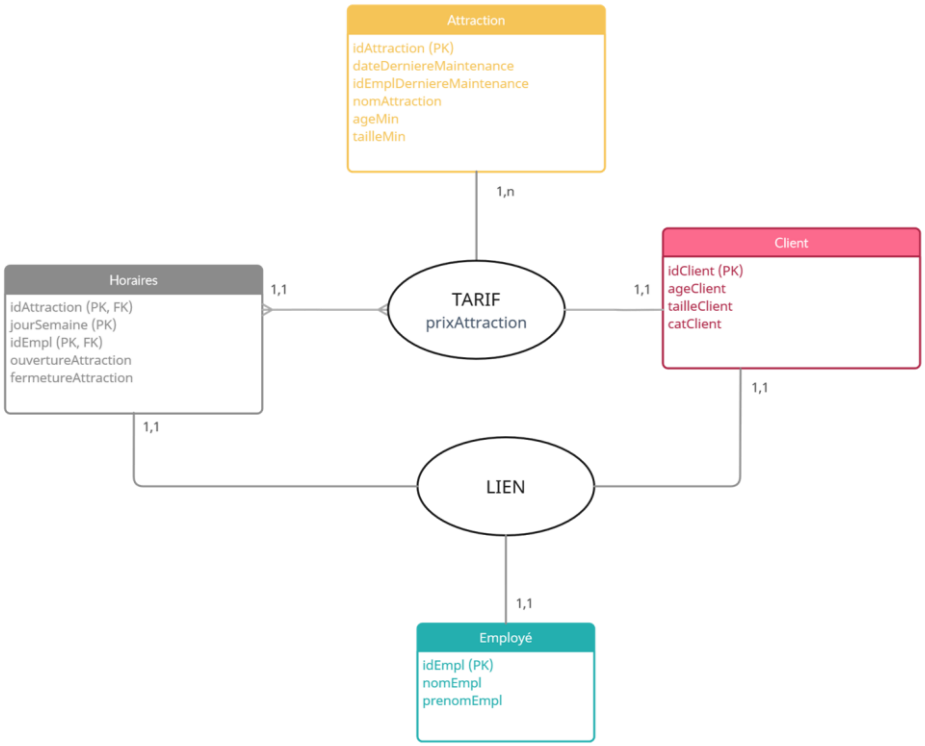
Ainsi on obtient les relations suivantes :

<R1(idEmpl, nomEmpl, prenomEmpl),{(1)}>
<R2(idAttraction, dateDerniereMaintenance, idEmplDerniereMaintenance, nomAttraction, ageMin, tailleMin),{(2)}>
<R3(idAttraction, jourSemaine, idEmpl, ouvertureAttraction, fermetureAttraction),{(3),(4)}>
<R4(idAttraction, catClient, jourSemaine, prixAttraction),{(5)}>
<R5(idClient, ageClient, tailleClient, catClient),{(6)}>
<R6(idClient, jourSemaine, idEmpl),{}>

On vérifie qu'on obtient un ensemble de relations en 3FN et l'algorithme est fini

Représentation schématique de nos nouvelles tables

Notre base sera principalement utilisée pour la consultation, en effet on peut facilement imaginer que le parc ne va pas rajouter d'attraction tous les 3 jours mais va sûrement devoir consulter quotidiennement les dates de maintenance, les tarifs, etc. Ainsi il faut privilégier une table où ces informations sont accessibles rapidement et avec le moins de jointure possible. Pour cela la version de l'algorithme de Bernstein est plus pratique car elle possède moins de table et donc va nécessiter moins de jointure



Par exemple une requête pour obtenir des informations sur une attraction comme l'âge minimum requit, la date de dernière maintenance et l'id de l'employé responsable. Avec les tables obtenues par décomposition il faut faire une jointure entre Rd, Re et Rf. Alors qu'avec les relations ci-dessus il suffit de joindre Attraction avec Horaires. Une requête très classique et rapide comme connaitre l'id de l'employé qui a fait la dernière maintenance d'une attraction ainsi que la date de celle-ci nécessitera une jointure dans le premier cas et pas dans le second. On peut facilement imaginer que l'algorithme de Bernstein offre ici une configuration des relations plus adaptée à notre situation.

Création des tables

Pour créer les tables et l'écriture des contraintes nous avons adopté la stratégie suivante :

Lorsqu'il s'agit de vérifier si la valeur entré est correcte et valide (par exemple un âge positif) on teste cela sous la forme de CONSTRAINT CHECK lors de la création de la table. Lorsqu'il s'agit de vérifier si différents attributs sont cohérents entre eux on teste ceci avec des TRIGGER (par exemple que les catégories correspondent bien à l'âge de la personne).

Concernant la table Client nous avons utilisé une séquence pour l'attribut idClient car beaucoup de clients seront ajoutés régulièrement et ainsi on évite des erreurs humaines qui peuvent produire des doublons. Pour les tables Employes et Attractions cela n'est pas nécessaire au vu du nombre de tuples.

Concernant la table Attraction, nous avons fait le choix de ne pas mettre idEmplDerniereMaintenance en clé étrangère (faisant référence à idEmpl de Employes) car on peut imaginer que l'employé en question ne travaille plus dans le parc.

Fonctions, Vues, Triggers et Index

Fonctions :

La première fonction modifier_prix_attraction() prend 4 paramètres : p_idAttraction, p_catClient, p_jourSemaine et p_nouveauPrix, l'objectif est de modifier le prix d'une attraction pour un jour de la semaine donné et une catégorie de client donnée.

La seconde fonction permet de supprimer une attraction et s'assure qu'elle soit aussi supprimée dans les tables qui lui font références (Horaires et Tarifs). Cette fonction est très utile car elle permet de s'assurer qu'il n'y ait pas d'erreur de référence sans que l'utilisateur doive s'en assurer lui-même.

Triggers :

Comme expliqué précédemment les Triggers permettent la cohérence des valeurs entre les différents attributs d'un même tuple. Pour cela nous avons imaginé 3 triggers, qui permettent de :

- S'assurer que l'horaire de fermeture de l'attraction est bien après l'horaire d'ouverture
- S'assurer que l'âge d'un client correspond à la bonne catégorie associée
- S'assurer que la date de dernière maintenance d'une attraction est bien antérieur ou équivalente à la date d'aujourd'hui

Vues :

Nous avons créé trois vues qui pourraient faciliter l'utilisation de la base. En effet on peut imaginer que certaines requêtes seront effectuées très souvent et des vues peuvent accélérer et simplifier l'exécution de celles-ci.

Ainsi on a :

- Une vue qui affiche les attractions nécessitant une maintenance car aucune maintenance n'a été faite dans les 60 derniers jours
- Une vue qui affiche les horaires d'ouverture et de fermeture de chaque attraction pour chaque jour de la semaine
- Une vue qui affiche le nom de l'attraction et ses informations tarifaires.

Ces vues peuvent être utiles pour les droits d'accès notamment. Par exemple les clients pourraient accéder à la 2nd vue et ainsi connaître les horaires sans connaître l'employé responsable de l'attraction et en ayant le nom de l'attraction plutôt que son id.

Index :

Il est intéressant de créer un index Bitmap sur la catégorie de client car celle-ci prendra très peu de valeurs différentes. Pour le reste le SGBD créera lors de la création de la Table des indexs adéquats.

Rôles

On crée 3 rôles différents : client_classic_role, employe_role et admin_role.

- client_classic_role : Peux insérer des tuples dans la table Clients (par exemple on peux imaginer la création d'un compte sur le site du parc) et lire les vues vue_horaires et vue_attraction.
- employe_role : Un employé peux accéder à la lecture et l'écriture dans les tables Employes et Attractions. Il peut lire toutes les autres tables. Il peut aussi accéder à la vue vue_maintenance.
- admin_role : Il a tous les droits sur toutes les tables

Améliorations possibles et critiques

Dans l'idée d'un agrandissement du parc il serait cohérent d'utiliser des séquences pour idEmpl et idAttractions. En fonction des pratiques et des requêtes récurrentes des employés on peut faire des vues et des fonctions pour répondre à leurs besoins. On peut aussi imaginer des rôles supplémentaires en fonction de la taille et de l'organisation du parc (manager, exposant, vacataires, directeur des ressources humaines, …)

Conclusion

Nous pensons avoir réalisé des tables qui ne font que survoler une utilisation réelle d'une base de données dans un parc. En réalité il est évident qu'il y aura beaucoup de choses à adapter et/ou à rajouter en fonction de la taille du parc, du type d'attractions, des pratiques réelles des utilisateurs, etc. Pour qu'une telle base soit vraiment optimisée et pratique, il faudrait qu'elle soit faite en concertation direct avec ceux qui l'utiliseront et réponde à leurs besoins à l'aide de nouvelles fonctions, des nouveaux rôles, etc. Néanmoins nous fournissons ici une proposition de structure générale et fonctionnelle avec quelques exemples d'utilisations simples reflétant qui pourrait être facilement implémentée.