Chapitre 10

Cohérence des données

Un système de base de données permet d'assurer un contrôle sur l'intégrité des données. Un ensemble de techniques concourent à rendre l'erreur d'encodage, sinon impossible, du moins difficile. Un certain nombre de règles peuvent s'appliquer pour vérifier que les données correspondent aux faits du réel. Ces règles ne peuvent évidemment pas opérer une vérification des faits. On parvient ainsi :

- à éliminer des données inexistantes : c'est par exemple le contrôle de validité d'une date ou la limitation d'une valeur numérique à un intervalle prévu.
- à déceler l'impossibilité d'une association entre deux tables, par suite de l'inexistence d'un tuple correspondant (par exemple utilisation d'un numéro de client ne correspondant à aucun client connu).
- à vérifier le respect d'une règle complexe respectée dans le contexte de l'entreprise (opération interdite dans certaines circonstances, nécessité d'une opération préalable, vérification quelconque).

Différents moyens vont être employés :

1. des contraintes placées sur les champs ou sur les lignes d'une table opèrent une vérification implicite lors de l'encodage des données (types de données et contraintes diverses) ;
2. des vérifications sur des liaisons entre tables permettent de maintenir l'intégrité référentielle (clés primaires et étrangères) ;
3. des contraintes placées sur la base de données sont vérifiées en permanence (implémentation plus rare et plus coûteuse en temps machine) ;
4. des procédures lancées automatiquement par les opérations de modification (*triggers*) qui corrigent les données rendues incohérentes.

10.1 Types de données

La plupart des SGBD actuels définissent les champs des tables en précisant le type des données mémorisées[1]. Nous avons vu au chapitre précédent que les différents systèmes utilisent de nombreux types de données, parfois incompatibles avec le standard SQL. Oracle implémente superficiellement le standard, mais le remplace en fait en interne par son propre système.

[1] L'exception qui confirme la règle est *SQLLite* la nouvelle base de données intégrée à PHP5. Cette base de données présente il est vrai des caractéristiques étonnantes : ce n'est pas un serveur, elle ne précise pas de type pour les données, ce qui la distingue de la plupart des SGBDR. C'est loin cependant d'être un gadget, puisqu'elle admet les sous-requêtes, les procédures stockées, les triggers et les transactions, tout ce qui manquait à mySQL 4. Vu la diffusion de PHP, on doit s'attendre à voir ce moteur de base de données se généraliser pour de petites gestions de données sur Internet. Sa vocation n'est évidemment pas de remplacer les gros serveurs.

De nombreux langages classiques de programmation, suivant le modèle d'Algol, permettent à leurs utilisateurs de créer leurs propres types de données (essentiellement des intervalles de types existants et des structures). Qu'en est-il des SGBDR ? Ici encore, la réponse dépend des systèmes : chacun d'entre eux offre des possibilités, mais sans aucun respect de la standardisation.

10.1.1 La norme SQL2

La norme SQL2 propose une commande DDL qui permet de définir un nouveau type de données. Cela peut être une simple abréviation pour ne pas écrire `VarChar(30)`, par exemple, mais il est possible aussi de préciser une valeur par défaut et une contrainte de vérification (voir plus loin)[2].

```
CREATE DOMAIN <NomDuDomaine>
[AS] <TypeDeBase>
[DEFAULT <Valeur>]
[CHECK (<Contrainte>)]
```

Il existe une commande `DROP DOMAIN`, qui fonctionne pour autant qu'aucun champ de la base de données ne fasse référence à ce domaine. InterBase ajoute la possibilité de préciser `NOT NULL` :

Créer un domaine numérique Actif à valeur non nulle comprise entre 18 et 65 et ayant 25 comme valeur par défaut

```
CREATE DOMAIN ACTIF AS INTEGER
DEFAULT 25
CHECK (VALUE BETWEEN 18 AND 65)
NOT NULL;
```

10.1.2 Le silence d'Oracle

Oracle ne dispose pas de moyen pour créer des types propres aux utilisateurs[3]. La commande `CREATE TYPE` que les manuels de base laissent généralement de côté permet de créer des objets ressemblant à des structures, mais cela n'a qu'un lointain rapport avec les domaines

[2]SQL Server ne permet pas de créer de domaines, mais des *User Data Types* (UDT). Ces UDT ne permettent que de spécifier le type primitif et la possibilité d'avoir des valeurs nulles. Si on veut spécifier une valeur initiale et des contraintes supplémentaires, on devra définir des règles et des valeurs par défaut. A titre d'exemple, voici ce que deviendrait notre définition d'un actif :

```
CREATE DEFAULT AgeDef AS 25
GO
CREATE RULE LimitesAge AS @  >= 18 OR @ <= 65
GO
EXEC sp_addtype Actif, INT, 'not null'
EXEC sp_bindrule LimitesAge, Actif
EXEC sp_bindefault AgeDef, Actif
GO
```

C'est plus lourd que la syntaxe standard, mais cela offre la facilité de pouvoir réutiliser chaque contrainte.

[3]La non-existence d'une commande n'est pas toujours facile à étudier. De nombreux groupes de discussions évoquent la question des domaines dans Oracle. La question est souvent mal comprise, ce qui semble indiquer une méconnaissance du standard SQL de la part des intervenants. Une réponse classique est de proposer l'emploi des `TYPES`, mais je n'ai jamais trouvé un seul exemple concret et utilisable.

vus précédemment. L'intention des développeurs d'Oracle est de faciliter l'inclusion de données provenant d'un langage objet. Les types mémoriseront les valeurs des variables d'instance. Pour utiliser les types comme domaines, deux obstacles principaux interviennent :

- les types complexes d'Oracle permettent de créer des champs qui ne sont pas atomiques, même si on ne place qu'un seul champ dans l'objet, il ne sera pas accessible avec la même syntaxe qu'un champ normal
- les types d'Oracle ne permettent pas de définir de contraintes, ce qui est l'une des raisons principales de vouloir définir un domaine. Ils sont donc incapables de simuler des domaines composés d'intervalles restreints des types standards.

À titre d'exemple, je proposerai la définition d'un client en utilisant un type adresse.

```
-- Création du type

CREATE TYPE Address_type AS OBJECT
(Street  varchar2(100),
 city  varchar2(30),
 state  varchar2(3),
 postcode number(4),
 modified date);

-- Création d'une table utilisant le type

CREATE TABLE Customer2
(Name varchar2(30),
 Address Address_type);

-- Insertion d'une donnée

INSERT INTO Customer2
VALUES ('Jason Smart',Address_type('44 Smith Street',
'Sydney', 'NSW', 2000, Sysdate));

-- Relecture des données[4]

SELECT C.Name, C.Address.Street FROM Customer2 C;
```

10.2 Clés primaires

10.2.1 Nécessité et fonction

La clé primaire d'une relation *R* est le plus petit sous-ensemble d'attributs qui permette de définir une relation comportant le même nombre d'éléments que *R*. Dans le meilleur des cas, la clé primaire se compose d'un seul champ (dit alors sans doublons) ; au pire, la clé primaire est composée de tous les champs (c'est souvent le cas dans des tables représentant une association entre deux entités). On parle de clés primaires simples ou complexes. La clé primaire permet d'identifier de manière univoque une ligne d'une relation. Toute table bien conçue doit comporter une clé primaire. On dit souvent qu'une information dans une base de données est accessible au moyen de trois paramètres : le nom de la table, le nom du champ et la valeur de la clé primaire :

```
SELECT Champ FROM Table WHERE PK = Valeur;
```

[4] Sans l'alias de table, cette requête ne fonctionne pas.

Il arrive parfois que deux ensembles d'attributs puissent servir de clés primaires : on parle alors de clés concurrentes. On choisit d'ordinaire la clé primaire la plus facile à manipuler (qui sera la plus simple ou la plus compacte). On utilise alors une contrainte d'unicité pour la clé concurrente.

10.2.2 Définition

Je conseille de toujours nommer les contraintes, pour deux raisons : elles sont plus faciles à supprimer, désactiver ou modifier et elles permettent d'obtenir des messages plus éclairants lorsqu'elles provoquent une erreur.

La clé primaire peut se définir :

– au niveau du champ (elle est forcément simple) : il est obligatoire, possible ou interdit d'ajouter `NOT NULL` selon les systèmes. Oracle l'autorise. Il reste possible de donner un nom à la contrainte ainsi définie.

```
-- Définition de la clé primaire avec le champ
CREATE TABLE Test (
idTest INT [NOT NULL][CONSTRAINT pk_Test] PRIMARY KEY,
...);
```

– au niveau des contraintes globales : qu'elles soient simples ou complexes, on considère toujours qu'il y a une liste de champs :

```
-- Définition sous forme de contrainte globale
CREATE TABLE Test(
idTest INT [NOT NULL],
...,
CONSTRAINT [ pk_Test ] PRIMARY KEY (idTest),
... /* autres contraintes */)
;
CREATE TABLE Posseder(
idPersonne INT,
idVoiture INT,
...
CONSTRAINT [pk_Posseder] PRIMARY KEY (idPersonne,idVoiture),
... /* autres contraintes */)
;
```

– par après : on ajoute la définition de la clé primaire dans une clause `ALTER TABLE`. Personnellement, je n'encourage pas cette pratique, la clé primaire faisant partie de la définition fondamentale d'une table.

```
-- Ajout tardif d'une clé primaire à une table existante
ALTER TABLE Posseder(
ADD CONSTRAINT [pk_Posseder]
PRIMARY KEY (idPersonne,idVoiture));
```

Lors de l'ajout d'une clé primaire à une table existante, les données préexistantes doivent respecter la contrainte d'unicité, faute de quoi, la définition avorte.

10.2.3 Génération automatique d'une clé primaire

On trouve difficilement des clés primaires dans le monde réel. Les noms et prénoms présentent souvent des homonymies imprévisibles qui rendent la génération d'une clé primaire peu fiable. Les numéros « sociaux » manipulés par les administrations sont des clés primaires artificielles, d'origine informatique, et présentent souvent une complexité peu compatible avec des encodages conviviaux[5]. À l'échelle d'une petit entreprise, il sera souvent plus simple de donner une valeur numérique arbitraire pour désigner un client, un fournisseur ou un article de manière univoque. Tous les systèmes ne manipulent pas ces numéros automatiques de la même façon.

Un générateur automatique est un mécanisme qui assure que la valeur d'une clé primaire sera unique : il permettra d'obtenir des valeurs croissantes. Access dispose d'un pseudo-type `COUNTER` pour obtenir des clés uniques, c'est en réalité un entier long doté doté d'un mécanisme de génération automatique. Chaque SGBD dispose d'un moyen plus ou moins simple pour réaliser ce travail. En général, les nombres obtenus sont croissants, mais certaines manipulations risquent de faire disparaître certaines valeurs[6].

(a) Solution MS-SQL Serveur

```
CREATE TABLE Voitures
( Id_Voiture INTEGER CONSTRAINT pk_Voitures PRIMARY KEY IDENTITY,
  Marque VARCHAR(20),
  Modele VARCHAR(20),
  Prix INT
) ;
```

Le mot-clé `IDENTITY`, éventuellement suivi d'une valeur initiale et d'un incrément, sert à indiquer que le champ en question sera automatiquement rempli avec des valeurs croissantes. On aurait pu écrire `IDENTITY(1,1)`.

(b) Solution mySQL

mySQL utilise une solution similaire dont voici un exemple :

```
CREATE TABLE `Armes` (
  `idArme` int(10) unsigned NOT NULL auto_increment,
  `Description` varchar(20) default NULL,
  PRIMARY KEY  (`idArme`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=11 ;
```

Il est possible d'ajouter une valeur initiale après la définition de la table (c'est prévu pour les exportations). Notons qu'il ne peut y avoir qu'un seul champ auto-incrémenté par table.

(c) Solution Oracle

Oracle considère que les champs incrémentés ne sont pas nécessaires. Il propose de définir des séquences automatiques, qui seront utilisées par les tables pour générer les nombres

[5] Les numéros nationaux ou de sécurité sociale ont pour vocation de fonctionner dans des gros systèmes, à l'échelle d'un pays, voire d'un continent. Il doit donc y avoir des millions de codes possibles, ce qui entraîne de nombreux caractères.

[6] Il n'est donc pas possible d'utiliser un générateur automatique pour numéroter des factures : la loi impose que les factures soient numérotées avec des séquences continues.

auto-incrémentés. Cela permet un meilleur contrôle du processus. Un exemple pratique d'utilisation consiste à utiliser une même séquence pour deux tables, par exemple des clients et des prospects. Ainsi, quand un prospect devient client, il garde le même numéro, puisqu'aucun n'a pu porter le même numéro. L'unicité de la clé primaire est garantie par le partage de la séquence. C'est ainsi que la liste des envois qui concernent un ancien prospect, liés à lui par une clé étrangère, pourront continuer à le référencer, bien qu'il ait changé de table.

```
CREATE TABLE Voitures2 (
IdVoiture INTEGER DEFAULT -1 CONSTRAINT pk_Voitures PRIMARY KEY,
Marque VARCHAR2(20),
Modele VARCHAR2(20),
Prix INT ) ;
/* Création d'une séquence */
CREATE SEQUENCE s_Voitures INCREMENT BY 1 START WITH 1;
```

Il est possible de modifier l'incrément par l'ordre `ALTER`, mais la valeur initiale ne peut pas être changée[7].

```
ALTER SEQUENCE s_Client INCREMENT BY 2;

CREATE SEQUENCE s_Client INCREMENT BY 1 START WITH 1;
```

Pour obtenir la prochaine valeur de la séquence , on va utiliser la notation *séquence*`.nextval`. On répétera plusieurs fois la dernière valeur à l'aide de *séquence*`.currval`. Voici, pour illustrer, la création d'un client, suivie d'une attribution de facture :

```
INSERT INTO clients VALUES(s_Client.nextval,'Brol');
INSERT INTO factures VALUES(147,s_Client.currval,'7-fev-2004');
```

Pour automatiser la création d'un numéro de client, on pourra par exemple attribuer une valeur initiale hors de la séquence au champ « automatique » (par exemple 0) puis modifier la valeur automatiquement à l'aide d'un trigger[8].

```
CREATE TRIGGER auto_Voitures
BEFORE INSERT on Voitures2
FOR EACH ROW
BEGIN
  SELECT s_Voitures.nextval INTO :new.idVoiture
  FROM Dual;
END;
```

Notons que SQL Developer propose une génération automatique d'un trigger pour les clés autoincrémentées. Il présuppose l'existence des table et séquence impliquées. Il effectue également un test sur la valeur proposée pour la clé et ne la modifie que si elle est nulle. Ce comportement est intéressant, mais peut ne pas convenir. Il présuppose que si une valeur a été proposée, elle sera conservée. Cela peut mener à des blocages si les clés proposées par les utilisateurs se situent dans l'intervalle des valeurs proposées par la séquence.

[7] Il est toujours possible de supprimer la séquence par `DROP SEQUENCE NumClient` ; pour la récréer ensuite avec une nouvelle valeur initiale.

[8] Nous reviendrons sur le sujet des *triggers* plus loin dans le cours.

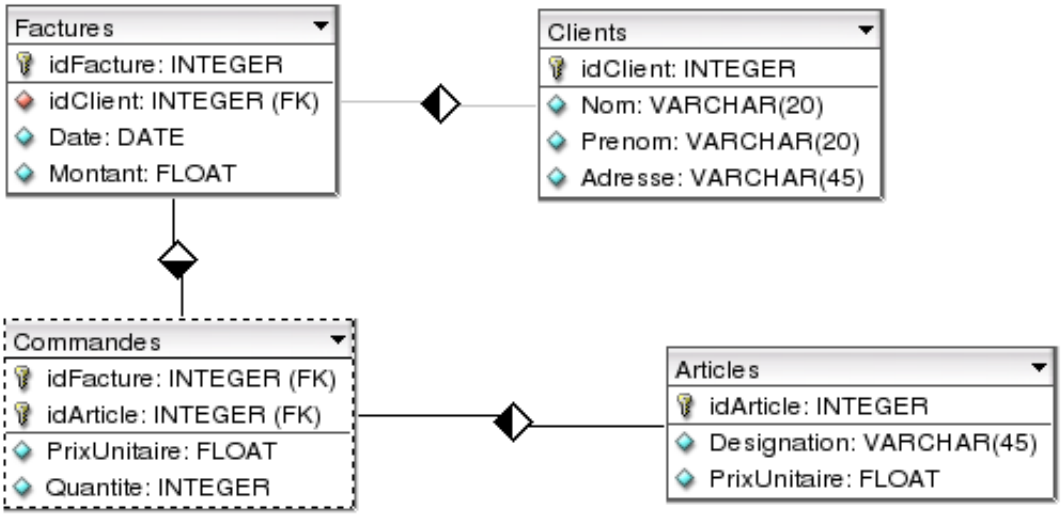

FIG. 10.1 – Contexte d'application d'une intégrité référentielle

10.3 Intégrité référentielle et clés étrangères

10.3.1 concepts

On appelle clé étrangère un champ qui fait référence à une clé primaire contenue dans une autre table. Elle sert à établir des liaisons entre tables. C'est le cas du numéro de client dans une facture, du numéro d'article dans une ligne de commande (voir figure 10.1). L'intégrité référentielle s'obtient en définissant des clés étrangères. Lorsqu'une clé étrangère est définie, toute valeur non nulle inscrite dans le champ doit avoir une valeur de clé primaire correspondante dans une autre table. En définissant ces contraintes d'intégrité, on garantit deux choses :

- qu'une valeur de clé étrangère fait toujours référence à une ligne dans la table liée (par exemple, tout numéro de client correspond à une ligne le décrivant dans la table des clients). En l'absence d'intégrité, on risque d'avoir des enregistrements orphelins (par exemple, une facture dont le numéro de client n'a pas d'équivalent dans la table des clients)
- que la suppression ou la modification d'une ligne référencée dans la table liée n'est pas possible car elle laisserait des lignes orphelines. Il sera donc impossible de supprimer un client ou de modifier sa clé primaire tant que la table des factures contient la valeur de sa clé primaire dans le champ défini comme clé étrangère.

10.3.2 définition d'une clé étrangère

Comme pour la clé primaire, on peut procéder de trois manières différentes : au niveau d'un champ unique, comme contrainte finale portant sur un ou plusieurs champs ou avec la commande `ALTER TABLE`. La syntaxe de définition au niveau du champ est la plus légère :

```
<NomChamp> <Type>
CONSTRAINT <NomContrainte>
REFERENCES <TableDest>(<Champ>)
```

La contrainte finale utilise l'expression `FOREIGN KEY` et une liste de champs. En pra-

tique, on préférera la technique de `ALTER TABLE`, qui résout agréablement le problème de définition des tables : en effet, une clé étrangère doit toujours être définie après les tables vers lesquelles elle définit une référence. On se simplifie la vie en définissant les tables sans clés étrangères dans un ordre quelconque, par exemple alphabétique, et en plaçant à la suite toutes les contraintes d'intégrité référentielle.

```
ALTER TABLE <NomTable>
ADD CONSTRAINT <NomContrainte>
FOREIGN KEY (<ListeChamps>)
REFERENCES <TableDest> (<ListeChamps>)
```

Création d'une base d'exemple, dans laquelle j'ai supprimé les champs superflus.

```
CREATE TABLE Clients
(  id_Client INT NOT NULL CONSTRAINT pkClient PRIMARY KEY,
   NomClient VARCHAR(20)
);
INSERT INTO Clients VALUES (1, 'Dupont');
INSERT INTO Clients VALUES (2, 'Durant');

CREATE TABLE Factures
(  id_Facture INT NOT NULL CONSTRAINT pkFactures PRIMARY KEY,
   id_Client INT
);
```

Ajout d'une règle d'intégrité référentielle

```
ALTER TABLE Factures ADD CONSTRAINT fkFacturesClients
FOREIGN KEY (Id_client) REFERENCES Client(Id_Client);
```

Ajout de 2 données dans les factures (violation de la règle pour la deuxième, pas de client n° 3)

```
INSERT INTO Factures VALUES (1,1);
INSERT INTO Factures VALUES (2,3);
ORA-02291: violation de contrainte (OCTOBRE.FKFACTURESCLIENTS)
d'intégrité - touche parent introuvable
INSERT INTO Factures VALUES (2,2);
```

Suppression d'un client lié à une facture (violation de la règle)

```
DELETE FROM Clients WHERE Id_Client = 2;
ORA-02292: violation de contrainte (OCTOBRE.FKFACTURESCLIENTS)
d'intégrité - enregistrement fils existant
```

Redéfinition de la contrainte pour permettre la suppression automatique du client et de ses factures.

```
ALTER TABLE Factures DROP CONSTRAINT fkFacturesClients;
ALTER TABLE Factures ADD CONSTRAINT fkFacturesClients
FOREIGN KEY (Id_client) REFERENCES Clients(Id_Client)
ON DELETE CASCADE;
DELETE FROM Clients WHERE Id_Client = 2;
SELECT * FROM Factures
```

```
ID_VENTE    ID_CLIENT
=========== ===========
1           1
```

On constate qu'à l'issue de la dernière requête, la facture associée au client supprimé a disparu.

Il est possible de définir quatre comportements en cas de modification ou suppression lors d'une infraction à un règle d'intégrité :

1. l'annulation de la commande (`NO ACTION`) ;
2. la suppression ou la modification en cascade des lignes qui faisaient référence aux lignes affectées (`CASCADE`) ;
3. le placement de la valeur par défaut dans les champs qui faisaient référence aux lignes modifiées ou supprimées (`SET DEFAULT`) ;
4. le placement de la valeur `NULL` dans les champs qui faisaient référence aux lignes modifiées ou supprimées (`SET NULL`).

ON DELETE **NO ACTION**	ON UPDATE **NO ACTION**
ON DELETE **CASCADE**	ON UPDATE **CASCADE**
ON DELETE **SET DEFAULT**	ON UPDATE **SET DEFAULT**
ON DELETE **SET NULL**	ON UPDATE **SET NULL**

Parmi ces huit possibilités, Oracle en a implémenté 2 (auxquelles on ajoutera, le refus d'action, implicite) :

```
ALTER TABLE Ventes ADD CONSTRAINT fkVentesClients
FOREIGN KEY (Id_client) REFERENCES Clients(Id_Client)
ON DELETE CASCADE;
```

ou

```
ALTER TABLE Ventes ADD CONSTRAINT fkVentesClients
FOREIGN KEY (Id_client) REFERENCES Clients(Id_Client)
ON DELETE SET NULL;
```

Remarque importante :

Lors de l'ajout d'une contrainte à une table contenant des données, le système vérifie que celles-ci satisfont la contrainte avant de la valider. Dans le cas où les données existantes ne permettent pas de satisfaire la contrainte, cette dernière n'est pas enregistrée. Le message d'erreur qui en résulte ne donne pas d'information sur les lignes qui enfreignent la nouvelle contrainte. Une recherche manuelle s'impose alors[9].

[9] L'apprenti administrateur devrait résister à la tentation d'encoder des données dans une table qui n'est pas totalement définie. De toutes façons, l'implantation de la base survient après une longue analyse, n'est-il pas vrai ?

10.4 Autres contraintes

10.4.1 Valeur par défaut

Si on ne veut pas obliger l'utilisateur à préciser une valeur probable, on peut spécifier une valeur par défaut pour un champ. Si on ne précise rien, le champ sera nul si c'est toutefois autorisé, autrement, il y aura une erreur. La valeur par défaut n'est pas à proprement parler une contrainte, puisqu'elle ne peut recevoir de nom.

Lors de son engagement, un employé est nécessairement attaché au service 'Formation'

```
...
Service VARCHAR2(15) DEFAULT "Formation" ,
...
```

Une valeur par défaut n'a pas beaucoup d'intérêt sur une clé primaire ou candidate, à moins qu'un *trigger* ne vienne la modifier dans la suite.

10.4.2 Clé candidate ou contrainte `UNIQUE`

La définition des clés primaires a été vue plus haut. L'unicité peut également concerner d'autres champs, qu'on appelle parfois clés alternatives. L'unicité ou la clé peut porter sur une colonne ou sur une combinaison de colonnes. C'est pourquoi on dispose de deux méthodes pour définir ces colonnes uniques : comme une contrainte appliquée au niveau de la définition de la colonne ou au niveau de la table. Dans ce dernier cas, on peut placer plusieurs noms de champs entre les parenthèses.

CONSTRAINT *<NomContrainte>*
UNIQUE (*<ListeDeChamps>*)

Dans notre exemple, la clé alternative sera le numéro national de l'employé, par définition unique. Pour des raisons d'efficacité (l'entreprise compte 50 employés), on a choisi une clé primaire plus simple à manipuler.

Création d'une table ayant une clé primaire et une colonne unique numéro national

```
CREATE TABLE Employes (
Id_Employe INT CONSTRAINT pkEmployé PRIMARY KEY,
NumNational VARCHAR(12) CONSTRAINT NumNat UNIQUE,
...)
```

10.4.3 `NULL` ou `NOT NULL`

La valeur `NULL` a bien des avantages, mais elle n'est pas toujours désirable. Les champs permettant une identification (clés primaires, ou noms de personnes ou de sociétés, pas nécessairement uniques d'ailleurs) doivent être spécifiés comme `NOT NULL`. À l'inverse, ceux qui autorisent la valeur `NULL` peuvent être marqués comme tels. L'utilisation du `NULL` devrait prendre en compte les deux impératifs d'efficacité suivant :

– l'autorisation des valeurs `NULL` est une incitation à la paresse. Le responsable de l'encodage peut sans cesse se dire : « je vérifierai plus tard » ou « je n'arrive pas à me décider, je demanderai à Untel ». On en arrive à disposer d'une base de données remplie de valeurs `NULL` qui ne répond jamais aux questions qu'on lui adresse.
– l'obligation d'encoder une donnée peut poser des problèmes réellement insolubles quand la donnée n'est vraiment pas disponible (groupe sanguin) ou parfois sans signification (date de la dernière condamnation pénale). Elle oblige l'encodeur à inventer lui-même des valeurs nulles qui ne seront pas traitées comme telles par le système (`'INCONNU'`, -1, ou `'9-9-99'`).

Un employé a toujours un nom et appartient nécessairement à un service

```
CREATE TABLE Employes( ...
Nom VARCHAR(20) NOT NULL,
Service VARCHAR(15) DEFAULT "Formation" NOT NULL, ...)
```

10.4.4 Contrainte `CHECK`

La contrainte `CHECK` permet de vérifier l'existence d'une condition lors de la création ou la modification d'une donnée. Si elle n'est pas vérifiée, l'insertion ou la modification sont rejetées. Il ne peut y avoir qu'une seule contrainte CHECK par colonne, mais elle peut contenir des opérateurs `AND` et `OR`.

On trouvera entre les parenthèses pratiquement tout ce qui peut figurer après `WHERE`.

Ajouter une colonne Salaire à la table Employés, en testant que le salaire doit être au moins supérieur à 750 Euros

```
ALTER TABLE Employes
ADD Salaire INT CONSTRAINT MinSalaire
CHECK (Salaire>= 750.0);
```

Les SGBDR sont plus ou moins permissifs par rapport à la manière de définir une contrainte. Oracle n'accepte pas la comparaison avec un autre champ, ni une requête imbriquée. Pour des contraintes aussi complexes, il faut alors se résoudre à définir un *trigger*.

10.5 Déclencheurs (*Triggers*)

Un *trigger* (en français déclencheur) est un petit morceau de code associé à une action précise sur une table. Comme il nécessite l'utilisation d'un langage de programmation, nous en réserverons l'étude dans les chapitres consacrés à l'étude du langage associé au serveur (PL/SQL pour Oracle). Nous en avons vu déjà deux exemples lors de la création des clés primaires automatiques.

Voici les caractéristiques d'un trigger :

1. il est toujours lié à une seule table.
2. il précise les opérations qui le déclenchent (l'un des trois verbes `INSERT`, `UPDATE`, `DELETE`).
3. il spécifie si son action se situe avant ou après l'insertion des données.
4. à la différence des contraintes diverses, il peut accéder à n'importe quelle donnée de la base pour effectuer des lectures ou des modifications. Il faut néanmoins prendre garde de ne pas déclencher des appels en cascades infinis.

5. il s'écrit dans un langage de programmation qui enrichit le langage SQL standard (définition et usage de variables, structures alternatives ou répétitives, éventuellement appel de procédures ou de fonctions).

10.6 Modification de la structure des données

10.6.1 Modifications lors de la création initiale des tables

Lors de la conception d'une base de données, on dispose généralement d'un dossier d'analyse qui a permis de définir clairement les besoins en termes de données. On a suivi le cheminement suivant :

- élaboration d'un modèle conceptuel des données (diagramme entités/associations) : ce modèle est fondamental. C'est de lui que dépend la qualité de la base finale.
- affinement en modèle organisationnel des données : c'est à ce niveau qu'on prévoit les données qui doivent vraiment figurer dans la base de données et quels utilisateurs y auront accès.
- création d'un modèle logique qui supprime certaines relations et reporte leurs propriétés sur des entités. On dispose maintenant d'outils permettant d'automatiser la transition vers le modèle logique et la traduction en instructions SQL.

Lors de l'implémentation de la base de données sur le serveur, il est prudent de travailler avec des scripts. Comme parfois des erreurs et des tâtonnements peuvent survenir, on veillera à rédiger ses scripts de la manière suivante :

1. commandes de suppression de toutes les contraintes :

```
ALTER TABLE XX DROP CONSTRAINT YY;
```

2. commandes de suppression de toutes les tables :

```
DROP TABLE XX;
```

3. commandes de création des tables et des clés primaires :

```
CREATE TABLE XX(...);
```

4. commandes de créations de contraintes d'intégrité :

```
ALTER TABLE XX ADD Constraint ZZ ...
```

5. commandes de création de *triggers* (voir plus avant dans le cours).

L'effacement de toutes les tables permet de créer un système performant. Une table modifiée par différents ajouts sera toujours plus lente d'accès, par suite de la dispersion des données. À ce stade, les données seront soit peu nombreuses et tapées à la console, soit importantes et inscrites dans des scripts également. Je n'ai pas pris en compte les index dans cette description, s'il y en a, ils seront traités comme les contraintes.

10.6.2 Modification de la structure des tables

Ces commandes ont été vue au chapitre précédent, mais figurent ici pour rappel.

Lors de l'évolution du logiciel, il est parfois nécessaire de modifier des tables : ajout de champs, modification de leurs caractéristiques, suppressions. Toutes ces opérations doivent être soigneusement planifiées : en effet, les nombreux programmes qui utilisent les données doivent rester compatibles avec les nouvelles données ou se voir modifiés. Certains systèmes

n'autorisent pas les changements de noms ou de types. Il faut alors trouver des astuces. Oracle est relativement conciliant à cet égard, mais rappelons que ce genre de facilités est à réserver à une base de données déjà fonctionnelle.

Ajout d'une colonne

```
ALTER TABLE Client ADD Surnom VARCHAR2(15) NULL;
```

Changement de type d'une colonne

```
ALTER TABLE Client MODIFY Surnom VARCHAR2(16);
```

Suppression d'une colonne

Notons d'emblée que la suppression d'une colonne dans une table va obliger une mise à jour de tous les programmes qui en font usage. Elle entraîne souvent aussi de grosses perturbations dans la base. On préfère alors masquer les données dans un premier temps et procéder à la suppression physique à un moment où les utilisateurs ne sont plus connectés. Attention, le masquage sous Oracle est néanmoins irréversible !

```
ALTER TABLE Client DROP COLUMN Surnom;
```

En utilisant les colonnes masquées :

```
ALTER TABLE Client SET UNUSED COLUMN Surnom;
```

Il est loisible de détruire physiquement les données par la suite :

```
ALTER TABLE Client DROP UNUSED COLUMNS;
```

Changement de nom d'une table

Le changement de nom peut parfois entraîner des problèmes avec des programmes existants (y compris les procédures stockées dans la base de données). Il sera parfois préférable d'utiliser d'autres solutions qui ne perturberont pas les programmes :

– utiliser une vue qui reprend exactement le contenu de la table qu'on veut renommer

```
CREATE VIEW Customer AS(
SELECT * FROM Client);
```

– définir un synonyme pour la table

```
CREATE SYNONYM Cliente FOR Client;
```

Il existe néanmoins une commande Oracle :

```
RENAME Client TO Customer;
```

Changement de nom d'une colonne

Les problèmes évoqués pour les tables se retrouvent ici aussi. On retrouve aussi la solution de la vue, mais elle oblige évidemment à l'emploi d'un autre nom que celui de la table.

– définition d'une vue :

```
CREATE VIEW Client2 AS(
SELECT idClient,Nom AS Name,Adresse
FROM Client);
```

– commande

```
ALTER TABLE Client
RENAME COLUMN Nom TO Name;
```

10.6.3 Désactivation et réactivation des contraintes

Lors de profonds remaniements dans les données, nous sommes souvent gênés par les contraintes qui nous obligent à chercher un ordre précis pour nos manipulations. Il est parfois même impossible de changer certaines données : une valeur de clé primaire qui sert de clé étrangère dans une autre table n'est pas modifiable dans un système qui ne possède pas l'option `ON UPDATE CASCADE`. Voici une séquence de commandes qui permettent de voir comment régler le problème.

```
SQL> select * from t1;
      IDT1 NOM1
---------- ---------------
         1 Un
         2 deux
SQL> select * from t2;
      IDT2 NOM2          IDT1
---------- ------- ----------
       100 one              1
       200 two              2
-- tentative de modifier la clé primaire 1
SQL> update t1 set idt1=1000 where idt1=1;
ORA-02292 : violation de contrainte (OCTOBRE.FK_T2_T1) d'intégrité -
enregistrement fils existant
-- désactivation de la contrainte et modification de la clé primaire 1
SQL> alter table t2 disable constraint fk_t2_t1;
Table modifiée.
SQL> update t1 set idt1=1000 where idt1=1;
1 ligne mise à jour.
-- tentative de réactiver la clé primaire
SQL> alter table t2 enable constraint fk_t2_t1;
ORA-02298 : impossible de valider (OCTOBRE.FK_T2_T1) -
clés parents introuvables
```

Les deux commandes `DISABLE CONSTRAINT` et `ENABLE CONSTRAINT` permettent respectivement de désactiver une contrainte pour modifier les tables (ici modifier la clé primaire du premier enregistrement) et de tenter de réactiver la contrainte (ce qui est impossible puisque la clé étrangère du premier enregistrement de l'autre table est devenue orpheline).

Pour réintroduire la cohérence dans la base, on peut faire plus attention lors des modifications, par exemple, en modifiant simultanément les données dans les deux tables :

```
-- modification de la clé primaire
SQL> update t1 set idt1=1000 where idt1=1;
-- modification de la clé étrangère
SQL> update t2 set idt1=1000 where idt1=1;
```

Il est possible aussi de repérer les erreurs décelées au moment de la réintroduction des contraintes. Pour cela, il faut disposer d'une table d'erreur dont voici la structure :

```
CREATE TABLE Problemes(
adresse ROWID,
utilisateur VARCHAR2(30),
nomtable VARCHAR2(30),
nomcontrainte VARCHAR2(30) );
```

Il devient possible de récupérer les erreurs lors de la tentative de réactivation de la contrainte :

```
SQL> alter table t2 enable constraint fk_t2_t1 exceptions into problemes;
SQL> SELECT * FROM Problemes;
ADRESSE            UTILISATEUR NOMTABLE  NOMCONTRAINTE
AAAHcIAAJAAAAIWAAA  OCTOBRE     T2        FK_T2_T1
```

On peut exploiter cette table pour créer une table avec les données problématiques :

```
CREATE TABLE Erreurs_T2 AS
    SELECT *  FROM T2
    WHERE rowid IN (SELECT adresse  FROM Problemes);
```