B

Les bases de l'interrogation de données en SQL

SQL (*Structured Query Language*) est un langage d'interrogation de bases de données. Cela signifie qu'il permet de consulter et modifier le contenu d'une base de données. En réalité, SQL permet aussi bien de consulter et modifier la structure d'une base de données (créer, modifier ou supprimer des tables et des relations) que de consulter et modifier les données (sélectionner, insérer, modifier ou supprimer des lignes à l'intérieur des tables). Il permet aussi de gérer les transactions et les droits d'accès (quand différents utilisateurs ont des droits d'accès différents). SQL est donc à la fois un langage de manipulation de données, de définition de données, de contrôle des transactions, et de contrôle des données. Dans ce chapitre toutefois et pour des raisons d'espace, nous nous bornerons aux aspects du langage relatifs à la manipulation des données et pas de la structure, des transactions ou des droits d'accès. Il existe quantité de livres beaucoup plus complets sur le langage SQL qui traitent ces différents aspects du langage en détail.

Les instructions en SQL s'appellent des « requêtes », ce qui est somme toute assez logique pour un langage qui prétend interroger des données. Mais à vrai dire, SQL est un langage assez sommaire en comparaison de langages de programmation. Il ne permet pas de faire des boucles ou des branchements conditionnels. En fait, il ne permet quasiment aucune forme d'algorithmique. C'est un langage purement déclaratif, c'est-à-dire permettant de décrire le résultat recherché mais pas de préciser la manière de l'obtenir. La traduction d'une requête en une séquence optimale d'algorithmes pour effectuer la requête est laissée au soin du SGBD.

Chaque requête commence par un mot-clé qui détermine le type d'action à réaliser. Dans le cadre de la manipulation des données, ces mots-clés sont au nombre de quatre : `SELECT` (pour rechercher des données), `INSERT INTO` (pour insérer de nouvelles lignes dans une table), `UPDATE` (pour modifier les

valeurs des attributs d'enregistrements existants), et `DELETE` (pour supprimer des enregistrements). Le mot-clé `FROM` permet d'indiquer la ou les tables sur lesquelles l'action doit porter. Voyons maintenant dans les grandes lignes comment s'utilisent ces différentes fonctions.

REQUÊTES SELECT

La structure de base des requêtes `SELECT` est la suivante :

```
SELECT [liste des colonnes à sélectionner séparées par des ,]
FROM [nom de la table qui contient les données recherchées]
```

Reprenons notre base de données des étudiants, cours et inscriptions, pour les besoins de la cause et découvrons le SQL à travers quelques exemples. Avant cela, précisons que si chaque SGBD a sa propre manière d'optimiser la traduction des requêtes en algorithmes, chaque système a aussi ses petites spécificités au niveau de la syntaxe du SQL. Il serait vain de vouloir établir ici les correspondances entre tous les SGBD possibles, et cela sortirait du cadre de ce livre. Nous avons donc dû choisir un logiciel de SGBD et baser tous nos exemples sur celui-ci. Notre choix s'est porté sur Microsoft Access, mais tout ce qui suit peut être facilement adapté à n'importe quel autre SGBD.

Sélectionner les colonnes souhaitées

Commençons à présent simplement par afficher la liste des étudiants (c'est-à-dire le contenu de la table Etudiant). La requête et son résultat apparaissent ci-dessous.

```
SELECT matricule, nom, prenom, [date de naissance], adresseFROM
Etudiant
```

matricule	nom	prenom	date de naissance	adresse
298489	Marcel	Pascale	21/05/1994	Rue des Hirondelles 2
300487	Berzok	Nicolas	14/09/1994	Rue de Bréderode 14
310452	Pins	Robert	1/08/1995	Rue des Lilas 13
325637	David	Emilie	19/04/1995	Place de l'Etoile 4
330777	Sineri	Hugues	13/11/1994	Square des Tilleuls 12
380654	Piesnet	Marie-Paule	8/06/1995	Avenue Pasteur 35

Par souci d'économie de code, quand on souhaite extraire toutes les colonnes d'une table, il est plus commode de remplacer la liste complète des champs par un astérisque : `SELECT * FROM Etudiant` donne le même résultat. Bien sûr, rien ne nous oblige à extraire toutes les colonnes. On pourrait tout aussi bien se contenter des matricules, noms et prénoms.

```
SELECT matricule, nom, prenom FROM Etudiant
```

matricule	nom	prenom
298489	Marcel	Pascale
300487	Berzok	Nicolas
310452	Pins	Robert
325637	David	Emilie
330777	Sineri	Hugues
380654	Piesnet	Marie-Paule

On peut aussi ajouter des colonnes contenant le résultat d'un calcul effectué au départ des autres colonnes. Pour ce faire, nous avons besoin de quelques opérateurs arithmétiques et de quelques fonctions de base. Les opérateurs arithmétiques sont classiques : +, -, *, /, % (modulo), & (concaténation de chaînes). Nous pouvons ainsi ajouter une colonne comprenant le nom complet (prénom et nom).

```
SELECT matricule, nom, prenom, prenom & " " & nom FROM Etudiant
```

matricule	nom	prenom	Expr1
298489	Marcel	Pascale	Pascale Marcel
300487	Berzok	Nicolas	Nicolas Berzok
310452	Pins	Robert	Robert Pins
325637	David	Emilie	Emilie David
330777	Sineri	Hugues	Hugues Sineri
380654	Piesnet	Marie-Paule	Marie-Paule Piesnet

Ou bien une colonne calculant l'âge de chaque étudiant. Pour ce faire, il nous faut calculer la différence (en jours) entre la date courante et la date de naissance. Avec Access, on obtient la date courante du système avec la fonction `Date()`. Il suffit alors de calculer la différence et de diviser le résultat par 365. Pour bien faire, on peut ensuite arrondir le résultat à l'entier le plus proche au moyen de la fonction `Round(x,p)` où x représente le nombre à arrondir, et p le nombre de décimales souhaité.

```
SELECT matricule, nom, prenom, Round((Date()-[date de naissance])/
365,0) FROM Etudiant
```

matricule	nom	prenom	Expr1
298489	Marcel	Pascale	20
300487	Berzok	Nicolas	20
310452	Pins	Robert	19

325637	David	Emilie	19
330777	Sineri	Hugues	19
380654	Piesnet	Marie-Paule	19

On pourrait encore extraire les initiales de chaque étudiant et les afficher dans une nouvelle colonne. Cela supposera d'avoir recours à une fonction pour sélectionner une partie seulement d'un champ textuel (ici le premier caractère à gauche, ce qui s'obtient avec la fonction `Left(x,n)` où *x* est la chaîne de caractère de départ, et *n* le nombre de caractères que l'on souhaite conserver). On peut aussi ajouter une colonne contenant une valeur arbitraire, par exemple l'année académique en cours. À ce stade, cela ne présente pas un intérêt évident, mais vous comprendrez plus tard pourquoi cela peut être utile.

```
SELECT matricule, nom, prenom, Left(prenom,1)&Left(nom,1), "2013-
2014" FROM Etudiant
```

matricule	nom	prenom	Expr1	Expr2
298489	Marcel	Pascale	PM	2013-2014
300487	Berzok	Nicolas	NB	2013-2014
310452	Pins	Robert	RP	2013-2014
325637	David	Emilie	ED	2013-2014
330777	Sineri	Hugues	HS	2013-2014
380654	Piesnet	Marie-Paule	MP	2013-2014

Chaque fois que nous ajoutons une colonne à celles existantes, le SGBD donne à ce nouveau champ un nom arbitraire (ici `Expr1`). On peut tout aussi bien déterminer ce nom soi-même, voire même modifier l'intitulé des colonnes existantes tel qu'il apparaît dans les résultats (cela ne modifie en rien le nom réel des colonnes concernées, d'ailleurs une requête `SELECT` ne modifie jamais rien, elle ne fait que produire une « lecture » des données). On détermine l'intitulé d'une colonne avec le mot-clé `AS` (une spécificité d'Access, la plupart des autres SGBD n'ont pas besoin de ce mot-clé pour comprendre que ce qui suit le champ est un alias).

```
SELECT matricule, Left(prenom,1)&Left(nom,1) AS Initiales,
Round((Date()-[date de naissance])/365,0) AS Age, "2013-2014"
AS Année FROM Etudiant
```

matricule	Initiales	Age	Année
298489	PM	20	2013-2014
300487	NB	20	2013-2014
310452	RP	19	2013-2014

325637	ED	19	2013-2014
330777	HS	19	2013-2014
380654	MP	19	2013-2014

Sélectionner les lignes recherchées

Nous avons envisagé plusieurs manières de déterminer les colonnes à afficher : sélection de colonnes existantes et création « à la volée » (c'est-à-dire pour les seuls besoins de la requête au moment de son exécution) de colonnes supplémentaires. Voyons maintenant comment agir sur la sélection des lignes.

Le premier outil à notre disposition est celui qui consiste à exclure de la présentation des résultats les lignes exactement identiques, c'est-à-dire redondantes. En principe, dans nos tables, aucune ligne ne devrait être parfaitement identique, puisque chaque table contient au minimum une clé primaire qui rend chaque ligne unique. Mais quand une requête ne sélectionne pas toutes les colonnes, et en particulier pas la colonne qui sert de clé primaire, il se peut que l'exécution de la sélection fasse apparaître des lignes identiques. Le comportement par défaut des SGBD par rapport à cette question diffère de l'un à l'autre. Certains excluent par défaut les lignes redondantes, les autres les conservent sauf contrordre. SQL résout le problème en permettant les deux options. Juste derrière le mot-clé `SELECT`, il suffit d'insérer le mot-clé `ALL` pour expliciter qu'on souhaite voir toutes les lignes, même les doublons, ou `DISTINCT` pour exiger que les lignes redondantes soient supprimées des résultats. Voyez la différence entre les résultats des deux requêtes ci-dessous.

```
SELECT ALL type FROM Discipline
```

type
Humaines
Sociales
Sociales
Exactes
Humaines
Exactes

```
SELECT DISTINCT type
FROM Discipline
```

type
Exactes
Humaines
Sociales

Intéressant, n'est-ce pas ? Mais on ne peut pas encore dire ici qu'on sélectionne les lignes. Pour déterminer spécifiquement quelles lignes on souhaite sélectionner, il nous faut un moyen de préciser dans la requête un ou plusieurs critères de recherche. C'est justement le rôle du mot-clé `WHERE` que l'on

ajoute à la fin d'une requête et que l'on fait suivre par des critères de recherche (qui porteront logiquement sur la valeur des colonnes extraites ou calculées). Les critères eux-mêmes sont définis en utilisant les opérateurs de comparaison (=, <, >, <=, >=, <> (différent)) et sont combinés en utilisant les opérateurs logiques (`AND`, `OR` et `NOT`). On peut utiliser les parenthèses pour lever toute ambiguïté sur l'ordre dans lequel des conditions multiples doivent être combinées.

Vous êtes prêt ? Essayons ! Commençons par afficher les étudiants, en sélectionnant seulement ceux dont le matricule est supérieur à 310000.

```
SELECT * FROM Etudiant WHERE Matricule >"310000"
```

matricule	nom	prenom	date de naissance	adresse
310452	Pins	Robert	1/08/1995	Rue des Lilas 13
325637	David	Emilie	19/04/1995	Place de l'Etoile 4
330777	Sineri	Hugues	13/11/1994	Square des Tilleuls 12
380654	Piesnet	Marie-Paule	8/06/1995	Avenue Pasteur 35

On voit que seuls 4 des 6 étudiants répondent à ce critère. On peut encore restreindre la sélection en ne conservant que ceux qui sont nés en 1995. Nous devons ajouter une condition supplémentaire dans la clause `WHERE` qui portera sur une valeur calculée : la fonction `Year(x)` nous sera utile pour obtenir l'année de chaque date de naissance.

```
SELECT * FROM Etudiant WHERE Matricule >"310000" AND Year([Date de
naissance])=1995
```

matricule	nom	prenom	date de naissance	adresse
310452	Pins	Robert	1/08/1995	Rue des Lilas 13
380654	Piesnet	Marie-Paule	8/06/1995	Avenue Pasteur 35
325637	David	Emilie	19/04/1995	Place de l'Etoile 4

Quand on souhaite restreindre les résultats aux enregistrements dont l'une des colonnes en particulier est nulle (ou au contraire exclure celles-là des résultats), la chose peut se corser un peu. Un champ numérique vide par exemple ne veut pas dire qu'il contient la valeur 0 (on écrirait alors simplement `WHERE champ=0`), et un champ textuel vide ne veut pas dire qu'il ne comprend que des espaces. Non, on vous parle de la possibilité qu'un champ ne comprenne rien. Les SGBD ont un mot pour désigner l'absence de valeur dans un champ, elles disent simplement que le champ en question est nul (`IS NULL`). Pas un chiffre, pas une lettre, c'est vrai que c'est assez nul. Si on cherche les étudiants dont la date de naissance est manquante par exemple, la clause conditionnelle à appliquer devient `WHERE [Date de naissance] IS NULL`. Et pour, au contraire, exclure ces lignes-là : `WHERE [Date de`

`naissance] IS NOT NULL`. Le SQL, ce n'est pas du Shakespeare, c'est pire, mais c'est tout de même assez proche de l'anglais.

Avec les chaînes de caractères, on peut se permettre des critères de sélection assez sophistiqués. On peut bien sûr simplement sélectionner sur base de la valeur complète d'une chaîne (par exemple avec une clause de type `WHERE prenom="Robert"`). Mais on peut aussi sélectionner sur base de la présence d'un certain motif à l'intérieur d'une chaîne. On utilise alors un mot-clé (on l'appelle en fait un « prédicat ») spécifique : `LIKE`. Devant ce mot-clé, on indique la chaîne de caractères sur laquelle portera la sélection, et derrière, le motif recherché, dans lequel on peut utiliser des caractères génériques (l'équivalent des pièces blanches au Scrabble) : « * » représente n'importe quelle séquence de caractères ou espaces de n'importe quelle longueur, « ? » représente n'importe quel caractère unique, et « # » représente un chiffre. Pour rechercher par exemple tous les étudiants qui habitent dans une rue (plutôt que dans une avenue ou autre), on précisera `WHERE adresse LIKE "Rue*"`. Access recherchera alors tous les enregistrements dont le champ adresse commence par « Rue ». On peut encore ruser davantage, comme rechercher tous les noms qui contiennent au moins un « i » : `WHERE nom LIKE "*i*"`. Ou tous ceux qui ont un « i » en seconde position : `WHERE nom LIKE "?i*"`. Voici ce que donnerait cette dernière condition :

```
SELECT * FROM Etudiant WHERE nom LIKE "?i*"
```

matricule	nom	prenom	date de naissance	adresse
310452	Pins	Robert	1/08/1995	Rue des Lilas 13
380654	Piesnet	Marie-Paule	8/06/1995	Avenue Pasteur 35
330777	Sineri	Hugues	13/11/1994	Square des Tilleuls 12

On le voit, les variantes sont infinies, et les raffinements possibles encore nombreux, trop nombreux pour qu'on puisse en dresser l'inventaire dans cet aperçu rapide des requêtes. Accordons-nous un dernier petit plaisir sélectif pour la route en combinant joyeusement toutes ces conditions : cherchons tous les étudiants (ils ne doivent pas être nombreux) dont la deuxième lettre du nom est un « i » ou la deuxième lettre du prénom est un « a », et qui habitent dans une rue. Voyez la manière dont les conditions sont organisées avec les parenthèses.

```
SELECT * FROM Etudiant
WHERE (nom LIKE "?i*" OR prenom LIKE "?a*") AND adresse LIKE "Rue*"
```

matricule	nom	prenom	date de naissance	adresse
310452	Pins	Robert	1/08/1995	Rue des Lilas 13
298489	Marcel	Pascale	21/05/1994	Rue des Hirondelles 2

Trier les résultats

Nous voici donc capables de sélectionner les lignes qui nous intéressent sur base de critères de sélection parfois assez subtils. Il nous manque à présent la possibilité de déterminer l'ordre dans lequel les lignes apparaissent dans les résultats. Nous pourrions par exemple souhaiter afficher la liste des cours classée par ordre alphabétique, ou bien par nombre de crédits décroissants (question d'afficher les cours les plus importants en premier). Ordonner des informations, voilà qui devrait être dans les cordes d'un ordinateur — ou plutôt les bus. Et avec quelques rudiments d'anglais, on devine aisément la clause dont nous avons besoin : `ORDER BY`, et ses subtils appendices : `ASC` (pour ascendant, c'est-à-dire le comportement par défaut) ou `DESC` (pour descendant). On obtient alors facilement :

```
SELECT * FROM Cours ORDER BY Titre
```

mnemonique	titre	titulaire	ects	Discipline (FK)
GESTS203	Analyse des états financiers	Faska Comte	3	3
LANGS201	Anglais I	Ian Smith	4	5
DROIS201	Droit commercial	Françoise Juge	3	1
INFOS202	Introduction à l'informatique	Nicolas Code	5	4
MATHS201	Mathématique II	Marie Cosinus	5	6
STATS202	Probabilités et inférence statistique	Lise Peutet	10	6
ECONS201	Théorie macroéconomique I	Robert Ycroit	6	2

```
SELECT * FROM Cours ORDER BY ects DESC
```

mnemonique	titre	titulaire	ects	Discipline (FK)
STATS202	Probabilités et inférence statistique	Lise Peutet	10	6
ECONS201	Théorie macroéconomique I	Robert Ycroit	6	2
MATHS201	Mathématique II	Marie Cosinus	5	6
INFOS202	Introduction à l'informatique	Nicolas Code	5	4
LANGS201	Anglais I	Ian Smith	4	5
GESTS203	Analyse des états financiers	Faska Comte	3	3
DROIS201	Droit commercial	Françoise Juge	3	1

Bien sûr, rien n'empêche de préciser plusieurs critères de tri en les séparant par des virgules et pourquoi pas, tant qu'à faire, d'ajouter également des critères de sélection (la clause `WHERE` doit alors précéder la clause `ORDER BY`) :

```
SELECT * FROM Cours WHERE ects>3 ORDER BY [Discipline (FK)]
ASC, ects DESC
```

mnemonique	titre	titulaire	ects	Discipline (FK)
ECONS201	Théorie macroéconomique I	Robert Ycroit	6	2
GESTS203	Analyse des états financiers	Faska Comte	3	3
INFOS202	Introduction à l'informatique	Nicolas Code	5	4
LANGS201	Anglais I	Ian Smith	4	5
STATS202	Probabilités et inférence statistique	Lise Peutet	10	6
MATHS201	Mathématique II	Marie Cosinus	5	6

Effectuer des regroupements et des traitements statistiques

Si SQL peut effectuer des opérations sur les colonnes (par exemple combiner deux champs pour en former un nouveau par concaténation ou par opération arithmétique) comme nous l'avons fait plus haut, le langage peut aussi réaliser des opérations sur les lignes, c'est-à-dire calculer des statistiques, comme par exemple calculer l'âge moyen des étudiants, ou la moyenne des notes obtenues à chaque cours. Tout ce dont nous avons besoin, c'est de fonctions d'agrégation qui puissent s'exécuter à travers les lignes. Pour rester simples, tous les SGBD offrent au minimum les cinq fonctions d'agrégation suivantes :

- `AVG` : Calcule la moyenne des valeurs d'une colonne
- `MAX` : Calcule la valeur maximale d'une colonne
- `MIN` : Calcule la valeur minimale d'une colonne
- `SUM` : Calcule la somme des valeurs d'une colonne
- `COUNT` : Compte le nombre de valeurs observées dans une colonne (ce qui revient à compter le nombre de lignes dans la table)

Calculons par exemple les nombres moyen, minimum et maximum de crédits ECTS de nos cours, et comptons également le nombre de cours qui ont été pris en compte dans ces statistiques.

```
SELECT AVG(ects) AS moyenne, MIN(ects) AS minimum, MAX(ects) AS
maximum, COUNT(ects) AS [nombre cours], SUM(1) AS [nombre calculé
autrement], SUM(ects) AS total FROM cours
```

moyenne	minimum	maximum	nombre cours	nombre calculé autrement	total
5,14285714285714	3	10	7	7	36

Un détail mérite peut-être un mot d'explication. C'est l'utilisation que nous avons faite de la fonction `SUM`. Nous avons utilisé cette fonction deux fois.

D'une part en l'appliquant au champ ECTS (`SUM(ects)`), ce qui revient à additionner le nombre de crédits de chaque cours, pour arriver au total de 36. Mais d'autre part aussi, en appliquant la fonction SUM à une valeur arbitraire (ici la valeur 1). Cela revient à ajouter une colonne dont la valeur serait 1 pour chaque ligne, et ensuite faire la somme de ces valeurs. Le résultat est identique à celui de la fonction `COUNT`, cela revient à compter le nombre de lignes.

Dans la requête qui précède, nous nous sommes contentés de calculer des statistiques pour l'ensemble de la table. Mais il sera bien souvent précieux de pouvoir calculer des statistiques sur des groupes, c'est-à-dire effectuer ce qu'on pourrait appeler des sous-totaux, ce qui suppose bien entendu de pouvoir déterminer sur quels critères on souhaite se baser pour regrouper les lignes. C'est précisément le rôle de la clause `GROUP BY`, au nom pour le moins clair.

Pour commencer, on pourrait par exemple calculer le nombre d'étudiants nés chaque année. La fonction d'agrégation sera un comptage (`COUNT(Matricule)` par exemple), et le regroupement portera sur l'année de naissance (`GROUP BY Year([Date de naissance])`).

```
SELECT YEAR([Date de naissance]) AS Année, COUNT(Matricule) AS Nombre
FROM Etudiant GROUP BY YEAR([Date de naissance])
```

Année	Nombre
1994	3
1995	3

Joindre les deux bouts

Jusqu'à présent, nous avons peine à y croire nous-mêmes, mais absolument toutes les requêtes que nous avons écrites se contentaient d'aller chercher de l'information dans une seule table à la fois. On en aurait presqu'oublié que l'objet du chapitre est la gestion des bases de données prétendument relationnelles. Venons-en donc à ces fameuses relations et voyons comment les matérialiser dans nos requêtes.

Fondamentalement, SQL est un langage assez accommodant. Si vous lui demandez d'aller chercher toutes les lignes qu'il trouve dans deux tables différentes, il ne sera pas contre. Nous pourrions par exemple lui demander d'aller chercher tous les enregistrements des tables Etudiant et Cours et d'en afficher toutes les colonnes. Il suffirait d'écrire ceci : `SELECT * FROM Cours, Etudiant`. SQL n'y verrait absolument aucun inconvénient. Mais notre éditeur ne serait pas du même avis. Parce que le tableau qui en résulterait contiendrait 10 colonnes et 42 lignes. Le nombre de colonnes ne devrait pas tellement vous surprendre : il y a 5 colonnes dans chacune des deux tables, il

est donc assez logique qu'en les juxtaposant, on se retrouve avec une table de 10 colonnes. Mais le nombre de 42 lignes est sans doute plus intriguant. Sauf si vous vous souvenez que notre base de données contient 6 étudiants et 7 cours, et que le produit de 6 par 7 donne… 42 !

Un mot d'explication s'impose. Quand on demande à SQL d'extraire de l'information de deux tables simultanément, il réalise entre elles ce qu'on appelle une jointure, c'est-à-dire le produit cartésien des deux tables. Il combine chaque enregistrement d'une table avec chacun des enregistrements de l'autre table. Si nous avions 200 étudiants et 20 cours, nous nous serions retrouvés avec 4000 lignes sur les bras. Il se trouve que le résultat de ce produit cartésien pourrait nous servir, si nous souhaitions justement insérer dans la table « Inscription » l'inscription de chaque étudiant à chaque cours. Nous pourrions réaliser cette belle opération en une seule fois, quel que soit le nombre d'étudiants et de cours. Mais ne brûlons pas d'étapes, nous y reviendrons.

Prenons plutôt un autre exemple. Souvenez-vous que notre base de données contient également une table Véhicule qui contient la plaque d'immatriculation et quelques autres informations sur la voiture de chaque étudiant. Il y a actuellement 6 enregistrements dans cette table que nous affichons ci-dessous.

```
SELECT * FROM Véhicule
```

Plaque	Matricule (FK)	Marque	Modèle	Type d'accès
ABC-123	298489	Renault	Clio	Complet
ABD-932	380654	Toyota	Yaris	Intérieur
BKJ-692	325637	VW	Golf	Complet
KJZ-098	330777	Renault	Scenic	Complet
YUS-298	310452	Ford	Ka	Intérieur
ZHD-564	300487	Peugeot	307	Extérieur

Ici encore, si nous nous contentions de sélectionner toutes les colonnes et toutes les lignes des tables Etudiant et Vehicule, nous hériterions d'une table beaucoup trop volumineuse. Avec 6 étudiants et 6 véhicules, le résultat de la requête comprendrait 36 lignes. Vous ne nous croyez pas ? Exécutons la requête suivante (pour éviter de nous attirer les foudres de notre éditeur, notez notre malicieux usage du mot-clé `TOP` pour limiter l'affichage des résultats aux 15 premières lignes, mais vous pourrez facilement imaginer les 21 autres) :

```
SELECT TOP 15 Matricule, Nom, Prenom, [Matricule (FK)], Plaque,
Marque FROM Etudiant, Véhicule ORDER BY Nom, Plaque
```

Matricule	Nom	Prenom	Matricule (FK)	Plaque	Marque
300487	Berzok	Nicolas	298489	ABC-123	Renault
300487	Berzok	Nicolas	380654	ABD-932	Toyota
300487	Berzok	Nicolas	325637	BKJ-692	VW
300487	Berzok	Nicolas	330777	KJZ-098	Renault
300487	Berzok	Nicolas	310452	YUS-298	Ford
300487	Berzok	Nicolas	300487	ZHD-564	Peugeot
325637	David	Emilie	298489	ABC-123	Renault
325637	David	Emilie	380654	ABD-932	Toyota
325637	David	Emilie	325637	BKJ-692	VW
325637	David	Emilie	330777	KJZ-098	Renault
325637	David	Emilie	310452	YUS-298	Ford
325637	David	Emilie	300487	ZHD-564	Peugeot
298489	Marcel	Pascale	298489	ABC-123	Renault
298489	Marcel	Pascale	380654	ABD-932	Toyota
298489	Marcel	Pascale	325637	BKJ-692	VW

Comme vous le voyez, malgré la présence de la clé étrangère (Matricule (FK)) dans la table Véhicule, notre brave SQL n'a pas pu réfréner son amour scalaire pour les produits du même nom. Autant vous le dire tout de suite, il n'y a pas moyen d'empêcher SQL de réaliser une jonction entre deux tables autrement que par un produit scalaire. C'est plus fort que lui. En revanche, vous connaissez déjà la parade. Ne venez-vous pas de découvrir de quelle manière on peut filtrer les résultats d'une requête pour les limiter aux seules lignes qui nous intéressent ? Mais oui, avec la clause `WHERE` bien entendu. Mais quel serait ici le critère de sélection judicieux à utiliser ? Bon sang, mais c'est bien sûr ! Le critère de sélection devrait être exactement celui qui matérialise la relation entre les deux tables, c'est-à-dire l'égalité entre la clé primaire et la clé étrangère. Essayons donc ceci :

```
SELECT Matricule, Nom, Prenom, [Matricule (FK)], Plaque, Marque FROM
Etudiant, Véhicule WHERE Etudiant.Matricule = Véhicule.[Matricule
(FK)] ORDER BY Nom, Plaque
```

Matricule	Nom	Prenom	Matricule (FK)	Plaque	Marque
300487	Berzok	Nicolas	300487	ZHD-564	Peugeot
325637	David	Emilie	325637	BKJ-692	VW

298489	Marcel	Pascale	298489	ABC-123	Renault
380654	Piesnet	Marie-Paule	380654	ABD-932	Toyota
310452	Pins	Robert	310452	YUS-298	Ford
330777	Sineri	Hugues	330777	KJZ-098	Renault

Mieux, n'est-ce pas ? Pourtant nous n'avons pas changé la manière dont SQL traite une requête combinant deux tables, il s'efforce toujours de combiner tous les enregistrements de l'une avec tous les enregistrements de l'autre. Mais cette fois nous avons pris soin de lui indiquer que toutes ces combinaisons ne nous intéressent pas. Seules nous intéressent celles dans lesquelles les clés primaire et étrangère coïncident. Et c'est exactement cela une relation entre deux tables. Ce sont deux champs qui sont mis en adéquation pour relier leurs enregistrements. Il est à ce stade aussi important de préciser que la représentation graphique des relations présentées plus haut dans le chapitre n'a de véritable concrétisation que par l'entremise de ces requêtes SQL liant les clés primaires à celles étrangères.

Dans la pratique et par souci de clarté, la plupart des SGBD permettent de réaliser des jointures SQL en suivant une syntaxe qui n'est pas plus courte mais a le mérite de rendre les choses plus explicites. On utilise pour cela les mots-clés `INNER JOIN` (encore une spécificité d'Access, la plupart des autres SGBD utilisent plutôt l'expression `JOIN BY` pour dire à peu près la même chose), dont le sens est à nouveau assez explicite. Les mots `INNER JOIN` relient les deux tables, la clause `ON` qui suit précise les critères de correspondance, c'est-à-dire logiquement l'égalité entre les clés primaire et étrangère. La requête ci-dessus aurait donc pu s'écrire :

```
SELECT Matricule, Nom, Prenom, [Matricule (FK)], Plaque, Marque
FROM Etudiant INNER JOIN Véhicule ON Etudiant.Matricule =
Véhicule.[Matricule (FK)]
ORDER BY Nom, Plaque.
```

Vous aurez peut-être remarqué une particularité nouvelle apparue dans nos requêtes joignant différentes tables. Vous ne voyez pas ? Regardez le critère de sélection de la requête précédente (ou la clause `ON` ci-dessus). Vous y êtes, nous avons fait précéder le nom du champ par le nom de la table dont il est issu puis d'un point : `Etudiant.Matricule` au lieu de `Matricule`. Cette notation permet de lever les ambiguïtés éventuelles. Aucune SGBD ne permettrait que deux colonnes portent le même nom dans une même table, mais aucune n'interdirait que deux tables différentes contiennent une colonne au nom identique. Tant que nos requêtes ne devaient se préoccuper que d'une seule table, le problème ne se posait pas. Mais maintenant que nos requêtes peuvent interroger plusieurs tables simultanément, le risque d'homonymie entre colonnes n'est plus exclu. On évite toute ambiguïté en accolant le nom de sa table au nom de chaque champ qui nous intéresse. Les

SGBD n'imposent généralement pas cette notation sauf en cas d'ambiguïté, mais de nombreux développeurs ont pris l'habitude dans toutes leurs requêtes de toujours expliciter la table de chaque champ. Cela rallonge sensiblement les requêtes mais ne fait de tort à personne.

Des requêtes comme critères de sélection d'autres requêtes : les sous-requêtes

Quand nous nous sommes intéressés aux critères de sélection et à la clause `WHERE`, nous n'avons envisagé que des conditions portant sur des critères déterminés et connus d'avance : une certaine année de naissance, la présence d'un certain caractère dans le nom ou le prénom, etc. Mais comment faire quand on ne connaît pas d'avance la valeur du critère de sélection, ou plus exactement quand cette valeur doit être calculée au départ des données elles-mêmes ?

Imaginez par exemple que nous souhaitions connaître tous les cours qui ont un nombre de crédits ECTS inférieur à la moyenne de l'ensemble des cours. En effectuant quelques statistiques ci-dessus, nous avions découvert que la moyenne des crédits est de 5,14, nombre que nous pourrions retrouver en exécutant la requête suivante : `SELECT AVG(ects) FROM cours`. Nous pouvons alors lister les cours dont les crédits sont inférieurs à cette moyenne en la codant manuellement dans une nouvelle requête : `SELECT * FROM Cours WHERE ects<5.14`. Mais ce n'est pas très pratique, il serait beaucoup plus rapide, efficace et sûr d'intégrer directement la première requête dans le critère de sélection de la seconde. La requête qui sert de critère de sélection est alors appelée une « sous-requête ». Ce qui ne sous-entend pas, qu'elle est une requête de second rang ou de moindre importance, mais plus simplement qu'elle est subordonnée à la requête principale.

```
SELECT * FROM Cours WHERE ects<(SELECT AVG(ects) FROM cours)
```

mnemonique	Titre	titulaire	ects	Discipline (FK)
DROIS201	Droit commercial	Françoise Juge	3	1
GESTS203	Analyse des états financiers	Faska Comte	3	3
INFOS202	Introduction à l'informatique	Nicolas Code	5	4
LANGS201	Anglais I	Ian Smith	4	5
MATHS201	Mathématique II	Marie Cosinus	5	6

Rien n'empêche d'imbriquer une sous-sous-requête dans une sous-requête, et ainsi de suite comme pour les poupées russes. La seule chose qui importe réellement quand une sous-requête sert de critère de sélection, c'est qu'elle renvoie bien un résultat unique quand elle suit un opérateur de comparaison (on ne peut pas tester une égalité avec plusieurs valeurs différentes). Dans

l'exemple qui précède, il est essentiel que la sous-requête `SELECT AVG( ects) FROM cours` renvoie une et une seule valeur (donc une seule ligne et une seule colonne) puisqu'on demande à SQL de comparer le résultat de cette requête avec la valeur du champ « ects ».

Ce serait différent si on remplaçait l'opérateur de comparaison (ici « < ») par le mot-clé `IN`. Ce dernier permet d'indiquer qu'on souhaite conserver les lignes à condition que la valeur du champ qui précède le mot `IN` apparaisse dans l'unique colonne de la sous-requête qui suit. Vous nous suivez ? Cherchons pour être concrets les cours qui sont suivis par au moins un étudiant, c'est-à-dire les cours dont le mnémonique apparaît au moins une fois dans la table des inscriptions (actuellement nous n'avons que 2 étudiants inscrits à 2 cours, DROIS201 et ECONS201). C'est ce que réalise la requête suivante :

```
SELECT * FROM Cours
WHERE Mnemonique IN (SELECT DISTINCT [Cours (FK)] FROM Inscription)
```

mnemonique	titre	titulaire	ects	Discipline (FK)
DROIS201	Droit commercial	Françoise Juge	3	1
ECONS201	Théorie macroéconomique I	Robert Ycroit	6	2

Dans ce mode d'utilisation (avec le mot-clé `IN`), la sous-requête peut renvoyer plusieurs lignes, mais il est important qu'elle ne renvoie qu'une seule colonne.

Fusionner les résultats de plusieurs requêtes : la clause UNION

Il arrive que les résultats recherchés ne puissent être extraits en une seule fois, à travers une seule requête. Cela arrive par exemple quand les données sources se trouvent dans plusieurs tables qui ne sont pas reliées par des relations directes. Il peut alors être nécessaire de fusionner les résultats de différentes requêtes pour les faire apparaître en un seul ensemble de lignes résultantes. Une telle fusion s'opère au moyen d'une clause `UNION` au nom évocateur.

Imaginons par exemple que nous souhaitions produire une liste de toutes les personnes impliquées dans l'université : étudiants et enseignants. L'information requise se trouve actuellement dans deux tables distinctes : Etudiant et Cours. Extraire l'information qui nous intéresse de chacune des deux tables séparément n'est pas très compliqué en soi : `SELECT prenom, nom FROM Etudiant ORDER BY prenom, nom` et `SELECT titulaire FROM Cours ORDER BY titulaire`. Avant de songer à fusionner les résultats de ces deux requêtes, une difficulté saute aux yeux : le nom des personnes n'est pas stocké de la même manière dans les deux tables : le nom des étudiants

est scindé en deux champs de la table Etudiant (nom et prénom) alors que celui des enseignants apparaît entièrement dans le champ « titulaire » de la table Cours. En l'état, il est donc impossible de fusionner les résultats de ces deux requêtes.

On ne peut en effet réaliser une opération de réunion de deux requêtes que si ces deux requêtes génèrent les mêmes colonnes dans le même ordre. Mais nous pouvons nous en sortir en générant un champ « Personne » pour les étudiants qui aurait un format analogue au champ « titulaire » de la table Cours, et veiller à intituler cette colonne de la même manière entre les deux requêtes : `SELECT prenom & " " & nom AS Personne FROM Etudiant ORDER BY prenom & " " & nom` et `SELECT titulaire AS Personne FROM Cours ORDER BY titulaire`. Maintenant que nos deux requêtes génèrent des résultats au format identique (mêmes colonnes dans le même ordre), nous pouvons utiliser la clause `UNION` pour les fusionner, comme dans l'exemple qui suit :

```
SELECT prenom & " " & nom AS Personne FROM Etudiant ORDER BY prenom &
" " & nom
UNION
SELECT titulaire AS Personne FROM Cours
```

Personne
Emilie David
Faska Comte
Françoise Juge
Hugues Sineri
Lise Peutet
Marie Cosinus
Marie-Paule Piesnet
Nicolas Berzok
Nicolas Code
Pascale Marcel
Robert Pins
Robert Ycroit

Comme on peut le voir dans les résultats ci-dessus, les lignes générées par les deux requêtes sont fusionnées et le critère de tri (ici défini par la première des deux requêtes) est appliqué après l'opération de fusion sur l'ensemble des résultats.

Par défaut, une opération de réunion élimine les doublons éventuels (c'est-à-dire que si un même nom apparaît deux fois dans l'ensemble des

lignes après fusion, les lignes redondantes seront supprimées). Il est possible de modifier ce comportement en remplaçant la clause `UNION` par `UNION ALL`.

Requêtes « action »

Après toutes ces consultations de données, passons enfin à l'action, c'est-à-dire aux requêtes qui permettent non plus de lire les données mais bien de les modifier. Nous avons rencontré dans les premiers paragraphes consacrés à SQL les trois requêtes possibles pour ce faire : `INSERT INTO`, `UPDATE` et `DELETE`. Commençons par la dernière, non seulement parce qu'elle est amusante et génératrice de frissons, mais surtout parce que c'est de loin la plus simple.

Supprimer des données : la requête DELETE

À la différence des requêtes sélection (et des deux autres types de requêtes qui modifient les données), `DELETE` ne demande que très peu d'arguments ou de précision. Le nom de la table dont vous souhaitez supprimer les données suffit (notez d'emblée qu'une requête ne peut modifier des données que dans une seule table à la fois). `DELETE FROM Cours`, et le catalogue des cours disparaît. `DELETE FROM Etudiant`, et l'université peut fermer boutique faute d'étudiants.

Eh oui, c'est qu'à la différence d'un traitement de texte ou d'un tableur, il est généralement vain de chercher la fonction « Annuler » d'un système de gestion de base de données, et plus encore de chercher la fonction « Enregistrer ». Quand une base de données exécute une requête, elle enregistre les modifications apportées directement dans la table, sur le support de stockage secondaire. L'explication tient simplement dans la raison d'être et le mode de fonctionnement d'un système de base de données. Si vous vous souvenez du début du chapitre, vous vous souviendrez aussi que les SGBD existent pour gérer des fichiers en exploitant leur structure logique indépendamment du stockage physique. Et elles réalisent cela en accédant directement aux données là où elles se trouvent sur le disque dur en fonction des clés ou index. Admirable ! Mais cela implique qu'à la différence d'un programme classique comme un traitement de texte, une base de données ne charge pas l'intégralité d'un fichier dans la mémoire vive de l'ordinateur avant de commencer à travailler. Elle travaille directement sur le disque dur. Et c'est très bien ainsi, surtout quand les bases de données sont volumineuses. Devoir attendre leur chargement complet en mémoire vive avant de pouvoir commencer à travailler serait particulièrement irritant, et bien sou-

vent voué à l'échec tant la taille des bases de données a tendance à dépasser celle des mémoires vives.

Dès lors, si vous exécutez une requête `DELETE` sur une table, ne venez pas vous étonner de ne plus pouvoir retrouver le contenu de la table par la suite. On manipule une requête `DELETE` avec beaucoup de délicatesse, et encore plus d'hésitation.

Cela étant, pour brutale et irrémédiable que soit une requête `DELETE`, elle n'en est pas pour autant binaire. Elle est parfaitement capable de supprimer des lignes beaucoup plus sélectivement. Et pour lui préciser quelle ligne vous souhaitez supprimer, utilisez la même clause `WHERE` que celle que vous auriez utilisée pour sélectionner des données. D'ailleurs, ce n'est pas une mauvaise idée d'afficher avec une requête `SELECT` les lignes qui seraient effacées si vous remplaciez simplement `SELECT … FROM` par `DELETE FROM`.

Essayons par exemple de supprimer les cours de langue de la table des cours. La discipline « Langues » correspond à la clé primaire 5. Nous pouvons donc afficher les lignes qui seraient affectées par la requête `DELETE FROM Cours WHERE [Discipline (FK)]=5` en exécutant d'abord `SELECT * FROM Cours WHERE [Discipline (FK)]=5` :

```
SELECT * FROM Cours WHERE [Discipline (FK)]=5
```

mnemonique	titre	titulaire	ects	Discipline (FK)
LANGS201	Anglais I	Ian Smith	4	5

Exécutez maintenant la requête `DELETE`, et Access vous confirmera qu'il a supprimé 1 ligne dans la table Cours. Notez que par extrême amabilité, Access vous demande en général confirmation avant de modifier des données à travers une requête action, en vous précisant le nombre de lignes qui seraient affectées par la requête. N'attendez pas la même bienveillance de la part des autres SGBD. La plupart ne s'embarrassent pas de vos hésitations de dernière minute. C'est vous le patron, si vous lancez une requête, le SGBD s'exécute, et vous assumez.

On pourrait de même décider de supprimer le cours d'économie (ah non, pas celui d'informatique !) avec la requête `DELETE FROM Cours WHERE Mnemonique="ECONS201"`. Mais cette requête-ci risquerait fort de poser problème, parce qu'à la différence du cours d'anglais, le cours d'économie est déjà suivi par deux étudiants. En supprimant le cours correspondant, ces deux étudiants se retrouvaient inscrits à un cours qui n'existe plus.

Quand on crée la structure d'une base de données dans un SGBD, on explicite normalement les relations, en insistant pour que le système veille à l'intégrité référentielle, c'est-à-dire qu'il veille à ce que toutes les valeurs

contenues dans des clés étrangères apparaissent bien dans la clé primaire correspondante. Or ici, si on supprimait le cours d'économie, cette règle d'intégrité référentielle se trouverait violée. Le SGBD devrait donc refuser d'exécuter cette requête et nous informer du problème. Qui ne peut se résoudre que de deux manières : soit on renonce à supprimer le cours d'économie, soit il faut préalablement supprimer l'inscription correspondante des deux étudiants.

Qu'à cela tienne, exécutons donc `DELETE FROM Inscription`. Affichons la table des inscriptions et constatons le résultat : les quatre inscriptions ont été supprimées.

```
SELECT * FROM Inscription
```

ID Jointure	Etudiant (FK)	Cours (FK)	Note	Obligatoire

Insérer des données : la requête INSERT INTO

En étant logique, nous aurions dû entamer SQL avec la requête qui permet d'alimenter les tables de la base de données, c'est-à-dire de créer des enregistrements. Une seule requête permet de créer des enregistrements, c'est la requête `INSERT INTO`. C'est aussi la seule requête qui n'utilise pas le mot-clé `FROM` pour désigner la table sur laquelle elle travaille mais plutôt le mot-clé `INTO`. Dans la mesure où ce terme est assez juste, ne lui en tenons pas rigueur. C'est enfin la seule requête qui n'a que faire de la clause `WHERE`.

Une requête d'insertion de données n'a besoin en réalité que de trois choses : le nom de la table où vous souhaitez insérer des enregistrements, la liste des colonnes de cette table pour lesquelles vous comptez fournir des valeurs (rien ne vous oblige à remplir toutes les colonnes, vous pouvez en laisser certaines vides, du moins si la structure de la table vous y autorise), et les valeurs en question.

```
INSERT INTO table(colonnes) VALUES(valeurs).
```

En principe, une requête `INSERT INTO` n'ajoute qu'un seul enregistrement à la fois (sauf utilisation d'une sous-requête, comme nous le verrons dans un instant). Ajoutons par exemple un cours d'éthique des affaires de quatre crédits à notre programme :

```
INSERT INTO Cours(Mnemonique, Titre, ECTS, [Discipline (FK)])
VALUES ("ETHIS201","Ethique des affaires",4,1)
```

Affichons ensuite le contenu de la table des cours pour en vérifier la bonne exécution :

```
SELECT * FROM Cours
```

mnemonique	titre	titulaire	ects	Discipline (FK)
DROIS201	Droit commercial	Françoise Juge	3	1
ECONS201	Théorie macroéconomique I	Robert Ycroit	6	2
ETHIS201	Ethique des affaires		4	1
GESTS203	Analyse des états financiers	Faska Comte	3	3
INFOS202	Introduction à l'informatique	Nicolas Code	5	4
MATHS201	Mathématique II	Marie Cosinus	5	6
STATS202	Probabilités et inférence statistique	Lise Peutet	10	6

Évidemment, ajouter des enregistrements un à un peut s'avérer fastidieux. `INSERT INTO` permet donc de remplacer la clause `VALUES` suivie de valeurs individuelles par une sous-requête `SELECT` qui extrait ou produit en masse les données à insérer. Il est temps maintenant d'alimenter en une fois la table des inscriptions en inscrivant nos 6 étudiants aux 7 cours (souvenez-vous qu'entretemps le cours d'anglais a été remplacé par un cours d'éthique des affaires). Exploitons pour cela cette sympathique habitude du langage SQL qui consiste à considérer une opération de jointure dans une requête comme un produit scalaire entre deux tables. Ainsi par exemple la requête `SELECT * FROM Cours, Etudiant` avait généré un tableau résultant comprenant 10 colonnes et 42 lignes. Nous n'avons pas besoin des 10 colonnes, mais les 42 lignes constituent exactement la base de ce que nous voulons insérer dans la table Inscription. Laissons à Access le soin de générer les clés primaires des 42 enregistrements nouveaux, et contentons-nous d'insérer les deux clés étrangères. Cela donne la requête suivante :

```
INSERT INTO Inscription([Cours (FK)],[Etudiant (FK)])
    SELECT Cours.Mnemonique, Etudiant.Matricule
    FROM Cours, Etudiant
    ORDER BY Cours.Mnemonique, Etudiant.Matricule
```

Affichons maintenant le contenu de la table des inscriptions (enfin disons les douze premières lignes) et voyons le résultat :

```
SELECT TOP 12 * FROM Inscription
```

ID Jointure	Etudiant (FK)	Cours (FK)	Note	Obligatoire
1	298489	DROIS201		False
2	300487	DROIS201		False
3	310452	DROIS201		False
4	325637	DROIS201		False
5	330777	DROIS201		False
6	380654	DROIS201		False

7	298489	ECONS201		False
8	300487	ECONS201		False
9	310452	ECONS201		False
10	325637	ECONS201		False
11	330777	ECONS201		False
12	380654	ECONS201		False

Efficace, non ?

Modifier des données : la requête UPDATE

Afin de boucler notre survol du langage SQL, il ne nous reste plus qu'à découvrir la modification des données avec la requête `UPDATE`. C'est une requête assez simple. Elle se contente de trois éléments : la table dont elle doit modifier les données, les modifications à apporter, et — si besoin — les conditions précisant quelles lignes elle doit modifier. Sa syntaxe a la forme suivante :

```
UPDATE table
SET champ1=valeur1, …, champK=valeurK
WHERE condition1…
```

Commençons par exemple par remplir la colonne des notes de la table des inscriptions qui reste désespérément vide. Nous pourrions par défaut donner une note de 10/20 à tous les étudiants :

```
UPDATE Inscription
SET Note=10
```

Vu l'absence de clause `WHERE`, cette requête aura pour effet de remplacer par la valeur 10 le contenu de toute la colonne « Note », pour tous les enregistrements de la table. Voilà au moins un mode de cotation dont on ne pourra pas contester le caractère équitable. Malheureusement, le caractère arbitraire non plus.

Mais imaginons que le professeur d'informatique s'oppose et souhaite remonter toutes les cotes de 3 points pour son cours. La requête devient alors :

```
UPDATE Inscription
SET [Note]=Note+3
WHERE [Cours (FK)]="INFOS202"
```

Affichons donc la moyenne de chaque cours pour nous assurer de la bonne exécution de nos ordres :

```
SELECT Cours.Mnemonique, Cours.Titre, AVG(Inscription.Note) AS
Moyenne,
COUNT(Inscription.Note) AS [Etudiants inscrits]
FROM Cours INNER JOIN Inscription ON
Cours.Mnemonique=Inscription.[Cours (FK)]
GROUP BY Cours.Mnemonique, Cours.Titre
ORDER BY AVG(Inscription.Note) DESC
```

Mnemonique	Titre	Moyenne	Etudiants inscrits
INFOS202	Introduction à l'informatique	13	6
STATS202	Probabilités et inférence statistique	10	6
MATHS201	Mathématique II	10	6
GESTS203	Analyse des états financiers	10	6
ETHIS201	Ethique des affaires	10	6
ECONS201	Théorie macroéconomique I	10	6
DROIS201	Droit commercial	10	6

Alors, merci qui? Merci `SELECT Titulaire FROM Cours WHERE Mnemonique="INFOS202"`.

Cachez ces requêtes que je ne saurais voir

Tous les systèmes d'entreprise et la plupart des sites Web dynamiques reposent sur l'utilisation d'une base de données par les programmes. La question de l'interfaçage entre le programme (ou son langage de programmation) et la base de données qu'on interroge en SQL se pose dès lors avec acuité sitôt qu'on entreprend le développement d'une application de ce type. En règle générale, cette intégration s'opère en utilisant dans le logiciel des interfaces prédéveloppées dans le kit de développement du langage utilisé (les « interfaces de programmation d'application » ou API). Ces interfaces fournissent la capacité d'établir une connexion à une base de données à l'intérieur d'un programme, d'envoyer des requêtes SQL à la base de données, et de récupérer les résultats renvoyés par la requête, généralement sous la forme d'une liste (ces fameux types « composites » décrits au chapitre précédent). Le langage de programmation peut alors reprendre la main pour manipuler les résultats contenus dans la liste, par exemple pour réaliser des calculs ou pour présenter à sa manière tout ou partie des résultats.

Conformément à l'évolution globale de l'informatique qui vise à en masquer la complexité et à la scinder en différentes couches aussi indépendantes que possible, les langages de programmation masquent de plus en plus la couche SQL et l'établissement des connexions aux bases de données en en déchargeant le développeur du logiciel. C'est particulièrement le cas des lan-

gages de programmation orientés objet contemporains (tel Python). Concrètement, le développeur définit des classes correspondant aux tables de sa base de données, dont les attributs ont la même structure que les colonnes de la table correspondante. Ces classes définissent le modèle de données du programme, c'est-à-dire l'équivalent orienté objet du schéma relationnel de la base de données (autrement dit la partie *M* ou « modèle » du paradigme MVC décrit au chapitre précédent).

Des API fournies par les langages modernes (comme Django par exemple, environnement de développement web très populaire ces temps-ci et basé sur le langage Python), se chargent alors d'assurer la synchronisation des objets du programme avec la base de données, permettant à l'utilisateur de créer, retrouver, modifier ou supprimer des objets, tandis que l'API se charge d'interagir avec la base de données pour assurer l'extraction des attributs des enregistrements existants, ou de répercuter dans les tables les opérations réalisées sur les objets par le programme. C'est donc l'API du langage qui génère à la volée les requêtes SQL (`INSERT`, `SELECT`, `UPDATE`, ou `DELETE`), si bien que le développeur d'un logiciel n'a plus à se préoccuper du code SQL, relégué à l'arrière-plan.