

Sources de Données — Ressource R207

IUT de Villetaneuse — R&T 1^{re} année

Responsable du cours : Laure PETRUCCI

25 janvier 2022

Table des matières

1	Stockage et accès aux données	3
1.1	Systèmes de gestion de données	3
1.1.1	Structuration des données	3
1.1.2	Réglementation	4
1.2	Bases de données relationnelles	4
1.2.1	Intégration	5
1.2.2	Indépendance	5
1.2.3	Disponibilité	5
1.2.4	Sécurité	5
2	Conception d'une Base de Données	6
2.1	Cahier des charges	6
2.2	Diagramme de classes	6
2.2.1	Classes	6
2.2.2	Objets	7
2.2.3	Relations entre classes : associations	8
2.2.4	Classe-association	9
2.2.5	Diagramme de classes	10
2.2.6	Élaboration d'un diagramme de classes	10
2.3	Schéma relationnel	11
2.3.1	Relations	11
2.3.2	Instanciation d'une relation	12
2.3.3	Clé primaire	12
2.3.4	Contrainte d'intégrité et valeur nulle	12
2.3.5	Clé étrangère	12
2.3.6	Contrainte d'intégrité de référence	13
2.4	Du diagramme de classes au schéma relationnel	13
2.4.1	Classes	13
2.4.2	Associations	13
2.4.3	Association plusieurs à plusieurs	13
3	SQL	15
3.1	Introduction	15
3.2	Création d'une table	15
3.3	Suppression et modification de tables	17
3.4	Manipulation du contenu d'une table	17
3.4.1	Insertion de tuples	17
3.4.2	Mise à jour de tuples	18
3.4.3	Suppression de tuples	18
3.5	Interrogation de la base de données	18
3.5.1	La clause SELECT	18
3.5.2	Projection	18

3.5.3	Opération de sélection	19
3.5.4	Tri des Tuples	20
3.5.5	Jointure	21
3.5.6	Union	24
3.5.7	Intersection	25
3.5.8	Différence	26
3.5.9	Les comparateurs ALL et ANY	26
3.5.10	Test d'existence	27
3.5.11	Fonctions sur les Attributs	28
3.6	Groupes	29
3.6.1	Détermination des Groupes	29
3.6.2	Fonctions sur les Groupes	29
3.6.3	Clause HAVING	30
4	Utilisation d'un SGBD dans un programme	31
4.1	Objectif	31
4.2	Client C d'un SGBD	31
4.2.1	La librairie libpq-fe.h	31
4.2.2	Un exemple complet	32
4.3	Client java d'un SGBD	32
4.3.1	Schéma d'une API java	32
4.3.2	Exemple complet	33
4.4	Client Python d'un SGBD	34

Chapitre 1

Stockage et accès aux données

1.1 Systèmes de gestion de données

Les données utilisées par les différents services proposés à des utilisateurs, qu'elles soient manipulées par des programmes ou des services web, sont stockées dans des fichiers ou des bases de données.

Les modèles principaux sont *relationnels* ou *non-relationnels*. Dans le premier cas, les données sont stockées dans des tables partageant un minimum d'informations et reliées entre elles via ces informations partagées. Les *bases de données relationnelles* seront le principal objet de ce cours. Elles sont stockées sous forme de tables. Les modèles non relationnels comprennent les *fichiers* ainsi que les bases de données *NoSQL*, adaptées au stockage distribué de grandes masses de données. Ces dernières reposent sur un stockage (*clef, valeur*). Un exemple de système de gestion de données NoSQL est MongoDB. Le stockage dans des fichiers est plutôt dédié à des données de petite taille n'induisant que peu de duplication d'informations.

1.1.1 Structuration des données

Le stockage sous forme de fichiers peut néanmoins être structuré. Par exemple *CSV* (*Comma-Separated Values*) est un format de fichiers ouvert permettant de décrire des tableaux (utilisé par divers tableurs). Chaque ligne du tableau est une ligne du fichier, et les colonnes sont séparées par un caractère dédié (en général une virgule `,` ou un point-virgule `;`). Le texte peut être délimité par des guillemets `"..."`.

Exemple 1.1 : La figure 1.1 montre un fichier CSV et la figure 1.2 sa représentation sous forme de tableau.

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

FIGURE 1.1 – Un fichier CSV

type	constructeur	capacité
A340	Airbus	228
B747	Boeing	432

FIGURE 1.2 – Tableau correspondant au CSV

◇

Le format de données *JSON* (*JavaScript Object Notation*) est un format de données simple, largement utilisé par les serveurs et services web. Il comprend deux types de données : des paires (nom, valeur) et des listes, ainsi que quatre types de valeurs : les booléens, les nombres, les chaînes de caractères et la valeur nulle (`null`).

Exemple 1.2 : La figure 1.3 présente le contenu d'un fichier JSON qui décrit un menu déroulant ainsi que son fonctionnement.

```
{
  "menu": {
    "id": "menu_file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```

FIGURE 1.3 – Un fichier JSON



1.1.2 Réglementation

Les données stockées sont soumises à des réglementations strictes, en particulier lorsqu'il s'agit de données personnelles.

Un mouvement important est celui de l'*Open Data* qui met à disposition des données collectées et produites par des services publics, tout en respectant la réglementation.

Les données accessibles sur le web peuvent faire l'objet de *web scraping* ou collectes de données, en particulier à des fins de référencement.

En France, la CNIL (*Commission Nationale de l'Informatique et des Libertés*) a pour objectif la protection des données personnelles. Ainsi, tout fichier ou base de données comportant des données personnelles, qu'elles soient explicites (par exemple nom, prénom) ou implicites (par exemple numéro de téléphone), doit être déclaré à la CNIL.

En Europe, le RGPD (*Règlement général sur la Protection des Données*) régit l'utilisation des données personnelles.

1.2 Bases de données relationnelles

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.

1.2.1 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 1.3 : 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.

1.2.2 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.

1.2.3 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.

1.2.4 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. Les éléments nécessaires à la compréhension d'un diagramme UML (*Unified Modelling Language*) simple seront présentés. Les éléments nécessaires du langage *SQL* pour l'écriture de scripts (étape 4) sont présentés dans le chapitre 3. L'intégration à un programme (étape 5) est réalisée dans le chapitre 4 pour quelques langages de programmation.

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.

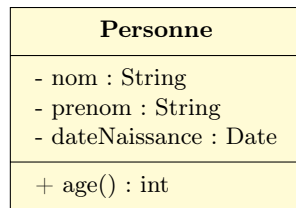


FIGURE 2.1 – Classe Personne

◇

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.

◇

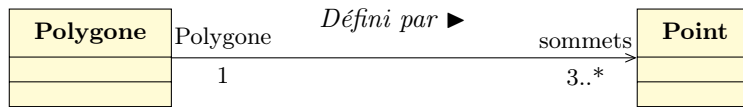


FIGURE 2.4 – Association entre des classes `Polygone` et `Point`

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.

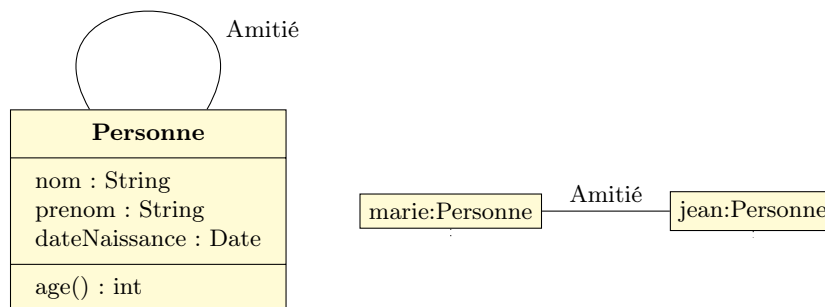


FIGURE 2.5 – Une association réflexive sur la classe `personne`, et le lien entre deux objets de cette classe ◇

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 sur 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).

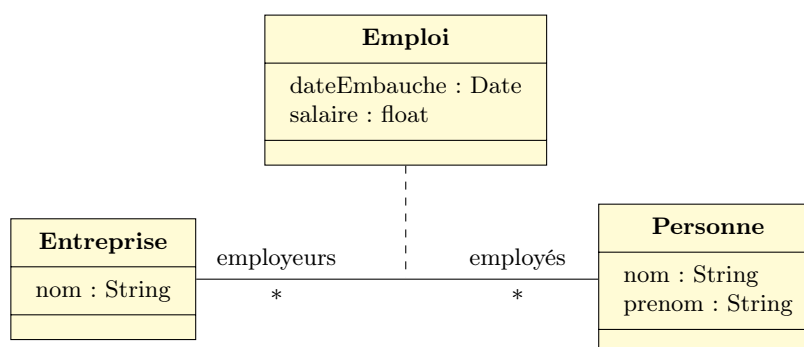


FIGURE 2.6 – Une classe-association

◇

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.

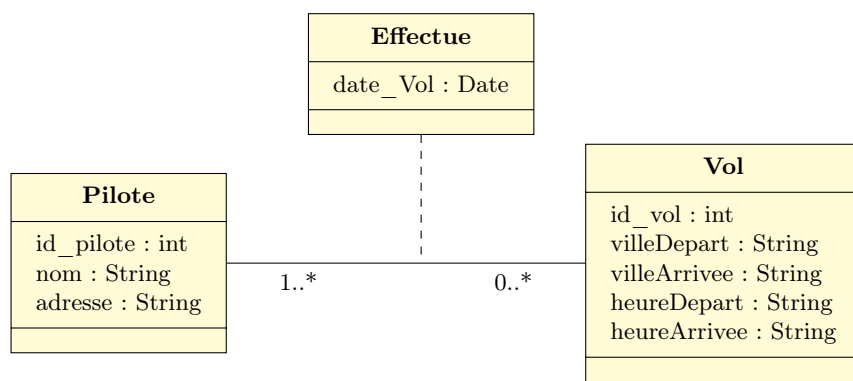


FIGURE 2.7 – Diagramme de classes pour les vols aériens

◇

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.

<code>PILOTE</code>	<code>id_pilote</code>	<code>nom</code>	<code>adresse</code>
	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 `MODELE`, comme dans les tables 2.2 et 2.3 respectivement.

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

◇

AVION	<u>id_avion</u>	type	compagnie
	10	A340	Air France
	20	B747	British Airways
	30	B747	Qantas

TABLE 2.2 – Les avions

MODELE	<u>type</u>	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 **MODELE**.

Lors de la suppression d’un tuple dans **MODELE**, 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

SQL

3.1 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.

3.2 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 3.1 : 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 3.2 : 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 3.3 : 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,
    constructeur varchar(30) not null,
    capacite smallint check(capacite > 0)
);
```

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 3.4 : 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)
);

```

3.3 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...;

```

3.4 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.

3.4.1 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 3.5 : 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 values (3,'B737','British Airways');
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 3.5.1).

```

select * from Avion;

id_avion | type | compagnie
-----+-----+-----
      2 | B707 | Qantas
      3 | B737 | British Airways
      4 | A320 | Air France
      5 | A320 |
      6 | A320 |
      8 | B737 | Air France

```

3.4.2 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 3.6 : La modification suivante s'applique au vol `BA302`. Elle change la date du vol.

```
update Effectue set date_vol='2020-02-07'
  where id_vol='BA302';
```

3.4.3 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 3.7 : Le script suivant supprime l'avion numéro 6.

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

3.5 Interrogation de la base de données

Les programmes travaillant avec des bases de données sont également amenés à en extraire de informations. Dans cette section, les éléments du langage SQL permettant d'effectuer de telles requêtes sont présentés, ainsi que des fonctions plus avancées.

3.5.1 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`.

3.5.2 Projection

Définition 11 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 $\{A_i\}$.

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

Pour faire une projection en SQL, on utilise la clause `SELECT` en désignant les attributs sur lesquels la projection est effectuée.

Exemple 3.8 : Quels sont les différents types d'avions ?

Pour répondre à cette question, il faut projeter la relation avion sur l'attribut `type`. La table 3.1 montre la colonne retenue, et la table 3.2 le résultat.

AVION	id_avion	type	compagnie
	2	B707	Qantas
	3	B737	British Airways
	4	A320	Air France
	5	A320	Air France
	8	B737	Air France

TABLE 3.1 – Projection de la relation AVION sur son attribut type

type
B707
B737
A320
A320
B737

TABLE 3.2 – Relation résultant de la projection

En SQL, il faut sélectionner le `type` des avions dans la table `Avion`.

```
select Avion.type from Avion;

type
-----
B707
B737
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.

Exemple 3.9 : Complétons l'exemple 3.8 en éliminant les doublons.

```
select distinct Avion.type from Avion;

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

3.5.3 Opération de sélection

Définition 12 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 .

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

Exemple 3.10 : Quels modèles d'avion peuvent accueillir au moins 300 passagers ?

Pour répondre à cette question, il faut sélectionner les tuples tels que `capacité` \geq 300. L'opération est schématisée dans la table 3.3, et le résultat dans la table 3.4. ◇

MODELE	type	constructeur	capacité
	A340	Airbus	228
	B747	Boeing	432
	A320	Airbus	150
	B707	Boeing	189
	B737	Boeing	210

TABLE 3.3 – Opération de sélection

type	constructeur	capacité
B747	Boeing	432

TABLE 3.4 – Résultat de la sélection

Dans un `SELECT`, la clause `WHERE` permet de spécifier *un critère de sélection* C . 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 3.11 : 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 '14:00:00' and '18:00:00';
```

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 3.12 : 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 ~'^D';
5
6 select Pilote.* from Pilote where Pilote.adresse is null;
```

3.5.4 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 3.13 : 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;
```

3.5.5 Jointure

Définition 13 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 3.14 : Considérons les relations `AVION` (table 3.5) et `MODELE` (table 3.6). Ces relations partagent un attribut qui est le `type` d'avion. La jointure `AVION` \bowtie_{type} `MODELE` permet de voir toutes les informations concernant un avion. Elle est présentée dans la table 3.7.

AVION	id_avion	type	compagnie
	2	B707	Qantas
	3	B737	British Airways
	4	A320	Air France
	5	A320	Air France
	8	B737	Air France

TABLE 3.5 – La relation AVION

MODELE	type	constructeur	capacité
	A340	Airbus	228
	B747	Boeing	432
	A320	Airbus	150
	B707	Boeing	189
	B737	Boeing	210

TABLE 3.6 – La relation MODELE

AVION \bowtie_{type} MODELE	id_avion	type	compagnie	constructeur	capacité
	2	B707	Qantas	Boeing	189
	3	B737	British Airways	Boeing	210
	4	A320	Air France	Airbus	150
	5	A320	Air France	Airbus	150
	8	B737	Air France	Boeing	210

TABLE 3.7 – Jointure entre AVION et MODELE

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

Définition 14 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 15 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 3.15 : Reprenons l'exemple 3.14, 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 3.8 et 3.9. Le jointure extérieure dans la table 3.10 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
	2	B707	Qantas
	3	B737	British Airways
	4	A320	Air France
	5	A320	Air France
	8	B737	Air France

TABLE 3.8 – La relation AVION

MODELE	type	constructeur	capacité
	A340	Airbus	228
	B747	Boeing	432
	A320	Airbus	150
	B707	Boeing	189
	B737	Boeing	210

TABLE 3.9 – La relation MODELE

AVION ⋈ _{type} MODELE	id_avion	type	compagnie	constructeur	capacité
	2	B707	Qantas	Boeing	189
	3	B737	British Airways	Boeing	210
	4	A320	Air France	Airbus	150
	5	A320	Air France	Airbus	150
	8	B737	Air France	Boeing	210
		A340		Airbus	228
		B747		Boeing	432

TABLE 3.10 – Jointure extérieure entre AVION et MODELE

◇

En SQL, la clause `SELECT` peut également être utilisée pour effectuer une jointure. 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 3.16 : 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 3.17 : Quels sont les différentes combinaisons d'avions et de pilotes utilisées sur les vols ?

On sélectionne les attributs `nom` 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`⋈`Effectue`⋈`Vol`.

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

nom	id_vol
Dupond	AF1232
Smith	QT17
Garratt	QT17
MacMachin	BA302
Garratt	BA302

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 3.18 : 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,Effectue.id_vol from Pilote
  left outer join Effectue
    on Pilote.id_pilote=Effectue.id_pilote;
```

nom	id_vol
Dupond	AF1232
Garratt	QT17
Smith	QT17
Garratt	BA302
MacMachin	BA302
Durand	

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

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

nom
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 3.19 : 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_depart, SP.heure_depart
  from Vol as tard join Vol as SP
    on tard.heure_depart > SP.heure_depart
  where SP.ville_depart = 'Sydney' and SP.ville_arrivee = 'Perth';

id_vol | heure_depart | heure_depart
-----+-----+-----
AF1232 | 15:30:00     | 14:30:00

```

3.5.6 Union

Définition 16 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 éventuels.

Exemple 3.20 : 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 3.11 et 3.12. Le résultat de l'union $P_{AF} \cup P_{BA}$ est présenté dans la table 3.13. 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 3.11 – Pilotes d'Air France

P_{BA}	id_pilote	nom	adresse
	3	Garratt	Perth
	4	Durand	Paris
	5	MacMachin	Glasgow

TABLE 3.12 – Pilotes de British Airways

$P_{AF} \cup P_{BA}$	id_pilote	nom	adresse
	1	Dupond	Nice
	2	Smith	Londres
	3	Garratt	Perth
	4	Durand	Paris
	5	MacMachin	Glasgow

TABLE 3.13 – Pilotes travaillant pour Air France ou pour British Airways

◇

En SQL, 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 16 qui requiert que les deux relations aient le même schéma.

Exemple 3.21 : Quels sont les avions d'Air France assurant des vols depuis Paris et ceux de British Airways assurant des vols depuis Londres ? ¹

Un premier `SELECT` (lignes 1–5) trouve les avions d'Air France assurant des vols depuis Paris, tandis que le second (lignes 7–11) trouve ceux de British Airways assurant des vols depuis Londres. Le résultat demandé est l'union de ces deux résultats (ligne 6).

1. Pour cela, il faut savoir sur quels avions sont effectués les vols, donc on rejoute `id_avion` à `Effectue` ainsi qu'à sa clé primaire et on la déclare clé étrangère.

```

1 select Avion.* from Avion,Vol,Effectue
2     where Avion.id_avion=Effectue.id_avion
3           and Vol.id_vol=Effectue.id_vol
4           and Avion.compagnie='Air France'
5           and Vol.ville_depart='Paris'
6 union
7 select Avion.* from Avion,Vol,Effectue
8     where Avion.id_avion=Effectue.id_avion
9           and Vol.id_vol=Effectue.id_vol
10          and Avion.compagnie='British Airways'
11          and Vol.ville_depart='Londres';
12
13 id_avion | type | compagnie
14 -----+-----+-----
15          | 3 | B737 | British Airways
16          | 4 | A320 | Air France

```

3.5.7 Intersection

Définition 17 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 3.22 : 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 3.11) et dans P_{BA} (table 3.12). La table $P_{AF} \cap P_{BA}$ résultante est présentée dans la figure 3.14.

$P_{AF} \cap P_{BA}$	id_pilote	nom	adresse
	3	Garratt	Perth

TABLE 3.14 – Pilotes travaillant pour Air France et pour British Airways

En SQL, 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 3.23 : Quels sont les avions assurant à la fois des vols depuis Londres et depuis Paris ?

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

```

1 select Avion.* from Avion,Vol,Effectue
2     where Avion.id_avion=Effectue.id_avion
3           and Vol.id_vol=Effectue.id_vol
4           and Vol.ville_depart='Londres'
5 intersect
6 select Avion.* from Avion,Vol,Effectue
7     where Avion.id_avion=Effectue.id_avion
8           and Vol.id_vol=Effectue.id_vol
9           and Vol.ville_depart='Paris';

```

3.5.8 Différence

Définition 18 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 3.24 : 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 3.11) mais pas dans P_{BA} (table 3.12). La table $P_{AF} \setminus P_{BA}$ résultante est présentée dans la figure 3.15.

$P_{AF} \setminus P_{BA}$	id_pilote	nom	adresse
	1	Dupond	Nice
	2	Smith	Londres

TABLE 3.15 – Pilotes d’Air France ne travaillant pas pour British Airways

◇

En SQL, 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 3.25 : Quels sont les avions assurant des vols depuis Londres mais pas depuis Paris ?

Les deux `SELECT` sont identiques à ceux de l’exemple 3.23, mais le résultat est obtenu en faisant la différence des deux à la ligne 5.

```
1 select Avion.* from Avion,Vol,Effectue
2   where Avion.id_avion=Effectue.id_avion
3         and Vol.id_vol=Effectue.id_vol
4         and Vol.ville_depart='Londres'
5 except
6 select Avion.* from Avion,Vol,Effectue
7   where Avion.id_avion=Effectue.id_avion
8         and Vol.id_vol=Effectue.id_vol
9         and Vol.ville_depart='Paris';
10
11 id_avion | type | compagnie
12 -----+-----+-----
13      3 | B737 | British Airways
```

◇

3.5.9 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 3.26 : 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.

```
1 select distinct Avion.type from Avion natural join Modele
2   where Modele.construteur='Boeing'
3         and Modele.capacite > ANY
```

```

4         (select Modele.capacite from Modele
5           where Modele.constructeur='Airbus');
6
7 type
8 -----
9 B707
10 B737

```

Exemple 3.27 : 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 3.26, à part l'opérateur de comparaison ligne 3.

```

1 select distinct Avion.type from Avion natural join Modele
2   where Modele.constructeur='Boeing'
3     and Modele.capacite > ALL
4       (select Modele.capacite from Modele
5         where Modele.constructeur='Airbus');

```

3.5.10 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 3.28 : Quels sont les avions qui assurent au moins un vol depuis Londres ?

Pour chaque avion (ligne 2), la requête imbriquée (lignes 3–5) sélectionne les vols qu'il effectue (jointure implicite réalisée par la ligne 4) au départ 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 Effectue natural join Vol
4                 where Effectue.id_avion=Avion.id_avion
5                   and Vol.ville_depart='Londres');
6
7 id_avion | type | compagnie
8 -----+-----
9         3 | B737 | British Airways

```

Exemple 3.29 : Quels sont les avions qui n'assurent pas de vol depuis Londres ?

La requête est similaire à celle de l'exemple 3.28, 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 Effectue natural join Vol
                    where Effectue.id_avion=Avion.id_avion
                      and Vol.ville_depart='Londres');

id_avion | type | compagnie
-----+-----
2 | B707 | Qantas
4 | A320 | Air France
5 | A320 |
8 | B737 | Air France

```

3.5.11 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 3.30 : Le script suivant affiche les noms et prénoms des pilotes respectivement en majuscule et minuscules.²

```
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
Garratt	Jo	GARRATT jo
MacMachin	Allan	MACMACHIN allan
Smith	Wendy	SMITH wendy

Exemple 3.31 : 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

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

Exemple 3.32 : Ce script utilise une racine carrée (pour montrer l'utilisation de fonctions numériques).

```
select Modele.type, sqrt(float8(capacite))
from Modele where Modele.capacite is not null;
```

type	sqrt
------	------

2. Il faut au préalable ajouter un attribut `prenom` à la table `Pilote`.

A340	15.0996688705415
B747	20.784609690826528
A320	12.24744871391589
B707	13.74772708486752
B737	14.491376746189438

3.6 Groupes

3.6.1 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*.

3.6.2 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 3.33 : Combien y a-t-il d'avions ?

On constitue un groupe avec tous les avions et on les compte :

```
select count(*) from avion;

count
-----
5
```

Exemple 3.34 : 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
-----+-----
B707 | 1
B737 | 2
A320 | 2
```

Exemple 3.35 : Quelle est la capacité moyenne des modèles d'avions ?

On groupe les avions et on fait la moyenne de l'attribut `capacité` sur le groupe :

```
select avg(capacite) from modele;

      avg
-----
241.8000000000000000
```

3.6.3 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 3.36 : 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 |      2
```

Chapitre 4

Utilisation d'un SGBD dans un programme

4.1 Objectif

Le but est 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 *recupérer* puis *traiter les résultats* de ces requêtes.

4.2 Client C d'un SGBD

4.2.1 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.

4.2.2 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 exécutée 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=nom_serveur 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 }
```

4.3 Client java d'un SGBD

4.3.1 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` 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.

4.3.2 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.

```

1 import java.sql.*;
2 import java.util.ArrayList;
3
4 public class GestionBD {
5     private Connection connexion;
6     private Statement stmt;
7
8     // constructeur : ouvre une connexion puis une session.
9     public GestionBD(String url,String pilote,String gestionnaireBD,
10         String motDePasse,PrintWriter out)
11         throws SQLException,ClassNotFoundException {
12         Class.forName(pilote);
13         this.connexion = DriverManager.getConnection(url,gestionnaireBD,motDePasse);
14         this.stmt = connexion.createStatement();
15     }
16
17     // fermeture de la connexion
18     public void fermerConnexion() throws SQLException {
19         this.stmt.close();
20         this.connexion.close();
21     }
22
23     // exécution d'une requête
24     public ResultSet executerRequete(String requete) throws SQLException {
25         return(this.stmt.executeQuery(requete));
26     }
27
28     // retourne le tableau de String de la méta-structure
29     // (les noms des attributs) du ResultSet
30     public String[] getTableauNomsAttributsResultSet(ResultSet rs)
31         throws SQLException {
32         ResultSetMetaData rsmd = rs.getMetaData();
33         int nbCol = rsmd.getColumnCount(); // nombre de colonnes
34         String[] tab = new String[nbCol];
35         for (int i = 0; i < nbCol; i++)
36             tab[i] = rsmd.getColumnName(i + 1);
37         return(tab);
38     }
39
40     public ArrayList <String> getArrayListTuplesResultSet(ResultSet rs,
41         String separateur1,boolean nomsAttributs)

```

```

42     throws SQLException {
43     String s;
44     int i;
45
46     ResultSetMetaData rsmd = rs.getMetaData();
47     int nbCol = rsmd.getColumnCount(); // nombre de colonnes
48     ArrayList <String> al = new ArrayList <String> ();
49     if (nomsAttributs) {
50         s = "";
51         for (i = 0; i < nbCol - 1; i++)
52             s = s + rsmd.getColumnname(i + 1) + separateur1;
53         s = s + rsmd.getColumnname(i + 1);
54         al.add(s);
55     }
56     while (rs.next()) {
57         s = "";
58         for (i = 0; i < nbCol - 1; i++)
59             s = s + rs.getString(i + 1) + separateur1;
60         s = s + rs.getString(i + 1);
61         al.add(s);
62     }
63     return(al);
64 }
65 } // 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://nom_serveur/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.getListTuplesResultSet(rs, ";", true);
9  for (int i = 0; i < al.size(); i++)
10     System.out.println(al.get(i));
11
12  gt.fermerConnexion();

```

4.4 Client Python d'un SGBD

L'utilisation avec python est similaire à celle en C (voir section 4.2).

```

#!/usr/bin/env python3
import psycopg2

# début : connection et création du curseur
connection = psycopg2.connect(database="mabase",
                             user="utilisateur",
                             host="nom_serveur",
                             password="motdepasse",
                             port="5432")
curseur = connection.cursor()

# requête
requete = "SELECT * FROM avion"
curseur.execute(requete)

```

```
# écriture des entêtes de colonnes
nom_col = [col[0] for col in curseur.description]
print(nom_col)

# récupération du résultat
resultat = curseur.fetchall()
# écriture du contenu de la table résultat
for ligne in resultat:
    print(ligne)

# fin
curseur.close()
connection.close()
```