Bases de Données — Module M2104

IUT de Villetaneuse — R&T 1re année

Responsable du cours : Laure Petrucci

31 janvier 2017

Table des matières

1 Introduction **3**
1.1 Intégration . . . 3
1.2 Indépendance . . . 3
1.3 Disponibilité . . . 4
1.4 Sécurité . . . 4

2 Conception d'une Base de Données **5**
2.1 Cahier des charges . . . 5
2.2 Diagramme de classes . . . 5
2.2.1 Classes . . . 5
2.2.2 Objets . . . 6
2.2.3 Relations entre classes : associations . . . 7
2.2.4 Classe-association . . . 8
2.2.5 Diagramme de classes . . . 9
2.2.6 Élaboration d'un diagramme de classes . . . 9
2.3 Schéma relationnel . . . 10
2.3.1 Relations . . . 10
2.3.2 Instanciation d'une relation . . . 11
2.3.3 Clé primaire . . . 11
2.3.4 Contrainte d'intégrité et valeur nulle . . . 11
2.3.5 Clé étrangère . . . 11
2.3.6 Contrainte d'intégrité de référence . . . 12
2.4 Du diagramme de classes au schéma relationnel . . . 12
2.4.1 Classes . . . 12
2.4.2 Associations . . . 12
2.4.3 Association plusieurs à plusieurs . . . 12

3 Normalisation **14**
3.1 Objectifs . . . 14
3.2 Dépendances fonctionnelles . . . 14
3.3 $1^{\text{ère}}$ Forme Normale . . . 14
3.4 $2^{\text{ème}}$ Forme Normale . . . 15
3.5 $3^{\text{ème}}$ Forme Normale . . . 16
3.6 Règles sur les relations . . . 16

4 Algèbre relationnelle **18**
4.1 Opérateurs . . . 18
4.1.1 Projection . . . 18
4.1.2 Sélection . . . 19
4.1.3 Union . . . 19
4.1.4 Intersection . . . 20
4.1.5 Différence . . . 20

4.1.6 Produit . . . 21
4.1.7 Jointure . . . 21
4.2 Combinaison d'opérations . . . 22

5 SQL **24**
5.1 Introduction . . . 24
5.2 Création d'une table . . . 24
5.3 Suppression et modification de tables . . . 26
5.4 Manipulation du contenu d'une table . . . 26
5.4.1 Insertion de Tuples . . . 26
5.4.2 Mise à Jour de Tuples . . . 27
5.4.3 Suppression de Tuples . . . 27
5.5 Interrogation de la base de données . . . 27
5.5.1 La clause `SELECT` . . . 27
5.5.2 Réalisation d'une Projection . . . 28
5.5.3 Opération de sélection . . . 28
5.5.4 Tri des Tuples . . . 29
5.5.5 Jointure . . . 29
5.5.6 Union . . . 31
5.5.7 Intersection . . . 31
5.5.8 Différence . . . 32
5.5.9 Les comparateurs `ALL` et `ANY` . . . 32
5.5.10 Test d'existence . . . 33
5.5.11 Fonctions sur les Attributs . . . 33
5.6 Groupes . . . 34
5.6.1 Détermination des Groupes . . . 34
5.6.2 Fonctions sur les Groupes . . . 34
5.6.3 Clause `HAVING` . . . 35
5.7 Vues . . . 36
5.7.1 Création d'une Vue . . . 36
5.7.2 Utilisation d'une vue . . . 36
5.7.3 Suppression d'une Vue . . . 36
5.8 Règles . . . 37

6 Client C d'un SGBD **38**
6.1 Objectif . . . 38
6.2 La librairie `libpq-fe.h` . . . 38
6.3 Un exemple complet . . . 38

7 Client java d'un SGBD **40**
7.1 Objectif . . . 40
7.2 Schéma d'une API java . . . 40
7.3 Exemple complet . . . 40

Chapitre 1

Introduction

Une *base de données* permet de stocker les données communes à plusieurs applications de façon plus efficace qu'un système de fichiers.

Elle est constituée d'un ensemble de données reliées entre elles, accessibles à plusieurs utilisateurs simultanément.

Définition 1 *Un* SGBD *(*Système de Gestion de Bases de Données*) est un logiciel permettant de :*
- *créer des bases de données*
- *les interroger*
- *les mettre à jour*
- *assurer les contrôles d'intégrité, de concurrence et de sécurité.*

Les objectifs principaux de l'utilisation d'un SGBD sont de garantir des caractéristiques d'intégration, d'indépendance, de disponibilité et de sécurité de la base de données.

1.1 Intégration

Dans un système de traitement de données *orienté fichier*, les applications utilisent généralement un grand nombre de fichiers. Il y a par conséquent un risque de duplication ou de perte d'informations.

Exemple 1.1 : Considérons des outils de gestion des étudiants de l'IUT :
- un logiciel d'inscription, manipulant un fichier des étudiants inscrits ;
- un logiciel de gestion des notes, manipulant un fichier avec les notes des étudiants.

Les informations sur un même étudiant sont saisies deux fois (une sur chaque logiciel). De plus tout étudiant devrait être inscrit dans les deux fichiers, ce qui n'est pas nécessairement le cas si l'on a oublié par exemple de rentrer ses notes, ou s'il a démissionné et qu'on ne l'a supprimé que de la liste des inscrits. ◊

A contrario, l'utilisation d'une *base de données* garantit :
- la centralisation de toutes les données en un *réservoir unique de données* commun à toutes les applications ;
- la centralisation de tous les contrôles d'intégrité et de cohérence.

1.2 Indépendance

L'utilisation d'un SGBD garantit également une *indépendance physique* qui rend transparents aux utilisateurs les changements :

- de support ou de chemin d'accès aux données ;
- de méthode d'accès aux données.

Ainsi, la localisation de la base de données sur le serveur qui la gère peut-être modifiée sans impact pour celui qui l'utilise.

1.3 Disponibilité

La garantie de la disponibilité permet à un utilisateur d'accéder à une base de données même si elle est utilisée par ailleurs. Par conséquent, tout utilisateur ignore l'existence d'utilisateurs concurrents.

1.4 Sécurité

Enfin, deux aspects principaux de la *sécurité des données* sont fournis :

- l'*intégrité* : les données sont protégées contre des modifications invalides ;
- la *confidentialité* : la gestion de profils utilisateurs avec différentes autorisations limite l'accès illégal aux données.

Chapitre 2

Conception d'une Base de Données

La création d'une base de données s'effectue en plusieurs étapes, allant de la description du problème à résoudre, jusqu'à l'implémentation de la base de données et des outils logiciels permettant de l'exploiter :

1. rédaction en langage naturel (français) des besoins (*cahier des charges*) ;
2. conceptualisation avec un*diagramme de classes UML* (*Unified Modeling Language*) ;
3. traduction de ce diagramme de classes en un *schéma relationnel* représentant les l'organisation des données dans la base ;
4. écriture de scripts `SQL` pour la création et l'interrogation de la base de données ;
5. intégration d'appels à des scripts `SQL` dans des programmes.

Dans ce chapitre, nous détaillons les étapes 1, 2 et 3. Le chapitre 3 montre comment obtenir un bon schéma relationnel, et le chapitre 4 explicite les opérations permettant d'en extraire des données. Les éléments nécessaires du langage `SQL` pour l'écriture de scripts (étape 4) sont présentés dans le chapitre 5. L'intégration à un programme (étape 5) est réalisée dans les chapitres 6 (pour le langage `C`) et 7 (pour `java`).

2.1 Cahier des charges

Un *cahier des charges* décrit, en langage naturel, le problème que l'on souhaite résoudre. Les différents points indiqués dans ce document doivent être respectés pour la réalisation du projet.

Exemple 2.1 : On souhaite développer une application informatique de gestion de vols sur une flotte d'avions. Cette application doit permettre de gérer l'affectation des pilotes aux différents vols. Pour pouvoir gérer l'ensemble des pilotes, la compagnie qui les emploie a besoin de leur nom et de leur adresse. Un même vol peut avoir lieu plusieurs fois dans la semaine, des jours différents, mais toujours à la même heure, avec les mêmes villes de départ et de destination.

Certaines contraintes doivent être respectées pour des raisons de disponibilité et de sécurité évidentes :

- un pilote ne peut pas effectuer deux vols en même temps ;
- lors d'un vol, il doit y avoir un pilote dans l'avion. ◊

2.2 Diagramme de classes

2.2.1 Classes

Une classe permet de décrire un ensemble d'objets similaires.

Définition 2 *Une* classe *est la* description formelle d'un ensemble d'objets *ayant des propriétés (*attributs *et* méthodes*) communes.*

Les classes peuvent être instanciées en attribuant des valeurs à leur différents attributs.

Définition 3 *Un* objet *est une instance d'une classe.*

Exemple 2.2 : Supposons que l'on souhaite stocker des informations relatives à des personnes. Pour ce faire, nous pouvons concevoir une classe `Personne` disposant d'un nom, d'un prénom et d'une date de naissance. Une méthode calculant l'âge d'une personne peut être associée à cette classe. La classe `Personne` est représentée dans la figure 2.1.

Personne
- nom : String - prenom : String - dateNaissance : Date
+ age() : int

FIGURE 2.1 – Classe `Personne`

◊

La représentation graphique, telle que présentée dans la figure 2.1 suit certaines conventions. Elle comporte trois parties :

- **Nom :** le *nom* de la classe doit évoquer le *concept* décrit par cette classe. Le nom commence (généralement) par une majuscule.
- **Attributs :** les *attributs* définissent la *structure* d'un objet de la classe. Chaque *attribut* est défini par un *nom*, un *type*, une *visibilité* (- pour privé ou + pour public). Leur *valeur* qui peut différer d'un objet à un autre. Dans le cas général, la visibilité d'un attribut est privée.
- **Opérations ou méthodes :** les *opérations* décrivent les *actions* qu'un objet peut effectuer. Elles peuvent prendre des *valeurs en entrée*, *modifier les attributs* et/ou *produire des résultats*. L'*implémentation d'une opération* est appelée une *méthode*. Dans le cas général, la visibilité d'une méthode est publique.

Définition 4 *Un* constructeur *est une opération appelée lors de la création d'un objet. Il porte le même nom que la classe.*

Exemple 2.3 : Complétons la classe `Personne` de l'exemple 2.2 avec des constructeurs, comme dans la figure 2.2. La méthode `Personne` peut être appelée de deux manières différentes :

- **sans argument :** l'objet est créé mais ses attributs ne sont pas instanciés ;
- **avec des arguments :** ceux-ci sont utilisés pour instanciés les attributs de l'objet créé.

De plus, les attributs ayant une visibilité privée, il est d'usage d'accéder à leur valeur via des méthodes publiques. C'est l'objectif des deux méthodes `getNom` et `getPrenom`. ◊

2.2.2 Objets

Un objet se présente donc comme une copie de la classe contenant des valeurs d'attributs qui lui sont spécifiques. Il comporte également une identité qui permet d'y faire référence. Les méthodes qui lui sont associées définissent son comportement.

Personne
- nom : String - prenom : String - dateNaissance : Date
+ Personne() + Personne(nom :String, prenom : String, dateNais : Date) + getNom() : String + getPrenom() : String + age() : int

FIGURE 2.2 – La classe `Personne` avec ses constructeurs

Exemple 2.4 : Un programme crée un objet de la classe `Personne` en appelant le constructeur avec les arguments suivants : `Personne(Dupont,Jean,"29/02/1972")`. Cet objet est représenté dans la figure 2.3. Nous remarquons que les constructeurs ne font pas partie des méthodes associées à l'objet (ce qui est normal puisqu'ils ne servent qu'à créer des objets d'une classe).

jeanDupont : Personne
nom : Dupont prenom : Jean dateNaissance : 29/02/1972
getNom() : String getPrenom() : String age() : int

FIGURE 2.3 – L'objet `jeanDupont` de la classe `Personne`

◊

2.2.3 Relations entre classes : associations

Définition 5 *Une* association *est une relation entre des classes qui décrit les connexions structurelles entre leurs instances.*

Représentation graphique

Une association est représentée comme dans la figure 2.4 :

- une association binaire (entre deux classes) est matérialisée par un *trait plein* entre les classes associées ;
- elle peut avoir un *nom* : celui-ci figure alors au milieu du lien d'association ;
- elle peut avoir un *sens de lecture* (▶ ou ◀) ;
- de part et d'autre du lien d'association peuvent figurer des *rôles*.

Exemple 2.5 : Considérons deux classes définissant des `Polygone` et `Point`. Elles sont reliées par une association telle que décrite dans la figure 2.4. ◊

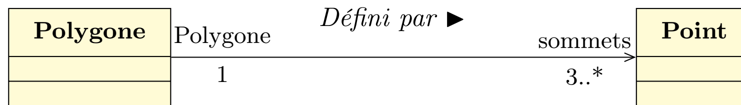

FIGURE 2.4 – Association entre des classes `Polygone` et `Point`

Multiplicité ou cardinalité

La *multiplicité* sur la terminaison cible fixe le nombre d'objets de la classe cible pouvant être associés à un seul objet donné de la classe source (la classe de l'autre terminaison de l'association) :

- exactement un : 1 ou 1..1
- plusieurs : $*$ ou 0..$*$
- au moins un : 1..$*$
- de un à six : 1..6

Exemple 2.6 : Dans l'exemple de la figure 2.4, les multiplicités indiquent qu'un `Polygone` est défini par au moins 3 `Point` (ses sommets). ♢

Définition 6 *Une association est dite* réflexive *quand les* deux extrémités *de l'association aboutissent à la* même classe*.*

Exemple 2.7 : Dans la figure 2.5 est représentée une association réflexive `amitié` qui modélise le lien d'amitié entre deux personnes. Dans la partie droite de la figure, deux objets de la classe `Personne`, `marie` et `jean` sont reliés par ce lien.

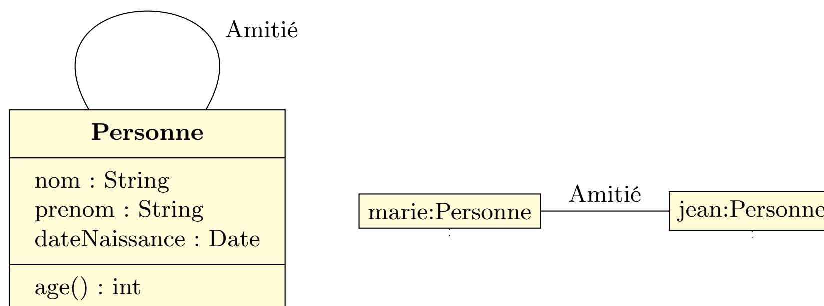

FIGURE 2.5 – Une association réflexive sur la classe personne, et le lien entre deux objets de cette classe

♢

2.2.4 Classe-association

Définition 7 *Une* classe-association *est une classe ne servant qu'à contenir les attributs d'une association. Elle est reliée à l'association par un* trait discontinu*.*

Exemple 2.8 : Soient des classes `Personne` et `Entreprise`. On souhaite modéliser la relation d'`Emploi` d'une personne dans une entreprise. La figure 2.6 montre qu'elles sont reliées par un classe-association `Emploi` qui comporte également des informations dur la date d'embauche et sur le salaire. Ces deux informations ne peuvent en effet pas faire partie des caractéristiques de l'entreprise (tous les employés n'ont pas été embauchés en même temps et avec le même salaire) ni de l'employé (qui peut avoir plusieurs emplois dans des entreprises différentes, avec des salaires et dates d'embauche différents). De plus, les cardinalités sont toutes `*` car une entreprise qui vient d'être créée n'a pas encore d'employé, mais peut en général en avoir un nombre indéterminé. De même une personne peut avoir plusieurs emplois à temps partiel dans une ou plusieurs entreprises, ou être au chômage (et ne pas être associée à une entreprise).

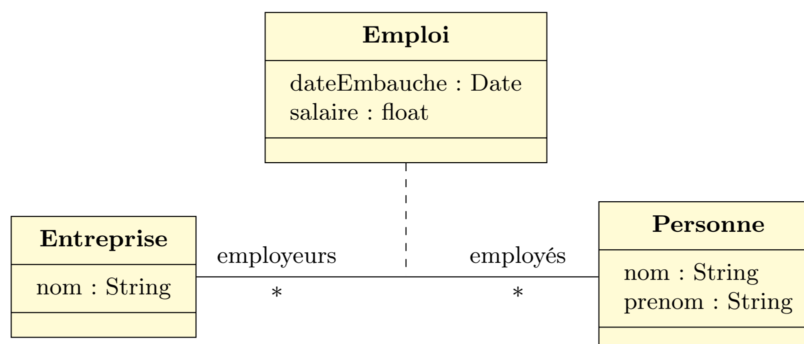

FIGURE 2.6 – Une classe-association

◊

2.2.5 Diagramme de classes

Les classes, associations et classes-associations vues dans les sections 2.2.1, 2.2.3 et 2.2.4 permettent de constituer un *diagramme de classes* modélisant le problème dans son intégralité. C'est le diagramme le plus important de la modélisation objet. Il permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier. Il procure une vue statique du système.

2.2.6 Élaboration d'un diagramme de classes

Il s'agit, à partir du cahier des charges (voir section 2.1), de trouver les *classes* du domaine étudié. Ceci se fait généralement en collaboration avec un expert du domaine en extrayant du texte les *concepts ou substantifs*. Les *associations* entre classes sont quant à elles issues des *verbes ou constructions verbales* mettant en relation plusieurs classes (telles que *est composé de*, *travaille pour*...). Les *attributs* des classes sont obtenus à partir de *substantifs ou groupes nominaux* (comme *la masse d'une voiture*, *le montant d'une transaction*...).

Dans une optique *bases de données*, on ne s'intéresse qu'aux *attributs* des classes, et non aux méthodes qui pourraient lui être associées. Lorsqu'elle est impliquée dans un projet *base de données*, une classe doit avoir un attribut jouant le rôle d'*identifiant unique*.

Exemple 2.9 : Analysons le cahier des charges de l'exemple 2.1 de gestion de vols aériens en surlignant en jaune les mots pouvant conduire à des classes, en bleu à des associations ou des classes-associations, et en vert à des attributs.

"On souhaite développer une application informatique de gestion de vols sur une flotte d' avions. Cette application doit permettre de gérer l'affectation des pilotes aux différents vols. Pour pouvoir gérer l'ensemble des pilotes, la compagnie qui les emploie a besoin de leur nom et de leur adresse. Un même vol peut avoir lieu plusieurs fois dans la semaine, des jours différents, mais toujours à la même heure, avec les mêmes villes de départ et de destination.

Certaines contraintes doivent être respectées pour des raisons de disponibilité et de sécurité évidentes :

- un pilote ne peut pas effectuer deux vols en même temps ;
- lors d'un vol, il doit y avoir un pilote dans l'avion."

On en déduit le diagramme de classes de la figure 2.7 dans lequel `id_pilote` est l'identifiant de la classe `Pilote` et `id_vol` celui de la classe `Vol`. Les cardinalités traduisent la contrainte de présence d'au moins un pilote sur chaque vol. Par contre un pilote peut n'être affecté à aucun vol.

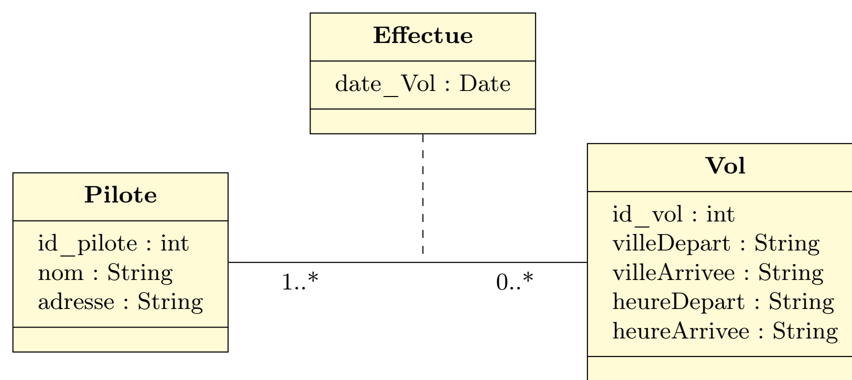

FIGURE 2.7 – Diagramme de classes pour les vols aériens

◊

2.3 Schéma relationnel

Un *schéma relationnel* fournit sous forme de *relations* une description simple des classes et de certaines associations du diagramme de classes.

2.3.1 Relations

Définition 8 *Une* relation *R est un ensemble* d'attributs*.*

Exemple 2.10 : Les deux classes de l'exemple 2.9 des vols aériens se traduisent par les relations suivantes :

```
PILOTE(id_pilote, nom, adresse)
VOL(id_vol, ville_départ, ville_arrivée, heure_départ, heure_arrivée)
```

La relation `PILOTE` a pour attributs : `id_pilote`, `nom` et `adresse`. ◊

2.3.2 Instanciation d'une relation

Une *relation* est instanciée par une table. Chaque ligne de la table, appelée *tuple* contient une instance de la relation. De la même manière qu'une relation correspond à une classe, un tuple d'une relation correspond à un objet de la classe.

Exemple 2.11 : La table 2.1 montre une instanciation de la relation `PILOTE`. Le pilote d'identifiant 4 s'appelle Durand et habite Paris.

PILOTE	id_pilote	nom	adresse
	3	Garratt	Perth
	4	Durand	Paris
	5	MacMachin	Glasgow

TABLE 2.1 – Instanciation de la relation PILOTE

◊

2.3.3 Clé primaire

Définition 9 *Une* clé primaire *d'une relation est un attribut ou un groupe d'attributs de la relation qui identifie un tuple unique.*

Propriété 1 *Une relation possède* une et une seule clé primaire*, mais peut contenir plusieurs clés qui pourraient jouer ce rôle (clés candidates).*

Dans le cas d'une relation issue d'une classe, la clé primaire correspond à l'identifiant de la classe. La clé primaire d'une relation est représentée en soulignant les attributs qui la composent.

Exemple 2.12 : `id_pilote` est la clé primaire de la relation `PILOTE`.

```
PILOTE(id_pilote, nom, adresse)
VOL(id_vol, ville_départ, ville_arrivée, heure_départ,heure_arrivée)
```

◊

2.3.4 Contrainte d'intégrité et valeur nulle

Tout SGBD relationnel doit vérifier *l'unicité et le caractère défini* (non nul) des valeurs de la clé primaire.

Lors de l'insertion de tuples dans une relation, il arrive qu'un attribut soit inconnu ou non défini. On introduit alors une valeur conventionnelle, appelée *valeur nulle*. Une clé primaire ne peut pas avoir une valeur nulle.

2.3.5 Clé étrangère

Définition 10 *Une* clé étrangère *dans une relation est une clé primaire dans une autre relation.*

Exemple 2.13 : Compétons l'exemple des vols aériens en ajoutant des relations `AVION` et `MODÈLE`, comme dans les tables 2.2 et 2.3 respectivement.

`type` est la *clé primaire* de la relation `MODÈLE` et une *clé étrangère* de la relation `AVION`. En effet `type` identifie le modèle d'avion, et chaque avion a un `type` connu.

◊

AVION

id_avion	type	compagnie
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

Table 2.2 – Les avions

MODÈLE

type	constructeur	capacité
A340	Airbus	228
B747	Boeing	432

Table 2.3 – Les modèles d'avions

2.3.6 Contrainte d'intégrité de référence

Pour assurer l'intégrité d'une base de données, lors de l'insertion d'un tuple dans une table, le SGBD doit vérifier que les valeurs des clés étrangères existent dans la table où elles sont clé primaire.

De même, lors de la suppression d'un tuple d'une table dont la clé primaire est clé secondaire d'une autre table, des mises à jour de cette dernière peuvent être nécessaires. Dans ce cas, plusieurs comportements sont possibles :

- interdire la suppression si la valeur de la clé primaire existe dans la table où elle est clé secondaire (recommandé) ;
- supprimer ces valeurs ;
- avertir l'utilisateur d'une incohérence ;
- remplacer ces valeurs par la valeur nulle.

Exemple 2.14 : Lors de l'insertion d'un tuple dans la table `AVION`, le système doit vérifier que la valeur de l'attribut `type` existe bien dans la table `MODÈLE`.

Lors de la suppression d'un tuple dans `MODÈLE`, on peut interdire la suppression si la valeur de `type` existe dans `AVION`. ◊

2.4 Du diagramme de classes au schéma relationnel

Chaque élément du diagramme de classes est reflété dans le schéma relationnel.

2.4.1 Classes

Chaque *classe* du diagramme UML est transformée de la manière suivante :

- elle devient une relation contenant tous les attributs de la classe ;
- si elle possède un identifiant, il devient la clé primaire, sinon il faut ajouter une clé primaire arbitraire.

2.4.2 Associations

Association un à plusieurs

Dans le cas d'une association un à plusieurs, il faut ajouter un attribut clé étrangère dans la relation de cardinalité n. Cette clé étrangère est la clé primaire de la relation de cardinalité 1.

Association un à un

Dans le cas d'une association un à un, il faut rajouter un attribut de type clé étrangère dans une des relations. Cette clé étrangère est la clé primaire de l'autre relation.

2.4.3 Association plusieurs à plusieurs

Dans le cas d'une association plusieurs à plusieurs, l'association ou la classe-association devient une relation dont la clé primaire est composée par la concaténation des clés primaires des classes

connectées à l'association. Ces attributs deviennent clé étrangères dans la nouvelle relation. Les attributs de la classe-association doivent être ajoutés à la nouvelle relation. Ces attributs peuvent faire partie de la clé primaire.

Exemple 2.15 : Considérons le diagramme de classes de la figure 2.7.

Les classes `Pilote` et `Vol` sont transformées en relations `PILOTE` et `VOL`, respectivement. Leurs identifiants, `id_pilote` et `id_vol` sont leurs clés primaires.

Ces deux classes sont reliées par une classe-association (plusieurs à plusieurs) `Effectue`, qui se traduit par une nouvelle relation `EFFECTUE` dont la clé primaire est `(id_pilote,id_vol,date_vol)`. Notons que cette clé primaire est composée de trois attributs :

- `id_pilote` et `id_vol` qui sont tous deux des clés étrangères. Elles permettent de référencer à la fois le pilote et le vol ;
- `date_vol` qui permet d'identifier un vol à une date spécifique. En effet, le pilote peut effectuer ce même vol plusieurs fois dans la semaine, et ceux-ci doivent pouvoir être distingués.

Par conséquent, on obtient le schéma relationnel suivant :

```
PILOTE(id_pilote, nom, adresse)
VOL(id_vol, ville_départ, ville_arrivée, heure_départ, heure_arrivée)
EFFECTUE(id_pilote, id_vol, date_vol)
```

◊

Chapitre 3

Normalisation

3.1 Objectifs

Le objectifs de la normalisation d'un schéma relationnel sont non seulement de s'assurer que celui-ci est *correct*, mais aussi d'éviter :

- la *redondance*, c'est-à-dire la duplication d'informations ;
- les *incohérences* par ajout ;
- la *perte d'informations* par suppression.

3.2 Dépendances fonctionnelles

Définition 11 *Soit $R(X, Y, Z)$ une relation.*
Si à une valeur de X n'est associée qu'une seule valeur de Y, on dit que X détermine *Y. On dit encore que Y* dépend fonctionnellement *de X.*

La dépendance fonctionnelle est notée $X \rightarrow Y$.

3.3 1ère Forme Normale

Définition 12 *Une relation est en* 1ère forme normale *si et seulement si chaque attribut ne contient qu'une seule valeur à chaque instant (on n'affecte pas plusieurs valeurs à un seul attribut).*

Exemple 3.1 : La relation `AVION` de la table 3.1 n'est pas en 1ère forme normale, contrairement à celle de la table 3.2. En effet, dans le premier cas, l'avion numéro 20 est associé à deux compagnies aériennes.

AVION

id_avion	type	capacité	compagnie
10	A340	228	Air France
20	B747	432	British Airways Qantas

TABLE 3.1 – Une relation `AVION` pas en première forme normale

◊

AVION	id_avion	type	capacité	compagnie
	10	A340	228	Air France
	20	B747	432	British Airways
	30	B747	432	Qantas

Table 3.2 – Une relation AVION en première forme normale

3.4 $2^{ème}$ Forme Normale

Définition 13 *Une relation est en* $2^{ème}$ forme normale *si et seulement si :*

- *elle est en* $1^{ère}$ *forme normale ;*
- *tout attribut n'appartenant pas à la clé primaire ne dépend pas d'une partie de cette clé.*

La seconde condition permet ainsi de choisir une clé primaire ne contenant pas d'attributs superflus.

Exemple 3.2 : Soit la relation AVION de la table 3.3 ayant pour clé primaire le couple (id_avion,constructeur). Les dépendances fonctionnelles sont :

1. id_avion → type
2. id_avion → compagnie
3. type → capacité
4. type → constructeur

Cette relation est en $1^{ère}$ forme normale. Par contre, les dépendances fonctionnelles 1 et 2 ne satisfont pas la seconde condition pour que la relation AVION soit en $2^{ème}$ forme normale. En effet, les attributs type et compagnie n'appartiennent pas à la clé primaire mais dépendent uniquement de id_avion qui n'est qu'une partie de la clé.

AVION	id_avion	constructeur	type	capacité	compagnie
	10	Airbus	A340	228	Air France
	20	Boeing	B747	432	British Airways
	30	Boeing	B747	432	Qantas

Table 3.3 – Une relation AVION pas en deuxième forme normale

Si l'on considère maintenant la relation AVION de la table 3.4 où la clé primaire est réduite à l'attribut id_avion, les dépendances fonctionnelles sont :

1. id_avion → type
2. id_avion → compagnie
3. type → capacité
4. type → constructeur

Ainsi, toutes les conditions sont satisfaites et la relation AVION est en $2^{ème}$ forme normale.

AVION	id_avion	constructeur	type	capacité	compagnie
	10	Airbus	A340	228	Air France
	20	Boeing	B747	432	British Airways
	30	Boeing	B747	432	Qantas

Table 3.4 – Une relation AVION en deuxième forme normale

◊

Remarque 1 *la vérification de la deuxième forme normale n'a de sens que dans le cas où la clé primaire est constituée de plusieurs attributs.*

3.5 3$^{\text{ème}}$ Forme Normale

Définition 14 *Une relation est en* 3$^{\text{ème}}$ forme normale *si et seulement si :*

- *elle est en* 2$^{\text{ème}}$ *forme normale ;*
- *tout attribut n'appartenant pas à la clé ne dépend pas d'un attribut non clé.*

La seconde condition garantit que chaque attribut ne dépend de rien d'autre que de la clé. Si ce n'est pas le cas, la relation peut être divisée en plusieurs relations, ce qui permet d'optimiser le stockage des données et les performances.

Exemple 3.3 : Considérons à nouveau la relation `AVION` de la table 3.4. Nous avons montré dans l'exemple 3.2 qu'elle est en 2$^{\text{ème}}$ forme normale. Ses dépendances fonctionnelles 3 et 4 invalident la seconde condition nécessaire pour qu'elle soit également en 3$^{\text{ème}}$ forme normale. En effet, les attributs `capacité` et `constructeur` n'appartiennent pas à la clé primaire et dépendent de l'attribut `type` qui ne fait pas partie de la clé.

Pour obtenir des relations en troisième forme normale, nous divisons la relation `AVION` en deux relations : `AVION` (table 3.5) et `MODÈLE` (table 3.6).

`AVION`

id_avion	type	compagnie
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

TABLE 3.5 – Une relation `AVION` en troisième forme normale

Les dépendances fonctionnelles de la relation `AVION` sont :

1. `id_avion` → `type`
2. `id_avion` → `compagnie`

Par conséquent, cette nouvelle relation `AVION` est en 3$^{\text{ème}}$ forme normale.

`MODÈLE`

type	constructeur	capacité
A340	Airbus	228
B747	Boeing	432

TABLE 3.6 – Une relation `MODÈLE` en troisième forme normale

Les dépendances fonctionnelles de la relation `MODÈLE` sont :

1. `type` → `capacité`
2. `type` → `constructeur`

Par conséquent, la relation `MODÈLE` est également en 3$^{\text{ème}}$ forme normale. ◊

3.6 Règles sur les relations

Définition 15 *Un* domaine *est un* ensemble de valeurs *identifiées par un nom. Il peut être défini :*

- *en intension (par exemple : entier, réel, chaîne de caractères) ;*
- *en extension (par exemple : villes = {Paris, Marseille, Lyon}).*

Propriété 2 *Les tuples et attributs satisfont les propriétés suivantes :*

- *chaque tuple d'une relation est unique ;*
- *l'ordre des tuples dans une relation n'a pas de signification ;*
- *chaque attribut prend ses valeurs dans un seul domaine ;*
- *plusieurs attributs peuvent prendre leurs valeurs dans un même domaine.*

Chapitre 4

Algèbre relationnelle

Une fois le schéma relationnel conçu, on souhaite pouvoir extraire des informations des tables. Pour ce faire, il est nécessaire de pouvoir exprimer les opérations à effectuer. L'algèbre relationnelle formalise les opérations de base permettant d'effectuer des requêtes sur une base de données relationnelle.

4.1 Opérateurs

L'*algèbre relationnelle* utilise une collection d'*opérateurs* qui agissent sur des relations et produisent des relations comme résultat.

Ces opérateurs peuvent avoir des arités ou des natures différentes :

- unaires : *projection*, *sélection* ;
- binaires (ou n-aires) : *jointure*, *division* ;
- ensemblistes : *union*, *produit*, *différence*, *intersection*.

Il sont détaillés dans les sections suivantes.

4.1.1 Projection

Définition 16 *La* projection *d'une relation R sur un ensemble d'attributs $\{A_i\}$ de R est un sous-ensemble de R réduit aux sous-tuples ne contenant que les attributs de $\{A_i\}$.*

La projection est notée $\Pi_{A_1,\ldots,A_n}$(nom-relation).

Cela revient à supprimer, dans la table de la relation R, les colonnes correspondant aux attributs n'appartenant pas à $\{A_i\}$, puis à supprimer les doublons éventuels.

Exemple 4.1 : Quels sont les différents types d'avions ?

Pour répondre à cette question, il faut projeter la relation avion sur l'attribut `type`. La table 4.1 montre la colonne retenue, et la table 4.2 le résultat une fois les doublons éliminés.

AVION	id_avion	constructeur	type	capacité	compagnie
	10	Airbus	A340	228	Air France
	20	Boeing	B747	432	British Airways
	30	Boeing	B747	432	Qantas

Table 4.1 – Projection de la relation `AVION` sur son attribut `type`

◊

$\Pi_{\texttt{type}}(\texttt{AVION})$

type
A340
B747

Table 4.2 – Relation résultant de la projection

4.1.2 Sélection

Définition 17 *La* sélection *sur une relation R suivant une condition C portant sur des attributs de R est un sous-ensemble de R dont les tuples satisfont C.*

La sélection est notée $\sigma_C(R)$.

L'opération de sélection revient à supprimer, dans la table de la relation R, les lignes ne satisfaisant pas la condition C.

Exemple 4.2 : Quels avions peuvent accueillir au moins 300 passagers ?

Pour répondre à cette question, il faut sélectionner les tuples tels que `capacité` ≥ 300. L'opération est schématisée dans la table 4.3, et le résultat dans la table 4.4.

AVION

id_avion	constructeur	type	capacité	compagnie
10	Airbus	A340	228	Air France
20	Boeing	B747	432	British Airways
30	Boeing	B747	432	Qantas

Table 4.3 – Opération de sélection

$\sigma_{\texttt{capacité} \geq 300}(\texttt{AVION})$

id_avion	constructeur	type	capacité	compagnie
20	Boeing	B747	432	British Airways
30	Boeing	B747	432	Qantas

Table 4.4 – Résultat de la sélection

◊

4.1.3 Union

Définition 18 *L'*union *est une opération ensembliste portant sur deux relations R_1 et R_2* de même schéma. *$R_1 \cup R_2$ est la relation de même schéma contenant les tuples de R_1 et ceux de R_2.*

L'union de deux relations R_1 et R_2 consiste en une copie des tuples de R_1 et de R_2, puis une suppression des doublons des doublons éventuels.

Exemple 4.3 : Quels sont tous les pilotes travaillant pour Air France ou pour British Airways ?

Supposons que soient au préalable construites deux relations `P_AF` et `P_BA` contenant les pilotes travaillant respectivement pour Air France et pour British Airways. Ces relations sont décrites dans les tables 4.5 et 4.6. Le résultat de l'union `P_AF`$\cup$`P_BA` est présenté dans la table 4.7. Nous remarquons que le pilote numéro 3, employé par les deux compagnies, n'apparaît qu'une fois. ◊

P_AF	id_pilote	nom	adresse
	1	Dupond	Nice
	2	Smith	Londres
	3	Garratt	Perth

Table 4.5 – Pilotes d'Air France

P_BA	id_pilote	nom	adresse
	3	Garratt	Perth
	4	Durand	Paris
	5	MacMachin	Glasgow

Table 4.6 – Pilotes de British Airways

P_AF∪P_BA	id_pilote	nom	adresse
	1	Dupond	Nice
	2	Smith	Londres
	3	Garratt	Perth
	4	Durand	Paris
	5	MacMachin	Glasgow

Table 4.7 – Pilotes travaillant pour Air France ou pour British Airways

4.1.4 Intersection

Définition 19 *L'*intersection *est une opération ensembliste portant sur deux relations R_1 et R_2 de même schéma. $R_1 \cap R_2$ est la relation ayant le même schéma que R_1 et R_2, et pour tuples ceux appartenant à la fois à R_1 et à R_2.*

Exemple 4.4 : Quels sont les pilotes travaillant pour Air France et British Airways ?

Ce sont les pilotes que l'on trouve à la fois dans P_AF (table 4.5) et dans P_BA (table 4.6). La table P_AF∩P_BA résultante est présentée dans la figure 4.8.

P_AF∩P_BA	id_pilote	nom	adresse
	3	Garratt	Perth

Table 4.8 – Pilotes travaillant pour Air France et pour British Airways

◊

4.1.5 Différence

Définition 20 *La* différence *est une opération ensembliste portant sur deux relations R_1 et R_2 de même schéma. $R_1 \setminus R_2$ est la relation ayant le même schéma que R_1 et R_2, et pour tuples ceux appartenant à R_1 mais pas à R_2.*

Exemple 4.5 : Quels sont les pilotes d'Air France ne travaillant pas pour British Airways ?

Ce sont les pilotes que l'on trouve dans P_AF (table 4.5) mais pas dans P_BA (table 4.6). La table P_AF\P_BA résultante est présentée dans la figure 4.9.

P_AF\P_BA	id_pilote	nom	adresse
	1	Dupond	Nice
	2	Smith	Londres

Table 4.9 – Pilotes d'Air France ne travaillant pas pour British Airways

◊

4.1.6 Produit

Définition 21 *Le* produit *est une opération ensembliste portant sur deux relations R_1 et R_2. $R_1 \times R_2$ est la relation ayant pour schéma la concaténation des schémas de R_1 et R_2, et pour tuples toutes les combinaisons de tuples de R_1 et R_2.*

Cette opération correspondant à produit cartésien n'est que très rarement utilisée.

Exemple 4.6 :

R_1

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3

TABLE 4.10 – Relation R_1

R_2

D	E
d1	e1
d2	e2

TABLE 4.11 – Relation R_2

$R_1 \times R_2$

A	B	C	D	E
a1	b1	c1	d1	e1
a1	b1	c1	d2	e2
a2	b2	c2	d1	e1
a2	b2	c2	d2	e2
a3	b3	c3	d1	e1
a3	b3	c3	d2	e2

TABLE 4.12 – Produit $R_1 \times R_2$

$\diamondsuit$

4.1.7 Jointure

Définition 22 *La* jointure *est une opération ensembliste portant sur deux relations R_1 et R_2. $R_1 \bowtie_{attributs} R_2$ est la relation dont les attributs sont ceux de R_1 et ceux de R_2, et dont les tuples sont obtenus en composant un tuple de R_1 et un tuple de R_2 ayant la même valeur pour les attributs précisés.*

La jointure est une opération essentielle pour effectuer des requêtes sur une base de données car elle permet de trouver des informations liées entre elles au travers de plusieurs relations.

Exemple 4.7 : Considérons à nouveau les relations `AVION` (table 4.13) et `MODÈLE` (table 4.14). Ces relations partagent un attribut qui est le `type` d'avion. La jointure `AVION` $\bowtie_{\texttt{type}}$ `MODÈLE` permet de voir toutes les informations concernant un avion. Elle est présentée dans la table 4.15.

`AVION`

`id_avion`	`type`	`compagnie`
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

TABLE 4.13 – La relation `AVION`

`MODÈLE`

`type`	`constructeur`	`capacité`
A340	Airbus	228
B747	Boeing	432

TABLE 4.14 – La relation `MODÈLE`

`AVION` $\bowtie_{\texttt{type}}$ `MODÈLE`

`id_avion`	`type`	`compagnie`	`constructeur`	`capacité`
10	A340	Air France	Airbus	228
20	B747	British Airways	Boeing	432
30	B747	Qantas	Boeing	432

TABLE 4.15 – Jointure entre `AVION` et `MODÈLE`

$\diamondsuit$

Lorsque les deux relations participant à une jointure ont des attributs communs, la jointure naturelle est utilisée.

Définition 23 *La* jointure naturelle *est une opération ensembliste portant sur deux relations R_1 et R_2. $R_1 \bowtie R_2$ est la relation dont les attributs sont ceux de R_1 et ceux de R_2, et dont les tuples sont obtenus en composant un tuple de R_1 et un tuple de R_2 ayant la même valeur pour les attributs communs aux deux relations.*

Il est parfois souhaitable de conserver les tuples d'une relation n'ayant pas de tuple correspondant dans l'autre. Pour ce faire, on utilise une jointure extérieure.

Définition 24 *La* jointure extérieure *est similaire à la jointure naturelle, mais conserve également les occurrences d'une relation qui n'ont pas de correspondant dans l'autre relation. Elle associe aux attributs non renseignés la valeur nulle.*

Exemple 4.8 : Reprenons l'exemple 4.7, en supposant que l'avion numéro 10 a été supprimé de la base de données (par exemple parce qu'il n'est plus utilisé pour des vols commerciaux). Les relations qui nous intéressent sont présentées dans les tables 4.16 et 4.17. Le jointure extérieure dans la table 4.18 montre que, bien qu'il n'y ait plus d'avions de type A340 en circulation, ce modèle existe toujours.

AVION

id_avion	type	compagnie
20	B747	British Airways
30	B747	Qantas

Table 4.16 – La relation AVION

MODÈLE

type	constructeur	capacité
A340	Airbus	228
B747	Boeing	432

Table 4.17 – La relation MODÈLE

AVION $\bowtie_{ext}$ MODÈLE

id_avion	type	compagnie	constructeur	capacité
20	B747	British Airways	Boeing	432
30	B747	Qantas	Boeing	432
$\perp$	A340	$\perp$	Airbus	228

Table 4.18 – Jointure extérieure entre AVION et MODÈLE

$\diamondsuit$

4.2 Combinaison d'opérations

Les opérations présentées dans la section 4.1 peuvent être combinées afin de pouvoir répondre à des requêtes relativement complexes.

Exemple 4.9 : Quelle est la capacité des avions de Qantas ?

Pour répondre à cette question, nous analysons ce qui est demandé. Tout d'abord, on ne s'intéresse qu'aux avions (table 4.19) de la compagnie Qantas. Ces avions sont obtenus par sélection : $r_1 = \sigma_{\texttt{compagnie}=\text{Qantas}}(\texttt{AVION})$ (table 4.20).

AVION

id_avion	type	compagnie
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

TABLE 4.19 – La relation AVION

r_1

id_avion	type	compagnie
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

TABLE 4.20 – Les avions de Qantas

La capacité que l'on cherche est décrite dans le modèle d'avion. Ce dernier étant caractérisé par son type, nous extrayons d'abord dans r_1 les types des avions de Qantas en utilisant une projection : $r_2 = \Pi_{\texttt{type}} r_1$ (table 4.21). Ensuite, nous allons utiliser la relation MODÈLE (table 4.22)pour retrouver la capacité des avions.

r_2

id_avion	type	compagnie
10	A340	Air France
20	B747	British Airways
30	B747	Qantas

TABLE 4.21 – Le type des avions de Qantas

MODÈLE

type	constructeur	capacité
A340	Airbus	228
B747	Boeing	432

TABLE 4.22 – Les modèles d'avions

Pour cela, on fait une jointure naturelle entre les relations r_2 et MODÈLE afin de ne garder que les modèles correspondant à des avions de la compagnie Qantas : $r_3 = r_2 \bowtie \texttt{MODÈLE}$ (table 4.23). Pour finir, on en extrait la capacité à l'aide d'une projection : $r_4 = \Pi_{\texttt{capacité}}(r_3)$ (table 4.24).

r_3

type	constructeur	capacité
B747	Boeing	432

TABLE 4.23 – Les modèles des avions de Qantas

r_4

type	constructeur	capacité
B747	Boeing	432

TABLE 4.24 – La capacité des avions de Qantas

◊

Chapitre 5

SQL

5.1 Introduction

Le langage SQL est à la fois :
- un *langage déclaratif* permettant de :
  - créer, modifier et interroger une base de données relationnelle ;
  - contrôler la sécurité et l'intégrité de la base ;
- un *langage relationnel* qui manipule des tables et produit des tables comme résultat.

La caractéristique du langage déclaratif est qu'il permet de décrire ce que l'on souhaite obtenir sans détailler les moyens de l'obtenir (par opposition à un langage procédural type langage C qui impose de décrire en détail toutes les actions nécessaires).

Le langage SQL se décompose en 3 familles d'opérations :
- *LDD* (*Langage de Définition des Données*) : description de la structure de la base de données (tables, attributs) ;
- *LMD* (*Langage de Manipulation de Données*) : manipulation des tables ;
- *LCD* (*Langage de Contrôle des Données*) : gestion du contrôle et de la sécurité de la base de données.

5.2 Création d'une table

La création d'une table s'effectue avec son nom et précise les attributs (colonnes) avec leur type et les contraintes éventuelles qui lui sont associées. Il peut également y avoir une contrainte portant sur la table elle même.

```
CREATE TABLE nomTable (
    nomColonne type contrainte_colonne,
    ...,
    contrainte_table,
    ...
);
```

Unes `contrainte_colonne` peut être :
- `NOT NULL` : l'attribut ne peut pas prendre la valeur nulle ;
- `UNIQUE` : la valeur de l'attribut doit être unique dans la table ;
- `PRIMARY KEY` : l'attribut est une clé primaire (donc non nul et unique) :
- `DEFAULT` *valeur* : précise une valeur par défaut pour cet attribut ;
- `CHECK` *condition* : l'attribut doit impérativement satisfaire la condition.

Une contrainte de table fait indique une clé étrangère en faisant référence à la table dans laquelle elle est clé primaire :

```
FOREIGN KEY (référenceColonne) REFERENCES référenceTable
```

Exemple 5.1 : Le script SQL suivant crée les tables `Pilote`,, `Vol` et `Effectue`.

```
create Table Pilote(
    id_pilote smallint primary key,
    nom varchar(20),
    adresse varchar(30)
);
create Table Vol(
    id_vol varchar(10) primary key,
    ville_depart varchar(20),
    ville_arrivee varchar(20),
    heure_depart time,
    heure_arrivee time
);
create Table Effectue(
    id_pilote smallint,
    id_vol varchar(10),
    date_vol Date,
    foreign key (id_pilote) references Pilote,
    foreign key (id_vol) references Vol,
    primary key (id_pilote,id_vol,date_vol)
);
```

◇

Exemple 5.2 : Une autre manière de créer la table `Effectue`, inclut la référence d'une clé étrangère à la table dont elle est issue directement dans la déclaration de l'attribut.

```
create Table Effectue(
    id_pilote smallint references Pilote,
    id_vol varchar(10) references Vol,
    date_vol Date,
    primary key(id_pilote,id_vol,date_vol)
);
```

◇

Les *contraintes* permettent d'exprimer des *conditions devant être respectées* par tous les tuples d'une table.

Exemple 5.3 : On peut vouloir préciser qu'un avion transporte des passagers en exprimant une contrainte sur sa capacité.

```
create Table Modele(
    type varchar(20) primary key,
    capacite smallint check(capacite>0),
    constructeur varchar(30) not null
);
```

◇

Dès la création d'une table, on peut préciser le comportement attendu quand des dépendances entre tables existent. Les clauses `ON UPDATE` et `ON DELETE` permettent d'indiquer les répercussions à effectuer lors de mises à jour ou de suppressions de tables ou de clés primaires :

- `NO ACTION` : les clauses `UPDATE` et `DELETE` ne sont pas exécutées pour que l'intégrité référentielle soit préservée ;
- `CASCADE` : toutes les clés étrangères sont mises à jour lors de la modification de la clé primaire et tous les enregistrements ayant la clé étrangère sont supprimés lors de la suppression de la clé primaire ;
- `SET NULL`, `SET DEFAULT` : la clé étrangère prend la valeur `NULL` (ou la valeur par défaut) lors de la modification ou la suppression de la clé primaire.

Exemple 5.4 : Il y a, dans la table `Effectue`, deux clés étrangères. Lorsqu'un pilote ou un vol est supprimé, la relation `Effectue` devient caduque et doit également être supprimée. Ceci se traduit par la déclaration `on delete cascade`.

```
create Table Effectue(
    id_pilote smallint,
    id_vol varchar(10),
    date_vol Date,
    foreign key (id_pilote) references Pilote on delete cascade,
    foreign key (id_vol) references Vol on delete cascade,
    primary key (id_pilote,id_vol,date_vol)
);
```

◊

5.3 Suppression et modification de tables

La suppression d'une table s'effectue avec la clause `DROP TABLE`. La modification d'une table est faite par la clause `ALTER TABLE`. Ces deux opérations peuvent avoir également un effet sur d'autres tables pour lesquelles une dépendance est déclarée.

```
DROP TABLE nom_table;}
ALTER TABLE nom_table SET...;}
```

5.4 Manipulation du contenu d'une table

Après avoir créé les tables d'une base de données, on peut y insérer des tuples, les mettre à jour ou les supprimer. Ils constituent le contenu de la base de données.

5.4.1 Insertion de Tuples

Le schéma général de la clause `INSERT` permettant l'insertion de tuples est le suivant. Il précise le nom de la table dans laquelle l'insertion se fait, ainsi que tout ou partie des valeurs contenues dans le tuple.

```
INSERT INTO table [(column [,...])] {
    DEFAULT VALUES | VALUES (expression [,...])}
```

Exemple 5.5 : Le listing suivant présente l'insertion des avions dans la table `Avion`. Dans les lignes 1, 3 et 6, les valeurs des trois attributs sont explicitement fournies. Il en est de même à la ligne 5 où le troisième attribut prend une valeur nulle. Les lignes 2 et 4 indiquent le nom et l'ordre des attributs dont les valeurs sont fournies.

```
1 insert into Avion values (2,'B707','Qantas');
2 insert into Avion (id_avion,type) values (3,'B737');
3 insert into Avion values (4,'A320','Air France');
4 insert into Avion (id_avion,type) values (5,'A320');
5 insert into Avion values (6,'A320',null);
6 insert into Avion values (8,'B737','Air France');
```

Pour vérifier le contenu de la table `Avion`, on utilise la clause `SELECT` (voir section 5.5.1).

```
select * from Avion;

 id_avion | type |  compagnie
----------+------+-----------
        2 | B707 | Qantas
        3 | B737 |
        4 | A320 | Air France
        5 | A320 |
        6 | A320 |
        8 | B737 | Air France
(6 rows)
```

5.4.2 Mise à Jour de Tuples

La clause `UPDATE TABLE` permet de modifier les tuples d'une table en précisant les attributs qui doivent être modifiés ainsi que leur nouvelle valeur, et les conditions qui doivent être satisfaites par les tuples sur lesquels appliquer la modification.

```
UPDATE table SET col=expression [,...]
    [WHERE condition]
```

Exemple 5.6 : La modification suivante s'applique au vol `BA302`. Elle change la date du vol.

```
update Vol set date_vol='07/02/2004'
    where Vol.id_vol='BA302';
```

5.4.3 Suppression de Tuples

La clause `DELETE FROM` permet de supprimer les tuples d'une table en précisant les conditions qu'ils doivent satisfaire.

```
DELETE FROM table [WHERE condition]
```

Exemple 5.7 : Le script suivant supprime l'avion numéro 6.

```
delete from Avion where Avion.id_avion=6;
```

5.5 Interrogation de la base de données

Les programmes travaillant avec des bases de données sont également amenés à en extraire de informations. Le chapitre 4 a montré comment modéliser des requêtes. Dans cette section, les éléments du langage SQL mettant en œvre ces opérateurs sont présentés, ainsi que des fonctions plus avancées.

5.5.1 La clause SELECT

La clause `SELECT` est essentielle pour faire des requêtes sur une base des données. Sa forme générale est :

```
SELECT [ALL | DISTINCT [ON (expression [,...])]]
    * | expression [AS nom_sortie] [,...]
    [FROM table [,...]]
    [WHERE condition]
```

Les sections suivantes montrent différents aspects de l'utilisation de `SELECT`.

5.5.2 Réalisation d'une Projection

Pour faire une projection, on utilise la clause `SELECT` en désignant les attributs sur lesquels la projection est effectuée. Contrairement à ce qui se passe en algèbre relationnelle, on obtient plusieurs fois la même ligne si les mêmes attributs figurent en plusieurs exemplaires dans la projection.

Exemple 5.8 : Quels sont les différents types d'avions ?
Il faut sélectionner le `type` des avions dans la table `Avion`.

```
select Avion.type from Avion;

     type
    ------
     B707
     B737
     A320
     A320
     A320
     B737
```

L'attribut `type` de la table `Avion` est désigné par `Avion.type`. On peut néanmoins utiliser simplement le nom de l'attribut (sans le nom de la table) s'il n'y a pas d'ambiguïté. ♢

L'utilisation du mot-clé `DISTINCT` élimine les doublons, comme l'opération de projection en algèbre relationnelle.

Exemple 5.9 : Complétons l'exemple 5.8 en éliminant les doublons. Ceci correspond à $\Pi_{type}(Avion)$.

```
select distinct Avion.type from Avion;

     type
    ------
     B707
     B737
     A320
```

♢

5.5.3 Opération de sélection

Dans un `SELECT`, la clause `WHERE` permet de spécifier *un critère de sélection*. Celui-ci est un prédicat, c'est-à-dire une expression logique composée d'une suite de conditions combinées par les opérateurs logiques `AND`, `OR` ou `NOT`. Seuls les tuples satisfaisant le prédicat font partie du résultat.

Un élément d'une expression peut prendre une des formes suivantes :

- comparaison à une valeur : `=`, `!=`, `<>`, `<`, `>`, `<=`, `>=` ;
- comparaison à une fourchette de valeurs : `between` ;
- comparaison à une liste de valeurs : `in` ;
- comparaison à un filtre : `like`, `~` ;
- test sur l'indétermination d'une valeur : `is null` ;
- comparaison à tous les éléments d'un ensemble : `all` ;
- comparaison à au moins un élément d'un ensemble : `any`.

Exemple 5.10 : Quels sont les vols partant entre 14h et 18h ?
Supposons que la table des vols soit la suivante :

```
     id_vol | ville_depart | ville_arrivee | heure_depart | heure_arrivee
    -------------------------------------------------------------------------
     AF1232 |    Paris     |   Singapour   |   15:30:00   |   22:00:00
     BA302  |   Londres    |    Sydney     |   09:15:00   |   16:54:00
     QT17   |   Sydney     |    Perth      |   14:30:00   |   15:30:00
```

Le script SQL suivant sélectionne les tuples de la table `Vol` dont l'attribut `heure_depart` est compris entre 14 et 18.

```
select Vol.* from Vol where Vol.heure_depart between 14 and 18;

     id_vol | ville_depart | ville_arrivee | heure_depart | heure_arrivee
    -----------------------------------------------------------------------
     AF1232 |    Paris     |   Singapour   |   15:30:00   |   22:00:00
     QT17   |    Sydney    |     Perth     |   14:30:00   |   15:30:00
```

◊

Exemple 5.11 : La commande des lignes 1–2 sélectionne les vols pour lesquels `ville_depart` est dans la liste `('Londres','Paris')`. Celle de la ligne 4 sélectionne les pilotes dont le nom commence par D. Enfin, à la ligne 6, sont sélectionnés les pilotes dont l'adresse n'est pas renseignée.

```
1  select Vol.* from Vol
2      where Vol.ville_depart in ('Londres','Paris');
3
4  select Pilote.* from Pilote where Pilote.nom_pilote ~'^D';
5
6  select Pilote.* from Pilote where Pilote.adresse is null;
```

◊

5.5.4 Tri des Tuples

Dans le cas général, les tuples résultats sont présentés dans l'ordre dans lequel ils ont été trouvés dans la base de données (par exemple l'ordre dans lequel ils ont été insérés). Toutefois, il est souvent souhaitable de présenter un résultat final trié.

Pour *trier le résultat*, on utilise la clause `ORDER BY` suivie éventuellement de `ASC` (tri ascendant) ou `DESC` (tri descendant).

Exemple 5.12 : Quels sont les avions n'appartenant pas à la compagnie Qantas, triés par numéro d'avion décroissant ?

```
select Avion.* from Avion
    where Avion.compagnie!='Qantas'
    order by Avion.id_avion desc;
```

◊

5.5.5 Jointure

La clause `SELECT` peut également être utilisée pour effectuer une *jointure* (voir section 4.1.7). Il faut alors :

- citer les attributs recherchés dans la clause `SELECT` ;
- lister dans la clause `FROM` les tables concernées par la jointure ;
- préciser dans la clause `WHERE` la *condition* portant sur les attributs sur lesquels la jointure est faite ;
- préciser dans la clause `WHERE` les éventuelles *conditions* particulières à la requête.

Exemple 5.13 : Quels sont les noms des pilotes qui ont assuré le vol Londres–Paris de 09 :15 ?

Dans la commande SQL suivante, l'attribut recherché est `Pilote.nom` (ligne 1). Ligne 2 sont indiquées les tables nécessaires : `Pilote`, `Effectue` et `Vol`. Les conditions sur les attributs qui expriment la jointure se trouvent lignes 3–4. Enfin les conditions sur les villes de départ, d'arrivée et l'horaire se trouvent aux lignes 5–7.

```
1 select Pilote.nom
2     from Pilote,Effectue,Vol
3     where Pilote.id_pilote=Effectue.id_pilote
4         and Effectue.id_vol=Vol.id_vol
5         and Vol.ville_depart='Londres'
6         and Vol.ville_arrivee='Paris'
7         and Vol.heure_depart='09:15:00';
```

◊

L'utilisation de la jointure naturelle (clause `NATURAL JOIN`) évite de spécifier les attributs sur lesquels la jointure est effectuée : SQL choisit automatiquement dans les tables les attributs de même nom pour effectuer la jointure.

Exemple 5.14 : Quels sont les différentes combinaisons d'avions et de pilotes utilisées sur les vols ?

On sélectionne les attributs `nom_pilote` de la table `Pilote` et `id_vol` de la table `Vol`. Les combinaisons de ces deux attributs sont celles présentes dans la jointure naturelle `Pilote` $\bowtie$ `Effectue` $\bowtie$ `Vol`.

```
select distinct Pilote.nom_pilote,Vol.id_vol
    from Pilote natural join Effectue natural join Vol;

    nom_pilote | id_vol
    -----------+-----------
     Dupond    |     4
     Garrat    |     2
     Garrat    |     20
     MacMachin |     20
     Smith     |     2
```

◊

Les clauses `LEFT OUTER`, `RIGHT OUTER` et `FULL OUTER` implémentent des jointures extérieures. Sont conservés les tuples qui ne vérifient pas la condition pour l'une ou l'autre table.

Exemple 5.15 : Quels sont les pilotes n'effectuant aucun vol ?

Sélectionnons d'abord tous les pilotes et leurs vols même s'ils n'ont pas effectué de vol :

```
select Pilote.nom_pilote,Effectue.id_vol from Pilote
    left outer join Effectue
        on Pilote.id_pilote=Effectue.id_pilote;

    nom_pilote | id_vol
    -----------+---------
     Dupond    | AF1232
     Dupond    | AF2321
     Smith     | QT71
     Garrat    | QT17
     Garrat    | BA302
     Durand    |
     MacMachin | BA203
```

Complétons cette commande pour ne grader que les pilotes associés à aucun vol :

```
select Pilote.nom_pilote,Effectue.id_vol from Pilote
    left outer join Effectue
        on Pilote.id_pilote=Effectue.id_pilote
    where id_vol is null;

    nom_pilote
    ----------
     Durand
```

◊

L'auto-jointure permet de traiter une requête comportant un critère comparant la valeur d'un attribut avec celle du même attribut dans un autre tuple de la même table. Pour distinguer les deux versions de la table, on utilise des *alias* définis grâce au mot-clé `AS`.

Exemple 5.16 : Quels sont les numéros des vols dont l'heure de départ est après celle du vol Sydney-Perth ?

En analysant cette question, on s'aperçoit qu'il faut considérer deux types de vols : celui qui fait Sydney-Perth et ceux qui lui sont comparés pour faire partie du résultat. Par conséquent, on travaille sur deux copies de la table `Vol`, appelées respectivement `SP` et `tard`.

```
select tard.id_vol,tard.heure_dep,SP.heure_dep
    from Vol as tard join Vol as SP
        on tard.heure_dep>SP.heure_dep
    where SP.ville_dep='Sydney' and SP.ville_arr='Perth';

     id_vol | heure_dep | heure_dep
    ---------+-----------+-----------
     AF1232   | 15:30:00   | 14:30:00
     BA203    | 15:15:00   | 14:30:00
     AF2321   | 16:30:00   | 14:30:00
```

◊

5.5.6 Union

L'opérateur `UNION` effectue l'union des résultats de deux requêtes `SELECT` en éliminant les doublons parmi les tuples. Les attributs sélectionnés dans les deux `SELECT` doivent être identiques. Ceci correspond à la définition 18 qui requiert que les deux relations aient le même schéma.

Exemple 5.17 : Quels sont les avions d'Air France assurant des vols pour Paris et ceux de British Airways assurant des vols pour Londres ?

Un premier `SELECT` (lignes 1–4) trouve les avions d'Air France assurant des vols pour Paris, tandis que le second (lignes 6–9) trouve ceux de British Airways assurant des vols pour Londres. Le résultat demandé est l'union de ces deux résultats (ligne 5).

```
1 select Avion.* from Avion,Vol
2     where Avion.id_avion=Vol.id_avion
3         and Avion.compagnie='Air France'
4          and Vol.ville_arrivee='Paris'
5 union
6 select Avion.* from Avion,Vol
7     where Avion.id_avion=Vol.id_avion
8         and Avion.compagnie='British Airways'
9         and Vol.ville_arrivee='Londres';
```

◊

5.5.7 Intersection

L'opérateur `INTERSECT` effectue l'intersection des résultats de deux requêtes `SELECT`. On obtient une table contenant les tuples communs aux deux tables de départ. Les attributs sélectionnés dans les deux `SELECT` doivent être identiques.

Exemple 5.18 : Quels sont les avions assurant à la fois des vols pour Londres et pour Paris ?

Un premier `SELECT` (lignes 1–3) trouve les avions assurant des vols pour Londres, tandis que le second (lignes 5–7) trouve ceux assurant des vols pour Paris. Le résultat demandé est l'intersection de ces deux résultats (ligne 4).

```sql
select Avion.* from Avion,Vol
    where Avion.id_avion=Vol.id_avion
        and Vol.ville_arrivee='Londres'
intersect
select Avion.* from Avion,Vol
    where Avion.id_avion=Vol.id_avion
        and Vol.ville_arrivee='Paris';
```

◊

5.5.8 Différence

L'opérateur `EXCEPT` effectue la différence des résultats de deux requêtes `SELECT`. On obtient une table contenant les tuples de la première requête qui n'apparaissent pas dans la seconde. Les attributs sélectionnés dans les deux `SELECT` doivent être identiques.

Exemple 5.19 : Quels sont les avions assurant des vols pour Londres mais pas pour Paris ?
Les deux `SELECT` sont identiques à ceux de l'exemple 5.18, mais le résultat est obtenu en faisant la différence des deux à la ligne 4.

```sql
select Avion.* from Avion,Vol
    where Avion.id_avion=Vol.id_avion
        and Vol.ville_arrivee='Londres'
except
select Avion.* from Avion,Vol
    where Avion.id_avion=Vol.id_avion
        and Vol.ville_arrivee='Paris';
```

◊

5.5.9 Les comparateurs ALL et ANY

Les comparateurs `ALL` et `ANY` permettent de comparer des éléments à ceux de la table résultant d'une requête. Cette requête `SELECT` se place dans la clause `WHERE` avec un opérateur de comparaison suivi de `ALL` ou `ANY`, utilisé comme suit :

- `ALL` : la condition est vraie si et seulement si elle est vraie pour toutes les valeurs produites ;
- `ANY` : la condition est vraie si et seulement si elle est vraie pour au moins une valeur produite.

Exemple 5.20 : Quels sont les types d'avions du constructeur Boeing dont la capacité est supérieure à celle d'au moins un avion du constructeur Airbus ?
Le `SELECT` des lignes 4–5 trouve les capacités des avions du constructeur Airbus. La ligne 3 effectue la comparaison au résultat de cette requête.

```sql
select distinct Avion.type from Avion
    where Avion.constructeur='Boeing'
        and Avion.capacite > ANY
            (select Avion.capacite from Avion
                where Avion.constructeur='Airbus');
```

◊

Exemple 5.21 : Quels sont les types d'avions du constructeur Boeing dont la capacité est supérieure à celle de tous les avions du constructeur Airbus ?
La requête est similaire à celle de l'exemple 5.20, à part l'opérateur de comparaison ligne 3.

```sql
select Avion.type from Avion
    where Avion.constructeur='Boeing'
        and Avion.capacite > ALL
           (select Avion.capacite from Avion
               where Avion.constructeur='Airbus');
```

◊

5.5.10 Test d'existence

La clause `EXISTS` permet d'exprimer une condition qui est satisfaite si la requête imbriquée renvoie au moins un tuple.

```
... WHERE EXISTS (SELECT ...);
```

Exemple 5.22 : Quels sont les avions qui assurent au moins un vol pour Londres ?
Pour chaque avion (ligne 2), la requête imbriquée (lignes 3–5) sélectionne les vols qu'il effectue (jointure réalisée par la ligne 4) à destination de Londres. Au final, ne sont retenus que les avions pour lesquels cette requête imbriquée renvoie un résultat.

```
1 select *
2     from Avion
3     where exists (select * from Vol
4                      where Vol.id_avion=Avion.id_avion
5                           and Vol.ville_arrivee='Londres');
```

◇

Exemple 5.23 : Quels sont les avions qui n'assurent pas de vol pour Londres ?
La requête est similaire à celle de l'exemple 5.22, sauf que l'on teste la non-existence du résultat de la requête imbriquée avec `NOT EXISTS`.

```
select *
    from Avion
    where not exists (select * from Vol
                          where Vol.id_avion=Avion.id_avion
                              and Vol.ville_arrivee='Londres');
```

◇

5.5.11 Fonctions sur les Attributs

Plusieurs fonctions permettent de traiter les chaînes de caractères :

- `CHAR_LENGTH(chaîne)` : renvoie la longueur de `chaîne` ;
- `POSITION(chaîne IN source)` : cherche la `chaîne` de caractères dans la chaîne `source`. Si elle est trouvée, sa position est retournée, sinon 0 est renvoyé ;
- `SUBSTRING(source FROM début FOR longueur)` : extrait la sous-chaîne de `source` commençant à la position `début` et ayant `longueur` caractères ;
- `UPPER(chaîne)` : convertit la `chaîne` en majuscules ;
- `LOWER(chaîne)` : convertit la `chaîne` en minuscules ;
- `EXTRACT(élément FROM source)` : extrait un `élément` d'une chaîne `source` telle que date ou heure ;
- `chaîne1 || chaîne2` : concaténation des deux chaînes de caractères.

Exemple 5.24 : Le script suivant affiche les noms et prénoms des pilotes respectivement en majuscule et minuscules.

```
select Pilote.nom, Pilote.prenom,
    upper(nom)||' '||lower(prenom) as nom_prenom
    from Pilote
    order by nom_prenom asc;

        nom     |  prenom   |    nom_prenom
    -----------+----------+----------------
     Dupond     | Laurent   | DUPOND laurent
     Durand     | Frédéric  | DURAND frédéric
     Garrat     | Jo        | GARRAT jo
     MacMachin  | Allan     | MACMACHIN allan
     Smith      | Wendy     | SMITH wendy
```

Exemple 5.25 : Le script suivant affiche pour chaque avion sa `compagnie`, la position de la sous-chaîne `an` dans le nom de la compagnie, la longueur de ce nom, et la sous-chaîne de 3 caractères à partir du cinquième dans ce nom.

```
select Avion.compagnie,
    position('an' in Avion.compagnie),
    char_length(Avion.compagnie),
    substring(Avion.compagnie from 5 for 3)
    from Avion;

        compagnie     | position | char_length | substring
    -----------------+----------+-------------+-----------
     Qantas          |        2 |           6 | as
     Air France      |        7 |          10 | Fra
     Air France      |        7 |          10 | Fra
     British Airways |        0 |          15 | ish
     Qantas          |        2 |           6 | as
     Air France      |        7 |          10 | Fra
```

De nombreuses fonctions sont disponibles. Leur liste peut-être obtenue (dans `psql`) en tapant `\df`.

Exemple 5.26 : Ce script utilise une racine carrée (pour montrer l'utilisation de fonctions numériques).

```
select Avion.id_avion,sqrt(float8(capacite))
    from Avion where Avion.capacite is not null;

     id_avion |       sqrt
    ----------+------------------
            2 | 12.2474487139159
            4 | 21.2132034355964
            8 | 18.7082869338697
           20 | 20.7846096908265
           30 | 20.7846096908265
           10 | 15.0996688705415
```

5.6 Groupes

5.6.1 Détermination des Groupes

Les groupes permettent de traiter un ensemble de tuples ayant des caractéristiques communes :

- un *groupe* est un sous-ensemble des tuples d'une table ayant la *même valeur pour un attribut* ;
- un groupe est déterminé par la clause `GROUP BY` suivie du *nom de l'attribut* sur lequel s'effectue le regroupement ;
- la clause `GROUP BY` réarrange la table résultat d'un `SELECT` par groupes ;
- lorsqu'une clause `GROUP BY` est précisée, on peut utiliser *des fonctions portant sur les groupes*.

5.6.2 Fonctions sur les Groupes

Les principales fonctions sur les groupes sont :

- `COUNT` : compte le *nombre d'occurrences* de l'attribut ;

- `SUM` : calcule la *somme des valeurs* de l'attribut ;
- `AVG` : calcule la *moyenne des valeurs* de l'attribut ;
- `MAX` : recherche la *plus grande valeur* de l'attribut ;
- `MIN` : recherche la *plus petite valeur* de l'attribut.

Exemple 5.27 : Combien y a-t-il d'avions ?
On constitue un groupe avec tous les avions et on les compte :

```
select count(*) from avion;

  count
 -------
      7
```

◊

Exemple 5.28 : Combien y a-t-il d'avions de chaque type ?
On groupe les avions par type et on les compte :

```
select type,count(*) from avion group by type;

  type | count
 ------+-------
  A320 |     2
  A340 |     1
  B707 |     1
  B737 |     1
  B747 |     2
```

◊

Exemple 5.29 : Quelle est la capacité moyenne des avions ?
On groupe les avions et on fait la moyenne de l'attribut `capacité` sur le groupe :

```
select avg(capacite) from avion;

       avg
 ----------------
  340.3333333333
```

◊

5.6.3 Clause HAVING

La clause `HAVING` est l'équivalent du `WHERE` appliqué aux groupes. Le critère spécifié dans la clause `HAVING` porte sur la valeur d'une fonction calculée sur un groupe.

Exemple 5.30 : Quelles sont les compagnies possédant au moins deux avions ?
On groupe les avions par compagnie, on compte le nombre d'avions pour chacune et on ne retient que celles qui en ont au moins deux :

```
select compagnie, count(*) from avion
    group by compagnie having count(*)>=2;

  compagnie   | count
 -------------+-------
  Air France  |     3
  Qantas      |     2
```

◊

5.7 Vues

Une *vue* est le nom donné à une requête. L'utilisation de *vues* permet de donner de la base de données une vision adaptée à l'utilisateur (en évitant par exemple de faire apparaître des données sensibles).

Une vue est *dynamique* : c'est une sorte de table virtuelle. Une fois créée, la vue est accessible comme toute autre table.

5.7.1 Création d'une Vue

La création d'une vue s'effectue à l'aide de la clause `CREATE VIEW`.

Exemple 5.31 : Créer une vue de nom `AF2Paris` des avions de la compagnie Air France ayant Paris pour destination.

```
create view AF2Paris as
    select avion.id_avion,avion.type,avion.constructeur,avion.capacite,vol.id_vol
        from avion,vol
        where avion.compagnie='Air France'
            and avion.id_avion=vol.id_avion
            and vol.ville_arr='Paris';
```

◊

5.7.2 Utilisation d'une vue

Les requêtes s'effectuent comme sur les tables.

Exemple 5.32 : Le script suivant affiche la vue `AF2Paris` :

```
select * from AF2Paris;

     id_avion | type | constructeur | capacite | id_vol
    ----------+------+--------------+----------+---------
            4 | A320 | Airbus       |      450 | AF2321
```

◊

Il ne peut y avoir *ni insertion ni suppression* directes, car la vue est définie par une requête. Ce n'est donc pas une table à part entière. Pour contourner cette difficulté, on peut utiliser une *règle* (voir section 5.8).

5.7.3 Suppression d'une Vue

Une vue peut être supprimée avec le clause `DROP VIEW` :

```
DROP VIEW name [ CASCADE | RESTRICT ]
```

Elle permet deux modalités :

- `CASCADE` : si la vue est supprimée, toutes les vues et contraintes où la vue intervient seront supprimées ;
- `RESTRICT` : si la vue intervient dans la définition d'une autre vue ou dans une contrainte d'intégrité, la commande est rejetée.

Exemple 5.33 : On supprime la vue `AF2Paris` :

```
drop view AF2Paris;
```

◊

5.8 Règles

Les *règles* permettent de spécifier quelle `action` doit être effectuée lors de la *réception d'un événement* donné. La syntaxe générale de création d'une règle est :

```
CREATE RULE nom_règle AS ON événement TO objet DO action
```

L'`objet` peut être une table ou une vue éventuellement assortie d'une condition dans une clause `WHERE`. L'`événement` est une action sur l'objet comme une insertion, une suppression ou une mise à jour. L'`action` peut être :

- une ou plusieurs requêtes ;
- ne rien faire : clause `NOTHING` ;
- ce qu'il faut faire à la place : clause `INSTEAD action`.

Exemple 5.34 : Lors de l'insertion dans la vue `AF2Paris`, créer les tuples idoines dans `Avion` et `Vol`.

Lors de cette insertion, on fait à la place (clause `INSTEAD`) d'autres actions : des insertions dans les tables `Avion` et `Vol` des valeurs idoines. Celles-ci font en particulier référence au nouvel ensemble de valeurs que l'on souhaite insérer (identfié par `new`).

```
create rule ins_af2paris as on insert to AF2Paris
    do instead (
           insert into Avion
               values(new.id_avion,new.type,new.constructeur,
                      new.capacite,'Air France');
           insert into Vol(id_vol,id_avion,ville_arr)
               values(new.id_vol,new.id_avion,'Paris'));
```

◊

Chapitre 6

Client C d'un SGBD

6.1 Objectif

Le but et maintenant de rédiger et implémenter une *API* (*Application Programming Interface*) permettant aux applications :

- d'*accéder* aux bases de données ;
- d'*exécuter* des requêtes SQL ;
- de *récupérer* puis *traiter les résultats* de ces requêtes.

6.2 La librairie `libpq-fe.h`

Dans un programme C, il faut inclure la librairie `libpq-fe.h` pour pouvoir accéder aux fonctions de `postgresql` :

```
#include </usr/include/postgresql-8.3/libpq-fe.h>
```

La compilation utilise la librairie `pq` :

```
gcc -lpq . . .
```

Les caractéristiques, fonctions et types principaux pour la gestion de la base de données à partir d'un programme C sont les suivants :

- *plusieurs connexions* à des bases de données peuvent avoir lieu simultanément. Chacune est identifiée par une variable de type `PGconn*` ;
- une connexion à la base de type `PGconn*` s'obtient en appelant la fonction `PQconnectdb` ;
- pour exécuter une requête, on utilise la fonction `PQexec` avec la connexion et la requête SQL en paramètre ;
- la fonction `PQexec` retourne un résultat de type `PGresult*`.
- une variable de type `PGresult*` pointe vers un tableau à deux dimensions contenant le résultat de la requête.
- plusieurs fonctions permettent d'examiner le résultat de type `PGresult*` d'une requête :
  - `PQnfields` renvoie son nombre d'attributs ;
  - `PQntuples` renvoie son nombre de tuples ;
  - `PQfname` renvoie le nom d'un de ses attributs ;
  - `PQgetvalue` permet d'obtenir le contenu d'une de ses cases.

6.3 Un exemple complet

Le programme suivant affiche la table `Avion`. Il commence par inclure la librairie définissant les primitives d'accès à une base postgresql (ligne 1). Aux lignes 3 et 4 sont déclarées les variables

nécessaires pour récupérer respectivement une connexion et un résultat de requête. Les lignes 9–14 établissent une connexion et vérifient qu'elle est bien établie. Ensuite, le texte de la requête est construit ligne 16 avant d'être éxécutée à la ligne 17. Le nombre d'attributs du résultat est récupéré à la ligne 19 et son nombre de tuples à la ligne 21. Ensuite, une boucle (lignes 24–26) parcourt et affiche les noms des attributs de la table, puis une seconde boucle (lignes 29–33) affiche son contenu. Lorsque l'on a terminé, on peut libérer la mémoire occupée par les variables contenant le résultat d'une requête avec la commande `PQclear` (ligne 35). Enfin, `PQfinish` ferme l'accès à la base de données à la ligne 36.

```
1  #include </usr/include/postgresql-8.3/libpq-fe.h>
2  int main(){
3      PGconn* connection;
4      PGresult* resultat;
5      char requete[200];
6      int nbcols, nbligs, field, row;
7
8      /* début : connection */
9      connection=PQconnectdb("host=aquanux dbname=mabase
10                         user=utilisateur password=motdepasse");
11     if (PQstatus(connection)==CONNECTION_BAD){
12         perror("Problème de connection\n");
13         exit(1);
14     }
15     /* requête */
16     strcpy(requete,"select * from Avion");
17     resultat=PQexec(connection,requete);
18     /* nombre d'attributs */
19     nbcols=PQnfields(resultat);
20     /* nombre de tuples */
21     nbligs=PQntuples(resultat);
22
23     /* écriture des noms des attributs de la table */
24     for(field=0;field<nbcols;field++)
25         printf("%s,",PQfname(resultat,field));
26     printf("\n");
27
28     /* écriture du contenu de la table */
29     for(row=0;row<nbligs;row++){
30         for(field=0;field<nbcols;field++)
31             printf("%s,",PQgetvalue(resultat,row,field));
32         printf("\n");
33     }
34     /* fin */
35     PQclear(resultat);
36     PQfinish(connection);
37 }
```

Chapitre 7

Client java d'un SGBD

7.1 Objectif

Le but et maintenant de rédiger et implémenter une *API* (*Application Programming Interface*) permettant aux applications :

- d'*accéder* aux bases de données ;
- d'*exécuter* des requêtes SQL ;
- de *récupérer* puis *traiter les résultats* de ces requêtes.

7.2 Schéma d'une API java

Les différentes étapes de l'accès à une base de données par un programme java sont les suivantes :

- *charger un pilote* en mémoire ;
- *établir la connexion* à une base de données ;
- *créer une session* ;
- *exécuter des requêtes* SQL et exploiter les résultats ;
- *fermer la connexion*.

Le chargement du pilote en mémoire :

- charge dynamiquement la classe dont le nom est passé en paramètre ;
- crée une instance spécifique du pilote ;
- elle se place dans la liste des pilotes utilisables pour le `DriverManager` ;
- le `DriverManager`r parcourt la liste des pilotes et en utilise un pour ouvrir une connexion avec une base de données.

Chaque pilote *implémente les classes et interfaces de l'API* JDBC pour un SGBD particulier. Les applications java utilisent toujours les *mêmes méthodes*, quel que soit le SGBD cible.

7.3 Exemple complet

Le programme suivant implémente la classe de gestion d'une base de données. L'exemple d'utilisation fourni ensuite affiche la table `Avion`.

La classe `GestionBD` comprend d'abord un constructeur (lignes 9-15). Il charge le `pilote` demandé à la ligne 12, puis établit la connexion en mettant également en place un `Statement` pour gérer les requêtes. La classe `GestionBD` fournit des méthodes publiques :

- lignes 17-21 : fermeture de la connexion ;
- lignes 24-26 : exécution d'une requête ;
- lignes 30-38 : récupération des noms des attributs d'une table résultat ;
- lignes 40-64 : récupération des tuples résultats.

```java
import java.sql.*;
import java.util.ArrayList;

public class GestionBD {
    private Connection connexion;
    private Statement stmt;

    // constructeur : ouvre une connexion puis une session.
    public GestionBD(String url,String pilote,String gestionnaireBD,
            String motDePasse,PrintWriter out)
            throws SQLException,ClassNotFoundException {
        Class.forName(pilote);
        this.connexion = DriverManager.getConnection(url,gestionnaireBD,motDePasse);
        this.stmt = connexion.createStatement();
    }

    // fermeture de la connexion
    public void fermerConnexion() throws SQLException {
        this.stmt.close();
        this.connexion.close();
    }

    // exécution d'une requête
    public ResultSet executerRequete(String requete) throws SQLException {
        return(this.stmt.executeQuery(requete));
    }

    // retourne le tableau de String de la méta-structure
    // (les noms des attributs) du Resultset
    public String[] getTableauNomsAttributsResultSet(ResultSet rs)
            throws SQLException {
        ResultSetMetaData rsmd = rs.getMetaData();
        int nbCol = rsmd.getColumnCount(); // nombre de colonnes
        String[] tab = new String[nbCol];
        for (int i = 0; i < nbCol; i++)
            tab[i] = rsmd.getColumnName(i + 1);
        return(tab);
    }

    public ArrayList <String> getArrayListTuplesResultSet(ResultSet rs,
            String separateur1,boolean nomsAttributs)
            throws SQLException {
        String s;
        int i;

        ResultSetMetaData rsmd = rs.getMetaData();
        int nbCol = rsmd.getColumnCount(); // nombre de colonnes
        ArrayList <String> al = new ArrayList <String> ();
        if (nomsAttributs) {
            s = "";
            for (i = 0; i < nbCol - 1; i++)
                s = s + rsmd.getColumnName(i + 1) + separateur1;
            s = s + rsmd.getColumnName(i + 1);
            al.add(s);
        }
        while (rs.next()) {
            s = "";
            for (i = 0; i < nbCol - 1; i++)
                s = s + rs.getString(i + 1) + separateur1;
            s = s + rs.getString(i + 1);
            al.add(s);
        }
        return(al);
    }
} // end class
```

Utilisation : Le code suivant utilise cette classe de gestion de base de données pour afficher le contenu de la table `Avion`. Il appelle d'abord le constructeur `GestionBD` qui charge le pilote et établit la connexion. Ensuite, il exécute la requête, récupère le résultat dans un `ResultSet`, dont il extrait les attributs puis les affiche. Enfin, il extrait la liste des tuples du résultat et les affiche. Enfin, la connexion est fermée.

```
1       GestionBD gt = new GestionBD("jdbc:postgresql://aquanux/mon_login",
2           "org.postgresql.Driver","mon_login","mot_de_passe",pw);
3
4       ResultSet rs = gt.executerRequete("select * from avion");
5       String[] tab = gt.getTableauNomsAttributsResultSet(rs);
6       for (int i = 0; i < tab.length; i++)
7           System.out.println(tab[i]);
8       ArrayList <String> al = gt.getArrayListTuplesResultSet(rs,";",true);
9       for (int i = 0;i < al.size(); i++)
10          System.out.println(al.get(i));
11
12      gt.fermerConnexion();
```