Bases de données relationnelles et Web

Dernière étape de notre parcours : les bases de données. Parce que la plupart des bases de données sont aujourd'hui accessibles sur le Web, et parce que la plupart des sites web sont, en fait, l'interface de bases de données, c'est dans ce chapitre également que le Web est abordé. Cette thématique des bases de données nous permettra aussi d'approfondir un thème qui apparaissait déjà, mais de manière plus discrète, dans les chapitres « Représentation numérique de l'information » et « Structuration et contrôle de l'information », celui de l'indépendance des données et des programmes : ce sont les programmes qui s'adaptent aux données et non le contraire. Enfin, dans ce chapitre, nous mettrons en œuvre un projet plus ambitieux que dans ceux qui précédent : la réalisation d'un site web complet, qui permettra au lecteur de mettre en perspective son parcours depuis qu'il a ouvert ce livre à la première page.

Cours

Lorsque l'on souhaite faire persister des informations dans des applications complexes, c'est-à-dire que les données durent au-delà de la durée d'exécution du programme, on doit écrire ces informations sur une mémoire de masse, par exemple un disque dur, dans un fichier ou dans une base de données. La différence entre un fichier et une base de données est que les informations sont en vrac dans un fichier alors que les bases de données tirent profit de la structure des informations, que nous avons introduite au chapitre précédent. Les bases de données sont gérées par des logiciels appelés *Systèmes de gestion de bases de données* (SGBD) qui permettent un accès direct à des données précises, par exemple, trouver l'âge de *Jean Pierre,* sans demander une lecture et une analyse de toutes les informations contenues dans un fichier, pour obtenir celle qui nous intéresse. En outre, les bases de données évitent la perte des informations en cas de pannes logicielles, erreurs dans les programmes, ou matérielles, comme la destruction d'un disque.

Les systèmes de gestion de bases de données sont partout autour de nous : la plupart des sites web utilisent des systèmes de gestion de bases de données relationnels pour stocker leurs informations, les systèmes d'informations des banques sont gérés par des systèmes de gestion de bases de données, le Trésor public utilise des systèmes de gestion de bases de données pour sa comptabilité, etc. Le type de logiciels utilisé est le même, malgré la diversité de la structure des données et des données elles-mêmes. Il est impensable aujourd'hui de réaliser un logiciel de gestion d'informations sans avoir recours à une base de données, notamment pour garantir la persistance de l'information.

Rappelons également que les données peuvent être en quantité importante : gigaoctets – 10^9 octets –, téraoctets – 10^{12} octets –, voire pétaoctets – 10^{15} octets. Ces données peuvent également être de formats très différents : nombres, textes, images, vidéos, etc. Gérer ces données implique de pouvoir les stocker, les retrouver et les mettre à jour de manière efficace.

La plupart des données manipulées sont structurées. Les données peuvent être élémentaires comme un nom ou un prénom, composées comme une adresse, voire multivaluées comme les numéros de téléphone d'une personne. Elles décrivent des entités. Le lien entre une entité et une donnée s'appelle une relation.

Une *base de données* (BD) est donc une collection de données structurées reliées entre elles par des relations. Elle permet l'interrogation et la manipulation de ces données par des langages de haut niveau.

Comme nous l'avons vu au chapitre précédent, l'une des applications des bases de données est le Web, puisqu'il s'agit d'une immense source d'informations. Il faut noter que les données présentes sur le Web proviennent la plupart du temps de bases de données relationnelles, même si ces informations sont souvent en format « semi-structuré », c'est-à-dire (X)HTML. Nous montrons donc également dans ce chapitre comment de telles interactions se produisent.

En résumé, ce chapitre est une initiation aux bases de données *relationnelles*, aux systèmes de gestion de bases de données et aux concepts et technologies du Web. Nous expliquerons tout d'abord les concepts fondamentaux des bases de données et de leur conception, puis comment manipuler un système de gestion de bases de données pour y accéder depuis un programme. Enfin, nous réalisons une rapide présentation des concepts et technologies du Web, en montrant comment une page web s'interface avec une base de données.

Le modèle relationnel

Le *modèle relationnel* est un modèle de données s'appuyant sur la *logique des prédicats* ou *logique du premier ordre* et sur la *théorie des ensembles*. Ce modèle

suppose que toute donnée peut être représentée sous la forme d'une relation *n*-aire, c'est-à-dire à *n* arguments. Rappelons qu'une relation à *n* arguments est un sous-ensemble du produit cartésien de *n* ensembles. Dans le cas pratique, une relation est un ensemble fini. De là vient le terme de base de données « relationnelle ».

Relation, attribut et n-uplet

Informellement, dans le modèle relationnel, les données sont stockées dans des tableaux à deux dimensions appelés *tables*. L'ordre entre les colonnes n'a pas d'importance. Par exemple, on ne peut pas parler de la première ou de la deuxième colonne d'une table : chaque colonne est désignée par son nom. On appelle *attribut* le nom donné à une colonne de la table. Une ligne de la table contient des valeurs pour chacun des attributs. Chacune de ces lignes est appelée *n-uplet*, ou *tuple*. L'ensemble des n-uplets d'une table s'appelle une *relation*. Une *base de données relationnelle* (BDR) est formée d'un ensemble de relations.

Équipes

nom	nomCourt	ville
Paris Saint-Germain	PSG	Paris
Olympique de Marseille	OM	Marseille
Olympique Lyonnais	OL	Lyon

Joueurs

equipe	no	nom	prénom	age
OM	1	RIOU	Rudy	31
OM	10	GIGNAC	André-Pierre	25
PSG	1	COUPET	Grégory	37
PSG	4	MAKELELE	Claude	37
OL	1	LLORIS	Hugo	23
OL	9	LOPEZ	Lisandro	26

Rencontres

date	local	visiteur	butl	butv
19/12/10	OM	OL	1	1
28/11/10	OL	PSG	2	2
7/11/10	PSG	OM	2	1

Un exemple de base de données relationnelle.

Nous allons maintenant définir ces notions plus formellement.

À chaque *attribut* A est associé un *domaine* noté $Dom(A)$, qui représente l'ensemble de valeurs possibles de l'attribut. L'ensemble des nombres entiers, l'intervalle de nombres réels [0.0, 10000.0] et l'ensemble {*rouge, vert, bleu*} constituent des exemples de domaines.

Un *schéma de relation* $R(A_1, A_2, \ldots, A_n)$ est constitué d'un nom R et d'une liste d'attributs $A_1, \ldots, A_n$. Chaque attribut A_i représente le rôle joué par le domaine $Dom(A_i)$ dans le schéma de relation R. Un *schéma de base de données* est un ensemble de schémas de relation. Un exemple de schéma de base de données relationnelles, composé de trois schémas de relations, est le suivant

```
Equipes(nom : chaîne, nomCourt : chaîne, ville : chaîne)
Joueurs(equipe : chaîne, no : entier, nom : chaîne, pré-
nom : chaîne, age : entier)
Rencontres(date : Date, local : chaîne, visiteur :
chaîne, butl : entier, butv : entier)
```

Une *relation* r de schéma $R(A_1, A_2, \ldots, A_n)$ est un sous-ensemble du produit cartésien des domaines de R : $r \subseteq Dom(A_1) \times Dom(A_2) \times \ldots \times Dom(A_n)$. On dit aussi que r est une *instance* du schéma de relation R.

Un *n-uplet* de r est noté $t = \langle v_1, v_2, \ldots, v_n \rangle$ où $v_i \in Dom(A_i)$. La valeur du n-uplet t sur les attributs $A_i, \ldots, A_j$ est notée $t[A_i, \ldots, A_j]$.

Contrainte d'intégrité

Les concepts définis jusqu'ici permettent de créer des modèles de données nommés schémas, puis de les instancier pour y conserver des valeurs dans des relations. Cependant, toutes les instances possibles d'un modèle ne sont pas forcément pertinentes. Des *contraintes d'intégrité* fournissent un moyen de restreindre les relations acceptables pour un schéma de base de données. C'est une propriété du schéma de la base de données, que doivent vérifier les instances à tout moment; par exemple, un salaire doit être compris entre 100 et 100 000 euros. On distingue les contraintes de domaine, les contraintes d'entité qui expriment l'unicité des valeurs d'un attribut et les contraintes référentielles qui expriment les liens entre deux relations.

Une contrainte d'entité, ou *clé*, est une contrainte très courante dans le modèle relationnel. Une clé représente l'ensemble minimal d'attributs qui permet d'identifier de façon unique chaque n-uplet d'une relation. Une *superclé* K d'un schéma de relation R est un ensemble d'attributs $K = \{A_{i1}, \ldots, A_{ik}\} \subseteq R$ tel que pour toute relation r sur R et pour tout n-uplet $t,u \in r$, $t[K] = u[K] \Rightarrow t = u$.

Une *clé candidate*, appelée *superclé minimale* ou tout simplement *clé*, K de R est une superclé telle que si $X \subseteq K$ est une superclé de R, alors $X = K$. Une relation peut posséder plusieurs clés. En général, tout schéma de relation possède au moins une clé. Par exemple, les attributs `nom` et `nom_court` sont deux clés de la relation `Equipes`. La plupart du temps, on choisit de mettre en avant, pour chaque relation, une clé candidate, appelée alors *clé primaire*.

Une contrainte référentielle, ou *clé étrangère*, explicite les liens entre les tables de la base de données. Elle permet d'assurer que les relations entre tables demeurent cohérentes. Quand une table possède une clé étrangère vers une autre table, il n'est pas possible d'ajouter un n-uplet dans la première si un n-uplet correspondant n'existe pas dans l'autre. L'expression $R[A_{i1}, \ldots, A_{ik}] \subseteq S[B_{j1}, \ldots, B_{jk}]$ est une *clé étrangère* si $\{B_{j1}, \ldots, B_{jk}\}$ est une clé de S et si pour tout r sur R et s sur S, pour tout $t \in r$, il existe $u \in s$ tel que $t[A_{il}] = u[B_{jl}]$ pour $1 \leq l \leq k$. Une telle contrainte impose que, lors d'une insertion dans la table r, la valeur des attributs $A_{i1}, \ldots, A_{ik}$ doit exister dans la table s. De même, lors d'une modification dans la table s, les n-uplets de r doivent être mis à jour de façon adéquate.

Reprenons et complétons l'exemple ci-avant afin d'illustrer l'ensemble de ces notions. La relation `Equipes` possède les deux clés {*nom*} et {*nom_court*}, la relation `Joueurs` {*equipe,no*} et {*nom,prenom*}, la relation `Rencontres` {*date, local, visiteur*}. Le lien entre `Joueurs` et `Equipes` est formalisé par la clé étrangère *Joueurs*[*equipe*] $\subseteq$ *Equipes*[*nom_court*]. Les liens entre `Rencontres` et `Equipes` sont *Rencontres*[*local*] $\subseteq$ *Equipes*[*nom_court*] et *Rencontres*[*visiteur*] $\subseteq$ *Equipes*[*nom_court*].

L'algèbre relationnelle

Le modèle relationnel a permis le développement de plusieurs langages d'interrogation de données. Certains de ces langages sont théoriques, basés sur la logique – *calcul relationnel de n-uplets, calcul relationnel de domaines, datalog* – ou sur la théorie des ensembles – *algèbre relationnelle*. D'autres sont des implémentations plus ou moins standard comme *SQL* qui s'appuie sur l'algèbre relationnelle et le calcul de n-uplets ou *Query by Example* (*QBE*) issu du calcul de domaines. Dans ce paragraphe, nous nous intéressons à l'algèbre relationnelle qui a des applications directes dans les systèmes de gestion de bases de données : elle permet de définir une sémantique simple des requêtes, en facilitant des optimisations pour leur évaluation.

L'algèbre relationnelle est une collection d'opérateurs algébriques, chaque opérateur ayant une ou deux relations en entrée et une relation en sortie, qu'il est possible de composer, cette possibilité étant la *propriété de fermeture* de l'al-

gèbre. Les principaux opérateurs de l'algèbre relationnelle sont la *sélection* (σ) qui extrait un sous-ensemble de n-uplets d'une relation, la *projection* (π) qui extrait un sous-ensemble des attributs d'une relation, le *renommage* ($\rho_{A \to B}$) qui renomme un attribut, le *produit cartésien* ($\times$) qui fait le produit cartésien de deux relations, la *jointure* ($\bowtie$) qui combine deux relations selon un certain critère, la *division* ($\div$) qui exprime le quantificateur universel $\forall$ et l'*agrégation* (γ) qui combine plusieurs n-uplets en un seul. En plus de ces opérateurs, l'algèbre relationnelle fait aussi appel aux opérateurs ensemblistes d'union, d'intersection et de différence. Nous détaillons maintenant ces opérateurs.

Opérateurs de l'algèbre relationnelle

La *sélection* σ, ou *restriction*, permet de sélectionner un sous-ensemble des n-uplets d'une relation vérifiant une condition booléenne. Cette dernière est une *formule de sélection* composée de constantes, de noms d'attributs, d'opérateurs de comparaison, d'application de fonctions et des opérateurs logiques $\vee$ (ou), $\wedge$ (et), et $\neg$ (non). Soit r une relation sur R et F une formule de sélection, la sélection $\sigma_F(r)$ est définie par $\sigma_F(r) = \{t \in r \mid t \text{ satisfait } F\}$. Par exemple, la requête « les joueurs de plus de 30 ans » s'écrit $\sigma_{age>30}(Joueurs)$. Il faut noter que cette condition est évaluée pour chacun des n-uplets. Il s'agit donc d'une condition locale.

La *projection* π permet de sélectionner les n-uplets correspondant à un sous-ensemble des attributs d'une relation. Soit r une relation sur R et $X \subseteq R$, la projection de r sur X est définie par $\pi_X(r) = \{t[X] \mid t \in r\}$. Par exemple, la requête « nom et prénom des joueurs » s'écrit $\pi_{\{nom,prenom\}}(Joueurs)$.

Le renommage ($\rho_{A \to B}$) renomme un attribut. Soit r une relation sur R, $A \in R$ et $B \notin R$, la relation r avec A renommé en B est définie comme $\rho_{A \to B}(r) = \{t \mid \exists u \in r \text{ tel que } t[B]=u[A] \wedge t[C] = u[C] \text{ si } C \neq B\}$.

À partir de deux relations r sur R et s sur S, le *produit cartésien* forme une relation dont les n-uplets sont la concaténation des n-uplets de r avec ceux de s, c'est-à-dire $r \times s = \{t \mid t[R] \in r \wedge t[S] \in s\}$.

La *jointure*, quant à elle, forme une relation dont les n-uplets sont la concaténation des n-uplets de r avec ceux de s qui vérifient la condition de jointure F, c'est-à-dire $r \bowtie_F s = \{t \mid t[R] \in r \wedge t[S] \in s \wedge t \text{ satisfait } F\}$. La jointure peut aussi s'exprimer comme un produit cartésien suivi d'une sélection, c'est-à-dire $r \bowtie_F s = \sigma_F(r \times s)$.Par exemple, la requête « nom des joueurs et nom et ville de leur équipe » s'écrit $\pi_{\{joueur,nom,ville\}}(\rho_{nom \to joueur}(Joueurs) \bowtie_{equipe=nomCourt} Equipes)$ ou de façon équivalente $\pi_{\{joueur,nom,ville\}}(\sigma_{equipe=nomCourt}(\rho_{nom \to joueur}(Joueurs) \times Equipes))$. Si on ne précise pas de condition, la jointure se fait sur les attributs communs aux deux relations. On parle alors de *jointure naturelle*. Dans ce cas, les

colonnes connectées ne sont pas répétées. Si la condition est une égalité, on parle d' *équijointure.* Enfin, si $r = s$, il s'agit d'une *autojointure*.

La *division* ÷ permet d'obtenir une relation dont le produit cartésien avec le diviseur est un sous-ensemble du dividende. Cet opérateur permet de répondre à des requêtes du type « Quels sont les joueurs qui ont joué dans toutes les rencontres de leur équipe ? ». Formellement, si $R(A_1, \ldots, A_n, A_{n+1}, \ldots, A_m)$ et $S(A_{n+1}, \ldots, A_m)$ sont deux schémas, la division de deux relations r sur R et s sur S est définie par $r \div s = \{t \mid \forall t' \in s, (t, t') \in r\}$. La division peut être obtenue à partir de la différence, du produit cartésien et de la projection : $r \div s = u \setminus v$ où $u = \pi_{\{A1, \ldots, An\}}(r)$ et $v = \pi_{\{A1, \ldots, An\}}((u \times s) \setminus r)$.

L'*agrégation* (γ) résume une relation en regroupant certains n-uplets en un seul. Cet opérateur permet de répondre à des requêtes du type « nombre moyen de buts marqués à domicile par chaque équipe ». Pour fusionner un ensemble de n-uplets, on utilise des fonctions spéciales appelées *fonctions d'agrégation* dont les plus courantes sont le comptage (*COUNT*), la somme (*SUM*), la moyenne (*AVG*), le minimum (*MIN*) et le maximum (*MAX*). Soit un schéma $R(A_1, \ldots, A_n)$, l'expression ${}_{B1, \ldots, Bm}\gamma_{F1(C1), \ldots, Fp(Cp)}(r)$ où B_i et C_i sont des attributs de R construit une relation de schéma $B_1, \ldots, B_m, F_1(C_1), \ldots, F_p(C_p)$. Les attributs $B_1, \ldots, B_m$ sont des *attributs de groupement*. L'opérateur γ commence par rassembler les n-uplets de r qui ont même valeur sur $B_1, \ldots, B_m$. Ensuite, pour chaque ensemble de n-uplets, les fonctions d'agrégation $F_1(C_1), \ldots, F_p(C_p)$ sont appliquées. L'exemple de requête donnée ci-avant s'écrirait donc ${}_{local}\gamma_{AV\ G(butl)}$ (*rencontres*).

À partir de deux relations r et s, l'*union* produit une relation contenant les n-uplets appartenant à r ou à s, $r \cup s = \{t \mid t \in r \vee t \in s\}$. La *différence* produit une relation contenant les n-uplets appartenant à r mais pas à s, $r \setminus s = \{t \mid t \in r \wedge t \notin s\}$. Enfin, l'*intersection* produit une relation contenant les n-uplets appartenant à r et à s, $r \cap s = \{t \mid t \in r \wedge t \in s\}$. Ces opérateurs ne sont définis que des relations r et s de même schéma.

Le langage SQL

L'algèbre relationnelle fait partie des fondements des bases de données relationnelles. C'est un modèle sémantique utilisé en interne par le système de gestion de bases de données pour manipuler les requêtes. Le langage SQL, quant à lui, est destiné aux utilisateurs de systèmes de gestion de bases de données. Il est, aujourd'hui, le langage standard d'interrogation des bases de données. Il est fondé sur l'algèbre relationnelle et le calcul relationnel de n-uplets. Il a été standardisé par l'*ANSI* en 1986, puis par l'ISO en 1989, 1992, 1999, 2003 et 2008. Ces standards successifs ont étendu les capacités du lan-

gage. En plus de ces standards, SQL possède de nombreux dialectes spécifiques à chaque système de gestion de bases de données. En effet, ces systèmes se sont développés parallèlement à la standardisation de la norme SQL, voire avant. En conséquence, seul un sous-ensemble de la norme est supporté par la grande majorité des systèmes, en particulier les systèmes de gestion de bases de données libres. Ce sous-ensemble correspond à la norme SQL-92. C'est cette version que nous présentons ici.

La norme SQL-92 se décompose en plusieurs parties. L'interrogation des données se fait par l'opération de *sélection* (`SELECT`), dont la sémantique est donnée par une expression de l'algèbre relationnelle. Le *langage de manipulation de données* (*Data Manipulation Language ou DML*) permet de modifier les données grâce à l'*insertion* (`INSERT`), la *mise à jour* (`UPDATE`) et la *suppression* (`DELETE`) de n-uplets. Le *langage de définition de données* (*Data Definition Language ou DDL*) se charge de la définition du schéma d'une base de données. Enfin, SQL-92 inclut d'autres opérations comme la gestion des droits ou celle des transactions.

Le langage de définition de données

Le langage de définition de données permet la modification du schéma d'une base de données. Il propose trois opérations : la création (`CREATE`), la suppression (`DROP`) et la modification (`ALTER`). Par exemple, la création d'une table a pour syntaxe : `CREATE TABLE ⟨table⟩ (⟨définitions de colonnes⟩ [⟨contraintes de table⟩])`. Chaque définition de colonne comporte le nom de la colonne, son type et éventuellement une spécification de contraintes d'intégrité. Il est également possible de définir des contraintes d'intégrité sur la table. Les types de données supportés sont les chaînes de caractères de taille fixe, c'est-à-dire complétées à droite par des espaces (`CHAR`(*taille*)), les chaînes de taille variable (`VARCHAR`(*taille*)), les entiers sur 32 bits (`INTEGER`), les nombres en précision fixe (*nbChiffres* chiffres dont *nbDecimales* après la virgule) (`NUMERIC`(*nbChiffres*, *nbDecimales*)), les nombres en virgule flottante simple précision (`REAL`), les nombres en virgule flottante double précision (`DOUBLE PRECISION`) et les types date et/ou heure (`DATE/TIME/TIMESTAMP`).

Concernant les contraintes d'intégrité, le langage DDL permet de spécifier qu'un ensemble d'attributs ne contient pas de doublons (`UNIQUE`), qu'un ensemble d'attributs ne contient pas de valeur manquante (`NOT NULL`), qu'un ensemble d'attributs est clé primaire (`PRIMARY KEY`), qu'une séquence d'attributs est clé étrangère (`FOREIGN KEY/REFERENCES`) ou de contraindre le domaine d'un attribut (`CHECK`).

Par exemple, la requête DDL

```
CREATE TABLE equipes(
nom VARCHAR(30) UNIQUE NOT NULL,
nomCourt CHAR(5) PRIMARY KEY,
ville VARCHAR(50)
);
```

crée la table `equipes`. Les contraintes `UNIQUE` et `NOT NULL` sur nom précisent que cet attribut est une clé candidate. L'attribut `nomCourt` est la clé primaire, donc annoté par `PRIMARY KEY`. Il est donc `UNIQUE` et `NOT NULL`.

Ce deuxième exemple

```
CREATE TABLE joueurs(
equipe CHAR(5) REFERENCES Equipes(nomCourt),
no INTEGER,
nom VARCHAR(30),
prenom VARCHAR(30),
age INTEGER CHECK(age BETWEEN 18 AND 40),
PRIMARY KEY(equipe, no),
CONSTRAINT joueurs_uniq_nom_prenom UNIQUE(nom, prenom)
);
```

crée la table `joueurs`. La contrainte référentielle précise que l'attribut `equipe` est une clé étrangère. La contrainte de domaine sur `age` limite les valeurs entre 18 et 40. La contrainte de table `PRIMARY KEY(equipe, no)` définit la clé primaire. Enfin, la contrainte de table nommée `joueurs_uniq_nom_prenom` précise l'autre clé candidate.

Nous terminons avec la requête de création de la table `rencontres`.

```
CREATE TABLE rencontres(
date_r : DATE,
local : CHAR(5) REFERENCES equipes(nomCourt),
visiteur : CHAR(5) REFERENCES equipes(nomCourt),
butl : INTEGER,
butv : INTEGER,
PRIMARY KEY(date_r, local, visiteur)
);
```

Le langage de manipulation de données

Maintenant que le schéma de la base de données est défini grâce au langage DDL, il reste à ajouter, dans la base de données, les données proprement dites. C'est le rôle du langage DML.

Une insertion de n-uplets a la syntaxe suivante : `INSERT INTO ⟨table⟩ [(⟨liste d'attributs⟩)] VALUES (⟨liste de valeurs⟩) | ⟨requete⟩`. On précise la liste des attributs à renseigner, les autres valeurs sont alors fixées à la valeur par défaut de l'attribut ou à `NULL`. La clause `VALUES` ajoute un n-uplet à la fois, alors que l'utilisation d'une requête à la place de `VALUES` permet d'insérer plusieurs n-uplets en une seule opération.

La requête suivante permet l'ajout du n-uplet ⟨'Paris Saint-Germain', 'PSG', 'Paris' ⟩ dans la relation `equipes` :

```
INSERT INTO equipes
VALUES('Paris Saint-Germain', 'PSG', 'Paris');
```

La deuxième requête insère un n-uplet en ne précisant que les attributs `nom` et `nomCourt`. Ce n-uplet n'aura pas de valeur pour l'attribut `ville` (`NULL`).

```
INSERT INTO equipes(nom, nomCourt)
VALUES('Olympique de Marseille', 'OM');
```

Enfin, une dernière requête insère dans la table `joueurs` des n-uplets extraits de la table `juniors`.

```
INSERT INTO joueurs
SELECT no, nom, prenom, age, 'OM'
FROM juniors
WHERE evaluation > 15;
```

L'instruction `UPDATE` modifie les valeurs des n-uplets d'une table : `UPDATE ⟨table⟩ SET ⟨ liste d'affectations ⟩ [WHERE ⟨condition⟩]`. L'utilisation d'une liste de valeurs permet de modifier la valeur de plusieurs attributs. La clause `WHERE` définit les n-uplets mis à jour.

La requête suivante augmente l'âge des joueurs de l'OM et les affecte au PSG.

```
UPDATE joueurs
SET age = age + 1, equipe = 'PSG'
```

```
WHERE equipe = 'OM';
```

L'instruction `DELETE` supprime des n-uplets d'une table : `DELETE FROM ⟨table⟩ [WHERE ⟨condition⟩]`. La clause `WHERE` précise les n-uplets à supprimer.

Par exemple, la requête suivante supprime toutes les équipes parisiennes.

```
DELETE FROM equipes
WHERE ville = 'Paris';
```

Si la clause `WHERE` est absente, tous les n-uplets de la table sont supprimés.

```
DELETE FROM joueurs;
```

L'interrogation des données

Le dernier aspect du langage SQL que nous étudierons est l'interrogation des données, grâce à l'instruction `SELECT` :

```
SELECT ⟨liste d'expressions⟩
FROM ⟨liste de tables⟩
WHERE ⟨conditions⟩
GROUP BY ⟨liste d'attributs⟩
HAVING ⟨conditions⟩
ORDER BY ⟨liste d'attributs⟩
```

La clause `SELECT` spécifie le schéma de sortie – projection –, la clause `FROM` précise les tables impliquées et leurs liens – produit cartésien et jointures –, la clause `WHERE` fixe les conditions que doivent remplir les n-uplets résultats – sélection –, `GROUP BY` indique les attributs de groupement – agrégation –, `HAVING` impose une condition sur les groupes, `ORDER BY` définit les critères de tris des résultats.

La clause `SELECT` est suivie d'une liste d'expressions. Chaque expression peut être un attribut, un littéral entre guillemets ou une expression calculée. Le caractère « * » est un joker signifiant « tous les attributs de la table ».

Il est à noter que la clause `SELECT` peut faire apparaître des doublons, c'est-à-dire plusieurs n-uplets identiques. Il est possible d'éliminer ces doublons, avec `DISTINCT` dans la clause `SELECT`, mais cette opération est coûteuse. De fait, contrairement aux tables du modèle et de l'algèbre relationnels, qui sont

des ensembles, les tables de SQL sont des multi-ensembles, et les requêtes *SELECT/FROM/WHERE* sont définies sur de tels multi-ensembles.

Les requêtes suivantes illustrent ces aspects.

– Nom et prénom des joueurs

SQL
```
SELECT nom, prenom
FROM joueurs;
```
Algèbre $\pi_{nom,prenom}(joueurs)$

– Tous les attributs des joueurs

SQL
```
SELECT *
FROM joueurs;
```
Algèbre $\pi_{equipe,no,nom,prenom,age}(joueurs)$

– Âge des joueurs (sans doublons)

SQL
```
SELECT DISTINCT age
FROM joueurs;
```
Algèbre $\pi_{age}(joueurs)$

La clause `FROM` accepte en paramètre une liste de tables. La requête porte alors sur le produit cartésien de ces tables. Pour effectuer une jointure entre les tables, il convient d'utiliser les opérateurs de jointure du langage. Le mot-clé `JOIN` placé entre deux tables indique qu'il faut en faire la jointure. La condition de jointure est précisée avec le mot-clé `ON`. Une jointure naturelle s'exprime avec le mot-clé `NATURAL JOIN`.

L'exemple suivant montre deux requêtes de jointure.

– Nom des joueurs et de leur équipe

SQL
```
SELECT j.nom AS nom, e.nom AS equipe
FROM joueurs j JOIN equipes e ON j.equipe =
e.nomCourt;
```
Algèbre $r_1 = \rho_{equipe \rightarrow club}(joueurs)$

$r_2 = \rho_{nom \rightarrow equipe}(equipes)$

$\pi_{nom,club}(r_1 \bowtie_{club=nomCourt} r_2)$

– Tous les résultats des rencontres

SQL
```
SELECT l.nom AS local,
       butl || '/' || butv AS resultat,
       v.nom AS visiteur?
FROM equipes l
JOIN rencontres r ON l.nomCourt = r.local
JOIN equipes v ON r.visiteur = v.nomCourt;
```

Algèbre $r_1 = {}_{\rho * \rightarrow l.*}(equipes)$

$r_2 = r_1 \bowtie l._{nomCourt=local}\ rencontres$

$r_3 = r_2 \bowtie_{visiteur=v.nomCourt}\ {}_{\rho * \rightarrow v.*}(equipes)$

$\pi_{l.nom, butl||'/'||butv, v.nom}(r_3)$

Dans cet exemple, des alias d'attributs et de table sont utilisés : la syntaxe `AS nom` dans la clause `SELECT` permet de renommer la colonne en sortie, la syntaxe `joueurs j` définit `j` comme alias de la table `joueurs`. Enfin, la seconde requête fait usage de l'opérateur de concaténation « || » pour mettre en forme les résultats.

La clause `WHERE` permet d'exprimer la sélection. SQL supporte des opérateurs de comparaison, `<`, `<=`, `=`, `>=`, `>`, `<>`, logiques, `AND`, `OR`, de test de valeur manquante, `IS NULL`, `IS NOT NULL`, de recherche textuelle, `LIKE`, de sélection d'intervalle, `BETWEEN`, de liste, `IN`, et d'imbrication de blocs `IN`, `EXIST`, `NOT EXIST`, `ALL`, `SOME`, `ANY`.

Les requêtes suivantes illustrent ces notions.

– Nom et prénom des joueurs de plus de 30 ans

SQL
```
SELECT nom, prenom FROM joueurs
WHERE age > 30;
```

Algèbre $\pi_{nom,prenom}(\sigma_{age>30}(joueurs))$

– Nom et prénom des joueurs qui ont entre 20 et 30 ans et dont le nom commence par 'L'

SQL
```
SELECT nom, prenom FROM joueurs
WHERE age BETWEEN 20 AND 30
AND nom LIKE 'L%';
```

Algèbre $r_1 = \sigma_{age>20\ \wedge\ age<30\ \wedge nom\ \ LIKE\ 'L\%'}\ (joueurs)$

$\pi_{nom,prenom}(r_1)$

– Nom des joueurs dont l'âge est inconnu

SQL
```
SELECT nom FROM joueurs
WHERE age IS NULL;
```

Algèbre $\pi_{nom}(\sigma_{age\ IS\ NULL}(joueurs))$

Les fonctions d'agrégation, les clauses `GROUP BY` et `HAVING` permettent de « résumer » un ensemble de n-uplets. La clause `GROUP BY` précise les attributs de groupement. Les fonctions d'agrégation sont ensuite appliquées sur chaque groupe. Enfin, la clause `HAVING` permet de spécifier des conditions sur les groupes. Il s'agit donc ici d'une condition *globale* à un groupe.

– Nombre moyen de buts marqués à domicile

SQL
```
SELECT AVG(butl) FROM rencontres;
```
Algèbre $\gamma_{AVG(butl)}(rencontres)$

– Nombre total de buts marqués à domicile par équipe

SQL
```
SELECT nom AS Equipe, SUM(butl) AS Buts
FROM equipes JOIN rencontres ON local = nomCourt
GROUP BY nom
```
Algèbre $r_1 = equipes \bowtie_{local=nomCourt} rencontres$

$_{nom}\gamma_{SUM(butl)}(r_1)$

– Nombre moyen de buts marqués à domicile par équipe ayant disputé plus de 5 rencontres

SQL
```
SELECT nom AS Equipe, AVG(butl) AS Moyenne
FROM equipes JOIN rencontres ON local = nom
      Court
GROUP BY nom
HAVING COUNT(*) > 5;
```
Algèbre $r_1 = equipes \bowtie_{local=nomCourt} rencontres$

$r_2 = {}_{nom}\gamma_{AVG(butl),COUNT(*)}(r_1)$

$\sigma_{COUNT(*)>5}(r_2)$

SQL permet également de regrouper les résultats de plusieurs requêtes à l'aide d'opérations ensemblistes. La requête ⟨*requete*1⟩ `UNION` ⟨*requete*2⟩ retourne l'union des résultats des deux requêtes. La requête ⟨*requete*1⟩ `INTERSECT` ⟨*requete*2⟩ retourne l'intersection des résultats des deux requêtes. La requête ⟨*requete*1⟩ `EXCEPT` ⟨*requete*2⟩ retourne la différence des résultats des deux requêtes. Pour ces trois opérations, les schémas des requêtes doivent être compatibles, c'est-à-dire avoir le même nombre et le même type d'attributs. Ces opérations s'effectuent sur des ensembles, c'est-à-dire que les doublons sont éliminés. Le mot-clé `ALL` ajouté après un opérateur ensembliste évite l'élimination des doublons et permet donc de manipuler des multi-ensembles.

La clause `ORDER BY` spécifie l'ordre de tri des résultats. Elle est suivie d'une liste d'attributs pour lesquels on précise l'ordre de tri – `ASC` pour croissant, `DESC` pour décroissant. Cette clause, qui n'a pas d'équivalent en algèbre relationnelle, ne doit apparaître qu'une seule fois dans une requête et obligatoirement à la fin. En effet, elle transforme un ensemble de n-uplets en liste de n-uplets pour représenter l'ordre. Il n'est donc pas possible d'appliquer un autre opérateur sur le résultat de cette clause.

– Nom et prénom des joueurs triés en ordre croissant selon le nom puis le prénom

```
SELECT nom, prenom FROM joueurs
ORDER BY nom ASC, prenom ASC;
```

Arrivés à ce point, nous avons étudié le modèle de données qui constitue le fondement des bases de données relationnelles, ainsi que deux langages d'interrogation de ces données : l'algèbre relationnelle et SQL. Il reste une question à laquelle nous n'avons pas répondu : comment choisit-on le schéma de base de données à créer ? L'étude de la conception de base de données va nous permettre de répondre.

Conception de bases de données relationnelles

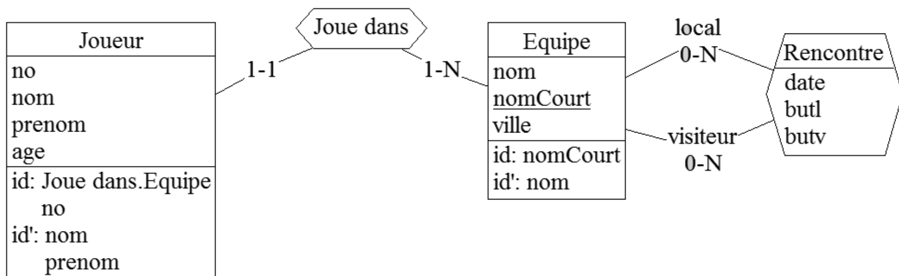
La *conception de bases de données relationnelles* a pour objectif la création d'un modèle relationnel pour une application, à partir de l'*analyse des besoins*. C'est une problématique très importante, car on constate que tous les modèles, pour une même application, ne se valent pas : un schéma de base de données mal conçu peut engendrer des anomalies lors des mises à jour de la base de données, par exemple des incohérences, des problèmes de performance ou un espace occupé trop important. Deux grandes approches sont possibles : la *normalisation* et la *modélisation conceptuelle*.

La *normalisation* consiste à représenter les besoins sous la forme de contraintes d'intégrité. Ces contraintes sont ensuite utilisées comme entrées dans un *algorithme de normalisation* qui génère un schéma de base de données satisfaisant certaines propriétés. On dit que le schéma résultant est en *forme normale*. La difficulté de la normalisation réside dans la nécessité de disposer de l'ensemble des contraintes d'intégrité pour démarrer le processus. L'énumération de ces contraintes étant une tâche ardue, la normalisation s'utilise en pratique rarement de façon indépendante.

La *modélisation conceptuelle* consiste à représenter les besoins sous la forme d'un *modèle conceptuel*. Ce dernier, plus expressif que le modèle relationnel, est ensuite traduit en modèle relationnel, en respectant des *règles de transformation de modèles*. La modélisation conceptuelle a pour objectifs de permettre une meilleure compréhension du problème – *abstraction* – et de permettre une conception progressive – *raffinement*. La démarche consiste à identifier les concepts fondamentaux, à les représenter dans le formalisme choisi, en général un formalisme visuel, puis à en dériver le schéma de la base de données. La difficulté de cette approche est la création du modèle conceptuel. En effet, même si c'est un modèle plus expressif, il demeure difficile à réaliser pour une application de grande taille. Il n'existe pas de recette pour obtenir à coup sûr un bon modèle conceptuel. La réussite d'une telle approche repose sur l'utilisation de bonnes pratiques de modélisation ainsi que sur l'expérience de l'équipe en

charge de cette tâche. Dans la suite de ce paragraphe, nous détaillons cette approche en nous appuyant sur le modèle entité-association.

Le modèle *entité-association* est un ensemble de concepts pour la modélisation des données d'une application. Chaque concept peut être représenté par un symbole graphique associé. Une *entité* est un objet du monde réel qui peut être identifié et que l'on souhaite représenter. Le *type d'entité* correspond à l'ensemble des objets ayant des caractéristiques communes. Une entité possède un ensemble de propriétés nommées *attributs*. Dans le modèle de base, un attribut est *simple*, ou *atomique*, c'est-à-dire qu'il ne possède qu'une seule valeur pour un n-uplet donné. Dans les modèles étendus, un attribut peut être *composé* ou *multivalué*, ce qui permet, par exemple, de gérer en un seul n-uplet les prénoms multiples d'une personne. Chaque entité possède un *identifiant* ou *clé*, c'est-à-dire un ensemble d'attributs qui différencie une instance d'entité parmi les instances de son type d'entité. Sur la figure ci-après, un type d'entité est représenté par un rectangle comportant trois cartouches. Le premier contient le nom du type d'entité, le second les attributs et le troisième les identifiants. L'identifiant primaire est noté `id`, les secondaires `id'`. On remarque que l'identifiant primaire du joueur est un *identifiant hybride* formé du type d'entité voisin et d'un attribut local. En effet, le numéro d'un joueur ne suffit pas pour l'identifier : il peut par exemple exister un numéro 10 – le meneur de jeu – par équipe. L'identification d'un joueur se fait donc grâce à son numéro, mais aussi avec son équipe.



Un diagramme entité-association.

Les entités sont reliées par des *associations*. Le *type d'association A* est un sous-ensemble du produit cartésien des types d'entités E_i participants : $A \subseteq E_1 \times E_2 \times \ldots \times E_n$. Une *instance d'association* a_i est un n-uplet $(e_1, \ldots, e_n)$ où e_i est de type E_i. Une association peut avoir des attributs. Le *degré* d'une association est le nombre d'entités participantes. Les associations les plus courantes sont *binaires*, c'est-à-dire de degré 2. Une entité participant à une association

peut avoir un *rôle* : le rôle qu'elle joue dans l'association. La *cardinalité* d'une association précise le nombre de fois où une instance d'entité peut participer à l'association. On rencontre généralement les cardinalités *un à un* (1 : 1), *un à plusieurs* (1 : N) ou *plusieurs à plusieurs* (N : N). La *cardinalité maximum* précise le nombre maximum de fois où une instance d'entité peut participer à l'association (1 ou N). La *cardinalité minimum* précise le nombre minimum de fois où une instance d'entité peut participer à l'association (0 ou 1). Sur la figure ci-avant, les types d'associations sont représentés par des hexagones contenant le nom de l'association. Éventuellement, l'hexagone peut contenir des attributs comme dans le cas de `Rencontre`. Le type d'association est relié aux types d'entité participants. Sur un lien entre un type d'entité E et un type d'association A apparaissent la cardinalité minimum et maximum. Ces dernières définissent le nombre d'instances de E qui peuvent participer à A. Sur le schéma, en considérant uniquement les cardinalités maximums, on constate que `Joue dans` est une association 1 : N et `Rencontre` une N : N. Il est également possible de préciser le rôle d'un participant sur le lien : dans le type d'association `Rencontre`, le type d'entité `Equipe` joue le rôle du `local` et du `visiteur`.

Une fois le modèle conceptuel construit, il reste à le transformer dans le modèle logique choisi. La *transformation de modèle* est le processus permettant de traduire un modèle M_1 en un modèle M_2. C'est une technique utilisée à plusieurs niveaux pour la conception de base de données : lors du passage du modèle conceptuel vers le modèle logique ou du modèle logique vers le modèle physique.

La traduction d'un modèle entité-association en un modèle relationnel se fait en respectant quelques règles simples. Chaque type d'entité E est transformé en un schéma de relation R dont les attributs sont les attributs de E. La clé primaire de R est choisie parmi les identifiants de E. Dans le cas général, un type d'association A entre les entités $E_1, \ldots, E_n$ est transformé en une relation R dont les attributs sont les identifiants des E_i et les éventuels attributs de A. La clé primaire de R est l'ensemble des identifiants des E_i participants. Dans le cas d'une association binaire 1 : 1 ou 1 : N, la clé de l'entité du côté N peut être ajoutée au schéma de relation représentant l'entité du côté 1. Dans le cas d'une association binaire 1 : 1, les deux entités participantes peuvent éventuellement être fusionnées. Ce dernier choix n'est pas forcément souhaitable si les entités participantes ont d'autres associations ou si les deux cardinalités minimales sont différentes de un.

La transformation du diagramme entité-association de notre exemple donne le schéma relationnel suivant. Le type d'entité `Joueur` devient le schéma de relation `joueurs` avec les attributs `no`, `nom`, `prenom` et `age`. Le type

d'entité `Equipe` devient le schéma de relation `equipes` avec les attributs `nom`, `nomCourt` et `ville`. Le type d'association 1 : N `Joue dans` est implémenté comme une clé étrangère dans `Joueur`. Le type d'association N : N `Rencontre` devient le schéma de relation rencontres qui possède comme attribut les identifiants des entités participantes et les attributs `date`, `butl` et `butv`.

Programmation et bases de données

Tout programme manipulant des données, il est assez naturel de se poser la question de l'accès à une base de données depuis un programme. Il existe pour cela différentes approches qui s'appuient sur des technologies variées.

La première possibilité consiste à utiliser un langage de programmation directement intégré au système de gestion de bases de données. On parle alors de *procédures stockées*. Par exemple, le standard SQL/PSM (*SQL/Persistent Stored Modules*) est une extension de la norme SQL. Il ajoute des possibilités de programmation dans un langage impératif, comme celui présenté au deuxième chapitre, aux capacités d'interrogation de SQL. La plupart des systèmes de gestion de bases de données proposent un ou plusieurs langages pour le développement de procédures stockées : PL/SQL pour Oracle, PL/pgSQL pour PostgreSQL, etc.

La deuxième possibilité, nommée *SQL embarqué*, intègre des requêtes SQL à un langage de programmation. Ce principe nécessite alors d'avoir recours à un *préprocesseur*. En effet, le programme incluant les ordres SQL n'est plus un programme valide pour le langage de programmation choisi. Avant la compilation du programme, le préprocesseur transformera les portions de code SQL pour que le programme puisse être compilé. Par exemple, SQLJ est une extension du langage Java qui permet d'utiliser du code SQL dans un programme. Certains systèmes de gestion de bases de données proposent ce type de préprocesseur. Le langage Pro*C/C++ est une extension des langages C/C++ permettant l'utilisation de SQL pour l'accès au système de gestion de bases de données Oracle.

La troisième approche fait appel à des *interfaces de programmation* (*API*) pour permettre l'accès aux systèmes de gestion de bases de données. Le principe est de fournir une bibliothèque de fonctions d'accès à un système de gestion de bases de données, pouvant être appelées à partir d'un langage donné. La norme SQL/CLI (*SQL/Call-Level Interface*) définit une interface standard pour cela. On peut également citer la bibliothèque *ODBC* (*Open DataBase Connectivity*), très communément utilisée, ou *JDBC* (*Java DataBase Connectivity*) qui est le standard de fait pour le langage Java.

Une dernière approche, qui fait appel à des bibliothèques de haut niveau, est rendue nécessaire du fait que la plupart des programmes sont écrits avec un langage objet et que la plupart des données sont conservées dans des bases de données relationnelles, mais ce problème dépasse le cadre de ce chapitre.

Quelle que soit l'approche choisie, l'accès à un système de gestion de bases de données à partir d'un langage de programmation suit un processus que nous présentons maintenant.

Principe général

Une *session* d'interaction avec un système de gestion de bases de données nécessite quatre grandes étapes : la connexion au système de gestion de bases de données et le choix de la base de données, l'exécution d'une requête, la récupération des résultats et la fermeture de la connexion. Les étapes d'exécution de la requête et de récupération des résultats peuvent évidemment être répétées autant que nécessaire lors d'une session.

L'ouverture d'une session avec un système de gestion de bases de données nécessite une connexion préalable. Le rôle de cette étape est d'authentifier la personne qui demande la connexion et de s'assurer que cette dernière possède bien les droits nécessaires pour cela. D'autre part, la plupart des systèmes de gestion de bases de données fonctionnent selon une architecture client/serveur. Lors de la connexion, il faut donc également préciser avec quel serveur on veut interagir. Toutes les informations utilisées pour l'ouverture de session sont en général regroupées au sein d'une *chaîne de connexion*. Cette dernière décrit en particulier le serveur à contacter – une adresse IP et un numéro de port —, le nom de la base de données, le nom de l'utilisateur, son mot de passe et diverses options spécifiques au type de connexion utilisée.

L'exécution d'une requête passe par l'appel d'une fonction spécifique de l'API. Cette dernière exécute donc des requêtes SQL sur le système de gestion de bases de données, à partir du langage de programmation. Cette fonction retourne un *ensemble de résultats*, aussi appelé *curseur*.

La récupération des résultats se fait grâce à ce dernier. Un curseur est une structure de données qui permet d'itérer sur les résultats de la requête.

Ce principe général d'accès à une base de données à partir d'un langage de programmation est naturellement implanté de façon différente en fonction du langage, du système de gestion de bases de données et de l'approche choisie. Le paragraphe suivant, « Publication de données sur le Web » offre un exemple concret de l'accès au système de gestion de bases de données MySQL à partir du langage PHP.

Publication de données sur le Web

Depuis les années 1990, l'information s'est installée sur le Web. Les premiers sites étaient composés de pages HTML (*HyperText Markup Language*) statiques, c'est-à-dire que tout leur contenu était figé. Pour modifier le contenu d'une telle page, il fallait directement éditer le code source de la page, et y apporter les modifications nécessaires. On retrouvera sur `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html` l'aspect et le contenu du premier site web du World Wide Web Consortium (W3C), publié en 1992.

Désormais, la plupart des données des sites web sont stockées dans des bases de données et les sites sont devenus de véritables applications manipulant données et présentation. Dans ce paragraphe, nous partons de la base de données sur les matchs de football, que nous avons construite, pour réaliser un site web consacré au football.

Qu'est-ce qu'un site web ?

Le terme *site web* est passé dans le langage courant. Et si l'on comprend qu'un site web est un ensemble de pages web affichant des informations ou permettant des interactions, on ignore souvent tout de l'architecture interne d'un tel site : comment il a été réalisé et comment il fonctionne. Plus précisément, il faut distinguer, d'une part, les infrastructures et matériels permettant l'hébergement et la diffusion du site et, d'autre part, le code qui correspond au « programme » du site web, exécuté sur cette infrastructure.

Nous présentons rapidement l'infrastructure matérielle nécessaire pour réaliser un site web, et renvoyons le lecteur au cinquième chapitre, pour de plus amples détails, puis nous présenterons les langages permettant l'écriture de pages web statiques, à savoir HTML (*HyperText Markup Language*) et CSS (*Cascading Style Sheets*), ainsi qu'un langage très utilisé, PHP (*Personal Home Page*), permettant l'écriture de pages web dynamiques, c'est-à-dire qui interagissent avec une base de données.

Internet On confond parfois le Web et Internet. Internet, développé depuis la fin des années 1960 aux États-Unis et utilisé commercialement à partir des années 1990, est un réseau informatique planétaire, composé physiquement de câbles de cuivre ou de fibres optiques, qui transmettent une information numérique d'un serveur à un autre. Ces données numériques sont interprétées grâce à des protocoles de communication à plusieurs niveaux : la couche physique, par exemple ADSL ou Wifi, la couche liaison, par exemple Ethernet ou PPPoE, la couche réseau, par exemple IPv4, la couche transport, par exemple TCP et, tout en haut, la couche application avec le standard HTTP

(*HyperText Transfer Protocol*) utilisé par les serveurs web et les navigateurs web comme Firefox, Explorer, Safari, etc. pour communiquer. Le Web est donc *une* application utilisant Internet, mais la plus largement utilisée, d'où la confusion. **Un serveur web ou HTTP** Le standard HTTP a été inventé par sir Tim Berners-Lee, qui par là même inventait le Web tel qu'on le connaît. Il est le fondateur et directeur du World Wide Web Consortium (W3C). Le W3C est une organisation de standardisation des technologies du Web qui a en particulier normalisé le langage HTML. Il définit le format des messages qui sont produits par un serveur à destination d'un navigateur. Le navigateur est une application qui reçoit des informations de la part du serveur web, puis interprète leur contenu pour afficher la page web demandée. Il faut donc distinguer HTTP qui est en quelque sorte l'enveloppe contenant, entre autres choses, le code de la page, et le contenu du message, en l'occurrence du HTML. Voici un exemple de communication en HTTP entre un navigateur et un serveur. Le client effectue une demande de transmission de page au serveur `www.exemple-foot.com`, par le biais de la commande GET suivie du nom de la page : `/index.html`, c'est-à-dire le fichier intitulé index.html se trouvant dans le répertoire racine du serveur et du format de transfert – ici HTTP1.1. Notons que le nom du serveur `www.exemple-foot.com` est transformé en adresse IP par un serveur de nom de domaines (DNS) qui connaît la correspondance entre le nom du serveur et son adresse.

```
GET /index.html HTTP/1.1
Host: www.exemple-foot.com
User-Agent: Mozilla 5.0
```

Le serveur répond quelque chose du genre :

```
HTTP/1.1 200 OK
Date: Mon, 1 Jan 2011 14:36:01 GMT
Server: Apache/2.2.9 (Debian)
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitio-
nal//EN">
<html> <head><title>Football</title></head>
```

```
<body> Texte de présentation du site de foot…</body>
</html>
```

Sans entrer dans les détails – qui sont disponibles à `http://www.w3.org/Protocols/rfc1945/rfc1945.txt` pour la version 1.0 et `http://www.ietf.org/rfc/rfc2616.txt` pour la version 1.1 –, nous remarquons que la première ligne indique un code 200 précisant que tout s'est bien passé – une page non trouvée renvoyant le code bien connu *Erreur 404*. Suit un ensemble d'informations décrivant le serveur et la connexion. À la suite de ces informations, il y a un saut de ligne, puis du texte qui correspond à une page web au format HTML, qui commence par `< !DOCTYPE html…` qui définit la structure de la page – ici du HTML 4.0 – puis la page à proprement parler, qui commence avec la balise `<html>`, comme nous l'avons vu au deuxième chapitre.

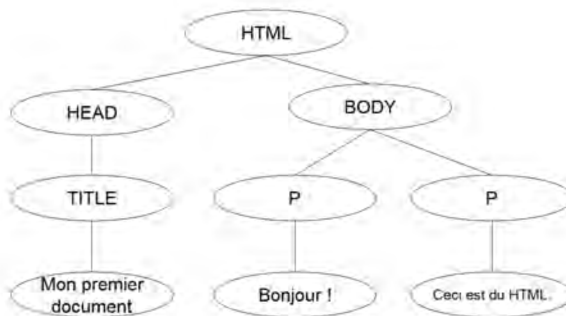
Les standards du Web

Il y a de nombreux standards définissant la manière de représenter les données sur le Web, ainsi que la manière de les présenter. Le standard de base de présentation de données est le format HTML – ou XHTML, qui est moins utilisé, mais compatible. Le format HTML4 est essentiellement un format de présentation, c'est-à-dire qu'il ne décrit pas le contenu sémantique du document, comme on le ferait avec une table d'une base de données sur des joueurs de football, où il y aurait un nom, un prénom, une date de naissance, etc., mais il indique plutôt que l'on veut mettre telle valeur en gras, telle valeur dans un tableau, etc. Voici un petit exemple de page HTML que nous allons commenter. Pour visualiser le rendu de cette page, il suffit de taper ce texte dans un éditeur de texte quelconque, puis, après l'avoir sauvegardé, de l'ouvrir à l'aide d'un navigateur. Dans la plupart des navigateurs, dans le menu *fichier*, une fonction permet d'ouvrir un fichier présent sur le disque de l'ordinateur utilisé.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
      <HEAD>
      <TITLE>Mon premier document</TITLE>
      </HEAD>
      <BODY>
      <P>Bonjour !
      <P>Ceci est du HTML.
```

```
    </BODY>
</HTML>
```

La première ligne `DOCTYPE` définit la version de HTML utilisée. Une version HTML correspond à une grammaire particulière, composée de mots précis : le vocabulaire HTML. En principe, si la page ne comporte pas d'erreurs de syntaxe, son rendu sera globalement le même quel que soit le navigateur utilisé. Parfois, même si certaines erreurs de syntaxe sont commises, le navigateur essaye de les corriger, mais dans ce cas le rendu varie souvent d'un navigateur à l'autre. Puisque l'objectif final d'une page web est d'être lue par un humain, de petites variations dans la présentation sont, la plupart du temps, sans conséquences. Viennent ensuite plusieurs *balises* – les majuscules étaient préférées dans la version 4.0, depuis les lettres minuscules ont pris le dessus, mais les deux casses sont autorisées – qui peuvent être soit ouvrantes, comme `<HTML>`, soit fermantes, comme `</HTML>`. Le plus simple est de se représenter un document HTML comme un *arbre*, illustré par la figure ci-après, dont la racine est le nœud `<HTML>`. XHTML est justement écrit dans le langage XML (*eXtensible Markup Language*) qui permet de décrire des arbres par l'utilisation de balises : une balise ouvrante indique le début d'un nœud, et toutes les balises contenues entre la balise ouvrante et la balise fermante composent le sous-arbre enraciné sous ce nœud.



Représentation arborescente d'un document HTML.

La liste de toutes les balises possibles est définie dans la spécification HTML écrite par le W3C, et disponible à l'adresse `http://www.w3.org/TR/html401/`. Au lieu de décrire ici toutes les balises une par une, nous allons nous contenter de montrer comment dessiner un tableau, ce qui est assez utile

quand on gère des données relationnelles. Voici un exemple de fichier HTML représentant un tableau de joueurs de football

```
<html>
<body>
<table border=1>
<tr><th>Nom</th><th>Prenom</th><th>Equipe</th></tr>
<tr><td>Benzema</td><td>Karim</td><td>Real  Madrid</td></
tr>
<tr><td>Malouda</td><td>Florent</td><td>Chealsea</td></
tr>
<tr><td>Sagna</td><td>Bacary</td><td>Arsenal</td></tr>
<tr><td>Lloris</td><td>Hugo</td><td>Olympique  Lyonnais</
td></tr>
</table>
</body>
</html>
```

dont le rendu par un navigateur est le suivant

Nom	Prenom	Equipe
Benzema	Karim	Real Madrid
Malouda	Florent	Chealsea
Sagna	Bacary	Arsenal
Lloris	Hugo	Olympique Lyonnais

Rendu d'un tableau en HTML.

Certaines balises ont des attributs, par exemple la balise `<table>` a un attribut `border` qui demande de dessiner un contour aux cellules. Chaque ligne de la table est définie par une balise `<tr>` (*table row*). À l'intérieur d'une ligne, chaque cellule est définie soit par une balise `<td>` (*table data*) soit par une balise `<th>` (*table header*).

Le langage HTML a été étendu en le langage XML (*eXtensible Markup Language*) – voir `http://www.w3.org/XML/` – qui est un format semi-structuré de données où l'utilisateur peut définir lui-même ses propres balises,

et ainsi structurer les données selon son souhait, comme dans une base de données.

La représentation sérialisée d'un document XML ressemble aux exemples que nous avons déjà donnés en HTML, à savoir des balises ouvrantes et fermantes. XML est très utilisé comme format d'échange de données sur le Web, comme format de stockage de fichiers de configuration et comme format d'intégration de données hétérogènes. Il existe même des systèmes de gestion de bases de données permettant de stocker des données XML et de les interroger via un langage de requête – il en existe essentiellement deux : XSLT et XQuery, tous deux développés par le W3C. Les systèmes de gestion de bases de données relationnelles commerciaux, comme Oracle, DB2, SQLServer, etc., supportent tous XQuery et le format XML. Des systèmes de gestion de bases de données purs XML gratuits existent également : eXistDB, Qizx, Zorba, etc. Par exemple, dans le cas précédent, on pourrait définir son propre fichier XML pour stocker des joueurs de football, que l'on représenterait comme suit :

```
<joueurs>
<joueur>
<nom>Benzema</nom><prenom>Karim</prenom><equipe>Real Ma-
drid</equipe>
</joueur>
 <joueur>
<nom>Malouda</nom><prenom>Florent</prenom><equipe>
Chelsea</equipe> </joueur>
<joueur>
<nom>Sagna</nom><prenom>Bacary</prenom><equipe>Arsenal</
equipe>
</joueur>
<joueur>
<nom>Lloris</nom><prenom>Hugo</prenom><equipe>Olympique
Lyonnais</equipe>
</joueur>
</joueurs>
```

Il faut ensuite définir une feuille de style – par exemple en XSLT – qui est un programme indiquant comment un navigateur doit afficher chaque élément.

Un exemple de programmation web : l'accès à MySQL depuis PHP

Nous donnons maintenant un exemple de programme, écrit dans le langage PHP, qui interagit avec une base de données et produit comme résultat le code source d'une page web, c'est-à-dire un texte représentant un document HTML. Le langage PHP ouvre l'accès à de nombreux systèmes de gestion de bases de données : MySQL, PostgreSQL, Oracle, etc. Nous utilisons, dans cet exemple, le système MySQL qui est l'un des systèmes de gestion de bases de données libres les plus utilisés. Le paragraphe suivant, « Exercices corrigés et commentés », fournit des explications plus approfondies sur l'installation d'un serveur web permettant d'exécuter ces programmes.

Des quatre approches présentées plus haut, au paragraphe « Programmation et bases de données », trois sont envisageables pour l'accès à MySQL depuis PHP. MySQL supporte les procédures stockées et il est possible de les invoquer depuis un programme PHP. En revanche, il n'existe pas d'approche de type SQL embarqué pour PHP. Différentes API donnent accès à un système de gestion de bases de données depuis PHP : certaines sont spécifiques à un produit, d'autres sont génériques et permettent de s'abstraire du type de système de gestion de bases de données utilisé. Enfin, la bibliothèque *Doctrine* interface des objets PHP avec un système de gestion de bases de données relationnel. Parmi ces possibilités, nous avons fait le choix de présenter l'API spécifique de MySQL, solution simple et répandue.

La fonction `mysql_connect` connecte au système de gestion de bases de données MySQL. Elle retourne une *ressource de connexion* – ou `FALSE` en cas d'échec – qui sera passée en paramètre de toutes les fonctions utilisant cette connexion. Elle prend en arguments les paramètres de connexion : `host, port, user, password` – dans cet exemple nous utilisons les paramètres par défaut.

Nous choisissons ensuite la base de données, qui doit avoir été créée préalablement, avec la fonction `mysql_select_db`. La notion de chaîne de connexion évoquée ci-avant est ici décomposée en plusieurs paramètres. La connexion est fermée en fin de session avec la fonction `mysql_close`. Cette dernière prend en paramètre la ressource de connexion à fermer. Les fonctions `mysql_error` et `mysql_errno` permettent de récupérer des informations dans le cas d'une erreur. Ce fichier est à sauvegarder avec l'extension `.php`, à placer dans le répertoire web – souvent nommé `www` – du serveur, et à visualiser dans un navigateur en se connectant sur ce serveur. Attention, ouvrir en double cliquant sur le fichier ne fera rien, si ce n'est ouvrir un éditeur de texte. Il faut lancer le navigateur, puis trouver l'URL, commençant par `http://localhost/...`, pour visualiser le fichier.

```
<?php
// Connexion, sélection de la base de données
$link = mysql_connect('localhost:3306', 'root', '')
or die('Could not connect: ' . mysql_error());
echo 'Connected successfully';
mysql_select_db('foot') or die('Could not select database');

// connexion à une base de données nommée "foot"
// sur l'hôte "localhost" avec un nom d'utilisateur root
// sans mot de passe

// Utilisation de la connexion…
mysql_close($link); ?>
```

La fonction `resource my_query (string $query, resource $connection)` exécute la requête `query` en utilisant la connexion `connection`. Si tout se passe bien, la fonction retourne un curseur nommé *ressource de résultats*. En cas d'erreur, `FALSE` est retourné. On utilise alors les fonctions `mysql_error` et `mysql_errno` pour obtenir des détails sur l'erreur.

```
<?php
// Exécution de la requête SQL
$query = 'SELECT nom, prenom FROM joueurs';
$result = mysql_query($query, $link) or die('Query failed:
' . mysql_error());
?>
```

La fonction `array mysql_fetch_row(resource $result)` permet de récupérer un n-uplet du résultat. Lors de chaque appel, le curseur mémorise le n-uplet à renvoyer lors de l'appel suivant. La fonction retourne un tableau où chaque valeur d'attribut du n-uplet occupe une cellule. Une valeur manquante est fixée à la valeur `PHP NULL`. La fonction retourne `FALSE` quand il n'y a plus de lignes. D'autres fonctions plus évoluées fonctionnent selon le même principe. Par exemple, la fonction `array mysql_fetch_array(resource $result)` renvoie un tableau associatif contenant les noms des attributs et non pas des indices. Enfin, certaines fonctions permettent d'obtenir des informations sur les résultats, appelées *métadonnées*

Fonction	Description
mysql_num_fields	Nombre d'attributs
mysql_num_rows	Nombre de n-uplets
mysql_affected_rows	Nombre de n-uplets affectés par l'ordre DML
mysql_field_flags	Détails d'un champ
mysql_field_name	Nom du champ
mysql_field_len	Taille du champ
mysql_field_table	Nom de la table
mysql_field_type	Type du champ

Fonctions d'interrogation des métadonnées.

Quand le curseur n'est plus utilisé, on libère la ressource avec `mysql_free_result`.

L'instruction `echo` permet de demander une sortie textuelle. Dans le cas de programmes PHP produits par un serveur web, il faut que le résultat, donc la sortie textuelle du programme, soit un fichier HTML, ce qui lui permettra d'être affiché par le navigateur. L'exemple de code suivant affiche le nom des joueurs de foot.

```
<?php
while ($row = mysql_fetch_row($result)) {
 echo "Joueur : $row[0] $row[1]";
 echo "<br />\n";
}
?>
```

En conclusion, cet exemple complet reprend les fragments du code présenté dans ce chapitre.

```
<?php
// Connexion, sélection de la base de données
$link = mysql_connect('localhost', 'user', 'password')
  or die('Could not connect: ' . mysql_error());
echo 'Connected successfully';
mysql_select_db('foot') or die('Could not select database');

// Exécution de la requête SQL
$query = 'SELECT nom, prenom FROM joueurs';
```

```
$result = mysql_query($query) or die('Query failed: ' .
mysql_error());

// Affichage des résultats en HTML
echo "<table>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
  echo "\t<tr>\n";
  foreach ($line as $col_value) {
echo "\t\t<td>$col_value</td>\n";
  }
  echo "\t</tr>\n";
}
echo "</table>\n";

// Libère le resultset
mysql_free_result($result);

// Ferme la connexion
mysql_close($link);
?>
```

Exercices corrigés et commentés

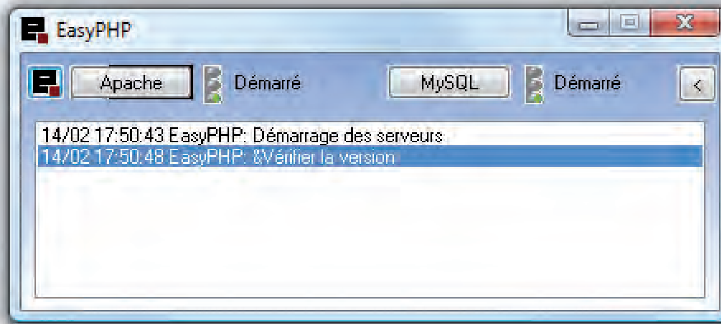
Installation du logiciel easyPHP

Nous présentons dans ce paragraphe l'installation d'un serveur web, d'un système de gestion de bases de données ainsi que sa configuration. Notre choix de logiciels s'est porté sur la distribution *easyPHP* qui peut être téléchargée gratuitement sur le site `http://www.easyphp.org/`, et nous détaillons son lancement sous Windows ; le lecteur transposera en fonction du système d'exploitation qu'il utilise. Nous lui laissons aussi le soin d'installer correctement le logiciel qui se compose, d'une part, du serveur web nommé Apache avec l'interpréteur PHP et, d'autre part, du système de gestion de bases de données MySQL.

Le répertoire d'installation sera nommé dans la suite MYSQLHOMEDIR, nous y mettrons les fichiers `.html` et `.php`. En lançant easyPHP, nous voyons apparaître dans la barre des programmes – en bas à droite – une lettre « e » qui correspond à l'icône représentant le programme

Icône EasyPHP.

En double cliquant sur l'icône, nous faisons apparaître la fenêtre d'état du serveur, symbolisée par deux feux de signalisation. Le feu de gauche correspond au serveur web Apache et celui de droite au système de gestion de bases de données MySQL. En principe, les deux feux doivent être au vert.

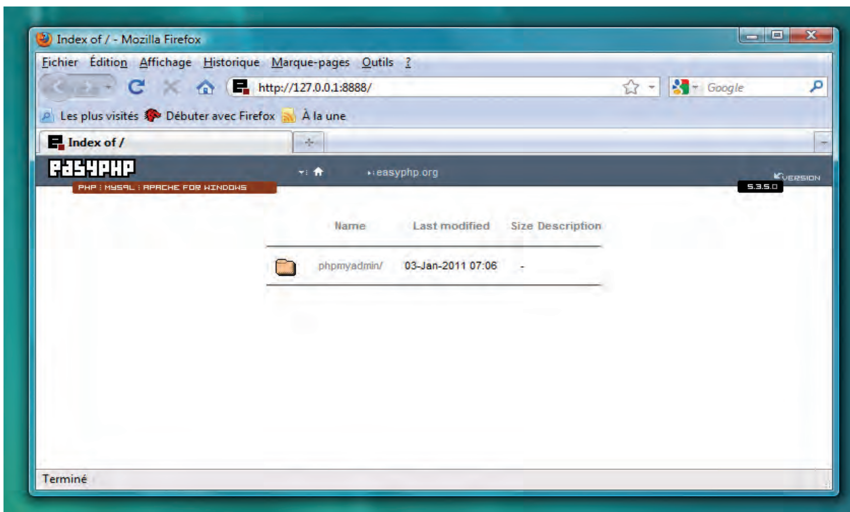


Fenêtre d'état du serveur EasyPHP.

Si ce n'est pas le cas, le problème vient la plupart du temps du fait que easyPHP n'a pas été lancé en mode administrateur. Au lieu de double cliquer sur l'icône pour lancer le programme, il faut alors faire un clic droit et choisir l'option *Exécuter en tant qu'administrateur*. Quand le serveur et le système de gestion de bases de données fonctionnent, nous visualisons le contenu du répertoire des fichiers qui peuvent être affichés par le serveur, en se rendant dans le répertoire www qui se trouve dans le répertoire d'installation de easyPHP. En principe, ce répertoire est vide. Nous y copions le répertoire *phpmyadmin* qui se trouve dans le répertoire d'installation de easyPHP. En faisant un clic droit et en choisissant l'option *web local*, un navigateur, par exemple Internet Explorer, Firefox, Chrome, etc., sera automatiquement lancé.

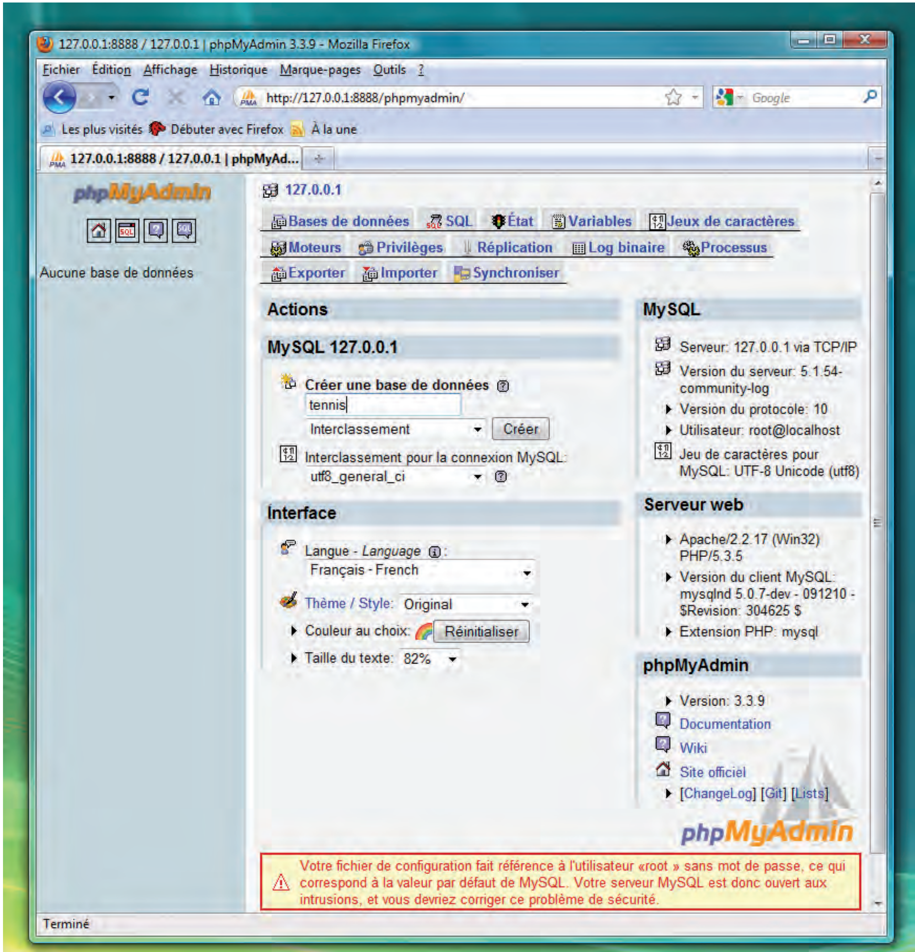
Cela sera le cas, même si la machine utilisée n'est pas connectée à Internet, puisqu'il s'agit d'une adresse locale. Bien entendu, si elle n'est pas connectée, personne d'autre que nous ne pourra visualiser les pages web créées. Si elle est connectée, les pages seront accessibles en tapant dans la barre l'adresse IP de la machine utilisée. Quand on crée un serveur web, il ne faut pas oublier de sécuriser la base de données par mot de passe, et d'ouvrir le port correspondant – par exemple 8888 – sur les divers pare-feu de la machine. On utilisera, dans ce cas, de préférence le port 80 qui peut être reconfiguré en faisant un clic droit sur l'icône d'EasyPHP et en choisissant le menu configuration->Apache et en remplaçant la ligne `Listen 127.0.0.1 :8888` par `Listen 127.0.0.1 :80`.

En lançant le navigateur, l'adresse qui apparaît dans la barre est `http://localhost:8888/` ou `http://127.0.0.1:8888/`. Si aucun numéro de port n'est indiqué, c'est le port 80 qui est utilisé par défaut.



Navigateur web.

La page affichée dans le navigateur présente le contenu du répertoire *www*, elle contient donc le lien nommé *phpmyadmin*. Si nous ajoutons d'autres fichiers dans ce répertoire et que nous rechargeons la page, nous les verrons apparaître dans la liste. En cliquant sur le lien *phpmyadmin* nous lançons l'interface graphique PHPMyAdmin, qui est en fait un ensemble de programmes PHP permettant de gérer la base de données. Nous voyons s'afficher la fenêtre



La page principale EasyPHP.

Nous considérons dans la suite que nous n'avons pas mis de mot de passe – ce qui est la configuration par défaut, bien que ce ne soit pas recommandé.

Schéma de la base

Nous construisons une base de données sur les matchs de tennis. La définition des tables est la suivante.

```
joueur(nom :chaîne, prenom :chaîne, age :entier,
            nationalite :chaîne)
```

```
rencontre(nomgagnant, nomperdant, nomtournoi, annee :en-
                        tier,
                 score :chaîne)
gain(nomjoueur, nomtournoi, annee :entier, rang :chaîne,
                 prime :entier)
sponsor(nom, nomtournoi, annee :entier, montant :entier)
           tournoi(nom :chaîne, pays :chaîne)
```

Le schéma de la base.

Les clés primaires sont soulignées, certaines sont multivaluées. Pour des raisons de simplicité, nous avons choisi le nom du joueur comme clé primaire. Cela signifie qu'il ne peut pas y avoir deux joueurs possédant le même nom de famille. Pour simplifier, le score sera une chaîne de caractères, par exemple `"6/3-6/4"`, et le rang également, `"gagnant"`, `"finaliste"`, etc.

Voici maintenant la définition des contraintes de clés étrangères :

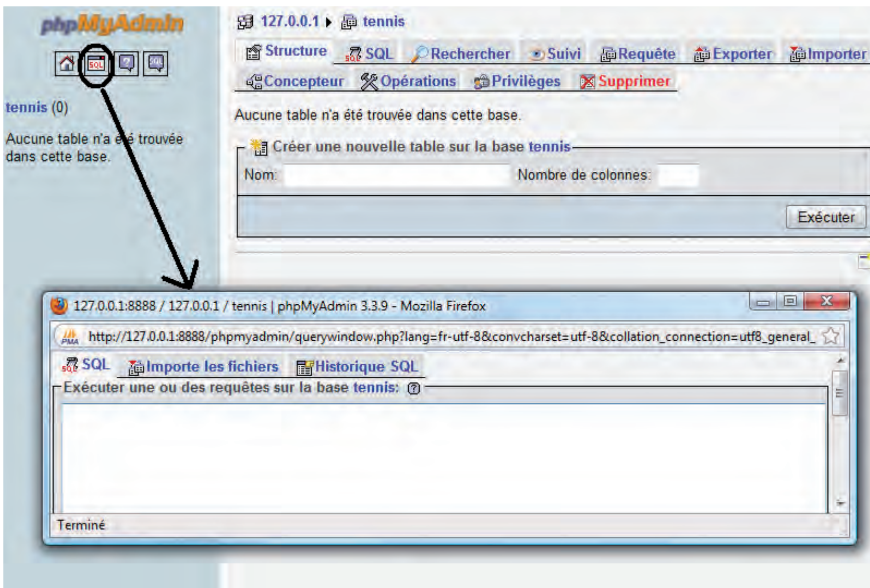
```
rencontre(nomgagnant) REFERENCES joueur(nom)
rencontre(nomperdant) REFERENCES joueur(nom)
rencontre(nomtournoi) REFERENCES tournoi(nom)
gain(nomjoueur) REFERENCES joueur(nom)
gain(nomtournoi) REFERENCES tournoi(nom)
sponsor(nomtournoi) REFERENCES tournoi(nom)
```

Notons que, d'une manière générale, il vaut mieux utiliser des noms sans espaces et n'utiliser que des lettres minuscules, sans accents, pour éviter les confusions.

Nous pouvons créer autant de bases différentes que nous voulons, et également gérer les droits des utilisateurs par rapport à ces bases, ce qui peut être utile dans le cas de la configuration d'un serveur utilisé par plusieurs utilisateurs. Alternativement, et c'est la solution que nous préconisons, il est possible d'installer le logiciel easyPHP sur chacun des ordinateurs utilisés.

Création de la base Nous remplissons le champ *Créer une base de données* avec le nom que nous souhaitons donner à notre base, par exemple `tennis`. Le champ interclassement permet d'indiquer comment sont stockées les chaînes de caractères dans la base, en particulier s'il faut faire une différence entre les majuscules et les minuscules – CS signifie : sensible à la casse et CI : insensible à la casse –, puis le type de langue à utiliser.

En cliquant sur le nom de la base de données `tennis`, nous voyons la liste des tables présentes dans cette base : pour l'instant, aucune. Cette touche permet de revenir en quelque sorte au menu principal de la base de données. Il est possible notamment de créer ou d'effacer des tables. L'interface d'administration se divise en deux parties : la partie gauche qui liste les tables déjà disponibles dans la base de données, et la partie droite d'administration à proprement parler. Dans la partie gauche, il y a une petite icône « SQL » qui sert à ouvrir une fenêtre pour entrer des instructions SQL. Cette fenêtre est à ouvrir lorsque nous aurons sélectionné la base. Nous y reviendrons plus tard.



Base de données Tennis.

Création des tables Il existe deux manières de créer des tables, l'une est graphique, l'autre utilise des instructions SQL. La manière graphique est plus simple, mais elle ne permet pas de gérer beaucoup de contraintes, en particulier de gérer les contraintes de clés étrangères, ce qui est trop limitatif dans notre exemple. Nous devons donc commencer en créant les tables qui n'ont pas de clés étrangères, puisque ces clés référencent des champs de tables déjà existantes.

Question 1

Donner un ordre correct pour la création des tables.

Correction. Nous pouvons créer dans l'ordre que nous souhaitons les tables qui n'ont pas de clé étrangère, c'est-à-dire la table `joueur` et la table `tournoi`. Dès que nous avons créé une table, nous pouvons créer toutes les tables qui ne font référence qu'à des tables existantes. Ainsi, dès que nous avons créé la table `tournoi`, nous pouvons créer la table `sponsor`.

Question 2

Créer les tables, en respectant bien les contraintes d'intégrité référentielles.

Correction. Dans l'interface graphique, nous entrons le nom de la table, par exemple `joueur`, et le nombre de colonnes, par exemple 4. S'ouvre alors une nouvelle page permettant de configurer les colonnes de la table. Il est possible d'ajouter ou de supprimer des colonnes par la suite, même si cela n'est pas recommandé, en particulier s'il y a déjà des données dans la table : car une valeur par défaut sera affectée pour cette nouvelle colonne aux données déjà présentes. Il ne faut remplir que les champs : *Champ*, *Type*, *Taille* et *Index*. Index permet de définir en particulier le fait qu'un attribut fait partie de la clé primaire (`PRIMARY KEY`). Nous illustrons cela dans la figure ci-après. Il faut indiquer les lignes des attributs qui participent à la clé primaire, ici seulement l'attribut `nom`. Les champs de type chaîne de caractères utilisent le type `VARCHAR`. La taille indique le nombre de caractères maximal. Les champs numériques utilisent le type `INT` pour les entiers, la taille indique le nombre de chiffres, ou `DECIMAL` pour les nombres à virgule. Dans ce cas, la taille est composée de deux valeurs *X*, *Y* où *X* représente le nombre de chiffres total et *Y* le nombre de chiffres après la virgule. Il est également très important de choisir comme type de la table la valeur `InnoDB`, afin de permettre la gestion des clés étrangères. Nous ne nous préoccupons pas des autres valeurs. Enfin, nous cliquons en bas à droite de l'écran sur le bouton `Sauvegarder` pour finaliser la création de la table.

127.0.0.1 ▸ tennis ▸ joueurs

Colonne	Type	Taille/Valeurs*1	Défaut2	Interclassement	Attributs	Null	Index	A_I
nom	VARCHAR	30	Aucun				PRIMARY	
prenom	VARCHAR	30	Aucun				---	
age	INT	2	Aucun				---	
nationalite	VARCHAR	30	Aucun				---	

Commentaires sur la table: | Moteur de stockage: InnoDB | Interclassement:

Création de la table joueur.

Lorsque nous créons une table, une requête SQL est générée. Nous conservons cette requête dans un fichier texte pour une utilisation ultérieure :

```
CREATE TABLE ‘joueur‘ (
 ‘nom‘ VARCHAR( 30 ) NOT NULL,
‘prenom‘ VARCHAR( 30 ) NOT NULL,
‘age‘ INT( 3 ) NOT NULL,
‘nationalite‘ VARCHAR( 30 ) NOT NULL,
PRIMARY KEY ( ‘nom‘ )
);
```

Les guillemets sont générés automatiquement et ne servent qu'à l'utilisation de noms de colonnes avec des espaces. D'une manière générale, si nous n'utilisons pas d'espaces, il est inutile d'utiliser ces guillemets, qui ne sont accessibles qu'avec la touche clavier `ALT-GR-7` et non pas la simple touche 4.

Nous créons, de même, la table `tournoi`, qui possède 2 colonnes, en utilisant l'éditeur graphique. Ensuite, nous utilisons la fenêtre SQL pour écrire le code SQL de création de tables nécessitant une ou plusieurs clés étrangères. Pour ce faire, nous écrivons une requête similaire à celle que nous venons de générer, en ajoutant les lignes de clés étrangères :

```
CREATE TABLE rencontre(
nomgagnant varchar(30,
nomperdant varchar(30,
nomtournoi varchar(30,
annee int(4,
score varchar(15),
PRIMARY KEY (nomgagnant, nomperdant, nomtournoi, annee),
FOREIGN KEY (nomgagnant) REFERENCES joueur (nom),
FOREIGN KEY (nomperdant) REFERENCES joueur (nom),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;
```

Pour lancer la création de la table, il faut appuyer sur « exécuter ». Nous créons ensuite les tables `sponsor` et `gain` avec les requêtes suivantes, à taper dans la fenêtre SQL. Même s'il est possible d'exécuter plusieurs requêtes à la suite, nous suggérons d'en taper une, puis de l'exécuter, et ensuite de taper la suivante, en vérifiant qu'elle a bien fonctionné.

```
CREATE TABLE sponsor(
nom varchar(30),
nomtournoi varchar(30),
```

```
annee int(4),
montant int(7),
PRIMARY KEY (nom, nomtournoi, annee),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;

CREATE TABLE gain(
nom varchar(30),
nomtournoi varchar(30),
annee int(4),
rang varchar(30),
prime int(6),
PRIMARY KEY (nom, nomtournoi, annee),
FOREIGN KEY (nom) REFERENCES joueur (nom),
FOREIGN KEY (nomtournoi) REFERENCES tournoi (nom)
) ENGINE=InnoDB;
```

Une fois ces tables créées, nous visualiserons le schéma de la base en cliquant sur le lien `tennis`, puis sur l'onglet `concepteur`

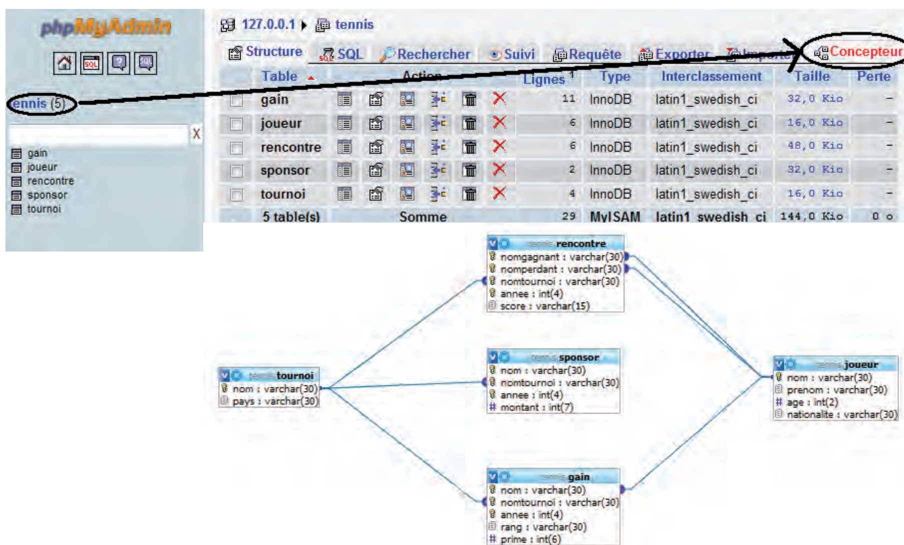
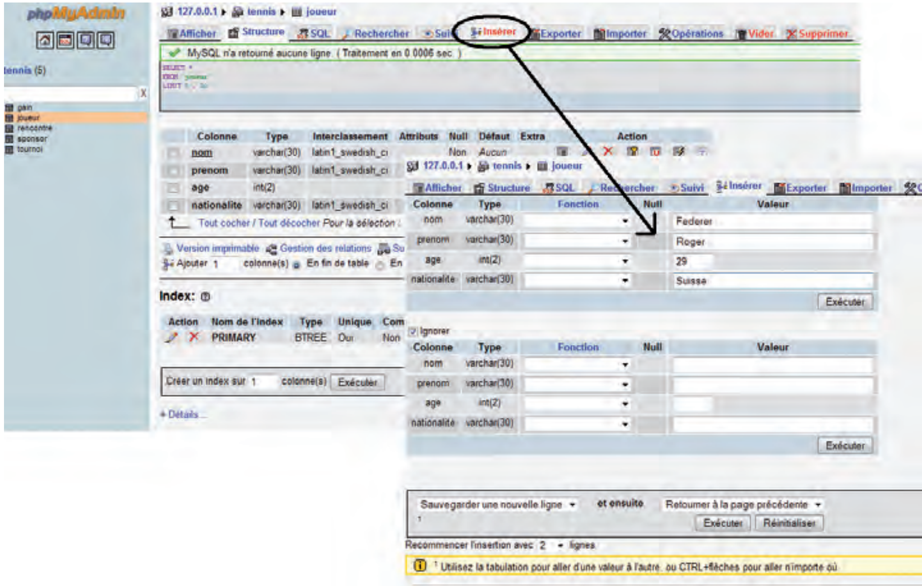


Schéma de la base.

Cela permet de contrôler que les contraintes d'intégrité référentielles sont bien placées correctement.

Insertion de données

En cliquant sur le nom d'une table dans le menu de gauche – par exemple `joueur` –, puis sur insérer, nous arrivons à un formulaire qui permet de remplir une table avec des données.



Formulaire d'insertion.

Une fois de plus, le code SQL est généré automatiquement et peut servir de patron pour écrire d'autres lignes d'insertion.

```
INSERT INTO joueur (nom, prenom, age, nationalite) VALUES
('Federer', 'Roger', '29', 'Suisse');
```

Nous devons nous rappeler les contraintes lorsque nous faisons des insertions : comme `nom` est la clé primaire de la table `joueur`, il sera impossible d'entrer deux fois un joueur dont le nom serait *Federer*. De même, il est impossible de créer une rencontre entre deux joueurs, si nous ne les avons pas préalablement insérés dans la table `joueurs`. De même, pour la table `sponsor`, il faut d'abord avoir entré le tournoi correspondant.

Question 3.

Ajoutez les joueurs Federer, Nadal, Murray, Soderling, Berdych et Djokovic.

Correction

```
INSERT INTO joueur (nom, prenom, age, nationalite) VALUES
('Nadal', 'Rafael', '24', 'Espagne'),
('Murray', 'Andrew', '23', 'Ecosse'),
('Soderling', 'Robin', '26', 'Suede'),
('Berdych', 'Tomas', '26', 'Rep. Tcheque'),
('Djokovic', 'Novak', '23', 'Serbie');
```

Nous vérifions que toutes les valeurs sont bien dans la table en cliquant sur `joueur` dans la barre de menu de gauche.

Question 4

Ajouter le sponsor BNP-Paribas au tournoi de Roland-Garros 2010 avec un montant de 9.000.000 et au tournoi Roland-Garros 2011 avec un montant de 10.000.000.

Correction. Il faut commencer par créer le tournoi de Roland-Garros, puis ajouter le sponsor pour l'année donnée. Une fois le tournoi créé, nous pourrons ajouter tous les sponsors que nous voulons.

```
INSERT INTO tournoi (nom, pays) VALUES
('Roland-Garros, 'France');

INSERT INTO sponsor (nom, nomtournoi, annee, montant) VA-
LUES
('BNP-Paribas', 'Roland-Garros, '2010', '9000000'),
('BNP-Paribas', 'Roland-Garros, '2011', '10000000');
```

Question 5

Ajouter les matchs suivants :

– Open d'Australie 2010 : Federer bat Murray (6/3-6/4-7/6)
– Roland-Garros 2010 : Nadal bat Soderling (6/4-6/2-6/4)
– Wimbledon 2010 : Nadal bat Berdych (6/3-7/5-6/4)
– US Open 2010 : Nadal bat Djokovic (6/4-5/7-6/4-6/2)
– Open d'Australie 2011 : Djokovic bat Murray (6/4-6/2-6/3)

Remplir également la table `gain`, en sachant que le gagnant remporte 1.000.000 et le finaliste 500.000.

Correction. Il faut d'abord créer les tournois inexistants. Le caractère d'échappement « \' » doit être utilisé pour indiquer l'apostrophe.

```
INSERT INTO tournoi (nom, pays) VALUES
```

```
('Open d\'Australie', 'Australie'),
('Wimbledon', 'Royaume Uni'),
('US Open', 'États-Unis');
```

Les joueurs ayant déjà été créés, nous entrons alors les rencontres

```
INSERT INTO rencontre (nomgagnant, nomperdant, nomtour-
noi, annee, score) VALUES
('Federer', 'Murray', 'Open d\'Australie', '2010', '6/3-6/4-
7/6'),
('Nadal', 'Soderling', 'Roland-Garros', '2010', '6/4-6/2-6/4'),
('Nadal', 'Berdych', 'Wimbledon', '2010', '6/3-7/5-6/4'),
('Nadal', 'Djokovic', 'US Open', '2010', '6/4-5/7-6/4-6/2'),
('Djokovic', 'Murray', 'Open d\'Australie', '2011', '6/4-6/2-
6/3');
```

Nous pouvons ensuite générer automatiquement les gains par les deux requêtes suivantes, que nous pouvons, bien entendu, aussi entrer à la main

```
INSERT INTO gain (nom, nomtournoi, annee, rang, prime)
SELECT nomgagnant, nomtournoi, annee, 'gagnant', 1000000
FROM rencontre;

INSERT INTO gain (nom, nomtournoi, annee, rang, prime)
SELECT nomperdant, nomtournoi, annee, 'finaliste', 500000
FROM rencontre;
```

Requêtes d'interrogation

Si l'on entre davantage de données dans la base, les requêtes donnent des résultats plus amusants. Lors d'une séance d'exercice, cette étape prend souvent beaucoup de temps, et il est préférable de distribuer un fichier contenant des instructions, `INSERT`. Ces fichiers peuvent être importés et exécutés via le menu prévu à cet effet dans PHPMyAdmin.

Voici quelques exemples de questions à poser, avec le code SQL de la requête correspondante. Bien entendu, selon le contenu de la base, l'énoncé de la question peut être modulé.

Requêtes faciles

– Afficher le `nom` des joueurs dont le `prenom` est Roger.

```
SELECT j.nom
FROM joueur j
WHERE j.prenom = 'Roger'
```

– Afficher les annee où s'est déroulé Roland-Garros. Indication : un tournoi s'est déroulé s'il y a des rencontres dans la base.

```
SELECT r.annee
FROM rencontre r
WHERE r.nomtournoi = 'Roland-Garros'
```

Requêtes avec jointure

– Afficher le nom et l'age des joueurs ayant gagné à Roland-Garros, peu importe l'année.

```
SELECT j.nom, j.age
FROM rencontre r, joueur j
WHERE r.nomgagnant = j.nom
AND r.nomtournoi = 'Roland-Garros'
```

– Afficher le nom des sponsors ayant sponsorisé un tournoi ayant lieu en France

```
SELECT s.nom
FROM sponsor s, tournoi t
WHERE t.nom = s.nomtournoi
AND t.pays = 'France'
```

Nous obtenons ici 2 résultats avec la même valeur, comme BNP-Paribas a sponsorisé un tournoi français pendant plusieurs années. On peut ajouter le mot-clé `distinct` pour enlever les doublons, ou bien faire un regroupement dans une clause `GROUP BY`.

– Afficher le nom et prenom des joueurs ayant gagné un tournoi sponsorisé par BNP-Paribas

```
SELECT j.nom, j.prenom
FROM joueur j, rencontre r, sponsor s
WHERE j.nom = r.nomgagnant
AND r.nomtournoi = s.nomtournoi
AND s.nom = 'BNP-Paribas'
```

Requêtes ensemblistes ou imbriquées

– Afficher le nom de tous les joueurs ayant joué au moins un match.

```
SELECT r.nomgagnant as nom
FROM rencontre r
UNION
```

```
SELECT r.nomperdant as nom
FROM rencontre r
```

– Afficher le nom des joueurs ayant joué à Wimbledon quelle que soit l'annee.

```
SELECT r.nomgagnant as nom
FROM rencontre r
WHERE r.nomtournoi = 'Wimbledon'
UNION
SELECT r.nomperdant as nom
FROM rencontre r
WHERE r.nomtournoi = 'Wimbledon'
```

– Afficher le nom des joueurs ayant gagné tous leurs matchs.

```
SELECT r.nomgagnant as nom
FROM rencontre r
EXCEPT
SELECT r.nomperdant as nom
FROM rencontre r
```

La commande EXCEPT n'étant pas implémentée en MySQL, il faut utiliser une requête imbriquée.

```
SELECT distinct (r.nomgagnant) as nom
FROM rencontre r
WHERE r.nomgagnant NOT IN (
SELECT p.nomperdant as nom
FROM rencontre p
)
```

– Afficher le nom des joueurs n'ayant jamais gagné moins de 1.000.000 par tournoi.

```
SELECT distinct(g.nom)
FROM gain g
WHERE g.prime >= 1000000
AND g.nom
NOT IN (
SELECT h.nom
FROM gain h
WHERE h.prime <1000000
)
```

On aurait de même pu utiliser un EXCEPT.

-

Requêtes agrégats

– Compter le nombre total de matchs joués à l'US Open et afficher le résultat dans une colonne nommée nbmatchs.

```
SELECT COUNT(*) as nbmatch
FROM rencontre r
WHERE r.nomtournoi = 'US Open'
```

– Calculer les gains totaux de 'Nadal' dans une colonne 'gaintotal'

```
SELECT SUM(g.prime) as gaintotal
FROM gain g WHERE g.nom = 'Nadal'
```

– Afficher le nom et les gains (dans une colonne 'gaintotal) des joueurs ayant gagné au moins 1000000 au total.

```
SELECT g.nom, SUM(g.prime) as gaintotal
FROM gain g
GROUP BY g.nom
HAVING SUM(g.prime) >= 1000000
```

– Afficher les nom et annee des tournois où la somme versée par les sponsors est supérieure aux gains reversés aux participants.

```
SELECT s.nomtournoi, s.annee
FROM sponsor s, gain g
WHERE g.nomtournoi = s.nomtournoi
AND g.annee = s.annee
GROUP BY g.nomtournoi, g.annee
HAVING SUM(montant) > SUM(prime)
```

Notons que cette requête ne retourne pas les tournois pour lesquels il n'y a pas eu de matchs, car la somme sur l'ensemble vide n'est pas définie.

Exercices non corrigés

Dans un exercice consacré aux bases de données, le plus classique est de demander l'écriture de requêtes permettant de répondre à certaines questions. Il faut donc préalablement avoir défini la structure de la base de données, soit par son schéma entité-association, soit en donnant directement les tables relationnelles, ce qui est souvent préférable, car ainsi tous les élèves utiliseront les mêmes noms de tables. On peut également demander aux élèves de modéliser une situation à partir d'un texte et de produire un modèle entité-association et puis des tables. Cette question est en réalité *difficile*. En pratique, elle demande souvent des allers-retours avec la personne ayant écrit le texte, ce qu'il est possible de faire en TD, mais impossible en examen. Nous conseillons donc dans

un premier temps de se restreindre à des requêtes sur un schéma déjà existant, ou bien à un énoncé très précis à modéliser.

Requêtes sur la base de données des joueurs de football

Exercice 1

Il faut s'imaginer que la table contient un grand nombre de données – par exemple, toutes les informations sur la L1 et L2 de football.

Requêtes sur une seule table sans jointure :

1. Nom et NomCourt de toutes les équipes.
2. NomCourt de toutes les équipes domiciliées à Paris.
3. Nom de toutes les équipes dont la ville commence par un « L ».
4. Nom de tous les joueurs dont le prénom est « Stéphane ».

Requêtes avec jointure :

1. Nom des équipes où joue au moins un joueur dont le nom commence par un « A ».
2. Nom des équipes où il n'y a aucun joueur dont le nom commence par « A ».
3. Nom des joueurs ayant participé à un match où au moins un but a été marqué.
4. Nom des équipes n'ayant marqué aucun but.
5. Nom des joueurs ayant joué un match le 19/12/10.
6. Dates des matchs où un joueur de plus de 35 ans a joué.

Requêtes avec agrégats :

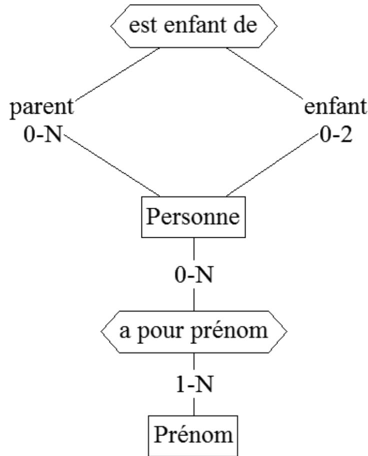
1. Moyenne d'âge de l'ensemble des joueurs.
2. Moyenne d'âge par équipe : indiquer le nom de l'équipe dans une colonne et la moyenne d'âge dans une autre colonne nommée AgeMoyen.
3. Nom et Prénom du joueur le plus vieux.
4. Nom et Prénom du joueur le plus vieux de chaque équipe.
5. Nom et Prénom du joueur le plus vieux de chaque rencontre.
6. Nombre total de buts inscrits par chaque équipe.
7. Nom des équipes ayant marqué au total plus de 40 buts.
8. Nom des équipes ayant marqué au total moins de 10 buts.
9. Nombre total de matchs joués par chaque équipe.

Une base de données généalogique

Exercice 2

Dans cet exercice, on considère une base de données très simple sur des personnes. Une personne doit posséder de 0 à 2 parents et chaque personne

peut avoir de 0 à *N* enfants. Chaque personne possède un nom, un numéro de Sécurité sociale (NSS) qui est un identifiant unique, un sexe (M/F). Un prénom est défini par un attribut valeur qui est également l'identifiant unique de l'entité. Nous avons construit une partie du modèle entité-association :



Modèle entité-association des personnes.

Questions de cours

1. Le modèle entité-association n'est pas complet. Ajouter les attributs nécessaires. Redessiner l'ensemble du modèle sur une feuille pour correspondre au texte explicatif.
2. Pourquoi a-t-on voulu créer une entité Prenom au lieu d'utiliser tout simplement un attribut ?
3. À quoi correspondent sur le schéma les mots parent et enfant ?
4. À quoi correspond la notation 0-2 sur le schéma ?

Dans les questions suivantes, on a transformé le modèle entité-association en modèle relationnel suivant, dans lequel les clés primaires sont soulignées.

```
prenom(valeur :chaîne)
personne(NSS :entier, nom :chaîne, sexe :chaîne)
apourprenom(NSS :entier, valeur :string)
enfantde(NSSParent :entier, NSSEnfant :entier)
```

1. Donner les contraintes de clés étrangères de ce schéma, afin de respecter le modèle entité-association.

2. En utilisant les tables du modèle relationnel, donner la requête SQL permettant d'afficher les NSS de tous les petits-enfants de toutes les personnes ayant comme prénom « Joe ».

3. Avec le modèle relationnel de base, peut-on écrire une requête permettant de trouver tous les descendants d'une personne donnée ? Pourquoi ?

4. Donner l'algorithme d'un programme, incluant des requêtes SQL et des instructions de boucle qui permettraient de réaliser une telle fonctionnalité.

La base de données d'un moteur de recherche

Exercice 3

Nous considérons, dans cet exercice, un moteur de recherche sur le Web, comme Google. Pour simplifier, nous ne nous intéressons pas au *ranking* des pages, c'est-à-dire à leur importance, mais uniquement au problème de retourner des pages contenant un mot-clé précis. Nous utilisons, pour ce faire, la base de données appelée recherche dont le schéma relationnel est donné dans la figure suivante, où les clés primaires sont soulignées.

```
page(URL : chaîne, titre : chaîne)
dico(id : entier, mot : chaîne)
index(id : entier, URL : chaîne)
Avec les contraintes de clé étrangères :
index(URL) REFERENCES page(URL) index(id) REFERENCES
dico(id)
```

Le champ titre de la table page représente le contenu de la balise HTML `<title>` de la page correspondant à l'URL. Nous ne nous posons pas le problème de savoir comment nous avons obtenu les mots de la page à partir du code HTLM de la page, qui est un problème en soi. Nous supposons données la table page et la table index correspondant à un ensemble de pages web.

1. Nous voulons ajouter la page d'URL `http://www.uvsq.fr/in111/cours. html` dans la base. Supposons que cette page contienne les mots « base de données », « cours » et « informatique » et que son titre soit « IN111 ». Sans écrire de code, expliquer ce qu'il faudra faire pour compléter correctement les tables de la base.

2. Donner les URL des pages dont le titre est « base de données ».

3. Donner les couples d'URL ayant au moins un mot en commun. Le résultat de la requête devra contenir deux champs `page1` et `page2`.

4. Donner, pour chaque mot apparaissant dans la base, le nombre de pages différentes dans lesquelles il apparaît et classer ces mots par ordre de fréquence croissante.

5. Donner les URL des pages contenant au moins 100 mots différents.

6. Donner la liste des mots apparaissant dans au plus 10 pages différentes.

7. Donner la liste des mots apparaissant dans toutes les pages de la base de données.

Une base de données géographique

Exercice 4

On se donne le schéma suivant d'une base de données.

```
Pays (NumPays :entier, Nom :chaîne, RefPresident :entier,
                    RefContinent :entier)
Presidents (NumPresident :entier, Nom :chaîne, Prenom
                    :chaîne, Age :entier)
   Continents (NumContinent :entier, Nom :chaîne)
     Superficie (RefPays :entier, Valeur :réel)
    Population (RefPays :entier, Valeur :entier)
```

Les clés primaires sont indiquées en souligné. Les clés étrangères sont données par les contraintes suivantes :

```
Pays(RefPresident) references Presidents(NumPresident)
Pays(RefContinent) references Continents(NumContinent)
     Superficie(RefPays) references Pays(NumPays)
    Population(RefPays) references Pays(RefPays)
```

Cette base de données simpliste permet de modéliser quelques informations sur les pays du monde. La table `Presidents` contient des informations sur les présidents en exercice – on appelle « Président » toute personne qui dirige un pays. La table `Continents` contient les noms des continents, par exemple Europe, Afrique, Amérique du Nord, Asie, etc. Les tables `superficie` et `population` donnent des informations supplémentaires sur les pays.

1. Dans quel ordre doit-on créer les tables dans la base de données ?

2. Avec la structure proposée, un pays peut-il avoir plusieurs présidents ? Un président peut-il être président de plusieurs pays différents ?

3. Donner un schéma dans le format entité-association de cette base qui soit compatible avec les tables relationnelles.

4. Proposer un autre schéma relationnel avec moins de tables. Quel est l'intérêt de la modélisation proposée dans l'énoncé ?

5. Nom du pays dont le NumPays = 1.

6. Donner le nom du président de la France.

7. Donner le nom du pays d'Europe le plus peuplé.

8. Donner la superficie totale de l'Europe.

9. Donner la somme totale du nombre de personnes habitant dans un continent d'une superficie < 15.000.000 km^2.

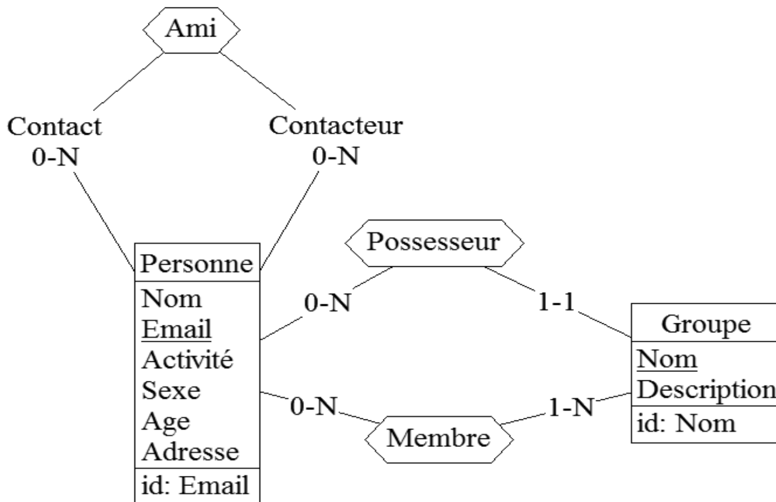
10. Compter le nombre de pays d'Afrique dont la population est > 50.000.000 d'habitants.

11. Donner la liste de tous les pays dont la densité, c'est-à-dire le quotient de la population par la superficie, est inférieure à 20.

La base de données d'un réseau social

Exercice 5

`Facebook.com` est un logiciel de réseau social qui permet à des amis de se constituer en réseaux sociaux. Ce site, qui existe depuis 2004, a atteint dernièrement une popularité record, avec près de 600 millions d'utilisateurs actifs en 2011. Nous allons étudier dans cet exercice une modélisation simpliste du système. Nous considérons dans un premier temps le modèle entité-association suivant.



1. Donner les instructions CREATE TABLE nécessaires pour implémenter ce modèle. Penser à bien définir les clés primaires de chaque table.

2. On souhaite indiquer que le couple (nom, adresse) est une clé candidate. Comment faire ?

3. On souhaite ajouter un attribut photo à l'entité Personne. Donner l'instruction SQL permettant de le faire. Cet attribut sera de type BLOB (*Binary Large OBject)*. De même, on veut ajouter un attribut password. Quels types SQL peut-on utiliser pour cet attribut ? Donner les avantages et inconvénients de chacun.

Requêtes SQL :

1. Nom, Prenom et email de tous les utilisateurs de la base.

2. Email de tous les amis de l'utilisateur `robert@tartempion.fr.`

3. Email de tous les amis des amis de l'utilisateur `robert@tartempion.fr.` Peut-on donner l'email de toutes les personnes liées à un degré de profondeur quelconque à `robert@tartempion.fr` ?

4. Nombre de membres du groupe *Club Tennis Versaillais*.

5. Pour chaque groupe, le nom du groupe et l'âge moyen des membres de tous les groupes appartenant à `robert@tartempion.fr`.

6. Donnez l'email des personnes dont tous les amis font partie de tous les groupes dont fait partie cette personne.

7. Donnez les noms des groupes où certains des membres ne se connaissent pas directement.

Exercice de modélisation : une banque

Exercice 6

La Banque des Petits Suisses (BPS) souhaite réaliser une base de données pour gérer les comptes de ses clients. Les informations à stocker sont les suivantes : chaque client est identifié par un numéro de client unique et possède un nom, un prénom et une adresse. Un compte est composé d'un numéro de compte et d'un montant initial déposé à l'ouverture du compte. Chaque client peut posséder autant de comptes qu'il le souhaite. La banque souhaite ensuite conserver une trace de toutes les opérations effectuées sur les comptes, qui peuvent être soit des dépôts soit des retraits.

Réaliser le modèle entité-association qui permettra de représenter les données de la BPS, puis donner le schéma relationnel correspondant. Écrire une requête permettant de calculer le solde d'un compte dont on connaît le numéro.

Questions d'enseignement

Dans ce paragraphe, nous donnons quelques pistes de réflexion pour mettre en place un cours d'introduction aux bases de données. Il est structuré autour des questions qui se posent quand on organise un tel cours.

Quels objectifs se fixer ?

Dans ce type de cours, il est raisonnable de s'orienter vers l'utilisation des bases de données. En effet, les cours de ce niveau sont suivis par des élèves qui vont poursuivre des études d'informatique, mais aussi par d'autres qui vont suivre des voies très différentes. À l'issue du cours, les élèves devront avoir acquis les connaissances nécessaires pour manipuler une base de données et un système de gestion de bases de données.

Quels thèmes aborder ?

Les sujets que l'on peut traiter dans un cours consacré aux bases de données sont nombreux et vastes. Le choix de l'objectif ci-avant va toutefois nous guider.

Tout d'abord, il faut décider quels modèles logiques présenter. Il se trouve que le modèle relationnel possède deux atouts importants : c'est le modèle le plus répandu, et il repose sur un nombre réduit de concepts à assimiler. C'est donc un choix assez naturel.

Les concepts du modèle relationnel vont être présentés. Pour faciliter leur assimilation, il est intéressant de s'appuyer sur des exemples présentés sous forme de tableau. Le lien entre n-uplet et ligne et entre attribut et colonne permet de mieux comprendre les définitions formelles.

La deuxième décision concerne le choix du langage d'interrogation. Il est difficile de faire l'économie de SQL, qui est le langage standard des bases de données relationnelles. Cependant, il peut être plus simple de ne pas introduire directement la syntaxe du langage. Une approche possible consisterait à utiliser une interface pour les premiers contacts. Par exemple, le logiciel *PhpMyAdmin* fournit une interface d'administration pour le système de gestion de bases de données *MySQL*. L'intérêt de ce type d'outil est qu'en plus de l'interface conviviale, il montre le SQL exécuté pour chaque action, ce qui facilite l'acquisition de la syntaxe. Une autre solution serait de débuter par le langage *Query by Example*. Il est par exemple implémenté dans le système de gestion de bases de données *Microsoft Access* qui fait partie de la suite bureautique *Office*. Ce langage est un langage graphique et intuitif pour l'interrogation d'une base de données. Les requêtes QBE peuvent être traduites facilement en SQL pour aider à acquérir ce dernier. Enfin, concernant SQL, il est important de se foca-

liser sur la norme et non pas sur l'un des nombreux dialectes implantés au sein des systèmes de gestion de bases de données. En effet, tant sur le point de la syntaxe que de la sémantique, certains systèmes de gestion de bases de données prennent d'importantes libertés vis-à-vis de la norme.

Une fois ces deux points traités, l'objectif fixé est atteint : un élève peut manipuler un système de gestion de bases de données. Pour compléter le cours, on suivra plusieurs pistes. Aborder la conception de bases de données donnera à l'élève des éléments pour élaborer ses propres bases de données. Étudier les mécanismes internes des systèmes de gestion de bases de données, comme par exemple l'optimisation de questions, montrera les problèmes que résolvent les systèmes de gestion de bases de données. Enfin, l'ouverture vers des langages de programmation et vers le Web donnera des perspectives intéressantes et attractives.

Comment organiser le cours ?

L'acquisition des connaissances décrites ci-avant nécessite une mise en pratique importante. Le cours doit s'appuyer sur des exemples pour illustrer les concepts nouveaux, voire intégrer des exercices ciblés. Les séances de travaux dirigés et pratiques serviront à asseoir les connaissances présentées en cours. Il est raisonnable de prévoir deux fois plus de travaux dirigés et de travaux pratiques que de cours.

En termes d'enchaînement des thèmes, après une introduction définissant et motivant les bases de données et les systèmes de gestion de bases de données, la présentation du modèle relationnel semble naturelle. Il serait également possible de débuter par la conception de bases de données avec le modèle entité-association par exemple. Toutefois, pour un cours d'introduction, ce choix impose de demeurer à un niveau d'abstraction élevé pendant plusieurs séances, ce qui n'est pas souhaitable. La suite logique consiste à présenter un langage d'interrogation. Une fois ces bases présentées, l'ordre des éventuels autres thèmes n'est pas contraint.

Quels outils choisir pour les travaux pratiques ?

Pour assimiler les concepts vus en cours, il est utile de les mettre en pratique. Pour cela, il faut tout d'abord choisir un système de gestion de bases de données. Il en existe de nombreux, tant commerciaux que gratuits ou libres. Nous allons ici suggérer quelques pistes de choix.

La première possibilité consiste à utiliser *Microsoft Access* qui est intégré à *Office* ou *OpenOffice Base* qui fait partie d'OpenOffice. Ces deux outils proposent une interface conviviale pour la création et la manipulation d'une base

de données relationnelle. Par défaut, l'interrogation se fait à l'aide d'un langage de type *Query by Example*, mais il est aussi possible d'accéder au code SQL. Le support de SQL dans ces deux outils reste cependant un peu limité.

Il existe également des systèmes de gestion de bases de données légers et simples à utiliser. Le système Firebird (`http://www.firebirdsql.org/`) en est un exemple. Ce genre de système propose en général toutes les fonctionnalités nécessaires pour l'apprentissage et est bien adapté à des bases de données de taille moyenne.

Une autre possibilité consiste à se tourner vers les systèmes de gestion de bases de données libres les plus connus : *PostgreSQL* (`http://www.postgresql.org/`) ou *MySQL* (`http://www.mysql.com/`). L'installation et la maintenance de ces outils sont parfois un peu plus complexes que les précédents. En contrepartie, les fonctionnalités disponibles sont d'un excellent niveau. De plus, ils s'intègrent bien dans l'installation plus globale d'un serveur web, comme par exemple avec le package *easyPHP*. Il est à noter que ces logiciels présentent l'avantage de pouvoir être installés par les utilisateurs sur leur ordinateur personnel.

Enfin, les systèmes de gestion de bases de données commerciaux phares comme Oracle, DB2 d'IBM ou SQL Server de Microsoft disposent d'une version gratuite plus légère, utilisable dans un contexte privé. Ces systèmes sont un peu plus délicats à installer et à administrer, mais leur manipulation aide à se faire une idée des outils utilisés dans un contexte professionnel.

En dehors des systèmes de gestion de bases de données, les outils CASE (*Computer-aided software engineering*) fournissent une aide pour la conception de bases de données. Le logiciel gratuit *DB-Main* (`http://www.db-main.be/`) pour la modélisation. Il permet entre autres de concevoir un diagramme entité-association puis de le transformer en relationnel.

Enfin, pour le développement, les environnements de développement intégré que sont *Eclipse* (`http://www.eclipse.org/`) et *Netbeans* (`http://netbeans.org/`) permettent de programmer dans divers langages.

Compléments

Dans ce chapitre, nous avons abordé un certain nombre de sujets à propos des bases de données et du Web. Certains des thèmes présentés mériteraient d'être développés et ne l'ont pas été par manque de place.

Concernant les systèmes de gestion de bases de données, l'étude des mécanismes internes comme la gestion des transactions ou l'organisation physique des données est nécessaire à une bonne compréhension et à un usage

plus poussé. Dans le même ordre d'idée, la compréhension des concepts de l'administration de bases de données permet une manipulation avancée des systèmes de gestion de bases de données. Sur le plan des modèles, d'autres modèles logiques que le modèle relationnel existent : les modèles réseau et hiérarchique sont antérieurs alors que les modèles objet, relationnel-objet et XML sont plus récents.

Système de gestion de bases de données

L'objectif d'un système de gestion de bases de données est de factoriser des modules de contrôle répondant aux besoins communs des applications : interrogation, cohérence, partage, etc. Plus précisément, ces objectifs sont les suivants : rendre les applications indépendantes du modèle physique de stockage comme le disque magnétique ou la mémoire flash – indépendance physique –, donner la possibilité de disposer de vues logiques de la base de données – indépendance logique –, fournir un langage d'interrogation déclaratif – un langage déclaratif est un langage qui permet de décrire le résultat que l'on souhaite obtenir, par exemple, « les joueurs de moins de 21 ans qui ont marqué plus de 10 buts », par opposition à un langage impératif, comme C ou Java, qui décrit une séquence d'opérations permettant d'arriver à la solution –, transformer automatiquement une question en programme – optimisation des questions –, gérer la cohérence des données, être robuste aux pannes – tolérance aux pannes –, gérer les accès concurrents de différents utilisateurs ou programmes, contrôler les accès aux données – gestion de la confidentialité – et enfin s'appuyer sur des standards.

Le groupe ANSI/X3/SPARC a proposé une architecture des systèmes de gestion de bases de données comportant trois niveaux. L'objectif de cette proposition est de rendre les vues des utilisateurs indépendantes de la façon dont les données sont stockées. Parmi ces trois niveaux, le niveau *externe* décrit le point de vue des utilisateurs, le niveau *conceptuel* stocke le schéma des données et le niveau *interne* gère le stockage physique des données. En plus des niveaux de représentation, la norme décrit les modules d'un système de gestion de bases de données. Une partie de ces modules est dédiée à la description de données, la seconde assure la manipulation des données. Ces deux ensembles de modules s'appuient sur le *dictionnaire de données*. Ce dernier est une base de données décrivant l'ensemble des objets manipulés par le système de gestion de bases de données.

Pour conclure sur les systèmes de gestion de bases de données, il nous faut évoquer quelques produits en commençant par un bref historique. Le premier prototype de système de gestion de bases de données relationnel

nommé *System/R* a été développé par IBM dans son centre de San José. Il donnera naissance à *SQL/DS*, puis à *DB2*. En parallèle, à l'université de Berkeley, le système *INGRES* est développé. Il supporte un langage de requête nommé *QUEL*. Le système *POSTGRES* succéda à *INGRES* et est aujourd'hui l'un des systèmes de gestion de bases de données *open source* le plus utilisé sous le nom *PostgreSQL*. *INGRES* a également donné naissance aux systèmes de gestion de bases de données *Informix et Sybase*. Ce dernier évolua ensuite vers *SQL Server* chez Microsoft. Les premières offres commerciales sont apparues avec *Oracle* en 1979, *SQL/DS* (IBM) en 1982, suivi de *base de données2* (IBM) en 1983. La première version du système *open source MySQL* date de 1994. Enfin, pour la gestion de bases de données de moindre taille, on peut citer *Microsoft Access*, *OpenOffice Base* et *Paradox*.

Le modèle relationnel

Le modèle relationnel est un modèle de données proposé en 1969 par Edgar T. Codd à IBM San-José. Dans sa forme initiale, seules des valeurs atomiques sont supportées, c'est-à-dire qu'il n'y a ni valeurs complexes, ni valeurs multivaluées. On parle de *première forme normale* (*1NF*). Une relation est en 1NF si le domaine de chacun de ses attributs ne contient que des valeurs atomiques. Ce type de relation est bien adapté aux applications de gestion. Cependant, cette contrainte peut être pénalisante dans le cas d'applications manipulant des objets complexes. C'est pour cette raison que des extensions au modèle relationnel ont vu le jour. On parle alors de bases de données en *non première forme normale* (NF^2). Parmi les extensions proposées, on peut citer le *modèle relationnel imbriqué* qui autorise qu'une valeur d'un domaine soit lui-même une relation et le *modèle objet-relationnel* qui intègre au relationnel les concepts objet.

Le modèle relationnel et les valeurs manquantes

Ce que nous avons présenté jusqu'alors du modèle relationnel supposait que chaque n-uplet possédait une valeur pour chaque attribut de la relation. Cependant, pour une application réelle, une telle supposition est déraisonnable. Il est donc nécessaire d'introduire un moyen de représenter l'absence de valeur – valeur manquante. Citons trois exemples de situations où cela peut se produire : l'information est pertinente mais n'existe pas pour l'entité correspondant au n-uplet – numéro d'immatriculation d'une personne ne possédant pas de voiture –, l'information n'est pas pertinente pour l'entité – le nom de jeune fille pour un homme – ou l'information existe mais est inconnue au moment de la saisie.

L'absence d'une valeur dans une relation est représentée par un marqueur spécial nommé *NULL*. L'utilisation d'un unique marqueur pour représenter les valeurs non renseignées pose le problème de son interprétation. Il est en effet impossible de déterminer à quelle situation correspond une valeur manquante dans une relation.

De plus, la prise en compte de ce marqueur dans les langages d'interrogation est complexe – logique à trois valeurs, logique à quatre valeurs. Nous verrons ci-après le cas du langage SQL.

En résumé, les marqueurs NULL sont délicats à manipuler et il est préférable de les éviter dans la mesure du possible. L'étude détaillée des implications de ce marqueur sort du cadre de cet ouvrage.

Les langages d'interrogation

Plusieurs langages ont été proposés pour l'interrogation d'une base de données relationnelle. Le *calcul relationnel de n-uplets* proposé par Codd est à l'origine des langages QUEL et SQL. Le *calcul relationnel de domaines* est plus proche de la logique du premier ordre. Le langage *Query by Example* (*QBE*) est une application pratique du calcul de domaines. Le langage *datalog* est un autre langage logique. Les relations entre les bases de données et la logique ont été intensivement étudiées. L'*algèbre relationnelle* est un ensemble d'opérations que l'on peut appliquer aux relations. Enfin, le langage *SQL* tire sa source du calcul de n-uplets et de l'algèbre relationnelle. Hormis l'algèbre relationnelle, ces langages sont tous déclaratifs, c'est-à-dire que l'on décrit ce que l'on veut obtenir mais pas la façon de l'obtenir. L'algèbre relationnelle est le seul langage à proposer des opérations qui doivent être ordonnées. L'étude de ces langages complets a montré qu'ils ont tous le même pouvoir d'expression. Ces langages sont dits *complets*, c'est-à-dire qu'ils ont le même pouvoir d'expression que la logique du premier ordre.

L'algèbre relationnelle

Parmi les opérateurs qui ont été décrits dans la littérature, certains peuvent être définis à partir d'autres. Il existe en fait plusieurs sous-ensembles minimaux d'opérateurs qui permettent d'écrire toute expression d'algèbre relationnelle. Par exemple, l'ensemble $\{\sigma,\pi,\times,\backslash,\cup\}$ est suffisant pour écrire toutes les formules d'algèbre relationnelle.

L'algèbre relationnelle décrit un ensemble d'opérateurs qui possèdent des propriétés algébriques. Par exemple, la propriété $\sigma_F(\sigma_G(r)) = \sigma_G(\sigma_F(r))$ met en évidence la commutativité de la composition de sélections. De telles propriétés sont à la base de l'*optimisation de questions*, c'est-à-dire la refor-

mulation d'une requête en une requête équivalente. Elles permettent de manipuler de manière symbolique des expressions d'algèbre relationnelle et de montrer l'équivalence entre certaines formules.

Chaque expression peut être représentée sous la forme d'un *arbre algébrique*. Chaque nœud de l'arbre représente un opérateur algébrique. Pour que l'expression soit évaluée, le système de gestion de bases de données doit remplacer chaque opérateur par l'algorithme réalisant l'opération. On obtient alors un *plan d'exécution* de la requête. C'est ce plan d'exécution que manipule le système de gestion de bases de données pour faire de l'optimisation de requêtes. Elle consiste à trouver le plan d'exécution le plus efficace pour une requête. La recherche du meilleur plan est un problème combinatoire pour lequel il est difficile de trouver une solution exacte. Le système de gestion de bases de données se contente donc en général d'un bon plan d'exécution. La recherche du plan d'exécution peut se faire à deux niveaux : au niveau algébrique, le système de gestion de bases de données va manipuler les expressions en utilisant les propriétés, au niveau physique, il va choisir les algorithmes les mieux adaptés. Pour choisir les expressions et les algorithmes à utiliser, il s'appuie sur les propriétés (*rule based optimization*) et sur un modèle de coût utilisant des statistiques sur les données (*cost based optimization*).

Prenons l'exemple de la requête suivante, permettant de trouver les noms des villes où il y a une équipe possédant un joueur de plus de 35 ans.

```
SELECT DISTINCT e.ville
FROM equipes e JOIN joueurs j ON e.nomCourt = j.equipe
WHERE j.age > 35
```

Cette requête est divisée en un certain nombre d'opérateurs de l'algèbre relationnelle. On voit qu'il est possible d'écrire plusieurs requêtes équivalentes en termes de résultat. En revanche, certaines pourront être plus rapides à exécuter que d'autres.

$$r_1 = \pi_{ville}(\sigma_{age>35}(equipes \bowtie_{nomCourt=equipe} \text{joueurs}))$$
$$r_2 = \pi_{ville}(equipes \bowtie_{nomCourt=equipe} \sigma_{age>35}(\text{joueurs}))$$

Le coût d'une jointure dépend de l'algorithme utilisé. Même s'il existe des algorithmes plus efficaces, pour illustrer notre propos simplement, nous considérons ici un algorithme naïf de jointure qui effectue une double boucle, c'est-à-dire parcourt tous les n-uplets de `equipes` et pour cha-

cun parcourt tous les n-uplets de `joueurs` et conserve ceux pour qui la condition de jointure est vraie. Un tel algorithme possède une complexité $O(|equipes| \times |joueurs|)$. Pour comprendre la différence de performance entre ces deux algorithmes, il faut faire des hypothèses sur le contenu des tables. Supposons qu'il y ait 20 équipes, 750 joueurs et 10 joueurs de plus de 35 ans. Dans ce cas r_1 fera 20 × 750 = 15000 comparaisons pour la jointure, puis parcourra les 750 lignes résultant de cette étape pour sélectionner les 10 villes des joueurs de plus de 35 ans, enfin la projection parcourra les 10 villes pour ne sélectionner que le résultat, soit un coût total c_1 = 20 × 750 + 750 + 10 = 15760. En revanche r_2 commence par parcourir les 750 joueurs pour trouver les 10 de plus de 35 ans, puis pour ces 10 joueurs, parcourt les 20 villes, et enfin la projection parcourt les 10 n-uplets restant après la jointure pour un coût total c_2 =750 + 20 × 10 + 10 = 960.

On voit donc qu'il est possible d'optimiser des requêtes SQL, et qu'une connaissance des cardinalités – c'est-à-dire le nombre de n-uplets dans une table, ou répondant à une sélection simple – permet une optimisation avancée. Nous avons illustré une des optimisations les plus simples et qui fonctionne toujours : descendre les sélections en dessous des jointures.

SQL

Le langage *SEQUEL* a été développé au début des années 1970 par Donald D. Chamberlin et Raymond F. Boyce (IBM) pour le système de gestion de bases de données *System/R*. SEQUEL est rapidement devenu le langage SQL. La description de SQL donné dans ce chapitre est très succincte et se concentre sur les concepts fondamentaux. Pour une description détaillée de SQL, le lecteur pourra s'orienter vers l'abondante littérature sur le sujet ou vers le site `http://sqlpro.developpez.com/` qui en fait une très bonne présentation.

Valeur manquante en SQL

La gestion des valeurs manquantes en SQL est réalisée grâce au marqueur `NULL`. Signalons tout d'abord que `NULL` est bien un marqueur et non pas une valeur : il indique l'absence de valeur pour un attribut d'un n-uplet. La conséquence de ce fait est qu'il est inapproprié de comparer une valeur avec `NULL`. Le marqueur `NULL` n'est ni égal, ni différent d'une valeur. Les tests spécifiques `IS NULL` et `IS NOT NULL` servent à vérifier si une valeur est manquante. Un prédicat classique comportant des marqueurs `NULL` sera évalué soit à vrai, soit à faux, soit à *inconnu* en fonction des cas. C'est cette *logique à trois valeurs* qui peut rendre les requêtes complexes en présence

de NULL. Des logiques avec encore plus de valeurs existent, par exemple la logique à quatre valeurs, qui permet justement de préciser le rôle attendu de la valeur nulle : vrai, faux, inconnu et incohérent. Pour plus de détails sur cette question, on pourra se référer au site http://sqlpro.developpez.com/cours/null/.

Conception de bases de données relationnelles

Le modèle entité-association a été initialement proposé en 1976 par Peter Chen. Il a ensuite été étendu vers des modèles plus expressifs puis vers des modèles objet. Il existe différentes variantes de la notation, parfois comportant des différences subtiles. Il faut donc être particulièrement attentif et rigoureux par rapport au formalisme utilisé. Un point particulièrement délicat concerne la syntaxe et la sémantique des cardinalités des types d'association. Par exemple, l'emplacement des cardinalités d'une association binaire sur un diagramme entité-association selon le formalisme original est inversé par rapport à un modèle conceptuel de données de la méthode *Merise*. Pour ajouter à la confusion, un diagramme de classe UML utilise, à la place des cardinalités, la notion de multiplicité qui possède une syntaxe proche du formalisme entité-association original mais diffère sur la sémantique. Pour terminer sur les modèles conceptuels, signalons que la réalisation d'un modèle conceptuel représentant un problème donné est une tâche ardue. Cette tâche fait appel à des techniques d'analyse des besoins qui sortent du cadre du présent ouvrage.

Pour aller plus loin

Un bon complément à ce cours est le livre Georges Gardarin, *Bases de données - Objet et relationnel*, Eyrolles, 2003.

Il brosse un panorama très complet des concepts des bases de données avec un niveau de détail qui le rend très accessible. Du fait de l'expérience de son auteur, ce livre présente également les mécanismes internes des systèmes de gestion de bases de données. Enfin, à la fin de chaque chapitre, une liste de références commentées permet au lecteur intéressé d'obtenir plus de détails. D'autres livres abordant les thématiques des bases de données dans leur ensemble sont reconnus comme de très bons supports.

Laurent Audibert, *Bases de données - De la modélisation au SQL*, Ellipses, 2009.

Hector Garcia-Molina, Jeffrey David Ullman et Jennifer Widom, *Database Systems : The Complete Book*, Prentice-Hall, 2008.

Raghu Ramakrishnan et Johannes Gehrke, *Database Management Systems*, McGraw-Hill, 2003.

Avi Silberschatz, Henry F. Korth et S. Sudarshan, *Database System Concepts*, McGraw-Hill, 2010.

Christopher John Date, *Introduction aux bases de données*, Vuibert, 2004.

Ramez Elmasri et Shamkant Navathe, *Conception et architecture des bases de données*, Pearson Education, 2004.

Mokrane Bouzeghoub, Mireille Jouve et Philippe Pucheral, *Le modèle relationnel - Algèbre, langages, applications*, Hermès - Lavoisier, 1998.

Concernant la conception de bases de données, on peut citer

Jean-Luc Hainaut, *Bases de données : concepts, utilisation et développement*, Dunod, 2009.

Heikki Mannila and Kari-Jouko Räihä, *The Design of Relational Databases*, Addison-Wesley, 1994.

Ces deux références, bien qu'étant généralistes, font le choix de se focaliser sur la conception de bases de données.

Enfin, le lecteur intéressé par les aspects plus théoriques des bases de données consultera

Mark Levene and George Loizou, *A Guided Tour of Relational Databases and Beyond*, Springer, 1999.

Serge Abiteboul, Richard Hull et Victor Vianu, *Fondements des bases de données*, Vuibert, 2000.

Jeffrey David Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

David Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.